Inspiring Motion
Since 1988

Maestro Administrative And Motion Api

March 2016 (Ver. 2.001)

www.elmomc.com

Important Notice

This document is delivered subject to the following conditions and restrictions:

- This document is copyrighted and all rights are reserved by Elmo Motion Control Ltd. This product may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, by Elmo Motion Control Ltd.
- This document contains proprietary information belonging to Elmo Motion Control Ltd. Such information is supplied solely for assisting users of the Maestro Network Motion Controllers.
- The text and graphics included in this document are for the purpose of illustration and reference only. The specifications on which they are based are subject to change without notice.
- Elmo Motion Control and the Elmo Motion Control logo are trademarks of Elmo Motion Control Ltd.

Document no. MAN-MAESTRO-API Ver 2.001

Copyright © 2016

Elmo Motion Control Ltd.

All rights reserved.

Revision History

Version	Release Date	Changes/Remarks
2.000	Feb 2016	Preliminary Draft of new version type
2.001	Mar 2016	Addition of SCARA and Three Link Robots kinematic to section 5.10.9 Addition of section 5.13.5 Online Splines Addition of option to section to 7.3 Interpolation Options (curve Types) Corrections to section 4.15 Maestro Emergency Error IDs with resolutions

Chapter 1: Introduction 21

1.1 Maestro over (Go) Standard 21
1.2 What the API Does 21
1.3 Terminology 22
1.4 How to Use this Document 26

Chapter 2: Maestro Overview 30

2.1 Using EASII Application 34
2.2 Maestro Operation Modes 35
2.2.1 NC Motions 35
2.2.2 Distributed / Standard DS-402 (stand-alone) Drive 35
2.2.3 Maestro Axes and Node Definitions 35
2.2.4 Maestro to Servo Drive Interfaces 37

Chapter 3: Maestro Hardware Connections 38

3.1 Maestro USB Connection 38
3.1.1 Connection Procedure 38
3.1.2 ipaddr – GMAS IP Address 39
3.1.3 ipmask – GMAS IP Subnet Mask 39
3.1.4 defgateway – GMAS Default Gateway 40
3.2 Maestro Network Configuration 41
3.2.1 XP Windows Setup 41
3.2.2 Windows 7 Seup 47
3.3 Maestro Master –Slave Hot Plug 53

Chapter 4: Error Handling 55

4.1 Function Block Errors 55
4.2 Buffered Errors 56
4.3 Maestro Error IDs 57
4.4 Error correction error IDs 73
4.5 Continued Maestro Error IDs 75
4.6 Maestro PVT/ECAM Motion Error IDs 85
4.7 Continued Maestro Error IDs 86
4.8 NC Driver Warning IDs 107
4.9 NC Profiler Error IDs 108
4.10 NC Profiler Caution IDs 116
4.11 Maestro Caution IDs 118
4.12 Internal Library Error IDs 119
4.13 Internal Library Warning IDs 124
4.14 EtherNetIP Communication Error IDs 125
4.15 Maestro Emergency Error IDs Originating from the Servo Drive 127
4.16 Maestro Abort Error IDs Originating from the Servo Drive 134
4.17 ELMO Error Codes 136
4.18 Functional and Communication Error Handling 142
4.18.1 General 142
4.18.2 Detecting Connection Errors 145
4.18.3 Detecting EtherCAT Application Level (AL) Errors 146
4.18.4 DS-402 Status Word Error 146
4.18.5 Handling Errors – Resolution 147

4.18.6 Policy Functions ... 148
4.18.7 MMC_RegErrPolicy ... 149
4.18.8 MMC_GetErrPolicy ... 155
4.18.9 MMC_ResetSystem ... 160

Chapter 5: Motion and Administrative Function Blocks ... 163

5.1 Compliance and Portability ... 163
5.2 Function Block States ... 163
5.2.1 Single Axis ... 163
5.2.2 Group of Axes ... 165
5.2.3 Function Block Status Bit Masks ... 166
5.3 Axis, Group, Global, Parameters ... 167
5.3.1 Legend ... 168
5.3.2 Parameters Tables ... 169
5.4 Axis Status ... 176
5.4.1 Blended Behavior Mechanism ... 178
5.4.2 Special function block insertion mechanism ... 179
5.5 Function Block Execution Order ... 180
5.5.1 Classification ... 181
5.5.2 Flags ... 181
5.5.3 Examples ... 181
5.6 Administrative Function Block Handling (for both Single and Multiple Axis) ... 183
5.6.1 Source Value ... 183
5.6.2 Reference Value ... 183
5.6.3 Logic Operation ... 183
5.6.4 Function Usage Situations ... 184
5.6.5 Download Firmware and Errors ... 186
5.7 User Units ... 187
5.7.1 System Flow Forward ... 188
5.7.2 System Flow Backward ... 189
5.7.3 Practical Interface ... 190
5.8 Single Axis Motion Control ... 191
5.8.1 MMC_Halt ... 193
5.8.2 MMC_Home ... 197
5.8.3 MMC_HomeDS402 ... 213
5.8.4 MMC_MoveAbsolute ... 218
5.8.5 MMC_MoveAdditive ... 224
5.8.6 MMC_MoveRelative ... 230
5.8.7 MMC_MoveVelocity ... 236
5.8.8 MMC_MoveContinuous ... 242
5.8.9 MMC_MoveAbsoluteRepetitive ... 246
5.8.10 MMC_MoveRelativeRepetitive ... 251
5.8.11 MMC_MoveAdditiveRepetitive ... 256
5.8.12 MMC_Stop ... 261
5.9 Single Axis Administrative Control ... 266
5.9.1 Elmo SuperImposed Motion ... 267
5.9.2 Special Function ... 268
5.9.3 MMC_AxisLink ... 269
5.9.4 MMC_AxisUnLink ... 272

5.9.5 MMC_Dwell 274
5.9.6 MMC_GetFBDepth 277
5.9.7 MMC_GetTotalFbDepth 280
5.9.8 MMC_Power 283
5.9.9 MMC_PositionProfile 287
5.9.10 MMC_ReadActualPosition 290
5.9.11 MMC_ReadActualTorque 293
5.9.12 MMC_ReadActualVelocity 296
5.9.13 MMC_ReadAxisError 299
5.9.14 MMC_ReadBoolParameter 303
5.9.15 MMC_GlobalReadBoolParameter 306
5.9.16 MMC_ReadDigitalInput(s) 309
5.9.17 MMC_ReadDigitalOutputs 314
5.9.18 MMC_ReadDigitalOutputs32Bit 317
5.9.19 MMC_ReadParameter 320
5.9.20 MMC_GlobalReadParameter 323
5.9.21 MMC_ReadStatus 326
5.9.22 MMC_Reset 329
5.9.23 MMC_ResetAsync 332
5.9.24 MMC_SetOverride 334
5.9.25 MMC_SetPosition 338
5.9.26 MMC_TouchProbeEnable 341
5.9.27 MMC_TouchProbeDisable 344
5.9.28 MMC_WriteBoolParameter 347
5.9.29 MMC_GlobalWriteBoolParameter 350
5.9.30 MMC_WriteDigitalOutputs 354
5.9.31 MMC_WriteDigitalOutputs32Bit 357
5.9.32 MMC_WriteParameter 361
5.9.33 MMC_GlobalWriteParameter 364
5.10 Multiple Axes Motion Control - Introduction 367
5.10.1 Coordinate System and kinematic transformation 368
5.10.2 ACS - Axes Coordinate System 369
5.10.3 MCS - Machine Coordinate System 369
5.10.4 PCS - Product Coordinate System 374
5.10.5 General Example 381
5.10.6 Example of Set Kinematic 383
5.10.7 Example - ACS Motion 384
5.10.8 Example - MCS Motion 385
5.10.9 Special Robot Transformations 386
5.10.10 Error IDs 403
5.11 Multiple Axes Motion Control - Transition and Buffer Modes 405
5.11.1 Single Axis Buffer Modes 406
5.11.2 Multi-Axes Transitions 407
5.11.3 Matrix of available Transition Modes 408
5.11.4 Polynomial transition curve vs circle arc 413
5.11.5 The MMC_S_FACTOR Parameter 414
5.11.6 Transition Inputs 415
5.11.7 Buffer modes 416
5.11.8 Selecting Polynomial or Simple Transition 417

5.11.9 2D and 3D Transition Examples ... 418
5.12 Multiple Axes Motion Control - Circular Modes ... 422
5.12.1 Border mode ... 422
5.12.2 Center mode ... 422
5.12.3 Radius mode ... 423
5.12.4 Angle Mode ... 424
5.12.5 PathChoice Data verification in MoveCircular functions ... 424
5.12.6 Move Polynomial Function Block ... 425
5.13 Multiple Axes Motion Control - Splines ... 427
5.13.1 Spline Mode MC_SPLINE_MODE_FT = 1 ... 428
5.13.2 Spline mode MC_SPLINE_MODE_V_AC_DC = 2 ... 429
5.13.3 Spline mode MC_SPLINE_MODE_NP = 3 ... 431
5.13.4 Spline mode MC_SPLINE_MODE_CV = 7 ... 433
5.13.5 Online Splines ... 435
5.13.6 Implementing the Spline Motion ... 442
5.14 Multiple Axes Motion Control – Functions ... 443
5.14.1 Function Block Status Bit Masks ... 443
5.14.2 MMC_GroupStop ... 444
5.14.3 MMC_GroupHalt ... 448
5.14.4 MMC_MoveCircularAbsolute ... 452
5.14.5 MMC_MoveCircularAbsoluteCenter ... 460
5.14.6 MMC_MoveCircularAbsoluteBorder ... 466
5.14.7 MMC_MoveCircularAbsoluteRadius ... 472
5.14.8 MMC_MoveCircularAbsoluteAngle ... 479
5.14.9 MMC_MoveLinearAbsolute ... 485
5.14.10 MMC_MoveLinearRelative ... 493
5.14.11 MMC_MoveLinearAdditive ... 501
5.14.12 MMC_MoveLinearAdditiveEx ... 506
5.14.13 MMC_MoveLinearAbsoluteRepetitive ... 511
5.14.14 MMC_MoveLinearRelativeRepetitive ... 516
5.14.15 MMC_MovePolynomAbsolute ... 521
5.14.16 MMC_PathSelect ... 525
5.14.17 MMC_MovePath ... 529
5.14.18 MMC_PathUnselect ... 534
5.15 Multiple Axes Administrative Control ... 537
5.15.1 Coordinated System and Kinematic Trnsformation Definitions ... 538
5.15.2 MC_KIN_NODE_DEF ... 539
5.15.3 MC_KIN_REF_CARTESIAN ... 541
5.15.4 MC_KIN_REF_DELTA ... 542
5.15.5 MC_KIN_REF_SCARA ... 544
5.15.6 MC_KIN_REF_THREE_LINK ... 546
5.15.7 MC_KIN_REF UNION ... 548
5.15.8 NC_MCS_Info_Struct ... 549
5.15.9 NC_MCS_Kin_Ref_Struct ... 551
5.15.10 MMC_SetKinTransform ... 552
5.15.11 MMC_SetKinTransformEx ... 557
5.15.12 MMC_SetCartesianTransform ... 565
5.15.13 MMC_TrackConveyorBelt ... 569
5.15.14 MMC_TrackRotaryTable ... 573

5.15.15 MMC_SetKinTransformDelta 578
5.15.16 MMC_SetKinTransformCartesian 581
5.15.17 MMC_SetKinTransformScara 584
5.15.18 MMC_SetKinTransformThreeLink 587
5.15.19 MMC_AddAxisToGroup 590
5.15.20 MMC_GroupDisable 593
5.15.21 MMC_GroupEnable 596
5.15.22 MMC_GroupReadActualPosition 599
5.15.23 MMC_GroupReadActualVelocity 603
5.15.24 MMC_GroupReadError 606
5.15.25 MMC_GroupReadStatus 609
5.15.26 MMC_GroupReset 612
5.15.27 MMC_GroupSetOverride 615
5.15.28 MMC_GroupSetPosition 620
5.15.29 MMC_RemoveAxisFromGroup 624
5.15.30 MMC_GroupReadParameter 627
5.15.31 MMC_GroupReadBoolParameter 630
5.15.32 MMC_GroupWriteParameter 633
5.15.33 MMC_GroupWriteBoolParameter 636
5.15.34 MMC_GetGroupMembersInfo 639

Chapter 6: Position, Velocity, Time (PVT) Motion 643

6.1 Overview 643
6.2 PV, PVT Profiler 643
6.3 PVT Interpolation Mode 644
6.3.1 Polynomial Interpolation Functions 645
6.3.1.1 Cubic polynomial – Polynomial Order 3 (eCUBIC_POLYNOM) 645
6.3.1.2 Quintic polynomial – Polynomial Order 5 (eQUINTIC_ON_CUBIC) 645
6.3.2 Sinusoidal interpolation functions 647
6.4 Data loading 651
6.5 PVT motion 651
6.6 On-The-Fly Mode 651
6.6.1 Initializing Table 652
6.6.2 Loading Data 653
6.6.3 Cyclic Mode 653
6.7 File example 654
6.8 Table Functions 654
6.8.1 MMC_TABLE_LIST_OUT 655
6.8.2 MMC_TABLE_LIST_IN 656
6.8.3 MMC_TABLE_DATA_OUT 657
6.8.4 MMC_TABLE_DATA_IN 657
6.8.5 MMC_GetTableList 658
6.8.6 MMC_GetTableInfo 661
6.9 PVT Functions 664
6.9.1 MMC_InitTable 665
6.9.2 MMC_InitTableEx 671
6.9.3 MMC_LoadTableFromFile 677
6.9.4 MMC_UnloadTable 682
6.9.5 MMC_MoveTable 685

6.9.6 MMC_AppendPointsToTable ... 689
6.9.7 MMC_GetTableIndex ... 693

Chapter 7: Electronic CAM ... 697

7.1 Overview ... 697
7.2 CAM Table ... 698
7.2.1 CAM Table File Format ... 698
7.2.2 CAM Table Loading ... 699
7.3 Interpolation Options (Curve Types) ... 700
7.3.1 Polynomial interpolation functions ... 700
7.3.2 Sinusoidal interpolation functions ... 703
7.4 Basic flow for CAM process ... 707
7.4.1 Using an Array in User Program ... 707
7.4.2 Using a File ... 707
7.5 Examples ... 708
7.5.1 Slave Relative ... 708
7.5.2 Slave Absolute ... 710
7.6 ECAM Functions ... 711
7.6.1 MMC_CamTableInit ... 712
7.6.2 MMC_CamTableSelect ... 716
7.6.3 MMC_CamTableUnload ... 720
7.6.4 MMC_CamTableAdd ... 723
7.6.5 MMC_CamTableAddEx ... 726
7.6.6 MC_CamTableSet ... 729
7.6.7 MMC_CamIn ... 732
7.6.8 MMC_CamOut ... 738
7.6.9 MMC_CamStatus ... 741
7.6.10 MMC_CamSetProperty ... 744
7.6.11 MMC_GearIn ... 748
7.6.12 MMC_GearInPos ... 753
7.6.13 MMC_GearOut ... 758
7.7 Application Example ... 761
7.7.1 Master/Slave Offset ... 761
7.7.2 Periodicity Modes of Operation ... 763

Chapter 8: API Services and Operations ... 768

8.1 Main Configuration Function Blocks ... 770
8.1.1 MMC_ChangeToPreOPMode ... 771
8.1.2 MMC_ChangeToOperationMode ... 774
8.1.3 MMC_ClearNodeFbList ... 777
8.1.4 MMC_CmdStatus ... 779
8.1.5 MMC_CloseConnection ... 782
8.1.6 MMC_Config ... 784
8.1.7 MMC_CreateSYNCTimer ... 787
8.1.8 MMC_DestroySYNCTimer ... 788
8.1.9 MMC_DownloadFoE ... 789
8.1.10 MMC_Exit ... 794
8.1.11 MMC_FreeFbStat ... 797
8.1.12 MMC_GetActiveVectorsNum ... 800

8.1.13 MMC_GetErrorCodeDescriptionByID ... 803
8.1.14 MMC_GetFoEStatus ... 806
8.1.15 MMC_GetEnquireFbStatus ... 812
8.1.16 MMC_GetAxisByName ... 815
8.1.17 MMC_GetGroupByName ... 818
8.1.18 MMC_GetGMASOperationMode ... 821
8.1.19 MMC_GetStatusRegister ... 824
8.1.20 MMC_GetResList ... 827
8.1.21 MMC_GetResSnapshot ... 830
8.1.22 MMC_GetVersion ... 833
8.1.23 MMC_GetVersionEx ... 836
8.1.24 MMC_GetLastError ... 839
8.1.25 MMC_InitConnection ... 840
8.1.26 MMC_IPCInitConnection ... 842
8.1.27 MMC_LoadParam ... 844
8.1.28 MMC_RpcInitConnection ... 846
8.1.29 MMC_RpcInitConnectionEx ... 848
8.1.30 MMC_ResetMultiAxisControl ... 850
8.1.31 MMC_ResExportFile ... 853
8.1.32 MMC_ResImportFile ... 856
8.1.33 MMC_SaveParam ... 859
8.1.34 MMC_SetEnquireFbStatus ... 862
8.1.35 MMC_SetDefaultParameters ... 864
8.1.36 MMC_SetDefaultParametersGlobal ... 866
8.1.37 MMC_SetIsToLoadGlobalParams ... 868
8.1.38 MMC_ShowNodeStat ... 870
8.1.39 MMC_GetActiveAxesNum ... 873
8.1.40 MMC_ToggleConsoleOutput ... 876
8.1.41 MMC_GetCyclesCounter ... 878
8.1.42 MMC_WriteGroupOfParameters ... 880
8.1.43 MMC_WriteGroupOfParametersEx ... 884
8.1.44 MMC_ReadGroupOfParameters ... 889
8.1.45 MMC_WaitUntilConditionFB ... 892
8.1.46 MMC_WaitUntilConditionFBEx ... 896
8.1.47 MMC_WriteMemoryRange ... 901
8.1.48 MMC_ReadMemoryRange ... 904
8.1.49 MMC_SetDefaultResources ... 907
8.1.50 MMC_KillRepetitive ... 909
8.1.51 MMC_UserCommandControl ... 911
8.1.52 MMC_SetAllFbExeModeImm ... 915
8.1.53 MMC_GetVerPath ... 917
8.1.54 MMC_DownloadVersion ... 917
8.1.55 MMC_ReadDownloadVersionStatus ... 917
8.1.56 MMC_SetVerPath ... 917

Chapter 9: Process Image(PI) ... 918

9.1 Introduction ... 918
9.2 Variable Types ... 918
9.3 PI User Functions ... 920

9.4 Read/Write RAW Data ... 921
9.5 Recorder PI Mechanism Support ... 922
9.5.1 Differentiation Between Regular Signal and PI Signal ... 922
9.5.2 Determining the Recorded Signals Length ... 922
9.5.3 Recording Large PI Variables ... 923
9.5.4 Nonstandard Variable Types ... 923
9.5.5 MMC_BeginRecordingEx ... 924
9.6 Wait Condition Function Block ... 928
9.7 PI Functions ... 929
9.7.1 MMC_ReadPIVarBOOL ... 930
9.7.2 MMC_ReadPIVarChar ... 933
9.7.3 MMC_ReadPIVarUChar ... 936
9.7.4 MMC_ReadPIVarShort ... 939
9.7.5 MMC_ReadPIVarUShort ... 942
9.7.6 MMC_ReadPIVarInt ... 945
9.7.7 MMC_ReadPIVarUInt ... 948
9.7.8 MMC_ReadPIVarFloat ... 951
9.7.9 MMC_ReadPIVarRaw ... 954
9.7.10 MMC_ReadPIVarLongLong ... 957
9.7.11 MMC_ReadPIVarULongLong ... 960
9.7.12 MMC_ReadPIVarDouble ... 963
9.7.13 MMC_ReadLargePIVarRaw ... 966
9.7.14 MMC_WritePIVarBool ... 969
9.7.15 MMC_WritePIVarChar ... 972
9.7.16 MMC_WritePIVarUChar ... 975
9.7.17 MMC_WritePIVarUShort ... 978
9.7.18 MMC_WritePIVarShort ... 981
9.7.19 MMC_WritePIVarUInt ... 984
9.7.20 MMC_WritePIVarInt ... 987
9.7.21 MMC_WritePIVarFloat ... 990
9.7.22 MMC_WritePIVarRaw ... 993
9.7.23 MMC_WritePIVarULongLong ... 997
9.7.24 MMC_WritePIVarLongLong ... 1000
9.7.25 MMC_WritePIVarDouble ... 1003
9.7.26 MMC_WriteLargePIVarRaw ... 1005
9.7.27 MMC_GetPIVarInfo ... 1008
9.7.28 MMC_GetPIVarInfoByAlias ... 1012
9.7.29 MMC_GetPIVarsRangeInfo ... 1017
9.8 PI Bulk Read User Functions ... 1021
9.8.1 MMC_ConfigureBulkReadPI ... 1022
9.8.2 MMC_PerformBulkReadCmdPI ... 1026
9.9 PI Functions and Implementation Examples ... 1029
9.9.1 C++ Example ... 1029
9.9.2 General PI Test IEC Example ... 1032
9.9.3 PI Full Example in C, and IEC with EtherCAT Configuration Settings ... 1032

Chapter 10: Data Recording ... 1033

10.1 Triggering a Recording ... 1033
10.2 Active Range Support ... 1034

10.3 Using Data Recording in the Maestro ... 1034
10.3.1 Excluding Triggers ... 1035
10.3.2 Including Triggers ... 1035
10.4 Recording Definitions and Parameters ... 1036
10.4.1 Recording Data Signals Bitmask Definitions ... 1036
10.4.2 Recording Parameters ... 1036
10.4.3 Recording Signal Parameters ... 1037
10.4.4 Trigger Modes ... 1045
10.5 Data Recording Functions ... 1047
10.5.1 MMC_BeginRecording ... 1047
10.5.2 MMC_StopRecording ... 1051
10.5.3 MMC_UploadData ... 1054
10.5.4 MMC_RecStatus ... 1057
10.5.5 MMC_UploadDataHeader ... 1060

Chapter 11: Bulk Parameters Reading ... 1065

11.1 Bulk Reading Functions ... 1065
11.1.1 MMC_ConfigBulkRead ... 1066
11.1.2 MMC_PerformBulkRead ... 1073

Chapter 12: API Events ... 1079

12.1 Communication Byte Order ... 1080
12.2 Communication ASYNC Replies (Events) From Drives ... 1080
12.3 Download Firmware Notifications ... 1082
12.4 Emergency Event ... 1082
12.5 Motion Ended Event ... 1082
12.6 HeartBeat Event ... 1083
12.7 PDO Receive Event ... 1083
12.7.1 Event Group equals to 5 or 6 ... 1084
12.7.2 Event Group equals to 11 ... 1084
12.7.3 Event Group Equals to 16 or 17 ... 1085
12.7.4 Event Group equals to 1 – 15 besides 5, 6, 11, 16 and 17 ... 1085
12.8 Home Ended Event ... 1086
12.9 Modbus Write Event ... 1086
12.10 Touch Probe Ended Event ... 1086
12.11 Node Connected Event ... 1087
12.12 Node Initialization Completed ... 1087
12.13 Node Error Event ... 1088
12.14 Stop ON Limit Event ... 1088
12.15 Table Underflow Event ... 1089
12.16 Global Async Reply Event ... 1089
12.17 Notification Function Block Event ... 1090
12.18 Policy Ended Event ... 1091
12.19 Communication Event Mechanism ... 1092
12.20 Events Mask and Enumeration ... 1093
12.21 Asynchronous Events Callback ... 1094
12.21.1 Callback Prototype ... 1094
12.21.2 Data Structure ... 1094
12.21.3 Event Extraction Example ... 1094

12.21.4 Net To local Conversion 1100
12.22 Notification and Events Function Blocks 1100
12.22.1 MMC_InsertNotificationFb 1101
12.22.2 MMC_ClearEventsMask 1104
12.22.3 MMC_DisableMotionEndedEvent 1107
12.22.4 MMC_EnableMotionEndedEvent 1110
12.22.5 MMC_GetEventsMask 1113
12.22.6 MMC_SetEventsMask 1116

Chapter 13: Error Correction Mechanism 1119

13.1 2-D Error Correction 1119
13.2 3-D Error Correction 1121
13.3 Data Representation 1122
13.3.1 1-D Representation 1122
13.3.2 2-D Representation 1123
13.3.3 3-D Representation 1124
13.4 Error Correction Functions 1125
13.4.1 MMC_LoadErrorCorrTable 1126
13.4.2 MMC_EnableErrorCorrTable 1129
13.4.3 MMC_GetErrorTableStatus 1132
13.4.4 MMC_DisableErrorCorrTable 1136
13.4.5 MMC_UnloadErrorCorrTable 1139

Chapter 14: Maestro Hardware and Software Limits Handling 1142

14.1 Introduction 1142
14.2 Interfaces 1142
14.2.1 Software Position Limits 1142
14.2.2 Status Register 1144
14.2.3 ACS\SingleAxis 1145
14.2.4 MCS 1145
14.2.5 MCS Limit Register (32 bits) 1145
14.2.6 Stop Parameters 1146
14.2.7 Stop-on-Limit Event 1146
14.3 Function Block Pre-Insertion Behavior 1147
14.3.1 Single Axis 1147
14.3.2 Multi Axes Group 1148
14.4 Real Time Behavior 1149
14.5 System Constraints And Limitations 1150

Chapter 15: Saving Maestro User Program Parameters 1152

15.1 Introduction 1152
15.2 The MMCUserParams C++ Class 1153
15.2.1 Open 1155
15.2.2 Close 1156
15.2.3 Read 1157
15.2.4 GetXmlFileRoot 1160
15.2.5 GetXmlFileDescrp 1161
15.2.6 SetSpeakDbgLvl 1162
15.2.7 UPXML Functions Code Examples 1163

15.2.8 UpxmlEg.xml - Input File Example 1165
15.2.9 Program output example 1166

Chapter 16: Connectivity and Configuration 1167

16.1 Network Function Blocks 1167
16.1.1 MMC_CloseUdpChannel 1168
16.1.2 MMC_GetDefGateway 1171
16.1.3 MMC_GetDhcp 1174
16.1.4 IMMC_GetIpAddr 1177
16.1.5 MMC_GetIpMask 1180
16.1.6 MMC_GetServerIp 1183
16.1.7 MMC_NetworkInfo 1186
16.1.8 MMC_NetworkScan 1191
16.1.9 MMC_OpenUdpChannel 1194
16.1.10 MMC_SetDefGateway 1197
16.1.11 MMC_SetDhcp 1200
16.1.12 MMC_SetIpAddr 1203
16.1.13 MMC_SetIpMask 1206
16.1.14 MMC_SetServerIp 1209
16.2 Host Communication 1212
16.3 Modbus Communication Function Blocks 1212
16.3.1 MMC_MbusIsRunning 1213
16.3.2 MMC_MbusReadCoilsTable 1216
16.3.3 MMC_MbusReadHoldingRegisterTable 1219
16.3.4 MMC_MbusReadInputsTable 1222
16.3.5 MMC_MbusStartServer 1225
16.3.6 MMC_MbusStopServer 1228
16.3.7 MMC_MbusWriteCoilsTable 1231
16.3.8 MMC_MbusWriteHoldingRegisterTable 1234
16.4 CANbus Drive Communication 1237
16.4.1 Master – Slave Relations 1238
16.4.2 CANopen DS-402 Modes of Operation 1238
16.4.3 PDO Mapping 1239
16.4.4 Using Event Groups 16 and 17 1242
16.4.5 Servo Drive Sub-Index 1243
16.4.6 SYNC and Time Stamp 1243
16.4.7 CAN Bulk Upload 1244
16.4.8 CAN – PDO, SDO Configurator 1244
16.5 CANbus Function Blocks 1245
16.5.1 MMC_CancelVirtualEncoder 1246
16.5.2 MMC_CancelParamEvPDO3 1248
16.5.3 MMC_CancelParamEvPDO4 1251
16.5.4 MMC_CfgRegParamEvPDO3 1254
16.5.5 MMC_CfgRegParamEvPDO4 1259
16.5.6 MMC_CfgUserParamEvPDO3 1264
16.5.7 MMC_CfgUserParamEvPDO4 1269
16.5.8 MMC_ChangeDefaultPDOConfiguration 1274
16.5.9 MMC_ChngOpMode 1277
16.5.10 MMC_ConfigEventModePDO3 1280

16.5.11 MMC_ConfigEventModePDO4 ... 1283
16.5.12 MMC_ConfigVirtualEncoder ... 1286
16.5.13 MMC_GetAxisByCanId ... 1289
16.5.14 MMC_GetPDOInfo ... 1292
16.5.15 MMC_GetSyncTime ... 1296
16.5.16 MMC_PDOGeneralRead ... 1299
16.5.17 MMC_PDOGeneralWrite ... 1302
16.5.18 MMC_ReceiveCANRawData ... 1305
16.5.19 MMC_SendCANRawData ... 1308
16.5.20 MMC_SendandReceiveCANRawData ... 1311
16.5.21 MMC_SendCmd ... 1315
16.5.22 MMC_SetHeartBeatConsumer ... 1318
16.5.23 MMC_SetSyncTime ... 1321
16.5.24 MMC_StartBulkUpload ... 1324
16.5.25 MMC_GetBulkUploadStatus ... 1327
16.5.26 MMC_GetBulkUploadData ... 1331
16.6 EtherCAT Drive Communication ... 1334
16.6.1 Elmo EtherCAT ... 1335
16.6.2 Elmo Slave Drives ... 1336
16.6.3 EtherCAT with Maestro ... 1338
16.6.4 EtherCAT Gateway ... 1338
16.7 EtherCAT and CANbus Function Blocks ... 1339
16.7.1 MMC_DisableEthercatConfigMode ... 1340
16.7.2 MMC_EnableEthercatConfigMode ... 1343
16.7.3 MMC_ECATIODisableDIChangedEvent ... 1345
16.7.4 MMC_ECATIOEnableDIChangedEvent ... 1348
16.7.5 MMC_ECATIOReadDigitalInput ... 1351
16.7.6 MMC_ECATIOReadAnalogInput ... 1354
16.7.7 MMC_ECATIOWriteAnalogOutput ... 1357
16.7.8 MMC_ECATIOWriteDigitalOutput ... 1360
16.7.9 MMC_GetCommStatistics ... 1363
16.7.10 MMC_GetEthercatCommStatistics ... 1367
16.7.11 MMC_GetCommDiagnostics ... 1376
16.7.12 MMC_GetReactorStatistics ... 1380
16.7.13 MMC_IsEthercatConfigMode ... 1383
16.7.14 MMC_ResetCommDiagnostics ... 1386
16.7.15 MMC_ResetCommStatistics ... 1389
16.7.16 MMC_SendSDO ... 1392
16.7.17 MMC_SendSdoAsync ... 1396
16.8 Interpreter Command Functions ... 1399
16.8.1 Get Function – Asynchronous Mode ... 1400
16.8.2 MMC_ElmoExecuteLabel ... 1401
16.8.3 MMC_ElmoSetParameter ... 1405
16.8.4 MMC_ElmoGetParameter ... 1407
16.8.5 MMC_ElmoGetArray ... 1409
16.8.6 MMC_ElmoGetArrayAndRetrieveData ... 1411
16.8.7 MMC_ElmoGetParameterAndRetrieveData ... 1413
16.8.8 MMC_ElmoSetArray ... 1415
16.8.9 MMC_ElmoQueryOperationFIFOIndex ... 1417

16.8.10 MMC_ElmoQueryOperationFIFORetrieveData 1418
16.8.11 MMC_ElmoQueryOperationFIFOIndexReset 1419
16.8.12 MMC_ElmoCall 1420
16.9 EtherNetIP Communication 1422
16.9.1 Terminology 1423
16.9.2 Configuring the Ethernet IP Device as Adapter 1425
16.9.3 Ethernet/IP Setup 1430
16.10 EtherNetIP Functions 1434
16.10.1 EipGetAdpTagRefByName 1435
16.10.2 EipWriteAdpTag 1437
16.10.3 EipReadAdpTag 1440
16.10.4 EipGetAssemblyRefByInstance 1443
16.10.5 EipGetAssemblyRefByName 1445
16.10.6 EipSetAssembly 1447
16.10.7 EipGetAssembly 1450
16.10.8 EipGetDevTagRefByName 1452
16.10.9 EipSetDevTag 1454
16.10.10 EipGetDevTag 1457
16.10.11 EipReadDevTagData 1459
16.10.12 EipSyncGetDevTag 1461
16.10.13 EipCheckDevTagReply 1463
16.10.14 EipOpenSession 1465
16.10.15 EIPCloseSession 1468
16.10.16 EipCreate 1470
16.10.17 EipDestroy 1472
16.10.18 Functions and Implementation Example 1474
16.11 DS-401 CANbus I/O Communications 1479
16.12 DS-401 Function Blocks 1479
16.12.1 MMC_CancelGeneralRPDO3 1480
16.12.2 MMC_CancelGeneralRPDO4 1482
16.12.3 MMC_CancelGeneralTPDO3 1485
16.12.4 MMC_CancelGeneralTPDO4 1487
16.12.5 MMC_ConfigGeneralRPDO3 1490
16.12.6 MMC_ConfigGeneralRPDO4 1493
16.12.7 MMC_ConfigGeneralTPDO3 1496
16.12.8 MMC_ConfigGeneralTPDO4 1498
16.12.9 MMC_DisableDS401DIChangedEvent 1501
16.12.10 MMC_EnableDS401DIChangedEvent 1504
16.12.11 MMC_ReadDS401DIGroup 1507
16.12.12 MMC_ReadDS401DInput 1510
16.12.13 MMC_WriteDS401DOGroup 1513
16.12.14 MMC_WriteDS401DOutput 1516

Chapter 17: Programming in C++ 1519

17.1 Introduction 1519
17.1.1 CMMCException 1520
17.2 The CMMCNode class 1521
17.2.1 CMMCNode Class Functions Code Example 1 1522
17.2.2 ConfigPDOEventMode 1534

17.2.3 EtherCATPIVarInfo ... 1535
17.2.4 EtherCATReadMemoryRange ... 1537
17.2.5 EthercatReadPIVar ... 1538
17.2.6 EtherCATWriteMemoryRange ... 1540
17.2.7 EthercatWritePIVar ... 1541
17.2.8 Reset ... 1543
17.2.9 ReadStatus ... 1544
17.2.10 SendSDOCmd ... 1545
17.2.11 SendSDODownload ... 1546
17.2.12 SendSDOUpload ... 1547
17.2.13 SendSDOUploadAsync ... 1548
17.2.14 RetreiveSDOUploadAsync ... 1549
17.2.15 PDOGeneralRead ... 1550
17.2.16 PDOGeneralWrite ... 1551
17.2.17 GetPDOInfo ... 1552
17.2.18 SetBoolParameter ... 1553
17.2.19 SetParameter ... 1554
17.2.20 GetBoolParameter ... 1555
17.2.21 GetParameter ... 1556
17.2.22 GetAxisError ... 1557
17.3 The CMMCAxis class ... 1558
17.3.1 DisableMotionEndedEvent ... 1559
17.3.2 EnableMotionEndedEvent ... 1560
17.3.3 SetDefaultManufacturerParameters ... 1561
17.3.4 GetAxisByName ... 1562
17.3.5 InitAxisData ... 1563
17.3.6 CMMCAxis Class Functions Code Examples ... 1564
17.4 The CMMCMotionAxis class ... 1565
17.4.1 CMMCMotionAxis Source Code Examples ... 1567
17.4.2 CAM Process Example ... 1568
17.4.3 GetFbDepth ... 1569
17.4.4 EnableMotionEndedEvent ... 1570
17.4.5 DisableMotionEndedEvent ... 1570
17.4.6 Dwell ... 1571
17.4.7 InsertNotificationFb ... 1572
17.4.8 WaitUntilConditionFB ... 1573
17.4.9 WaitUntilConditionFBEX ... 1573
17.4.10 KillRepetitive ... 1574
17.4.11 InitPVTTable ... 1575
17.4.12 InitPTTable ... 1577
17.4.13 LoadPVTTableFromFile ... 1579
17.4.14 LoadPTTableFromFile ... 1580
17.4.15 AppendPVTPoints ... 1581
17.4.16 AppendPTPoints ... 1583
17.4.17 GetTableList ... 1584
17.4.18 GetTableInfo ... 1585
17.4.19 MovePVT ... 1586
17.4.20 UnloadPVTTable ... 1586
17.4.21 CamGetStatus ... 1587

17.4.22 CamTableInit 1588
17.4.23 CamTableSelect 1590
17.4.24 CamIn 1591
17.4.25 CamOut 1594
17.4.26 CamTableUnload 1594
17.4.27 CamTableAdd 1595
17.4.28 CamTableAddEx 1597
17.4.29 CamTableSet 1598
17.5 The DLLMMCPP_API MMC_MOTIONPARAMS_GROUP class 1599
17.5.1 MMC_MOTIONPARAMS_GROUP() 1600
17.6 The CMMCGroupAxis class 1604
17.6.1 CMMCGroupAxis Class Functions Code Example 1 1607
17.6.2 CMMCGroupAxis Class Functions Code Example 2 1617
17.6.3 CMMCGroupAxis Class Functions Code Example 3 1629
17.6.4 CMMCGroupAxis Class Functions Code Example 4 1640
17.6.5 CMMCGroupAxis(CMMCGroupAxis& axis) 1648
17.6.6 InitAxisData 1650
17.6.7 GetGroupAxisByName 1651
17.6.8 SetDefaultParams 1652
17.6.9 SetCartesianKinematics 1654
17.6.10 SetDeltaRobotKinematics 1654
17.6.11 SetKinematic 1655
17.6.12 SetKinTransform 1656
17.6.13 SetCartesianTransform 1657
17.6.14 ReadCartesianTransform 1659
17.6.15 TrackConveyorBelt 1660
17.6.16 TrackRotaryTable 1663
17.6.17 SetKinTransformDelta 1666
17.6.18 SetKinTransformCartesian 1667
17.6.19 SetKinTransformScara 1668
17.6.20 SetKinTransformThreeLink 1669
17.6.21 RemoveAxisFromGroup 1670
17.6.22 MoveCircularAbsolute 1671
17.6.23 MoveCircularAbsoluteCenter 1673
17.6.24 MoveCircularAbsoluteBorder 1674
17.6.25 MoveCircularAbsoluteRadius 1675
17.6.26 MoveCircularAbsoluteAngle 1676
17.6.27 MoveLinearAbsolute 1677
17.6.28 MoveLinearRelative 1678
17.6.29 GroupSetOverride 1679
17.6.30 SetBoolParameter 1680
17.6.31 SetParameter 1681
17.6.32 GetBoolParameter 1682
17.6.33 GetParameter 1683
17.6.34 GroupSetPosition 1684
17.6.35 GroupReadStatus 1685
17.6.36 GetStatusRegister 1685
17.6.37 GetMcsLimitRegister 1686
17.6.38 ReadStatus 1687

17.6.39 Reset ... 1688
17.6.40 GetMembersInfo ... 1688
17.6.41 GroupEnable ... 1689
17.6.42 GroupDisable ... 1689
17.6.43 GroupReset ... 1690
17.6.44 GroupReadError ... 1690
17.6.45 GroupReadActualVelocity ... 1691
17.6.46 AddAxisToGroup ... 1692
17.6.47 GroupReadActualPosition ... 1693
17.6.48 GroupStop ... 1694
17.6.49 GroupHalt ... 1695
17.6.50 MoveLinearAbsoluteRepetitive ... 1696
17.6.51 MoveLinearRelativeRepetitive ... 1697
17.6.52 MovePolynomAbsolute ... 1698
17.6.53 MoveLinearAdditive ... 1700
17.6.54 MovePath ... 1701
17.6.55 PathDeselect ... 1703
17.6.56 PathSelect ... 1704
17.6.57 PathGetLengths ... 1705
17.6.58 EthercatWriteMemoryRange ... 1706
17.7 The CMMCSingleAxis class ... 1707
17.7.1 CMMCSingleAxis Class Functions Code Example 1 ... 1710
17.7.2 CMMCSingleAxis Class Functions Code Example 2 ... 1714
17.7.3 CMMCSingleAxis Class Functions Code Example 3 ... 1726
17.7.4 SetDefaultParams ... 1738
17.7.5 SetDefaultHomeDS402Params ... 1743
17.7.6 SetDefaultHomeParams ... 1744
17.7.7 Home ... 1745
17.7.8 HomeDS402 ... 1746
17.7.9 MoveAbsolute ... 1748
17.7.10 MoveAdditive ... 1752
17.7.11 MoveRelative ... 1754
17.7.12 MoveVelocity ... 1756
17.7.13 MoveAbsoluteRepetitive ... 1757
17.7.14 MoveRelativeRepetitive ... 1760
17.7.15 MoveAdditiveRepetitive ... 1763
17.7.16 MoveTorque ... 1766
17.7.17 GetFbDepth ... 1768
17.7.18 PositionProfile ... 1769
17.7.19 TouchProbeDisable ... 1770
17.7.20 TouchProbeEnable ... 1770
17.7.21 SetOpMode ... 1771
17.7.22 GetOpMode ... 1772
17.7.23 PowerOn ... 1774
17.7.24 PowerOff ... 1775
17.7.25 GetActualPosition ... 1775
17.7.26 GetActualVelocity ... 1776
17.7.27 GetActualTorque ... 1776
17.7.28 Halt ... 1777

17.7.29 Stop ... 1778
17.7.30 GetAxisError ... 1779
17.7.31 GetDigInput[s] ... 1779
17.7.32 GetDigOutputs32Bit ... 1780
17.7.33 GetDigOutputs ... 1780
17.7.34 SetDigOutputs32Bit ... 1781
17.7.35 SetDigOutputs ... 1782
17.7.36 SetOverride ... 1783
17.7.37 ConfigPDO ... 1784
17.7.38 CancelPDO ... 1786
17.7.39 ChangeDefaultPDOConfig ... 1787
17.7.40 ElmoSetAsyncArray ... 1788
17.7.41 ElmoSetAsyncParam ... 1789
17.7.42 ElmoGetAsyncIntParam ... 1791
17.7.43 ElmoGetAsyncFloatParam ... 1793
17.7.44 ElmoGetAsyncIntArray ... 1794
17.7.45 ElmoGetAsyncFloatArray ... 1795
17.7.46 ElmoGetSyncParam ... 1796
17.7.47 ElmoGetSyncArray ... 1797
17.7.48 ElmoCallAsync ... 1798
17.7.49 ElmoExecute ... 1799
17.7.50 ElmoIsReplyAwaiting ... 1799
17.7.51 ElmoGetReply ... 1800
17.7.52 ConfigVirtualEncoder ... 1801
17.7.53 CancelVirtualEncoder ... 1803
17.7.54 SetPosition ... 1803
17.7.55 SetParameter ... 1804
17.7.56 SetBoolParameter ... 1805
17.7.57 AxisLink ... 1806
17.7.58 AxisUnLink ... 1807
17.7.59 GetBoolParameter ... 1807
17.7.60 GetParameter ... 1808
17.8 The CMMCDS401Axis class ... 1809
17.8.1 ConfigGeneralRPDO3 ... 1810
17.8.2 ConfigGeneralRPDO4 ... 1811
17.8.3 CancelGeneralRPDO3 ... 1812
17.8.4 CancelGeneralRPDO4 ... 1812
17.8.5 ConfigGeneralRPDO4 ... 1813
17.8.6 ConfigGeneralTPDO3 ... 1813
17.8.7 ConfigGeneralTPDO4 ... 1814
17.8.8 CancelGeneralTPDO3 ... 1814
17.8.9 CancelGeneralTPDO4 ... 1815
17.9 The CMMCDS406Axis class ... 1816
17.9.1 GetActualPosition ... 1818
17.10 The CMMCECATIO class ... 1819
17.10.1 ECATIOEnableDIChangedEvent ... 1820
17.10.2 ECATIODisableDIChangedEvent ... 1820
17.10.3 ECATIOReadDigitalInput ... 1821
17.10.4 ECATIOWriteDigitalOutput ... 1822

17.10.5 ECATIOReadAnalogInput ... 1822
17.10.6 ECATIOWriteAnalogOutput ... 1823
17.11 The CMMCPPGlobal class ... 1824
17.11.1 CMMCPPGlobal Class Functions Code Example ... 1826
17.11.2 RegisterRTE ... 1836
17.11.3 RegisterWarningClbk ... 1836
17.11.4 SetThrowFlag ... 1837
17.11.5 SetThrowWarningFlag ... 1837
17.11.6 SetPrintErrorFlag ... 1838
17.11.7 SetPrintWarningFlag ... 1838
17.11.8 ThrowMessage ... 1839
17.11.9 SetConnectionType ... 1839
17.11.10 SetMessageFileName ... 1840
17.11.11 GetSyncTime ... 1840
17.11.12 SetSyncTime ... 1841
17.11.13 CreateSYNCTimer ... 1842
17.11.14 DestroySYNCTimer ... 1843
17.11.15 GetConnectionReg ... 1843
17.11.16 ConfigBulkRead ... 1844
17.11.17 PerformBulkRead ... 1846
17.11.18 RegisterConnection ... 1847
17.11.19 GetConnectionReg ClearConnectionReg ... 1847
17.12 The CMMCConnection class ... 1848
17.12.1 CMMCConnection Class Functions Code Example 1 ... 1851
17.12.2 CMMCConnection Class Functions Code Example 2 ... 1861
17.12.3 Event Type Definitions ... 1873
17.12.4 ConnectIPC ... 1875
17.12.5 ConnectIPCEx ... 1875
17.12.6 ConnectRPC ... 1876
17.12.7 ConnectRPCEx ... 1877
17.12.8 SetGlobalBoolParameter ... 1878
17.12.9 GetGlobalBoolParameter ... 1879
17.12.10 GetGlobalParameter ... 1883
17.12.11 SetIsToLoadGlobalParams ... 1884
17.12.12 SetHeartBeatConsumer ... 1885
17.12.13 CallbackFunc ... 1886
17.12.14 RegisterEventCallback ... 1887
17.12.15 RegisterSyncTimerFunction ... 1888
17.13 The CMMCNetwork class ... 1889
17.13.1 CMMCNetwork Class Functions Code Example ... 1890
17.13.2 GetCommDiagnostics ResetCommDiagnostics ... 1900
17.13.3 ResetCommStatistics ... 1901
17.13.4 GetNetworkInfo ... 1902
17.14 The CMMCHostComm class ... 1903
17.14.1 CMMCHostComm Class Functions Code Example 1 ... 1904
17.14.2 MbusStartServer ... 1916
17.14.3 MbusStopServer ... 1917
17.14.4 MbusReadHoldingRegisterTable ... 1918
17.14.5 MbusWriteHoldingRegisterTable ... 1919

17.14.6 MbusIsRunning 1919
17.14.7 MbusReadCoilsTable 1920
17.14.8 MbusWriteCoilsTable 1921
17.14.9 MbusReadInputsTable 1922
17.14.10 SetModbus[LongSwapped][Short] 1923
17.15 The CMMCModbusBuffer class 1924
17.16 The CMMCModbusSwapBuffer class 1925
17.17 The CMMCErrorCorr class 1926
17.17.1 LoadErrorCorrTable 1927
17.17.2 UnloadErrorCorrTable 1928
17.17.3 EnableErrorCorrTable 1929
17.17.4 DisableErrorCorrTable 1930
17.17.5 GetErrorTableStatus 1931
17.17.6 IsTableEnabled 1932
17.17.7 IsTableEnabled 1933
17.18 The CMMCBulkRead class 1934
17.18.1 CMMCBulkRead Source Code Examples 1935
17.18.2 CMMCBulkRead 1936
17.18.3 Config 1937
17.18.4 BulkRead 1937
17.19 The CMMCUserParams class 1938
17.19.1 Using the XML structure 1938
17.19.2 General 1939
17.19.3 XML Data File Example 1940
17.19.4 Errors Applicable to the CMMCUserParams Class 1941
17.19.5 CMMCUserParams class Functions and Examples 1942
17.20 The CMMCEIPSession class 1942
17.20.1 EIPCloseSession 1943
17.20.2 EipCreate 1943
17.20.3 EipDestroy 1943
17.20.4 EipOpenSession 1944
17.21 The CMMCEIPDataType class 1945
17.21.1 EthernetIP Source Code Examples 1946
17.21.2 EipTagInit 1952
17.21.3 EipSetTag 1952
17.21.4 EipGetTag 1953
17.21.5 EipCheckReply 1953
17.21.6 EipGetData 1954
17.22 TCP/IP and UDP/IP C++ User Libraries 1955
17.23 The CMMCUDP class 1956
17.23.1 Synchronous and Asynchronous Behavior 1957
17.23.2 Mode Of Operation 1957
17.23.3 UDP Code Examples 1957
17.23.4 CMMCUDP Class Functions Code Example 1960
17.23.5 Create 1970
17.23.6 SendTo 1971
17.23.7 ReceiveFrom 1971
17.23.8 IsWriteable 1972
17.23.9 IsReady 1972

17.23.10 Connect 1973
17.23.11 Send 1974
17.23.12 Receive 1975
17.23.13 Create (overloaded) 1976
17.23.14 GetIP 1977
17.23.15 SetMaxSize 1977
17.23.16 SetTimeout 1977
17.24 The CMMCTCP class 1978
17.24.1 Synchronous and Asynchronous Behavior 1978
17.24.2 Mode of Operation 1978
17.24.3 Accept 1979
17.24.4 IsReadable 1979
17.24.5 IsWritable 1980
17.24.6 Connect 1981
17.24.7 Send 1982
17.24.8 Receive 1983
17.24.9 Create 1984
17.24.10 TCP Code Examples 1985
17.25 The CMMCEoE Class 1987
17.25.1 CMMCEoE Class Functions Code Example 1 1988
17.25.2 CMMCEoE Class Functions Code Example 2 1998
17.25.3 ElmoSetAsyncArray 2000
17.25.4 ElmoSetAsyncParameter 2001
17.25.5 ElmoSetArray 2002
17.25.6 ElmoSetParameter 2003
17.25.7 ElmoGetAsyncArray 2004
17.25.8 ElmoGetAsyncParameter 2004
17.25.9 ElmoGetArray 2005
17.25.10 ElmoGetParameter 2006
17.25.11 ElmoReadData 2007
17.25.12 ElmoCall 2008
17.25.13 ElmoCallAsync 2008
17.25.14 ElmoAck 2009
17.25.15 ElmoExecuteLabelAsync 2009
17.25.16 ElmoExecuteLabel 2010

Chapter 18: IEC 61131-3 Special Functions 2011

18.1.1 ElmoIECLibVers 2011
18.1.2 ElmoIECRTVers 2011
18.1.3 Elmo_RetainLoad 2012
18.1.4 Elmo_RetainSave 2013
18.1.5 MMC_SetImmediateExec 2014

Chapter 19: Appendix 2015

19.1 Axis Parameters (Explanations) 2015

Chapter 1: Introduction

This document describes the administrative and motion API of the Elmo Maestro motion controllers (Gold Maestro and Platinum Maestro). The Maestro is Elmo's family of Network Motion Controllers. They are network-based, and operate in conjunction with Elmo's intelligent servo drive family, to provide a total network motion controller solution.

The Maestro is designed to support both the existing SimplIQ Line servo drives, based on standard CAN Open network architecture, as well as the Gold Line family, with EtherCAT networking.

As true network controllers, Elmo's Maestro family, SimplIQ, and servo drives, share the motion processing workload in distributed motion control architecture. The best servo performances are achieved by combining Elmo's servo drives, and the new real-time motion control capabilities of the Maestro main controllers.

The Maestro operates as a Network Motion Controller to support:

- Full, Real-Time, Multi-Axis motion synchronization
- Advanced user programming capabilities based on well-known standards
- Deterministic control over Motions, IO's, and processes in the system

1.1 Maestro over (Go) Standard

The Maestro offers real-time motion control support, for full multi-axis system synchronization, using the industry interface PLCopen for Motion Control standard. This is the Maestro over standard, applicable to the Platinum Line (Platinum Maestro), Gold Line (Gold Maestro), and SimplIQ Line of motion controllers. The use of native C and C++ programming support (run on the Maestro target) dramatically accelerates execution of user level programs, while maintaining the same PLCopen Motion API definitions as a standard software API.

The operation of C and C++ based programs is optimal, and will result in the best overall system performance, since they generate machine code that runs directly on the target Maestro hardware processor.

1.2 What the API Does

The purpose of the PLCopen Motion API is to produce a standardized motion control API solution without compromising on system performances, based on the PLCopen Standard. This is achieved by the Maestro software architecture, since the low-level controller real-time motion engine directly implements the PLCopen motion API, therefore no intermediate layers are required, and performances are optimal.

Software interfaces to the Maestro Motion API are implemented to the users' convenience, via a dedicated API library. This library supports the following:

- Interfacing the Maestro PLCopen Motion API from a host computer via Ethernet TCP/IP
- Interfacing the Maestro PLCopen Motion API from user programs, running on the Maestro product

An identical API library is used at the Development / Host PC with the API library accessing the same API server to perform the desired user operations.

This same Maestro API operates with EtherCAT communication and for the SimplIQ family of network controllers.

The user has the ability to store such programs on the Maestro FLASH, and to run them at power-up. This naturally results in a faster and more optimized method of operating the Maestro motion API.

1.3 Terminology

The terminology used in this document covers language used throughout the servo drive, controller, and communication industry and are not necessarily specific to Elmo Motion Control Ltd.

Term	Explanation
ACS	Axes Coordinate System: The system of coordinates related to the physical motors.
Axis	Axis is the most basic motion object and is used to control the motion of a single motor/axis.
Blending	A method for consecutive function blocks to cooperate in the transition from the first to the next.
CAN	Controller Area Network. Data link layer protocol for serial communication as specified in ISO 11898-1 (1999).
CiA	CAN in Automation international users and manufacturers group e.V. It is a non-profit association promoting Controller Area Network (CAN).
COB	Communication Object, consisting of one or more CAN frames. Any information transmitted via CANopen has to be mapped into COBs.
COB-ID	COB-Identifier. Identifies a COB uniquely in a CAN network. The identifier also determines the priority of that COB in the data link layer.
CoE	CANopen over EtherCAT. Defines a standard way to access the CANopen protocol and includes an object dictionary, SDO, PDO, and emergency messages.
Contour curve	An inserted curve that modifies the original path. It is the resulting curve after blending.
Coordinate system	The reference system in which a coordinate or path is described.
Corner deviation	The shortest distance between the programmed corner point and the contour curve.
Corner distance	Distance of the start point of the contour curve to the programmed target point.
Direction	The orientation components of a vector in space. Note: This is different from the MC_Direction input.
Drive	A unit controlling a motor via the current and timing in its coils.
EoE	Ethernet over EtherCAT. Fully Ethernet compatible and defines a standard way to exchange or tunnel standard Ethernet frames. Used to create Maestro Master and Drives as Slaves in EASII and other applications for both diagnostics and download of files.
FB	Function Block
FIFO	First In, First Out. An abstraction in ways of organizing and manipulation of data relative to time and prioritization. This expression describes the principle of a queue processing technique or servicing conflicting demands by ordering process by first-come, first-served

Term	Explanation
	(FCFS) behavior: what comes in first is handled first, what comes in next waits until the first is finished, etc.
FoE	File over EtherCAT. Similar to TFTP, enables access to any data structure in the device, and defines a standard way to download and upload firmware and other files over the EtherCAT network. Used as a download of hardware configuration files and updates. Refer to the functions 6.3.10 MMC_DownloadFoE, and 6.3.11 MMC_GetFoEStatus
G-MAS	Gold Maestro Application Software also known as the Gold Maestro Network Motion Controller performs synchronized multi axis motions in the system (such as circle, line etc.), using a real time communication protocol so that all drives are synchronized to a specific SYNC signal in the system It operates as a master, independent of any host system. In operational mode, it periodically sends data to the slaves that may override the data that a user sends from a host system.
GoS	G-MAS over SimplIQ
Group	Group of axes
Group-FB	The set of function blocks that can operate on a group of axes.
HPT	High Priority Task as against LPT (Low Priority Task), and MPT (Medium Priority Task), used in Embedded Linux, RTOS, and Parallel Programing for programming in robotics. This refers to the task priority in running threads, which may or may not lock resources.
IO	Input and output
IPC	Inter-process communication (IPC) is a set of techniques to exchange data among multiple threads in one or more processes. Processes may be running on one or more network-linked systems. IPC techniques are divided into methods for passing messages, synchronization, shared memory, and remote procedure calls (RPC). IPC may vary, depending on the bandwidth and communication latency between the threads, and the type of data being communicated. C Programs located on the Maestro use the IPC method.
Maestro	Refers to the Gold Maestro and Platinum Maestro motion controllers.
Masking	A form of cloaking of communication addresses. The netmask is a bitmask used to separate the bits of the network identifier from the bits of the host identifier. It is written in the same notation used to denote IP addresses.
MCS	Machine Coordinate System: The system of coordinates that is related to the machine. Sometimes called World Coordinate System or Base Coordinate System. With Cartesian built machines, MCS is a Cartesian Coordinate system The coordinate system from the physical multiple axes ACS is linked to the MCS via a kinematic transformation (forward and backward conversion).
Motor	An actuator focused to a movement, converting electrical energy into a force or torque.

Term	Explanation
Mutex	Mutual exclusion. Mutual exclusion algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections. A critical section is a piece of code in which a process or thread accesses a common resource. The critical section by itself is not a mechanism or algorithm for mutual exclusion. A program, process, or thread can have the critical section in it without any mechanism or algorithm, which implements mutual exclusion.
Orientation	The rotational components of a vector in space.
Path	Set of continuous positions and orientation information in multi-dimensional space. This may be geometrically described as a space curve that the axes group TCP moves along.
Path Data	Description of a path, which can include additional information like velocity and acceleration.
PDO	Process Data Object
PDS	Power Drive System
Platinum Maestro	Platinum Maestro Application Software also known as the Platinum Maestro Network Motion Controller performs synchronized multi axis motions in the system (such as circle, line etc.), using a real time communication protocol so that all drives are synchronized to a specific SYNC signal in the system It operates as a master, independent of any host system. In operational mode, it periodically sends data to the slaves that may override the data that a user sends from a host system.
Position	Position means a point in space that is defined by different coordinates. Depending on the used system and transformation, it can consist of up to six dimensions (coordinates), three Cartesian coordinates in space and three coordinates for the orientation. In ACS, there can be even more than six coordinates. If the same position is defined in different coordinate systems, the values of the coordinates are different.
PVT	Position velocity time interpolation mode
RPC	A remote procedure call (RPC) is an inter-process communication with the Maestro host allowing a program to initiate a subroutine or procedure. The programmer essentially writes the same code whether the subroutine is local to the executing program, or remote. For example, the EASII uses RPC to communicate with the Maestro family motion controllers.
RPDO	Receive Process Data Object. Communication object of a device, which contains output data.
Scara	A special kinematic for robot or handling applications.
SDO	Service Data Object. Peer-to-peer communication with access to the Object Dictionary of a CANopen device.
Speed	Speed is the absolute value of the velocity without direction.
Synchronization	Combines an axis or axes group (as slave) with an axis as master in order for the slave to execute its synchronized path with the progress of the master, and therefore linked to a single-dimension source for synchronization.

Term	Explanation
TCP	Tool Centre point, the point in the machine that is commanded to move, typically to the center or the head of the tool. It can be described in different coordinate systems.
TPDO	Transmit Process Data Object. Communication object of a device, which contains input data.
Tracking	Is characterized by an axis group that tracks with its movement, the movement of another axis group.
Trajectory	Time dependent description of the path the TCP of an axes group moves along. Additionally to the geometrical description of the space curve, time dependent state variables like velocity, acceleration, jerk, forces etc. are also specified.
Velocity	For a group of axes this means: For ACS, the velocities of the different axes. For MCS and PCS it provides the velocity of the TCP.
XML	Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable
XML DOM	Document Object Model (DOM), is an application programming interface (API) for a valid HTML and well-formed XML documents.
RapidXml	RapidXml is an attempt to create the fastest XML DOM parser possible while retaining usability, portability and reasonable W3C compatibility. It is an in-situ parser written in C++, with parsing speed approaching that of strlen() function executed on the same data
CAM Table	Content addressable memory (CAM) table refers to a dynamic table in the Maestro Shared Memory, which maps master position to ECAM Slave position.
ECAM Master	Axis, which functions as master in ECAM operation (aka CAM process).
ECAM Slave	Axis, which functions as slave in ECAM operation (aka CAM process).
Coupling	The stage in which master and ECAM Slave are synchronized by CAM process (aka Engagement). It occurs when ECAM Master reaches to MasterSyncPosition (see on the follows).
Decoupling	The stage where CAM process ended. That is to say, ECAM Master and ECAM Slave synchronization is ended (aka Disengagement).
Sync Position	Master position in which ECAM Master and ECAM Slave should be synchronized.
Start Position	ECAM Master position in which master may start a Ramp-In process before Coupling. It is defined as a backward distance from Sync Position. At this phase we do not support Ram-In for CAM process, therefore Start Position for the time being is equivalent to Sync Position (see above).
In-CAM	A situation in which the ECAM Master reaches the 'Start Position' within CAM table, regardless of whether the ECAM Master and ECAM Slave are synchronized. As said above, at this phase it is equivalent to synchronization.

1.4 How to Use this Document

This document allows a programmer in C, C++, and IEC 61331, to communicate and operate the Maestro Network Motion Controllers. It is designed to aid the programmer in setting specific parameters for the appropriate function blocks used by the customer. This section describes how to use the detailed information within each function block, set, and customize its parameters.

The chapters in this document are divided according to the following sections:

Chapter 1: Introduction

Chapter 2: Maestro Overview, explains the operation modes of the Maestro.

Maestro Hardware and Software Limits Handling. Explanation of the function block behavior when queued.

Chapter 3: Maestro initial connection for both XP and Window 7.

Chapter 4: Motion and Administrative Function Blocks, describes and details the various single and multiple axes.

Chapter 5: PVT Motion explained with its applicable functions.

Chapter 6: API Services and Operations, describes the main general function blocks referring to the following:

- Main configuration variables
- Data Recording. Refer to section 10.5 Data Recording Functions for further details.
- Resource file uploading and downloading
- Download of new firmware version

Chapter 7: EtherCAT Process Image (PI) explanation and functions

Chapter 8: Data Recording. This allows the user to record internal controller variables, store them in local a temporary array, and upload them to a host computer using either one of the controller's communication channels.

Chapter 9: Bulk Parameters Reading to perform reading of parameters from multiple drives with their relevant functions.

Chapter 10: API Events, including the mechanism to handle events in the Maestro.

Chapter 11: Error Correction Mechanism, describes the mechanism to correct drive position errors, and the functions which are used to apply the correction.

Chapter 12: Maestro Hardware and Software Limits Handling description and details

Chapter 13: Saving Maestro User Program Parameters using XML scripts

Chapter 14: Connectivity and Configuration contains all the Network, Modbus (Host), CANbus (drive), EtherCAT (drive), Interpreter Command, and EtherNETIP communications to the Maestro server.

Chapter 15: Error handling. All Maestro and Servo Drive errors, warnings lists by ID code with possible reasons and recommendations.

Chapter 16: Programming in C++ with equivalent functions based on the C functions. These are wrapper functions using similar parameter details as their similarly named C functions.

Chapter 17: IEC 61131-3 Special Functions unique to IEC using simple API functions

Chapter 18: Appendix; includes explanation of the Axis Parameters and future supported functions.

Each chapter describes function blocks and their parameters according to the API source files described in **Chapter 1: Maestro Overview**. Some chapters have specific parameters that are applicable throughout a section and are therefore explained prior to the function block listings for that section. For example, the chapter **5.4 Axis Status** contains definitions of the **Axis Status Bit Masks**, whose variables are used as enumerator values in most function blocks. Certain parameters only apply within a specific function block and their details are recorded after the definitions of that function block. For example, the explanation of **Homing Functions** is detailed after the function block **5.8.2 MMC_Home**.

1.1.1. MMC_Halt

Call this function to command a controlled motion stop for a specific Axis.

```
int MMC_HaltCmd(
        IN MMC_CONNECT_HNDL hConn,
        IN MMC_AXIS_REF_HNDL hAxisRef,
        IN MMC_HALT_IN* pInParam,
        OUT MMC_HALT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

Function Parameters

hConn

Connection handle returned by Init Connection command

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the *MMC_HALT_IN* input structure that receives the HALT Information

pOutParam

Points to the *MMC_HALT_OUT* output structure receiving information as a result of calling the HALT function.

Remarks

The axis is moved to the state Discrete Motion, until the velocity is zero. With the output set to Done, the state is transferred to StandStill.

MMC_Halt is used to stop the axis under normal operation conditions. In non-buffered mode, it is possible to set another motion command during deceleration of the axis, which aborts the MMC_Halt and executes immediately.

If this command is active, the next command can be issued, e.g. a driverless vehicle detects an obstacle and needs to stop. When MMC_Halt is issued, then, before the standstill is reached the obstacle is removed and the motion can be continued by setting another motion command, so that the vehicle continues its motion and does not stop.

Scope

MMC_Halt will only operate from Standstill or Continuous Motion. It will not operate from Stopping, Homing, ErrorStop, or Disabled. Refer to the State diagram in *Figure 6*.

Figure 1-1: Function block layout example

Each function block (**Figure 1-1)** begins a section with its title and a short explanation of the function. The **Description** explains the usage of the function block with the **Scope** describing the conditions for its usage. **Motion Mode** defines whether the function block is supported in NC or Distributed (Non-NC) mode.

NOTE: **Links are highlighted in blue bold.**

Source defines the file source of the function block, with the **Function Parameters** describing the main parameters of the function itself. The logical definition must be retained in order for C to read the parameters of the function block correctly.

Remarks describe the function and its usage in detail, with their respective.

MMC_HALT_IN Structure

```
typedef struct{
        float fDeceleration;
        float fJerk;
        MC_BUFFERED_MODE_ENUM eBufferMode;
        unsigned char ucExecute;
}MMC_HALT_IN;
```

Parameters

fDeceleration

Float value of the deceleration (decreasing energy of the motor). Any positive value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eBufferMode

MC_BufferMode defines the behavior of the axis. Enumerator modes are as follows, but only the Aborting Mode is suported:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

Buffered mode only applies to Distributed and not NC axis. To use buffered for NC, requires user interface programming.

ucExecute

Start the execution command (Relevant only for future IEC or PLC programming). Boolean TRUE/FALSE values.

MMC_HALT_OUT Structure

```
typedef struct{
        unsigned int uiHndl;
        unsigned short usStatus;
        short usErrorID;
}MMC_HALT_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

usErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections ***3.3.2 G-MAS Error IDs***, and ***3.3.3 NC Profiler Error IDs on pages 3-35 – 3-39***. Displays an error code as -ve or +ve integers.

Figure 1-2: Function block layout example (cont.)

The input and output structures for each function(**Figure 1-2**) display the accepted structure for the input and output with their respective parameters. The parameters and sub-parameters and listed on the left side with their various descriptions opposite, and any references to other sections. The description describes the usage of each parameter and sub-parameter. With sub-parameters, the description defines specific values or enumerator values for the parameters with their explanation and/or a reference to such.

Figure 1 describes the function block for MMC_Halt

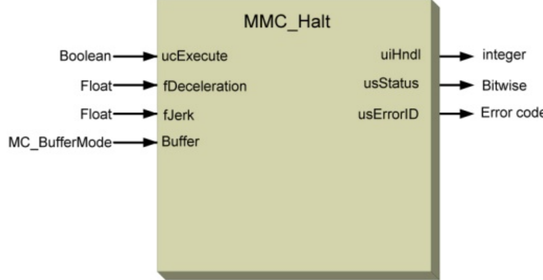


Figure 1: MMC_Halt function block

Function Block Code Example

```
int rc;
MMC_HALT_IN     stHalt_in;
MMC_HALT_OUT    stHalt_out;
//
// Inserting the structure parameters:
stHalt_in.fDeceleration       = 100000.0;          // Value of the acceleration
stHalt_in.fJerk                       = 20000.0;          // Value of the Jerk
stHalt_in.eBufferMode         = MC_ABORTING_MODE;  // MC_BufferMode Defines the behavior of the
axis
stHalt_in.ucExecute           = 1;
//
rc = MMC_HaltCmd (hConn, iAxisRef, &stHalt_in, &stHalt_out);
if (rc != 0)
{
    HandleError();
}
```

Implementation Example

The example below shows the behavior of MMC_Halt in combination with MMC_MoveVelocity.

1. A rotating axis is ramped down with MMC_Halt.
2. Another motion command overrides the MMC_Halt command. MMC_Halt allows this, in contrast to MMC_Stop. The axis can accelerate again without reaching standstill.

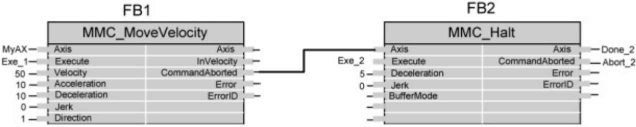


Figure 2: Combination of two blocks for MMC_Halt – Example

Figure 1-3 diagrammatic explanation and example of a function block

At the end of each function block detailed description (**Figure 1-3)**, is a diagram showing its inputs/outputs. Further information is provided with a real program C code, and implementation, examples of its usage.

Chapter 2: Maestro Overview

The Maestro general communication and API architecture is described below in Figure 2-1.

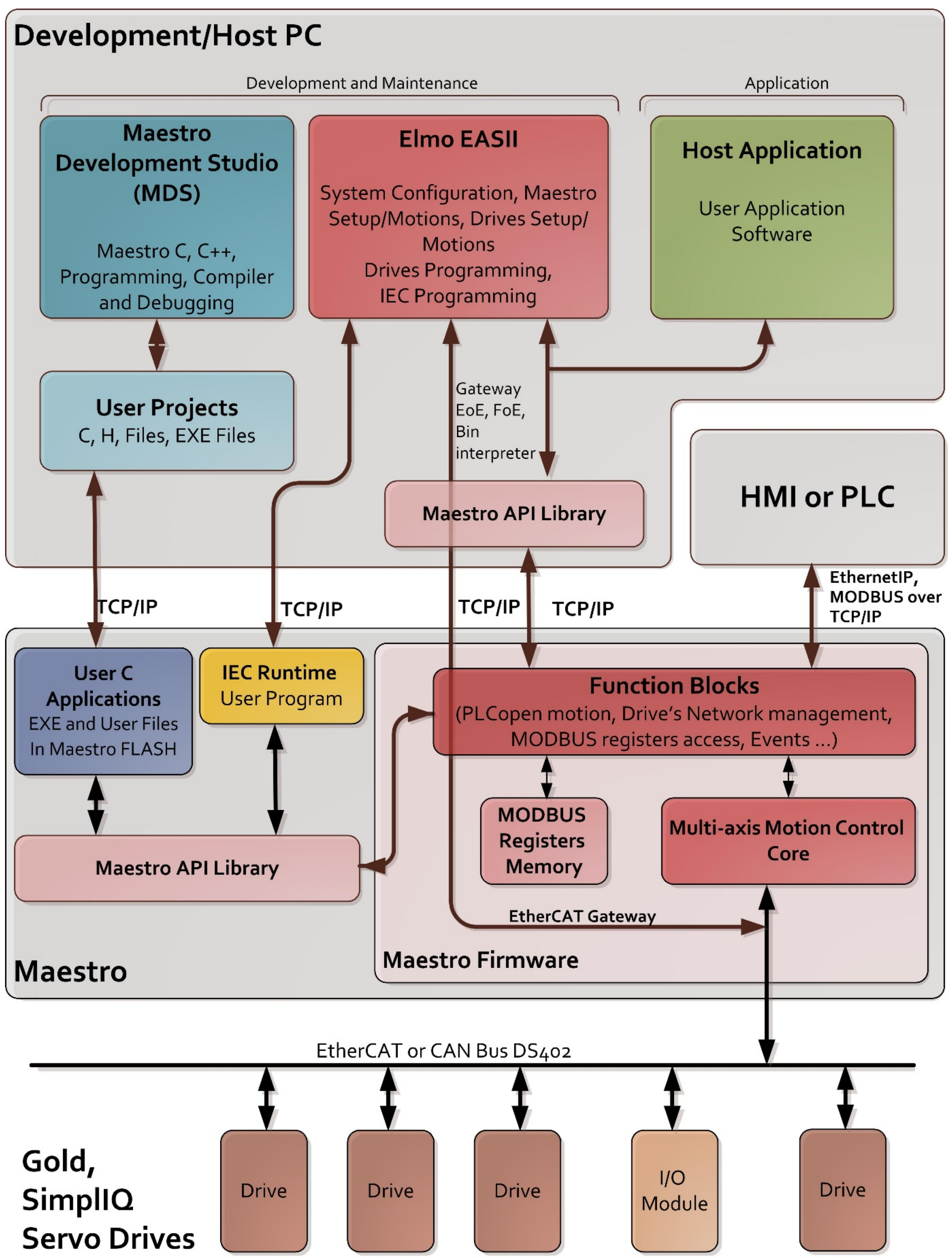


Figure 2-1: GoS System Software Structure – Development and Application

The Maestro supports the following programing interfaces:

Operation System	Library
Hosts (TCP/IP)	.NET Library Win32 Library – C and C++ Libraries
Internally (IPC)	C and C++ Libraries IEC 61131-3
VxWorks Host	.NET Library Win32 Library – C and C++ Libraries

The Maestro API implements direct binary communication interface using the TCP/IP and internal connection shown in **Figure 2-1**. Using the direct binary API is a faster and more efficient connection method to produce best system performances. The underlying Motion API is the PLCopen (the API server), and Elmo's Software API's above it, which export the same functions in both cases. An identical API library is used. Elmo provides the dll/lib for Win32 based environments and a library for interfacing the Maestro from within the Maestro, when writing user programs. The only difference is the initial "Initialization" method. The API library accesses the same API server to perform the desired user operations.

The API defines a set of function blocks, with the following attributes:

Attribute	Explanation
Simplicity	Ease of use, towards the application program builder and installation & maintenance
Efficiency	In the number of function blocks, directed to efficiency in design and understanding
Consistency	Conforming to IEC 61131- 3 standard The IEC function blocks reflect the actual function block's as they appear in the IEC window of the Elmo Application Studio (EAS) application. Therefore the 'C' function output parameters by the name of **usStatus** includes Error - a 1 bitwise parameter, Done, etc., when compared with the same IEC function block, the IEC version should display and include all relevant bits (error, done, busy, etc…). **Elmo's IEC 61131- 3 function blocks and functions have all array parameters on the input side whether or not it functions as input or output parameter, and immaterial whether the C function equivalent has the same parameters as input or output.** **It should be noted that while every effort is made to make sure that all C functions conform to the outputs of IEC functions, in practice, this is not always practical due to the nature of the IEC functions.**
Universality	Hardware independent
Flexibility	Future extensions / range of application

These function blocks are sectioned according to their motion axes, and communication protocols. The API therefore consists of a series of grouped source files divided by the following subjects:

General	Source: GMAS\includes\MMC_general_API.h Includes all main configuration and firmware download function blocks.
Main Definitions	Source: GMAS\includes\MMC_definitions.h Includes all main and basic definitions for the function blocks.
Single Axis Motion	Source: GMAS\includes\MMC_PLCopen_single_API.h Includes administrative and motion function blocks involved in the single axis motion.
Group Axes Motion	Source: GMAS\includes\MMC_PLCopen_group_API.h Includes administrative and motion function blocks involved in multi-axes motion.
Error Correction Mechanism	Source: GMAS\includes\ Includes functions for 1-D, 2-D, and 3-D corrections.
Bulk Parameters Reading	Source: GMAS\includes\ Includes functions to read multiple parameters from multiple axis at the same instant.
Network Communications	Source: GMAS\includes\MMC_network_API.h Includes all basic network functions blocks necessary to communicate with the Maestro Network Motion Controller.
Modbus Communications	Source: GMAS\includes\MMC_host_comm_API.h Includes all Modbus function blocks necessary to communicate with the Maestro Network Motion Controller.
Drive Communications	Source: GMAS\includes\MMC_drive_comm_API.h Includes all CANbus, EtherCAT function blocks, and Interpreter Command functions necessary to communicate with the Maestro Network Motion Controller.
EtherCAT Communications	Source: GMAS\includes\MMC_ECATIO_API.h Includes major EtherCAT communication functions for analog and digital I/Os.
DS-401 Communications	Source: GMAS\includes\MMC_DS401_API.h Includes major DS-401 communication functions for DI and DO intended for I/O modules.
Error Correction	Source: GMAS\includes\MMC_ErrorCorr_API.h Includes error correction functions for 1D, 2D and 3D modes.
API Events	Source: GMAS\includes\MMC_events_API.h Includes function blocks that read and write events to and from the Maestro.
C++ Functions	Source: GMAS\includes\CPP\MMCXXXXXXX.h Includes all C++ class function mirroring functions and function blocks

described in detail for C programming

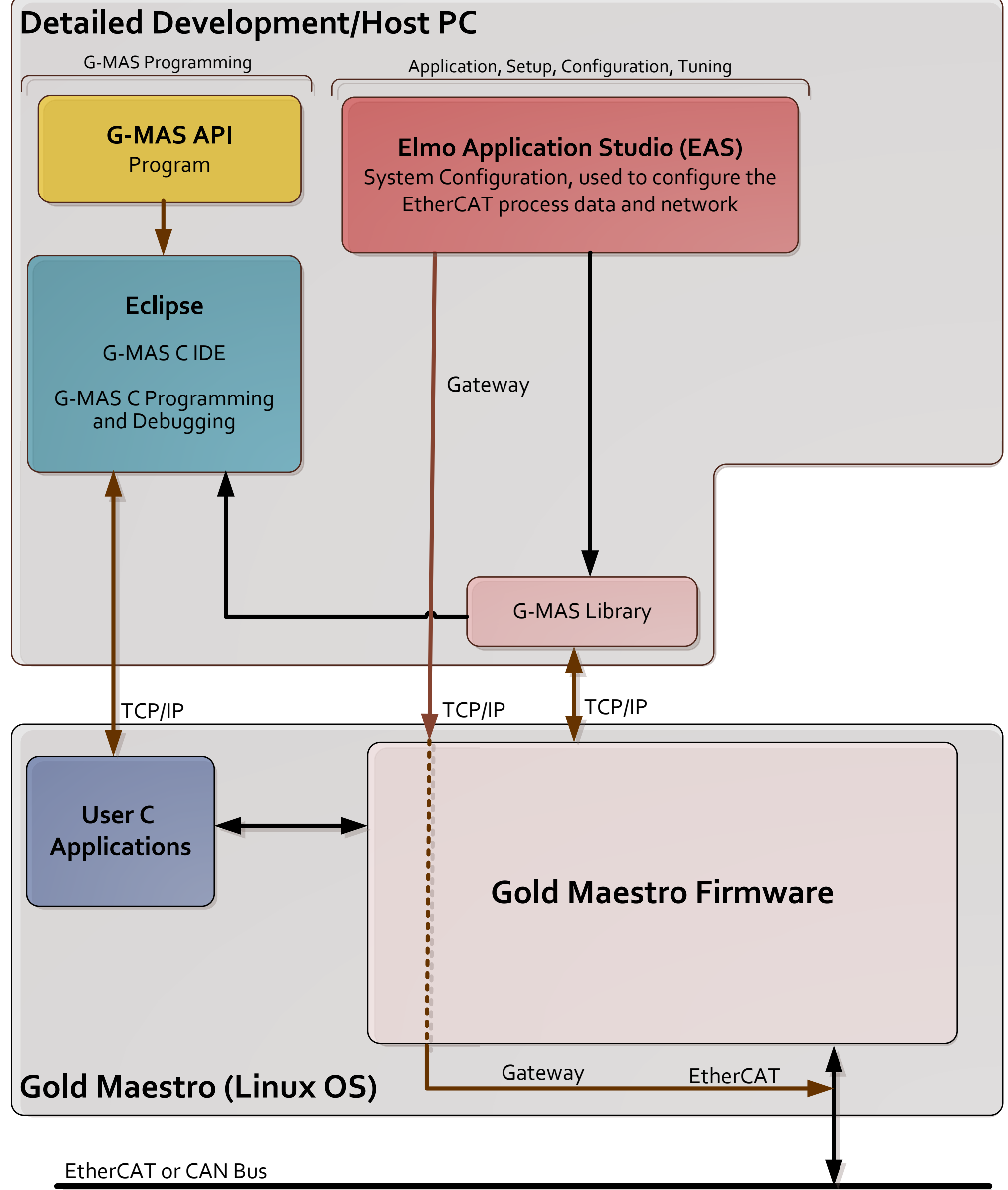


Figure 2-2: GoS System Software Structure –Host PC Development

In addition, since the Maestro operates as a master, independent of any host system, in operational mode, it periodically sends data to the slaves, that may override the data that a user sends from a host system. Therefore, for example, the user cannot tune an axis if the axis is in operational mode. To prevent this and allow the Elmo Application Studio (EAS) application (Figure 2-2) to operate via the Maestro CANbus and EtherCAT, specific API functions are called to change the Maestro operation and allow these applications to function.

The following table lists the function blocks called as part of a special API to allow the EAS application to function and revert the Maestro to operational mode when their operation is completed:

Application	Function Blocks
EAS	MMC_ChangeToPreOPMode MMC_ChangeToOperationMode GetGMASOperationMode MMC_EnableEthercatConfigMode MMC_DisableEthercatConfigMode MMC_IsEthercatConfigMode

Important: **It should be noted that connecting to the Maestro is only allowed using one user application at a time. Connecting two applications to the Maestro in parallel may cause serious problems to the Maestro library. When performing multiple IPC connections to the Maestro, the multiple connections must be opened from the same user application.**

2.1 Using EASII Application

For the EAS application to monitor and perform motions, the Maestro cannot operate in the background. The special API function *MMC_ChangeToPreOPMode* changes the EtherCAT and CANbus communication from the Maestro to Pre-Operation mode, causing the following:

Communication	Operation
EtherCAT	**No process cycle operates**
CANbus	**No outputs via CAN, and no state machines run**

These API functions change the Maestro mode so that the Maestro operation is transparent and no messages transfer between the Maestro and the drives.

In order to configure the EtherCAT network (EtherCAT Configuration Mode) via the EAS application, the Maestro must be set to EtherCAT Configuration mode. The user is then able to perform the operations. The API then employs the specific functions to change the Maestro back to operational mode.

2.2 Maestro Operation Modes

To optimize the device network usage, the Maestro supports two modes of operating axes present on the Device Network:

- NC Axes – for Numeric Control Axes
- Distributed – for axes not under strict numeric control

The main difference between these modes is the way the motion profile is calculated, and as a result, the synchronization level achieved.

In general, for axes not requiring low level (network) motion synchronization, the Distributed mode should be used, allowing the servo drives to generate their own motion trajectory, thus reducing network load. In this case, synchronized motions like ECAM, based on an external master encoder can still be executed. For highly synchronized motions, generated by the Master controller (referred to under the PLCopen definitions as group vector motions), the NC mode should be used.

2.2.1 NC Motions

In this mode, the Maestro controls the motion, handling the axis (and motion) State (as defined by the PLCopen Standard), and calculating the motion profile as part of its real-time loop process (NC Cycle). Servo drives operating with a Maestro master under this mode will run under the DS-402 motion modes e.g.; Interpolated position, or one of the Cyclic Sync modes (Position/Velocity).

2.2.2 Distributed / Standard DS-402 (stand-alone) Drive

In this mode, the Maestro uses the servo drives own DS-402 operation modes, where the drive itself controls its own profiling as part of its Real Time process. The Maestro only synchronizes start/stop and general activation functions, but is not responsible to the low-level real-time profile generation.

The Maestro can mix NC and Distributed axes in the same network configuration, thus optimizing usage of network and processor resources. The definition of the axis type (NC or Distributed), can be changed during operation using the *ChangeOpMode* (operation mode) command.

2.2.3 Maestro Axes and Node Definitions

The Maestro controls the following axis types:

- Single Axis as NC axis
- Single Axis as Distributed axis
- Group of axes, as NC axes (only). A group is a collection of axes, which can execute spatial vector motions

In the Maestro architecture, all axes names have to be defined in advance, in the system resource file, a dedicated (XML format) file, which defines the following:

- Number of active axes in the system
- For each axis, whether it is operating in NC or Distributed mode.
- Groups of axis must be predefined by the user. However, the link of actual axes to groups can be performed in run-time (using specific API functions, e.g. AddAxisToGroup() and RemoveAxisFromGroup().

- Basic Maestro network cycle (used for NC as well as Distributed), to access the axis position, commands etc.

For each of the above, the Maestro hold an internal software object Node. Currently, two types of nodes are defined:

- Single Axis Nodes: NC or Distributed.
- Group (Vector) Nodes: NC only, and can be Group Single axis (NC) only.

Additional Nodes types are supported by the Maestro, such as DS-401 IO, DS-301, and DS-406 modules.

The Max Axes and Node numbers and their combinations are defined as follows:

Tc	The Minimal Time of the Master NC Cycle. In EtherCAT communication, Tc defines the Minimal Distributed Clock Cycle Time of the system. Its value is currently defined as 1 msec (1000 µs), and the function is defined as: Tc=(1+N)x250 µsec (where : 1 <= N <= 39). In CAN Bus communication, Tc defines the Sync Time of the CAN network. Its value is currently defined as 3 msec and the function is defined as: Tc=Nx1 msec (where 1 <= N <= 100).
N	The max number of Single Axis Nodes in the system. Currently limited to 64 axes (NC and Distributed, altogether).
V	The max number of Group/Vector nodes that can be simultaneously defined in the system. Currently limited to 16 axes (this is in addition to the 64 single axis nodes).
Va	The max number of Group/Vector nodes that can be simultaneously running in the system. Currently limited to 6 Group/Vectors.
Vn	The max number of physical axes that can be simultaneously linked to a specific Group/Vector node. Currently limited to 16 physical axes.
Mc	The max number of devices that can be accessed in a single Master Cycle, via the Communication link. This number depends if CAN or EtherCAT communications are used. This number also depends on the current selected Tc.
Mp	The max number of NC axes that can be handled in a single Master Cycle. This includes both Single axis, as well Vector/Group nodes. Generally speaking, Mc and Mp can be different numbers. Currently, they are equal and limited to 20.

In order to reduce the Maestro cycle computations, each axis can define an axis period and an axis offset time that is related of course to the Tc base time.

For example: A typical NC system nodes/axes distribution may have:

- 8 Physical axes (nodes: a1 ÷ a8) – all NC.
- 1 group (node v1, linked to physical axes a6, a7, a8).
- Group v1 is running on each master cycle (Tc), calculating a spatial (vector) profiled motion (in 3D space), and the actual projections to the physical axes linked to it are a6, a7, a8.
- The physical axes nodes are running together as follows:
    - a1, a2 are running together, each 2×Tc cycles.
    - a3, a4 are running together, each 2×Tc cycles.
    - a5 is running each 4×Tc cycles.

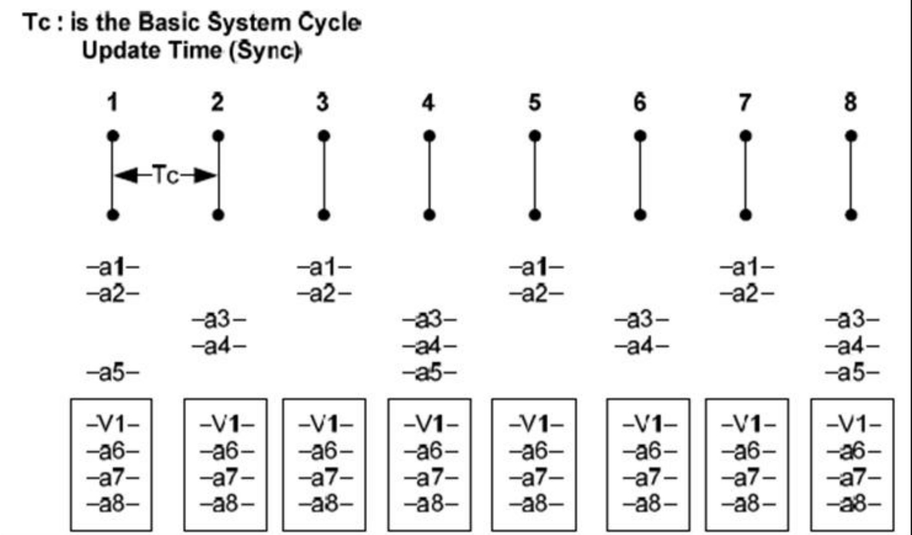


Figure 2-3: Typical NC Configuration

In this example, the Max Mp is 6 (although we have 8 axes). The cycle is repeating itself once every 4xTc cycles.

2.2.4 Maestro to Servo Drive Interfaces

The Maestro manages all motion commands sent to the servo drives, via the CANopen DS-402 standard. This is relevant to the Maestro CAN hardware interface, and to the EtherCAT protocol implementing CoE (CAN Over EtherCAT).

For axes (Nodes) that operate in NC mode, Maestro uses the DS-402 motion modes: Interpolated position, or one of the Cyclic Sync modes (Position/Vel).

For axes (Nodes) operating in Distributed mode, where the servo drive manages its own profiler and real-time motion execution, it is assumed that the servo drive supports the relevant requested motion modes.

Motion Modes that are part of the PLCMotion API definition, but are NOT supported by the DS-402 interface, will not be available in standard DS-402 servo drives when working in Distributed mode (unless specific Vendor Types objects are defined, e.g. ECAM in drive level, etc. as implemented for example in Elmo servo drives).

Note: **Although the above describes and relates to Elmo DS-402 compatible servo drives, the Maestro design does not limit the operation of any DS-402 compatible servo drives as well.**

Chapter 3: Maestro Hardware Connections

3.1 Maestro USB Connection

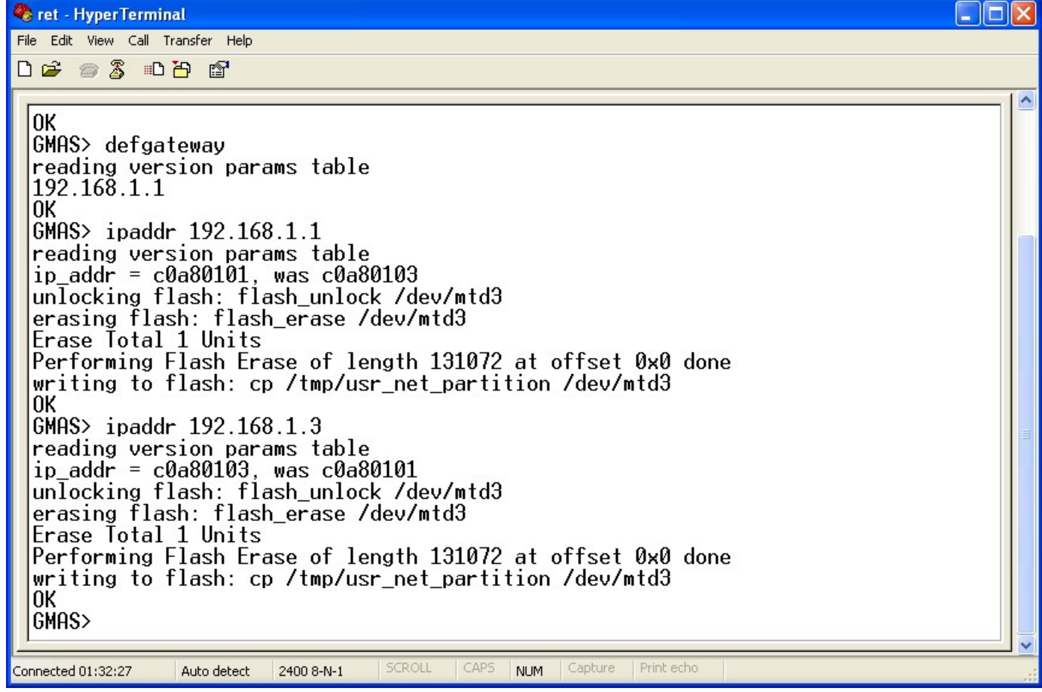
This section describes how to communicate with the Maestro when connected via a USB connection from a host system to allow the GMAS ports to be configured and operate via the LAN. This connection imitates a COM port connection at a COM Port. Therefore, if an RS232 Terminal connection is opened at the host system, *udev* execution opens a terminal from which it is possible to perform the following basic operations.

- Change / Read IP Address.
- Change / Read Gateway.
- Change / Read Subnet mask.
- Change / Read Server IP.
- Change / Read download version path.

3.1.1 Connection Procedure

The following procedure describes how to connect the host system to the Maestro via USB connection and configure the communication parameters of the Maestro.

1. Make sure that the Maestro is powered on.
2. Connect the USB connection from the host system to the Maestro. The Maestro should emit a sound signifying that a connection is made.
3. Open the Device Manager and locate the Ports section in the hierarchal structure. Verify which COM port is defined for the Elmo GMAS.
4. At the host computer, open a communications Terminal to a COM port.
   The prompt should display *GMAS>*.



5. At the prompt, enter any command detailed in sections **3.1.2** - **3.1.4** to perform the appropriate operation at the Maestro.
   For example; To request the IP address enter *ipaddr*.
   The Maestro IP address is returned.

Note: **By default, the Maestro IP address is set to 192.168.1.3. However, the customer may prefer to integrate the Maestro with his network system and therefore may wish to change the default value. Use this procedure to perform this action.**

3.1.2 ipaddr – GMAS IP Address

Purpose	1. Set a new IP address. 2. Request display of GMAS's IP address.			
Syntax	ipaddr			
Parameters	None or string			
Attributes				
	Parameter, string	Interpreter	N/A	N/A
Examples				
	Input	**Output**		
	ipaddr	*10.10.10.1*		
	Ipaddr 10.10.20.2	*OK*		

3.1.3 ipmask – GMAS IP Subnet Mask

Purpose	1. Set a new IP subnet mask. 2. Request display of GMAS's IP subnet mask.			
Syntax	ipmask			
Parameters	None or string			
Attributes				
	Parameter, string	Interpreter	N/A	N/A
Examples				
	Input	**Output**		
	ipmask	*255.255.255.0*		
	ipmask 255.255.255.0	*OK*		

3.1.4 defgateway – GMAS Default Gateway

Purpose	1. Set a new default gateway. 2. Request display of GMAS's default gateway.			
Syntax	defgateway			
Parameters	None or string			
Attributes				
	Parameter, string	Interpreter	N/A	N/A

Examples		
	Input	**Output**
	defgateway	*10.10.10.1*
	Defgateway 10.10.10.2	*OK*

3.2 Maestro Network Configuration

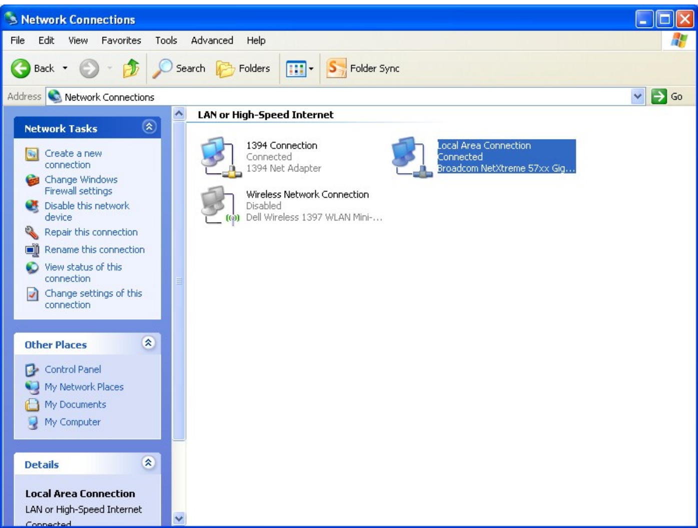
This section introduces the procedure to connect between the Maestro and the PC for both XP and Windows 7 operating systems. By default the Maestro is set with the following IP settings:

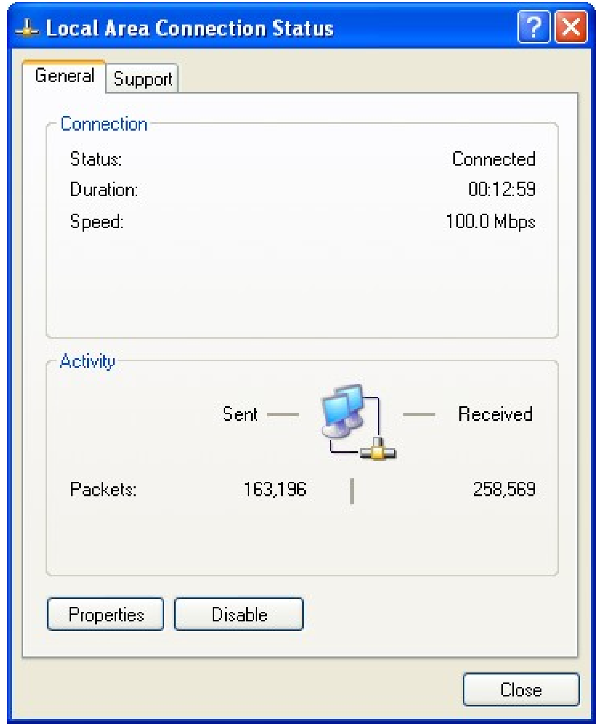
Setting	IP Address
IP address	192.168.1.3
Subnet mask	255.255.255.0
Default Gateway	192.168.1.1

3.2.1 XP Windows Setup

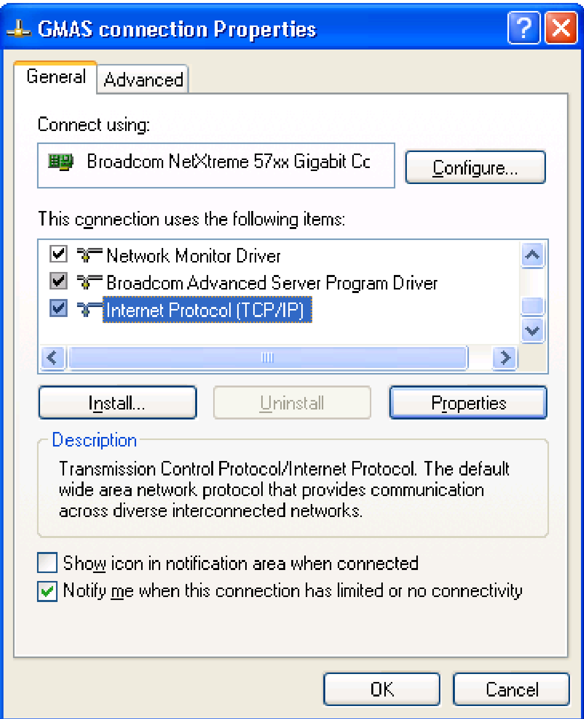
To set the PC configuration in XP Windows

1. Select Start-> Control Panel->Network Connections->Local area connection->Properties

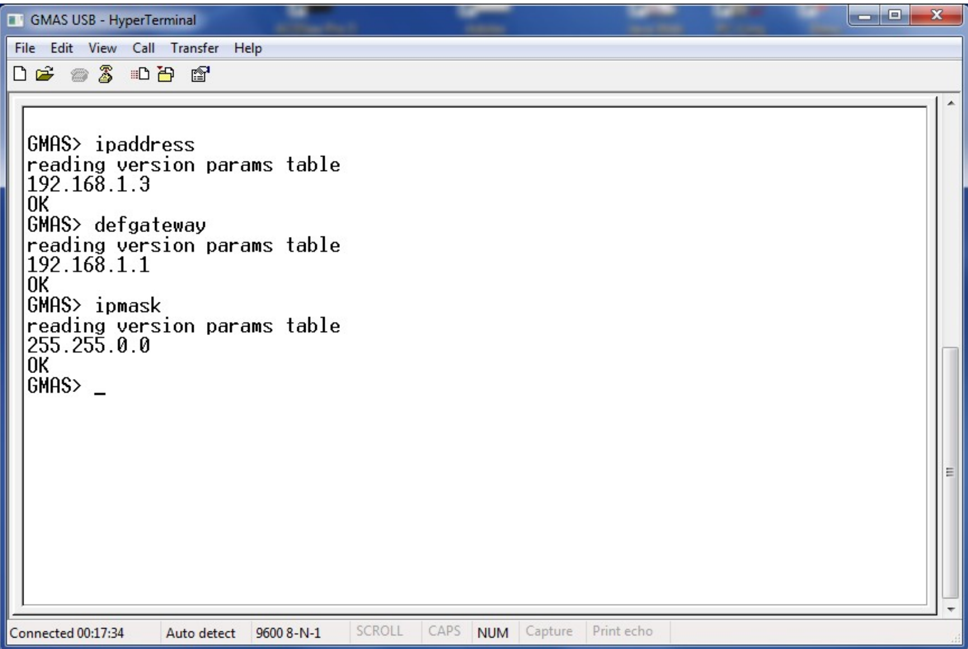




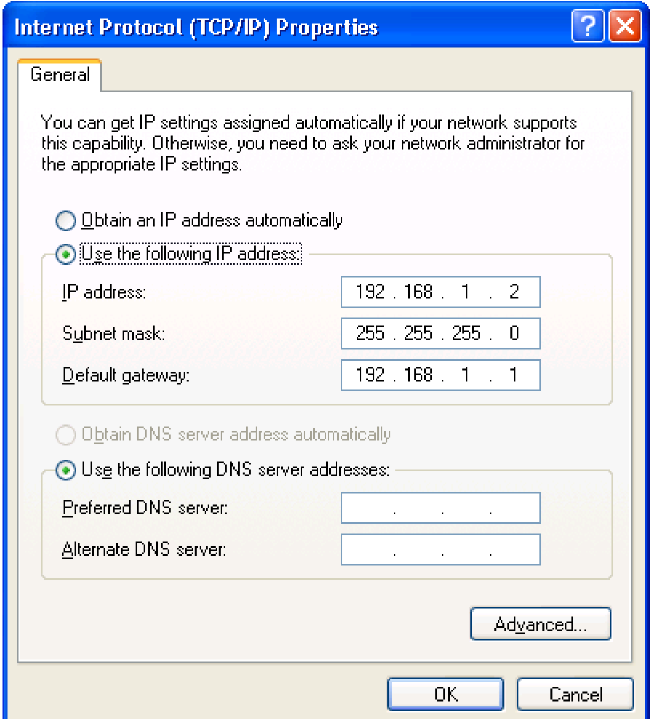
2. In the General tab select Internet Protocol(TCP/IP).

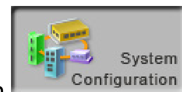


3. Perform the USB Connection procedure as described in the section above **3.1.1.**
4. From the terminal window, check the IP Address, Default Gateway, and Subnet Mask, of the Maestro.

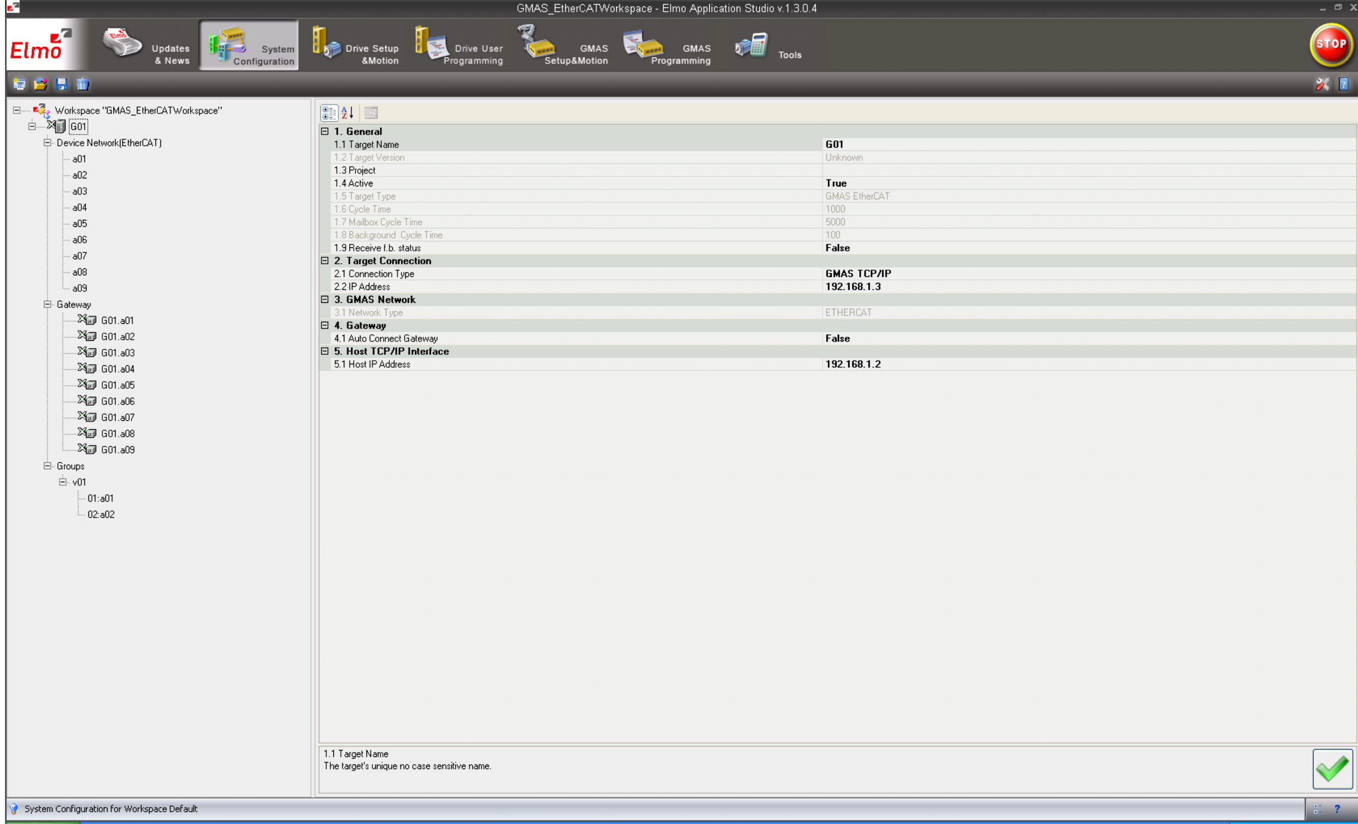


5. Click Properties and select the radio button *Use the following IP Address*.



6. Enter the IP Address, Default Gateway, and Subnet Mask from the terminal window.
7. Click **OK**.
8. Open the EAS application.
9. Click System Configuration 

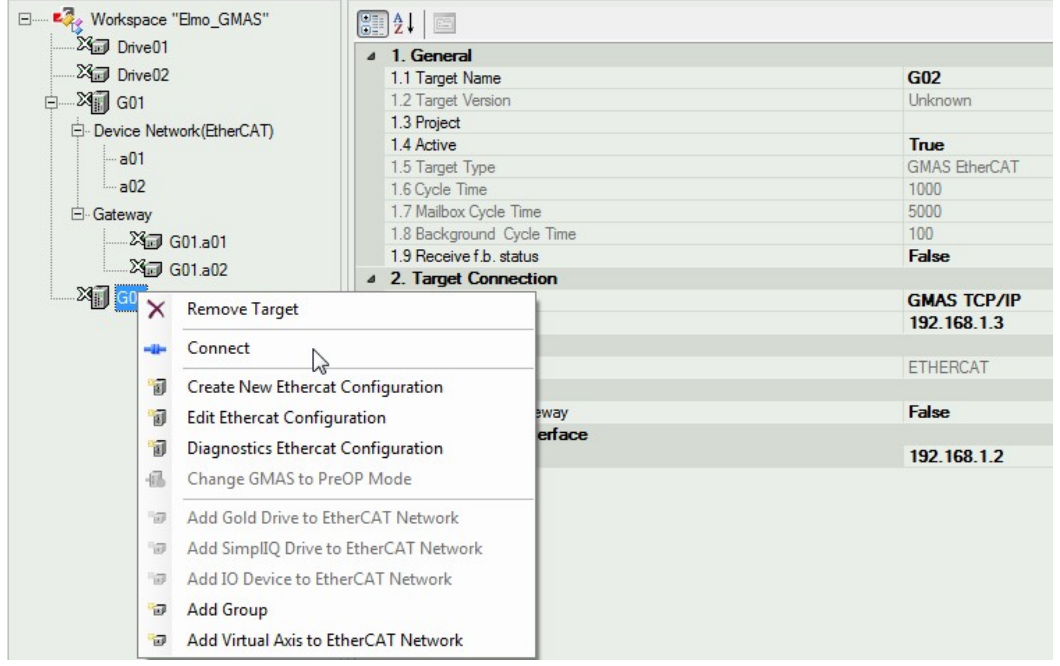
10. In the Connection type select **GMAS TCP/IP**.



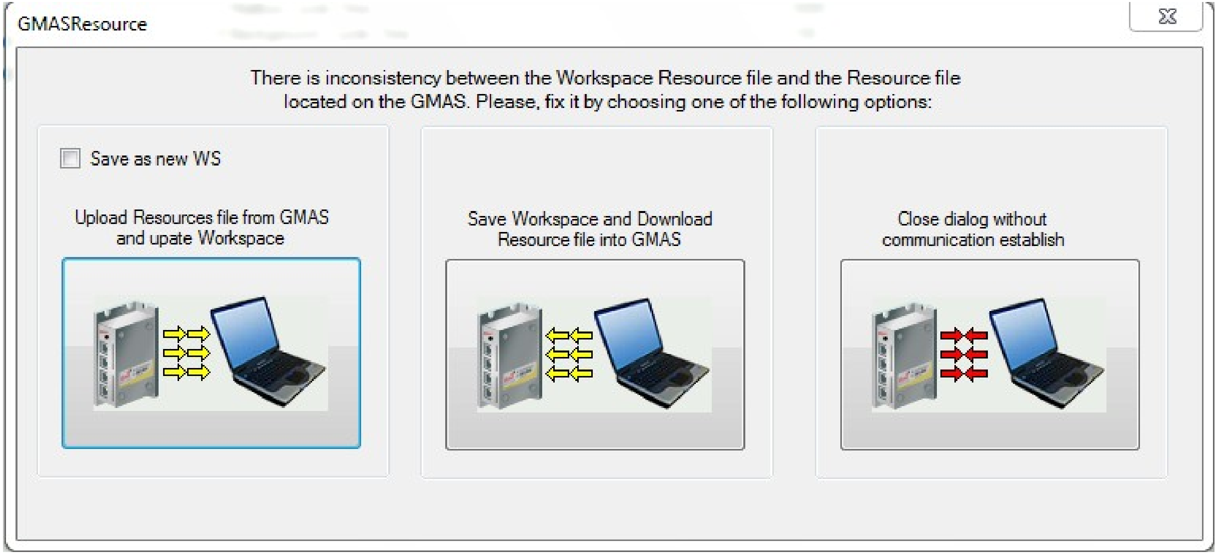
11. In **IP Address** select the same addresses entered to the Internet Protocol Properties window above.

12. Click **Apply** to save the drive's properties.

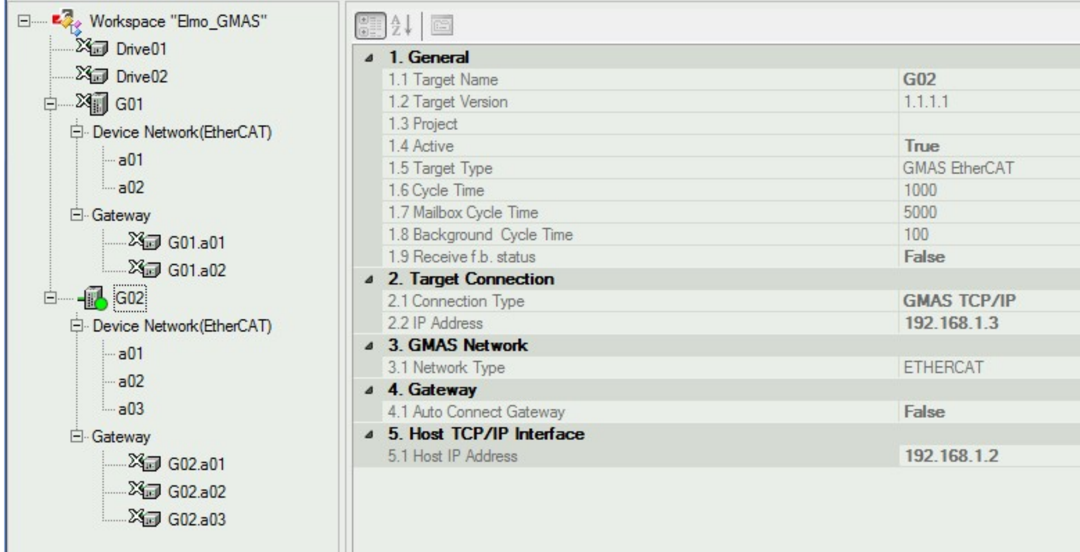
13. Right click the drive's name and click **Connect.** The GMASResource data window opens.



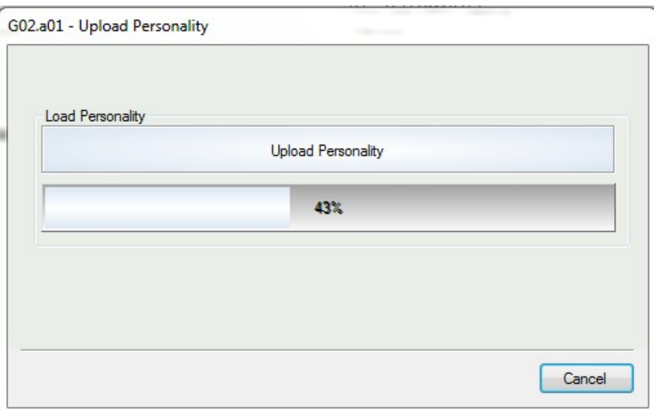
14. In the GMASResource, select *Upload Resource file from GMAS and update Workspace*. The Resource file is uploaded to the host system.



On completion the Maestro is connected (green marker), and drives connected to the Maestro appear.



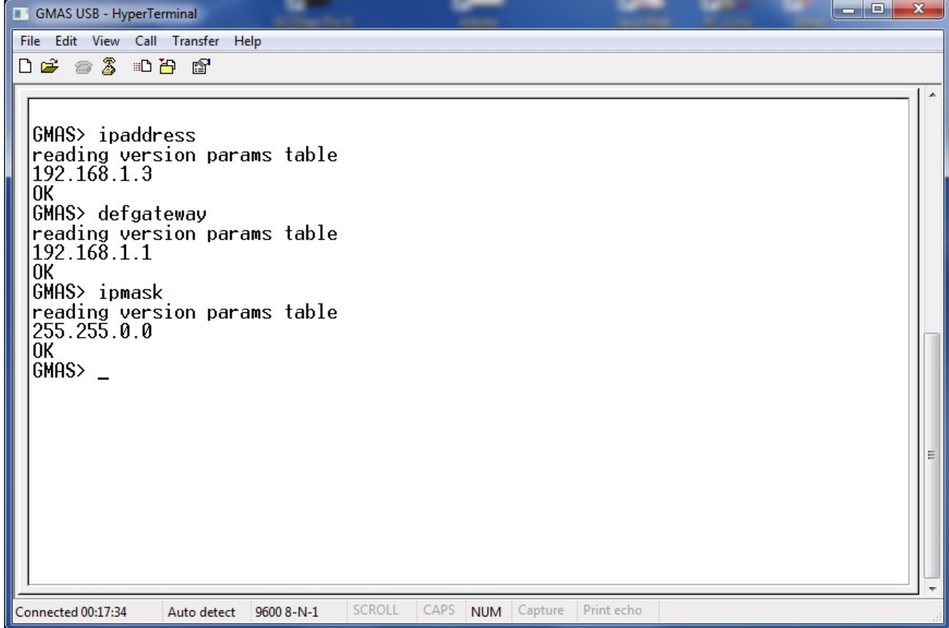
15. Select the first drive and right-click to choose **Connect**. The drive Personality is uploaded, and the drive is connected. Connect the other drives linked to the Maestro.



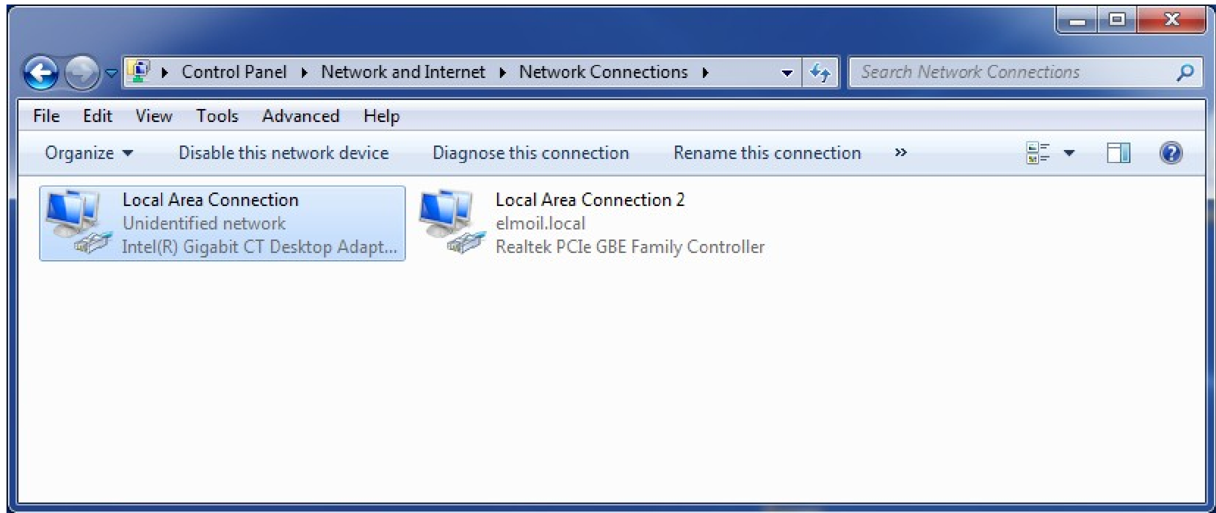
3.2.2 Windows 7 Seup

To set the PC configuration in Windows 7

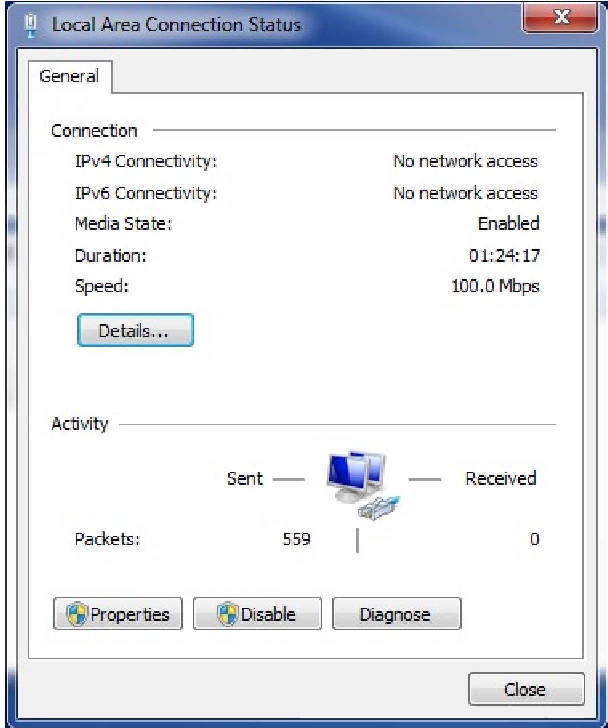
1. Connect the USB connection from the Host system to the Maestro.
2. Perform the USB Connection procedure as described in the section above 3.1.1.
3. From the terminal window, check the IP Address, Default Gateway, and Subnet Mask, of the Maestro.



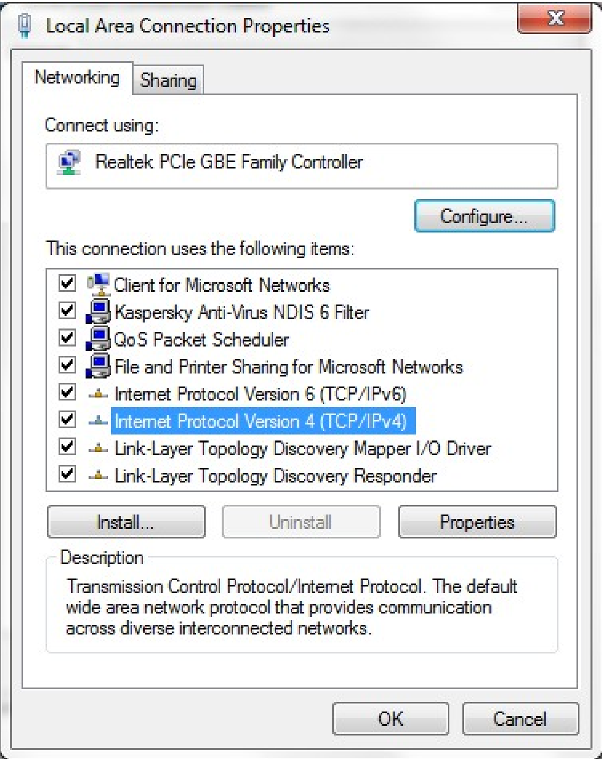
4. Open the Network Connection window, and locate the Local Area Connection to the Maestro.



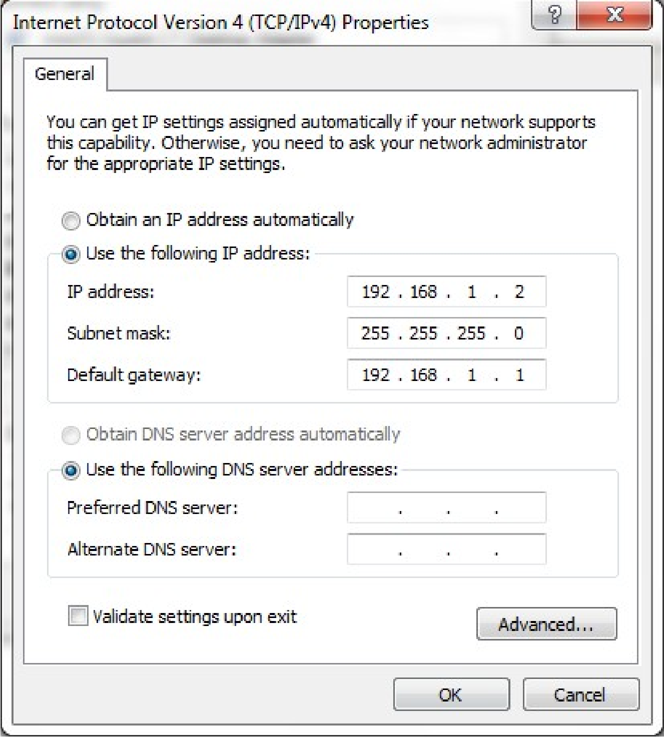
5. Right-click on the Connection, and select Status. Make sure that the IPv4 and IPv6 Connectivity show No network access.



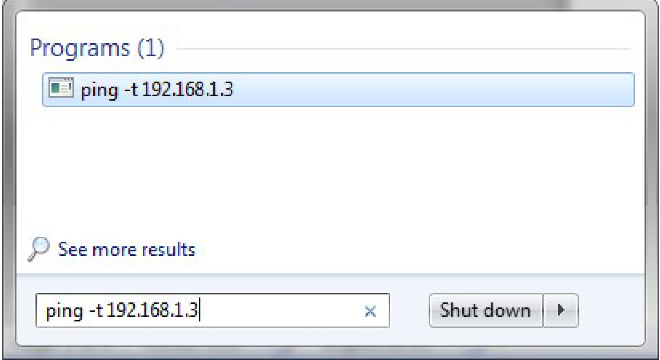
6. Click **Properties**, and select *Internet Protocol Version 4 (TCP/IPv4)*.



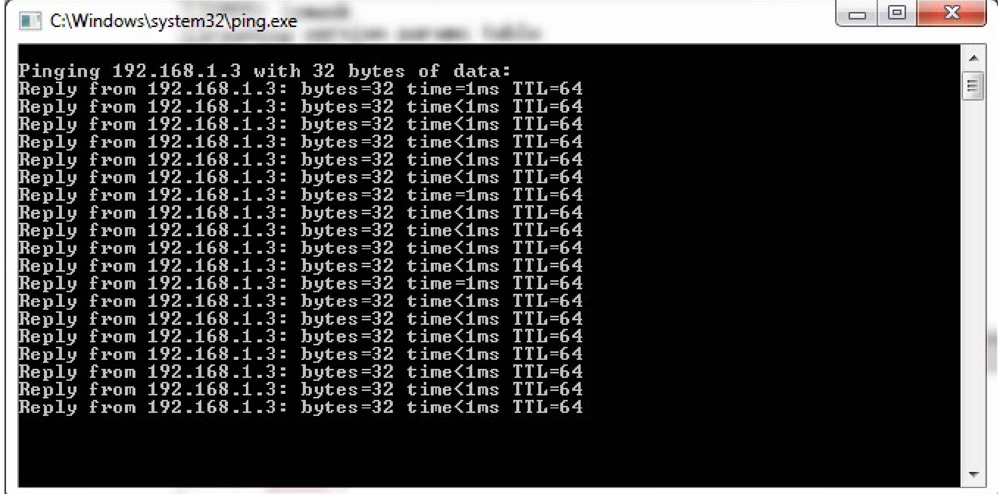
7. Select **Properties** and enter the Default Gateway, and Subnet Mask obtained from the Telnet window.



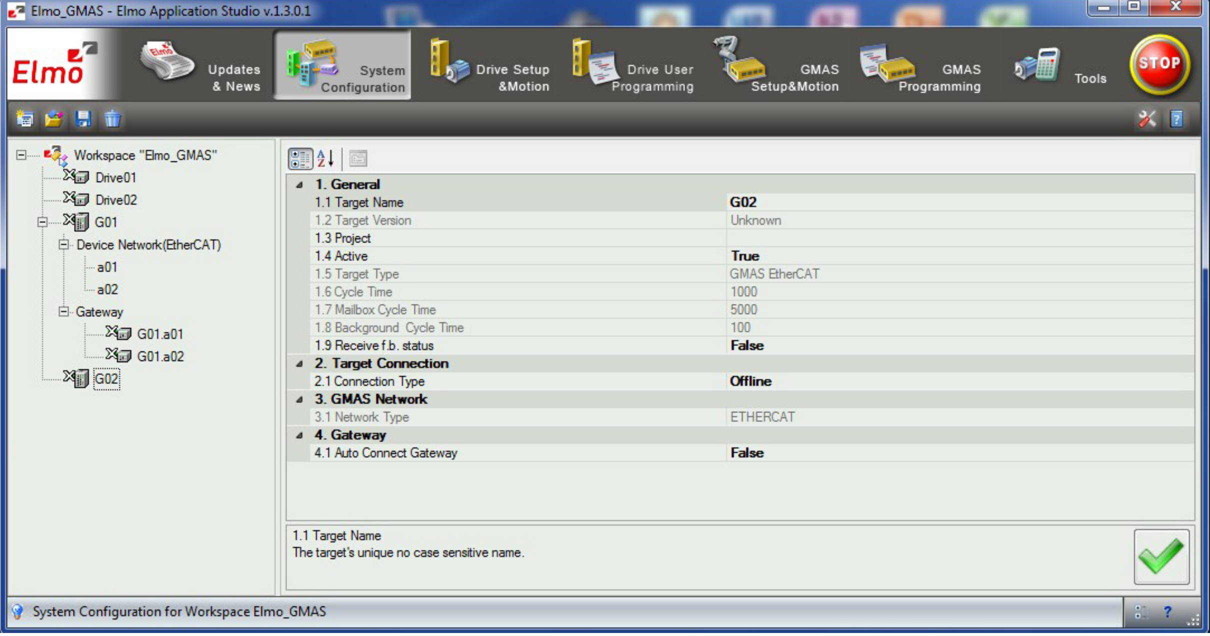
8. In the Internet Protocol Version 4 (TCP/IPv4 Properties window, Insert an **IP Address** similar to the Maestro own address but different at the fourth set of digits, as shown e.g. 192.168.1.2, and then click **OK**.
9. Check the connection to the Maestro by pinging it. Enter the following at the Windows prompt:
   Ping –t <Maestro IP Address> e.g.192.168.1.3



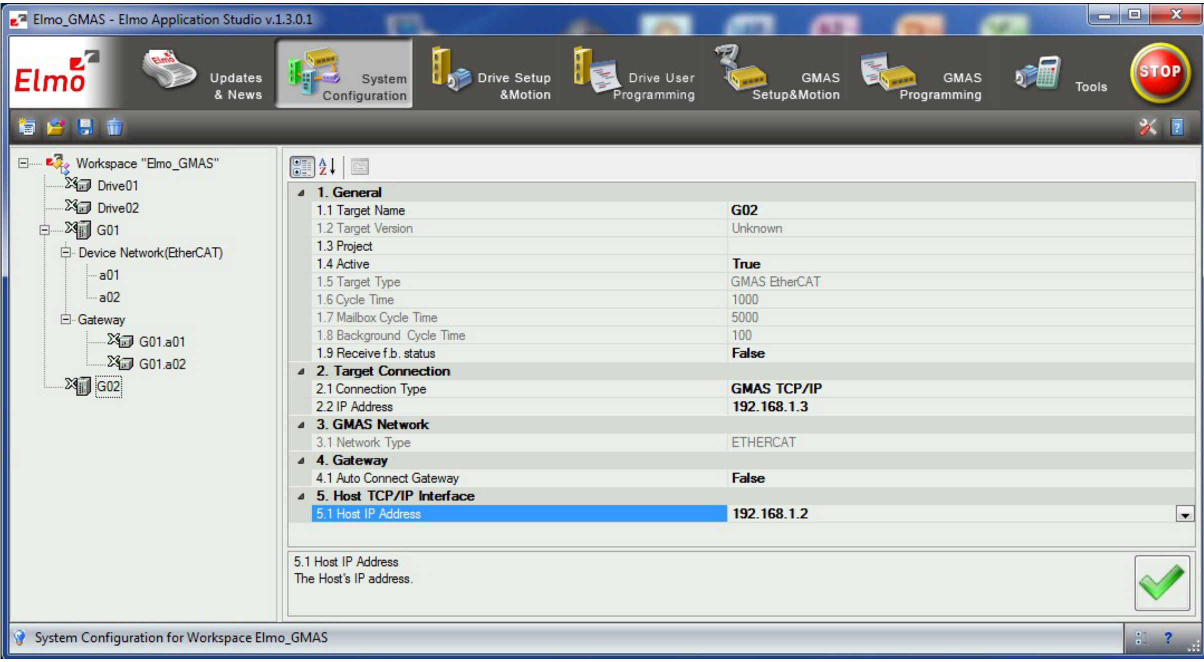
A Command Prompt should open with the reply results demonstrating a connection to the Maestro.



10. Open the EAS Application at the System Configuration window and right-click on the Workspace to setup a new Maestro.

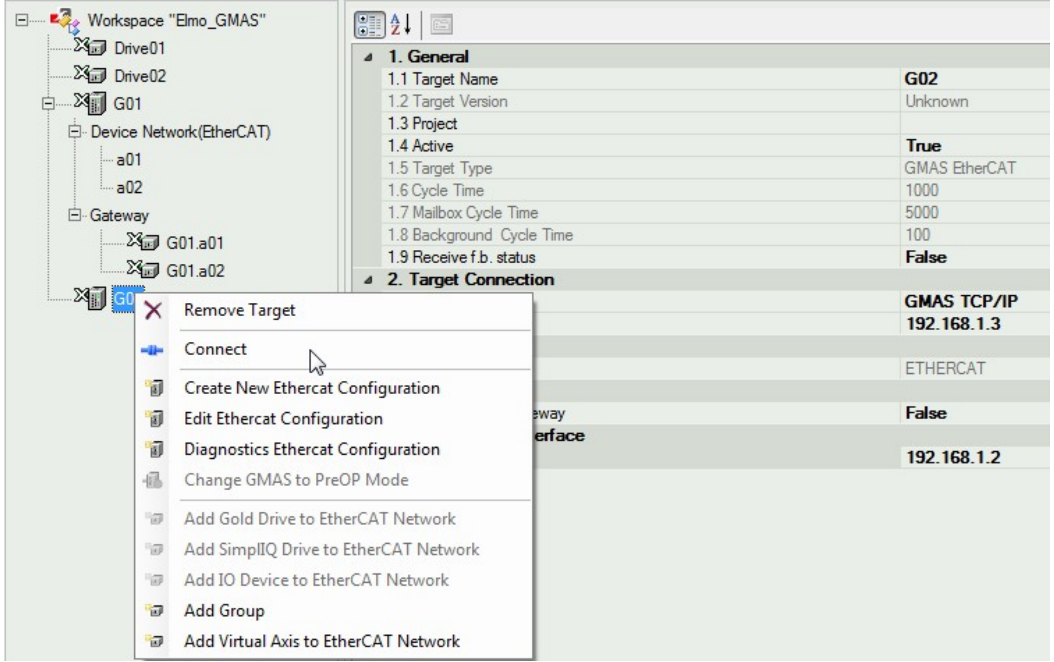


11. Setup the EtherCAT configuration of the Maestro with respective TCP/IP addresses setup in the Windows configuration and from the Terminal feedback, and click .

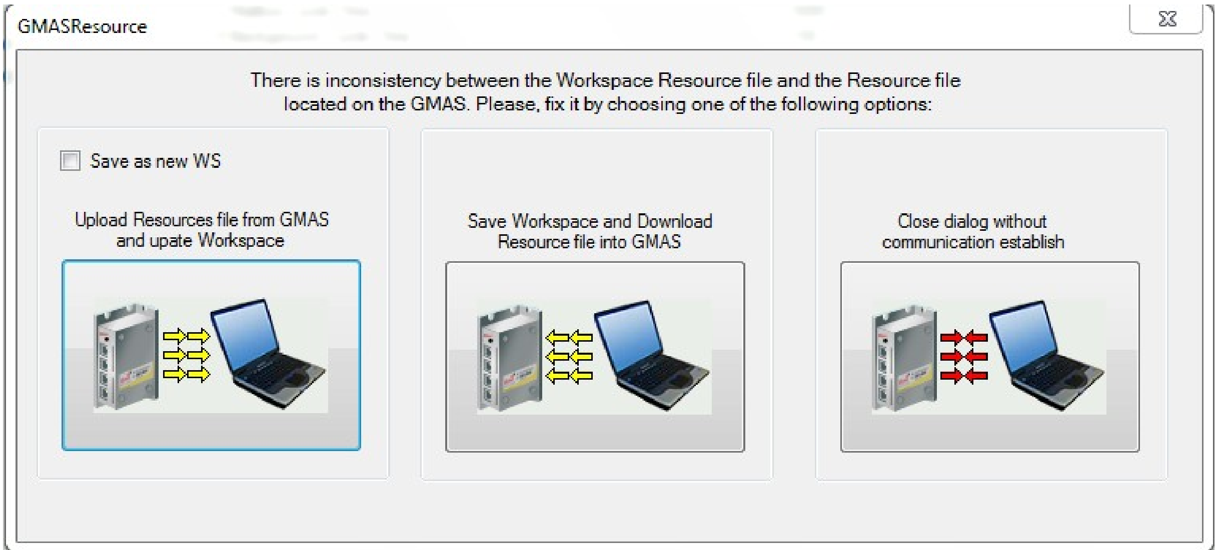


12. Click **Apply** to save the drive's properties.

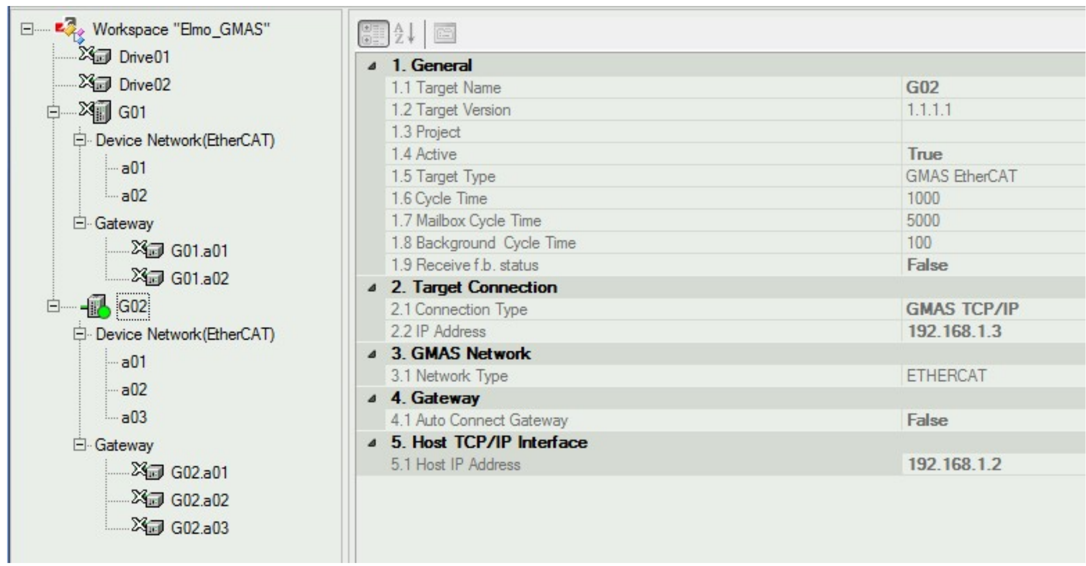
13. Right click the drive's name and click **Connect.** The GMASResource data window opens.



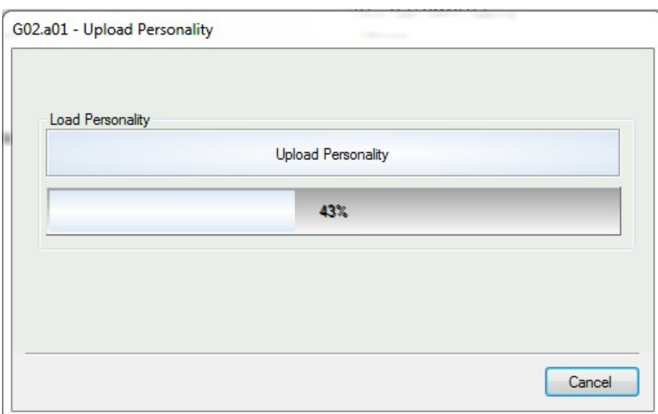
14. In the GMASResource, select *Upload Resource file from GMAS and update Workspace*. The Resource file is uploaded to the host system.



On completion the Maestro is connected (green marker), and drives connected to the Maestro appear.



15. Select the first drive and right-click to choose **Connect**. The drive Personality is uploaded, and the drive is connected. Connect the other drives linked to the Maestro.



3.3 Maestro Master –Slave Hot Plug

Previously when a slave is disconnected from the EtherCAT network, the Maestro performed a scan on the entire EtherCAT network to determine the topology. The active slaves which have not disconnected are scanned and their EEPROMs read. In the EASII EtherCAT Diagnostics window, when a Slave is diconnected from the Master, the Maestro EtherCAT master enters a mode called Wrong configuration and displays Default input values.

Wrong configuration	Physical configuration does not match the configuration created in the EtherCAT configurator.
Default input values	All inputs values are displayed as 0

When the configuration is "wrong" Maestro constantly checks the identity of the devices connected to the EtherCAT network by reading the EtherCAT slaves EEPROMs containing the following information:

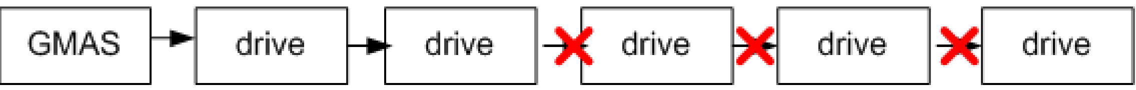
- Product code
- Vendor ID
- Serial number
- Revision number

The Maestro then compared this stored data in the EEPROM against the configuration stage data. The reading of the EEPROM content was performed for all slaves connected to the Maestro, and was performed periodically until the configuration is no longer "wrong" i.e. the disconnected slave is reconnected.

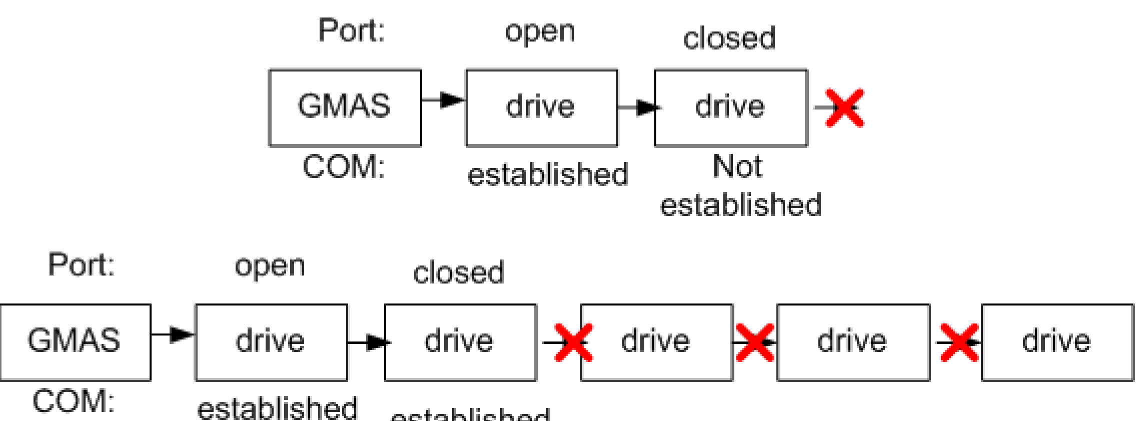
This scan is redundant since slaves which have not disconnected were already identified and another identification is unnecessary. The Hot Plug mechanism identifies only the reconnected slaves and thus spares the redundant EEPROM reads.

If a slave was already validated (already connected and operative) as a configuration slave when the Maestro was initialized, it is not necessary to read its EEPROM even in "Wrong configuration" mode. It is only necessary to read a slave EEPROM when it connects to the network in order to validate that it is the desired slave.

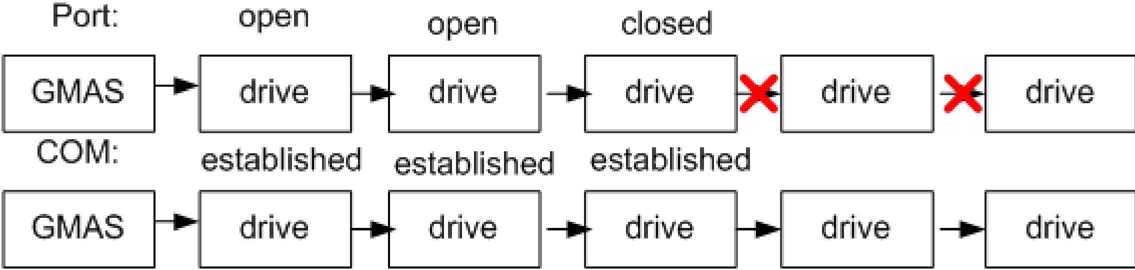
The Wrong configuration mode occurs when the number of slaves is smaller than the number of slaves in the xml (slaves disconnected):



When in Wrong configuration mode, instead of reading all the EEPROMS and validating, the Maestro looks for the disconnection point and waits for a connection at this point, and checks the slave by reading its EEPROM:



GMAS verified the slave's identity and adds it to the valid slaves. If further slaves are connected to this slave, then they are also checked.



If a wrong slave or slaves are connected, the Maestro will continue scanning, and will only stop scanning the network when a valid configuration is located.

Chapter 4: Error Handling

4.1 Function Block Errors

The function blocks provide an internal basic error checking on the input data, which is implementation dependent. For instance, if MaxVelocity is set to 6000, and the Velocity input to a function block is set to 10,000, then either the system slows down or an error is generated. In the case where an intelligent servo drive is coupled via a network to the system, the MaxVelocity parameter is probably stored on the servo drive. The function block provides for and handles the error generated by the drive internally. With another implementation, the MaxVelocity value could be stored locally. In this case, the function block will generate the error locally.

Both centralized and decentralized error handling methods are possible when using the motion control function blocks.

Centralized error handling is used to simplify programming of the function block. Error-reaction is the same independent of the instance in which the error has occurred.

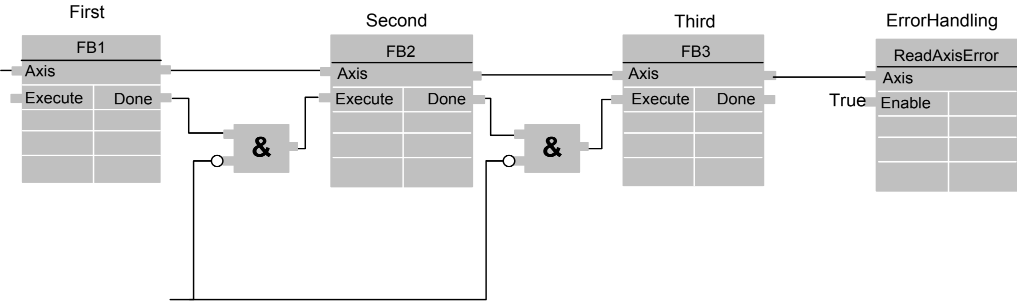


Figure 4-1: Centralized error handling

Decentralized error handling allows for different reactions depending on the function block in which an error occurred.

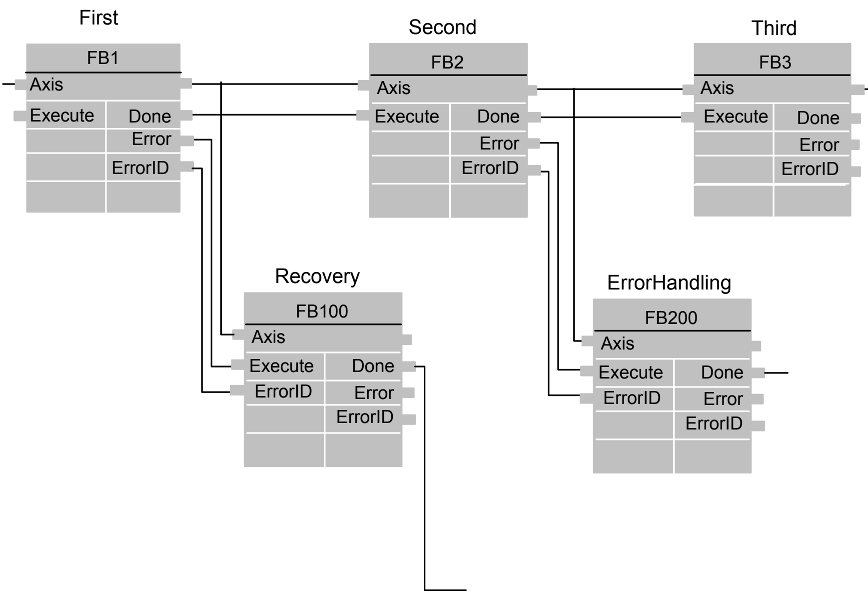


Figure 4-2: Decentralized error handling

4.2 Buffered Errors

All buffered commands will abort if the applicable axis moves to the state ErrorStop. The Error output of the relevantly aborted function blocks are SET. Any subsequent commands will be rejected and the error output is SET.

If a function block has an error e.g. due to a wrong set of parameters, the error output is set, and the behavior is depending on the application program. For instance, with two function blocks, the first instance FB1 executes any motion command on an axis, and starts a new command on a second function block instance FB2 in buffered mode on the same axis. This command is buffered and waits until FB1 is completed (Done). Before the first instance FB1 has finished its command, let one of the following situations occur:

1. The axis goes to state ErrorStop (e.g. due to a following error or over-temperature). FB1 sets the output ERROR. FB2 (as well as any other function block instance that is waiting to execute a buffered command on this axis) sets its ERROR output and shows with the output ErrorID, that it cannot execute its job, because the axis is in a state that doesn't allow it. All buffered commands are cleared. After the axis error is reset by MMC_Reset, it can be commanded again.
2. The FB1 sets its Error output (e.g. due to an invalid parameterization). FB2 becomes active and executes the given command immediately afterwards, and the application should handle the error situation.

4.3 Maestro Error IDs

The following table lists the Maestro error IDs with an explanation for each error. These errors are a result of a continuous event check by the Maestro, causing the system to halt its motion. The system enters an *Errorstop* state and must be Reset to continue its motion.

ID	Explanation	Resolution
0	Indicates that everything is OK, no errors and warnings	
-1	NC driver error	Critical error!!! Refer to Elmo for support.
-2	NC driver memory mapping failure	Critical error!!! Close all resources and restart the Maestro.
-3	Record length is larger than the maximal recorder buffer size. Alternatively, input arguments to the MMC_UploadDataCmd function are not valid.	Verify that NumberOfSignals*NumberOfRows < 1,000,000. Verify the following legacy of the "From " and "To " parameters on MMC_UploadDataCmd function: 1. "From " < "To ". 2. ("To " - "From ") <= 1400. 3. "To " < RecordLen[byte]
-4	Illegal Record Gap value. Space between two successive records in units of CycleTime.	Verify that Recording Gap in the MMC_BEGIN_RECORDING_IN structure is greater than zero.
-5	There is a call to begin recording when the Recorder is already operating	Wait until the recording completes or stop the recording process using MMC_StopRecordingCmd function.
-6	Incorrect Node handle (axis or group of axes)	Verify that you are using the correct axis reference. If you are using the recording mechanism, verify that Recording group input and Recording Parameter input on the MMC_BEGIN_RECORDING_IN structure is correctly defined. If you use the SetKinematicTransformation function, verify that "NC_NODE_HNDL_T" array is correctly defined in the MMC_SETKINTRANSFORM_IN structure.
-7	The Node's function blocks' list is empty	The problem is in the function block list management. Refer to Elmo for support and please describe the exact scenario where it occurred.
-8	Node state is unsuitable for the active function block	You are trying to perform an illegal node state transition. Check that your drives are connected to the Maestro. Refer to the allowed transitions in the PLCOpen state machine definition.
-9	Node is not found	This problem is in the Resource file. Please check that your Resource file is correct.

ID	Explanation	Resolution
		If correct, then check and validate that the axis name and ID is the same as in your application.
-10	You are trying to perform an operation that cannot operate from Disabled mode	To see the allowed node state, refer to the PLCOpen state machine definition.
-11	No free Nodes in the list	Be careful that you have not added more than 96 nodes to the Active Nodes list
-12	The node type is incorrect	You are trying to use an inappropriate node type: 1. You may use only SingleAxis (0) or MultiAxis (1) types. 2. Using a function that is related to single axis, only use a SingleAxis node type. 3. Using a function that is related to multi axis only use a MultiAxis node type.
-13	Incorrect function block type parameter	You are trying to use an incorrect function block number. Allowed numbers are [1…80]. Refer to Elmo for help.
-14	Free function blocks list is empty	You have entered the maximum number of function blocks to the node function block list. The maximum size is 1000 function blocks. Wait a delay time to allow the function blocks to execute. This will result in freeing the active function blocks list. To get the current function block depth use the MMC_GetFbDepthCmd function.
-15	Function block type not supported by Maestro	You tried to use a function that is not presently supported by Maestro. If you still wish to use this function, refer to Elmo for support.
-16	Function block pointer not found	Check if you have defined any nodes in the Resource file before operating with nodes. This error can be caused by an internal function block list management problem. Refer to Elmo for support.
-18	The function block Handle validity check failed	This is an internal problem. Refer to Elmo for support.
-19	The function block is already removed from the active list and marked as free	When the function block has completed its operation, it is automatically removed from the list and marked as free. If you work in IEC mode, it remains on the list even after the function block execution. To remove it from the list, run the **MMC_CmdStatus** function after the function block has completed its action.
-21	Axis used by other group.	You cannot use the same axis in two different groups or in a group and as a single axis node. You can define the same axis in two different places.

ID	Explanation	Resolution
		However, when the axis is activated from one place, it cannot be activated from another place.
-22	One of the parameters is out of the permitted range	If you receive this message while running the motion of a single axis function block, please check the following parameters: 1. 0 < Velocity <= 100E9 (for move velocity -100E9 <= Velocity <= 100E9). 2. 0 < Acceleration <= 1000E9. 3. 0 < Deceleration <= 1000E9. 4. 0 < Jerk <= 2E12. 5. 0 < Buffer Mode < 7. 6. 0 < Direction < 5. For each of the following functions being used, check the appropriate parameter values: **Multi-axis motion function, check the parameters above** (for a single axis) and in addition check the parameters: 1. 0 <= CoordSystem < 4. 2. 0 <= Circle Mode <= 4. 3. 0 <= PathChoice < 3. 4. 0 <= Transition Mode < 6. **SetOverride function, check the parameters:** 1. 0 <= Velocity Factor <= 1. 2. 0 < Velocity Factor eNum < 3. **ReadDigitalInput function, check the parameter:** 0 < Input Number < 32. **PowerON\OFF function, check the parameter:** 1 <= Buffer Mode <= 2. **HomingDS402 function, check the parameter:** 1 <= Homing method <= 35. **To change the coordinate system between two consequential function blocks**, use the specific buffer mode MC_BUFFERED_MODE. The Move Path function supported transition mode is TM_NONE_MODE **When setting the Kinematic transformation function**, check the parameters: 1. NumAxes <= Actual number of axis on the group. 2. McsToAcsFuncID == NC_TR_SHIFT_FUNC (1). **Resexportfile or Resimportfile function, check the following parameter:** 0 <= DownloadType <= 3.

ID	Explanation	Resolution
		Recorder function with one of the following trigger types: 1. TG_RECORDING_TRIGGER_TYPE_EDGE_WindowIn 2. TG_RECORDING_TRIGGER_TYPE_EDGE_WindowOut 3. TG_RECORDING_TRIGGER_TYPE_LEVEL_WindowInside 4. TG_RECORDING_TRIGGER_TYPE_LEVEL_WindowOutside. The uiRP.TG_RECORDING_TRIGGER_LEVEL_1 must be lower than uiRP.TG_RECORDING_TRIGGER_LEVEL_2. **ErrorCorrectionTable functions, check the parameters:** 1. 0 <= ErrorTableNumber <= 3 2. 0 <= TableResolution (axis grid size) <= 22 **HeartBeatTime functions, check the following parameter:** 0 < HeartBeatTime <= 65535 **CanSetSyncTime functions, check the following parameter:** SyncTime < 1000
-23	Axis Update Cycle Period or Shift is not compatible for the group	Verify that cycle period and cycle shift are equal in both, Group and axis.
-24	Incorrect coordinate system was used or the coordinate system is not enabled at the present	Use group function with an accepted and proper coordinate system defined, allowed parameters is: MC_ACS_COORD(1) MC_MCS_COORD(2) If you are using one of the coordinate systems above and still obtain this error then it siugnifies that this current function block does not support this coordinate system. For example; when using the MoveCircularAbsolute function, only MCS coordinate system is supported.
-25	Group cannot receive motion command because it's disabled	Run the Group Enabled function before sending motion commands
-26	Inappropriate connection status	To Download Firmware you need to be connected with only one RPC connection and without any IPC connections. Make sure that you connect to target Maestro with only one EASII application (only one PC). Make sure that you do not have any active application in the Maestro, if you have, stop the prosess or reset the Maestro. Make sure that you do not have any application execution in the StartUp script at: mnt/jffs/usr/UserStartUp.sh.
-28	Cannot open a UDP socket	Check that the socket is not already open. Check that the connection handler is correct, and that the function loads.
-29	Open UDP socket fail	Refer to Elmo for support.
-30	Bind UDP socket fail	Refer to Elmo for support.

ID	Explanation	Resolution
-31	Calling this function from an IPC connection is not permitted	This function cannot operate in IPC connection mode. Run this function in RPC connection mode.
-32	Incorrect connection type parameter	The connection type parameter of the node is not permitted for this action. For example, If you are trying to download firmware, only perform it through the RPC connection. If the error continues, refer to Elmo for support.
-33	Cannot perform any of the operations listed in the Resolution when at least one of the axis is powered on	If you wish to operate one of the following functions, please verify that all your axes are powered Off. 1. Download the file through TFTP. 2. Download the firmware. 3. Exit the Maestro application. 4. Enable a communication gateway. 5. Enable Maestro config mode (preop mode).
-34	Update of the UDP socket on-connection table fails	Cannot update the UDP socket table due to an incorrect connection index or error in the table. Refer to Elmo for support.
-35	Update of the UDP address on-connection table fails	Cannot update the UDP address table due to input error or error in the table. Refer to Elmo for support.
-36	Update of the event mask on-connection table fails	Cannot update the event mask table due to incorrect connection index or error in the table. Make sure that the requested mask is not negative. Otherwise, refer to Elmo for support.
-37	Get UDP socket from connection table fails	Cannot get the UDP socket with the given index. Check the index boundaries. Verify that you really have a socket on this index. Refer to Elmo for support.
-39	Cannot read event mask	Refer to Elmo for support.
-40	Cannot operate this specific function in the current mode	There are certain functions that can only work in CONFIG_MODE_CONFIG. To change the mode from CONFIG_MODE_REGULAR to CONFIG_MODE_CONFIG run the MMC_ConfigCmd function. To return to CONFIG_MODE_REGULAR mode run MMC_ExitCmd" function.

ID	Explanation	Resolution
-43	Set DHCP function sent with the wrong input parameter	When you send the Set_DHCP function you must update the input MMC_DHCP_SET_IN structure. The structure contains one object, set according to the following specification: 1. 0 - Disable DHCP. 2. 1 - Enable DHCP.
-44	Internal process failure	Refer to Elmo for support.
-45	Failed to read the download version status from the flash params table	Refer to Elmo for support.
-46	Failed to recreate the configuration table	Refer to Elmo for support.
-47	Cannot open the directory where resource files are located	Check if the following directory exist /mnt/jffs/MMC/config/resources on Maestro. If you do not have such a directory, create it and download a new resource file to the directory.
-48	Open file failed	Cannot open resource file, check that the resource file exists in the following path: /mnt/jffs/MMC/config/resources/MMCResources.xml. If yes, check the file is correct. If no, create it using EAS and download a new resource file to the directory.
-49	Motion parameter assignment failed	Assign only one of the following motion types to the function: 1. eCAN_OPEN_MOVE_VELOCITY 2. eCAN_OPEN_MOVE_ABSOLUTE 3. eCAN_OPEN_MOVE_RELATIVE 4. eCAN_OPEN_MOVE_STOP 5. eCAN_OPEN_MOVE_HALT 6. eCAN_OPEN_MOVE_HOME
-50	Node in Distributed mode	In Distributed mode, you cannot load a new motion function block when the axis is currently in motion. Wait until the motion is complete and resend the function.
-51	Homing failed	Increase the delay to the HomeTimeOut value. If this does not solve the problem, try to simulate the same homing scenario without Maestro.
-52	Distributed base processor failure	Refer to Elmo for support.

ID	Explanation	Resolution
-53	Maestro does not support this function or this working mode (NC\Distributed)	Try to change the profile mode. If you are in NC mode, change to Distributed mode. If you are in Distributed mode, change to NC mode. If this does not solve the problem, refer to Elmo for support.
-54	Group contains less than 2 axes.	You cannot perform the Group Enable command if you have less than two axes in the group. You can add axes to a group in two ways: 1. Create a new resource file using EAS that has a group with more than one axis inside and download the resource file to Maestro. 2. Use MMC_AddAxisToGroup function to add axis to group dynamically during the running of the application.
-55	Connection handler is incorrect	Critical error!!! You sent an incorrect connection handler to the function. Check that the connection handler is not zero. Check that you perform the connection process successfully, and then use the correct connection handler.
-56	Incorrect motion operation mode	When operating in Distributed mode you must set the appropriate operation mode for each movement function block type: 1. Move Abs\Rel\Add function must work in position profile operation mode. 2. Move velocity\continues function must work in Velocity profile operation mode. 3. Move homing function must work in Homing operation mode. If you try to enable group, all the group members should be in proper NC motion mode. Change the operation mode using the MMC_ChngOpMode function.
-57	The communication type is not appropriate to this function	There are functions for specific connection types (CAN\EtherCAT). If you do not have a proper connection type, you cannot run this function. Try to find another alternative or refer to Elmo for support.
-58	Wrong function argument type	The inputs to the function are not as expected in the API definition. Go to the API function block definition, find the function and verify that all the inputs are correct.
-59	Group is not found	Check that the name of the group in the resource file is the same as in the input structure of this function.

ID	Explanation	Resolution
		Resource file location - "/mnt/jffs/MMC/config/resources/ " Input structure name - "MMC_AXISBYNAME_IN " Function name - "MMC_GetGroupByNameCmd "
-60	Network scan failure	Critical error!!! The Network Scan function must only work with CAN network. Check the hardware connections between the Maestro and the drive. Also, make sure to place terminators in the empty sockets. Verify that the drive's are powered on. Verify that your resource file is appropriate for the setup. If after this validation, the error persists, refer to Elmo for support.
-61	Network get statistics failure	Critical error!!! The GetCommStatistics function only operates with EtherCAT network. 1. Check the hardware connections between the Maestro and the drive(s). 2. Verify that the servo drives are power on. 3. Verify that your resource file is appropriate to the setup. 4. Verify that you have proper EtherCAT network configuration file located on the Maestro at: "/mnt/jffs/MMC/config/Ethercat/cfg.xml" - you can create this file using the EAS application. If after this validation the GetCommStatistics function still does not operate, refer to Elmo for support.
-62	Network reset statistics failure	The ResetCommStatistics function only operates with EtherCAT network: 1. Check the hardware connections between the Maestro and the drive's. 2. Verify that the servo drives are powered on. 3. Verify that your resource file is appropriate to the setup. 4. Verify that you have proper EtherCAT network configuration file that located on the Maestro here: "/mnt/jffs/MMC/config/Ethercat/cfg.xml" - you can create this file using the EAS application. If after this validation, the ResetCommStatistics function still does not operate, refer to Elmo for support.
-63	FS(File System) failure.	Critical error!!! Refer to Elmo for support.
-64	Dynamic memory allocation fail	Critical error!!! Refer to Elmo for support.

ID	Explanation	Resolution
-65	XML parser failure	Critical error!!! Check that all of the following files exist: 1. /var/MMC/config/resources/MMCResources.xml 2. /var/MMC/config/parameters/MMCGeneralPrm.xml 3. /var/MMC/config/parameters/<NODE Name #1>.xml 4. /var/MMC/config/parameters/<NODE Name # Number of nodes>.xml 5. /var/MMC/config/parameters/<group Name #1>.xml 6. /var/MMC/config/parameters/<NODE Name #Number of groups>.xml
-66	Open communication failure.	Critical error!!! Check that you are not already connected. Check that the PC and Maestro are sufficiently connected (check ping between them). If you working with EtherCAT, check that, the cfg.xml is created and located at: /mnt/jffs/MMC/config/Ethercat/cfg.xml and is defined properly.
-67	Close communication failure.	Check if you are not already connected.
-68	Communication scan bus process failure	This function relevant only for CAN network. Refer to Elmo for support.
-69	An attempt was made to add an axis member to a group when the axis or ID is already present in the group.	Try to change the ID of the axis you wish to add, to an unused ID. If you still receive the error, then this node handle is already in the group.
-70	Max number of members in group is reached	Do not add more than 16 axes to a group. If you use the MMC_RemoveAxisFromGroup function, do not use an index larger than 16 to remove the axis.
-72	Profiler recalc end velocity failure	Critical error!!! Refer to Elmo for support.
-74	Real time driver not initialized	Critical error!!! You have a serious problem with your resource file. Check the following parameters: 1. Maximum number of nodes must be less than 96. 2. **For EtherCAT** a. Minimal cycle time is 1000[usec] = 1[msec]. b. Maximal cycle time is 100,000[usec] = 100[msec]. c. Minimal MB cycle time is 1000[usec] = 1[msec]. d. Maximal MB cycle time is 200,000[usec] = 200[msec]. e. Minimal BG cycle time is 100[usec] = 0.1[msec].

ID	Explanation	Resolution
		f. Maximal BG cycle time is 2000[usec] = 2[msec]. g. CycleTime <= MB CycleTime. h. MB CycleTime <= BG CycleTime * 1000. 3. **For CAN** a. Minimal cycle time is 1000[usec] = 1[msec]. b. The baud rate must be one of the following * 125 * 250 * 500 * 800 * 1000. 4. Maximum number of nodes in a group is 16. 5. Missing NC_AXIS definition of non-group type. 6. Missing MODULO definition of non-group type. 7. Missing DIVIDER definition of non-group type. 8. The Type of each node in the "TYPE" field must be one of the following: a. "ds301 " b. "ds401 " c. "ds402 " d. "ds406 “ e. "virtual " 9. The connection type in the "NET_TYPE " field must be one of the following: a. "CAN " b. "EtherCAT " 10. If you use the /var/MMC/config/parameters/MMCGeneralPrm.xml file to update global parameters, take care to set their low and high limits correctly. 11. DIVIDER value must be in the following range [0...4]. 12. MODULO value must be in the following range [0..(DIVIDER-1)].
-75	Wrong parameter cast type	Refer to Elmo for support.
-76	Write attempt to Read Only parameter	You cannot change this parameter. Refer to the Maestro Administrative and Motion API. If you have any further questions refer to Elmo for support.
-77	Incorrect transformation function	When you running MMC_SetKinTransform function you should set the transformation function type as input to the MMC_SETKINTRANSFORM_IN structure.

ID	Explanation	Resolution
		Only the single NC_TR_SHIFT_FUNC (1) function is supported by the Maestro. Set only this function. For further support, contact Elmo.
-78	Error in opening Real time driver	Critical error!!! Refer to Elmo for support.
-79	Internal process failure	Refer to Elmo for support.
-80	Internal process failure	Refer to Elmo for support.
-81	Incorrect member node indexed	When you wish to remove a node from the group, you must use valid indexes. The allowed range is [0…15].
-82	Group object creation failure	The group creation failed. Check the resource file on the Maestro located at: /mnt/jffs/MMC/config/resources/MMCResources.xml. 1. Verify that the group name is unique. 2. Verify that the group ID is unique. 3. Verify that all the named Members of the group are defined with their complimentary Node name. 4. Verify that the DIM parameter is same as the number of Members. 5. Verify that the number of Members is less than 16. 6. Verify that the TYPE is "group ".
-83	Resource is not a group	Refer to Elmo for support.
-84	Resource is an incorrect communication type	Open the resource file on the Maestro, located at: /mnt/jffs/MMC/config/resources/MMCResources.xml Verify that the NET_TYPE is one of the following: 1. CANbus 2. ETHERCAT If they are not, set the NET_TYPE correctly and restart the Maestro.
-86	Resource is an incorrect node type	The Type of each node in the "TYPE" field must be one of the following: a. "ds301 " b. "ds401 " c. "ds402 " d. "ds406 " e. "virtual ".
-87	Resource is set with an incorrect cycle time	Check the CycleTime configurations in the resource file located at: /mnt/jffs/MMC/config/resources/MMCResources.xml.

ID	Explanation	Resolution
		For EtherCAT: a. Minimal cycle time is 1000[usec] = 1[msec] b. Maximal cycle time is 100,000[usec] = 100[msec] c. Minimal MB cycle time is 1000[usec] = 1[msec] d. Maximal MB cycle time is 200,000[usec] = 200[msec] e. Minimal BG cycle time is 100[usec] = 0.1[msec] f. Maximal BG cycle time is 2000[usec] = 2[msec] g. CycleTime <= MB CycleTime h. MB CycleTime <= BG CycleTime * 1000. **For CAN:** a. Minimal cycle time is 1000[usec] = 1[msec]
-88	Resource is using the incorrect baudrate	This error is only relevant for CAN connections. Verify that the baud rate in the resource file is one of the following: * 125 * 250 * 500 * 800. * 1000
-89	Parameter file is out of range	If you use the following file: /var/MMC/config/parameters/MMCGeneralPrm.xml as input for Axis Parameters to the Maestro, make sure to set their low and high limits correctly. To see the limits, refer to the Axis Parameter table.
-90	Resource maximum number of nodes reached	The maximum number of axes has been reached. Reduce the axes number to 96.
-92	Resource file has an incorrect node cycle period divider	The DIVIDER value must be in the following range [0...4].
-93	Resource file has an incorrect node cycle shift modulo	The MODULO value must be in the following range [0..(DIVIDER-1)].
-94	The resource tried to create a node object and failed	The problem is in the resource file. Check the following parameters: 1. Maximum number of nodes must be less than 96. **2. For EtherCAT** a. Minimal cycle time is 1000[usec] = 1[msec] b. Maximal cycle time is 100,000[usec] = 100[msec] c. Minimal MB cycle time is 1000[usec] = 1[msec] d. Maximal MB cycle time is 200,000[usec] = 200[msec] e. Minimal BG cycle time is 100[usec] = 0.1[msec]

ID	Explanation	Resolution
		f.Maximal BG cycle time is 2000[usec] = 2[msec] g. CycleTime <= MB CycleTime h. MB CycleTime <= BG CycleTime * 1000. 3. **For CAN** a. Minimal cycle time is 1000[usec] = 1[msec]. b. The baud rate must be one of the following. * 125 * 250 * 500 * 800 * 1000. 4. Maximum number of nodes in a group is 16. 5. Missing NC_AXIS definition of non-group type. 6. Missing MODULO definition of non-group type. 7. Missing DIVIDER definition of non-group type. 8. The Type of each node in the "TYPE" field must be one of the following: a. "ds301 " b. "ds401 " c. "ds402 " d. "ds406 " e. "virtual " 9. The connection type in the "NET_TYPE " field must be one of the following: a. "CAN " b. "EtherCAT " 10. If you use the /var/MMC/config/parameters/MMCGeneralPrm.xml file to update global parameters take care about their low and high limits. 11. DIVIDER value must be in the following range [0...4]. 12. MODULO value must be in the following range [0..(DIVIDER-1)].
-95	Resource tried to create a group object and failed	Refer to Elmo for support.
-96	Resource has reached the maximum number of nodes in the group	Check in the Resource file, and make sure that you do not exceed the maximum limit of nodes in a Group. The maximum number of nodes in a group is 16.
-97	The parameters for the shared memory key failed	Refer to Elmo for support.

ID	Explanation	Resolution
-99	The parameters for the Get shared memory, failed	Refer to Elmo for support.
-100	Communication initialization failure	Refer to Elmo for support.
-101	One group member is not in StandStill	You tried to move the Group's state to Enabled while its state is Disabled. One of the group members is not in StandStill state. Check the state of all axes in the group. For each axis that is in Disabled state, run the Power ON command using the MMC_PowerCmd function. If an axis is in error state, run the MMC_ResetCmd function to reset the error and then run the Power ON command. If an axis is in motion state, wait until it completes the motion before running the Group Enable function.
-103	Synchronized motion is prohibited	You are not able to operate any motion function such as MoveAbsolute, MoveRelative, MoveVelocity, Stop, Halt etc.to a single axis drive, when the same drive is a member of a group whose state is enabled or any other state other than GroupDisabled. To operate motion of this axis you have the following possible solutions: * Disable the group where it is a member and then run the single axis motion function. * Remove this axis from all groups where it is a member, and then run the single axis motion function. * Move this axis using a MultiAxis functions like MoveLinearAbsoulute, MoveLinearRelative, GroupStop, GroupHalt etc.
-104	Incorrect recording parameter	You exceeded the maximum signal number (signal index). When you define which signal to record in the Recording Vector, the maximum signal number can only be 57.
-105	Change of coordinate system while in motion is not permitted	You cannot change the coordinate system when the group is in motion. Wait until the motion is completed, and then change the coordinate system. Use the MMC_GetFbDepthCmd function to examine how many function blocks are currently in the list. If you receive "0 ", there are no function blocks in the list and you can change the coordinate system.
-106	MCS coordinate system is not set	If you try to operate a function in the MCS coordinate system, you must to run the MMC_SetKinTransform function before executing the new function.

ID	Explanation	Resolution
		If you perform any changes in your group members (add\remove axis) you must run the MMC_SetKinTransform function again.
-108	The motion is not 2D circular angle motion	You tried to operate the MoveCircularAbsolute function in Angle mode but the number of axis in your group is more than 2. Use MoveCircularAbsolute function with only 2 axes in the group.
-109	Parameter index has an incorrect value	When using Read\Write Real\Bool parameter functions, the parameter index must be limited to [0…51].
-110	Get diagnostics has a communication failure	Critical error!!! Check your cable connections. Verify that you configured the EtherCAT network with the configuration file properly. The EtherCAT configuration file must be located at: /mnt/jffs/MMC/config/Ethercat/cfg.xml If you still get this error, refer to Elmo for support.
-111	Parameter has no value	When you using Read\Write Real\Bool parameter functions, the parameter value contain NULL. Refer to Elmo for support.
-112	Internal process failure	Refer to Elmo for support.
-113	Already in desired mode	If you want to Enable\Disable one of the following: * CommGateWay * ConfigMode But you cannot, you are already in the desired state.
-114	Internal process failure	Refer to Elmo for support.

ID	Explanation	Resolution
-115	You have chosen an incompatible Buffer mode vs. Transition mode combination	If you use a group motion function, you must enter a Buffer mode and Transition mode.Not all combinations are permitted. The list of all permitted combinations: 1. MC_ABORTING_MODE & MC_TM_NONE_MODE 2. MC_BUFFERED_MODE & MC_TM_NONE_MODE 3. MC_BLENDING_LOW_MODE & MC_TM_DEFINED_VELOCITY_MODE 4. MC_BLENDING_LOW_MODE & MC_TM_CORNER_DISTANCE_MODE 5. MC_BLENDING_LOW_MODE & MC_TM_MAX_CORNER_DEVIATION_MODE 6. MC_BLENDING_LOW_MODE & MC_TM_SWITCH_RADIUS_MODE 7. MC_BLENDING_PREVIOUS_MODE & MC_TM_DEFINED_VELOCITY_MODE 8. MC_BLENDING_PREVIOUS_MODE & MC_TM_CORNER_DISTANCE_MODE 9. MC_BLENDING_PREVIOUS_MODE & MC_TM_MAX_CORNER_DEVIATION_MODE 10. MC_BLENDING_PREVIOUS_MODE & MC_TM_SWITCH_RADIUS_MODE 11. MC_BLENDING_NEXT_MODE & MC_TM_DEFINED_VELOCITY_MODE 12. MC_BLENDING_NEXT_MODE & MC_TM_CORNER_DISTANCE_MODE 13. MC_BLENDING_NEXT_MODE & MC_TM_MAX_CORNER_DEVIATION_MODE 14. MC_BLENDING_NEXT_MODE & MC_TM_SWITCH_RADIUS_MODE 15. MC_BLENDING_HIGH_MODE & MC_TM_DEFINED_VELOCITY_MODE 16. MC_BLENDING_HIGH_MODE & MC_TM_CORNER_DISTANCE_MODE 17. MC_BLENDING_HIGH_MODE & MC_TM_MAX_CORNER_DEVIATION_MODE 18. MC_BLENDING_HIGH_MODE & MC_TM_SWITCH_RADIUS_MODE 19. MC_BUFFERED_MODE & MC_TM_SWITCH_RADIUS_MODE

4.4 Error correction error IDs

ID	Explanation	Resolution
-116	You have a problem with the header of the Error correction input file	Check that the error correction input file exists in the correct path you associated it. The error correction input file has a 5-rows header. Each row must contain a valid value according to the description: 1. Table size (number of points) 2. Start offset – the offset from the beginning of the shared memory segment 3. Error Table dimension (1, 2, or 3) 4. Start position - #Xpos #Ypos ... (as dimension number) 5. Axis grid size - Axis step on the correction table (in units $Log_2(X)$) (number of values as dimension number) 6. Table dimensions - #Row's #Column's
-117	The Error correction table is wrong	Check the data table in the file you created is correct. The size of the table must be as described in the header. All the values in the table must be numbers. All the delimiters between the numbers must be a tab.
-118	Cannot add another table	You have reached the maximum number of error correction tables. The maximum number of tables is 4. If you wish to add to this table, remove another table using DisableErrorTable and then the UnloadErrorTable functions. Another reason for this error maybe that you have reached the maximum limit of values in all ErrorCorrectionTables. The maximum limit is 100,000.
-119	Error Correction Table is already taken	You cannot allocate the same table twice.
-120	Cannot unload error correction table	You are trying to unload an error correction table that is already enabled. To unload the table run DisableErrorTable function and then run the UnloadErrorTable function.
-123	Cannot operate with current error correction table	You are trying to operate with an error correction table that is not loaded. Load the table using LoadErrorTable function and then try again.

ID	Explanation	Resolution
-124	Error correction table is already mapped to this axis	You cannon allocate an error correction table to an axis when another error table is already allocated to this axis. If you wish to allocate this error correction table to this axis, free this axis of any other error correction tables using DiableErrorTable and UnloadErrorTable functions.

4.5 Continued Maestro Error IDs

ID	Explanation	Resolution
-125	Cannot change axis operation mode	You cannot change the operation mode of the axis when the axis is a member of a group. First, remove this axis from all groups where it is a member, and then, run the function.
-126	You have exceeded the limit of the ModBUS table	Make sure that you have not exceeded the limit of any of the ModBUS tables. Their limits are as follows: Holding register - 31743 Coils - 1,024 Inputs - 1,024
-127	ModBUS table ID is incorrect	When referencing the ModBUS table, the table ID should be legal. The table ID should be an integer between [1 ... 247].
-128	The ModBUS server is already running	You cannot start the ModBUS server when the ModBUS is already in RUNNING state. To check the current ModBUS state use MMC_MbusIsRunning function. To change the ModBUS to non-RUNNING state, use MMC_MbusStopServer function.
-129	The file transfer through TFTP failed	**Download** Make sure that: Your TFTP server is turned on and points to the correct directory. The file name you enter is correct and the file exists. You set the correct IP address. **Upload** Make sure that: Your TFTP server is turned on and points to the correct directory. You set the correct IP address.
-130	Transition mode is out of range	Use only one of the following transition modes: MC_TM_NONE_MODE = 0 MC_TM_MAX_VELOCITY_MODE = 1 MC_TM_DEFINED_VELOCITY_MODE = 2 MC_TM_CORNER_DISTANCE_MODE = 3 MC_TM_MAX_CORNER_DEVIATION_MODE = 4 MC_TM_SWITCH_RADIUS_MODE = 5 MC_TM_CORNER_DIST_TC_POLYNOM = 6 MC_TM_CORNER_DIST_CV_POLYNOM3 = 7 MC_TM_CORNER_DIST_CV_POLYNOM5 = 8
-131	The error table number is out of	You can only use one of the following numbers:

ID	Explanation	Resolution
	range	0,1,2,3.
-132	Error table resolution is out of range	Set a permitted resolution between [1.....22].
-133	The current segment in the error table is out of range	Check the following parameters: * Start offset should be a positive number. * Start offset should be a less then 100,000. * (Error table size + Start offset) sould be less then 100,000.
-134	Trigger window parameters is out of range	You entered illegal recording parameters to the recording parameters vector (Rp). When you set the MMC_BEGIN_RECORDING_IN structure, make sure that the Rp[4] value are less or equal to Rp[5].
-135	The file type that you chose does not exist	When you download\upload file through TFTP you can only use one of the following file types: MMC_PARAMETER_FILE_DOWNLOAD = 0 MMC_RESOURCE_FILE_DOWNLOAD = 1 MMC_SNAPSHOT_RESOURCE_FILE_DOWNLOAD = 2 MMC_ETHERCAT_CFG_FILE_DOWNLOAD = 3 MMC_PERSONALITY_FILE_DOWNLOAD = 4
-136	Incorrect "SetIsToLoad" value	The input value uses a logical convention. You can only set [0\1] values as input.
-137	Set override input arguments are out of range	You have set the incorrect input arguments to the override function. Check the following inputs: * Velocity factor should be between 0.1 and 1. * Update the velocity factor Idx to only be NC_OVERRIDE_MAX_VEL_FACTOR (3).
-138	The number of axes is out of range	When using the "MMC_SetKinTransform" function, you cannot set the number of axes to larger than the actual number of axes in the group. The number of axes should be the actual number of axes that comprise the group motion.
-139	Transformation function is out of range	The transformation function can be only one of the following: * NC_TR_NONE_FUNC (0) – where there are no transitions between ACS and MSC. * NC_TR_SHIFT_FUNC (1) – where there are linear transformations between ACS to MCS but no reverse transformations between MCS and ACS.
-140	Buffer mode is out of range	You can only choose one of the following modes: MC_NULL_BUF_MODE = 0

ID	Explanation	Resolution
		MC_ABORTING_MODE = 1 MC_BUFFERED_MODE = 2 MC_BLENDING_LOW_MODE = 3 MC_BLENDING_PREVIOUS_MODE = 4 MC_BLENDING_NEXT_MODE = 5 MC_BLENDING_HIGH_MODE = 6
-141	Homing method is out of range	The homing methods range is [1 .... 35]
-142	Digital input number is out of range	The input digital range is [0 .... 31]
-143	Maximum velocity parameter is out of range	Check the maximum allowed velocity using the "MMC_ReadParameter" function. The maximum velocity parameter enumerator is "MMC_MAX_VELOCITY_PARAM" (15).
-144	Maximum acceleration parameter is out of range	Check the maximum allowed acceleration using the "MMC_ReadParameter" function. The maximum acceleration parameter enumerator is "MMC_MAX_ACCELERATION_PARAM" (19).
-145	Maximum deceleration parameter is out of range	Check the maximum allowed deceleration using the "MMC_ReadParameter" function. The maximum deceleration parameter enumerator is "MMC_MAX_DECELERATION_PARAM" (23).
-146	Maximum jerk parameter is out of range	Check the maximum allowed jerk using the "MMC_ReadParameter" function. The maximum jerk parameter enumerator is "MMC_MAX_JERK_PARAM" (26).
-147	Direction mode is out of range	The direction mode can be only one of the following: MC_POSITIVE_DIRECTION = 1 MC_SHORTEST_WAY = 2 Not implemented as yet MC_NEGATIVE_DIRECTION = 3 MC_CURRENT_DIRECTION = 4
-148	Zero or negative velocity input	Change the velocity parameter to a positive value.
-149	Coordinate system is out of range	The coordinate system can be only one of the following: MC_ACS_COORD = 1 MC_MCS_COORD = 2 MC_PCS_COORD = 3
-150	Circle mode is out of range	The circle mode can be only one of the following: MC_BORDER_CIRC_MODE= 1 MC_CENTER_CIRC_MODE = 2 MC_RADIUS_CIRC_MODE = 3

ID	Explanation	Resolution
		MC_ANGLE_CIRC_MODE = 4
-151	Path choice is out of range	The path choice can be only one of the following: MC_NONE_PATH_CHOICE = 0 MC_CLOCKWISE = 1 MC_COUNTERCLOCKWISE = 2
-152	Arc short long is out of range	TThe arc short long can only be one of the following: MC_NONE_ARC_CHOICE = 0 MC_SHORT = 1 MC_LONG = 2
-153	Input position is out of range	The maximum allowed position defined by Modulu parameter in the SetPosition function.
-154	Unable to set the requested slave state	Before beginning the FOE procedure, make sure to set the Maestro and change all slaves automatically to BOOT state. At the end of the FOE procedure, make sure to reset the Maestro and change all the slaves automatically back to OPERATIONAL state. If one of the slaves fails to change, state you will receive this error. Try changing the slave state manually using the EtherCAT configurator in EAS. Otherwise, Refer to Elmo for support.
-155	FoE File system error	Make sure that the requested file is successfully copied to the Maestro and located at: "/tmp/[file name]". Check that the file permissions are set to "rw". Otherwise, Refer to Elmo for support.
-156	Write FoE packet error	Refer to Elmo for support.
-157	Unable to close FoE communication	Refer to Elmo for support.
-158	You have reach the maximum number of FoE retries	Verify that you have performed the bulk read configuration successfully, before reading the data.
-159	Failure during FoE TFTP transfer	Make sure that the input arguments of the DownloadFoE function are correct. The server should be set to PC IP. The filename should also be present in the PC.
-160	The Maestro unable to open FoE connection with any one of the slaves	Check the communication cables between the Maestro and drives. Otherwise, Refer to Elmo for support.
-161	Not in use at this time	Not in use at this time

ID	Explanation	Resolution
-162	Cannot retrieve the recorded Bulk Read values array	Refer to Elmo for support.
-163	Wrong input to ConfigBulkRead function	Check the following parameters: ucIsPreset should be set to 0 or 1. uBulkReadParams-> eBulkReadPreset should be set between 1 and 4.
-164	You have exceeded the maximum data packet size	The maximum packet size is 1400 [bytes]. To reduce the packet size, reduce one of the following parameters: Number of axes Number of signals for each axis
-165	Bulk read memory allocation failed	Refer to Elmo for support.
-166	Trying to read from an unallocated memory segment	Verify that you have performed the bulk read configuration successfully, before reading the data.
-167	Number of axes is out of range	The number of axes can be between 1 and 100.
-168	Configuration enumerator is out of range	The function bulk read can work separately and parallel for two different applications.
-169	Unable to create FoE Download Thread	Refer to Elmo for support.
-170	EtherCAT system function failure	Check the communication cables between the Maestro and drives. Otherwise, refer to Elmo for support.
-171	There is no open EtherCAT communication on the Maestro	Refer to Elmo for support.
-172	You have entered an incorrect number of slaves	The number of slaves should be greater than zero and less or equal to the actual number of slaves
-173	You have entered the wrong slave ID to the input structure	The input structure of the function contains the list of slave ID's. Each of them should be valid, and should reference an equal or less than, but greater than zero, actual number of slaves.
-174	Wrong input enumerator to IEC control function	Set a legal enumerator value to the input structure
-175	IEC system error	Refer to Elmo for support.

ID	Explanation	Resolution
-176	The present type of parameter is not suitable for this function	If you are using the following: [read\write] Axis Parameter function — Only use axis related parameters [read\write] Group Parameter function — Only use group related parameters [read\write] Global Parameter function — Only use global related parameters
-177	Cannot retrieve the node pointer	Make sure that you have entered a correct Axis\Group reference.
-178	The node is in motion	You cannot clear the function block list while the Axis\Group is in motion. Stop the axis\group and repeat the function command.
-179	Unable to clear the function block list	Refer to Elmo for support.
-180	Transition mode is not supported	Please use another transition mode.
-181	Wrong coefficient ratio	When using the NC_TR_SHIFT_FUNC function in a Kinematic transformation function, always maintain the ratio between the coefficients (actual used in the axes): NC_BACK_TR_RATIO_COEF•NC_BACK_SHIFT_COEF = 1 The maximum permitted deviation is 1E-5
-182	Inappropriate state for the SetKinTransform function	The SetKinTransform function can only operate in one of the following states: 1. GROUP_DISABLED 2. GROUP_STANDBY
-183	Illegal input value to MMC_SetEnquireFbStatusCmd	The input variable will not accept all numerical values (including -1, and 2). It will only accept 0 or 1.
-184	The path choice input is not acceptable	Please verify that the path choice parameter is correctly set for the circle mode you are using: • Border mode - Must be MC_NONE_PATH_CHOICE • Center mode - Must be MC_NONE_PATH_CHOICE • Radius mode - Must not be MC_NONE_PATH_CHOICE • Angle mode - Must be MC_NONE_PATH_CHOICE
-185	The arc short long input is not acceptable	Please verify that the arc short long parameter is correctly set for the circle mode you are using: • Border mode - Must be MC_NONE_ARC_CHOICE • Center mode - Must not be MC_NONE_ARC_CHOICE • Radius mode - Must not be MC_NONE_ARC_CHOICE • Angle mode - Must be MC_NONE_ARC_CHOICE

ID	Explanation	Resolution
-186	**General profiler error**	**This is an emergency error. Call Elmo immediately.**
-187	An incorrect event mode input was set to the function	Please reset the input Event mode to any of the following: (0) - No notification (1) - Cycle notification (2) - Immediate notification
-188	This function is only supported in a 3D situation	If you are using the circular radius function, define the system kinematics (using MMC_SetKinTransform) so that it includes all Cartesian directions (X,Y,Z)
-189	Cannot initiate another bulk upload process	Only one bulk upload process is allowed at a time. Make sure that the current bulk process has completed using "MMC_GetBulkUploadStatusCmd", and then try again.
-190	A message arrival has timed out	Refer to Elmo for support.
-191	The response message is unexpected when considering the request message (DS301 layer).	Refer to Elmo for support.
-192	Failed to create the thread managing the upload process	Refer to Elmo for support.
-193	Bulk upload initialization process failed	Refer to Elmo for support.
-194	Internal resources allocation error	Refer to Elmo for support.
-195	The delay setting between messages failed	Refer to Elmo for support.
-196	Trying to upload a buffer that is zero or negatively sized	Refer to Elmo for support.
-197	Trying to upload a buffer that is larger than the IPC packet size	Make sure that the high buffer limit is lower than 65,600.
-198	No pending messages in the queue	Refer to Elmo for support.
-199	No valid message – zero length	Refer to Elmo for support.
-200	Unexpected message – communication layer.	Refer to Elmo for support.
-201	The sequence number is unexpected	Refer to Elmo for support.
-202	Illegal bulk upload buffer limits	Make sure that the high buffer limit is higher that low buffer limit. Make sure that the difference between high and low limit, is lower than 1400 (IPC packet size).
-203	The current kinematic direction is	Make sure that you only select a supported kinematic direction.

ID	Explanation	Resolution
	not supported	
-204	N axes definition are not consecutive	When you define the kinematic directions in the vector, make sure that all used N axes are defined consecutively.
-205	The personality file format is wrong	Make sure that the personality file located at: var/MMC/config/resources/Personality.xml is not corrupt.
-206	The requested data is not present in the personality file	Make sure that the personality file located at: /var/MMC/config/resources/Personality.xml is up to date.
-207	Not all the axes are defined in the Kinematic group	When you perform a kinematic definition you should set all the members of the group to the kinematic group.
-208	Unsupported definition of only one Cartesian direction	Elmo does not support the Cartesian coordinate system in only one direction. If you wish to continue using the Cartesian coordinate system in only one direction: Either simply run the linear FB's in ACS mode. The disadvantage of using ACS mode is that you cannot use the transformation mechanism. Alternatively, another solution is to use this axis as N axis.
-209	Trying to select an unsupported coordinated system e.g. PCS	Currently, only the following coordinate systems are supported: • ACS. • MCS. Error generated is NC_NOT_SUPPORTED_COORDINATE_SYSTEM
-210	Current transition mode is not supported in ACS mode	When inserting an FB in ACS mode,only set one type of transition mode. The only supported transition mode in ACS is "MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES"
-211	Contradiction between high and low software limits	When setting the high or low software limit, make sure that the "low" limit is lower than the "high" limit and vice versa.
-212	Motion with the current function block is forbidden with the present hardware limit	When the axis is located at a hardware limit, the only motion permitted is using MC_MoveAbsolute\Relative or other permitted function blocks.
-213	Motion with current function block is forbidden at the ACS software limit	When the axis is located at a software limit, the only motion permitted is using MC_Move[Linear]Absolute\Relative or other permitted function blocks.
-214	Multi Axis motion in the current direction is forbidden due to the presence of an ACS limit	When at least, one of the axes in the group is located at an ACS software limit, the only motion allowed is when the direction of each problematic axis is opposite to the direction of the limit: * software high limit – negative * hardware FLS limit – negative

ID	Explanation	Resolution
		* software low limit – positive * hardware RLS limit – positive Refer to the section 3.2. Status Register in the Maestro Administrative and Motion API.
-215	Motion with current function block is forbidden at the MCS software limit	When at least, one of the kinematic directions of the group is located at a software limit, the only motion permitted is using MC_MoveLinearAbsolute\Relative or other permitted function blocks.
-216	Single Axis motion towards a software limit is forbidden	Check the software limit values of the current axis using the MC_ReadParameter function with the following enumerators: * MMC_software_LIMIT_HIGH_POS_PARAM - for high limit * MMC_software_LIMIT_LOW_POS_PARAM - for low limit Then make sure to request movement of the axis to positions only within the limits.
-217	Multi Axis motion toward a software limit is forbidden	Check the software limit values in all members of the current group: **When working in ACS mode** Use the MC_ReadParameter function, for each member with the following enumerators: * MMC_software_LIMIT_HIGH_POS_PARAM - for high limit * MMC_software_LIMIT_LOW_POS_PARAM - for low limit When working in MCS mode Use the MC_GroupReadParameter function (in group level) with the following enumerators: * MMC_MCS_HIGH_LIMIT_ARRAY - for high limit * MMC_MCS_LOW_LIMIT_ARRAY - for low limit Then make sure to request movement of the group only to positions where all members\directions are between these limits.
-218	Single Axis motion in the current direction is forbidden due to the presence of a limit	When the axis is located at a hardware\software limit, the only motion allowed is in the opposite direction to the limit: * software low limit – positive * software high limit - negative * hardware FLS limit – negative * software RLS limit – positive Refer to the section 3.2.3 32bits Status Register in the Maestro Administrative and Motion API.
-219	Multi Axis motion to current direction is forbidden due to the presence of an MCS limit	When at least, one of the kinematic directions of the group are in MCS software limit, the only motion allowed is when the motion direction of each problematic kinematic direction opposes the direction of the limit:

ID	Explanation	Resolution
		* software low limit - positive * software high limit – negative Refer to the section 3.2.3 32bits Status Register in the Maestro Administrative and Motion API.

4.6 Maestro PVT/ECAM Motion Error IDs

ID	Explanation	Resolution
-221	Bulk upload send acknowledgement failed	Refer to Elmo for support
-222	Failed to open the file containing the PVT/ECAM table	Check that the file exists, and has the relevant permissions, etc.
-223	Problem in header of the PVT/ECAM file	Check that the file is compliant with the format Elmo supplied/Refer to Elmo for support
-224	Trying to execute ECAM motion on PVT table, or vice versa	Execute the PVT motion on the PVT table, or ECAM motion on the ECAM table
-225	Failed to read the table values	Check that the file is compliant with the format Elmo supplied/Refer to Elmo for support
-226	Failed to insert the PVT/ECAM table to the Maestro RAM	Verify that the table size does not exceed the following limit: TableSize = (24 + Dimention*(12 + NumberOfPoints*16) + NumberOfPoints*8)[bytes] should be lower than 1[Mbyte]. Verify that you have unused memory entries. The maximum number of inserted table points is 20. If the number of inserted table points is >20, perform UnloadTable, and then reload the table pointss in table groups of 20.
-227	The actual dimension of the vector does not match the dimension mentioned in header or trying to execute a vector motion on a 1D table	Change the input parameters according to requirements
-228	The path ID provided is not legal	Provide a legal hMemHandle, as returned by MMC_LoadTableFromFileCmd() or MMC_InitTableCmd()
-229	Trying to move an unallocated segment	Use MMC_InitTableCmd() and MMC_AppendPointsToTableCmd()/MMC_LoadTableFromFileCmd()
-230	The read and expected number of points do not match	Provide a correct expected number of points in header /edit the table so that the values will match
-231	Trying to execute a PVT/ECAM motion on some axis/vector, but the path was prepared for another axis/vector	Use the correct axis/vector
-232	Provided a path that contains two or less points	Provide a longer path
-233	Journal handle out of range (similar to NC_PVT_ECAM_MEM_HANDLE_OUT_OF_RANGE)	Refer to the resolution of the error NC_PVT_ECAM_MEM_HANDLE_OUT_OF_RANGE
-234	File name contains invalid characters	Provide a legal file name

4.7 Continued Maestro Error IDs

ID	Explanation	Resolution
-235	Buffered\Blended function block insertion at a limit is forbidden	When the axis\group is at a limit, buffered\blended motion is forbidden. At the limit, insert any motion function block in abortion mode, or when the function block queue of the axis\group is empty.
-236	Torque out of range	Please set torque within the allowed range.
-237	Parameter index is out of range	Please use only accessible indexes.
-238	Cannot define the same node twice	When you define the kinematic of the system, assign each member in a group to a kinematic direction. However, the same member cannot be assigned twice. Make sure that no member is defined twice in the hNode array.
-239	Could not retrieve the current index	Refer to Elmo for support
-240	Trying to append data too close to the current index	Refer to Elmo for support
-241	Trying to append data to uninitialized table	Use MMC_InitTableCmd() prior to MMC_AppendPointsToTableCmd()
-242	Trying to append data beyond the end of the table/overlapping the current index	Provide a smaller segment to append
-243	The start index provided is beyond the end of table	Provide a legal start index
-244	The end index calculated is beyond the end of table	Provide a smaller segment to append
-245	No data to append	Provide data to append
-246	"Holes" inside the table	Append to index adjacent to previously appended
-247	Underflow threshold value is larger than the segment length	Underflow threshold value cannot be bigger than the segment length. Provide smaller underflow threshold value.
-248	Maximum Modbus packet size exceeded	You cannot request to read or write a packet larger than 250 memory cells at one function call.
-249	Axis Link mode is out of range.	You should set one of the following Link types : 0 – Target Mode 1 – Actual Mode
-250	One of the axes is not in StandStill state.	When linking two axes, both of them should be in StandStill state.
-251	A slave Linked axis cannot be physical axis.	When linking two axes, the slave cannot be a physical axis. The slave axis should be a virtual axis, defined in the

ID	Explanation	Resolution
		system resource file.
-252	The offset variable of the Slave Linked axis should equal zero.	When trying to link or unlink two axes, the offset variable of the slave axis should be zero. Send a motion command to the slave axis with zero target position before linking or unlinking the axis.
-253	Cannot unlink an axis when slave axis in motion.	When trying to unlink two axes, the slave axis should be in a non-motion state. If the axis is in motion, wait until the motion has completed and repeat the unlinking process. Otherwise, run the Stop procedure for the slave axis and repeat the unlinking process.
-254	Cannot link the same axis twice.	When the axis is already linked as a Master or Slave, it cannot be linked again. If you wish to link the current axis to some other axis, first unlink it and then retry the linking process.
-255	Axis Linking is only supported for DS402 nodes.	When trying to link two axes, both must be DS402 nodes.
-256	Cannot open parameters file	Cannot open parameters file, check that the parameters file exists in the following path: "/mnt/jffs/MMC/config/parameters/MMCParameters.xml ". If yes, check the file is correct. If no, create it by using ""MMC_SaveParamCmd" function or copy a new file from PC using "MMC_ResImport" function or through EAS interface.
-257	Cannot use character variable for recording trigger	Please use other variable for trigger
-258	Current parameter cannot be part of queued write	Change this parameter in immediate mode
-259	Wrong number of parameters	The number of parameters should be between 1 and 5.
-260	Dwell parameter is small	The dwell parameter presented in microseconds and cannot be less than the cycle time.
-261	Maximum limit of Immediate function blocks is reached	When an immediate administrative function block is inserted, do not insert more than 5 FB's to same motion FB.
-262	Cyclic mode out of range	When working with PVT\ECAM, the sequence can be run in cyclic mode or in single mode. Choose the mode using "ucIsCyclic" parameter: 0 - non-cyclic mode 1 - cyclic mode

ID	Explanation	Resolution
-263	Dynamic mode out of range	When working with PVT\ECAM, the sequence can be run in dynamic mode or static mode. In dynamic mode, you can add additional point on the fly. Choose the mode using "ucIsDynamic" parameter: 0 - static mode 1 - dynamic mode
-264	Auto mode out of range	When working with PVT\ECAM in Dynamc mode, you can select where to enter the additional points in the table. In auto mode the additional points will be inserted from the last inserted point. In manual mode the additional points will be inserted from the start index provided by user. Choose the mode using "ucIsAutoAppend" parameter: 0 - manual mode 1 - auto mode
-265	Position mode out of range	When working with PVT\ECAM, you can define whether the position in the file\table is absolute or relative to the last commanded position. Choose the mode using "ucIsPosAbsolute" parameter: 0 - relative mode 1 - absolute mode
-266	Time mode out of range	When working with PVT\ECAM, you can define whether the time of the each point in the file\table is absolute or relative to the last point. Choose the mode using the "ucIsTimeAbsolute" parameter: 0 - absolute mode 1 - relative mode
-267	File mode not allowed for this function	When working with PVT\ECAM, you should choose the type of the motion table NC_MOTION_TABLE_TYPE_ENUM. If you trying to run one of the following functions: * MMC_InitTable * MMC_AppendPointsToTable The mode of the table cannot be a file type, only an array type. Set the table type using "eTableType" variable.
-268	Cannot append to static table	When working with PVT\ECAM, you can add points dynamically using the Append function. In order to provide the appendment, you should initialize the table in dynamic mode, using the "ucIsDynamic" variable.

ID	Explanation	Resolution
-269	Small time interval	When working with PVT\ECAM, the time interval between two consecutive points cannot be less than the cycle time.
-270	Spline insertion is forbidden	A spline table cannot be inserted when you already have a PVT\ECAM table in the memory. Remove the PVT\ECAM table using "MMC_UnloadTable" function and then retry.
-271	PVT\ECAM insertion is forbidden	A PVT\ECAM table cannot be inserted when you already have spline table in the memory. Remove the spline table using "MMC_PathUnselect" function and then retry.
-272	Cannot insert non-immediate administrative function block to pending node	When a motion function block is entered with execute "0", it will move the axis\group to the pending state. The axis will stay in this state until the user enters another motion function block with execute "1". In pending state, any non-immediate administrative function block insertion is forbidden. Make sure that the last motion function block prior to the non-immediate administrative function block, is inserted with execute "1".
-273	Cannot use absolute time	When you operate with PVT in cyclic mode, you cannot use absolute time. In cyclic mode set the IsAbsoluteTime flag to **0** in order to work in relative time mode.
-274	Cannot copy the parameters file to target destination	Refer to Elmo for support.
-275	Memory overlap between error correction tables	When you insert error correction table, specify the start offset where the table will be inserted. If the new insertion will cause a memory overlapping with another previously inserted error correction table, the new table will not insert and an error is generated. Remove the unnecessary tables from Maestro and then retry to insert the table. Otherwise try to choose a better start offset.
-276	Cannot enable virtual encoder	In order to enable the virtual encoder two conditions should be fulfilled: * Virtual encoder in disabled * RPDO3 are not configured
-277	Contradiction between coordinate systems	When operating with Splines or Multi axis PVT, define the requested coordinate system (ACS\MCS\PCS) on both, table insertion and motion functions. If the coordinate system is different for the insertion and motion functions, an error is generated. Select the same coordinate system in both functions.

ID	Explanation	Resolution
-278	Support only Cartesian axes	When operating with Splines or Multi axis PVT in MCS mode, the only supported axes is the Cartesian axes, X, Y, Z. Run the function SetKinTransform with the requested vector and choose only Cartesian directions, with no "N" axes, Polar axes, or "S".
-279	Append block is too large	The maximum size of append block is 170 doubles. Maximum number of points is calculated as: MaxPoints = 170/[2*Dimention+1]. 1D - 56 points 2D - 35 points 3D - 24 points etc...
-280	Table size is too large	You are trying to insert too large a PVT\Spline table. Try to reduce the table size and try again.
-281	Node sent a CAN SDO abort reply	This error occurs after the Maestro sends an SDO download\upload request to the node and receives an SDO abort response. The response code is in accordance with the CiA SDO download/upload protocol found in the EtherCAT Application Manual section 6.2. Abort SDO Transfer Protocol. Please check that the destination object index and sub-index are supported by the CANopen standards or by the specific vendor standards.
-282	SDO request timeout	Maestro sent SDO download\upload request and did not receive any response in the default timeout period. Please check physical connections with the node and that it is powered on.
-283	CAN driver failure	CAN chip driver failure occurred, please refer to ELMO for support.
-284	Segmented SDO toggle bit unchanged	Segmented SDO toggle bit remained unchanged since last segment transfer. Please see DS301 CiA "application Layer and communication profile" document for further details. Refer to ELMO for support.
-285	Invalid PDO number	PDO number is invalid for this API, please see API documentation for allowed PDO numbers for this API (try 0,1 instead of 3,4).
-286	Status word fault bit reset timeout	Maestro failed to reset the device status word fault bit in the default timeout period. Please check the reason for the device fault in the device manual and correct it.

ID	Explanation	Resolution
-287	Motor off request from the drive Failed during drive initialization	Please check device for errors preventing it from changing to ds402 FSTM state SWITCH_ON_DISABLED, SWITCHED_ON or READY_TO_SWITCH_ON. Refer to ELMO for support
-288	Wrong PDO length	Invalid generic PDO length request, possible generic lengths range is 1-8 bytes.
-289	CAN error interrupt	CANbus error interrupt received from CAN chip. Please check the bus state for termination problems, or try to reduce the bus load. If no device is connected on the bus, then connect a device and check if Maestro recovered from fatal error state.
-290	Inconsistency between Maestro operation mode request and device actual state	Device failed to change operation mode in the default timeout period after Maestro change operation mode request. Please check device manual for possible reasons for this inconsistency
-291	Device interpreter returned Error response	Device interpreter returned response with error bit (6th bit of the 4th byte) set. See device documentation for reasons for this error.
-292	Invalid interpreter command length	Interpreter command length is invalid. For set and get commands, the maximal length is 8 bytes. For drive program execution commands the maximal length is 80 bytes
-293	UDP message send failed	Operating system problem - please refer to ELMO for support
-294	Axis is virtual	Attempt to perform an operation that is incompatible with the virtual axis. To perform this operation, work with a real axis.
-295	"strtol" command failed	Operating system problem - please refer to Elmo for support
-296	Status word IP bit failed to zero	Status word IP bit failed to zero. Please refer to Elmo for support
-297	Resource file does not specify the IO type	Configuration resource file does not indicate what is the type of DS401 device connected to Maestro, please change your configuration.
-298	Invalid PDO group number	PDO group number is incompatible for this API, please see API documentation for allowed group numbers for this API
-299	Another reset operation is in progress for this node	Attempt to perform reset operation on a node that is already in reset process

ID	Explanation	Resolution
-300	Maestro failed to create telnet server thread	Operation system problem, please refer to ELMO for support
-301	Cannot connect between Table-Related function and Non Table-Related function types	There are two types of motion functions, Table-Related: * Spline * PVT * ECAM And Non-Table-Related: * Move[Linear]Absolute[Repetitive] * Move[Linear]Relative[Repetitive] * MoveCircularAbsolute * MovePolynomAbsolute * MoveVelocity When building the motion sequence, only use the same function block type without amalgamating them.
-302	Cannot connect two consequent polynomial functions blocks	The Maestro does not support insertion of two consequent polynomial function blocks.
-303	Abort mode not supported	This function block does not support abort mode. To interrupt the motion use the Stop\GroupStop function and then insert the motion function block in buffered mode.
-304	Bulk Read: Not supported signal	When configuring bulk read using a non-preset mode, select the requested signals and place it in the input array: stInput.uBulkReadParams.ulBulkReadParameters Each entry in this array holds a requested signal. Setting a signal that is not supported by Maestro will produce an error. Check which signals are supported by the Maestro and only set the supported signals. **NOTE:** If there are unused entries, set their places to "0".
-306	Set override IDX is out of range	Set only "0" as the "usUpdateVelFactorIdx" variable.
-307	Maestro failed to create RPC server thread	Operation system problem, please refer to ELMO for support
-308	Maestro failed to create IPC server thread	Operation system problem, please refer to ELMO for support
-309	Maestro failed to schedule thread	Operation system problem, please refer to ELMO for support
-310	Maestro failed to open a socket	Operation system problem, please refer to ELMO for support

ID	Explanation	Resolution
-311	Maestro failed to socket change socket owner	Operation system problem, please refer to ELMO for support
-312	Maestro failed to change socket option	Operation system problem, please refer to ELMO for support
-313	Maestro failed to get IP address from socket	Operation system problem, please refer to ELMO for support
-314	Maestro failed to bind a socket	operation system problem, please refer to ELMO for support
-315	Maestro failed to perform listen on a socket	Operation system problem, please refer to ELMO for support
-316	Maestro failed to perform accept on a socket	Operation system problem, please refer to ELMO for support
-317	Maestro failed to create thread attribute	Operation system problem, please refer to ELMO for support
-318	Maestro failed to detach a thread	Operation system problem, please refer to ELMO for support
-319	Maestro failed to perform accept on a socket	Operation system problem, please refer to ELMO for support
-320	Maestro failed initialize IPC server	Please refer to ELMO for support
-321	the maximal number of connections reached	The maximal number of IPC\RPC connections has reached. Please try to close some of the connections
-322	Maestro failed open EtherCAT communication	Please refer to ELMO for support
-323	Device failed to change EtherCAT state after Maestro request in the default timeout period.	Please check device manual for possible reasons or refer to ELMO for support
-324	another node initialization is in progress	Node sent two boot - up messages, the latter of the two was discarded - please check that node initialized successfully.
-325	EtherCAT AL error reset failed	Maestro failed to reset slave EtherCAT AL error. Please check the slave status and remove error reason.
-326	Maestro failed to write to slave register	Maestro attempted to perform a write to a register on the slave and failed. Please check slave status.
-327	There are no nodes on the resource file	There are no nodes on the resource file. Please create a new network configuration
-328	Attempt to set a bad sync factor	The sync factor set is incompatible with the current cycle time and object limitations. Please check the documentation for possible sync factors formula

ID	Explanation	Resolution
-329	Attempt to set a bad heartbeat factor	The heartbeat factor set is incompatible with the current cycle time and object limitations. Please check the documentation for possible heartbeat factors formula
-330	Kinematic type out of range	When setting the kinematic transformation using "MMC_SetKinTransformex" function, you can set one of the following kinematic types: * NC_CARTESIAN_TYPE = 0 * NC_DELTA_ROBOT_TYPE = 1 The kinematic types can be found in the enum definition "NC_KIN_TYPE" at "MMC_PLCopen_group_API.h" file.
-331	Delta Robot: Direct kinematic failed	Please ensure that the mechanic inputs of the delta robot are properly set. Note that the mechanic inputs and the target position should be inserted in [um] units. For more info, refer to Elmo.
-332	Delta Robot: inverse kinematic failed	Please ensure that the mechanic inputs of the delta robot are properly set. Note that the mechanic inputs and the target position should be inserted in [um] units. For more info, refer to Elmo.
-333	PCS coordinate system is not set	If you desire to insert the FB in PCS mode, you must to define the Cartesian transformation. The Cartesian transformation is defined using "MMC_SetCartTransform" function.
-334	Delta Robot: one of the theta kinematic directions are missing	When you set configuration for delta robot kinematic, you should choose all the theta kinematic directions in your kinematic definition and map them to real axes. The theta axes are: * Thetha1 - NC_ACS_A1_AXIS_TYPE * Thetha2 - NC_ACS_A2_AXIS_TYPE * Thetha3 - NC_ACS_A3_AXIS_TYPE
-335	Delta Robot: one of mechanic values are negative or equal zero	When you set configuration for delta robot kinematic, you should enter the mechanical parameters of the robot. All the mechanical parameters should be a positive numbers. Note that the mechanic inputs and the target position should be inserted in [um] units.
-336	Delta Robot: the ratio between mechanic values are wrong	When you set configuration for delta robot kinematic, you should enter the mechanical parameters of the robot. * (R_base - R_endeffector)/Arm Higher than 0.01 * ForeArm/Arm Higher than 1.75 * (R_base - R_endeffector + Arm) Lower than ForeArm

ID	Explanation	Resolution
		Note that the mechanic inputs and the target position should be inserted in [um] units.
-337	Cannot get validation function	Current combination of Value type and Operation type is not supported. This is the list of supported combinations: * Equal - char\uchar\short\ushort\int\uint\float\double * Higher - char\uchar\short\ushort\int\uint\float\double * Lower - char\uchar\short\ushort\int\uint\float\double * Lower Equal - char\uchar\short\ushort\int\uint\float\double * Higher Equal - char\uchar\short\ushort\int\uint\float\double * Mask AND - char\uchar\short\ushort\int\uint\
-338	Operation type out of range	The current operation mode is not supported. This is the list of supported types: * Equal * Higher * Lower * Lower Equal * Higher Equal * Mask AND
-339	Not support global parameter	When you are using condition FB, use only axis\group related parameters.
-340	Parameter not fit to node type	When you are using condition FB, make sure that the type of the parameter fit to type of the reference node. When reference is Axis - use only axis related parameters. When reference is Group - use only group related parameters.
-341	Parameter type error	Refer to Elmo for support.
-342	High error correction value in table (NC_ERROR_TABLE_HIGH_CORRECTIO N_VALUE)	Each correction value in the table cannot be higher than the "dMaxCorrectionDelta" variable, inserted at the "mmc_loaderrortablefromfile" function input. When "dMaxCorrectionDelta" variable is set to zero, then there is no limit for the correction values.
-343	EtherCAT register read failure	Maestro failed to read from an EtherCAT register, please check the device EtherCAT state
-344	Exceeded maximal register length	you have tried to read\write a range of memory which is larger than the allowed range.
-345	PI parser wrong attribute	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)

ID	Explanation	Resolution
-346	PI parser Variable offset too large	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-347	PI parser Variable offset duplicated	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-348	PI parser ElmoPI element was not found	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-349	PI parser bitsize too large	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-350	PI parser mismatching variable number	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-351	PI parser inputs duplicated	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-352	PI parser outputs duplicated	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-353	PI parser inputs or outputs element is missing	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-354	PI parser wrong variables number	EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-355	PI manager exceeded max memory	The number of memory needed for current EtherCAT configuration is larger than the maximal memory allowed. Please create new EtherCAT configuration with less or smaller process image variables
-356	PI manager - index too large	you have requested to access a PI variable which does not exist. Try a smaller index

ID	Explanation	Resolution
-357	PI manager - invalid direction	The allowed direction are ePI_INPUT or ePI_OUTPUT, you have tried an invalid direction
-358	PI manager - incompatible bit size	You have tried to access a PI variable that is too large\ too small for the current API. Please use the proper API
-359	change of operation mode FB: not all variables were mapped	You have attempted to perform a command which requires mapping of one or more variables which are not mapped. in order to use ChangeOpMode FB API you need to map the following objects: 0x6060, 0x6061, 0x60B1, 0x60FF, 0x6040, 0x6041, 0x607A, 0x6064 if you wish to exit operation mode cyclic torque you also need to map 0x6071 (target torque)
-360	slave failed to change state in the allowed timeout period	The EtherCAT slave failed to change its state in the allowed timeout period, please check the slave status
-361	PI parser - mismatching var type	One of the variables has a variable type incompatible with its bit size EtherCAT master XML corrupted. Please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-362	No nodes are physically connected to the network	No nodes are physically connected to the network, please connect at least one node
-363	Wrong kinematic direction	Only following kinematic directions are acceptable: NC_PROFILER_X_AXIS_TYPE =0 NC_PROFILER_Y_AXIS_TYPE = 1 NC_PROFILER_Z_AXIS_TYPE = 2 NC_PROFILER_U_AXIS_TYPE = 3 NC_PROFILER_V_AXIS_TYPE = 4 NC_PROFILER_W_AXIS_TYPE = 5 NC_PROFILER_N1_AXIS_TYPE = 6 NC_PROFILER_N2_AXIS_TYPE = 7 NC_PROFILER_N3_AXIS_TYPE = 8 NC_PROFILER_N4_AXIS_TYPE = 9 NC_PROFILER_N5_AXIS_TYPE = 10 NC_PROFILER_N6_AXIS_TYPE = 11 NC_PROFILER_N7_AXIS_TYPE = 12 NC_PROFILER_N8_AXIS_TYPE = 13 NC_PROFILER_N9_AXIS_TYPE = 14 NC_PROFILER_S_AXIS_TYPE = 15 NC_MCS_X_AXIS_TYPE = 16 NC_MCS_Y_AXIS_TYPE = 17 NC_MCS_Z_AXIS_TYPE = 18 NC_MCS_U_AXIS_TYPE = 19 NC_MCS_V_AXIS_TYPE = 20 NC_MCS_W_AXIS_TYPE = 21 NC_ACS_A1_AXIS_TYPE = 22 NC_ACS_A2_AXIS_TYPE = 23 NC_ACS_A3_AXIS_TYPE = 24

ID	Explanation	Resolution
		NC_ACS_A4_AXIS_TYPE = 25 NC_ACS_A5_AXIS_TYPE = 26 NC_ACS_A6_AXIS_TYPE = 27
-364	Cannot set User Units ratio in current state	The User Units Ration can be set only in Disabled state
-365	One of the members has different Position_To_Kinematics ratio	In order operate with Group, all the members should have the same Position_To_Kinematics ratio. The Position_To_Kinematics is calculated using Position and Kinematics UU ratio. Position_To_Kinematics = Position_UU/Kinematics_UU
-366	Cannot set User Units ratio when the axis is part of active group	In order to set new User Units ratio, switch off the group and axis to Disable state and then try again.
-367	Cannot insert to consequential Repetitive Function Blocks	In order to terminate the repetitive motion use one of the following options: * Stop * Halt * Kill Repetitive Then try to insert the new Repetitive motion again
-368	Cannot insert Motion Function Block when repetitive motion is active	In order to insert the new motion, terminate the repetitive motion and then try to insert the new motion again. In order to terminate the repetitive motion use one of the following options: * Stop * Halt * Kill Repetitive
-369	Velocity out of range – DS-402	The maximum velocity value in NON-NC mode is 2^{32}
-370	Acceleration out of range – DS-402	The maximum acceleration value in NON-NC mode is 2^{32}
-371	Deceleration out of range – DS-402	The maximum deceleration value in NON-NC mode is 2^{32}
-372	Position out of range – DS-402	The maximum position value in NON-NC mode is 2^{32}
-373	Set position - wrong node type	Set position can be operated only for DS-402 node type
-374	Set position - wrong state	Set position can be operated only in one of the following node states: * Axis in Disabled state * Axis in StandStill and the FB queue is not contain motion function blocks
-375	Set position - not support queued mode	Refer to Elmo for support.
-376	Modulo - contradiction between high and low values	In order to set modulo range, set the following parameters: * MMC_MODULO_LOW - 100 * MMC_MODULO_HIGH – 101 The "MMC_MODULO_HIGH" parameter should be always higher than "MMC_MODULO_LOW" parameter.
-377	Direction mode is not supported	When using MoveVelocity FB, only Positive and negative directions are supported.

ID	Explanation	Resolution
-378	Direction mode is out of range	Only 4 direction modes are supported: * Positive direction = 1 * Shortest way = 2 * Negative direction = 3 * Current direction = 4
-379	Wrong error correction range for modulo axis	When using rotation axis as reference axis for error correction table, the table range and modulo range should be same.
-380	Change modulo axis type error	In order to change the modulo axis type, the axis should be in Disabled state
-381	Execution mode is out of range	Set only one of the following execution modes: * eMMC_EXECUTION_MODE_IMMEDIATE = 0 * eMMC_EXECUTION_MODE_QUEUED = 1
-382	Cannot set current operation mode for virtual axis	When you work with virtual axis you can set only the following operation modes: For CAN: OPM402_INTERPOLATED_POSITION_MODE = 7 For EtherCAT: • OPM402_CYCLIC_SYNC_POSITION_MODE = 8 • OPM402_CYCLIC_SYNC_VELOCITY_MODE = 9 • OPM402_CYCLIC_SYNC_TORQUE_MODE = 10
-383	Operation mode is out of range	Please set only one of the following operation modes: • OPM402_PROFILE_POSITION_MODE = 1 • OPM402_PROFILE_VELOCITY_MODE = 3 • OPM402_TORQUE_PROFILE_MODE = 4 • OPM402_HOMING_MODE = 6 • OPM402_INTERPOLATED_POSITION_MODE = 7 • OPM402_CYCLIC_SYNC_POSITION_MODE = 8 • OPM402_CYCLIC_SYNC_VELOCITY_MODE = 9 • OPM402_CYCLIC_SYNC_TORQUE_MODE = 10
-384	Cannot set requested operation mode when the axis is in pending state	Please try to change the operation mode when the axis is not pending state. In order to take the axis out from pending state, the last motion function block should be inserted with execute bit "1".
-385	Queued change of operation mode cannot be done for CAN drive	When the field bus is CAN you cannot use the "ChangeOpModeFB" in queued mode. Use it only in Immediate mode.

ID	Explanation	Resolution
-386	Queued change of operation mode cannot move to IP mode	When you are using "ChangeOpModeFB" for EtherCAT axis, set only operation mode that fit to EherCAT axis. • OPM402_PROFILE_POSITION_MODE = 1 • OPM402_PROFILE_VELOCITY_MODE = 3 • OPM402_TORQUE_PROFILE_MODE = 4 • OPM402_HOMING_MODE = 6 • OPM402_CYCLIC_SYNC_POSITION_MODE = 8 • OPM402_CYCLIC_SYNC_VELOCITY_MODE = 9 • OPM402_CYCLIC_SYNC_TORQUE_MODE = 10
-387	One of the Group members is in pending state	The group cannot be Enabled when at least one of the axes are in pending state. Remove the entire function blocks from all group members and try again to enable the group. In order to clean the function block queue use one of the following options: * MMC_Stop * MMC_ClearNodeFbList
-388	PI parser - known PI variable wrong bit size and type	One of Maestro known PI variables has an incorrect bit size and type EtherCAT master XML corrupted, please create new EtherCAT configuration Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-389	Target Torque is not mapped (0x6071)	Target torque is not mapped, it is essential to map this object in order to change to motion mode cyclic torque. You should map the target torque object (0x6071) in the EASII configurator.
-390	Motion mode is not supported by drive	You have attempted to change the drive motion mode to a motion mode it does not support according to object 0x6502
-391	Motor ON time out	Drive failed to respond to an initialization motor on request from Maestro during a timeout period. Try to reset the drive and EtherCAT configuration
-392	Abort attempt when a change operational mode FB is in the queue	Abort FBs are not allowed when a change operational mode function block is in the queue
-393	Function obsolete	The requested function is obsolete and is no longer supported
-394	Requested target torque is out of range	Please check the MMC_MAX_DESIRED_TORQUE_PARAM parameter and do not exceed its value range.
-395	Requested target torque is out of DS402 range	The torque value cannot be higher than 16bit value (after conversion to rated current). In order to convert the torque to rated current you should use a convertion ratio that can be obtained from the 0x6075 object.
-396	Torque velocity is out of range	The torque velocity range is between 0 and MMC_MAX_TORQUE_VELOCITY_PARAM parameter.
-397	Torque acceleration is out of range	The torque acceleration range is between 0 and MMC_MAX_TORQUE_ACCELERATION_PARAM parameter.

ID	Explanation	Resolution
-398	Zero rated current ratio	Refer to Elmo for support.
-399	Not support blended motion	When you are using move torque command you can set only two types of buffered modes: * MC_ABORTING_MODE = 1 * MC_BUFFERED_MODE = 2 Blended modes are not supported.
-400	Change operation mode for pending axis is forbidden	When you are using "ChangeOpModeFB" function the axis should be in non-pending state. In order to exit from pending state, clean the function block queue using one of the following options: * MMC_Stop * MMC_ClearNodeFbList If it is part of a sequence, insert the last motion function block with execute bit "1".
-401	Wrong stop function ID	Refer to Elmo for support.
-402	Immediate change operational mode operation is in progress	Function block insertion and other set operational mode operations are forbidden while an immediate change operational mode operation is in progress
-403	Failed to retreive EtherCAT slave state	Maestro failed to retreive an EtherCAT slave state. EtherCAT network problem, please verify the EtherCAT network connections or try to reboot the slaves.
-404	PI parser - Alignment object is present on the ElmoPI section	EtherCAT master XML corrupted, please create new EtherCAT configuration. Older versions of the Elmo Application Studio do not support this Maestro version. Make sure you are working with the latest release of Elmo Application Studio (EASII)
-405	DS402 drive control variables are not mapped	You have created a configuration of a DS402 device but the mandatory variables: ControlWord (0x6040) and\or StatusWord (0x6041) are not mapped. Please create a new EtherCAT configuration and map these variables or change the device type to DS301
-406	Position control variables are not mapped	Actual position (0x6064) and\or Target position (0x607A) are not mapped. Please create a new EtherCAT configuration and map these variables in order to perform this operation.
-407	No inputs or outputs	You have created a configuration without any inputs or without any outputs, this configuration is forbidden Please create a new EtherCAT configuration and map at least one input and one output.
-408	Target velocity (0x60FF) is not mapped	You cannot perform movevelocity API while Target velocity is not mapped, please create new EtherCAT configuration and map this variable The only mode this API supports while target velocity is not mapped is cyclic position
-409	Homing attained bit is on	Home immediate (method 35) is not supported after a previously performed homing without re - entering homing operation mode or power on/off
-410	Homing detection time limit out of range	according to the DS402 standards the Detection time limit object (0x2020_5) is limited to 32000
-411	Profile torque not supported	DS402 profile torque is not supported. Please use another

ID	Explanation	Resolution
		DS402 profile (cyclic torque is advised).
-412	Cannot request master state	Your attempt to request the EtherCAT master state has failed. This is an EtherCAT network problem, please check the network state.
-413	Cannot scan EtherCAT network	Your attempt to scan the EtherCAT network has failed. This is an EtherCAT network problem. Please check the network state.
-414	EtherCAT slave is in Application Layer (AL) error	An EtherCAT slave is in AL error. This is an EtherCAT network problem, please reboot or remove the problematic slave. If the problem remains try to update the slave ESI file and create new EtherCAT configuration.
-415	Operation is in progress	You have attempted to perform an operation while the previous operation is in progress. Please wait until the operation is completed.
-416	Finite State Machine (FSTM) Power interruption	An interruption occured during the power FSTM. Please attempt to perform the operation at a later stage
-417	PI additional data corrupted	A problem occured during the parsing of PI variable data. Please create a new ETherCAT configuration
-418	CAN bus is passive	The CAN bus chip received an interrupt and is now in passive mode. Please check the CAN network status and repair in order to recover
-419	CAN bus is in overrun state	The CAN bus is in overrun state, please change the CAN network configuration in order to reduce the bus load.
-420	Node was reinitialized	A node returned to the network and was reinitialized and must be reset.
-421	PI variable was not found	Requested PI variable is not present on the current configuration. Please check if the input string is correct.
-422	The number of requested PI variables is too large	You have requested information regarding a number of PI variables which is too large. Please reduce the number of PI variables requested.
-423	Provided index is invalid	Please provide valid First and last index. Last index must be larger then the first index
-424	Cannot perform requested operation with the provided PI variable	Not all types of PI variables are available for this operation, please use another variable.
-425	Immediate write not supported	Write group of parameter can be used with PI varaibles only in queued mode. in order to write a PI varaible you may use the standard write PI variables function (single variable)
-426	Ilegal operation for a BOOL variable	BOOL operations are limited. Please use a proper operation (see documentation).
-427	PI Large variables recording invalid input	A first part of a large varaible was configured, but a second part was not consecutive. Large variables should occupy two entries.

ID	Explanation	Resolution
-428	Invalid direction of the trigger	The type of trigger does not match the PI direction
-429	Given Floating point number is invalid	Invalid floating point number was entered, please insert a valid floating point number.
-430	Invalid trigger type for BOOL variable	Not all trigger types are valid for BOOL variables, please use one of the "mask" type triggers.
-431	User command fail to start	Please make sure that the requested user program is of type gexe, and is present in /mnt/jffs/usr.
-432	User command failed to stop	Please make sure that the requested user program is of type gexe, and is present in /mnt/jffs/usr.
-433	User command failed to remove file	Please make sure that the requested user program is of type gexe, and is present in /mnt/jffs/usr.
-434	User command cannot run more than five programs	When using a User command you cannot start running more than five programs.
-435	Touch probe counter zero value only	The Touch probe counter parameter can only be set to zero.
-436	Cannot switch the motor ON for the axis. Working with modulo 2 and Drive actual position is not in modulo range (modulo low to modulo high).	Make sure to set the Drive position in the modulo range before enabling the axis.
-437	No variable configured for PI bulk read	You have tried to configure bulk read for PI variables but no variables where configured. Please specify at least one variable with PI direction ePI_INPUT or ePI_OUTPUT.
-438	Duplicate node ID was found in the Maestro resource file	Maestro resource file is corrupted, please create a new configuration.
-439	Duplicate node name was found in the Maestro resource file	Maestro resource file is corrupted please create a new configuration.
-440	Memory not allocated	Memory of destination is not allocated
-441	System call for sending an SDO failed	Verify EtherCAT configuration and network status
-442	Only DS-402 nodes can be stopped	Do not register a stop policy on a node which is not a DS-402 node
-443	Only DS-402 nodes can be powered off	Do not register a power off policy on a node which is not a DS-402 node
-444	Wrong error type was selected	The selected error type does not exist, please select another error type
-445	Error handling in progress	An error is currently being handled, please retry after error handling policy is applied.
-446	Configuration is not valid	Current network configuration is not valid, please check the network topology. Verify slaves identification, check for redundant\missing slaves. Create new EtherCAT configuration if you wish to mainitain this physical configuration.
-447	This error policy cannot apply all	Policy for this error cannot apply all

ID	Explanation	Resolution
-448	Wrong slave identity	The slave connected is different from the slave that was configured. Disconnect this slave and connect the original one. If you wish to work with this slave you have to create a new EtherCAT configuration.
-449	System error occured	System error occured (see Error handling APIs), in order to recover you need to invoke the MMC_ResetSystem API.
-450	The error is fatal and cannot be recovered	The GMAS is in error state which is unrecoverable by the MMC_ResetSystem function. Please check the configuration and the system.
-451	Threshold of selected error cannot be changed	Do not try to change the threshold of the selected error.
-452	You have tried to get\register too many members.	The number of errors to be registered\get are limited please reduce the number of requested errors.
-453	Node is not connected	Please connect the node to the network, if node is already connected please wait for node connected event
-454	Node is in error	The node has entered ERROR STOP state and therefore policy cannot be continued.
-455	Policy applied partially	System policy was applied but the Maestro was unable to perform all policy actions on all nodes. See nodes parameters POLICY_FAIL_CODE and POLICY_FAILURE_STATE .
-456	Policy of only event cannot apply all	When registering a policy for a node error, if the only policy is an event, you cannot apply this policy to all axes, please zero the "apply all" policy bit .
-457	Power off must be registered with stop	When registering the power off policy, stop policy has to be registered as well
-458	Cannot perform SAFEOP policy on CAN communication	SAFEOP policy is relevant only when the communication type is EtherCAT. You may not use this policy when the communication type is CAN.
-459	No reaction policy is not allowed for this error	Registering a policy of no response for this error is not allowed, please register at least one active policy .
-460	Couldn't disable group	GMAS failed to disable the group
-461	FoE operation is in progress	You cannot perform this operation while an FoE operation is in progress. Please wait for the operation to finish and retry.
-462	Max cycles exceeded timeout	(STATE_TRANSITION_CYCLES + AL_ERROR_RESET_CYCLES) * (mailbox cycle time) + 2 * 0.1 * MOTOR_ON_CMD_MAX_TIMEOUT_MS will not exceed the library timeout (20 seconds)
-463	Error is relevant for DS402 nodes only	You are trying to register a motion related policy for a non DS402 node. The error policy is not relevant for this node, please verify that you used the proper axis ref.
-464	Policy does not exist	You are trying to set a policy that does no exist. Please zero redundant policy bits and retry.

ID	Explanation	Resolution
-465	Maestro is initializing	Maestro is currently initializing, please try after the parameter GMAS_INIT_STATE value is 1.
-466	Maestro is in maintenance mode	When in maintenance mode not all Maestro features are supported, please exit this mode in order to use all functionalities.
-467	SAFEOP policy can only apply on system errors	Registering the SAFEOP policy is possible only for system errors (or when Apply to all policy is selected)
-468	Ports open timeout	Slaves ports failed to open in the allowed timeout, check the network topology and verify vs the Ethercat configuration.
-469	Bulk read double variable low high parts mismatch	When perfroming bulk read for a variable of type double, you must always put both parts on the buffer with the low part preceeding the high part.
-470	Bulk read - wrong variable type	Variables of this type are not supported for Bulk read, please use another method to read the variable.
-471	Power state machine is in progress	Operation cannot be performed while Power state machine is in progress, please wait until the power operation is finished.
-472	EtherCAT Master XML version Mismatch	EtherCAT master XML is not compatible for current Maestro version please create new EtherCAT Configuration via EASII.
-473	Failed to create EtherCAT master	Please create new EtherCAT configuration
-474	Failed to connect EtherCAT master	Please create new EtherCAT configuration
-475	Failed to set EtherCAT auto recovery timeout	Please create new EtherCAT configuration
-476	Failed to stop EtherCAT cyclic operation	Please create new EtherCAT configuration
-477	Failed to start EtherCAT cyclic operation	Please create new EtherCAT configuration via EASII.
-478	Failed to parse EtherCAT master XML, file currupted.	Please create new EtherCAT configuration via EASII.
-500	ECAM error - table not selected	CAM table has not been selected. One must invoke MC_CamTableSelect at least once before MC_CamIn invocation.
-504	ECAM error - invalid node type	Group is not supported on this phase.
-506	ECAM error - invalid monotonic	Monotonic on master column must be either ascending or descending.
-507	ECAM error - none modifiable table.	An attempt to modify read only table failed. One cannot modify read only CAM table.
-508	ECAM error - invalid scaling parameter.	Scaling parameters cannot be zero(0)
-517	ECAM error - invalid master distance parameter.	Master distance must be zero at this phase.

ID	Explanation	Resolution
-518	ECAM error - invalid master sync position.	Master sync position must be defined relatively within the range of master positions in CAM table.

4.8 NC Driver Warning IDs

ID	Explanation	Resolution
2000	Drive warning: The Axis is already in Power OFF mode	Only run the PowerOFF function when the axis is not in PowerOff state. To read the current status of the axis call the MMC_ReadStatusCmd function
2001	Drive warning: The Axis is already in Power ON mode	Only run the PowerON function when the axis is not in PowerOn state. To read the current status of the axis call the MMC_ReadStatusCmd function
2002	The Axis is already in current Operation mode	Do not run the "Change Operation Mode" function when the axis is presently active in the same operation mode.
2003	The Axis TouchProbe is already in Enabled state	Do not run the TouchProbe Enable function repeatedly.
2004	The Axis TouchProbe is already in Disabled state	Do not run TouchProbe Disable function repeatedly. When the touch probe is triggered, the state changes automatically to disabled state.

4.9 NC Profiler Error IDs

The following table lists the NC profiler error IDs. Many of these errors are a result of human programming errors, although some may be caused by limitations of the Maestro itself. In most situations where an error of the type below is produced, it may be possible to overcome the error without the necessity to reset the system.

Error	Explanation	Resolution
-1000	Error to get active FB ptr	Try to execute the process again (Internal process failure).
-1001	Error to get next FB ptr	Try to execute the process again (Internal process failure).
-1002	Error to get previous FB ptr	Try to execute the process again (Internal process failure).
-1003	Error to get last FB ptr	Try to execute the process again (Internal process failure).
-1004	Error to get previous FB ptr on reverse flow	Try to execute the process again (Internal process failure).
-1005	Error to get last FB ptr on reverse flow	Try to execute the process again (Internal process failure).
-1006	Error to get next FB ptr on reverse flow	Try to execute the process again (Internal process failure).
-1007	Error to get previous FB ptr (Reject MSEP)	Try to execute the process again (Internal process failure).
-1008	Error to get next FB ptr (Reject MSEP)	Try to execute the process again (Internal process failure).
-1010	Get active FB pointer rejected	Try to execute the process again (Internal process failure).
-1011	Cannot get FB by number in the queue	Try to execute the process again (Internal process failure).
-1012	The FB pointer that entered to profiler is NULL	Try to execute the process again (Internal process failure).
-1021	Linear Segment length is zero (No longer relevant from version 1031)	This error only relates to Maestro MultiAxes linear functions, where the Maestro does not support zero length movement. Verify that your destination location is not equal to the start position, or very close to the start location (less than 1E-6). In a group of axes, one of the axes must move a discrete distance, while the others may remain static. To get the current position, call the function MMC_ReadActualPositionCmd.
-1022	Circular Segment length is zero (No longer relevant from version 1031)	This error only relates to MultiAxis circular functions, where the Maestro does not support zero length movement. Verify that your destination location is not equal to the start position, or very close to the start location (less than 1E-6). In a group of axes, one of the axes must move a discrete distance, while the others may remain static. To get the current position, call the function MMC_ReadActualPositionCmd.
-1023	Segment length is zero on ACS mode (No longer relevant from version	This error related only to MultiAxis functions in ACS mode, where the Maestro does not support zero length movement.

Error	Explanation	Resolution
	1031)	Verify that your destination location is not equal to the start position, or very close to the start location (less than 1E-6). In a group of axes, one of the axes must move a discrete distance, while the others may remain static. To get the current position, call the function MMC_ReadActualPositionCmd.
-1024	Single axis Segment length is zero (No longer relevant from version 1031)	This error only relates to SingleAxis functions, where the Maestro does not support zero length movement. Verify that your destination location is not equal to the start position, or very close to the start location (less than 1E-6). To get the current position, call the function MMC_ReadActualPositionCmd.
-1026	3D circle mode is not defined properly	For circular movement functions. When using a MC_CENTER_CIRC_MODE circle mode, the circular motion "PI" or "2PI" (180 or 360 degrees) cannot be created.
-1027	Contradiction between circle params	This error is caused by inputting improper parameters to the circular function. If you are using the circular radius function: The auxiliary point is a radius vector (value and direction). The radius vector must be orthogonal with a vector created between the start point and the end-point of the circle (circle surface). The maximal permitted deviation (result of dot product between radius vector and a circle surface) is 1E-2 [user units]. If you are using circular center function: The auxiliary point is a center point of the circle. The center point of a circle must be at the same distance from start and end-points, meaning, constant radius. The maximal permitted deviation of these distances is 1 [user unit]. **NOTE:** The definition of a circle is where each point on the perimeter of a circle is the same distance from the center.
-1028	Profiler Error: Transition mode is not supported	This function does not support any transition modes. Set the buffer mode to "MC_BUFFERED_MODE" and transition mode to "MC_TM_NONE_MODE"
-1029	Obtain negative value in CalcMaxPossibleVendOpt()	Try to execute the process again (Internal process failure).
-1030	Profiler Error: Small distance between start and end point in Circular Radius mode	This error originates from the MoveCircular function in Radius mode. The length of the straight line between the start and end-

Error	Explanation	Resolution
		points cannot be less than $1e^{-9}$ [user units] The length is defined by: $\sqrt{(End[0]-Start[0])^2+(End[1]-Start[1])^2+(End[2]-Start[2])^2}$
-1031	Cross product between radius and StartToEnd point vector is very small	This error originates from the MoveCircular function in Radius mode. The cross product between the radius vector defined in the auxiliary point, and the vector between the start and end-points are very small. The minimal permitted value for the cross product is $1e^{-9}$ [user units]. Increase the distance between the start and end-points, or increase the radius vector length.
-1032	Radius vector length is zero (Radius mode)	This error originates from the MoveCircular function in Radius mode. The length entered for the Auxiliary point vector must be larger than $1e^{-9}$ [user units] To calculate the radius length use the formula: $\sqrt{\mathbf{Aux}[0]^2+\mathbf{Aux}[1]^2+\mathbf{Aux}[2]^2}$
-1034	The radius of the circle is very small (center or angle mode)	This error originates from the MoveCircular function in either Center or Angle mode. In these modes, when you enter the input parameters to the function, you are defining the end-point and the center of the circle point. You cannot enter parameters that create a circle with radius smaller than $1e^{-9}$ [user units] To calculate the radius of the circle use the formula: $\sqrt{(\mathbf{Aux}[0]-\mathbf{Start}[0])^2+(\mathbf{Aux}[1]-\mathbf{Start}[1])^2+(\mathbf{Aux}[2]-\mathbf{Start}[2])^2}$
-1035	Cannot calculate two possible centers in circle radius mode	Refer to Elmo for support.
-1039	Zero max velocity on current FB	The Maestro cannot operate a function block where the maximum velocity is zero. The current function block will cause no movement of an axis. Enter the movement function with a non-zero length movement.
-1040	Small Angle parameter	This error originates from the MoveCircular function in Angle mode.

Error	Explanation	Resolution
		In this mode, the Angle input should be inserted as a value to the Aux[N] vector, in the "0" location. The units of the angle are degrees. For instance, if you wish to create a full circle, you should enter the following value: Aux[0] = 360; The minimum permitted absolute (can be negative) value (|x|) of the angle is 1 degree. Make sure you enter an angle with a value above the minimum limit.
-1041	Sweep angle equal to 180° or 360°	This error originates from the MoveCircular function in center mode. In this mode, circular motion cannot operate near to values of 180 or 360 degrees. The exact forbidden area is: * 180° ± 0.58° * 360° ± 0.58°
-1043	Cannot operate Circle Angle function in non 2D mode	When using Circle Angle function, the exact number of axes in the group must be 2.
-1045	Profiler Error: Two points are very close	This error originates from the MoveCircular function in Border mode. In this mode three points are defined: * Start point SP * Border point BP * End point EP Prior to running the function block, check that each set of these points are not too close to each other (SP vs. BP, SP vs. EP, BP vs. EP). If all of the points are close together, an error will be created. The logic of close proximity checking: If((ABS(Point[1].x - Point[2].x) <$1e^{-7}$) && (ABS(Point[2].y - Point[3].y) <$1e^{-7}$) && (ABS(Point[1].z - Point[3].z) <$1e^{-7}$)) Then, the points are very close (not permitted) Else, the points are OK (permitted)

Error	Explanation	Resolution
-1046	Three points on the same line	This error originates from the MoveCircular function in Border mode. In this mode, the three points Start point(SP), Border point(BP), and End point(EP), are defined. Prior to running the function block, make sure that the Start point(SP), Border point(BP), and End point(EP) are not located on the same straight line (two points define the line). If they are located on the same line, an error is created.
-1047	Profiler Error: Operate with empty vector	When trying to select a spline path to a specific vector, the vector should contain at least one member.
-1048	Profiler Error: Operate with a large vector	When trying to select a spline path to a specific vector, the vector should contain no more than 16 members.
-1050	Transition curve cannot be inserted in Line-Line mode - angle between two lines close to 0°	The profiler can only insert a transition curve between two lines when the angle between the lines is greater than 0.06°.
-1051	Transition curve cannot be inserted in angle between two lines close to 0°	The profiler can only insert a transition curve between two lines when the angle between the lines is greater than 0.06°.
-1052	Transition curve cannot be inserted in angle between two lines close to 180°	The profiler can only insert a transition curve between two lines when the angle between the lines is less than 179.94°.
-1053	Line Circle transition curve not created	The end point of the first function block does not intersect with the start point of the second function block. Maximum permitted deviation between these two points is $1e^{-3}$ [user units]
-1054	Circle Line transition curve not created	The end-point of the first function block does not intersect with the start point of the second function block. Maximum permitted deviation between these two points is $1e^{-3}$ [user units]
-1055	Cannot create a "Switch Radius" transition curve between two Circles	The end-point of the first function block does not intersect with the start point of the second function block. Maximum permitted deviation between these two points is $1e^{-3}$ [user units]
-1056	Cannot create a Spline profile with a short length between points	Verify that the radius of the first Circle is not equal to the radius of the Switch Radius parameter (transition parameter). If these radii are equal, change the value of one of them. If not, refer to Elmo for support.

Error	Explanation	Resolution
-1057	Cannot create a Spline profile with zero time execution between points	When using Spline profiles, make sure that the minimal length between two consecutive points is 1 [user units]. Make sure that the execution time between a movement of two consecutive points is greater than zero. Check that the segment length is not zero. The segment length is defined by: $\sqrt{\begin{matrix}(\mathrm{End}[0]-\mathrm{Start}[0])^2+(\mathrm{End}[1]-\mathrm{Start}[1])^2\\ +(\mathrm{End}[2]-\mathrm{Start}[2])^2\end{matrix}}$ It is recommended not to allow the velocity to be too high.
-1058	Cannot create a transition curve in Circle-Line profile with the given transition parameters	The radius of the transition curve is too large. The radius of the transition curve must be smaller than half the circle radius in the next function block. Change the transition parameter to a smaller value.
-1059	Cannot create a transition curve in Line-Circle profile with the given line length	The radius of the transition curve is too large. The radius of the transition curve must be smaller than half the circle radius in the previous function block. Change the transition parameter to a smaller value.
-1060	Cannot create a transition curve in Circle-Line profile with the given line length	Maestro cannot create a transition curve because the line length in the first function block is less than 1 [user units].
-1061	Cannot create a transition curve in the profile with the given input parameters	Maestro cannot create a transition curve because the line length in the second function block is less than 1 [user units].
-1062	Cannot create a transition curve in the profile with the given input parameters	Maestro cannot create a transition curve because the transition curve does not intersect with the circle. Refer to Elmo for support.
-1063	Cannot create a transition curve in the profile with the given input parameters	Maestro cannot create transition curve because the transition curve does not intersect with the line. Refer to Elmo for support.
-1064	Cannot create a transition curve with the given transition parameter	Maestro cannot create transition curve because the transition curve does not intersect with the line. Refer to Elmo for support.
-1065	Cannot create a transition curve with a small radius in the first function block	The transition parameter used for this mode is inappropriate. To create a transition curve between two function blocks using the transition mode MS_TM_SWITCH_RADIUS_MODE, the minimum value of the transition parameter is 1 [user units]
-1066	Cannot create a transition curve	To create a profile with a Circle-Line transition curve, verify that the radius of the first function block (circle motion) Is

Error	Explanation	Resolution
		larger than $1e^{-6}$ [user units]
-1067	Cannot create a Line-Circle transition curve in SwitchRadius mode	Operation cannot be executed with the input parameters.
-1068	Transition curve cannot be inserted in Line-Line mode - angle between two lines close to 0°	Operation cannot be executed with the input parameters.
-1069	Not in use	Not in use
-1070	Cannot create a Line-Circle transition curve	The start point of the transition curve does not intersect with the line in the first function block. Refer to Elmo for support.
-1071	Calculation error of the Line-Circle transition curve	Try to execute the process again (Internal process failure). Operation cannot be executed with the input parameters.
-1072	Cannot create Line-Circle transition curve in SwitchRadius mode	Operation cannot be executed with the input parameters.
-1073	Calculation error in the Circle-Circle transition curve	Operation cannot be executed with the input parameters.
-1074	Calculation error in the Circle-Line transition curve	Operation cannot be executed with the input parameters.
-1075	Cannot allocate the spline	Check the format of the Spline input file. If you find a problem in the file, correct it and run the function again. If you cannot locate the problem, try reducing the number of points in the input file.
-1076	Cannot de-allocate the spline	Verify that the current handle is not already de-allocated.
-1077	Error in the Spline input file	Verify that the spline input file is present in the given path. Verify that the format of the file is correct. If the above input file checks are OK, then the error is otherwise. Refer Elmo for support.
-1078	One of the input parameters is wrong	Check that the number of axis members in the group is the same as the spline dimension parameter in the input file. Check whether the spline mode value is allowed. Verify that the axis handle does not exceed the maximum of 20.
-1079	Spline allocation error	If you attempting to move path, first run the MMC_PathSelectCmd function. If you attempting to unselect the path using the MMC_PathUnselectCmd function, perform it once only.
-1080	Spline memory handle is out of	The spline memory handle should have a value

Error	Explanation	Resolution
	range	between 1 - 19.
-1081	Cannot create Spline motion with less than 3 points	Create Spline input file with more than 2 points.
-1082	Profiler Error: Auxiliary point to close to Start point	When creating a polynomial segment, always ensure that the distance between Auxiliary and Start point is at least 1E-6. The distance between the points calculated as vector distance between auxiliary and start vector. Distance = $\sqrt{((Aux[0] - Start[0])^2 + (Aux[1] - Start[1])^2 \dots \dots (Aux[n]}$
-1083	Start point to close to End point	When creating a polynomial segment, always ensure that the distance between Start and End point is at least 1E-6. The distance between the points calculated as vector distance between start and end vector. Distance = $\sqrt{((Start[0] - End[0])^2 + (Start[1] - End[1])^2 \dots \dots (Start[n] - End[n])^2)}$
-1084	Auxiliary point to close to End point	When creating a polynomial segment, always ensure that the distance between Auxiliary and End point is at least 1E-6. The distance between the points calculated as vector distance between auxiliary and end vector. Distance = $\sqrt{((Aux[0] - End[0])^2 + (Aux[1] - End[1])^2 \dots \dots (Aux[n] -}$
-1085	High ratio between polynomial sub-segments	When creating a polynomial segment, define the three points: * Start point * Auxiliary point * End point These points define two sub-segments * The length of the straight line between Auxiliary and Start point - AS. * The length of the straight line between Auxiliary and End point - AE. The maximum allowed ratio between these length's is "5": * AS/AE lower than 5. * AE/AS lower than 5. Please ensure that the ratio will be within the limits.

4.10 NC Profiler Caution IDs

The following table lists the NC profiler caution IDs. These caution IDs are caused by the Maestro reading and automatically overcoming an error. These IDs are therefore situations where the Maestro recalculates the motion profile to overcome a possible issue.

Error	Explanation	Resolution
1001	The transition curve does not enter to Line-Line mode	The given input creates a profile with two coincident lines in the same direction
1002	The transition curve does not enter to Circle-Line mode	The direction in the end-point of the circle is similar to the direction of the line start point.
1003	The transition curve does not enter to Circle-Circle mode	The direction in the end-point of the first circle is similar to the direction of the second circle start-point.
1004	Contradiction between Buffer mode and Transition mode	Set the appropriate Buffer mode for the Transition. If you do not set the appropriate Buffer/Transition parameters, the profiler automatically sets a default parameters: Buffer mode - MC_BUFFERED_MOD Transition mode - MC_TM_NONE_MODE
1005	The function block has set a segment length of zero	This error only relates to Maestro MultiAxes linear functions, where the Maestro does not support zero length movement.
1006	Linear Segment length is zero	This error only relates to Maestro MultiAxes linear functions, where the Maestro does not support zero length movement. Verify that your destination location is not equal to the start position, or very close to the start location (less than $1e^{-6}$). In a group of axes, one of the axes must move a discrete distance, while the others may remain static. To get the current position, call the function MMC_ReadActualPositionCmd.
1007	Circular Segment length is zero	This error only relates to MultiAxis circular functions, where the Maestro does not support zero length movement. Verify that your destination location is not equal to the start position, or very close to the start location (less than $1e^{-6}$). In a group of axes, one of the axes must move a discrete distance, while the others may remain static. To get the current position, call the function MMC_ReadActualPositionCmd.

1008	Segment length is zero on ACS mode	This error related only to MultiAxis functions in ACS mode, where the Maestro does not support zero length movement. Verify that your destination location is not equal to the start position, or very close to the start location (less than $1e^{-6}$). In a group of axes, one of the axes must move a discrete distance, while the others may remain static. To get the current position, call the function MMC_ReadActualPositionCmd.
1009	Single axis Segment length is zero, or, the single axis motion fails	This error only relates to SingleAxis functions, where the Maestro does not support zero length movement. Verify that your destination location is not equal to the start position, or very close to the start location (less than $1e^{-6}$). This warning is also generated when a single axis motion fails. To get the current position, call the function MMC_ReadActualPositionCmd.

4.11 Maestro Caution IDs

The following table lists the Maestro caution IDs. These caution IDs are caused by the Maestro being in a specific mode and not able to change mode according to the request.

Error	Explanation	Resolution
2000	The Axis is already in Power OFF mode	Only run the Power OFF function when the axis is not in Power OFF state. To read the current status of the axis call the MMC_ReadStatusCmd function
2001	The Axis is already in Power ON mode	Only run the Power ON function when the axis is not in PowerOn state. To read the current status of the axis call the MMC_ReadStatusCmd function
2002	The Axis is already in current Operation mode	Do not run the "Change Operation Mode" function when the axis is presently active in the same operation mode.
2003	The Axis TouchProbe is already in Enabled state	Do not run the TouchProbe Enable function repeatedly.
2004	The Axis TouchProbe is already in Disabled state	Do not run TouchProbe Disable function repeatedly. When the touch probe is triggered, the state changes automatically to disabled state.
2005	Fast reference is not mapped	You have tried to read\write to fast ref parameter but it was not mapped. Please create a new EtherCAT configuration and map the fast reference

4.12 Internal Library Error IDs

These are errors, which result in the Maestro sending a Maestro library error to the host or other application connected to the Maestro. Their cause may vary from communication to servo drive errors. The following table lists the Internal library Errors.

ID	Explanation	Resolution
100	Warning	Get warning from Maestro or Profiler or EASII, refer to ErrorID variable to get the exact warning number
0	OK	No error. No warning. All OK.
-1	Maestro error	Error occurred on Maestro side. Please refer to ErrorID for further information.
-2000	Send data failed.	Refer to Elmo for support.
-2001	Timeout received as response to sending data.	Receive timeout in response from Maestro, instead of receiving data. Check the communication between the PC and the Maestro. If the communication is OK, but you still receive this error, refer to Elmo for support.
-2002	Wait response fails	Refer to Elmo for support.
-2003	Received failure response to sending data.	Check the communication between the PC and the Maestro. If the communication is OK, but you still receive this error, refer to Elmo for support.
-2004	Received response of incorrect data size.	Check the communication between the PC and the Maestro. If the communication is OK, but you still receive this error, refer to Elmo for support.
-2005	CANbus received response data timeout	Check the cable connections between the Maestro and the drives. Check the CANbus communication between the Maestro and the drives. If the communication is OK, but you still receive this error, refer to Elmo for support.
-2006	Function transfer error	The size of the function input structure is corrupt. Make sure that you have inserted a proper input structure for the function. Make sure that the internal variables of the input structure are assigned correctly.

ID	Explanation	Resolution
-2007	Input Data error	When using the Elmo ASCII interpreter, only send supported commands. All commands are case sensitive.
-2008	No UDP Callback connection established.	Before running this function, run the MMC_OpenUdpChannelCmd function. When sending interpreter commands or Send\Receive SDO commands to the Drive you must send the correct axis reference. The axis reference cannot be greater than the number of active axes.
-2009	Unsupported function under the current connection type	You cannot run the MMC_IPCInitConnection function when you working in WIN32 environment. To initialize connection run MMC_RpcInitConnection.
-3010	UPXML (User Parameters XML), encountered XML Syntax Error in the parameters file.	Replace the relevant XML User Parameters file with another that is in XML syntax format. Be aware of nested, hierarchy and open/close key-element structures and names.
-3011	UPXML (User Parameters XML) failed allocation memory.	Try one or both of the following: Close some non-operational programs running under the Maestro and retry. Replace the User Parameters XML file with a smaller file size, e.g. one with less entries.
-3012	UPXML (User Parameters XML), failed to open XML parameters file.	Check for the correct path and file name linking to the XML User Parameters file.
-3013	UPXML (User Parameters XML) input parameters name (including path) is too long.	Shorten the XML User Parameters path (and or) name.
-3014	UPXML (User Parameters XML) logical Sequence Error.	Change your program logical sequence. Avoid using: Read before Open, or repeating Open before Close.
-3020	UPXML (User Parameters XML) Entry Not Found	The program did not find an XML parameters file entry with the specific name requested. Do one or all of the following: 1. Replace the XML file with one that has the relevant value your program is searching for. 2. Change your program: a. Correct the spelling of the entry name requested. b. If using the function Read for a double (single | array), long (singe | array) or Boolean, set the 'UPXML_SET_DEF_REQ_FLG' when Open (in this case you received the default value and warning but no error).

ID	Explanation	Resolution
-3021	UPXML (User Parameters XML), found but has Unexpected character	The program found the entry you requested from the XML parameters file but its value has an unexpected character (or no characters). Do one or all of the following: 1. Replace the XML file with one that has the relevant value your program is searching for. 2. Change your program. If using the function Read for a double (single | array), long (singe | array) or Boolean, set the 'UPXML_SET_DEF_REQ_FLG' when Open (in this case you received the default value and warning but no error).
-3022	UPXML (User Parameters XML), found a value outside of Min/Max range.	The program found the entry you requested from the XML parameters file but its value is outside the MIN/MAX range. Do one or all of the following: 1. Replace the XML file with one with the value within the MIN/ MAX range. 2. Change your program: a. If using the function Read for a double (single | array), long (singe | array) or Boolean, set the 'UPXML_SET_DEF_REQ_FLG' when Open (in this case you received the default value and warning but no error). b. Change the MIN/MAX range within the program.
-3023	User Param (XML) Failed to open file for write.	Check file protection, name and path; Check free space on Gmas.
-3024	User Param (XML) Sequence error (read with not open; open more then once;	
-3025	User Param (XML) ENTRY NOT Found	
-3026	User Param (XML) Illegal parm for PutFloatFormat	Check your parameter you call with to 'PutFloatFormat()'
-3030	MMC_LIB_UTILTFTP_PARSE_OPT	Util: Tftp activ. arg. parsing errOR
-3031	MMC_LIB_UTILTFTP_CREATE_LOCK	Util: Tftp Cannot create sync lock
-3032	MMC_LIB_UTILTFTP_INIT	Util: Tftp initialisation failure
-3033	MMC_LIB_UTILTFTP_START_SERVER	Util: Tftp Server failed to start
-3034	MMC_LIB_UTILTFTP_INVLD_HOST_NAME	Util: Tftp Invalid hostname specified
-3035	MMC_LIB_UTILTFTP_INVLD_FILE_NAME	Util: Tftp Invalid filename specified
-3036	MMC_LIB_UTILTFTP_CLIENT_SEND_FILE	Util: Tftp Send file failed
-3037	MMC_LIB_UTILTFTP_CLIENT_GET_FILE	Util: Tftp Get file failed
-3051	Write PI to ePI_INPUT is forbidden	You cannot enter write to PI VAR with direction ePI_INPUT.

ID	Explanation	Resolution
-3052	PI VAR TYPE mismatch	The input argumnet type does not match the PI VAR type.
-3053	PI SIZE INVALID	When using read/write of type RAW the size should be PI_REG_VAR_SIZE or PI_LARG_VAR_SIZE.
-3070	EXCEED BULK READ PI SIGNALS NUMBER	Try to init bulk object with size that is greater then NC_MAX_BULK_READ_PI_SIGNALS or smaller then 1
-3071	PI BULK FAIL TO ALLOCATE	Failed to allocate instance of PI BULK
-3072	PIVAR ALREADY APPEND	The PIVar is already append to a bulk read
-3073	CANNOT APPEND TO PI BULK ARRAY FULL	Tried to append to bulk read which is full
-3074	PI BULK ALREADY INITIALIZED	The PI BULK already initialized
-3080	UDP, CREATE: IP_ERR	Is valid IP?
-3081	UDP, CREATE: SOCKET_ERR	Is too many (times call for Create)?
-3082	UDP PORT_ERR	Check you Port value (is it 0?)
-3083	UDP CREATE BIND_ERR	
-3084	UDP CREATE OPT_ERR	
-3085	UDP CREATE CTRL_ERR	
-3086	UDP ISREADABLE INI_ERR	Check if 'CMMCUDP' Class initialized correctly?
-3087	UDP ISREADABLE SEL_ERR	Is time-out?
-3088	UDP SEND SOCKINV_ERR	Is Initialized correctly?
-3089	UDP RECEIVE SOCKINV_ERR	Is initialize correctly?
-3090	UDP CONNECT IP_ERR	Check for Illegal IP...
-3091	UDP CONNECT SOCKET_ERR	Is too many (times call to Connect)?
-3092	UDP CONNECT CTRL_ERR	
-3093	UDP ISPENDING INI_ERR	Is 'CMMCUDP' Class initialized correctly?
-3094	UDP ISPENDING SEL_ERR	Is 'CMMCUDP' Class initialized correctly?
-3095	TCP ISREADABLE INI_ERR	Is 'CMMCUDP' Class initialized correctly?
-3096	TCP ISREADABLE SEL_ERR	Is Time-out?
-3097	TCP CREATE SOCKET_ERR	
-3098	TCP CREATE OPT_ERR	
-3099	TCP CREATE BIND_ERR	
-3100	TCP CRTCLBK SOCKET_ERR	
-3091	UDP CONNECT SOCKET_ERR	Is too many (times call to Connect)?
-3092	UDP CONNECT CTRL_ERR	

ID	Explanation	Resolution
-3093	UDP ISPENDING INI_ERR	Is 'CMMCUDP' Class initialized correctly?
-3094	UDP ISPENDING SEL_ERR	Is 'CMMCUDP' Class initialized correctly?
-3095	TCP ISREADABLE INI_ERR	Is 'CMMCUDP' Class initialized correctly?
-3096	TCP ISREADABLE SEL_ERR	Is Time-out?
-3097	TCP CREATE SOCKET_ERR	
-3098	TCP CREATE OPT_ERR	
-3099	TCP CREATE BIND_ERR	
-3100	TCP CRTCLBK SOCKET_ERR	

4.13 Internal Library Warning IDs

ID	Explanation	Resolution
3020	UPXML (User Parameters XML) Entry Not Found.	The program did not find an XML parameters file entry with the specific name requested. Do one or all of the following: 1. Replace the XML file with one that has the relevant value your program is searching for. 2. Change your program to correct the spelling of the entry name requested.
3021	UPXML (User Parameters XML), found but has an unexpected character	The program found the entry you requested from the XML parameters file but, its value has an unexpected character (or no characters). Replace the XML file with one that has the relevant value your program is searching for.
3022	UPXML (User Parameters XML), found a value outside of the MIN/MAX Range	The program found the entry you requested from the XML parameters file but its value is outside the MIN/MAX range. Do one or all of the following: 1. Replace the XML file with one that has a value within your MIN/ MAX range. 2. Change the MIN / MAX range to include the value necessary.
3023	User Param (XML) Update Elem Val. (exist) in Lvl 3	
3024	User Param (XML) Create (Upd tree) new Elem in Lvl 2	
3025	User Param (XML) Create (Upd tree) new Elem in Lvl 1	

4.14 EtherNetIP Communication Error IDs

The following table describes the EtherNetIP error codes caused by faulty communication.

ID	Explanation	Resolution
0	EIP_ERR_OK	No Error
-1	EIP_ERR_NO_MEMORY_CREATED	no tag was created on Maestro. Check XML structure
-2	EIP_ERR_REFERENCE_OUT_OF_RANGE	invalid tag reference value.
-3	EIP_ERR_REFERENCE_NOT_FOUND	Maestro cannot find tag by given reference.
-4	EIP_ERR_INVALID_INSTANCE	invalid tag instance value
-5	EIP_ERR_INVALID_BUFFER	Invalid device tag buffer. buffer address is NULL.
-6	EIP_ERR_NETPATH_FAILURE	PLC address is missing. Check device NETWORKPATH in XML
-7	EIP_ERR_INVALID_TAG_NAME	Tag name is missing. Check tag NAME in XML
-8	EIP_ERR_ILLEGAL_UNCONNECTED_REQUEST	General unconnected request error. mainly refers to EthernetIP communication problem with PLC
-9	EIP_ERR_DATA_NOT_RECEIVED	Not in use
-10	EIP_ERR_SET_EVENT_FAILURE	Waiting for device object event failed. Refers to synchronous device tag reading.
-11	EIP_ERR_SESSION_FAIL	EthernetIP session opening failed.
-12	EIP_ERR_EIPTASK_FAIL	EthernetIP task invocation has failed.
-13	EIP_ERR_INIT_MUTEX_FAIL	Insufficient resources. Call Elmo support.
-14	EIP_ERR_NO_OPEN_SESSION	No open session. One may probably has not call to EipOpenSession.
-15	EIP_ERR_MEMORY_CREATED	Memory has already been allocated. EipCreat was already invoked.
-16	EIP_ERR_OPEN_FILE_FAILURE	Failed to open XML file. Check if XML file exists and if path is correct.
-17	EIP_ERR_PARSE_FILE_FAILURE	Failed to parse XML file. Validate XML correctness.
-18	EIP_ERR_CLOSE_FILE_FAILURE	Failed to close XML file. Check write permissions of XML file.
-20	EIP_ERR_ASSEMBLY_CREATE_FAIL	Failed to allocate memory for assemblies. Check definitions in XML.
-21	EIP_ERR_DEVTAG_CREATE_FAIL	Failed to allocate memory for device tags. Check definitions in XML.
-22	EIP_ERR_ADPTAG_CREATE_FAIL	Failed to allocate memory for adaptor tags. Check definitions in XML.
-23	EIP_ERR_SEMAPHORE_INIT_FAIL	System error. Call Elmo support.

ID	Explanation	Resolution
-24	EIP_ERR_DESTROY_MUTEX_FAIL	System error. Call Elmo support.
-25	EIP_ERR_INVALID_ASSEMBLY	Invalid definition for assembly. Instance must be greater than zero and less than 255. Supported 'COMFORMAT' attribute is: BOOL, SINT, INT, DINT or REAL.
-26	EIP_ERR_INVALID_DEVICE	Invalid definition for device tag. Supported 'TYPE' attribute is: BOOL, SINT, INT, DINT or REAL.
-27	EIP_ERR_INVALID_ADAPTER	Invalid definition for adaptor tag. Supported 'TYPE' attribute is: BOOL, SINT, INT, DINT or REAL.
-28	EIP_ERR_REQUESTS_LIMIT_REACHED	Number of outstanding object requests for device tags exceeded MAX_REQUESTS limit. Call for Elmo support.
-29	EIP_ERR_OUT_OF_MEMORY	Unconnected request out of memory error. Call for Elmo support.
-30	EIP_ERR_INVALID_NETWORK_PATH	Unconnected request invalid network path error. Check device NETWORKPATH in XML

4.15 Maestro Emergency Error IDs Originating from the Servo Drive

Gold servo drives issue an emergency code in response to an abnormal condition. All emergencies can be masked. For a description of emergency codes that can be masked, refer to object 0x2F21.

The Emergency object COB-ID is 0x81 to 0xFF. The structure of the manufacturer-specific emergency message is as follows:

Byte	Content
0	Error code
1	
2	Error register
3	Elmo error code
4	Error code data field 1
5	
6	Error code data field 2
7	

Note: **Unused bytes must be set to zero.**

The following table lists the Emergency Errors displayed in the Maestro, but originating from the servo drive.

Error Code (Hex)	Error Register (Hex)	Elmo Error Code (Dec)	Data Field	Explanation	Drive Error Resolution
FF01				Request by user program “emit” function	Please contact Elmo For support.
FF02	81	See Section 4.17	0	Either of the possibilities: • IP mode underflow • Interpolation queue full (overfow) • Reference received in a wrong index. This EMCY is transmitted when operating in Profile Interpolated mode, sub mode 0 and RPDO with object 0x60C1 sub index 2 was received. • Bad PVT send order	IP mode failure: • Underflow: Interpolated periods defined via object 0x2F75 is elapsed • Interpolation queue full (overflow): the new set point is received before previous set point was fully processed. • Bad sub-index • Wrong set-points order in case of IP sub mode is -1. In case of underflow: • Check that a new set-point (object 0x60C1) arrived prior to the time defined by 0x2F75. • Check master Sync or RPDO message rate: it might be lower than the period time (0x60C2). In case of overflow: • Check master Sync or RPDO

					message rate: it might be higher than the period time (0x60C2). In case of bad index: • Check that the mapped set-point object (sub index of 0x60C1) is relevant to the IP sub-mode. In case of wrong set-point order: Check that the order of set points 0x60C1.1 and 0x60C1.2 are according to the IP sub-mode definition.
		210		Too large a difference in ECAM table entries.	Please contact Elmo For support.
FF10	81	See Section 4.17	0	Failed to start motor	Commutation auto-phasing failed, and the motor could not be started. A request to initiate the motor using a CANopen Controlword failed. Possible problems may be: • Inhibit/abort switches are active. • Commutation auto-phasing failed. • The PAL is not initiated/burned. • Too little time has passed since the last fault (typically 7.5 msec) or the last motor disable. • Profiler initiation failed due to conflicts between one of the profiler parameter/objects (reason in **EE[2]**). Check the reason in **EC** command
FF20	05	0	0	Safety Torque Off in use	One or two of the safety inputs are in safety state. Check safety indications that are reported in **SR** bits 14 and 15. Check safety inputs. Turn on the safety inputs
FF40	81	0	0	Gantry Slave Disabled	One or two of the safety inputs are in safety state. Check safety indications that are reported in **SR** bits 14 and 15. Check safety inputs. Turn on the safety inputs
2340	03	0	0	Short circuit: motor or its wiring may be defective, or drive is faulty.	The current has exceeded a range which is considered as a phase-to-phase or phase-to-ground short. Check HW. Check if personality is suitable to HW. Check current controller tuning, of feedback noise in case of analog

					sensor.
3120	05	0	0	Under-voltage: power supply is shut down or it has too high an output impedance.	The amplifier is not measuring the minimum required voltage. Check minimum allowed value in **WI[37]** (burnt) and **WI[38]** (actual) command. Check if personality is suitable to HW. Check bus voltage connections (**VL**) or bus voltage reading **AN[6]**
3130	11	0	0	AC fail, loss of phase	Activated in specific HW version of the drive. Refer to the drive User manual. Check hardware configuration dependent on drive. Check extended digital input **XI** command. Check that the motor phases are connected properly.
3310	05	0	0	Over-voltage: power-supply voltage is too high or servo drive could not absorb kinetic energy while braking a load. A shunt resistor may be required.	The amplifier is measuring a voltage which is higher than the allowed threshold. Check maximum allowed voltage in **WI[35]** (burnt) and **WI[36]** (actual) command. Check if personality is suitable to HW. Check bus voltage connections (**VL**) or bus voltage reading **AN[6]**
4310	09	0	0	Temperature: drive overheating. The environment is too hot or heat removal is not efficient. Could be due to large thermal resistance between drive and its mounting.	The drive is sensing a temperature which exceeds the maximum allowed temperature limit. Check actual temperature in **TI[1]** (in Celsius), **TI[2]** in Fahrenheit. Check if personality is suitable to HW Increase **TS** or decrease **XP[2]**
5280	81	0	0	Gantry position error. Gantry yaw or stepper closed loop position error.	Resolution: • Check that the value of **ER[5]** is appropriate to your system and profile • Try to run tuning again • In case of gantry, check that you use optimal commutation for both master and slave • Lower the acceleration or speed of your motion profile
5281				Timing Error	Please contact Elmo For support.
5441	21	0	0	Motor disabled by: • INHIBIT or ABORT and • FLS and RLS are switched on simultaneously in IP or	The function of Abort\Inhibit\FLS\RLS is defined via **IL[]** command. Refer to the Command reference.

				CSP operation modes. Note that FLS\RLS are ignored when **XA[4]**=4	
6180				Stack overflow: fatal exception. May occur if CPU cannot handle a real-time load due to too low a sampling time.	Please contact Elmo For support.
6181				CPU exception: fatal exception.	Please contact Elmo For support.
6200				User program aborted by an error	Please contact Elmo For support.
6300	01	See Section 4.17	0	RPDO failed. Object mapped to an RPDO returned an error during interpretation or a referenced motion failed to be performed.	Check if data of RPDO is correct with respect to the mapped object. Check the reason for the failure as indicated in “Elmo Error Code” (byte 3 aliases to the EC command). Optional reasons may be: parameter is out of range, operating mode is not supported, error mapping in a progress, motor failed to start, etc.
6320				Cannot start due to inconsistent database. This type of database inconsistency is reflected in status SR report and in CD CPU dump report.	Please contact Elmo For support.
7121	21	0	0	Motor stuck - the motor is powered but is not moving according to the definition of **CL[2]** and **CL[3].**	A stuck motor indication can be requested by using **CL[2]**, **CL[3]** and **CL[4]** according to the following logic: If the motor speed is lower than **CL[2]** (in counts/sec) and the measured current is higher than **CL[3]** (in amperes), and if this is observed for more than **CL[4]** msec, the motor is considered to be in the *Motor Stuck* state. Resolution: • Check **CL[2]**, **CL[3]**, **CL[4]** settings. • Check for physical obstructions • Check sensor wiring
7300	81	0	0	Resolver or Analogue Encoder feedback failed	Feedback error: Resolver feedback is not ready – Resolver angle was not located yet. Analog encoder or Resolver feedback is either lost or with too low amplitude. Battery Alarm: Absolute Position may

					be incorrect due to battery power loss. For analog feedbacks check the threshold level in **CA [48]** and **CA[49]** For an absolute encoder check the reason in **EE[1]**. The fault causes a commutation search on the next motor enable. Check the feedback settings
7380				Feedback loss: no match between encoder and Hall locations. Available in encoder + Hall feedback systems	Please contact Elmo For support.
7381	81	0	0	Two digital Hall sensors changed at the same time.	Only one sensor can be changed at a time. Error occurs because digital Hall sensors must be changed one at a time. Digital halls run too fast or disconnected. Check hall settings and hall connections. Try to run the commutation wizard again.
7382	81	0	0	Commutation process fail during motor on	Commutation process fail during motor on for the reasons: • For locking the phase • Planar motor when on alignment process Fault reaction can be selected via 0x605E (**OF[6]**) Check settings of encoder type and encoder resolution. Try running the expert tuner commutation process again.
8110	11	See Section 4.17	0x4000 (Sync lost) 0x2000 (rPDO lost) 0x200 (NMT lost) 0x100 (SDO lost)	CAN message lost (corrupted or overrun)	HW or SW buffer is overflowed. Check the rate of the messages for Sync, NMT or RPDO.
8130	11	See Section 4.17	Node ID	Heartbeat event	Consumer heartbeat time elapsed. Check if producer message was sent in time. Check the consumer time in object 0x1016.
8140	11	See Section 4.17	0	Recovered from bus off	EMCY is sent after the drive recovered from bus-off state • Check the physical connection of the CAN

					• Check that the baud rate of all nodes is the same as the master's • Check the CAN 120 Ω terminators
8200				Protocol error (unrecognized NMT request)	Please contact Elmo For support.
8210	21	0	0	Attempt to access a non-configured RPDO RPDO received is not mapped or contains incorrect number of bytes.	Check the RPDO mapping and the RPDO sent by the host. Check if number of RPDO bytes equals to the bytes that were mapped
8311	21	0	0	Peak current has been exceeded. Possible reasons are drive malfunction or bad tuning of the current controller.	Please contact Elmo For support.
8380				Cannot find electrical zero of motor when attempting to start motor with an incremental encoder and no digital Hall sensors. Applied motor current may not suffice for moving motor from its place.	Please contact Elmo For support.
8381				Cannot tune current offsets	Please contact Elmo For support.
8480	81	0	0	Speed tracking error **DV[2]** - **VX** (for **UM=2** or **UM=4**, **5**) exceeded speed error limit **ER[2]**. This may occur due to: • Bad tuning of the speed controller • Too tight a speed error tolerance • Inability of motor to accelerate to the required speed due to too low a line voltage or insufficient motor power	Resolution: • Check that the value of **ER[2]** is appropriate to your sensor and profile. • Try to run tuning again, and consider using feed forward. • Check that you use optimal commutation • Lower the acceleration or speed of your motion profile
8481	81	0	0	Speed limit exceeded: **VX**<**LL[2]** or **VX**>**HL[2]**. (Compatibility only)	Over speed indication . (Compatibility only) The motor speed has exceeded the value which is defined in **HL[2]** or **LL[2]**. **VX**<**LL[2]** or **VX**>**HL[2]**The motor main speed is reported in **VX**. Perform tuning. To cancel this protection, set **HL[2]**=0. **HL[2]** is in user units
8611	21	0	0	Position tracking error **DV[3]** - **PX** (**UM=5**) or **DV[3]** - **PY** (**UM=4**) exceeded	Resolution: • Check that the value of **ER[3]** is appropriate to your sensor and

				position error limit **ER[3]**. This may occur due to: • Bad tuning of the position or speed controller • Too tight a position error tolerance • Abnormal motor load, or reaching a mechanical limit	profile. • Try to run tuning again. • Check that you use optimal commutation • Lower the acceleration or speed of your motion profile.
8680	81	0	0	Position limit exceeded: **PX**<**LL[3]** or **PX**>**HL[3]** (**UM=5**), or **PY**<**LL[3]** or **PY**>**HL[3]** (**UM=4**). (Compatibility only)	Position limit exceeded: **PX**<**LL[3]** or **PX**>**HL[3]** (**UM=5**), or **PY**<**LL[3]** or **PY**>**HL[3]** (**UM=4**). (Compatibility only) The main feedback is reported in **PX**. Check **HL[3]**, **LL[3]** Motor position exceeded feedback's limits. To cancel this protection set **HL[3]**=**LL[3]**=0. These parameters are in user units

4.16 Maestro Abort Error IDs Originating from the Servo Drive

This protocol is used to implement the Abort SDO Transfer service.

Client to server *or* server to client

0		1-3	4-7
7…5	4…0		
Cs = 4	**X**	**M**	**D (data)**

where:

Cs Command specifier 4: Abort transfer request

X Not used; always 0.

M Multiplexor. Represents index (bytes 1,2) and sub-index (byte 3) of SDO.

D Four-byte abort error code (see Table 5-1: SDO Abort Codes) code giving reason for abort, encoded as Unsigned32 value.

The following table lists the Abort Errors displayed in the Maestro, but originating from the servo drive.

Abort Error Code(Hex)	Explanation	Drive Error Resolution
0x05030000	Toggle bit not alternated.	**Retry to boot the system (Maestro and servo drives). If the same situation prevails, contact Elmo For support.**
0x05040001	Invalid or unknown client/server command specifier.	
0x05040002	Invalid block size	
0x05040003	Invalid sequence number in SDO block upload	
0x05040005	Out of memory.	
0x06010000	Unsupported access to an object.	
0x06010001	Attempt to read a write-only object.	
0x06010002	Attempt to write a read-only object.	
0x06020000	Object does not exist in object dictionary.	
0x06040041	Object cannot be mapped to PDO.	
0x06040042	Number and length of objects to be mapped exceeds PDO length.	
0x06040043	General parameter incompatibility.	
0x06060000	Access failed due to hardware error	
0x06070012	Data type does not match, service parameter too long	
0x06090011	Sub-index does not exist	
0x06090030	Value range of parameter exceeded (only for write	

	access)	
0x06090031	Value of parameter written too high	
0x06090032	Value of parameter written too low	
0x06090036	Maximum value is less than minimum value	
0x08000000	General error. When the abort code is 0x08000000, the actual error can be retrieved using the **EC** command	
0x08000020	Data cannot be transferred to or stored in application	
0x08000022	Data cannot be transferred to or stored in application due to present device state	
0x08000024	There is no data available to transmit	

4.17 ELMO Error Codes

The following table (Table 7-2: ELMO error codes) lists the ELMO Error codes and their description with the relevant Elmo code where applicable.

Description	ELMO Error Code(Dec)
Will not be updated	1
Bad command	2
Bad index	3
PAL does not support this sensor	4
Mode cannot be started - bad initialization data	7
CAN message was lost – hardware error	9
Cannot be used by PDO	10
Cannot write to flash memory	11
Cannot reset communication - UART is busy	13
Array '[ ]' is expected, or empty expression in array	16
Format of **UL** command is not valid - check the command definition	17
Command syntax error	19
Bad Set Point sending order	20
Operand out of range	21
Zero division	22
Command cannot be assigned	23
Bad operation	24
Profiler mode not supported in this unit mode (**UM**)	26
Bad ECAM setting, refer to **EE[6]**	27
Out Of limit range	28
CAN get object return an abort when called from interpreter	31
Communication overrun, parity, noise, or framing error	32
Bad sensor setting, check **CA[18], CA[19]**	33
There is a conflict with another command	34
Max bus voltage (**BV**) or max current (**MC**) is not valid	35
Commutation method (**CA[17]**) or commutation table does not fit to sensor	36
Hall sensors are defined to the same place, check **CA[4], CA[5]** or **CA[6]**	37

Description	ELMO Error Code(Dec)
PORT C mux is with conflict with other **GO[]** or cannot be assigned to the output	38
Operating in Wizard experimental mode	40
Command is not supported by this product	41
No Such Label	42
Return error from subroutine	45
Program does not exist or not compiled	47
Motor could not start - fault reason in **EE[5]**	48
ECAM must be disabled during init (**RM=**0)	49
Stack overflow	50
Inhibit OR Abort inputs are active, cannot start motor	51
PVT queue full	52
Bad database	54
Bad context	55
Motor must be off (**MO=0**)	57
Servo (**SO**) must be on	58
Bad unit mode	60
Database reset	61
Socket change not allowed while capture is enabled	62
Amplifier not ready	66
Recorder is busy or data is uploading	67
Required profiler mode is not supported	68
Recorder usage error	69
Recorder data Invalid	70
Homing is busy	71
Modulo range must be even	72
Please set position	73
Bad profile database, see **EE[2]** or 0x2081 for the failed object number	74
Download is in progress	75
Error mapping is not allowed while Homing is in progress (**HM[1]\HF[1]**)	76
Out of program range	78

Description	ELMO Error Code(Dec)
Sensor setting error, possible conflict with other socket settings	79
ECAM data inconsistent	80
Download failed see specific error in **EE[3]**	81
Program is running	82
Command is not permitted in a program	83
STO is not active OR During STO Diagnostic	85
Reserved	87
Not allowed while Error mapping (**PC[1]**) is in progress	94
User program time out	96
RS232 receive buffer overflow	97
Current offsets are beyond the allowed limit	98
Bad auxiliary sensor configuration	99
The requested PWM multiplication value is not supported	100
Absolute encoder setting problem	101
Output Compare (**OC[1] \ OC[21]**) or Emulation (**EA[1]**) are busy	102
Output compare sensor is not QUAD Encoder	103
Output Compare Table Length OR Data	104
Speed loop KP out of range	105
Encoder emulation parameter (**EA[2] to EA[7]**) is out of range	107
Encoder emulation (**EA[1]**) is already in progress	108
Too long number	110
Please wait until analog sensor initialized	121
Motion mode is not supported or with initialization conflict	122
Profiler queue is full	123
Personality not loaded	125
User Program failed - variable out of program size	126
Bad variable index in database - internal compiler error	128
Variable is not an array	129
Variable name does not exist	130
Cannot record local variable	131
Variable is an array	132

Description	ELMO Error Code(Dec)
Number of function input arguments is not as expected	133
Cannot run local label/function with **XQ** command	134
Frequency identification failed	135
Not a number	136
Position Interpolation buffer underflow	138
The number of break points exceeds maximal number	139
An attempt to set/clear break point at the not relevant line	140
Boot Identity parameters section is not clear	141
Checksum of data is not correct	142
Numeric Stack underflow	144
Numeric stack overflow	145
Executable command within math expression	147
Nothing in the expression	148
Parentheses mismatch	151
Bad operand type	152
Overflow in a numeric operator	153
Address is out of data memory segment	154
Beyond stack range	155
Bad op-code	156
Out of flash memory range	158
Flash verify error	159
Program is not halted	161
Not enough space in program data segment	163
An attempt to access flash while busy	165
Out of modulo range	166
Speed too large to start motor	168
Time out using peripheral.(overflow or busy)	169
Cannot erase sector in flash memory	170
Cannot read from flash memory	171
Cannot write to flash memory	172
Executable area of program is too large	173

Description	ELMO Error Code(Dec)
Program has not been loaded	174
Cannot write program checksum - clear program (**CP**)	175
User code, variables and functions are too large	176
Capture/Compare conversion error in analog encoder.	177
CAN bus off	178
Consumer **HB** event	179
DF is not supported in this communication type	180
Writing to Flash program area, failed	181
PAL Burn Is In process or no PAL is burnt or incompatible PAL version	182
Capture option already used by other operation	184
This element may be modified only when interpolation is not active	185
Interpolation queue is full	186
Incorrect Interpolation sub-mode	187
Gantry slave disable	188
CAN message was lost - software	189
Profile Acceleration is out of range	190
Motor over temperature	191
Main feedback error. Refer to **EE[1]**	200
Commutation sequense failed	201
Encoder-Hall sensor mismatch. Refer to **XP[7]**	202
Current limit was exceeded	203
External inhibit input detected	204
AC fail: Loss of phase	205
Digital halls run too fast or disconnected	206
Speed error limit exceeded. Refer to **ER[2]**	207
Position limit error exceeded. Refer to **ER[3]**	208
Cannot start motor . Bad data base. Refer to CD	209
Bad ECAM table	210
Cannot find zero position without digital halls	216
Over speed. Refer to **HL[2]**	217
Motor stack	221

Description	ELMO Error Code(Dec)
Out of position limits. Refer to **HL[3], LL[3]**	222
Numerical overflow	223
Gantry slave is not enabled	224
Cannot start motor because of internal problem	229
Undervoltage protection	233
Overvoltage protection	235
Safety switch	237
Short protection	241
Over temperature protection	243
Additional inhibit input	245

Table 7-1: ELMO error codes

4.18 Functional and Communication Error Handling

4.18.1 General

When the Maestro is in Operational mode, the Maestro is actively sensitive to monitor specific errors which are governed by a policy effecting the resolution of these errors.

The following table describes the various methods employed by the Maestro to resolve the different errors under the Maestro policy.

Error	Policy	Affected Axes	Error Stop
System errors e.g. Cyclic errors Cyclic errors e.g wrong Working Count, Missed Frames	Stop. Insert Stop function block Power Insert Power function block to Motor Off drives Enter SafeOP state Send appropriate Event Apply to all – policy is applied toward all nodes in the system (mandatory in system error) Error handling delayed until all previous error handling execution is completed	All axes User can define how specific error is handled by Maestro. Either no action is performed, or integration is checked/unchecked depending on user request for: Stop function block: Axis stops at predefined stop deceleration Power function block: Motor is shut down after axis stops Move system to SAFEOP Notify user of error via event	Dependent on policy All axes
PHY e.g. ECAT processing unit error, PDI Errors		Detected axis belongs to active vector – all vector members	Dependant on policy Applied to relevant axis
		Only detected axis	Dependant on policy Applied to relevant axis
		Detected axis + all axes User can define how specific error is handled by Maestro. Either no action is performed, or integration is checked/unchecked depending on user request for: Stop function block: Axis stops at predefined stop deceleration Power function block: Motor is shut	Dependant on policy All axes

Error	Policy	Affected Axes	Error Stop
		down after axis stops Move system to SAFEOP Notify user of error via event	
Node errors during Policy execution	Policy execution continues and delays the new policy execution. If one of the policy stages fails, user receives in event the policy execution phase that failed (e.g. motor off stage), an informative error code indicating reason for failure.	Relevant axis	Yes
EtherCAT Application Layer (AL Error)	**For single axis** Send appropriate Event **For axis part of active vector** Insert Stop function block Send power to sequential function block Send appropriate Event	Axis is immediately entered to error stop Policy applied on other group members No effect on entire system	Yes
Axis related errors e.g. (same as heartbeat), unexpected motor off, Heartbeat Error, Drive fault	**For single axis** Send appropiate Event **For axis part of active vector** Insert Stop function block Send power to sequential function block Send appropriate Event	Axis is immediately entered to error stop Policy applied on other group members No effect on entire system	
Function Block e.g. execution failure	A dedicated policy is applied for the axis\group Policy: Insert Stop function block Send power to sequential function block Send appropriate Event	The axis or vector (if part of active vector)	

Error	Policy	Affected Axes	Error Stop
Emergency	Policy: Insert Stop function block Send power to sequential function block Enter SafeOP state Send appropriate Event	If error does not enter axis to FAULT or motor off, the policy is invoked towards the EMCY axis If axis is part of active vector, policy is applied to all vector	
Drive Level DS-402 Quick Stop	Axis enters to ERROR STOP immediately. When error detected, the Maestro applies a predefined policy on the axis group members. The policy is not invoked on the axis causing the error.	Other group members	
Profiler	For future use	TBD	No
Software Limit	For future use	TBD	No

4.18.2 Detecting Connection Errors

4.18.2.1 System Level Cyclic Errors

The Maestro control monitors the EtherCAT communication errors so that the user is notified about the error and the optional user resolution is performed. The Maestro control compares the number of errors to the user requested threshold. If the threshold is passed, the Maestro is notified and will be invoked.

4.18.2.2 Working Counter (WC) Decreases – Cyclic Errors

The process image read /write commands are monitored with the Working Counter when all the slaves are in Operational state. When a message is recieved, the WC will be compared against the number of active slaves. If the WC shows a decrease compared to the active number of slaves, a counter is incremented on each cycle that this condition is true. When the predefined threshold is passed, the predefined (user policy) resolution is invoked.

4.18.2.3 Timeouts – Missed Frames

The Maestro control monitors the number of unanswered messages sent to the EtherCAT network. Each unanswered message increments a counter which is compared to the user threshold. When this counter reaches the threshold, the user resolution is invoked.

4.18.2.4 PHY Errors

PHY errors are read from registers. A default single threshold is defined for the number of:

- Invalid frame counter
- RX error counter
- Forwarded RX error counter
- ECAT Processing Unit error counter
- PDI Error counter
- Lost Link counter

When the value of one of these error counters passes the threshold, the user resolution is invoked.

4.18.2.5 Node Connected\Disconnected (Heartbeat Error)

When a node connection status changes the Maestro control is notified, which in turn communicates the connection status change to the user. The connection status is monitored by the hot plug mechanism. If a disconnection is noted, the axis enters to ERROR_STOP immediately.

When the Maestro control detects a change in connection status a Heartbeat error event or node connected event is sent to the user and after receiving the Heartbeat error event, the predefined policy is applied to the other group members, only if the group is enabled and the other members are not currently in an error condition. After the node is reconnected, perform MMC_Reset to restart the Axis.

4.18.3 Detecting EtherCAT Application Level (AL) Errors

When an AL error is detected by the Maestro control the user is notified the axis will enter Error_Stop and an event is sent. The dedicated policy is then invoked on the slave's group members. An AL error is reset by the MMC_Reset() function.

4.18.4 DS-402 Status Word Error

Sometimes the drive changes its status because of a constraint. e.g. motor turns off, despite not being shut down by the master. The drive indicates these status changes by its Status Word. In order to reflect the actual status of the drive, the Maestro monitors the Status Word and verifies that its value is correlated with the drive PLC Open state stored in the Maestro. Whenever a mismatch is detected, the corresponding user selected resolution is invoked, and the Axis state is updated accordingly. The following unexpected Status Word situations are handled:

- Motor shut down – current axis state is Standstill but the Status Word indicates a different state other than Operation Enabled
- Quick Stop activated – the drive initiates a Quick Stop reaction, and the axis state is changed
- Fault – the drive software indicates a fault or fault reaction active

When one of the situations above occurs, the axis enters **Error Stop** state.

In the case of Quick Stop, the system waits until the Quick Stop procedure ends in the drive level until it enablesresetting of the axis.

When a Status Word error occurs, the predefined policy is activated on the other group members.Whenever one of the errors above is detected. The axis enters **Error Stop** immediately, and the policy is invoked on the axes of other group members.

NOTE: The Maestro support only QSTOP options that end in switch on disabled state (0 to 4) and initializes the axis for the default option:

+2	Slow down on quick stop ramp and transit into Switch On Disabled

The Maestro will initialize the drive to this option on system init and after any reset. If the user has configured a different option and the drive's final state is quick stop active, the user will only be able to reset the axis and no resolution will be invoked on the group except group error.

4.18.4.1 Function Block Error

When a function block error occurs for any reason, and the error is detected (with no other error handling in progress) the user's predefined policy is invoked on the axis and its group members. If a group function block error occurs the group enters Error Stop and no resolution is invoked.

4.18.4.2 Emergency

If an Emergency message arrives from the drive, the Maestro will attempt to perform the predefined policy towards the axis, on the condition it did not enter the Fault Error state as this would result in an immediate Error Stop. In every case the predefined policy is invoked towards the axis group members.

4.18.5 Handling Errors – Resolution

After the error has been detected, a resolution is invoked according to the user policy. Several policies are enabled according to the relevant error. A combination of these resolutions can be invoked (besides no resolution option). The default resolution is an event notification and insert Stop function block.

The following table describes and explains the Resolution Types.

Resolution Types	Explanation
No Reaction	No reaction to the error will be performed by the Maestro
Insert Stop Function Block	Stop function block will be inserted and will abort every other function block in the queue
Insert Power Function Block	The user will be able to select this option if it is necessary that the drive set Power Off causing Motor Off after the Stop function block has completed its operation.
Move System to SAFEOP	The user can determine that in the case of a system error (or if an Apply-to-All policy is selected), all slaves will change state to SAFEOP.
Send Notification By Event	If the user selects this option he will receive a UDP event regarding the error. The Error events will only be sent according to the policy regardless of the events mechanism bit masks. This event is not sent on error detection (in contrary to existing events such as Heartbeat error or Emergency error), but after the error policy was executed. The event indicates the ending of an error policy and returns the stage where the policy failed (or final stage if policy ended successfully) i.e. it will be sent only after the error handling is completed. In case of a system error, the state is always the final state but the return value might indicate that the policy was not applied on some of the axes. In this situation, the user can read two parameters on the axis\group level indicating at which stage the policy failed to apply on the axis and the reason for the failure.
Apply to All	When selected, the policy is applied not only on axis\group but on the entire system. Every axis which is not in Error Stop will be stopped and\or motor off and\or set SAFEOP. The resolution will also be applied on the axis causing the error. Finally all nodes are entered **Error Stop**. When the policy is Apply To All, MMC_SystemReset is necessary after a policy is applied. This policy may be applied only to a PHY error, cyclic error and missed frames. It is a mandatory policy for missed frames and cyclic errors When this policy is selected, Maestro will enter fatal error state and when MMC_SystemReset() is invoked successfully, the fatal error will be cleared.

4.18.6 Policy Functions

The user can use the following functions to select the policy for a specific error:

Policy
MMC_RegErrPolicy
MMC_GetErrPolicy
MMC_ResetSystem

4.18.7 MMC_RegErrPolicy

This function registers and defines an error policy.

```
int MMC_RegErrPolicy(
MMC_CONNECT_HNDL hConn,
MMC_REGERRPOLICY_IN* pInParam,
MMC_REGERRPOLICY_OUT* pOutParam
);
```

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_REGERRPOLICY_IN** input structure that receives the MMC_RegErrPolicy Information

pOutParam

Points to the **MMC_REGERRPOLICY_OUT** output structure receiving information as a result of calling the MMC_RegErrPolicy function.

Remarks

None

Scope

All

MMC_REGERRPOLICY_IN Structure

```
typedef struct{
        NC_POLICY_ENTRY pPolicies[MAX_REG_POLICY];
        unsigned short usAxisRef;
        unsigned char ucNum;
        unsigned char pSpare[64];
} MMC_REGERRPOLICY_IN;
```

Parameters

NC_POLICY_ENTRY pPolicies[MAX_REG_POLICY]

An array of errors and policies to be registered, where [MAX_REG_POLICY] is the maximum number of policies that can be registered

NC_POLICY_ENTRY

```
typedef struct nc_policy_entry{
        ERRORS eErrType;
        unsigned char ucPolicy;
        unsigned char ucThreshold;
} NC_POLICY_ENTRY;
```

ERRORS eErrType

Enumerator of error that the policy is to be registered on.

eNODE_ERROR_ECAT_PHY_ERROR	= 0,
eSYS_ERROR_CYCLIC_ERROR	= 1,
eSYS_ERROR_MISSED_FRAMES	= 2,
eNODE_ERROR_ECAT_AL_ERROR	= 3,
eNODE_ERROR_UNEXPEC_MO_0	= 4,
eNODE_ERROR_DRIVE_FAULT	= 5,
eNODE_ERROR_QSTOP	= 6,
eNODE_ERROR_HBT	= 7,
eNODE_ERROR_EMCY	= 8,
eNODE_ERROR_FB	= 9,
eNODE_ERROR_MAX	

ucPolicy

Bitwise value that defines the policy options:

0	No reaction
0x1	Send notification by event
0x2	All drives perform stop function block
0x4	All drives enter SAFEOP state - Move Axis\System to SAFEOP

0x8	Perform EEPROM scan of the entire network
0x80	Apply policy to the entire system

ucThreshold

The value of the threshold, after which the resolution will operate. Any +ve value.

usAxisRef

Axis ref. of relevant axis for axis errors. +ve value.

ucNum

Number of entries in the array to be registered. +ve character.

pSpare[64]

An entry of a spare policy type as n unsigned character with a limit of 64 bits. Not in use at this moment.

MMC_REGERRPOLICY_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
} MMC_REGERRPOLICY_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 4-3 describes the function block for MMC_RegErrPolicy.

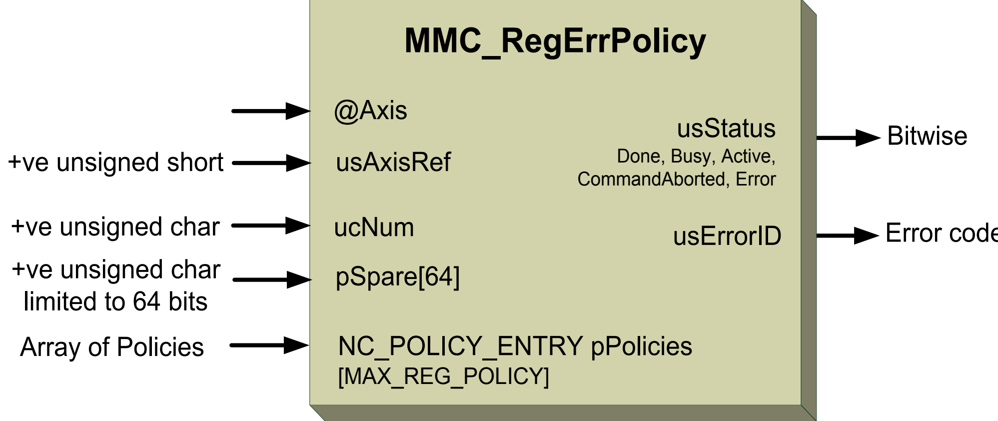


Figure 4-3: MMC_RegErrPolicy function block

4.18.7.2 Function Block Code and Implementation Example

```
MMC_REGERRPOLICY_IN RegIn;
MMC_REGERRPOLICY_OUT RegOut;
unsigned short usPolicies;
int i = 0;

printf("how many policies?\n");
scanf("%hu",&usPolicies);

printf("which axis?\n");
scanf("%hu",&usAxis);

RegIn.usAxisRef = usAxis;

if(usPolicies <= MAX_REG_POLICY)
{
      for(i = 0; i < usPolicies; i++)
      {
printf("which error 0 - eNODE_ERROR_MAX, otherwise finish\n");
scanf("%hu",&usAns);

switch(usAns)
{
      case eNODE_ERROR_ECAT_PHY_ERROR:
      {
         printf("eNODE_ERROR_ECAT_PHY_ERROR selected 0x%X\n",eNODE_ERROR_ECAT_PHY_ERROR);
         RegIn.pPolicies[i].eErrType = eNODE_ERROR_ECAT_PHY_ERROR;
      }
      break;
      case eSYS_ERROR_CYCLIC_ERROR:
      {
         printf("eSYS_ERROR_CYCLIC_ERROR selected 0x%X\n",eSYS_ERROR_CYCLIC_ERROR);
         RegIn.pPolicies[i].eErrType = eSYS_ERROR_CYCLIC_ERROR;
      }
      break;
      case eSYS_ERROR_MISSED_FRAMES:
      {
         printf("eSYS_ERROR_MISSED_FRAMES selected 0x%X\n",eSYS_ERROR_MISSED_FRAMES);
         RegIn.pPolicies[i].eErrType = eSYS_ERROR_MISSED_FRAMES;
      }
      break;
      case eNODE_ERROR_ECAT_AL_ERROR:
```

```
        {
           printf("eNODE_ERROR_ECAT_AL_ERROR selected 0x%X\n",eNODE_ERROR_ECAT_AL_ERROR);
           RegIn.pPolicies[i].eErrType = eNODE_ERROR_ECAT_AL_ERROR;
        }
        break;
        case eNODE_ERROR_UNEXPEC_MO_0:
        {
           printf("eNODE_ERROR_UNEXPEC_MO_0 selected 0x%X\n",eNODE_ERROR_UNEXPEC_MO_0);
           RegIn.pPolicies[i].eErrType = eNODE_ERROR_UNEXPEC_MO_0;
        }
        break;
        case eNODE_ERROR_DRIVE_FAULT:
        {
           printf("eNODE_ERROR_DRIVE_FAULT selected 0x%X\n",eNODE_ERROR_DRIVE_FAULT);
           RegIn.pPolicies[i].eErrType = eNODE_ERROR_DRIVE_FAULT;
        }
        break;
        case eNODE_ERROR_QSTOP:
        {
           printf("eNODE_ERROR_QSTOP selected 0x%X\n",eNODE_ERROR_QSTOP);
           RegIn.pPolicies[i].eErrType = eNODE_ERROR_QSTOP;
        }
        break;
        case eNODE_ERROR_HBT:
        {
           printf("eNODE_ERROR_HBT selected 0x%X\n",eNODE_ERROR_HBT);
           RegIn.pPolicies[i].eErrType = eNODE_ERROR_HBT;
        }
        break;
        case eNODE_ERROR_EMCY:
        {
           printf("eNODE_ERROR_EMCY selected 0x%X\n",eNODE_ERROR_EMCY);
           RegIn.pPolicies[i].eErrType = eNODE_ERROR_EMCY;
        }
        break;
        case eNODE_ERROR_FB:
        {
           printf("eNODE_ERROR_FB selected 0x%X\n",eNODE_ERROR_FB);
           RegIn.pPolicies[i].eErrType = eNODE_ERROR_FB;
        }
        break;
}

printf("enter threshold\n");
scanf("%hhu",&RegIn.pPolicies[i].ucThreshold);

RegIn.pPolicies[i].ucPolicy = 0;

printf("event? 0/1");
scanf("%hu",&usAns);
if(usAns)
{
        RegIn.pPolicies[i].ucPolicy |= ePOLICY_EVENT;
}

printf("stop axis? 0/1");
scanf("%hu",&usAns);
if(usAns)
{
        RegIn.pPolicies[i].ucPolicy |= ePOLICY_STOP;
}

printf("power off axis? 0/1");
scanf("%hu",&usAns);
if(usAns)
{
        RegIn.pPolicies[i].ucPolicy |= ePOLICY_POWER_OFF;
```

```
}

printf("safeop axis? 0/1");
scanf("%hu",&usAns);
if(usAns)
{
	RegIn.pPolicies[i].ucPolicy |= ePOLICY_SAFEOP;
}
printf("applly to all? 0/1");
scanf("%hu",&usAns);
if(usAns)
{
	RegIn.pPolicies[i].ucPolicy |= ePOLICY_APP_TO_ALL;
}
	}
}
RegIn.ucNum = usPolicies;
rc = MMC_RegErrPolicy(ConnHndl,&RegIn,&RegOut);
if(rc)
{
	printf("MMC_RegisterErrorPolicy failed, error %d\n",RegOut.sErrorID);
}
```

4.18.8 MMC_GetErrPolicy

Commands the axis to perform the Search Home sequence.

```
int MMC_GetErrPolicy(
MMC_CONNECT_HNDL hConn,
MMC_GETERRPOLICY_IN* pInParam
MMC_GETERRPOLICY_OUT* pOutParam
);
```

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETERRPOLICY_IN** input structure that receives the MMC_GetErrPolicy command.

pOutParam

Points to the **MMC_GETERRPOLICY_OUT** output structure receiving information as a result of calling the MMC_GetErrPolicy function.

Remarks

None

Scope

All

MMC_GETERRPOLICY_IN Structure

```
typedef struct{
    ERRORS pErrType[MAX_REG_POLICY];
    unsigned short usAxisRef;
    unsigned char ucNum;
}MMC_GETERRPOLICY_IN;
```

Parameters

ERRORS pErrType[MAX_REG_POLICY]

An array of errors that their policy should be returned, where [MAX_REG_POLICY] is the maximum number of policies that can be returned.

usAxisRef

Axis ref of the axis its policies (if axis related) should be returned. +ve value.

ucNum

Number of policies to be returned. +ve character.

MMC_GETERRPOLICY_OUT Structure

```
typedef struct{
    NC_GET_POLICY_ENTRY pPolicies[MAX_REG_POLICY];
    unsigned short usStatus;
    short sErrorID;
    unsigned char pSpare[64];
} MMC_GETERRPOLICY_OUT;
```

Parameters

NC_GET_POLICY_ENTRY pPolicies[MAX_REG_POLICY]

An array of returned policies data.

NC_GET_POLICY_ENTRY

```
typedef struct nc_get_policy_entry{
    unsigned char ucPolicy;
    unsigned char ucThreshold;
    unsigned char ucCurrentVal;
} NC_GET_POLICY_ENTRY;
```

ucPolicy

Bitwise value that defines the policy options:

0	No reaction
0x1	Send notification by event
0x2	All drives perform stop function block
0x4	All drives enter SAFEOP state - Move Axis\System to SAFEOP
0x8	Perform EEPROM scan of the entire network
0x80	Apply policy to the entire system

ucThreshold

The current value of the registered threshold, after which the resolution will operate. Any +ve value.

ucCurrentVal

Current value of the error counter. Any +ve value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

pSpare[64]

An entry of a spare policy as n unsigned character with a limit of 64 bits.

Figure 5-10 describe the function block for MMC_GetErrPolicy.

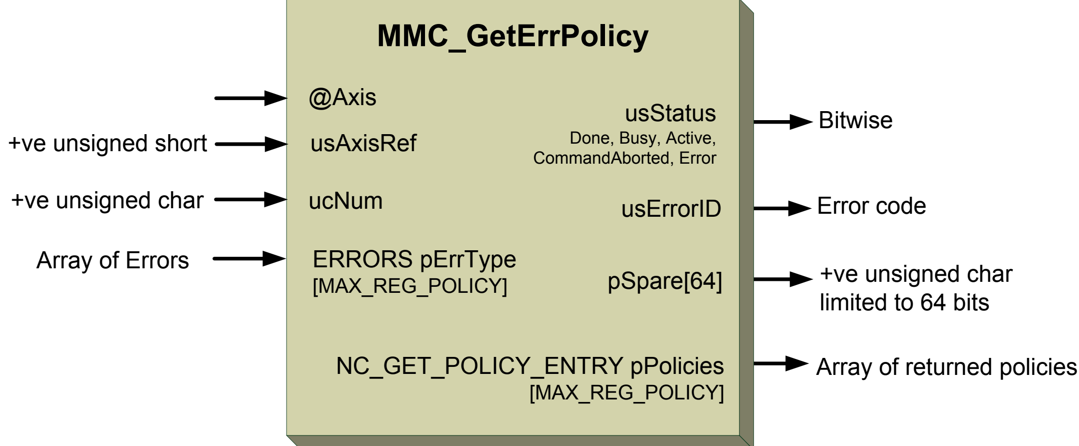


Figure 4-4: MMC_GetErrPolicy function block

4.18.8.2 Function Block Code and Implementation Example

```
MMC_GETERRPOLICY_IN GetIn;
MMC_GETERRPOLICY_OUT GetOut;
unsigned short usPolicies;
int i = 0;

printf("how many policies?\n");
scanf("%hu",&usPolicies);

printf("which axis?\n");
scanf("%hu",&usAxis);

GetIn.usAxisRef = usAxis;

if(usPolicies <= MAX_REG_POLICY)
{
      for(i = 0; i < usPolicies; i++)
      {
          printf("which error 0 - 7\n");
          scanf("%hu",&usAns);
          switch(usAns)
          {
     case eNODE_ERROR_ECAT_PHY_ERROR:
         {
         printf("eNODE_ERROR_ECAT_PHY_ERROR selected\n");
         GetIn.pErrType[i] = eNODE_ERROR_ECAT_PHY_ERROR;
         }
         break;
     case eSYS_ERROR_CYCLIC_ERROR:
         {
         printf("eSYS_ERROR_CYCLIC_ERROR selected\n");
         GetIn.pErrType[i] = eSYS_ERROR_CYCLIC_ERROR;
         }
         break;
     case eSYS_ERROR_MISSED_FRAMES:
         {
         printf("eSYS_ERROR_MISSED_FRAMES selected\n");
         GetIn.pErrType[i] = eSYS_ERROR_MISSED_FRAMES;
         }
         break;
     case eNODE_ERROR_ECAT_AL_ERROR:
         {
```

```
        printf("eNODE_ERROR_ECAT_AL_ERROR selected\n");
        GetIn.pErrType[i] = eNODE_ERROR_ECAT_AL_ERROR;
        }
        break;
    case eNODE_ERROR_UNEXPEC_MO_0:
        {
        printf("eNODE_ERROR_UNEXPEC_MO_0 selected\n");
        GetIn.pErrType[i] = eNODE_ERROR_UNEXPEC_MO_0;
        }
        break;
    case eNODE_ERROR_DRIVE_FAULT:
        {
        printf("eNODE_ERROR_DRIVE_FAULT selected\n");
        GetIn.pErrType[i] = eNODE_ERROR_DRIVE_FAULT;
        }
        break;
    case eNODE_ERROR_QSTOP:
        {
        printf("eNODE_ERROR_QSTOP selected\n");
        GetIn.pErrType[i] = eNODE_ERROR_QSTOP;
        }
        break;
    case eNODE_ERROR_HBT:
        {
        printf("eNODE_ERROR_HBT selected\n");
        GetIn.pErrType[i] = eNODE_ERROR_HBT;
        }
        break;
    case eNODE_ERROR_EMCY:
        {
        printf("eNODE_ERROR_EMCY selected\n");
        GetIn.pErrType[i] = eNODE_ERROR_EMCY;
        }
        break;
    case eNODE_ERROR_FB:
        {
        printf("eNODE_ERROR_FB selected\n");
        GetIn.pErrType[i] = eNODE_ERROR_FB;
        }
        break;
         }
      }
}

GetIn.ucNum = usPolicies;

rc = MMC_GetErrPolicy(ConnHndl,&GetIn,&GetOut);
if(rc)
{
      printf("MMC_GetErrPolicy failed error %d\n",GetOut.sErrorID);
}
```

4.18.9 MMC_ResetSystem

This function resets the entire system errors, including error counters of PHY, cyclic and missed frames errors. In addition it changes all nodes to INIT and then to OPERATIONAL state, resulting in motor off of all drives. In addition this function might reset Maestro fatal errors and return it to fully operational status.

```
int MMC_ResetSystem(
MMC_CONNECT_HNDL hConn,
MMC_RESETSYSTEM_IN* pInParam,
MMC_RESETSYSTEM_OUT* pOutParam
);
```

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_REGERRPOLICY_IN** input structure that receives the MMC_RegErrPolicy Information

pOutParam

Points to the **MMC_REGERRPOLICY_OUT** output structure receiving information as a result of calling the MMC_RegErrPolicy function.

Remarks

None

Scope

All

MMC_RESETSYSTEM_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_RESETSYSTEM_IN;
```

Parameters

ucDummy

No data required. Dummy data input.

MMC_RESETSYSTEM_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
} MMC_RESETSYSTEM_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 4-5 describes the function block for MMC_ResetSystem.

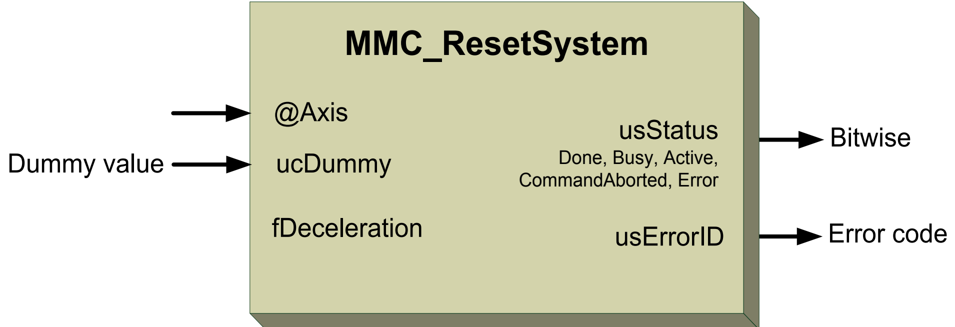


Figure 4-5: MMC_ResetSystem function block

4.18.9.2 Function Block Code and Implementation Example

```
MMC_RESETSYSTEM_IN ResetIn;
MMC_RESETSYSTEM_OUT ResetOut;

rc = MMC_ResetSystem(ConnHndl,&ResetIn,&ResetOut);
if(rc)
{
	printf("MMC_ResetSystemErrors failed error %d\n",ResetOut.sErrorID);
}
```

Chapter 5: Motion and Administrative Function Blocks

The Maestro supports the Single Axis (including synchronized master/slave) and group coordinated motions with their related administration function blocks.

The following subsections describe all the function blocks, their usage with examples of their implementation.

5.1 Compliance and Portability

As part of the PLCopen specification, the function blocks are designated specific abbreviations in the function block descriptions, depending on whether the function is described as:

- Basic input and output variables are marked as B in the function block definitions.
- For higher-level systems and future extensions any subset of the extended input and output variables, are marked as E.
- Vendor specific additions are marked with V.

5.2 Function Block States

5.2.1 Single Axis

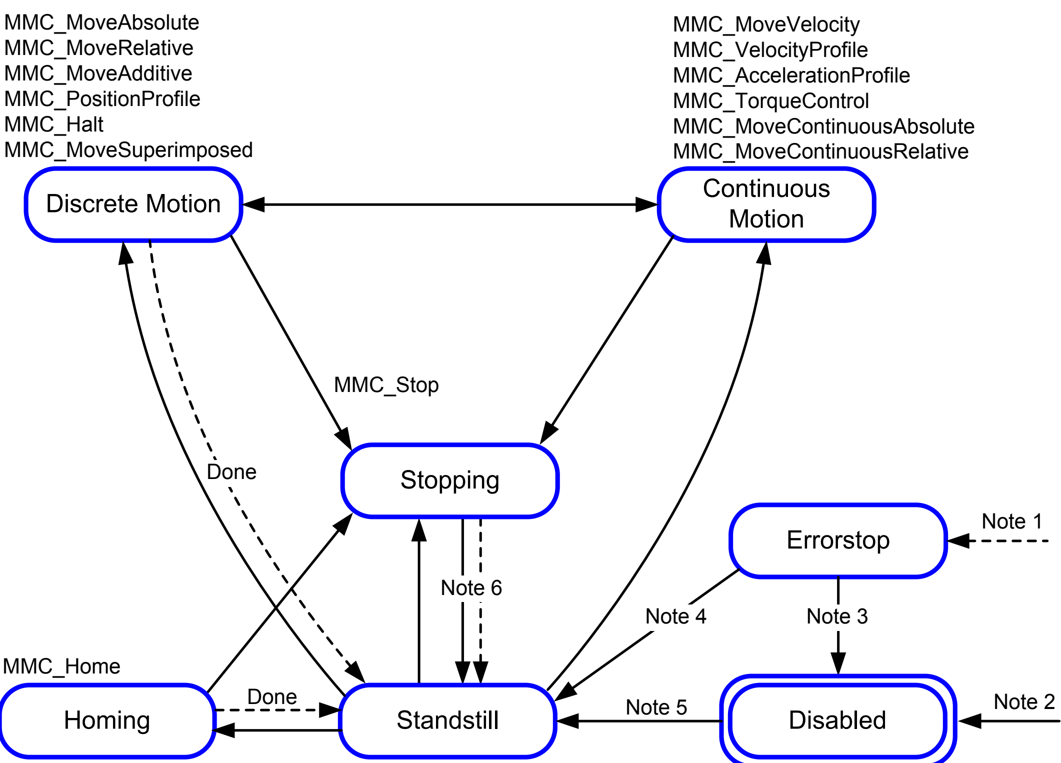


Figure 5-1: Single axis function block states

Note the following in **Figure 5-1**:

1. From any state. An error in the axis occurred.
2. From any state. MMC_Power.Enable = FALSE and there is no error in the axis.

3. MMC_Reset AND MMC_Power.Status = FALSE

4. MMC_Reset AND MMC_Power.Status = TRUE AND MMC_Power.Enable = TRUE

5. MMC_Power.Enable = TRUE AND MMC_Power.Status = TRUE

6. MMC_Stop.Done = TRUE AND MMC_Stop.Execute = FALSE

The axis is always in one of the defined states described in **Figure** 5-1. A change of state is reflected immediately when issuing the corresponding motion command. Any motion command that initiates a transition changes the state of the axis and consequently, modifies the way the current motion is computed. The above state diagram is an abstraction layer of the axis's real state, comparable to the image of the I/O points within a cyclic (PLC) program.

Note: **The response time Immediate is system dependent, coupled to the state of the axis, or an abstraction layer in the software.**

Figure 5-1 is focused on a single axis. Arrows within the state diagram show the possible state transitions between the states. State transitions due to an issued command are shown by full arrows. Dashed arrows are used for state transitions that occur when a command of an axis has terminated or a system related transition (like error related). The motion commands, which transit the axis to the corresponding motion state, are listed above the states. These motion commands may also be issued when the axis is already in the according motion state.

In general, for all states, the axis state can be read using the command MMC_ReadStatus with the following applicable definitions:

Done	Signals that the function block has perform its operation.
Valid	A valid set of outputs is available at the function block.
Disabled	The state Disabled describes the initial state of the axis. In this state, the movement of the axis is not influenced by the function blocks. Power is Off and there is no error in the axis. If the MMC_Power function block is called with Enable=TRUE while being in Disabled, the state changes to Standstill. The axis feedback is operational before entering the state StandStill. Calling MMC_Power with Enable=FALSE in any state except ErrorStop transfers the axis to the state Disabled, either directly or via any other state. Any on-going motion commands on the axis are aborted (CommandAborted).
Error	Signals that an error has occurred within the function block. This may be a communication or axis (servo drive) error.
ErrorStop	ErrorStop is valid as highest priority and applicable in case of an error. The axis can be either power enabled or disabled and can be changed via MMC_Power. However, as long as the error is pending the state remains in ErrorStop. The intention of the ErrorStop state is to bring the axis to stop, if possible. There are no further function block commands accepted until the reset command, MMC_Reset has been performed from the ErrorStop state. The transition to ErrorStop refers to errors from the axis and axis control, and not from the function block instances. These axis errors may also be reflected in the output of the function block instances errors.

ErrorID Error identification

StandStill Power is on, there is no error in the axis, and there are no motion commands active on the axis. The axis is not in motion.

Busy Shows that the function block is operating and new output values can be expected.

Stopping Refer to **Figure** 5-1.

Discrete Motion Refer to **Figure** 5-1

Continuous Motion Refer to **Figure** 5-1

Homing Refer to **Figure** 5-1.

5.2.2 Group of Axes

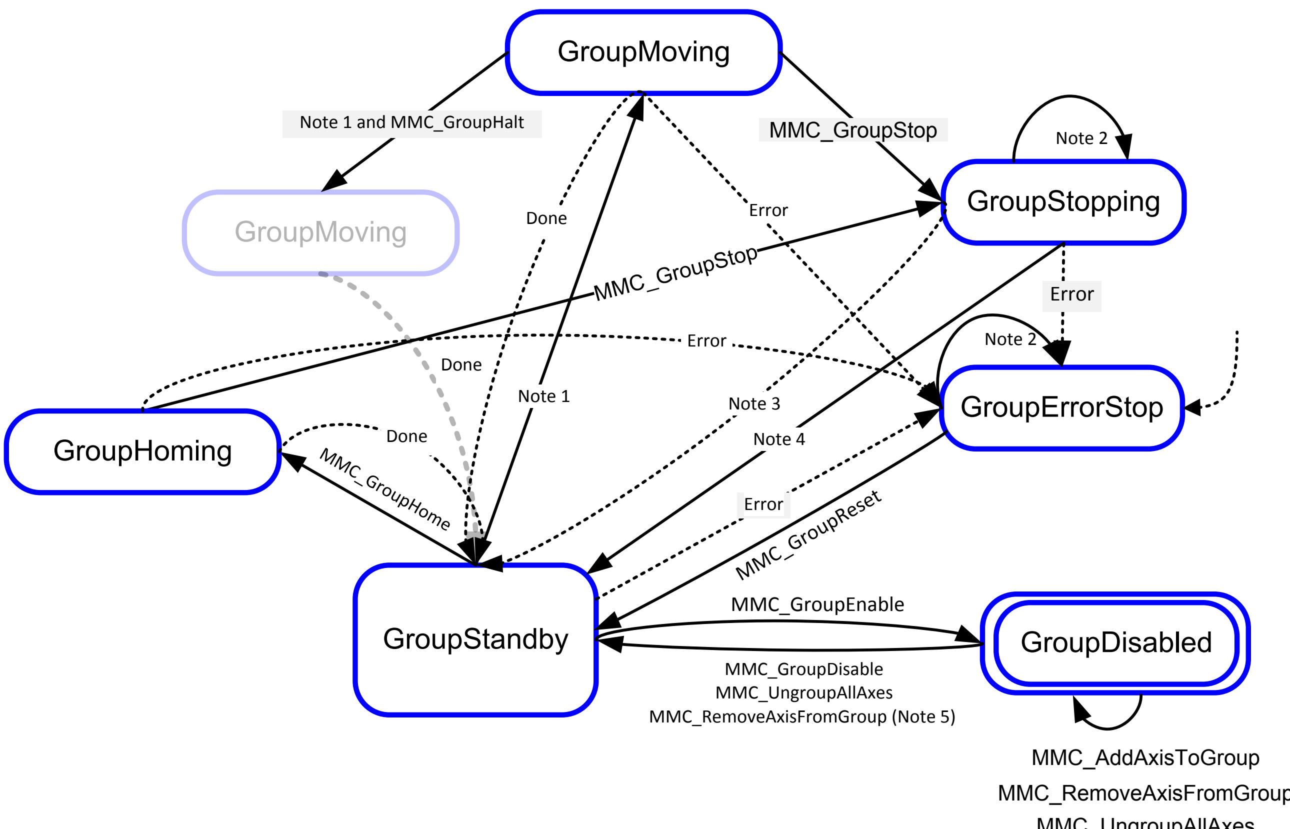


Figure 5-2: Group states

The following notes apply to **Figure 5-2** above.

1 Applicable for all non-administrative (moving) function blocks.

2 In the states GroupErrorStop or GroupStopping, all Function Blocks can be called, although they will not be executed, except MMC_GroupReset for GroupErrorStop and any occurring Error– they will generate the transition to GroupStandby or GroupErrorStop respectively

3 MMC_GroupStop.DONE AND NOT MMC_GroupStop.EXECUTE

4 Transition is applicable if last axis is removed from the group

5 Transition is applicable while group is not empty.

The state-diagram of the group describes the commanded state of a group of axes, and is supplementary to the single axis state diagram. Therefore, where axes are in a group state, the single axis state diagram is also active per axis and the two state diagrams are interdependent. The basic group states are:

GroupDisabled The initial state at power up where a group can be created. Issuing *MMC_GroupEnable* cancels this state.

GroupStandby In this state, the group is enabled and no function block has control on one of the axes in the group. In this state, the group can additionally be altered and homed if needed (State *GroupHoming*).

GroupMoving If a function block has control on (one of the axis of) the group, the state changes to *GroupMoving*.

GroupStopping A special state that deals with the *MMC_GroupStop* command, which automatically transfers to the state *GroupStandby* as soon as Done is SET and Execute is FALSE in MMC_GroupStop.

GroupErrorStop In case an error arises (in one of the axis), the state changes to *GroupErrorStop*, which can only be left via issuing *MMC_ResetGroup*.

The Group motion commands will always lead to a synchronized motion state in the single axis state diagram. In case of a *GroupStandby*, all axes of the group are also in single axis state *StandStill*.

A *GroupErrorStop* will not lead to *ErrorStops* of the grouped axes as the error may only affect the group. In case of a single axis ErrorStop, the Group will also change to *GroupErrorStop* as the single error effects the group.

5.2.3 Function Block Status Bit Masks

The following table list the function block status bit masks for Single axis motion.

Bitwise value	Parameter	Explanation of parameter
0x0200	MC_FB_CMD_ABORT_BIT_MASK	Function Block Command Abort Bit Mask. Vendor Defined. It indicates that the function block has received an abortive command
0x0100	MC_FB_BUSY_BIT_MASK	Function Block Busy State Bit Mask
0x0080	MC_FB_ACTIVE_BIT_MASK	Function Block Active State Bit Mask
0x0040	MC_FB_DONE_BIT_MASK	Function Block Done State Bit Mask
0x0020	MC_FB_ABORTED_BIT_MASK	Function Block Aborted State Bit Mask
0x0010	MC_FB_ERROR_BIT_MASK	Function Block Error State Bit Mask
0x0008	MC_FB_IN_GEAR_MASK	Function Block In Gear State Bit Mask
0x0004	MC_FB_IN_SYNC_MASK	Function Block In Sync State Bit Mask
0x0002	MC_FB_IN_VELOCITY_MASK	Function Block In Velocity State Bit Mask
0x0001	MC_FB_END_OF_PROFILE_MASK	Function Block End Of Profile State Bit Mask

5.3 Axis, Group, Global, Parameters

The axis, group, global, parameters define the MMC_PARAMETER_LIST_ENUM eParameterNumber values. The parameters define the values of the Axis, Group, Global Status in the following function blocks:

MMC_ReadBoolParameter	MMC_GlobalWriteBoolParameter
MMC_ReadParameter	MMC_GlobalWriteParameter
MMC_ReadStatus	MMC_GroupReadParameter
MMC_WriteBoolParameter	MMC_GroupReadBoolParameter
MMC_WriteParameter	MMC_GroupWriteBoolParameter
MMC_GlobalReadBoolParameter	MMC_GroupWriteParameter
MMC_GlobalReadParameter	MMC_GroupReadStatus
MMC_WriteGroupOfParameters	MMC_ReadGroupOfParameter

The functions MMC_WriteGroupOfParameters and MMC_ReadGroupOfParameter have the following advantages:

- Option to Read /Write up to five sets of parameters in a single function, instead of using five functions to Read/Write a single parameter.
- The user is not required to be concerned whether the parameter is global, specific, or real. These functions will handle all types.
- The function MMC_WriteGroupOfParameters can be inserted into a function block queue, to be performed at a later stage, rather than immediately.

The parameters are subdivided into four separate tables, which define the Boolean and Real aspects of the Global and Specific Axis/Vector parameters as shown in Figure 5-3 below. These are listed in the section **5.3 Axis, Group, Global, Parameters**. For a full explanation of these parameters, Refer to the section **19.1 Axis Parameters (Explanations) on page 2015**. These tables will be updated periodically.

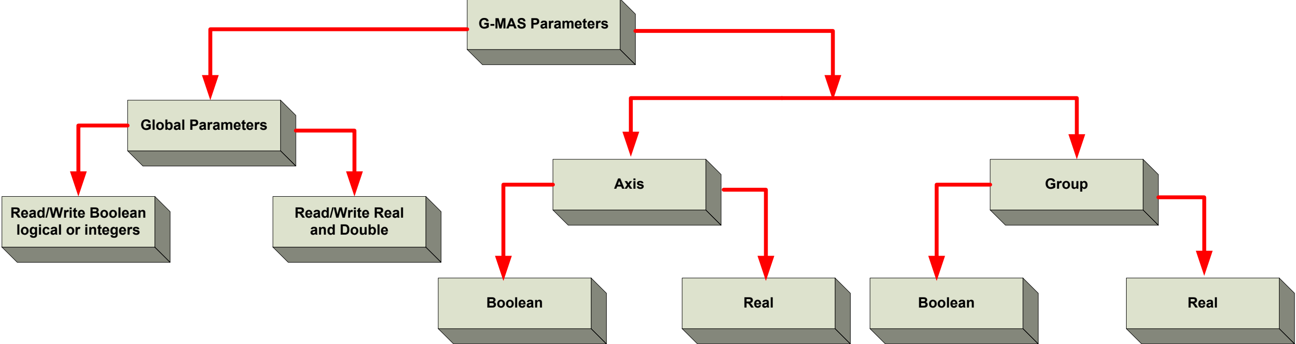


Figure 5-3: Division of the axis parameters

Use the following legend to explain the letter symbols used in the Parameters table heading.

5.3.1 Legend

Symbol	Explanation
Type	Parameter return value type, with possible values: BOOL - 1, REAL - 2, Undefined – 0 Boolean is defined as an integer, whereas Real is a floating point number
Bitwise Type	Parameter Type – bitwise with options: 1 Global variable (first bit) 2 Axis related (second bit) 3 Group related (third bit)
Default Value	Parameter default value (Y x 10^z)
Min. Value	Parameter minimum value (Y x 10^z)
Max. Value	Parameter maximum value (Y x 10^z)
Bitwise Permission	Permission – bitwise, with user values 0 - 6, and all other bits for internal uses. Parameters are Read/Write and/or Save to FLASH on Maestro: 0: Read only (first bit) 1: Read/Write (second bit) 2: Save parameters to file in FLASH (third bit) 3: Internal use 4: Internal use 5: Internal use 6: The parameter is not supported in queued write and when using the WaitUntilCondition function (WriteGroupOfParameters Function). The Parameters are written and saved to the Maestro as long as the G MAS is powered On. When switched Off, the Parameters are no longer written or saved.
Array Size	Array size – if 0, not array. If not zero, size of array.

5.3.2 Parameters Tables

Index	Boolean Global Parameters table - Description	Type	Bitwise Type	Bitwise Permission	Min. Value	Max. Value	Default Value
6	Not in Use						
8	Not in Use						
9	Not in Use						
29	CYCLE_TIME	0	1	69	0	1000000	1000
30	RES_ID	0	1	69	0	1000000	0
49	CONNECTION_TYPE	0	1	69	1	2	2
53	SUPPORT_BLENDED_FB_PENDING	0	1	70	0	1	0
54	FB_CNT_BEFORE_ACTIVE_FB_ON_END_VELOCITIES_RECALCULATION	0	1	70	0	10	0
55	ESTIMATED_TIME_TOBE_ACTIVE_FB_ON_END_VELOCITIES_RECALCULATION	0	1	70	0	10000	60
80	IS_FATAL_ERROR	0	1	69	0	1	0
81	LAST_SYSTEM_ERROR	0	1	69	-32767	32767	0
93	ErCr_AUTOLOAD	0	1	6	0	65535	0
111	GMAS_INIT_STATE	0	1	69	0	1	0

Index	Boolean Axis Parameters table - Description	Type	Bitwise Type	Bitwise Permission	Min. Value	Max. Value	Default Value	Array Size
1	AXIS_MODE	0	2	5	0	2	0	0
2	OP_MODE	0	2	5	1	10	1	0
3	AXIS_STATE	0	6	5	0	2147483647	0	0
5	DRIVE_ID	0	2	5	0	128	128	0
11	ACT_POS	0	2	5	-2147483647	2147483647	0	0
12	AIM_POS	0	2	5	-2147483647	2147483647	0	0
14	ACT_VEL	0	2	5	0	2147483647	0	0
17	IN_CONST_VEL	0	2	69	0	1	0	0
21	IN_AC	0	2	69	0	1	0	0
25	IN_DC	0	2	69	0	1	0	0
33	COMM_I_EV_USR_1	0	2	6	-2147483648	2147483647	0	0
34	COMM_I_EV_USR_1_AUX	0	2	6	-2147483648	2147483647	0	0
37	COMM_I_EV_USR_2	0	2	6	-2147483648	2147483647	0	0
38	COMM_I_EV_USR_2_AUX	0	2	6	-2147483648	2147483647	0	0
41	COMM_I_EV_USR_3	0	2	6	-2147483648	2147483647	0	0
42	COMM_I_EV_USR_3_AUX	0	2	6	-2147483648	2147483647	0	0
45	COMM_I_EV_USR_4	0	2	6	-2147483648	2147483647	0	0
46	COMM_I_EV_USR_4_AUX	0	2	6	-2147483648	2147483647	0	0

Index	Boolean Axis Parameters table - Description	Type	Bitwise Type	Bitwise Permission	Min. Value	Max. Value	Default Value	Array Size
52	MOTOR_ON_CMD_MAX_TIMEOUT_MS	0	2	38	100	40000	7000	0
56	MAX_CURRENT_PARAM	0	2	6	0	65535	2000	0
69	DRIVE_TRACKING_ERROR	0	2	5	-2147483648	2147483647	0	0
71	END_MOTION_REASON	0	6	5	0	256	0	0
72	AXIS_ERROR_MASK	0	2	1	0	65535	0	0
73	LAST_DRIVE_EMERGENCY	0	2	5	0	65535	0	0
74	AXIS_ASYNC_ERROR_CODE	0	2	1	0	65535	0	0
75	FB_DEPTH	0	6	5	0	1000	0	0
76	ANALOG_INPUT	0	2	5	0	65535	0	0
77	EMCY_SET_ERR	0	2	6	0	1	1	0
78	ETHERCAT_DRV_OUTPUT	0	2	6	0	4294967295	0	0
79	DIGITAL_INPUT_LOGIC	0	2	6	0	4294967295	0	0
82	LAST_NODE_INIT_ERROR	0	2	69	-32767	32767	0	0
83	LAST_SDO_ABORT_RAW_DATA	0	2	69	0	255	0	8
85	FAST_REFERENCE	0	2	70	0	4294967295	0	0
87	DIGITAL_INPUT_PARAM	0	2	5	0	4294967295	0	0
88	CAN_DRV_OUTPUT	0	2	6	0	255	0	0
89	DS402_CONTROL_WORD	0	2	5	0	65535	0	0

Index	Boolean Axis Parameters table - Description	Type	Bitwise Type	Bitwise Permission	Min. Value	Max. Value	Default Value	Array Size
90	DS402_STATUS_WORD	0	2	5	0	65535	0	0
91	STATUS_REGISTER	0	6	5	0	4294967295	0	0
102	MODULO_AXIS	0	2	6	0	1	0	0
105	MODULO_ACTUAL_CYCLE	0	2	5	-2147483648	2147483647	0	0
106	MODULO_TARGET_CYCLE	0	2	5	-2147483648	2147483647	0	0
107	AUXILIARY_POSITION	0	2	5	-2147483647	2147483647	0	0
108	UU_ENUM_POS	0	2	6	0	20	0	0
109	UU_ENUM_VEL	0	2	6	0	20	0	0
110	FAST_REFERENCE_MODE	0	2	6	0	1	0	0

Index	Boolean Group Parameters table - Description	Type	Bitwise Type	Bitwise Permission	Min. Value	Max. Value	Default Value
4	GROUP_ID	0	4	5	0	128	128
28	SPATIAL_OPTION	0	4	6	0	1	0
92	MCS_LIMIT_REGISTER	0	4	5	0	4294967295	0

Index	Real Axis Parameters table - Description	Type	Bitwise Type	Bitwise Permission	Min. Value	Max. Value	Default Value
10	SET_POS	1	2	14	-1.00E+15	1.00E+15	0
13	SET_VEL	1	2	13	0	1.00E+11	0
31	SW_LIM_HIGH	1	2	6	-1.00E+15	1.00E+15	1.00E+15
32	SW_LIM_LOW	1	2	6	-1.00E+15	1.00E+15	-1.00E+15
35	COMM_F_EV_USR_1	1	2	6	-2147483647	2147483647	0
36	COMM_F_EV_USR_1_AUX	1	2	6	-2147483647	2147483647	0
39	COMM_F_EV_USR_2	1	2	6	-2147483647	2147483647	0
40	COMM_F_EV_USR_2_AUX	1	2	6	-2147483647	2147483647	0
43	COMM_F_EV_USR_3	1	2	6	-2147483648	2147483647	0
44	COMM_F_EV_USR_3_AUX	1	2	6	-2147483648	2147483647	0
47	COMM_F_EV_USR_4	1	2	6	-2147483648	2147483647	0
48	COMM_F_EV_USR_4_AUX	1	2	6	-2147483647	2147483647	0
50	ERROR_CORRECTION_PARAM	1	2	5	-1.00E+15	1.00E+15	0
63	TARGET_RADIUS	1	2	6	0	1.00E+15	10000
64	TARGET_TIME	1	2	70	0	6000	0
67	MAX_TRACKING_ERROR_POSITION	1	2	6	0	1.00E+15	1.00E+15
68	MAX_TRACKING_ERROR_TIME	1	2	70	0	6000	0
70	GMAS_TRACKING_ERROR	1	2	5	-1.00E+15	1.00E+15	0

Index	Real Axis Parameters table - Description	Type	Bitwise Type	Bitwise Permission	Min. Value	Max. Value	Default Value
86	SET_ACDC_PARAM	1	2	5	-1.00E+14	1.00E+14	0
94	MAX_DESIRED_TORQUE_PARAM	1	2	6	0	1.00E+05	1.00E+05
95	MAX_TORQUE_VELOCITY_PARAM	1	2	6	0	2.00E+08	2.00E+08
96	MAX_TORQUE_ACCELERATION_PARAM	1	2	6	0	2.00E+09	2.00E+09
97	USER_UNITS_POSITION	1	2	6	1	2147483647	1
98	USER_UNITS_KINEMATICS	1	2	6	1	2147483647	1
99	POSITION_OFFSET	1	2	5	-1.00E+15	1.00E+15	0
100	TARGET_POS_UU	1	2	134	-1.00E+15	1.00E+15	0
101	ACTUAL_POS_UU	1	2	134	-1.00E+15	1.00E+15	0
103	MODULO_LOW	1	2	6	-1.00E+15	1.00E+15	-1.00E+15
104	MODULO_HIGH	1	2	6	-1.00E+15	1.00E+15	1.00E+15

Index	Real Global Parameters table - Description	Type	Bitwise Type	Bitwise Permission	Min. Value	Max. Value	Default Value
7	Not in Use						

Index	Real Group Parameters table - Description	Type	Bitwise Type	Bitwise Permission	Min. Value	Max. Value	Default Value	Array Size
15	MAX_VEL	1	6	14	-1.00E+11	1.00E+11	1.00E+11	0
18	SET_AC	1	6	21	0	1.00E+14	0	0
19	MAX_AC	1	6	22	0	1.00E+14	1.00E+14	0
22	SET_DC	1	6	21	0	1.00E+14	0	0
23	MAX_DC	1	6	22	0	1.00E+14	1.00E+14	0
26	MAX_JERK	1	6	30	0	1.00E+17	1.00E+17	0
51	S_Factor_For_Polynomial_Transition	1	4	6	0	2	1.4	0
57	LIMIT_STOP_DECELERATION	1	6	22	0	1.00E+14	1.00E+14	0
58	LIMIT_STOP_JERK	1	6	30	0	1.00E+17	1.00E+17	0
59	SET_VECTOR_VEL	1	4	13	0	1.00E+14	0	0
60	MCS_SW_LIMIT_LOW_POS_ARRAY	1	4	6	-1.00E+15	1.00E+15	-1.00E+15	16
61	MCS_SW_LIMIT_HIGH_POS_ARRAY	1	4	6	-1.00E+15	1.00E+15	1.00E+15	16
62	MCS_S_DIRECTION	1	4	70	-1.00E+15	1.00E+15	0	0
65	PROFILE_TIME	1	6	69	0	400000	0	0
66	OVERALL_MOTION_TIME	1	6	69	0	400000	0	0
84	SPEED_OVERRIDE	1	6	6	0	1	1	0

5.4 Axis Status

For specific function blocks the Axis Status and Bit Mask is defined. The axes status bit mask derives from 4 btye group within the Maestro allocated to this mask separated into the various Common, Group and Single axes, parameters as shown in **Figure 5-4**.

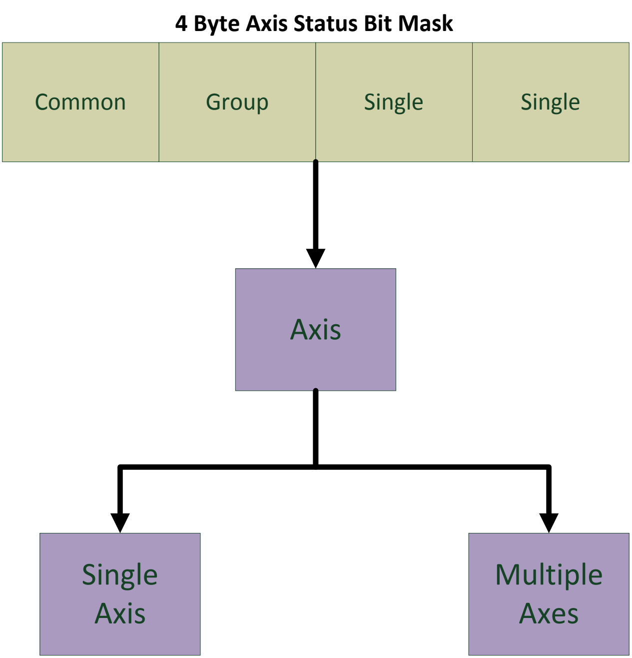


Figure 5-4: Status Bit Mask

The following table describes the Axis Status Bit Masks and their values. These are the outputs of the variable **ulState** related to the axis state (returned in the MMC_ReadStatus function block):

Bitwise value	Parameter	Explanation of parameter
0x80000000	NC_AXIS_DONE_MASK	Axis/Group Motion Done Bit Mask
0x40000000	NC_AXIS_VALID_MASK	Axis/Group Data Valid Bit Mask
0x20000000	NC_AXIS_ERROR_MASK	Axis/Group Error Bit Mask
0x10000000	NC_AXIS_PENDING_MASK	Axis/Group Pending Bit Mask
0x00000800	NC_AXIS_BUSY_MASK	Axis Busy State Bit Mask
0x00000400	NC_AXIS_ERROR_STOP_MASK	Axis **Error Stop** State Bit Mask
0x00000200	NC_AXIS_DISABLED_MASK	Axis Disabled State Bit Mask
0x00000100	NC_AXIS_STOPPING_MASK	Axis Stopping State Bit Mask
0x00000080	NC_AXIS_STAND_STILL_MASK	Axis Stand Still State Bit Mask
0x00000040	NC_AXIS_DISCRETE_MOTION_MASK	Axis Discrete Motion State Bit Mask
0x00000020	NC_AXIS_CONTINUOUS_MOTION_MASK	Axis Continuous Motion State Bit Mask
0x00000010	NC_AXIS_SYNCHRONIZED_MOTION_MASK	Axis Synchronized Motion State Bit Mask

0x00000008	NC_AXIS_HOMING_MASK	Axis Homing State Bit Mask
0x00000004	NC_AXIS_CONSTANT_VELOCITY_MASK	Axis Constant Velocity State Bit Mask
0x00000002	NC_AXIS_ACCELERATING_MASK	Axis Accelerating State Bit Mask
0x00000001	NC_AXIS_DECELERATING_MASK	Axis Decelerating State Bit Mask

The following table is the of status bit masks for multiple axes.

Bitwise value	Parameter	Explanation of parameter
0x00020000	NC_GROUP_STANDBY_MASK	Group Standby State Bit Mask
0x00010000	NC_GROUP_DISABLED_MASK	Group Disabled State Bit Mask
0x00008000	NC_GROUP_HOMING_MASK	Group Homing State Bit Mask
0x00004000	NC_GROUP_ERROR_STOP_MASK	Group Error State Bit Mask
0x00002000	NC_GROUP_MOVING_MASK	Group Moving State Bit Mask
0x00001000	NC_GROUP_STOPPING_MASK	Group Stopping State Bit Mask

By definition, both the Axis, Vector, Accelerating (AC), Decelerating (DC), and Constant Velocity, State Bits will be set according to the sign of the velocity derivative. Therefore, in reference to Figure 5-5, the example shows the changes in state between AC, DC, and at constant velocity.

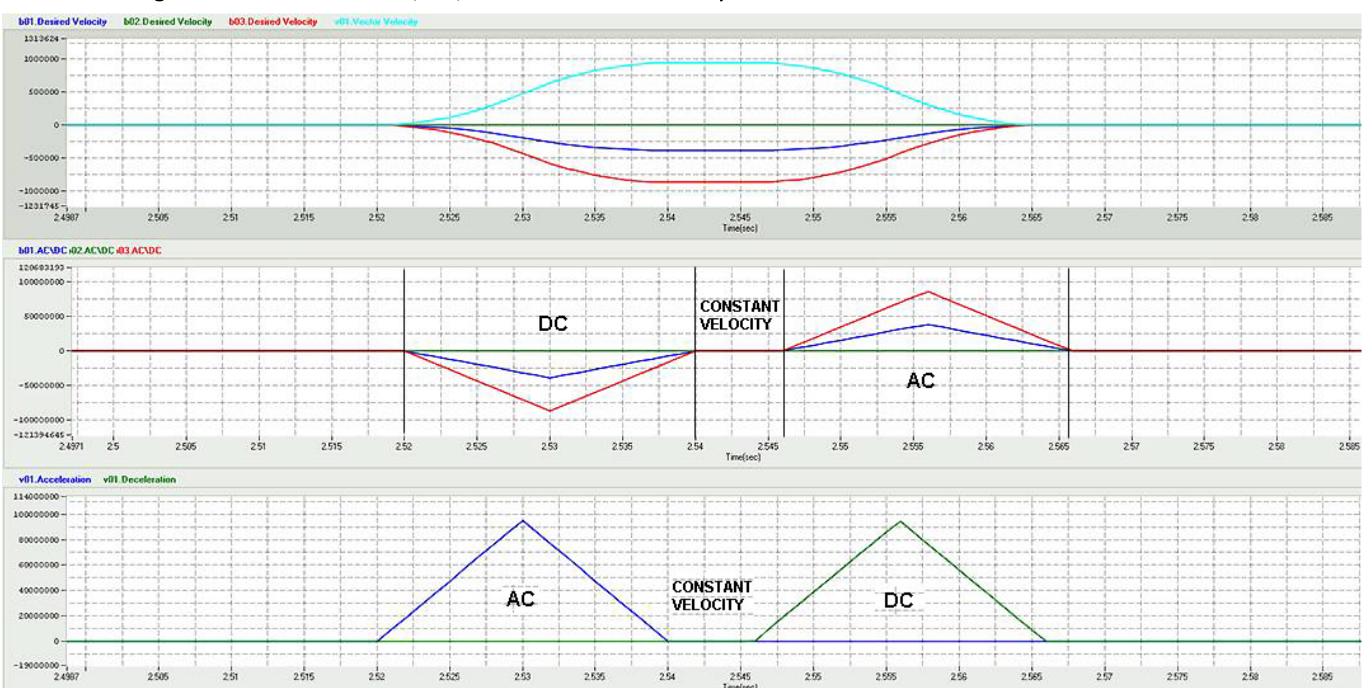


Figure 5-5 Example of Acceleration/Deceleration axis bits

To use this table, consider the following example of an error received from the Maestro; 0x60000408. The error is broken into the following sections:

Bitwise value	Explanation of parameter
0x00000008	Axis Homing State Bit Mask
0x00000400	Axis **Error Stop** State Bit Mask
0x20000000	Axis/Group Error Bit Mask
0x40000000	Axis/Group Data Valid Bit Mask
0x60000408	TOTAL

The diagnostics of the error describes an Axis or group of axes data validity error causing an error stop during homing.

5.4.1 Blended Behavior Mechanism

FB1	FB2	FB3	FB4	FB5	FB6	FB7	FB8		FB1	FB2	FB3	FB4
0	0	0	0	1	1	1	1		0	0	0	1

Within the Maestro, function blocks are queue aligned in the execution process. Elmo has designed an execution process using the *ucExecute* Boolean variable entered to each axis group function block queue, with a specific execute flag bit. The execute bit is an input to the motion function blocks. This input defines the following:

- PreCalc - whether the recalc end velocities routine will be executed
- RealTime - whether the current function block will be executed.

Each time the execute bit "1"is entered with a function block, it causes all the previous function blocks to set their execute state to "1". For example, for five function blocks in a queue:

- Enter FB1 Execute bit "0" - no motion.
- Enter FB2 Execute bit "0" - no motion.
- Enter FB3 Execute bit "0" - no motion.
- Enter FB4 Execute bit "0" - no motion.
- Enter FB5 Execute bit "1" - perform recalc end velocities and start motion of all five function blocks.

In another example the following occurs in a function block queue:

- Enter FB1 Execute bit "0" - no motion.
- Enter FB2 Execute bit "0" - no motion.
- Enter FB3 Execute bit "1" - perform recalc end velocities and start motion of all three function blocks.
- Enter FB4 Execute bit "0" - no additional motion(motion limited by FB3).
- Enter FB5 Execute bit "0" - no additional motion(motion limited by FB3.

- Delay - wait until the motion of the first three function blocks is ended.
- Now, two function blocks remain in the queue (FB4, FB5), waiting, but not executed.
- Enter FB6 Execute bit "1" – to perform recalc end velocities and start motion of all three function blocks present in the queue (FB4, FB5, FB6).

Each time when the last inserted function block is inserted with execute bit "0", this causes the Pending bit to rise in the Axis state mask.

5.4.2 Special function block insertion mechanism

There are two special circumstances in the transition\arch insertion:

- Insert transition when enter to pending (pending bit is rising on current function block – execute bit 0 after a function block with execute bit 1)
- Insert transition when the last function block is active

For the Pending situation, the Maestro allows the user to insert the second function block in blended mode with a transition, and continue the axes operation, only when the global parameter MMC_SUPPORT_BLENDED_FB_PENDING (53) is set to 1 (true). Otherwise, the operation is halted, and the second function block is inserted in MC_BUFFERED_MODE mode. The Boolean Global Parameter MMC_SUPPORT_BLENDED_FB_PENDING described in Parameters Tables

above, is initialized to 0 and can be changed using one of the following functions, and changing the parameter to 1:

- MMC_WriteBoolParameter
- MMC_GlobalWriteBoolParameter
- MMC_GroupWriteBoolParameter

In the second situation, it is not normally possible to insert a function block with blended mode (and transition arc) when the last function block is an active function block in an axis or group. To solve this situation, and allow the axes operation to effectively continue, the BufferMode is changed to MC_BUFFERED_MODE and the TransitionMode to MC_TM_NONE_MODE in the function block to be inserted. The transition arc is not inserted. If the user requires that the function block is inserted with an arc, then insert the previous function block with execute bit 0. To obtain further information regarding the status of the function block list, invoke MMC_GetFBDepth. If the Depth value is greater than one, then the last function block is not in active execution.

5.5 Function Block Execution Order

This section describes the function block queue execution order. Motion function blocks are present in the "main" queue, while each queue member has a pointer to an "auxiliary" queue containing the administrative function blocks, as shown in Figure 5-6. The following Administrative Function Blocks are added to maintain the logic of this approach:

- Dwell (Precise wait).
- WriteGroupOfParameters Ability to write up to 5 parameters in a single function block call.
- Stop
- WaitUntilConditionFB
- Halt
- Power

In the following example, the 2nd MoveAbsolute will not be called until the two Dwells and the SetParameter functions have completed.

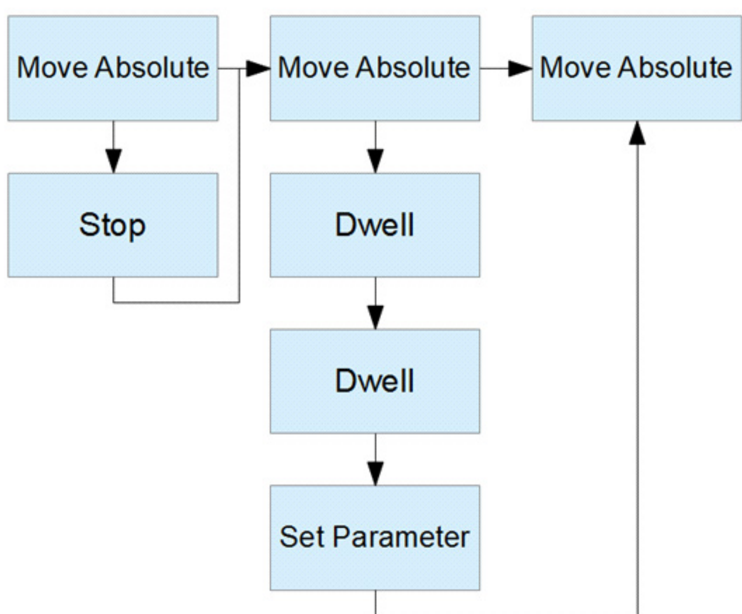


Figure 5-6: Motion and Administrative Function Blocks

The function blocks queue is implemented as a bi-directional linked list, and in some cases it is necessary to distinguish between motion traversing function blocks, e.g. profiler pre-calculations, and entire traversing function blocks (all), e.g. marking them as aborted. Therefore, two pairs of pointers are required, one to connect between the motion function blocks (mprev, mnext), and the other to manage the "auxiliary" administrative queue (aprev, anext). The system is then able to traverse either through the motion function block's queue, or through all function blocks, providing a generic solution for all function block type handling.

5.5.1 Classification

Generally, administrative function blocks can be divided the into two categories:

Immediate execution	The execution time for these function blocks is immediate, and the active function block can be advanced in the same cycle, e.g. SetParameter.
Non-immediate	The execution of these function blocks might take a while, and if inserted, there is no need for the profiler to traverse past the function block, e.g. Dwell.

5.5.2 Flags

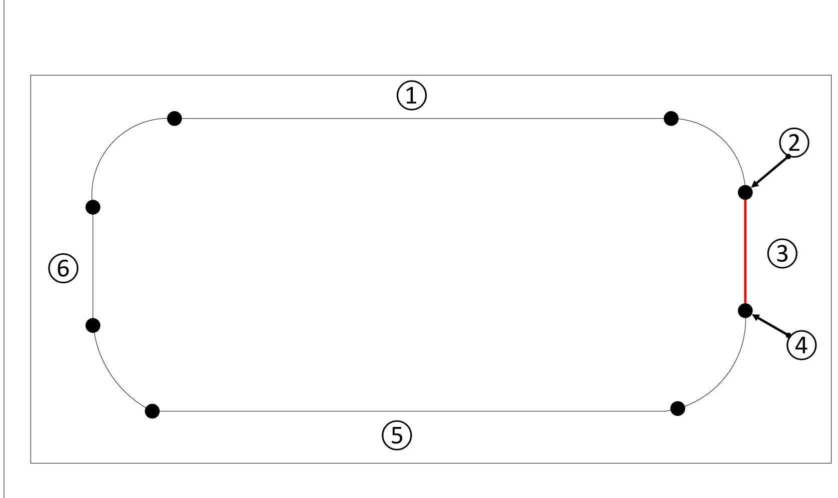
Each motion function block in a queue is a potential root (head) of an administrative function blocks list. This root (head) now consists of a pre-calculation flag, which changes to negative when any non-immediate function block is inserted to the current top administrative queue. This informs the profiler service functions to ignore this function block, and return NULL (for example for GetPrevFB API).

However, when an immediate execution function block is processed, the system can move to the next function block in the same clock interrupt. This requires setting the parameters, which will affect the next function block, and execute the next function block immediately, in the same clock interrupt.

5.5.3 Examples

The following examples demonstrate the principle involved in using this approach to the Administrative Function Block handling.

5.5.3.1 2D Motion with Precise Torque Change

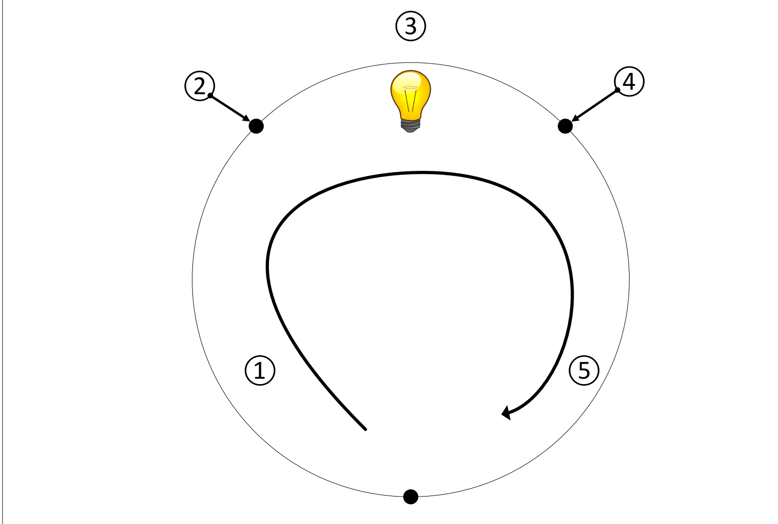


Infinite sequence where we want to set high current to administrative axis that make a push.

① - 2D linear move.
② - Set high max current value.
③ - 2D linear move.
④ - Set low max current value.
⑤ - 2D linear move.
⑥ - 2D linear move.

The motion sequence performed with blending transitions between function blocks, all the sequence without stopping.

5.5.3.2 IO Change at Precise Location

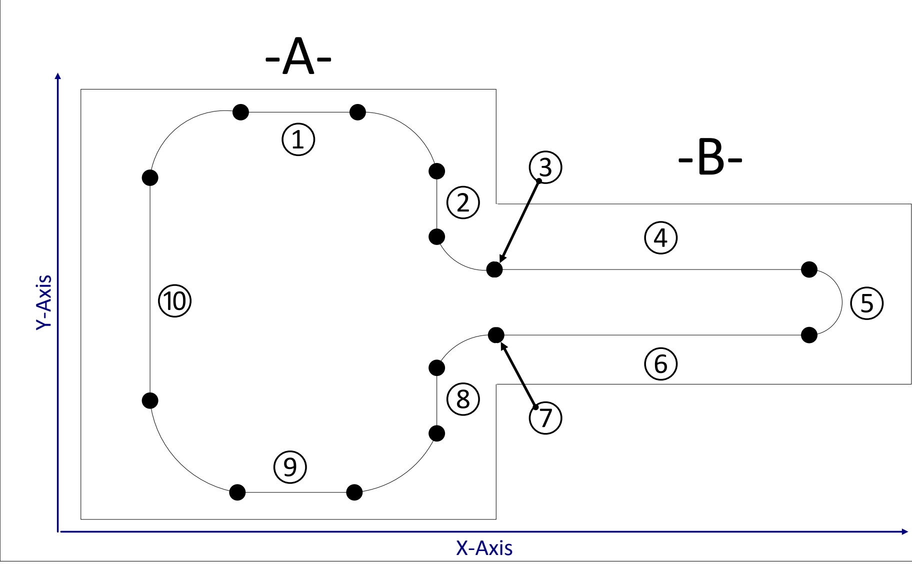


Infinite sequence where we want to turn light on, using digital output in a bounded specific location.

① - 2D Circle motion.
② - Set digital output.
③ - 2D Circle motion.
④ - Clear digital output.
⑤ - 2D Circle motion.

The motion sequence performed with blending transitions between function blocks, all the sequence without stopping.

5.5.3.3 Change Software Limits of Y axis During Motion



Infinite sequence where we want to change the SW limits of 'Y' axis during motion according to the area where we work.

① - 2D linear move.
② - 2D linear move.
③ - Update High\Low SW limits of 'Y' for "-B-".
④ - 2D linear move.
⑤ - 2D circle move.
⑥ - 2D linear move.
⑦ - Update High\Low SW limits of 'Y' for "-A-".
⑧ - 2D linear move.
⑨ - 2D linear move.
⑩ - 2D linear move.

The motion sequence performed with blending transitions between function blocks, all the sequence without stopping.

5.6 Administrative Function Block Handling (for both Single and Multiple Axis)

The function block **MMC_WaitUntilConditionFB** detailed in section **8.1.45** is inserted to the queue as an administrative function block When activated, the Maestro performs a user pre-defined condition test which if results in "TRUE", indicates that this function block is finished, and the Maestro moves to the next function block. This conditional function block does not necessarily need to belong to the axis queue upon which it is operating. The operation of this function block allows synchronization of numerous axes that are not part of a group, to start their motion together. In addition, it allows synchronization of numerous Maestro's on the network by starting a motion when a specific bit on a shared IO is raised.

In order to define the condition the following data is provided:

- Variable to check (Source Value)
- Reference value
- Logic operation

5.6.1 Source Value

The Source Value is a parameter from the list of Axis, Group, Global, Parameters detailed in section **5.3.**

In order to define parameter, the user inserts the following:

- Axis reference
- Parameter enumerator
- Parameter index (only for array type)

5.6.2 Reference Value

Constant "double" value, provided by the user.

5.6.3 Logic Operation

This is the list of supported logic operations:

- EQUAL
- LOWER
- HIGHER
- LOWER_EQUAL
- HIGHER_EQUAL
- MASK_AND
- MASK_LAST

5.6.4 Function Usage Situations

If it is required to synchronize the Motion Start between 30 axes, insert this conditional function block to each axis and then insert a requested motion in the next function block. Once the condition is fulfilled, all motions will start simultaneously.

Node #1
Condition FB → Motion FB → Motion FB → Motion FB

Node #2
Condition FB → Motion FB → Motion FB → Motion FB

Node #3
Condition FB → Motion FB → Motion FB → Motion FB

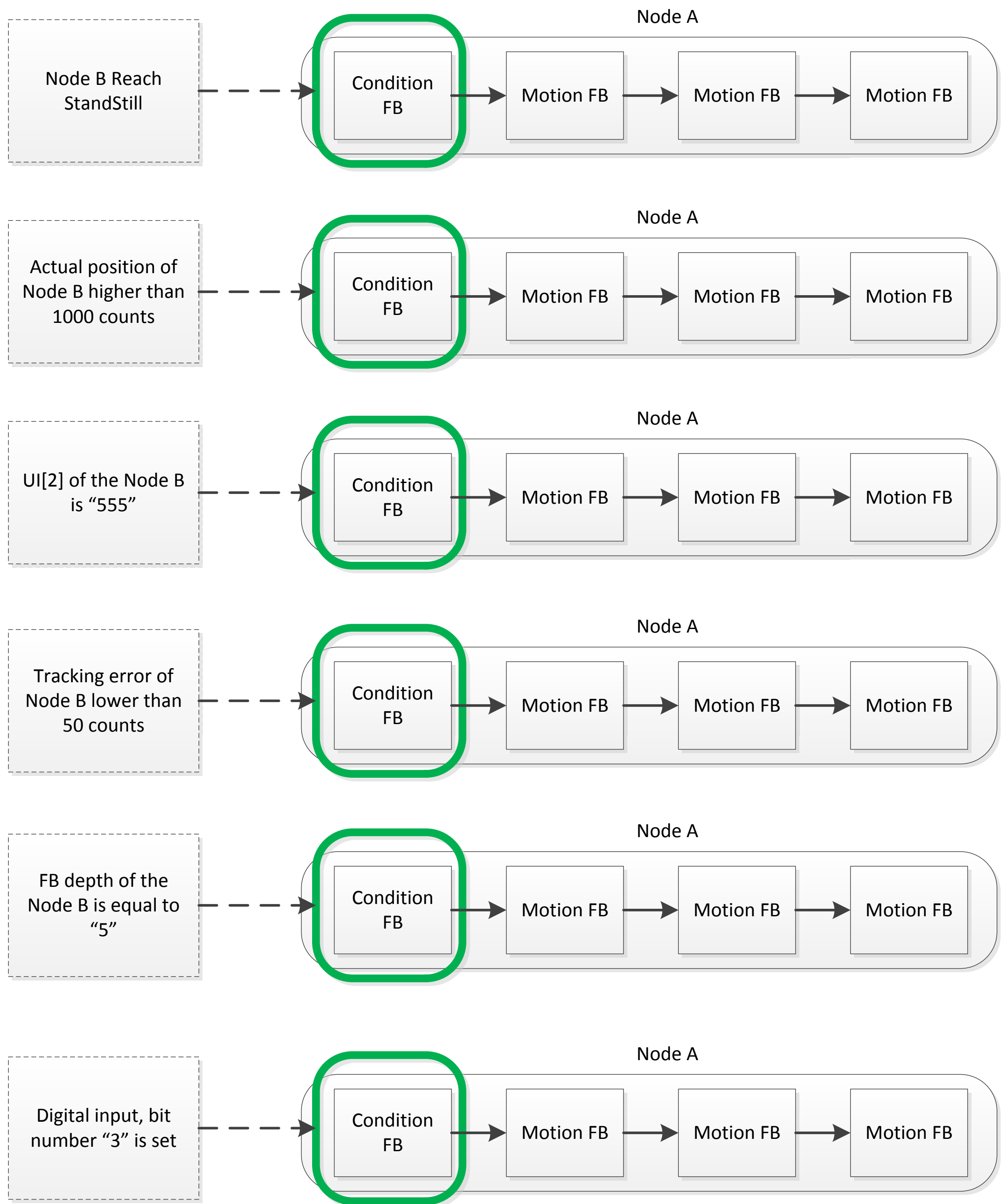
Same Condition

Node #28
Condition FB → Motion FB → Motion FB → Motion FB

Node #29
Condition FB → Motion FB → Motion FB → Motion FB

Node #30
Condition FB → Motion FB → Motion FB → Motion FB

The axes / groups can be synchronized when the synchronization point is defined using any type of parameter and any type of logical comparison. The comparable parameters can refer to any desired node (Single/Multi) selected:


Node A
Node B Reach StandStill
Condition FB
Motion FB
Motion FB
Motion FB
Node A
Actual position of Node B higher than 1000 counts
Condition FB
Motion FB
Motion FB
Motion FB
Node A
UI[2] of the Node B is "555"
Condition FB
Motion FB
Motion FB
Motion FB
Node A
Tracking error of Node B lower than 50 counts
Condition FB
Motion FB
Motion FB
Motion FB
Node A
FB depth of the Node B is equal to "5"
Condition FB
Motion FB
Motion FB
Motion FB
Node A
Digital input, bit number "3" is set
Condition FB
Motion FB
Motion FB
Motion FB

5.6.5 Download Firmware and Errors

The following download firmware function blocks are not detailed in this document, but will be necessary to download and update the firmware. For further details, refer to Elmo.

Function Block	Services and Operation
MMC_GetVerPath	Download firmware variables involving update and read firmware version.
MMC_DownloadFirmware	
MMC_ReadDownloadVersionStatus	
MMC_SetVerPath	

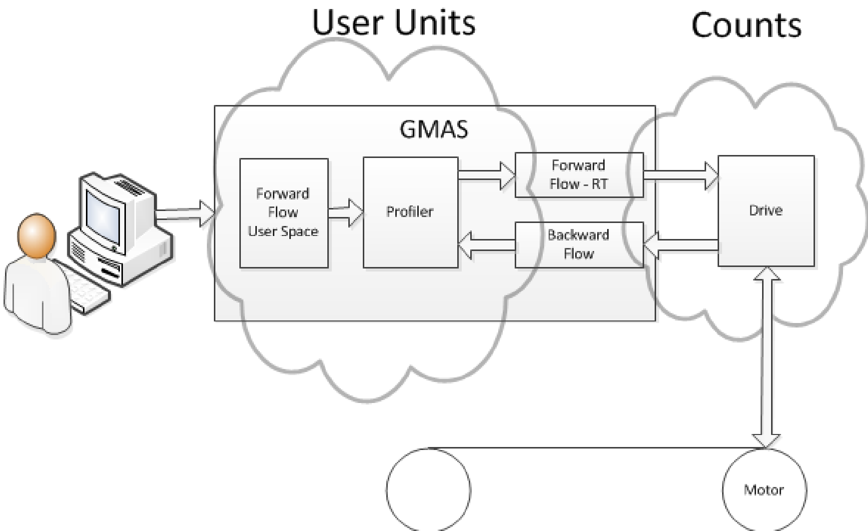
The following list describes the firmware download errors possible during an update operation.

Error Parameter	ID	Explanation
DOWNLOAD_VERSION_ERROR_FS	-1	Download Firmware. Target file system error
DOWNLOAD_VERSION_ERROR_TFTP	-2	Download Firmware. TFTP fail
DOWNLOAD_VERSION_ERROR_BURN	-3	Download Firmware. Copy to flash is fail.
DOWNLOAD_VERSION_ERROR_CRC	-4	Download Firmware. Wrong CRC.
DOWNLOAD_VERSION_NON_LINUX_IMAGE	-5	Download Firmware. Image is not Linux type.
DOWNLOAD_VERSION_ERROR_DF_IN_PROCESS	-6	Download Firmware. Error DF in process.
DOWNLOAD_VERSION_DIFFERENT_MD5SUM	-7	Wrong MD5 CHKSum
FALLBACK_VERSION_ERROR_CRC	-8	Fallback version has CRC error !!!
BOOTED_WITH_FALLBACK_VER_ERROR	-9	Booted with fallback error

5.7 User Units

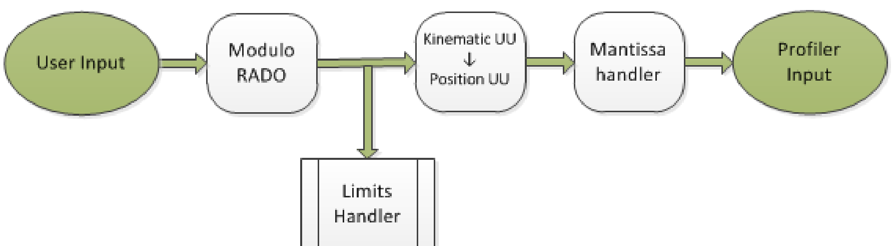
Until now, the Maestro did not support User Units, and therefore the user entered motion parameters and received the axis positions in axis units. In order to work in degrees, or inches – the user had to calculate the ratio in Counts, and then perform the motion, as the drive only measures Counts. Now, this is all performed automatically. The drive continues to work in Counts, when operating via the Maestro – the Maestro converts everything to user units.

User Units defines a conversion process wherby absolute units such as Degrees, RPM, RPM^2 are changed to a standard user unit, or physical axis units, allowing easy and faster communication between the Maestro and slave drives. This increases the speed of communication between the Maestro and various servo drives, as the only back conversion necessary is between the Maestro and the PC or other interface as shown in the diagram below.

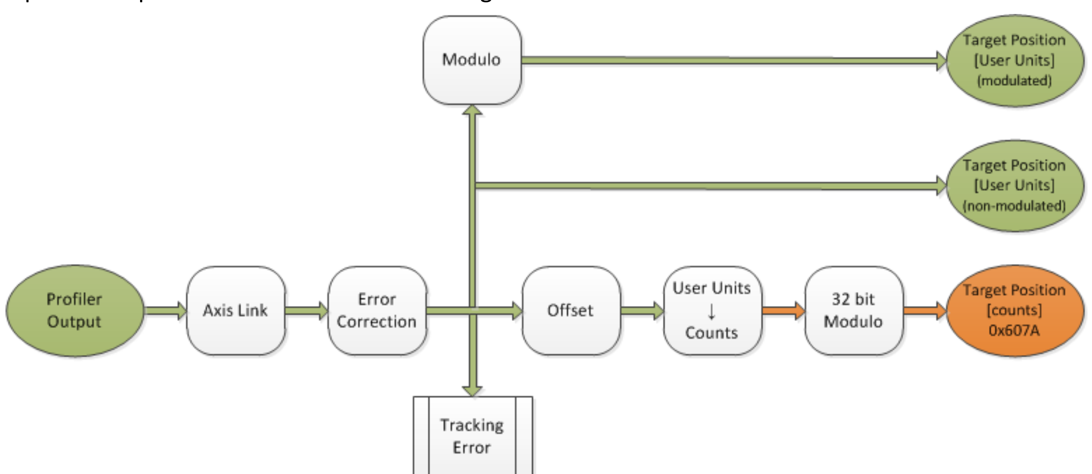


5.7.1 System Flow Forward

When data is input to the Maestro, it undergoes a conversion to User Units via the profiler input as shown in the following diagram. The user inputs may consist of position kinematics, modulo, etc.

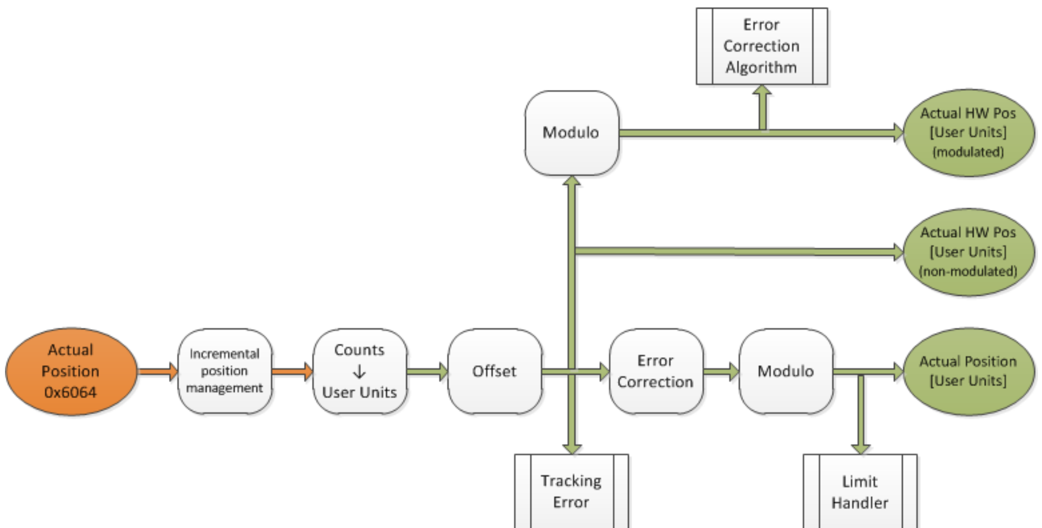


The output of the profiler is shown in the next diagram.

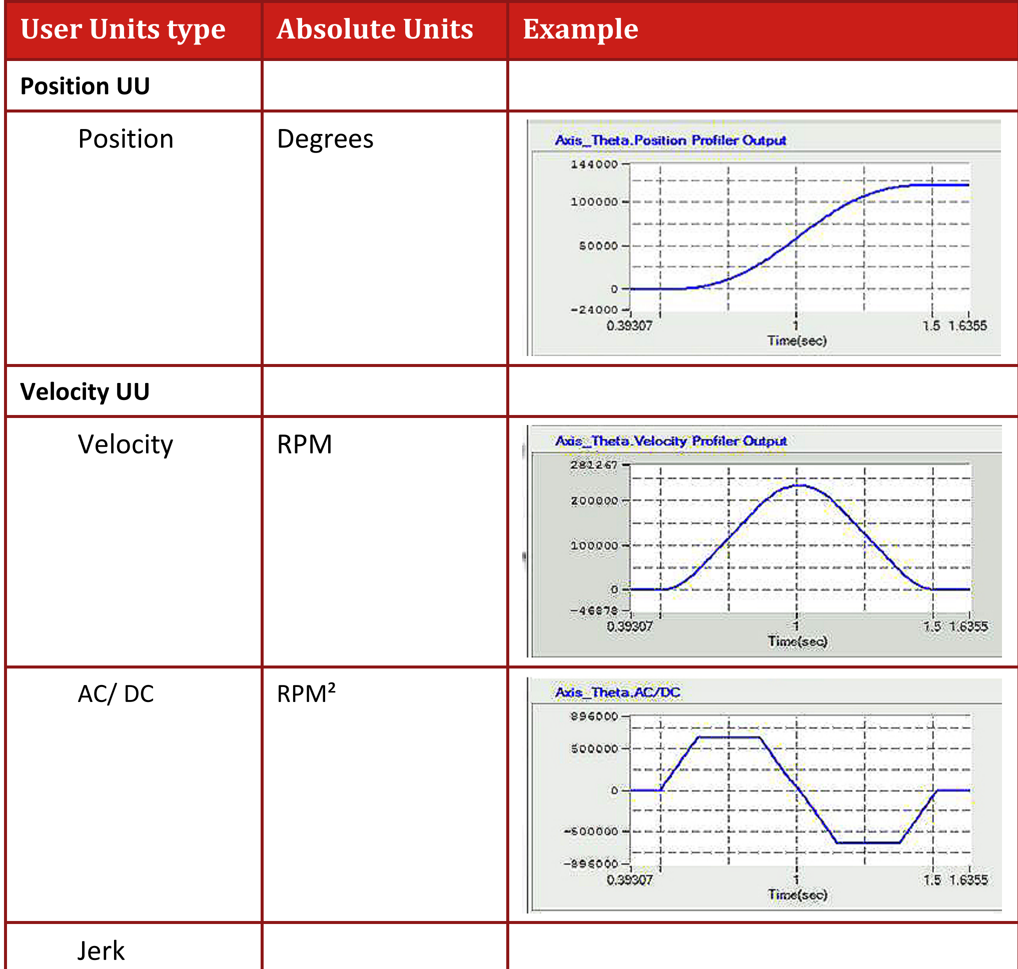


5.7.2 System Flow Backward

The system backward flow described in the next diagram shows the feedback from the drive to the Maestro. The Actual position measured at the drive is in counts is transfered to the Maestro in converted Actual Position user units.



As a result of this conversion, programming the drive via the Maestro becomes more intuitive, allowing different units for position and kinematics. The following example displays the flexible optional conversions.



User Units type	Absolute Units	Example
Position UU		
Position	Degrees	Axis_Theta.Position Profiler Output
Velocity UU		
Velocity	RPM	Axis_Theta.Velocity Profiler Output
AC/ DC	RPM²	Axis_Theta.AC/DC
Jerk		

Therefore there are two different user units the user must set:

- Position UU (User units) – For the Position variables
- Velocity UU(User Units) – For the Velocity, AC/DC, Jerk variables

After the user units are set, all the following are converted into user units:

- Motion commands.
- Position readings.
- Error correction
- Limit management
- Modulo
- Set Position
- Axis Link
- Tracking Error

Additionally, the user units are supported in all motion modes:

Single Axis

NC

NON-NC

Multi Axis

ACS

MCS

Special Kinematics

5.7.3 Practical Interface

Use the functions SetParameter or WriteGroupOfParameters with the User Unit parameters set in section 5.3 Axis, Group, Global, Parameters, and the parameters tables in section 5.3.2, e.g.

USER_UNITS_POSITION = 94 – define the position user units ratio.

USER_UNITS_VELOCITY = 95 – define the velocity user units ratio (Vel/AC/DC/Jerk).

Note that the ratio is a double precision floating point variable, which can only be a positive number, but can be greater than 1.

5.8 Single Axis Motion Control

Motion Control describes the motion of the single axes, either NC or Distributed according to the definitions described in section **2.2 Maestro Operation Modes** and **2.2.3 Maestro Axes and Node Definitions** above. The following single axis function blocks are described:

Single Axis
MMC_Halt
MMC_Home
MMC_HomeDS402
MMC_MoveAbsolute
MMC_MoveAdditive
MMC_MoveRelative
MMC_MoveVelocity
MMC_Stop
MMC_MoveAbsoluteRepetitive
MMC_MoveRelativeRepetitive
MMC_MoveAdditiveRepetitive

Since the Maestro API operates, equally well for the SimplIQ , Gold, and Platinum range of motion controllers, for each of the motion axis function blocks, the motion profile must be defined whether as NC or Distributed, with the appropriate mode applicable to each possible variation.

Mode	Controller	Motion	
		NC	**Distributed**
DS-402	SimplIQ	Interpolated position mode (IP)	Depends on Profile of drive: Profile position mode (PP) Profiled velocity mode (PV) Torque profiled mode (PT) Homing mode (HM)
	Gold	Cyclic sync position mode Cyclic sync velocity mode	Depends on Profile of drive: Profile position mode (PP) Profiled velocity mode (PV) Homing mode (HM)

Mode	Controller	Motion	
		NC	**Distributed**
	Platinum	Cyclic sync position mode Cyclic sync velocity mode	Depends on Profile of drive: Profile position mode (PP) Profiled velocity mode (PV) Homing mode (HM)

5.8.1 MMC_Halt

Call this function to command a controlled motion stop for a specific axis.

MMC_LIB_API int MMC_HaltCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_HALT_IN* pInParam,
OUT MMC_HALT_OUT* pOutParam
);

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_HALT_IN** input structure that receives the HALT Information

pOutParam

Points to the **MMC_HALT_OUT** output structure receiving information as a result of calling the HALT function.

Remarks

The axis is moved to the state Discrete Motion, until the velocity is zero. With the output set to Done, the state is transferred to StandStill.

MMC_Halt is used to stop the axis under normal operation conditions.

If this command is active, the next command can be issued, e.g. a driverless vehicle detects an obstacle and needs to stop. When MMC_Halt is issued, then, before the standstill is reached the obstacle is removed and the motion can be continued by setting another motion command, so that the vehicle continues its motion and does not stop.

Scope

MMC_Halt will only operate from Standstill or Continuous Motion. It will not operate from Stopping, Homing, ErrorStop, or Disabled. Refer to the State diagram in **Figure 5-1**.

MMC_HALT_IN Structure

```
typedef struct{
        float fDeceleration;
        float fJerk;
        MC_BUFFERED_MODE_ENUM eBufferMode;
        unsigned char ucExecute;
}MMC_HALT_IN;
```

Parameters

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eBufferMode

MC_BUFFERED_MODE_ENUM defines the buffering behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_HALT_OUT Structure

```
typedef struct{
	unsigned int uiHndl;
	unsigned short usStatus;
	short sErrorID;
}MMC_HALT_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-7 describes the function block for MMC_Halt as applied within the IEC 61131 programming.

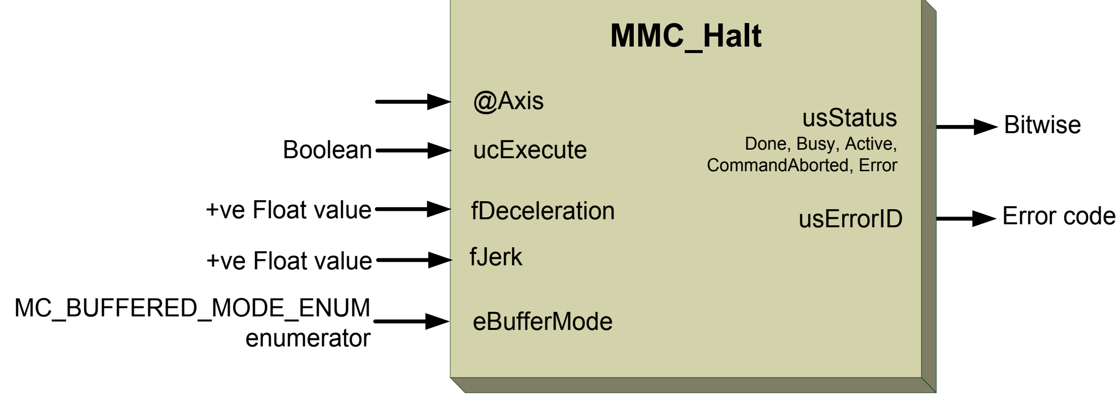


Figure 5-7: MMC_Halt function block

5.8.1.2 Function Block Code Example

```
int rc;
MMC_HALT_IN    stHalt_in;
MMC_HALT_OUT   stHalt_out;
//
// Inserting the structure parameters:
stHalt_in.fDeceleration     = 100000.0;      // Value of the acceleration
stHalt_in.fJerk             = 20000.0;       // Value of the Jerk
stHalt_in.eBufferMode       = MC_ABORTING_MODE;  // MC_BUFFERED_MODE_ENUM Defines the behavior
of the axis
stHalt_in.ucExecute         = 1;
//
rc = MMC_HaltCmd (hConn, iAxisRef, &stHalt_in, &stHalt_out);
if (rc != 0)
{
    HandleError();
}
```

5.8.1.3 Implementation Example

The example below shows the behavior of MMC_Halt in combination with MMC_MoveVelocity.

1. A rotating axis is ramped down with MMC_Halt.
2. Another motion command overrides the MMC_Halt command. MMC_Halt allows this, in contrast to MMC_Stop. The axis can accelerate again without reaching standstill.

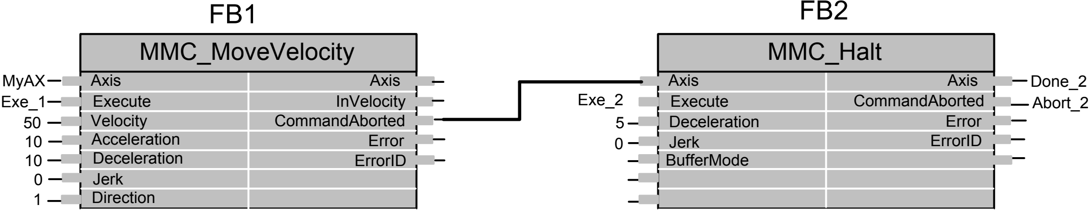


Figure 5-8: Combination of two blocks for MMC_Halt – Example

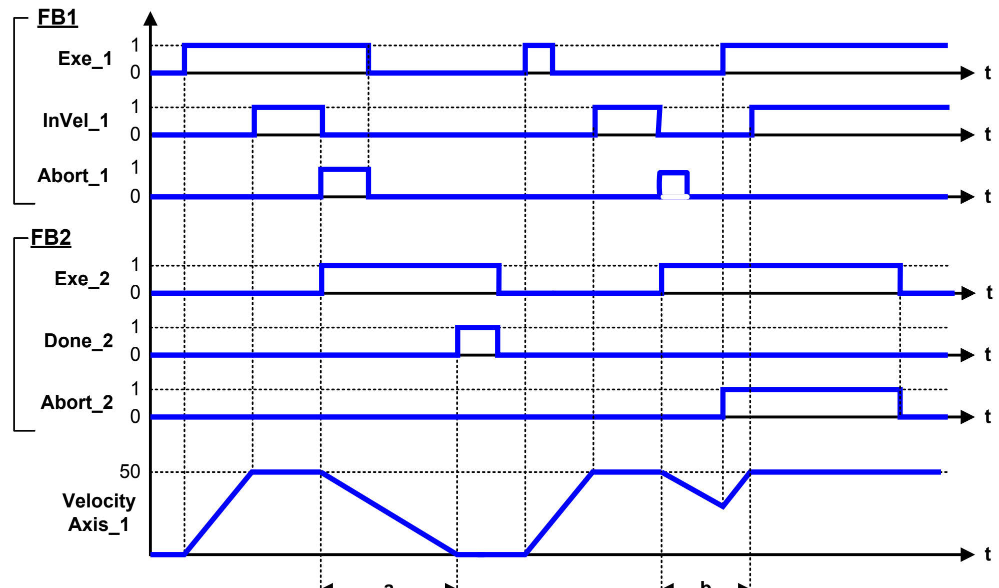


Figure 5-9: Timing diagram of two blocks for MMC_Halt – Example

5.8.2 MMC_Home

Commands the axis to perform the Search Home sequence.

```
MMC_LIB_API int MMC_HomeCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_HOME_IN* pInParam,
OUT MMC_HOME_OUT* pOutParam
);
```

Motion Mode NC - Not Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_HOME_IN** input structure that receives the Home command.

pOutParam

Points to the **MMC_HOME_OUT** output structure receiving information as a result of calling the HOME function.

Remarks

The details of this sequence are manufacturer dependent and can be set by the axis' parameters. The Position input is used to set the absolute position when the reference signal is detected. This function block's movement ends at StandStill.

MMC_Home is a generic function block, which performs a system specified homing procedure. This can also be constructed by the Homing function blocks described in section **Homing Functions**.

Issuing MMC_Home in any other state other than StandStill will go to ErrorStop, even if MMC_Home is issued from the state Homing itself.

Scope

MMC_Home will only operate from Standstill. It will not operate from any form of Motion, Stopping, Homing, ErrorStop, or Disabled. Refer to the State diagram in **Figure** 5-1

MMC_HOME_IN Structure

```
typedef struct{
double dbPosition;
float fAcceleration;
float fVelocity;
float fDistanceLimit;
float fTorqueLimit;
MMC_HOME_MODE_ENUM eHomingMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
MC_HOME_DIRECTION_ENUM eDirection;
MC_SWITCH_MODE_ENUM eSwitchMode;
unsigned int uiTimeLimit;
unsigned char ucExecute;
}MMC_HOME_IN;
```

Parameters

dbPosition

Target position for the motion when conditions are met. Any double -ve or +ve values in technical unit [u].

fVelocity

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s.

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2.

fDistanceLimit

Limit of the drive distance. If StepAbsSwitch condition is not met within a DistanceLimit travel, an error is issued. 0 means no distance limit. fDistanceLimit is used for DS402 homing methods -1, and -2 only (Home On Block for Gold Line servo drives only). This is sent to the drive in object 0x2020 and halted in case the distance is crossed. Any float +ve or –ve value in technical unit [u].

fTorqueLimit

Limit of the torque force of the drive. 0 means no torque limit. Any positive float value in Torque units

eHomingMode

MC_HomingMode enumerator type can have 1-of-5 values as described in section **Homing Functions** below:

MC_ABS_SWITCH

MC_LIMIT_SWITCH

MC_REF_PULSE

MC_DIRECT

MC_BLOCK

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Homing can only be performed from Standstill. The Aborting mode is not supported. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

MC_HOME_DIRECTION_ENUM
eDirection

Specifies the direction of the motion if any. MC_Direction enumerator type can have 1-of-4 values depending on the MC_HomingMode parameter:

MC_POSITIVE	For MC_ABS_SWITCH, MC_BLOCK, and MC_REF_PULSE, starts in positive direction always. For MC_LIMIT_SWITCH starts in Positive direction searching positive LimitSwitch. The direction is automatically reversed from LimitSwitch initial state.
MC_NEGATIVE	For MC_ABS_SWITCH, MC_BLOCK, and MC_REF_PULSE, starts in negative direction always. For MC_LIMIT_SWITCH starts in Negative direction searching negative LimitSwitch. The direction is automatically reversed from LimitSwitch initial state.
MC_SWITCH_POSITIVE	For MC_ABS_SWITCH depends on Switch status at Execute edge. If Switch is Off, direction is positive, if On it is negative.
MC_SWITCH_NEGATIVE	For MC_ABS_SWITCH depends on Switch status at Execute edge. If Switch is On, direction is positive, if Off it is negative.

eSwitchMode

Sensor condition to finalize StepAbsSwitch in any Switch mode. MC_SwitchMode enumerator type can have 1-of-6 values:

MC_ON	When sensor is ON
MC_OFF	When sensor is OFF
MC_EDGE_ON	Rising Edge ON when Off to On transition in sensor
MC_EDGE_OFF	Rising Edge OFF when On to Off transition in sensor
MC_EDGE_SWITCH_POSITIVE	Edge depends on motion direction
MC_EDGE_SWITCH_NEGATIVE	As previous parameter but opposite

uiTimeLimit

Limit of the time for the drive to reach Home. If StepAbsSwitch condition is not met in the *TimeLimit*, error is issued. Any numerical value in *milli-seconds*.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_HOME_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_HOME_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-10 describe the function blocks for MMC_Home as applied within the IEC 61131 programming.

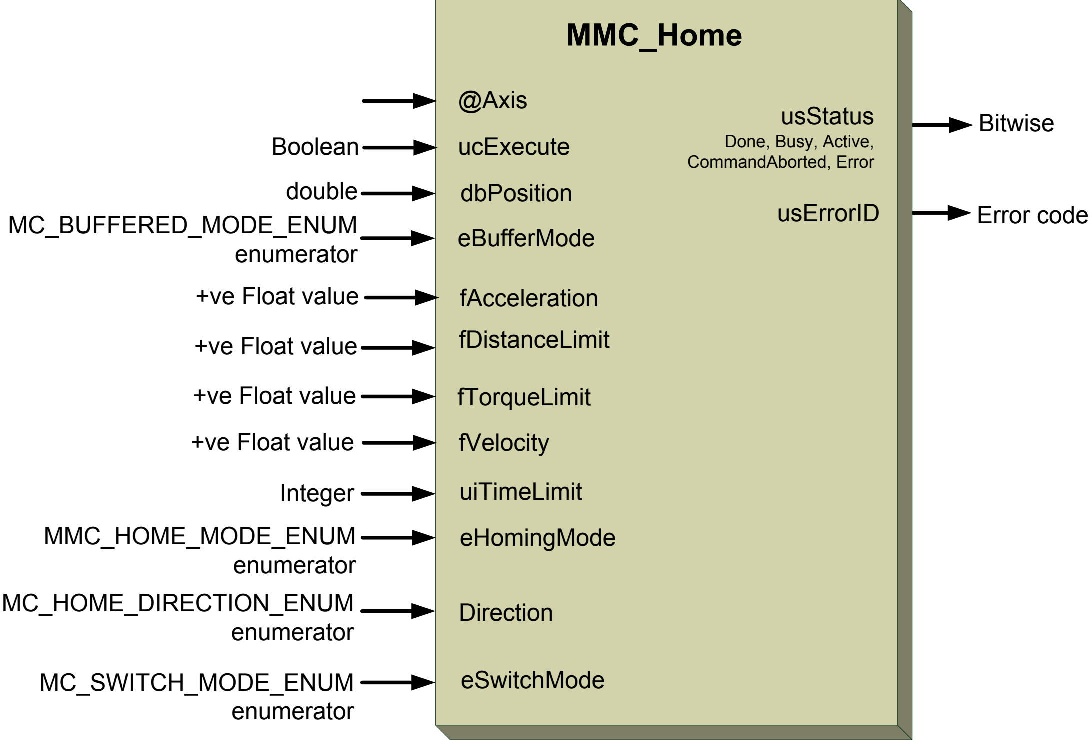


Figure 5-10: MMC_Home function blocks

5.8.2.2 Homing Functions

The homing function supports the MC_HomingMode, as an ENUM datatype, which has the following modes:

HomeAbsSwitch	Absolute Switch homing plus limit switches
HomeLimitSwitch	Homing against limit switches
HomeBlock	Homing against hardware parts blocking movement
HomeRefPulse	Homing using encoder reference pulse Zero Mark
HomeDirect	Static Homing, forcing a position from a user reference

These modes act in the state diagram within the function block MC_Homing, are issued before an axis becomes operational, and use the state Homing during the homing command. After finalizing the homing procedure, the axis moves to the state StandStill. These modes are called active homing.

In order to describe and detail your own homing procedure, and even create user specific and customized homing procedures, function blocks are defined. Using one or more connected function blocks, you can create complex homing sequences.

The following defined function blocks match the homing procedures. They do not change the state of the axis after completion, and therefore remain in the Homing mode. However, they can easily be coupled to another homing function block. To exit the homing state, a dedicated function block has been defined: MC_FinishHoming.

Note: **In order to operate the various Homing Modes in the drive, the Maestro will call the relevant DS-402 Homing operating mode within the drive.**

Homing Functions*	
Function Name	**Description**
MC_AbsSwitch *	Absolute Switch homing plus limit switches
MC_LimitSwitch *	Homing against limit switches
MC_Block *	Homing against hardware parts blocking movement
The following function blocks lead to a final position, automatically clearing the Homing State:	
MC_RefPulse *	Homing using encoder reference Pulse Zero Mark
MC_Direct *	Static Homing forcing a position from a user reference
In addition, homing functions are necessary while the machine is operating, i.e. not in the state Homing. This Homing-on-the-fly is called passive homing. They have no effect on the State Diagram, and similar to the administrative function blocks, can be called in any movement state. They consist of:	
MC_StepReferenceFlyingSwitch *	
MC_StepReferenceFlyingRefPulse*	
MC_AbortPassiveHoming	

5.8.2.2.1 Homing Procedures

A homing procedure couples a position to a specific axis. Homing is dependent on the encoders (absolute versus relative), and system used (linear versus circular). Absolute encoders do not need a movement during the homing procedure, since the exact positions can be directly transferred to the system.

For other encoder types, a movement is necessary, since there is no knowledge of the exact position within the system. This movement is at low speed in some direction until a certain measuring point is reached. Such a measuring point can be scanned from both directions to increase the precision.

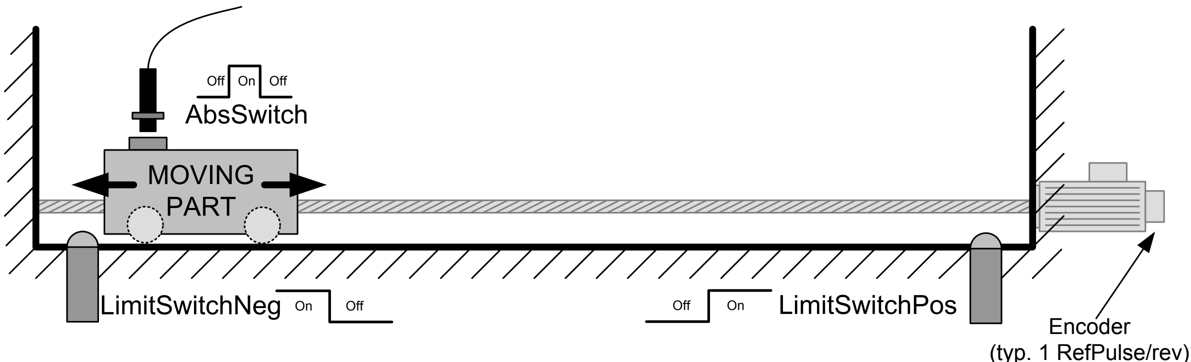


Figure 5-11: Three step sequence

The above example (Figure 5-11) describes a simple sequence consisting of three steps:

1. Search for Signal
2. Search for another Signal
3. Move axis to a pre-defined position

5.8.2.2.2 HomeAbsSwitch

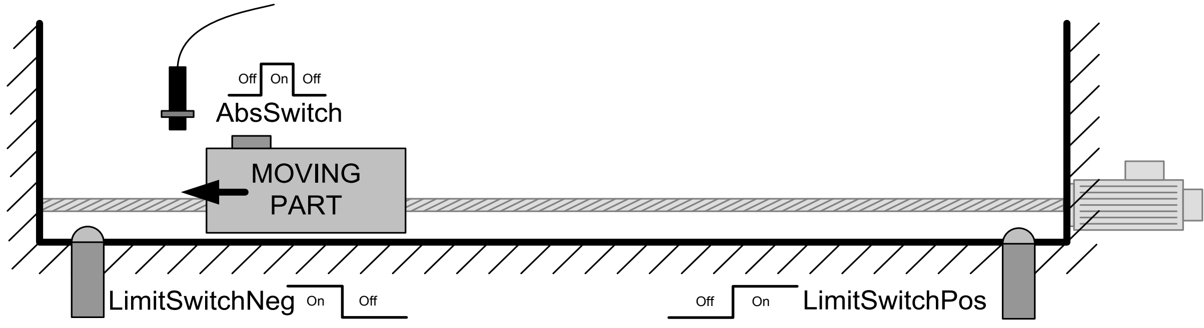


Figure 5-12: Homing by search for an external switch, which is positioned absolute

This homing procedure (Figure 5-12) uses the MC_HomingMode function MC_ABS_SWITCH, and performs a homing function by searching for an absolute positioned external physical switch. It is called Absolute Switch because it might be active only in a specific area of the Moving Part limits, and be Off in the areas surrounding this switch area. Notice that an Absolute Switch has two Off (or On) areas, as shown in the above example. The real meaning of On and Off will depend on the switch logic and controller input logic configurations.

Additional limit switches can be applicable for the security of the system e.g. if homing started in a direction that the absolute reference cannot be found.

The physical layout has the risk that homing is started in the wrong direction (moving away from the switch). To support and correct such a situation, it implements a special behavior when Limit Switches are found (or the *AbsSwitch* itself is On at Execute):

- Axis State is set to Homing (if not already in that state).
- The homing is commanded in the most likely direction where the sensor is to be found.
- The velocity is defined by the input.
- The torque is limited.
- Both Time and Distance Limits cause an error if exceeded.
- If any LimitSwitch is found during Homing (any of them), then a special process is started in the opposite direction, the AbsSwitch is searched to switch Off (or On depending in *SwitchMode* setting). The Edge (passed by), and homing process is restarted in the original direction and with the same conditions. This ensures that the end conditions are always same.
- If the SwitchMode is either MC_SwitchNegative or MC_SwitchPositive, then the special process is also started in opposite direction depending from the switch state at Execute.
- If the *SetPosition* input is disconnected, the function block does not modify actual position. However, if the *SetPosition* input is connected, then this function block modifies the actual position to the *SetPosition* value when the homing condition is met.

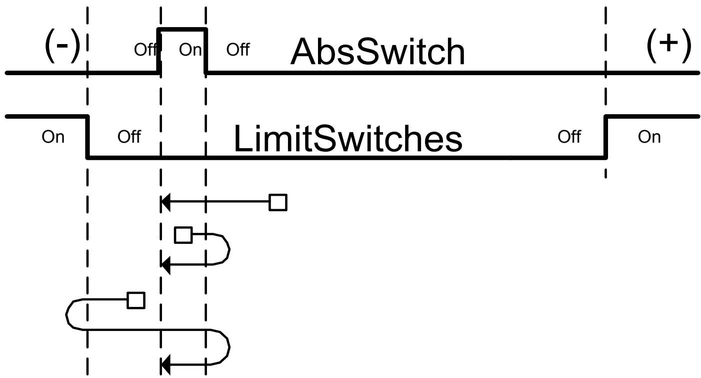


Figure 5-13: HomeAbsSwitch example

The procedure itself requires limited intelligence. In the example above (Figure 5-13), three different starting positions (indicated by the small squares which are starting points) lead to three completely different motion sequences, but with an identical result. This behavior is possible with just one function block as shown below in Figure 5-14.

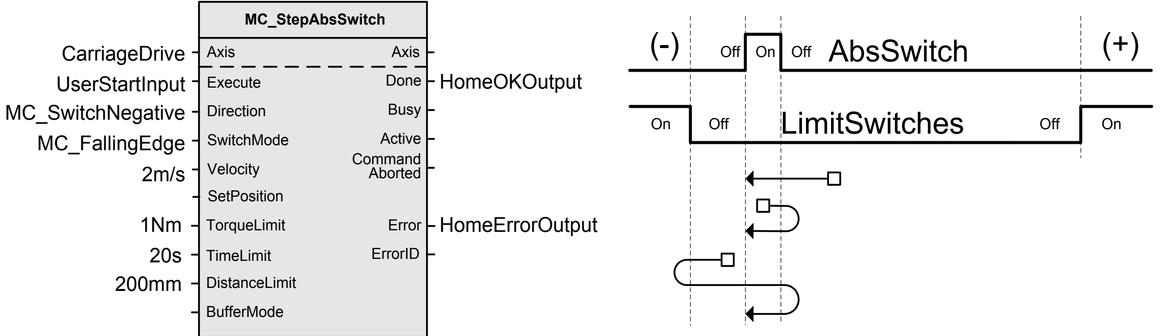


Figure 5-14: Example of a homing step against falling edge of sensor in negative direction

Figure 5-14 describes a homing step against a sensor falling edge in the negative direction. The actual position and axis state are not modified.

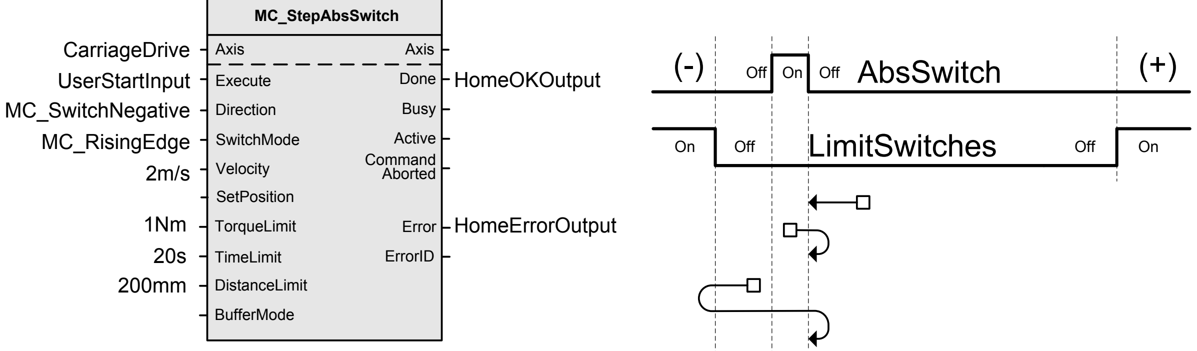


Figure 5-15: Example of a homing step against rising edge of sensor in negative direction

Figure 5-15 describes a homing step against sensor rising edge in the negative direction. The actual position and axis state are not modified.

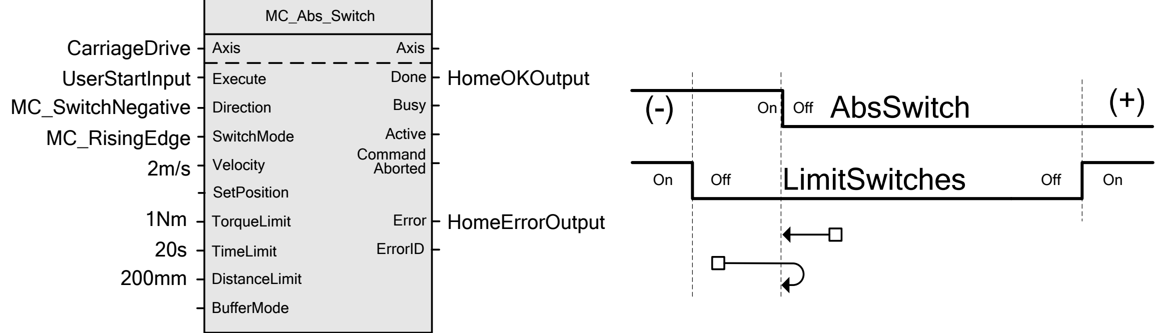


Figure 5-16: Example of an overlapping switch configuration

Figure 5-16 describes an overlapping switch configuration, which behaves similarly to operating via the limit switches. This example also does not modify the actual position.

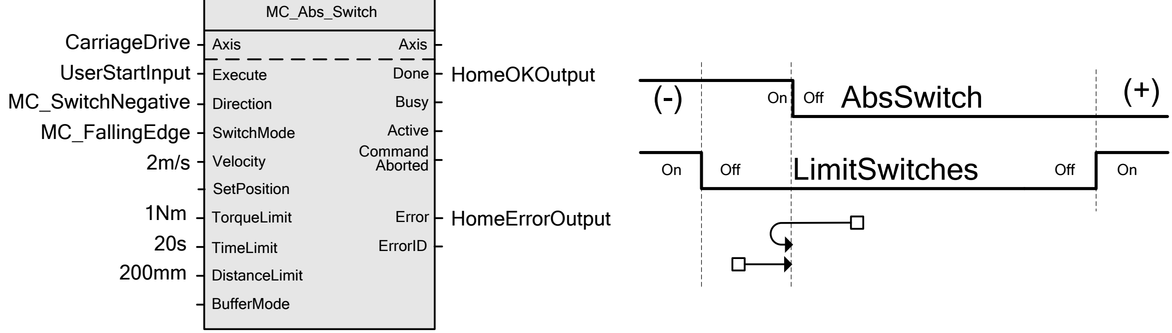


Figure 5-17: Additional example of a homing step against falling edge in the same negative direction

Figure 5-17 describes a homing step against the falling edge this time on the same negative direction. However, the actual position at the edge detection is modified and moved away from the homing state of the axis due to of data in SetPosition.

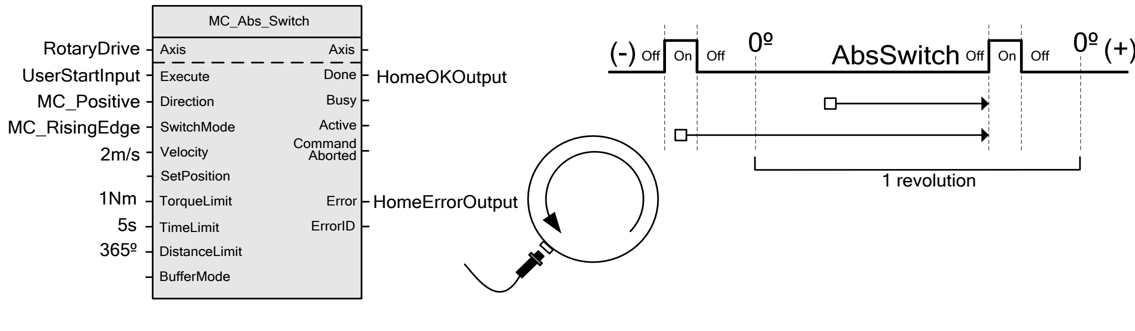


Figure 5-18: Example of Homing with MC_Positive direction

If the input Direction is set to a fixed direction (MC_Positive (**Figure** 5-18) or MC_Negative) then the initial switch state is ignored. This is used for example in the rotary axis where only one sense of rotation is allowed. In this situation, the axis actual position and state are not modified.

5.8.2.2.3 HomeLimitSwitch

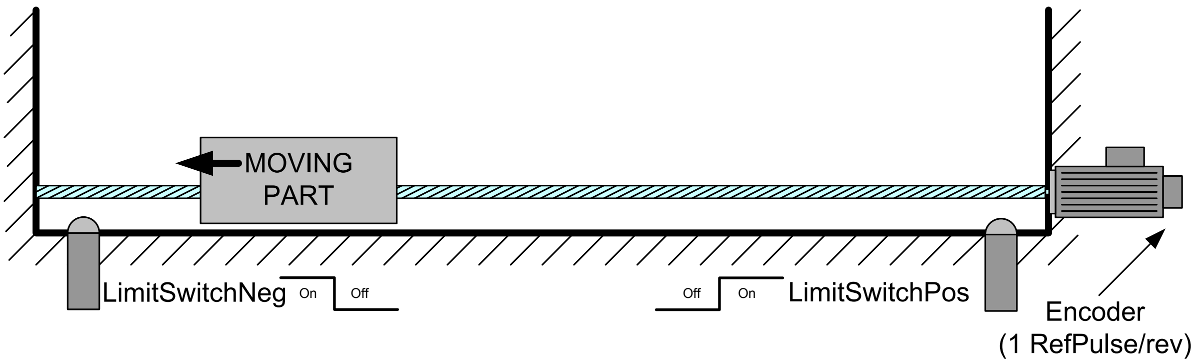


Figure 5-19: Homing by search for sensor using Limit Switches

This homing procedure uses the MC_HomingMode function MC_LIMIT_SWITCH, and performs a homing function searching for sensor using only *LimitSwitches*. A LimitSwitch has one Off (or On) area. In this procedure the following occurs:

- The axis state changes to Homing (if not already in this state).
- Home is commanded by the user in the desired homing direction at the selected Velocity.
- If a LimitSwitch is found On, on rising Execute, then the process is started in the opposite direction as specified. The search is for the LimitSwitch Off (or On, depending on LimitSwitchMode setting) Edge (released), and the process is restarted in the original direction. This ensures that the end conditions are always the same.
- The torque is limited.
- The Time and Distance Limits cause error if exceeded.
- If the SetPosition input is disconnected, this function block does not modify the actual position. However, if SetPosition input is connected, the function block modifies the actual position to the SetPosition Value when homing condition is met.

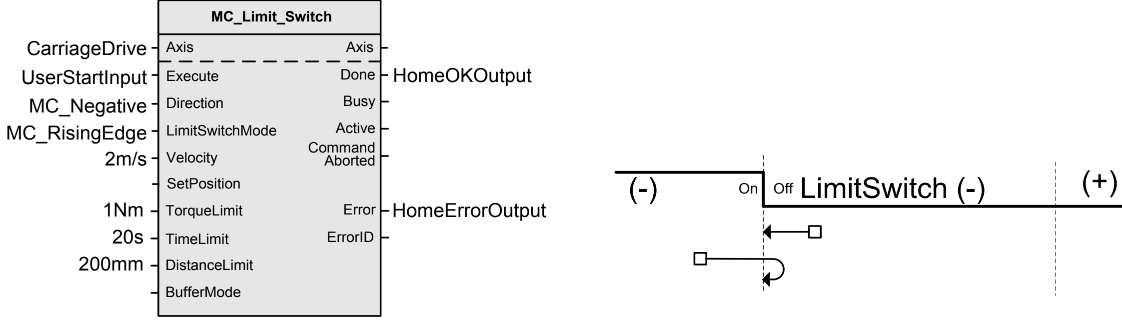


Figure 5-20: HomeLimitSwitch example

Figure 5-20 describes a situation where the result in execution originates from two different starting points (squares). The actual Position is not modified and the axis remains in homing state.

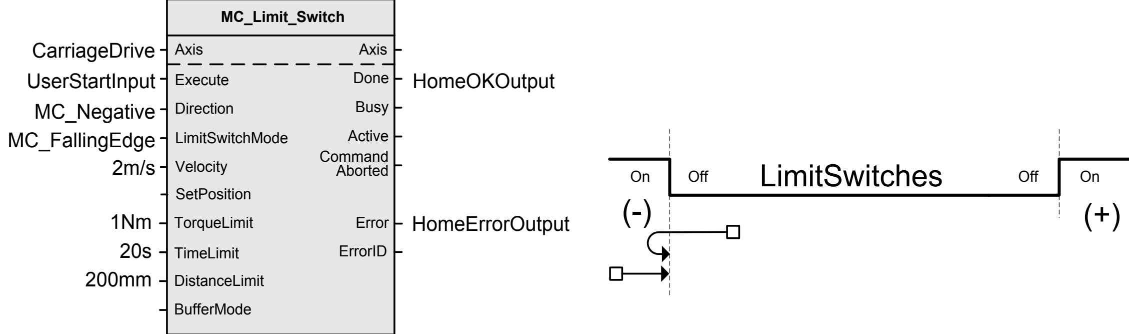


Figure 5-21: HomeLimitSwitch example with a different edge detection

Figure 5-21 describes the same conditions as in Figure 5-20, but enforcing different edge detection (MC_FallingEdge). The Actual Position is not modified.

5.8.2.2.4 HomeBlock

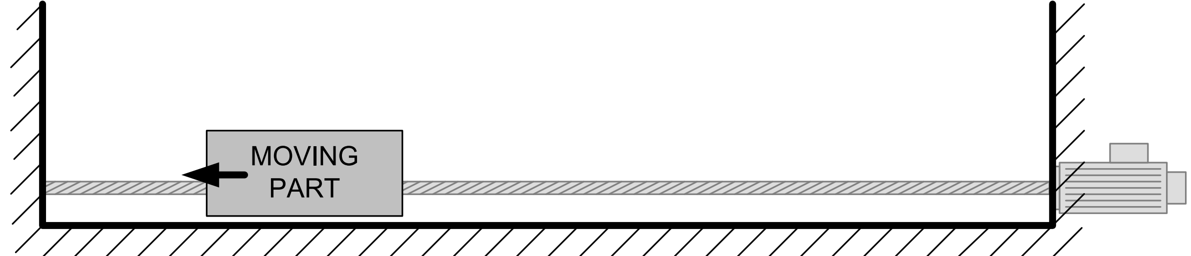


Figure 5-22: Homing against a physical object

This homing procedure uses the MC_HomingMode function MC_BLOCK, and performs a homing against a physical object, mechanically blocking the movement. In this mode, there is no limit switch or Reference Pulse. Adequate torque limits are required to prevent damaging mechanics during the homing process. The Block condition is that the torque limit is reached and the actual velocity falls below the value of the fVelocity input for at least the uiTimeLimit.

Note: **This homing procedure only applies to Gold Line servo drives.**

The following conditions also apply:

- Axis state changes to Homing (if not already in homing).
- The homing procedure is commanded by the user in the desired homing direction at the selected Velocity.

- The Torque is limited.
- Time and Distance Limits can cause an error, if exceeded.
- Process is finished when Torque is in limit condition and real velocity is below 5% of selected velocity.
- If the SetPosition input is disconnected, this function block does not modify the actual position. However, if the SetPosition input is connected, then this function block modifies the actual position to the SetPosition Value when the condition is met.

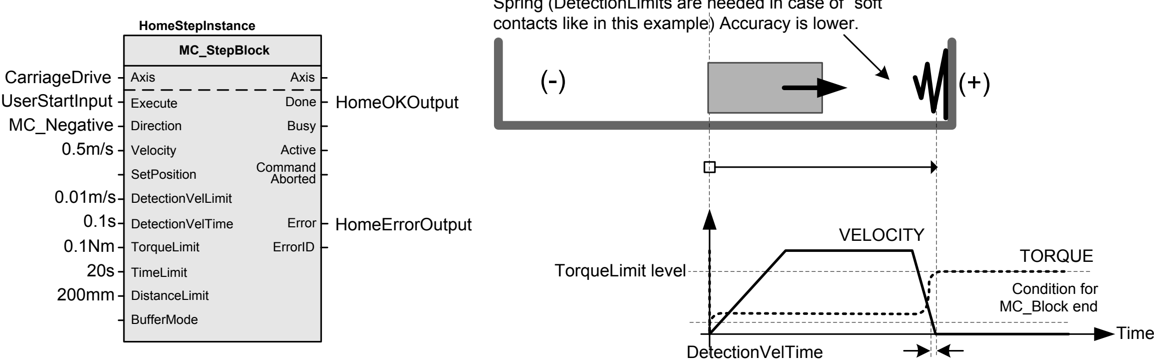


Figure 5-23: HomeBlock example

5.8.2.2.5 HomeRefPulse

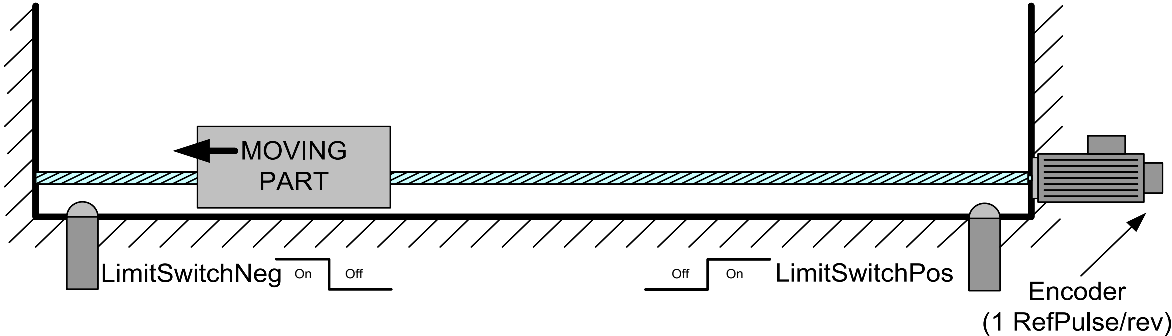


Figure 5-24: Homing by search for a Zero pulse in the encoder

This homing procedure uses the MC_HomingMode function MC_REF_PULSE, and performs homing by searching for Zero pulse (also called Marker or reference pulse) in the encoder. It starts the homing step procedure at rising edge and searches for the reference pulse, which appears once per encoder revolution. The advantage in using a Reference Pulse for homing is the high accuracy and precision that can be achieved compared to traditional optical, mechanical, or magnetic sensors.

The following conditions are applicable:

- The axis state changes to Homing (if not already in that mode).
- The homing is commanded by the user in the desired homing direction at the programmed velocity.
- At the first occurrence of the reference pulse, the procedure is completed.
- Torque is limited. Time and Distance Limits can cause an error, if exceeded.

- If the SetPosition input is not connected, the function block does not modify the actual position. However, if the SetPosition input is connected, then the function block modifies the actual position to the SetPosition value when the condition is met.

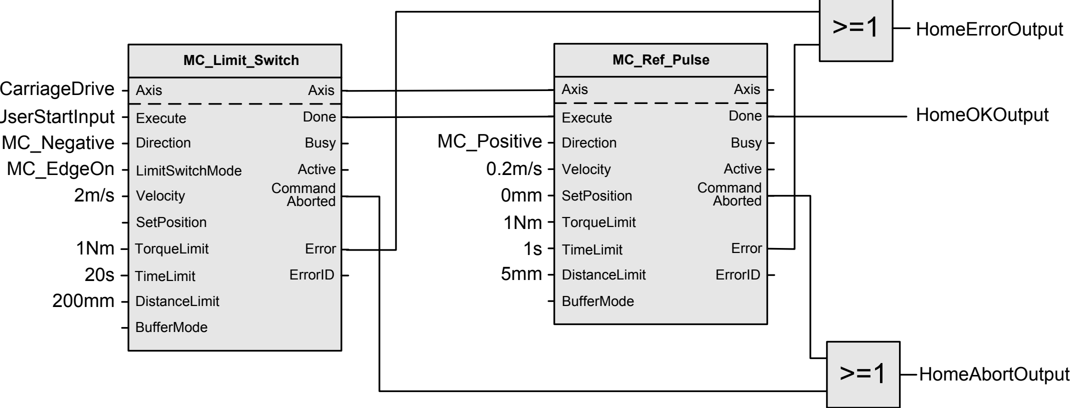


Figure 5-25: HomeRefPulse example using separate function blocks

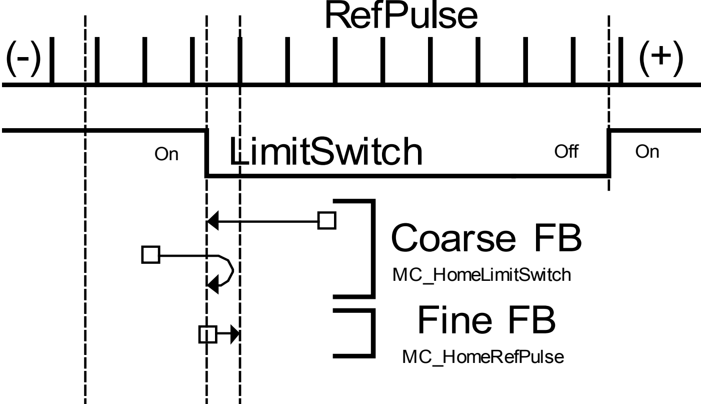


Figure 5-26: HomeRefPulse example

To perform Homing using this function it is common to perform a rough or coarse approach against a mechanical sensor at a higher velocity, and after detecting a reference pulse continue at a lower velocity. This is a traditional two-step homing using Coarse by external switch in reverse and Fine by reference pulse in forward as shown in Figure 5-26. To simplify, both functions could be grouped together in a single function block. The advantage of having separate function blocks (Figure 5-25) is that any combination is possible (MC_StepBlock and after MC_RefPulse, etc.), stating different velocity and conditions for each step, providing a higher flexibility without increasing the complexity of the homing function block.

5.8.2.2.6 HomeRefPulseSet

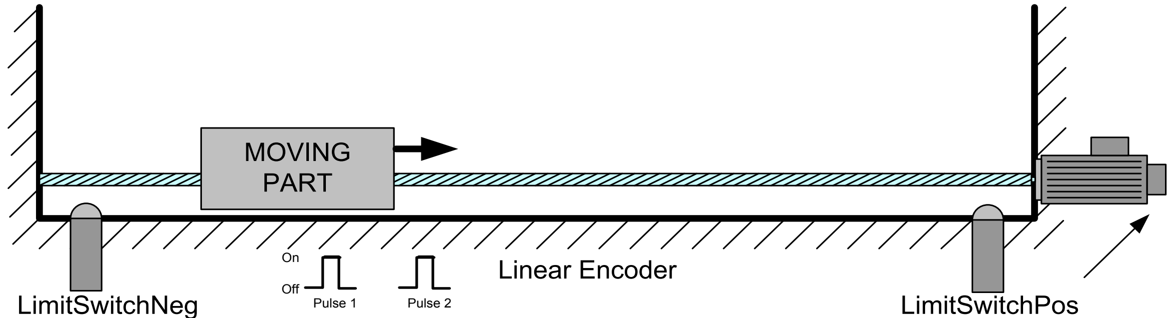


Figure 5-27: HomeRefPulseSet by searching for a set of reference pulses

This homing procedure performs homing by searching for a set of reference pulses (also called marker or zero pulse) from the encoder. The reference pulses appear repeatedly on the distance coded linear encoder length. The advantage in using distance coded reference pulses for homing is the reduced need for traveling distance and reduced time.

A possible scenario:

1. Start in a preferred direction with low speed.
2. Find first ref pulse. If block or end switch are found, instead stop and retry the full procedure in inverse direction.
3. Find second ref pulse, if block or end switch are found instead stop and retry the full procedure in inverse direction.
4. Calculate absolute position.
5. If the procedure still finds block or end switch conditions when doing the retry, abort the process and signal a fault situation.

5.8.2.2.7 HomeDistanceCoded

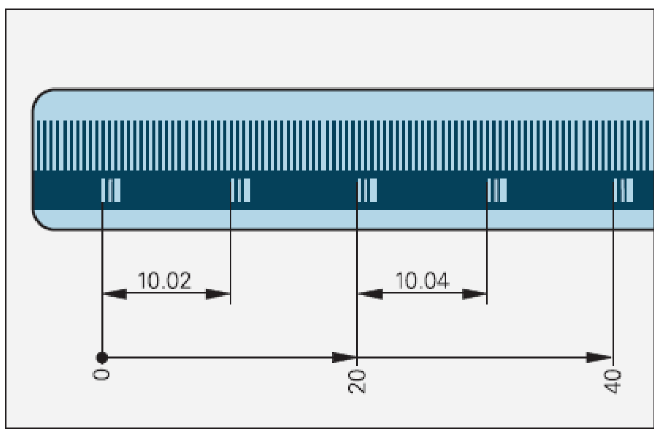


Figure 5-28: Schematic representation of an incremental graduation with distance-coded reference marks

With the incremental measuring method, the graduation consists of a periodic grating structure. The position information is obtained by counting the individual increments (measuring steps) from some point of origin. Since an absolute reference is required to ascertain positions, the scales or scale tapes are provided with an additional track that bears a reference mark. The absolute position on the scale, established by the reference

mark, is gated with exactly one signal period. The reference mark must therefore be scanned to establish an absolute reference or to find the last selected datum.

In some cases, this may necessitate machine movement over large lengths of the measuring range. To simplify such Reference Runs, many encoders feature distance-coded reference marks—multiple reference marks that are individually spaced according to a mathematical algorithm. The subsequent electronics find the absolute reference after traversing two successive reference marks—only a few millimeters traverse (**Figure** 5-28).

With distance-coded reference marks, the absolute reference is calculated by counting the signal periods between two reference marks. After calculation, it sets the actual position. If there are no coded marks found within a certain time or distance traveled, it generates an error.

5.8.2.2.8 HomeDirect

This homing procedure uses the MC_HomingMode function MC_DIRECT, and performs a static homing by directly forcing an actual position to the SetPosition input value. No physical motion is performed in this mode. This is equivalent to a MC_SetPosition action, but clears the Homing State.

5.8.2.2.9 HomeAbsolute

This homing procedure performs a static homing function by setting the actual position to the position of an absolute encoder.

No physical motion is performed in this mode. It is equivalent to issuing MC_SetPosition with SetPosition obtained from the absolute encoder reading, but after finalizing the Homing State.

5.8.2.2.10 MC_FinishHoming

This function block transfers an axis from the state Homing to the state StandStill and finalizes the homing procedure. In addition, it can perform a relative movement to move the axis in the allowed area (working area), in a situation whereby the homing steps left the axis on the wrong side of the switch.

5.8.2.3 Function Block Code Example

```
int rc;
MMC_HOME_IN   stHome_in;
MMC_HOME_OUT  stHome_out;
//
// Inserting the structure parameters:
stHome_in.fAcceleration  = 100000.0;    // Value of the acceleration
stHome_in.fDistanceLimit = 100000.0;    // Limit of the drive distance
stHome_in.fTorqueLimit   = 2.0;    // Limit of the torque force of the drive
stHome_in.eHomingMode    = MC_DIRECT;   // MC_HomingMode enumerator type
stHome_in.eDirection     = MC_POSITIVE_DIRECTION;// MC_Direction Enumerator type
stHome_in.eBufferMode    = MC_BUFFERED_MODE;     // MC_BUFFERED_MODE_ENUM Defines the behavior
of the axis
stHome_in.eSwitchMode    = MC_ON; // Sensor condition to finalize StepAbsSwitch in any Switch
mode
stHome_in.dbPosition     = 100000.0;    // Target position for the motion
stHome_in.fVelocity = 5000.0;     // Velocity in
stHome_in.uiTimeLimit    = 100;                 // Limit of the time for the drive to reach
Home
stHome_in.ucExecute = 1;
//
rc = MMC_HomeCmd (hConn, iAxisRef, &stHome_in, &stHome_out);
if (rc != 0)
{
    HandleError();
}
```

5.8.2.4 Implementation Example

TBD

5.8.3 MMC_HomeDS402

Commands the axis to perform the Search Home sequence and can be set by the axes parameters.

```
MMC_LIB_API int MMC_HomeDS402Cmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_HOMEDS402_IN* pInParam,
OUT MMC_HOME_OUT* pOutParam
);
```

Motion Mode NC - Not Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_HOMEDS402_IN** input structure that receives the Home DS-402 command.

pOutParam

Points to the **MMC_HOME_OUT** output structure receiving information as a result of calling the Home DS-402 function.

Remarks

The Position input is used to set the absolute position when the reference signal is detected. This function block's movement ends at StandStill.

MMC_HomeDS402 is a generic function block, which performs a system specified homing procedure. This can also be constructed by the Homing methods described in section 9.4 in the manual www.elmomc.com/support/manuals/MAN-CAN402IG.pdf.

Scope

MMC_HomeDS402 will only operate from Standstill. It will not operate from any form of Motion, Stopping, Homing, ErrorStop, or Disabled. Refer to the State diagram in **Figure 5-1**.

Issuing MMC_HomeDS402 in any other state other than StandStill will go to ErrorStop, even if MMC_HomeDS402 is issued from the state Homing itself.

MMC_HOMEDS402_IN Structure

```
typedef struct{
double dbPosition;
float fAcceleration;
float fVelocity;
float fDistanceLimit;
float fTorqueLimit;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned int uiHomingMethod;
unsigned int uiTimeLimit;
unsigned char ucExecute;
}MMC_HOMEDS402_IN;
```

Parameters

dbPosition

The position parameter is actually an offset of the position. When the homing procedure has completed, the absolute position set is –Position. Therefore, if the ID position is 500, the position at the end of the homing procedure will be -500. Any -ve or +ve double values in technical unit [u].

fVelocity

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2

fDistanceLimit

Limit of the drive distance. if StepAbsSwitch condition is not met within a DistanceLimit travel, an error is issued. 0 means no distance limit. fDistanceLimit is used for DS402 homing methods -1, and -2 only (Home On Block for Gold Line servo drives only). This is sent to the drive in object 0x2020 and halted in case the distance is crossed. Any +ve or –ve float value in technical unit [u]

fTorqueLimit

Limit of the torque force of the drive. 0 means no torque limit. Any positive float value in Torque units

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Homing can only be performed from Standstill. The Aborting mode is not supported. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

uiHomingMethod

DS-402 standard for homing of drive. Integer of value 1 – 36, with 4 values reserved. Refer to the CiA definition or Drive reference manual.

uiTimeLimit

Limit of the time for the drive to reach Home. If StepAbsSwitch condition is not met in the TimeLimit, error is issued. Any numerical value in milli-seconds.

ucExecute

Start the execution command. TRUE/FALSE Boolean value.

MMC_HOME_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_HOME_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block:

- MC_TimeLimitExceeded
- MC_DistanceLimitExceeded
- MC_TorqueLimitExceeded

Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-29 describe the function blocks for MMC_HomeDS402 as applied within the IEC 61131 programming.

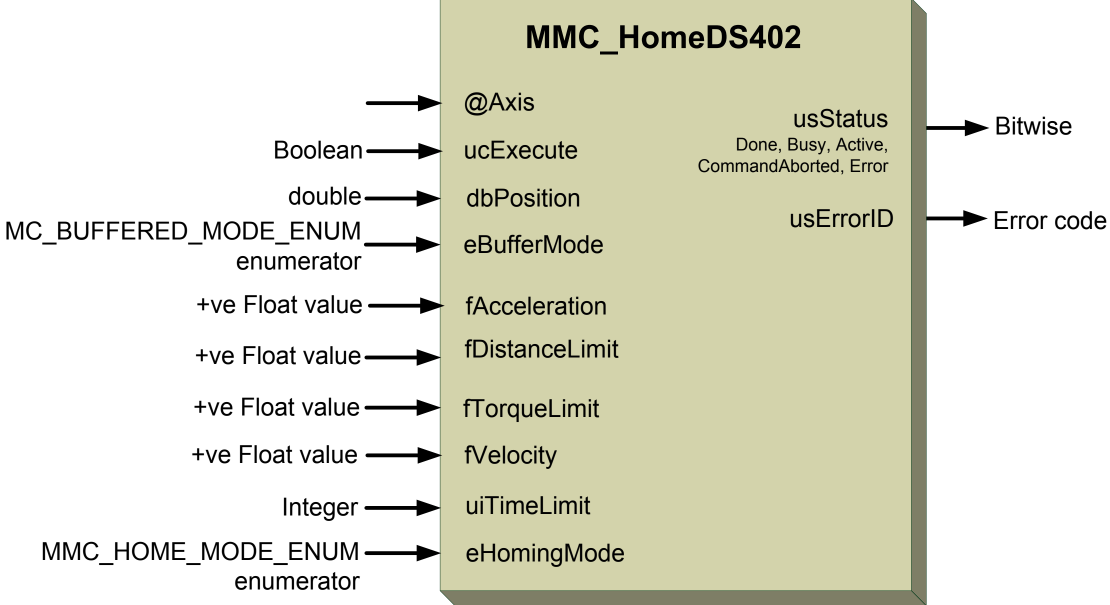


Figure 5-29: MMC_HomeDS402 function blocks

5.8.3.2 Function Block Code Example

```
int rc ;
MMC_HOMEDS402_IN  stHomeDS402_in;
MMC_HOME_OUT      stHome_out;
//
// Inserting the structure parameters:
stHomeDS402_in.fAcceleration  = 100000.0;    // Value of the acceleration
stHomeDS402_in.fDistanceLimit = 100000.0;    // Limit of the drive distance
stHomeDS402_in.fTorqueLimit   = 2.0;     // Limit of the torque force of the drive
stHomeDS402_in.eBufferMode    = MC_ABORTING_MODE; // MC_BUFFERED_MODE_ENUM Defines the
behavior of the axis
stHomeDS402_in.uiHomingMethod = 1;       // DS-402 standard for homing of drive
stHomeDS402_in.dbPosition     = 100000.0;    // Target position for the motion
stHomeDS402_in.fVelocity = 5000.0;       // Velocity in
stHomeDS402_in.uiTimeLimit    = 100000;             // Limit of the time for the drive to
reach Home
stHomeDS402_in.ucExecute = 1;
//
rc = MMC_HomeDS402Cmd (hConn, iAxisRef, &stHomeDS402_in, &stHome_out) ;
if (rc != 0)
{
    HandleError() ;
}
```

5.8.4 MMC_MoveAbsolute

Commands a discreet controlled motion for a single axis to a specified absolute position.

```
MMC_LIB_API int MMC_MoveAbsoluteCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVEABSOLUTE_IN* pInParam,
OUT MMC_MOVEABSOLUTE_OUT* pOutParam
);
```

Motion Mode	NC - All Buffering modes are supported.	Distributed - Only MC_ABORTING_MODE Buffered mode is supported.
Source	GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVEABSOLUTE_IN** input data structure using the MMC_MoveAbsolute function.

pOutParam

Points to the **MMC_MOVEABSOLUTE_OUT** output structure receiving information as a result of calling the MMC_MoveAbsolute function.

Remarks

This action completes with velocity zero if no further action is pending.

Scope

MMC_MoveAbsolute will only operate from Standstill or Continuous Motion. It will not operate from Stopping, Homing, ErrorStop, or Disabled. Refer to the State diagram in **Figure** 5-1.

MMC_MOVEABSOLUTE_IN Structure

```
typedef struct{
double dbPosition;
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_DIRECTION_ENUM eDirection;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
}MMC_MOVEABSOLUTE_IN;
```

Parameters

dbPosition

Target position for the motion when conditions are met. Any -ve or +ve double values in technical unit [u].

fVelocity

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eDirection

Specifies the direction of the motion, if any. The MC_DIRECTION_ENUM enumerator type can have 1-of-4 values:

MC_NONE_DIRECTION = 0

MC_POSITIVE_DIRECTION = 1

MC_SHORTEST_WAY = 2 Not implemented as yet

MC_NEGATIVE_DIRECTION = 3

MC_CURRENT_DIRECTION = 4

The Enum type MC_SHORTEST_WAY is focused to a trajectory, which will go through the shortest route (option not implemented as yet). The decision which direction to move is based on the current position from where the command is issued. In general, make sure to set the parameter to MC_POSITIVE_DIRECTION to be sent.

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered The next function block affects the axis as soon as the previous movement is completed.

BlendingLow The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).

BlendingPrevious Blending with the velocity of function block 1 at the end-position of this block

BlendingNext Blending with the velocity of function block 2 at end-position of function block1

BlendingHigh Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVEABSOLUTE_OUT Structure

typedef struct{

unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;

}MMC_MOVEABSOLUTE_OUT;

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-30 describes the function block for MMC_MoveAbsolute as applied within the IEC 61131 programming.

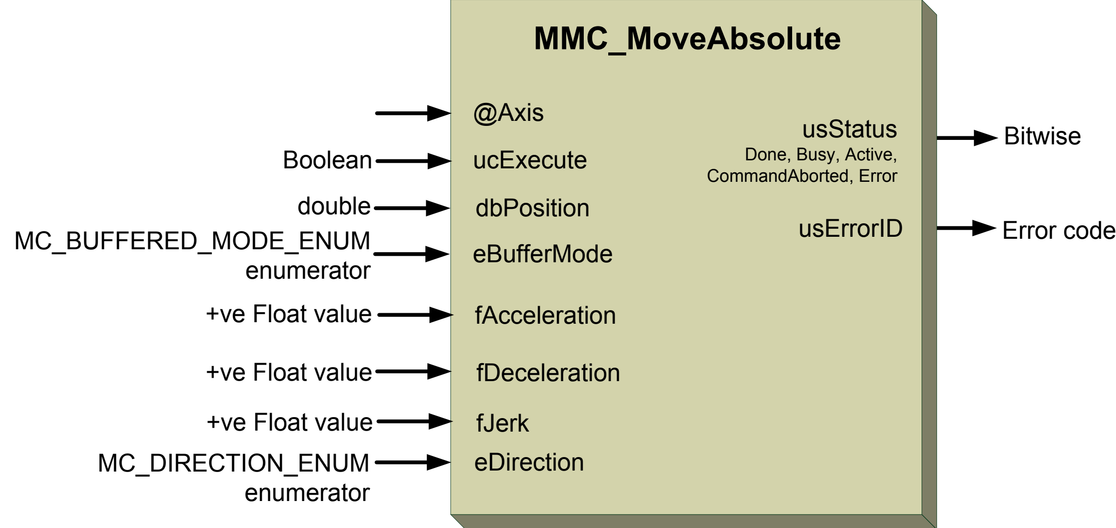


Figure 5-30: MMC_MoveAbsolute function block

5.8.4.2 Function Block Code Example

```
int rc;
MMC_MOVEABSOLUTE_IN    stMove_Abs_in;
MMC_MOVEABSOLUTE_OUT   stMove_Abs_out;
//
// Inserting the structure parameters:
stMove_Abs_in.fAcceleration = 100000.0;     // Value of the acceleration
stMove_Abs_in.fDeceleration = 100000.0;     // Value of the deceleration
stMove_Abs_in.fJerk      = 20000000.0;      // Value of the Jerk
stMove_Abs_in.eDirection     = MC_POSITIVE_DIRECTION; // MC_Direction Enumerator type
stMove_Abs_in.eBufferMode    = MC_BUFFERED_MODE; // MC_BUFFERED_MODE_ENUM Defines the behavior
of the axis
stMove_Abs_in.dbPosition     = 100000.0;    // Target position for the motion
stMove_Abs_in.fVelocity      = 5000.0;      // Velocity in
stMove_Abs_in.ucExecute      = 1;
//
rc = MMC_MoveAbsoluteCmd (hConn, iAxisRef, &stMove_Abs_in, &stMove_Abs_out);
if (rc != 0)
{
    HandleError();
}
```

5.8.4.3 Implementation Example

The following figure shows two examples of the combination of two absolute move function blocks:

1. The left part of timing diagram in **Figure** 5-32 illustrates the case where the second function block is called after the first one. If the first block reaches the commanded position of 6000 (and the velocity is 0) then the output Done causes the Second function block to move to the position 10000.
2. The right part of the timing diagram illustrates the case if the second move function Block starts the execution while the first function block is still executing. In this case the first motion is interrupted and aborted by the Test signal during the constant velocity of the First function block. The second function block moves directly to the position 10000 although the position of 6000 is not yet reached.

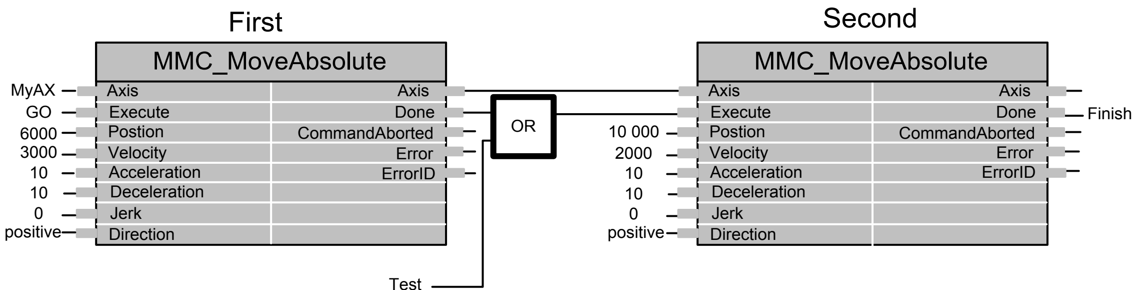


Figure 5-31: Combination of two blocks moved absolute - Example

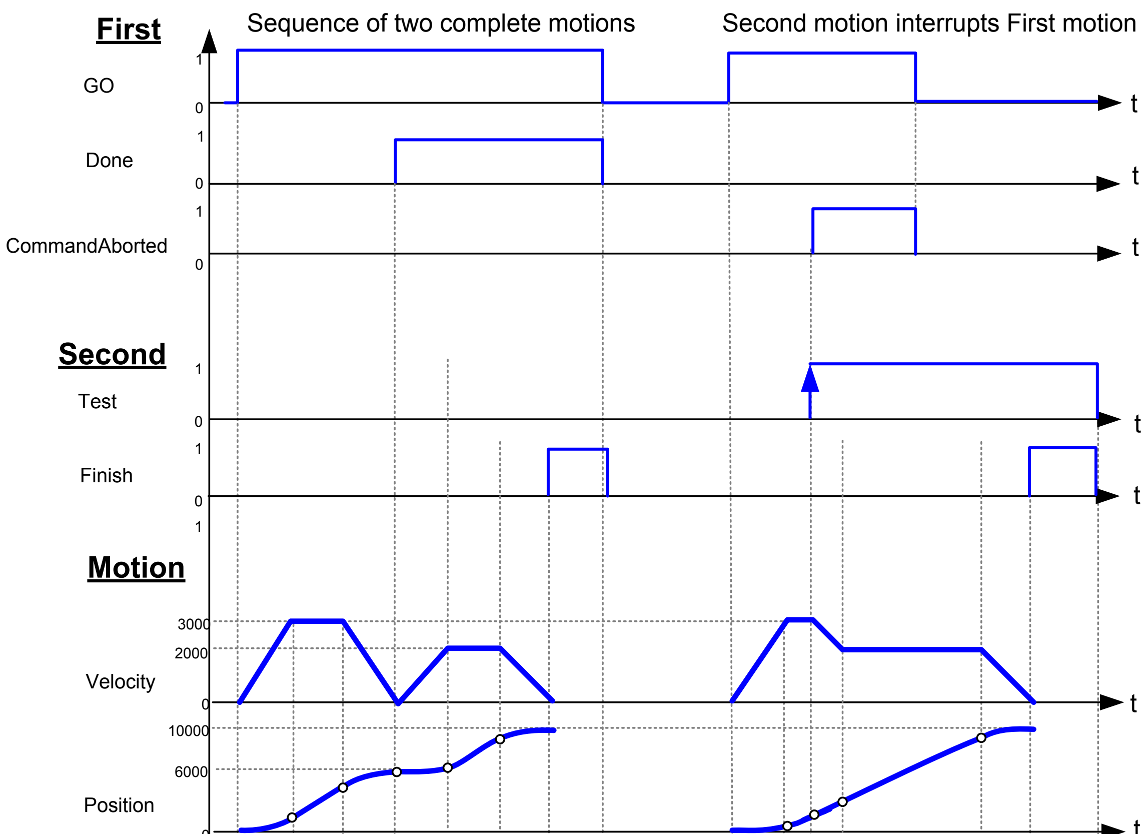


Figure 5-32: Timing diagram for MMC_MoveAbsolute – Example

5.8.5 MMC_MoveAdditive

Commands a controlled motion of a specified relative distance additional to the most recent commanded position in the discrete motion state.

```
MMC_LIB_API int MMC_MoveAdditiveCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVEADDITIVE_IN* pInParam,
OUT MMC_MOVEADDITIVE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVEADDITIVE_IN** input data structure using the MMC_MoveAdditive function.

pOutParam

Points to the **MMC_MOVEADDITIVE_OUT** output structure receiving information as a result of calling the MMC_MoveAdditive function.

Remarks

The most recent commanded position may be the result of a previous MMC_MoveAdditive motion, which was aborted. If the function block is activated in the Continuous Mode, the specified relative distance is added to the actual position at the time of the execution.

Scope

MMC_MoveAdditive will only operate from Standstill, Continuous Motion, or Discrete Motion. It will not operate from Stopping, Homing, ErrorStop, or Disabled. Refer to the State diagram in **Figure** 5-1.

MMC_MOVEADDITIVE_IN Structure

```
typedef struct{
double dbDistance;
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_DIRECTION_ENUM eDirection;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
}MMC_MOVEADDITIVE_IN;
```

Parameters

dbDistance

Relative additive distance for the motion. Any -ve or +ve double values in technical unit [u].

fVelocity

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eDirection

Specifies the direction of the motion, if any. The MC_DIRECTION_ENUM enumerator type can have 1-of-4 values:

MC_NONE_DIRECTION = 0

MC_POSITIVE_DIRECTION = 1

MC_SHORTEST_WAY = 2 Not implemented as yet
MC_NEGATIVE_DIRECTION = 3

MC_CURRENT_DIRECTION = 4

The Enum type MC_SHORTEST_WAY is focused to a trajectory, which will go through the shortest route(option not implemented as yet). The decision which direction to move is based on the current position from where the command is issued. In general, make sure to set the parameter to MC_POSITIVE_DIRECTION to be sent.

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVEADDITIVE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVEADDITIVE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-33 describes the function block for MMC_MoveAdditive as applied within the IEC 61131 programming.

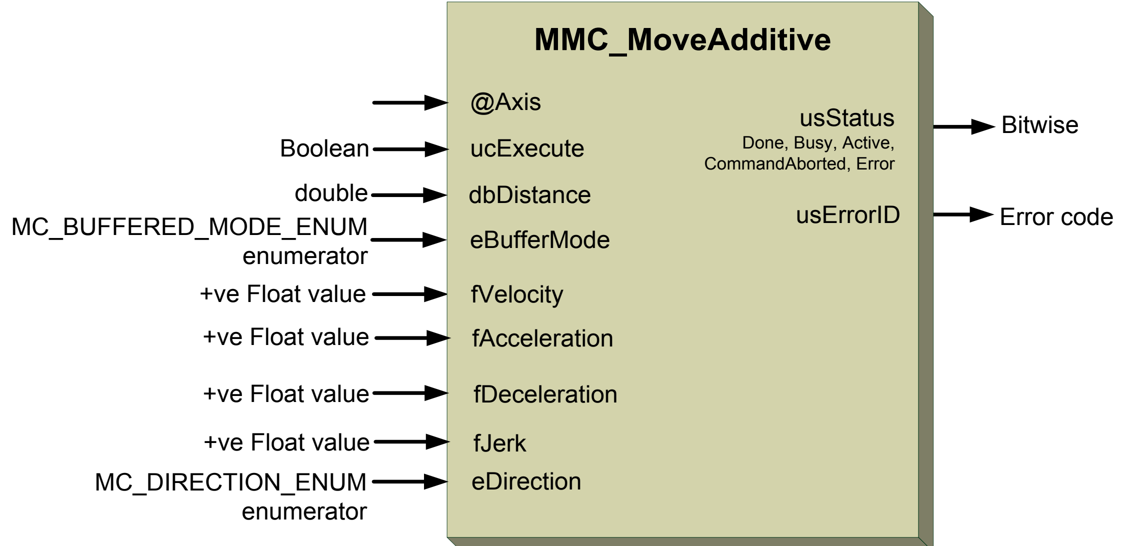


Figure 5-33: MMC_MoveAdditive function block

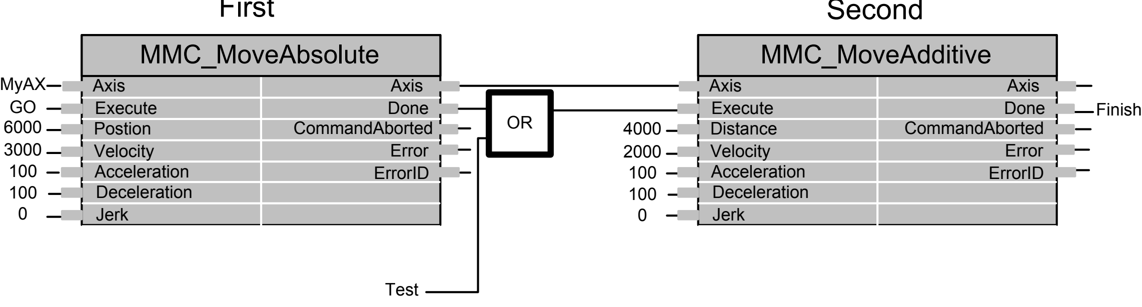
5.8.5.2 Function Block Code Example

```
int rc ;
MMC_MOVEADDITIVE_IN    stMoveadd_in ;
MMC_MOVEADDITIVE_OUT   stMoveadd_out ;
//
// Inserting the structure parameters:
stMoveadd_in.fAcceleration = 100000.0;     // Value of the acceleration
stMoveadd_in.fDeceleration = 100000.0;     // Value of the deceleration
stMoveadd_in.fJerk     = 20000000.0;  // Value of the Jerk
stMoveadd_in.eDirection     = MC_POSITIVE_DIRECTION; // MC_Direction Enumerator type
stMoveadd_in.eBufferMode    = MC_BUFFERED_MODE; // MC_BUFFERED_MODE_ENUM Defines the behavior
of the axis
stMoveadd_in.dbDistance     = 100000.0;     // Relative distance for the motion
stMoveadd_in.fVelocity      = 5000.0;       // Velocity in
stMoveadd_in.ucExecute      = 1;
//
rc = MMC_MoveAdditiveCmd (hConn, iAxisRef, &stMoveadd_in, &stMoveadd_out) ;
if (rc != 0)
{
    HandleError() ;
}
```

5.8.5.3 Implementation Example

The following figure shows the example of the combination of move absolute and additive function blocks.

1. The left part of timing diagram illustrates the case if the Second function block is called after the First one. If First reaches the commanded distance 6000 (and the velocity is 0) then the output Done causes the Second function block to move to the distance 10000.
2. The right part of the timing diagram illustrates the case if the Second move function block starts the execution while the First function block is still executing. In this case the First motion is interrupted and aborted by the Test signal during the constant velocity of the First function block. The Second function block adds on the previous commanded position of 6000 the distance 4000 and moves the axis to the resulting position of 10000.



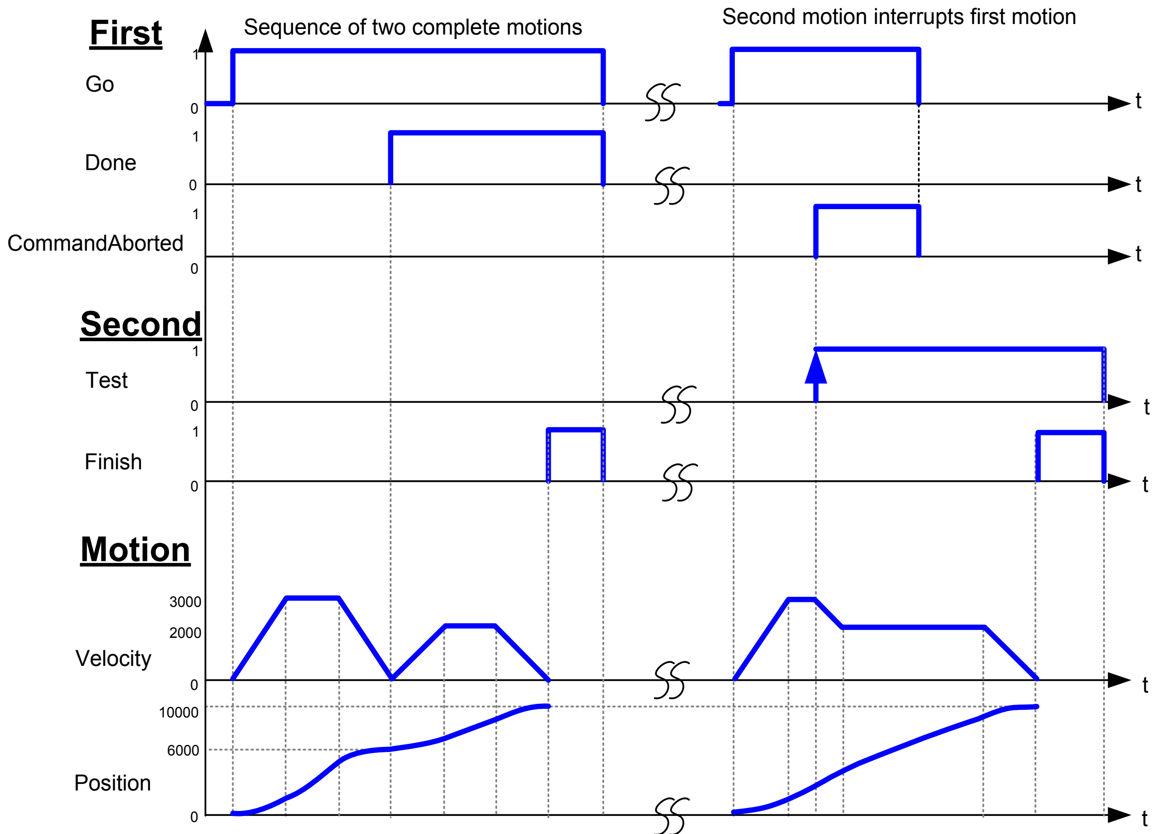


Figure 5-34: Combination of two blocks for MMC_MoveAdditive with Timing diagram – Example

5.8.6 MMC_MoveRelative

Commands a discreet controlled motion of a specified distance relative to the set position at the time of the execution.

```
int MMC_MoveRelativeCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVERELATIVE_IN* pInParam,
OUT MMC_MOVERELATIVE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVERELATIVE_IN** input data structure using the MMC_MoveRelative function.

pOutParam

Points to the **MMC_MOVERELATIVE_OUT** output structure receiving information, as a result of calling the MMC_MoveRelative function.

Remarks

This action completes with velocity zero if no further action are pending.

Scope

MMC_MoveRelative will only operate from Standstill or Continuous Motion. It will not operate from Stopping, Homing, ErrorStop, or Disabled. Refer to the State diagram in **Figure** 5-1.

MMC_MOVERELATIVE_IN Structure

```
typedef struct{
double dbDistance;
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_DIRECTION_ENUM eDirection;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
}MMC_MOVERELATIVE_IN;
```

Parameters

dbDistance

Relative additive distance for the motion. Any -ve or +ve double values in technical unit [u].

fVelocity

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eDirection

Specifies the direction of the motion, if any. The MC_DIRECTION_ENUM enumerator type can have 1-of-4 values:

MC_NONE_DIRECTION = 0

MC_POSITIVE_DIRECTION = 1

MC_SHORTEST_WAY = 2 Not implemented as yet

MC_NEGATIVE_DIRECTION = 3

MC_CURRENT_DIRECTION = 4

The Enum type MC_SHORTEST_WAY is focused to a trajectory, which will go through the shortest route(option not implemented as yet). The decision which direction to move is based on the current position from where the command is issued. In general, make sure to set the parameter to MC_POSITIVE_DIRECTION to be sent.

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting — Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered — The next function block affects the axis as soon as the previous movement is completed.

BlendingLow — The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).

BlendingPrevious — Blending with the velocity of function block 1 at the end-position of this block

BlendingNext — Blending with the velocity of function block 2 at end-position of function block1

BlendingHigh — Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVERELATIVE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVERELATIVE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-35 describes the function block for MMC_MoveRelative as applied within the IEC 61131 programming.

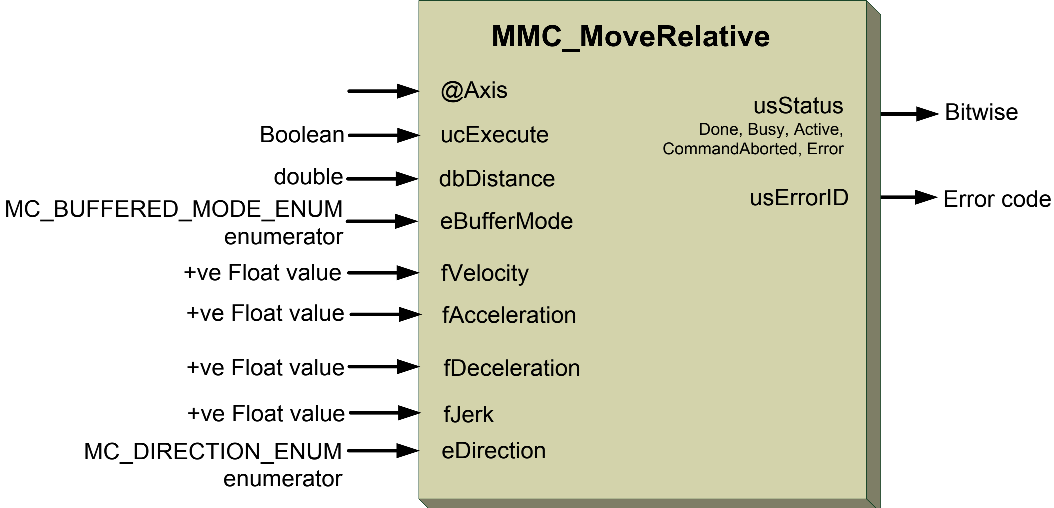


Figure 5-35: MMC_MoveRelative function block

5.8.6.2 Function Block Code Example

```
int rc;
MMC_MOVERELATIVE_IN    stMoveReltv_in;
MMC_MOVERELATIVE_OUT   stMoveReltv_out;
//
// Inserting the structure parameters:
stMoveReltv_in.fAcceleration  = 100000.0;    // Value of the acceleration
stMoveReltv_in.fDeceleration  = 100000.0;    // Value of the deceleration
stMoveReltv_in.fJerk      = 20000000.0;  // Value of the Jerk
stMoveReltv_in.eBufferMode     = MC_BUFFERED_MODE;// MC_BUFFERED_MODE_ENUM Defines the behavior
of the axis
stMoveReltv_in.eDirection     = MC_POSITIVE_DIRECTION; // MC_Direction Enumerator type
stMoveReltv_in.dbDistance     = 100000.0;    // Relative distance for the motion
stMoveReltv_in.fVelocity = 5000.0;      // Velocity in
stMoveReltv_in.ucExecute = 1;
//
rc = MMC_MoveRelativeCmd (hConn, iAxisRef, &stMoveReltv_in, &stMoveReltv_out);
if (rc != 0)
{
    HandleError();
}
```

5.8.6.3 Implementation Example

The following figure shows the example of the combination of two relative move function blocks.

1. The left part of timing diagram illustrates the case if the second function block is called after the First one. If First reaches the commanded distance 6000 (and the velocity is 0) then the output Done causes the Second function block to move to the distance 10000.
2. The right part of the timing diagram illustrates the case if the Second move function blocks starts the execution while the First function block is still executing. In this case the First motion is interrupted and aborted by the Test signal during the constant velocity of the First function block. The Second function block adds on the actual position of 3250 the distance 4000 and moves the axis to the resulting position of 7250.

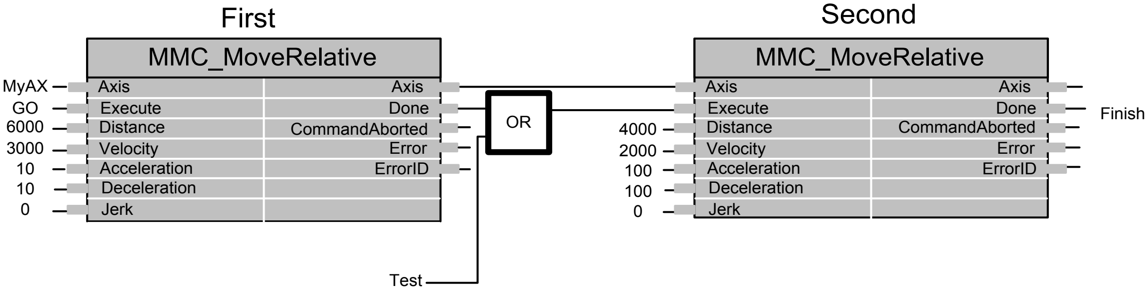


Figure 5-36: Example of two MoveRelative function blocks

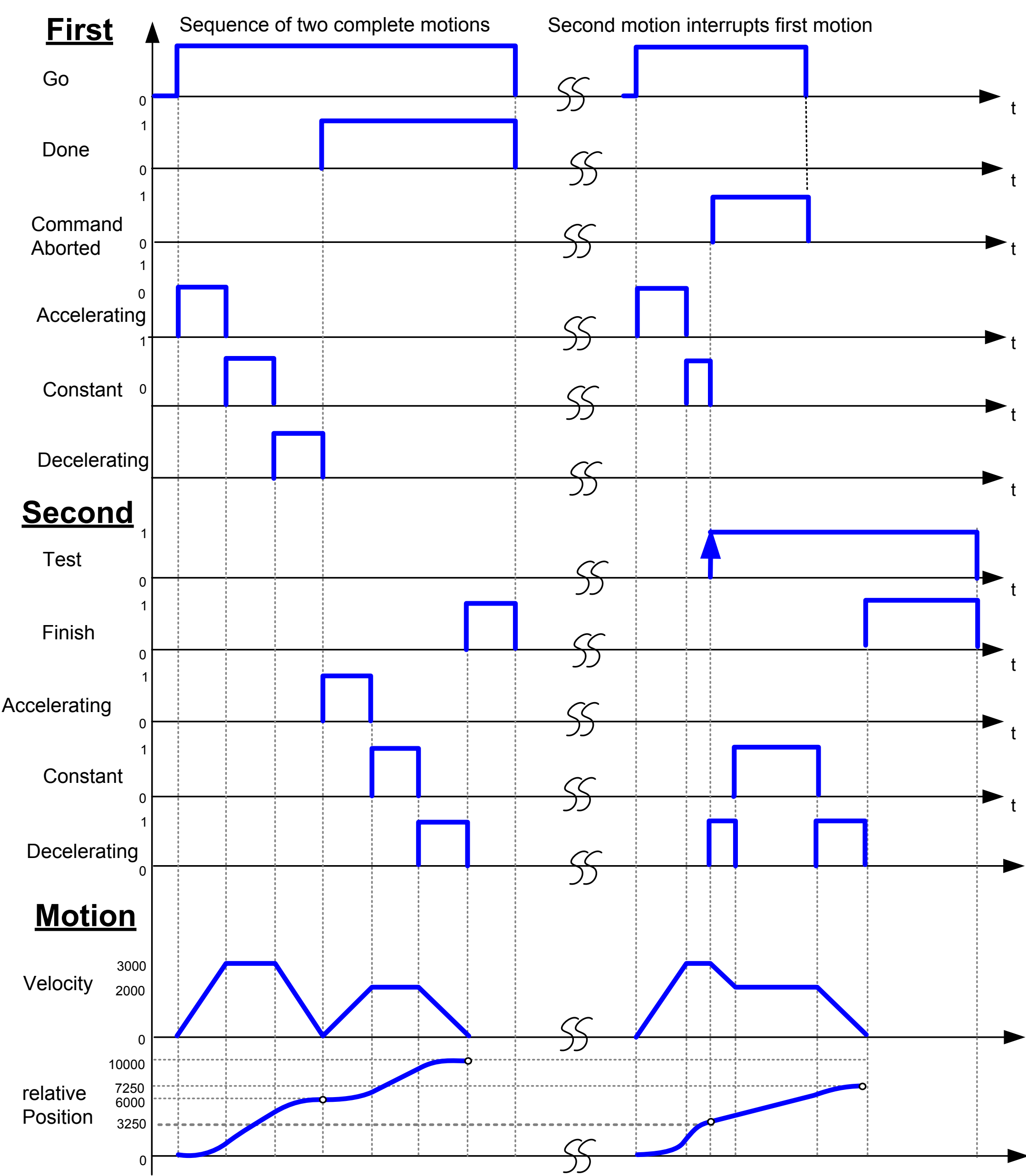


Figure 5-37: Combination of two blocks for MMC_MoveRelative with Timing diagram – Example

5.8.7 MMC_MoveVelocity

Commands a continuous controlled motion at a specified velocity.

```
MMC_LIB_API int MMC_MoveVelocityCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVEVELOCITY_IN* pInParam,
OUT MMC_MOVEVELOCITY_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVEVELOCITY_IN** input data structure using the MMC_MoveVelocity function.

pOutParam

Points to the **MMC_MOVEVELOCITY_OUT** output structure receiving information as a result of calling the MMC_MoveVelocity function.

Remarks

To stop the motion, the function block has to be interrupted by another function block issuing a new command.

The signal InVelocity has to be reset when the block is aborted by another block.

Scope

MMC_MoveVelocity will only operate from Standstill or Discrete Motion. It will not operate from Stopping, Homing, ErrorStop, or Disabled. Refer to the State diagram in **Figure** 5-1. Therefore, MMC_MoveVelocity should only be used with the MC_BUFFERED_MODE_ENUM eBufferMode enumerator value of MC_ABORTING_MODE.

MMC_MOVEVELOCITY_IN Structure

```
typedef struct{
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_DIRECTION_ENUM eDirection;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
}MMC_MOVEVELOCITY_IN;
```

Parameters

fVelocity

Value of the maximum velocity (not necessarily reached). The velocity states the direction and can be a positive or negative value in u/s.

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

MC_DIRECTION_ENUM eDirection

Specifies the direction of the motion, if any. MC_Direction enumerator type can have 1-of-4 values:

MC_NONE_DIRECTION = 0

MC_POSITIVE_DIRECTION = 1

MC_SHORTEST_WAY = 2 Not implemented as yet

MC_NEGATIVE_DIRECTION = 3

MC_CURRENT_DIRECTION = 4

The Enum type MC_SHORTEST_WAY is focused to a trajectory, which will go through the shortest route(option not implemented as yet). The decision which direction to move is based on the current position from where the command is issued.

In general in the interim, make sure to set the parameter to MC_POSITIVE_DIRECTION. In the future, all above parameters will be available.

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting — Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered — The next function block affects the axis as soon as the previous movement is completed.

BlendingLow — The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).

BlendingPrevious — Blending with the velocity of function block 1 at the end-position of this block

BlendingNext — Blending with the velocity of function block 2 at end-position of function block1

BlendingHigh — Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVEVELOCITY_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVEVELOCITY_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-38 describes the function block for MMC_MoveVelocity as applied within the IEC 61131 programming.

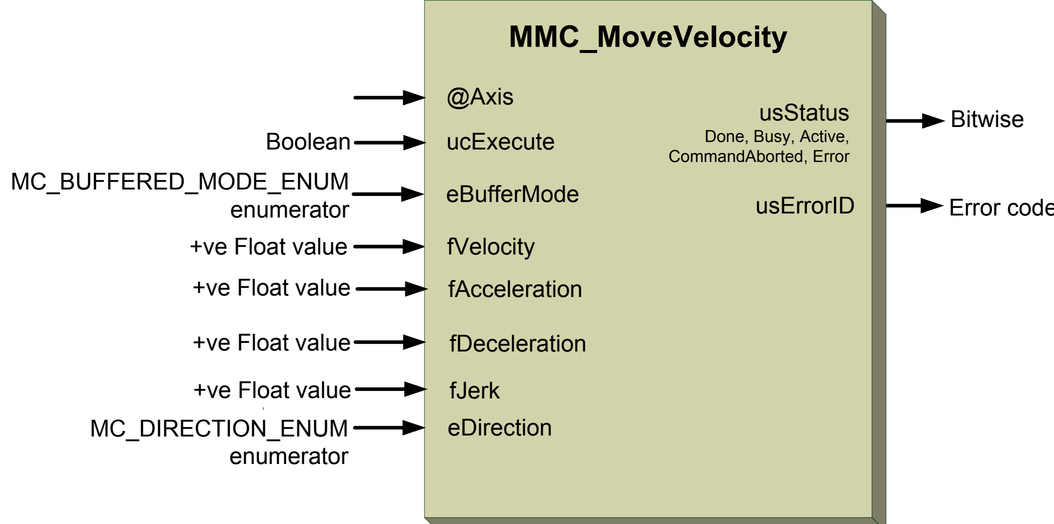


Figure 5-38: MMC_MoveVelocity function block

5.8.7.2 Function Block Code Example

```
int rc;
MMC_MOVEVELOCITY_IN    stMoveVel_in;
MMC_MOVEVELOCITY_OUT   stMoveVel_out;
//
// Inserting the structure parameters:
stMoveVel_in.fAcceleration  = 100000.0;    // Value of the acceleration
stMoveVel_in.fDeceleration  = 100000.0;    // Value of the deceleration
stMoveVel_in.fJerk      = 20000000.0;  // Value of the Jerk
stMoveVel_in.eBufferMode     = MC_BUFFERED_MODE; // MC_BUFFERED_MODE_ENUM Defines the behavior
of the axis
stMoveVel_in.eDirection      = MC_POSITIVE_DIRECTION; // MC_Direction Enumerator type
stMoveVel_in.fVelocity = 5000.0;      // Velocity in
stMoveVel_in.ucExecute = 1;
//
rc = MMC_MoveVelocityCmd (hConn, iAxisRef, &stMoveVel_in, &stMoveVel_out);
if (rc != 0)
{
    HandleError();
}
```

5.8.7.3 Implementation Example

The following figure shows two examples of the combination of two *MoveVelocity* function blocks:

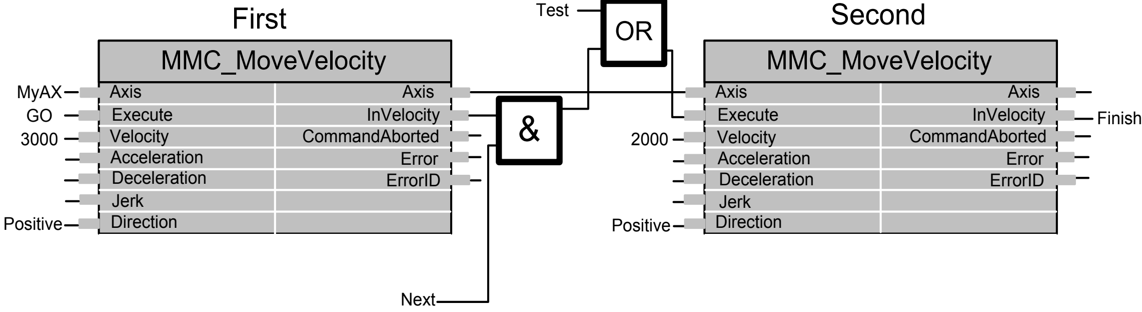
1. The left part of timing diagram illustrates the case if the Second function block is called after the First one is completed. If First reaches the commanded velocity 3000 then the output First.InVelocity AND the signal Next causes the Second function block to move to the velocity 2000. In the next cycle First.InVelocity is Reset and First.commandAborted is SET. Therefore, the Execute of the 2nd function block is Reset. In addition, as soon as the axis reaches Velocity 2000 the Second.InVelocity is set for 1 cycle (because the Second.Execute is set for only 1 cycle).
2. The right part of the timing diagram illustrates the case if the Second move Function Block starts the execution while the First function block is not yet InVelocity.

The following sequence is shown:

a. The First motion is started again by Go at the input First.Execute.

b. While the First function block is still accelerating to reach the velocity 3000 the First function block will be interrupted and aborted because the Test signal starts the Run of the Second function block.

c. Now the Second function block runs and decelerates the velocity to 2000.



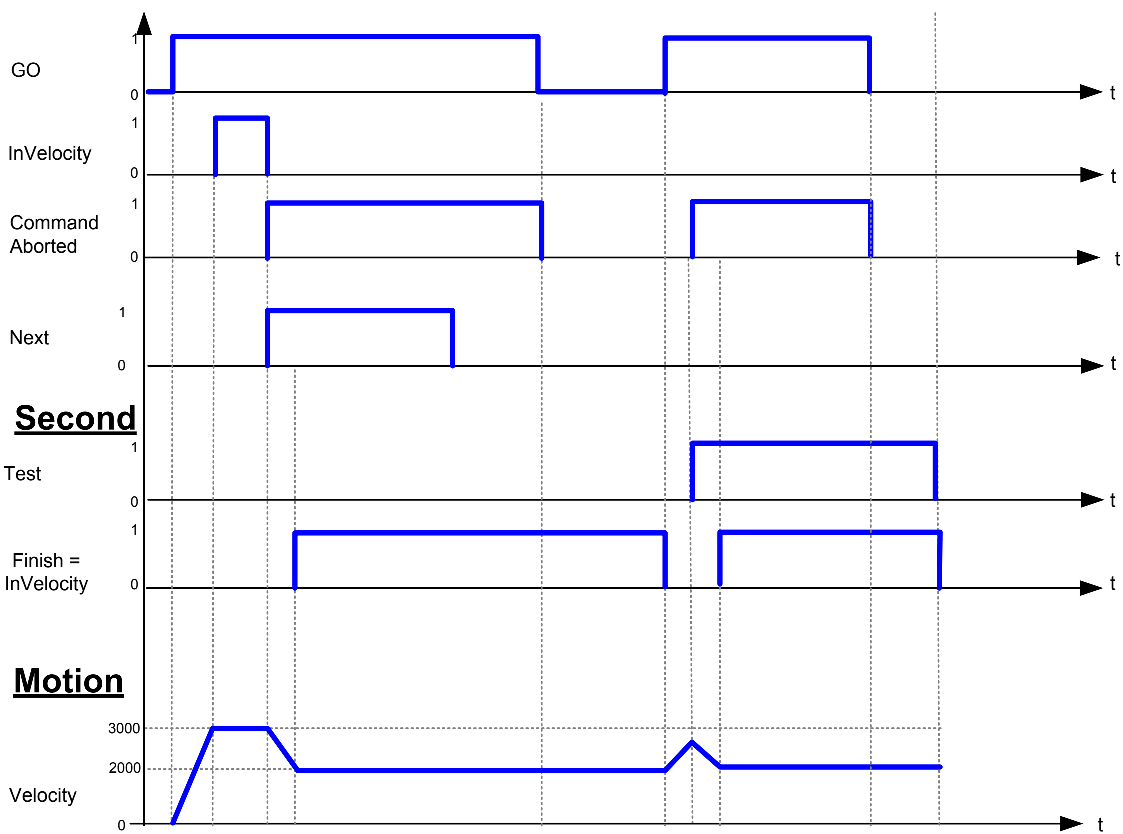


Figure 5-39: Combination of two blocks for MC_MoveVelocity with Timing diagram – Example

Note: **In the above example, the 2nd function block in mode Aborting (If in buffered mode the velocity would reach 3000 before actuating the next function block).**

5.8.8 MMC_MoveContinuous

This function sends a Move Continuous command to MMC server for a specific Axis.

```
MMC_LIB_API int MMC_MoveContinuousCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVECONTINUOUS_IN* pInParam,
OUT MMC_MOVECONTINUOUS_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVECONTINUOUS_IN** input data structure using the MMC_MoveContinuousCmd function.

pOutParam

Points to the **MMC_MOVECONTINUOUS_OUT** output structure receiving information as a result of calling the MMC_MoveContinuousCmd function.

Remarks

None

Scope

MMC_MoveContinuous will only operate from Standstill or Discrete Motion. It will not operate from Stopping, Homing, ErrorStop, or Disabled. Refer to the State diagram in **Figure 5-1**. Therefore, MMC_MoveContinuous should only be used with the MC_BUFFERED_MODE_ENUM eBufferMode enumerator value of MC_ABORTING_MODE.

MMC_MOVECONTINUOUS_IN Structure

```
typedef struct{
double dbDistance;
float fVelocity;
float fEndVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
}MMC_MOVECONTINUOUS_IN;
```

Parameters

dbDistance

Relative additive distance for the motion. Any -ve or +ve double values in technical unit [u].

fVelocity

Value of the maximum velocity (not necessarily reached). The velocity states the direction and can be a positive or negative value in u/s.

fEndVelocity

Value of the end velocity. The velocity states the direction and can be a positive or negative value in u/s.

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

MC_BUFFERED_MODE_ENUM eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE = 1
MC_BUFFERED_MODE = 2
MC_BLENDING_LOW_MODE = 3
MC_BLENDING_PREVIOUS_MODE = 4

	MC_BLENDING_NEXT_MODE = 5
	MC_BLENDING_HIGH_MODE = 6
Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. +ve character values.

MMC_MOVECONTINUOUS_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVECONTINUOUS_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-38 describes the function block for MMC_MoveContinuous as applied within the IEC 61131 programming.

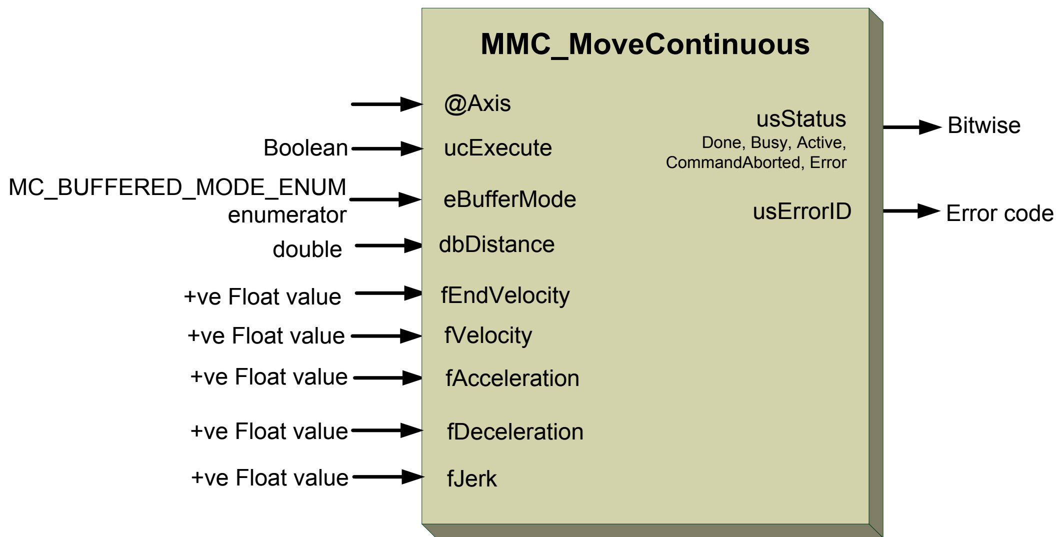


Figure 5-40: MMC_MoveContinuous function block

5.8.9 MMC_MoveAbsoluteRepetitive

This function receives as one of the input arguments, the command to move to the absolute target position. The axis moves between the current and target position until interrupted by any allowed function block in Aborting mode.

```
MMC_LIB_API int MMC_MoveAbsoluteRepetitiveCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVEABSOLUTEREPETITIVE_IN* pInParam,
OUT MMC_MOVEABSOLUTEREPETITIVE_OUT* pOutParam
);
```

Motion Mode	NC - All Buffering modes are supported.	Distributed - Not supported.
Source	GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVEABSOLUTEREPETITIVE_IN** input data structure using the MMC_MoveAbsoluteRepetitive function.

pOutParam

Points to the **MMC_MOVEABSOLUTEREPETITIVE_OUT** output structure receiving information, as a result of calling the MMC_MoveAbsoluteRepetitive function.

Remarks

It should be noted that all repetitive mode motion in iterpolated opMode will not operate in Abort Mode. An error is produced (GMAS Error ID -8).

Scope

The function block can only be called in StandStill mode. Calling MMC_Stop, MMC_Halt in aborting mode will stop the motion.

MMC_MOVEABSOLUTEREPETITIVE_IN Structure

```
typedef struct{
double dbPosition;
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_DIRECTION_ENUM eDirection;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned int uiExecDelayMs;
unsigned char ucExecute;
}MMC_MOVEABSOLUTEREPETITIVE_IN;
```

Parameters

dbPosition

Target position for the motion when conditions are met. Any -ve or +ve double values in technical unit [u].

fVelocity

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eDirection

Specifies the direction of the motion, if any. The MC_DIRECTION_ENUM enumerator type can have 1-of-4 values:

MC_NONE_DIRECTION = 0

MC_POSITIVE_DIRECTION = 1

MC_SHORTEST_WAY = 2 Not implemented as yet

MC_NEGATIVE_DIRECTION = 3

MC_CURRENT_DIRECTION = 4

The Enum type MC_SHORTEST_WAY is focused to a trajectory, which will go through the shortest route(option not implemented as yet). The decision which direction to

move is based on the current position from where the command is issued. In general, make sure to set the parameter to MC_POSITIVE_DIRECTION to be sent.

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

uiExecDelayMs

The delay in execution of the next action (in msecs). Any +ve integer value.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVEABSOLUTEREPETITIVE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVEABSOLUTEREPETITIVE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-41 describes the function block for MMC_MoveAbsoluteRepetitive as applied within the IEC 61131 programming.

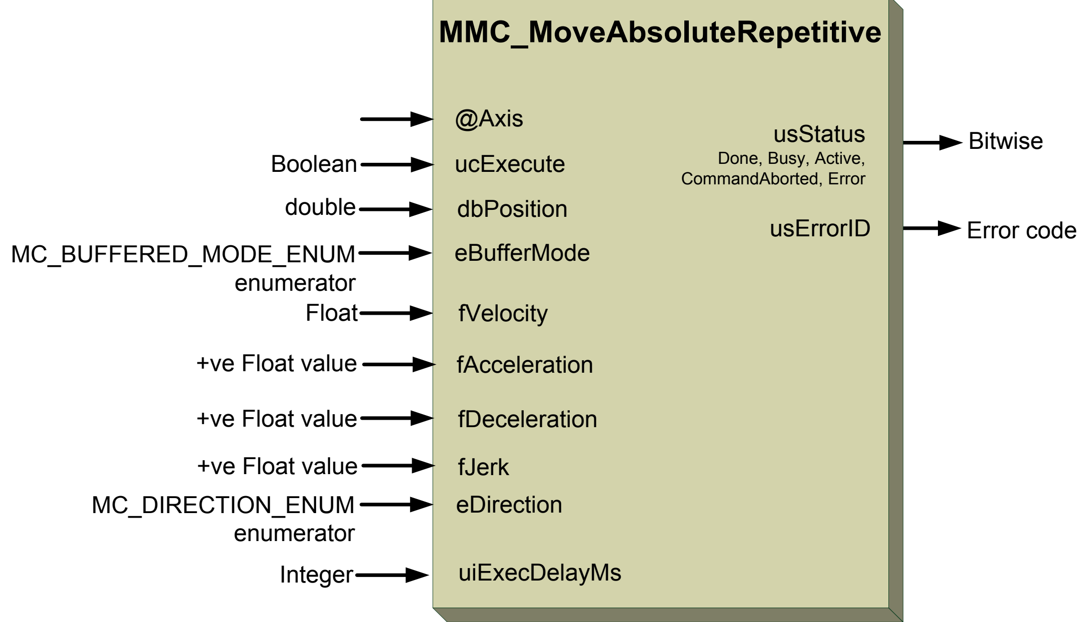


Figure 5-41: MMC_MoveAbsoluteRepetitive function block

5.8.9.2 Function Block Code Example

```
int rc;
MMC_MOVEABSOLUTEREPETITIVE_IN   stMoveAbsRepIn;
MMC_MOVEABSOLUTEREPETITIVE_OUT  stMoveAbsRepOut;
//
stMoveAbsRepIn.dbPosition       = 100000.0;
stMoveAbsRepIn.eBufferMode      = MC_BUFFERED_MODE_ENUM;
stMoveAbsRepIn.eDirection       = MC_POSITIVE_DIRECTION;
stMoveAbsRepIn.fAcceleration    = 1000000.0;
stMoveAbsRepIn.fDeceleration    = 1000000.0;
stMoveAbsRepIn.fJerk            = 10000000.0;
stMoveAbsRepIn.fVelocity        = 100000.0;
stMoveAbsRepIn.ucExecute        = 1;
stMoveAbsRepIn.uiExecDelayMs    = 0;
//
rc = MMC_MoveAbsoluteRepetitiveCmd (hConn, iAxisRef, &stMoveAbsRepIn, &stMoveAbsRepOut);
if (rc != 0)
{
HandleError();
}
```

5.8.10 MMC_MoveRelativeRepetitive

This function receives as one of the input arguments, the command to move to a distance relative to the current position. The axis moves between the current and target position until interrupted by any allowed function block in Aborting mode.

```
MMC_LIB_API int MMC_MoveRelativeRepetitiveCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVERELATIVEREPETITIVE_IN* pInParam,
OUT MMC_MOVERELATIVEREPETITIVE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed – Not supported
Source	GMAS\includes\MMC_PLCopen_single_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVERELATIVE_IN** input data structure using the MMC_MoveRelativeRepetitive function.

pOutParam

Points to the **MMC_MOVERELATIVE_OUT** output structure receiving information, as a result of calling the MMC_MoveRelativeRepetitive function.

Remarks

It should be noted that all repetitive mode motion will not operate in Abort Mode.

Scope

The function block can only be called in StandStill mode. Calling MMC_Stop, MMC_Halt in aborting mode will stop the motion.

MMC_MOVERELATIVEREPETITIVE_IN Structure

```
typedef struct{
double dbDistance;
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_DIRECTION_ENUM eDirection;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned int uiExecDelayMs;
unsigned char ucExecute;
}MMC_MOVERELATIVEREPETITIVE_IN;
```

Parameters

dbDistance

Relative additive distance for the motion. Any -ve or +ve double values in technical unit [u].

fVelocity

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eDirection

Specifies the direction of the motion, if any. The MC_DIRECTION_ENUM enumerator type can have 1-of-4 values:

MC_NONE_DIRECTION = 0

MC_POSITIVE_DIRECTION = 1

MC_SHORTEST_WAY = 2 Not implemented as yet

MC_NEGATIVE_DIRECTION = 3

MC_CURRENT_DIRECTION = 4

The Enum type MC_SHORTEST_WAY is focused to a trajectory, which will go through the shortest route (option not implemented as yet). The decision which direction to move is based on the current position from where the command is issued. In general, make sure to set the parameter to MC_POSITIVE_DIRECTION to be sent.

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

uiExecDelayMs

The delay in execution of the next action (in msecs). Any +ve integer value.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVERELATIVEREPETITIVE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVERELATIVEREPETITIVE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-42 describes the function block for MMC_MoveRelativeRepetitive as applied within the IEC 61131 programming.

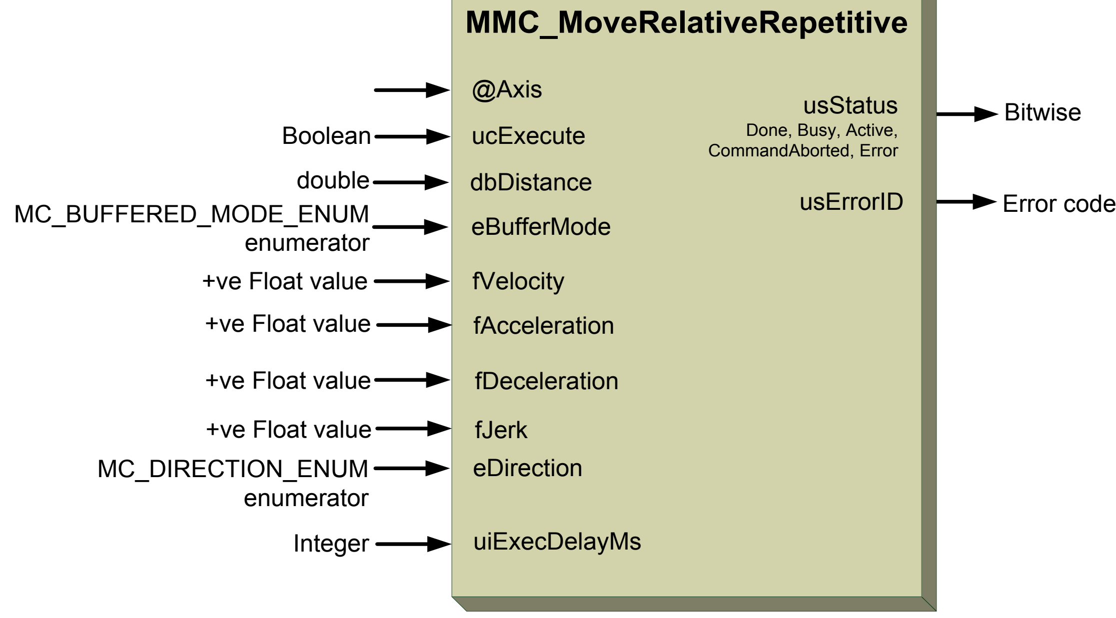


Figure 5-42: MMC_MoveRelativeRepetitive function block

5.8.10.2 Function Block Code Example

```
int rc;
MMC_MOVERELATIVEREPETITIVE_IN   stMoveRelRepIn;
MMC_MOVERELATIVEREPETITIVE_OUT  stMoveRelRepOut;
//
stMoveRelRepIn.dbDistance       = 100000.0;
stMoveRelRepIn.eBufferMode      = MC_BUFFERED_MODE_ENUM;
stMoveRelRepIn.eDirection       = MC_POSITIVE_DIRECTION;
stMoveRelRepIn.fAcceleration    = 1000000.0;
stMoveRelRepIn.fDeceleration    = 1000000.0;
stMoveRelRepIn.fJerk            = 10000000.0;
stMoveRelRepIn.fVelocity        = 100000.0;
stMoveRelRepIn.ucExecute        = 1;
stMoveRelRepIn.uiExecDelayMs    = 0;
//
rc = MMC_MoveRelativeRepetitiveCmd (hConn, iAxisRef, &stMoveRelRepIn, &stMoveRelRepOut);
if (rc != 0)
{
HandleError();
}
```

5.8.11 MMC_MoveAdditiveRepetitive

This function receives as one of the input arguments, the command to move to a distance relative to the final position of the last command. The axis moves between the current and target position until interrupted by any allowed function block in Aborting mode.

```
MMC_LIB_API int MMC_MoveAdditiveRepetitiveCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVEADDITIVEREPETITIVE_IN* pInParam,
OUT MMC_MOVEADDITIVEREPETITIVE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed – Not supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVEADDITIVEREPETITIVE_IN** input data structure using the MMC_MoveAdditiveRepetitive function.

pOutParam

Points to the **MMC_MOVEADDITIVEREPETITIVE_OUT** output structure receiving information, as a result of calling the MMC_MoveAdditiveRepetitive function.

Remarks

It should be noted that all repetitive mode motion will not operate in Abort Mode.

Scope

The function block can only be called in StandStill mode. Calling MMC_Stop, MMC_Halt in aborting mode will stop the motion.

MMC_MOVEADDITIVEREPETITIVE_IN Structure

```
typedef struct{
double dbDistance;
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_DIRECTION_ENUM eDirection;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned int uiExecDelayMs;
unsigned char ucExecute;
}MMC_MOVEADDITIVEREPETITIVE_IN;
```

Parameters

dbDistance

Relative additive distance for the motion. Any -ve or +ve double values in technical unit [u].

fVelocity

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^2

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eDirection

Specifies the direction of the motion, if any. The MC_DIRECTION_ENUM enumerator type can have 1-of-4 values:

MC_NONE_DIRECTION = 0

MC_POSITIVE_DIRECTION = 1

MC_SHORTEST_WAY = 2 Not implemented as yet

MC_NEGATIVE_DIRECTION = 3

MC_CURRENT_DIRECTION = 4

The Enum type MC_SHORTEST_WAY is focused to a trajectory, which will go through the shortest route(option not implemented as yet). The decision which direction to move is based on the current position from where the command is issued. In general, make sure to set the parameter to MC_POSITIVE_DIRECTION to be sent.

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

uiExecDelayMs

The delay in execution of the next action (in msecs). Any +ve integer value.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVEADDITIVEREPETITIVE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVEADDITIVEREPETITIVE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-43 describes the function block for MMC_MoveAdditiveRepetitive as applied within the IEC 61131 programming.

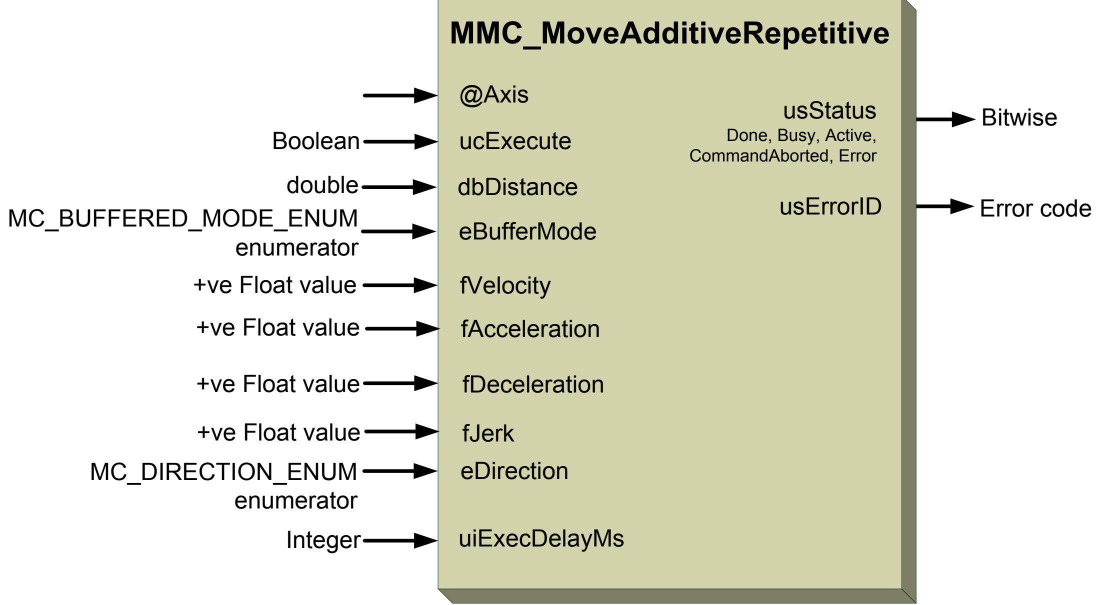


Figure 5-43: MMC_MoveAdditiveRepetitive function block

5.8.11.2 Function Block Code Example

```
int rc;
MMC_MOVEADDITIVEREPETITIVE_IN   stMoveAddRepIn;
MMC_MOVEADDITIVEREPETITIVE_OUT  stMoveAddRepOut;
//
stMoveAddRepIn.dbPosition       = 100000.0;
stMoveAddRepIn.eBufferMode      = MC_BUFFERED_MODE_ENUM;
stMoveAddRepIn.eDirection       = MC_POSITIVE_DIRECTION;
stMoveAddRepIn.fAcceleration    = 1000000.0;
stMoveAddRepIn.fDeceleration    = 1000000.0;
stMoveAddRepIn.fJerk            = 10000000.0;
stMoveAddRepIn.fVelocity        = 100000.0;
stMoveAddRepIn.ucExecute        = 1;
stMoveAddRepIn.uiExecDelayMs    = 0;
//
rc = MMC_MoveAdditiveRepetitiveCmd (hConn, iAxisRef, &stMoveAddRepIn, &stMoveAddRepOut);
if (rc != 0)
{
    HandleError();
}
```

5.8.12 MMC_Stop

Commands a controlled motion stop and transfers the axis to the state Stopping.

```
MMC_LIB_API int MMC_StopCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_STOP_IN* pInParam,
OUT MMC_STOP_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_STOP_IN** input data structure using the MMC_Stop function.

pOutParam

Points to the **MMC_STOP_OUT** output structure receiving information as a result of calling the MMC_Stop function.

Remarks

It aborts any ongoing function block execution. While the axis is in state Stopping, no other function block can perform any motion on the same axis. After the axis has reached velocity zero, the Done output is set to TRUE immediately. The axis remains in the state Stopping as long as Execute is still TRUE or velocity zero is not yet reached. As soon as Done is SET and Execute is FALSE the axis goes to state StandStill.
The state Stopping remains operational as long as the input Execute is true. The Done output is set when the stop ramp is completed. The following should be noted:

- This function block is primarily intended for emergency stop functions or exception situations
- As long as Execute is high, the axis remains in the state Stopping and may not execute any other command.
- If Deceleration = 0, the behavior of the function block is implementation specific.

Scope

MMC_Stop will only operate from Homing, Standstill or all Motion modes. It will not operate from ErrorStop, or Disabled. Refer to the State diagram in **Figure 5-1**. Calling the function block MMC_Stop in the state StandStill changes the state to Stopping and back to Standstill when Execute = FALSE.

MMC_STOP_IN Structure

```
typedef struct{
float fDeceleration;
float fJerk;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
}MMC_STOP_IN;
```

Parameters

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function

block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_STOP_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_STOP_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-44 describes the function block for MMC_Stop as applied within the IEC 61131 programming.

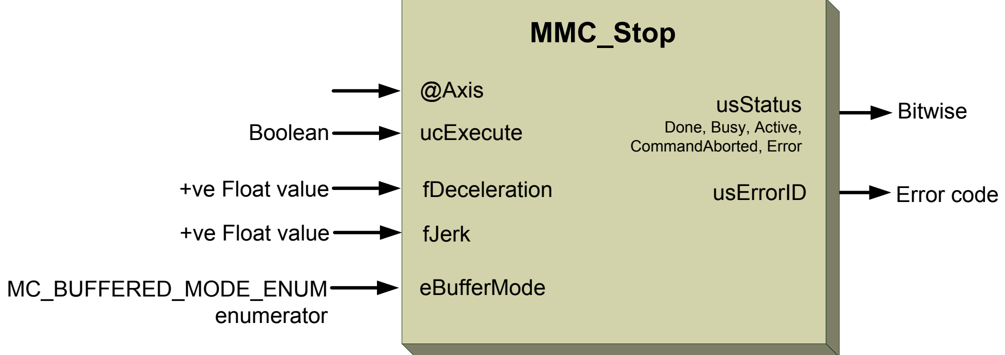


Figure 5-44: MMC_Stop function block

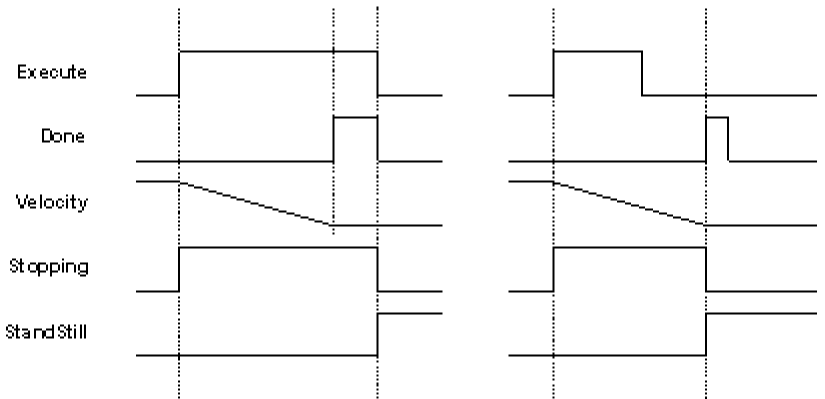


Figure 5-45: MMC_Stop timing diagram

5.8.12.2 Function Block Code Example

```
int rc;
MMC_STOP_IN    stStop_in;
MMC_STOP_OUT   stStop_out;
//
// Inserting the structure parameters:
stStop_in.fDeceleration     = 1000000.0;    // Value of the acceleration
stStop_in.fJerk             = 2000000.0;    // Value of the Jerk
stStop_in.eBufferMode       = MC_ABORTING_MODE;  // MC_BUFFERED_MODE_ENUM Defines the behavior
of the axis
stStop_in.ucExecute         = 1;
//
rc = MMC_StopCmd (hConn, iAxisRef, &stStop_in, &stStop_out);
if (rc != 0)
{
    HandleError();
}
```

5.8.12.3 Implementation Example

The example below shows the behavior of MMC_Stop in combination with an MMC_MoveVelocity.

1. A rotating axis is ramped down with function block MMC_Stop.
2. The axis rejects motion commands as long as MC_Stop parameter Execute = TRUE. The function block MMC_MoveVelocity reports an error indicating the busy MMC_Stop command.

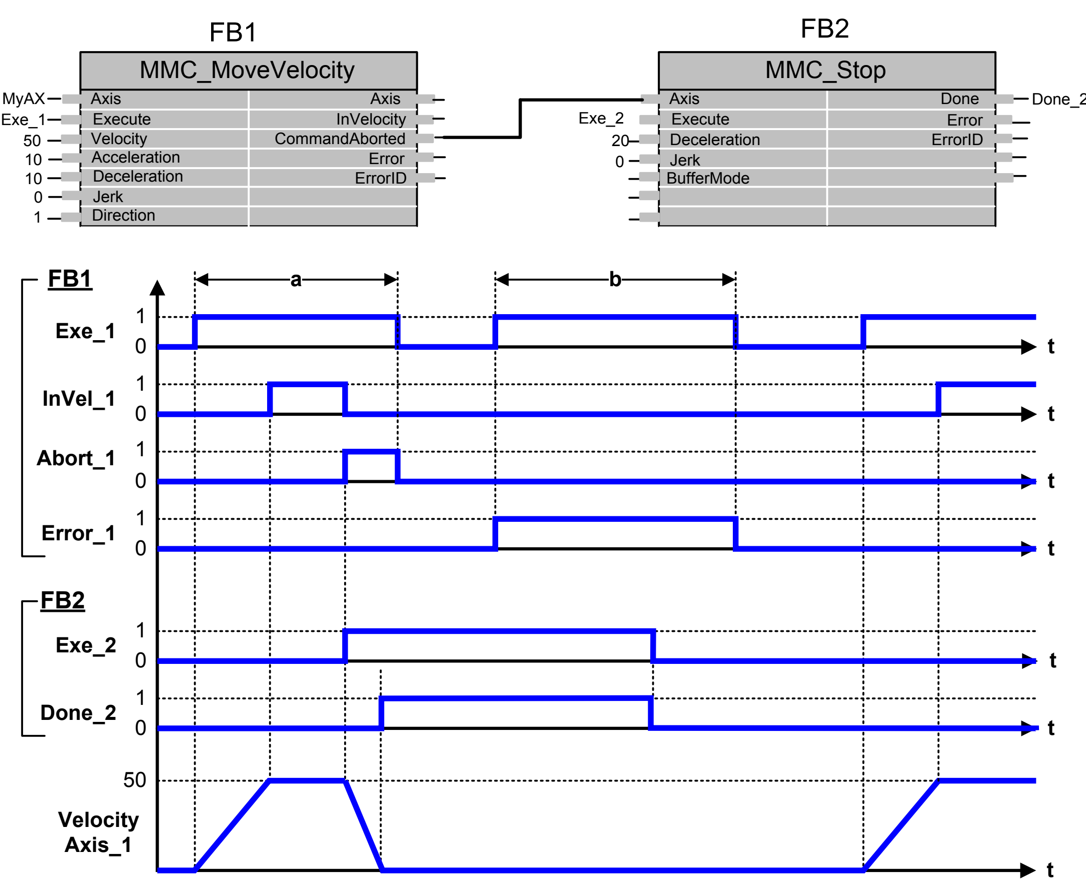


Figure 5-46: Combination of two blocks for MMC_Stop with their Timing diagram – Example

5.9 Single Axis Administrative Control

Administrative function blocks are blocks where no driving motion is involved. The following single axis administrative function blocks are described:

Single Axis
MMC_AxisLink
MMC_AxisUnLink
MMC_GetFBDepth
MMC_GetTotalFbDepth
MMC_GlobalReadBoolParameter
MMC_GlobalReadParameter
MMC_GlobalWriteBoolParameter
MMC_GlobalWriteParameter
MMC_PositionProfile
MMC_Power
MMC_ReadActualPosition
MMC_ReadActualTorque
MMC_ReadActualVelocity
MMC_ReadAxisError
MMC_ReadBoolParameter
MMC_ReadDigitalInput(s)
MMC_ReadDigitalOutputs
MMC_ReadDigitalOutputs32Bit
MMC_ReadParameter
MMC_ReadStatus
MMC_Reset
MMC_SetOverride
MMC_SetPosition
MMC_TouchProbeDisable
MMC_TouchProbeEnable
MMC_WriteBoolParameter
MMC_WriteDigitalOutputs
MMC_WriteDigitalOutputs32Bit
MMC_WriteParameter

5.9.1 Elmo SuperImposed Motion

Super Imposing a motion is the ability to impose an additional profiler to the ongoing motion. This is widely used in situations where the end position is unknown and requires changes during the motion, without crossing the initial location. The imposed Position, and Velocity are added to the ongoing motion.

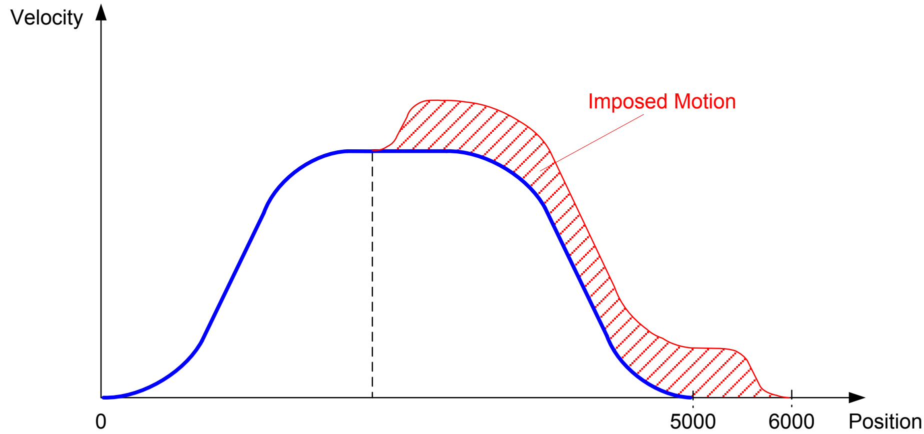


Figure 5-47 Elmo Superimposed Motion

The functions MMC_AxisLink and MMC_AxisUnlink perform this operation. This ability to perform SuperImposed motions is also integrated as part of the GMAS Script Manager functions in the EAS application. The Imposed motion is always a Virtual axis, and this can be Recorded in EAS, before and after the Superimposed is set, as shown in Figure 5-48.

Figure 5-48 Elmo Superimposed Motion Recorded in EAS

5.9.2 Special Function

The special function MMC_GETREACTORSTATISTICS communicates between the CANbus/EtherCAT and the single axis control, and is an administrative function block for the following:

- MMC_ID_AXIS_RELATE_MASK
- eMMC_FUNC_ID_AXIS_RELATED_DS301
- eMMC_FUNC_ID_AXIS_RELATED_DS401
- eMMC_FUNC_ID_AXIS_RELATED_DS402
- eMMC_FUNC_ID_AXIS_RELATED_DS406

5.9.3 MMC_AxisLink

This function creates a link between two axes defined as master (Primary) and slave (Minor).

```
MMC_LIB_API int MMC_AxisLink(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_AXISLINK_IN* pInParam,
OUT MMC_AXISLINK_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_AXISLINK_IN** input data structure using the MMC_AxisLink function.

pOutParam

Points to the **MMC_AXISLINK_OUT** output structure receiving information as a result of calling the MMC_AxisLink function.

Remarks

Linking two axes allows an object to move towards the approximate position of a target and then adjust the movement to accurately pinpoint the target. The initial movement is by the master (Primary) axis, whereas the adjustment is performed by the slave (Minor) axis. The resultant is sum of the master axis's position command, plus the slave axis's position command or the slave axis's actual position.

Scope

All

MMC_AXISLINK_IN Structure

```
typedef struct mmc_axislink_in{
unsigned long ulInputParameter1;
unsigned long ulInputParameter2;
unsigned long ulInputParameter3;
unsigned long ulInputParameter4;
unsigned short usSlaveAxisReference;
unsigned char ucMode;
}MMC_AXISLINK_IN;
```

Parameters

ulInputParameter1

For future use. Present value is 0 or no value.

ulInputParameter2

For future use. Present value is 0 or no value.

ulInputParameter3

For future use. Present value is 0 or no value.

ulInputParameter4

For future use. Present value is 0 or no value.

usSlaveAxisReference

Axis reference of the slave (Minor axis)

ucMode

The position of the master (Primary axis) which can have the following values:

0 – Target Position

1 – Actual position

MMC_AXISLINK_OUT Structure

```
typedef struct mmc_axislink_out{
unsigned short usStatus;
short sErrorID;
}MMC_AXISLINK_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-49 describes the function for MMC_AxisLink as applied within the IEC 61131 programming.

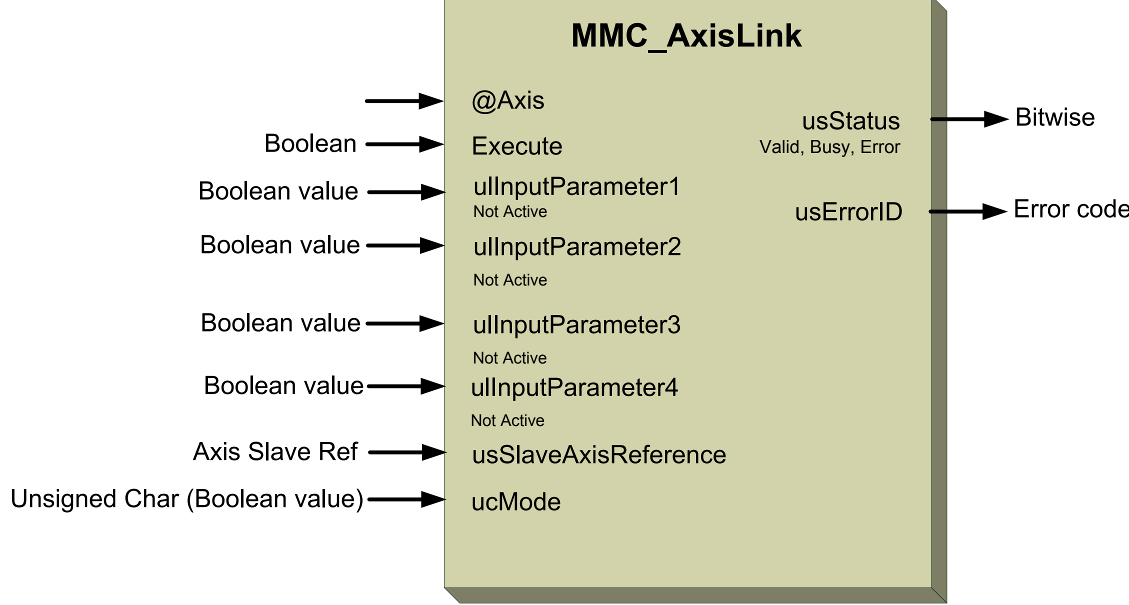


Figure 5-49: MMC_AxisLink function

5.9.4 MMC_AxisUnLink

This function breaks the link between two axes defined as master (Primary) and slave (Minor).

```
MMC_LIB_API int MMC_AxisUnLink(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_AXISUNLINK_IN* pInParam,
OUT MMC_AXISUNLINK_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_AXISUNLINK_IN** input data structure using the MMC_AxisLink function.

pOutParam

Points to the **MMC_AXISUNLINK_OUT** output structure receiving information as a result of calling the MMC_AxisLink function.

Remarks

None

Scope

All

MMC_AXISUNLINK_IN Structure

typedef struct mmc_axisunlink_in{
unsigned char ucdummy;
}MMC_AXISUNLINK_IN;

Parameters

ucdummy

Dummy value

MMC_AXISUNLINK_OUT Structure

typedef struct mmc_axisunlink_out{
unsigned short usStatus;
short sErrorID;
}MMC_AXISUNLINK_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-50 describes the function for MMC_AxisUnLink as applied within the IEC 61131 programming.

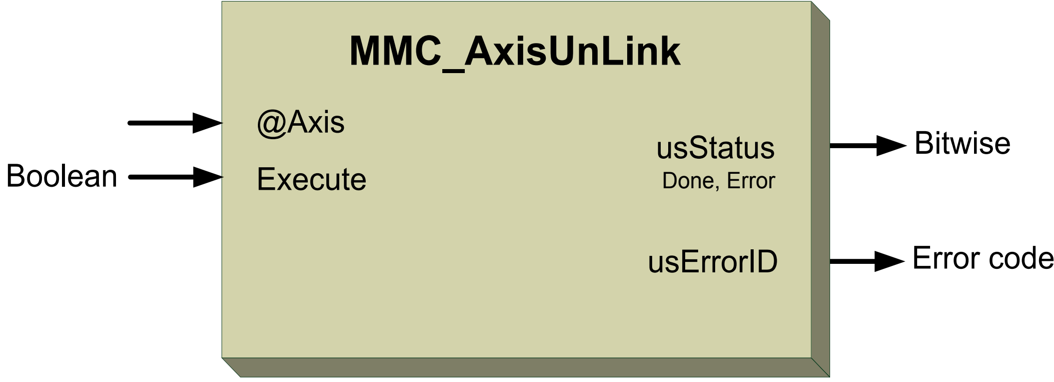


Figure 5-50: MMC_AxisUnLink function

5.9.5 MMC_Dwell

This function sends a temporary halt status command to the Maestro.

```
MMC_LIB_API int MMC_DwellCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_DWELL_IN* pInParam,
OUT MMC_DWELL_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Supported

Source GMAS\includes\MMC_general_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_DWELL_IN** input data structure using the MMC_Dwell function.

pOutParam

Points to the **MMC_DWELL_OUT** output structure receiving information as a result of calling the MMC_Dwell function.

Remarks

This function causes a "sleep state" between the motion and operation of the function blocks.

Scope

All

MMC_DWELL_IN Structure

```
typedef struct MMC_DWELL_IN{
unsigned long ulDwellTimeMs;
} MMC_DWELL_IN;
```

Parameters

ulDwellTimeMs

The time during which the Maestro rests in millisecs. Any +ve value.

However, the value cannot be smaller than the cycle time.

MMC_DWELL_OUT Structure

```
typedef struct MMC_DWELL_OUT{
unsigned int uiHandle;
unsigned short usStatus;
short sErrorID;
} MMC_DWELL_OUT;
```

Parameters

uiHandle

Handle to a journal entry where the pointer to the shared memory is located, and indicates the memory area where the spline is allocated. In fact this is the unique identifier between PathSelect, MovePath and PathUnselect commands. MC_PATH_REF is the journal entry path reference.

uiHandle produces +ve Integer values.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 5-51 describes the function for MMC_Dwell as applied within the IEC 61131 programming.

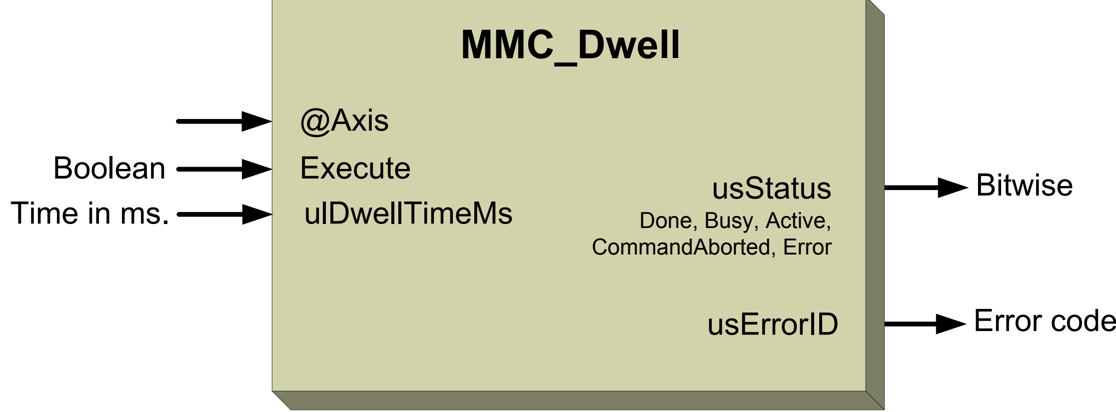


Figure 5-51: MMC_Dwell function

5.9.6 MMC_GetFBDepth

Sends a command to receive the number of function blocks in the Node Queue waiting to be activated, or presently activated, which did not receive the Done or Abort status.

```
MMC_LIB_API int MMC_GetFbDepthCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETFBDEPTH_IN* pInParam,
OUT MMC_GETFBDEPTH_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETFBDEPTH_IN** input data structure using the MMC_GetFBDepth function.

pOutParam

Points to the **MMC_GETFBDEPTH_OUT** output structure receiving information as a result of calling the MMC_GetFBDepth function.

Remarks

None

Scope

All

MMC_GETFBDEPTH_IN Structure

```
typedef struct{
unsigned int uiHndl;
}MMC_GETFBDEPTH_IN;
```

Parameters

uiHndl

Returned function block handle. **Not in use at this moment.**

MMC_GETFBDEPTH_OUT Structure

```
typedef struct{
unsigned int  uiFbInQ;
unsigned short usStatus;
short sErrorID;
}MMC_GETFBDEPTH_OUT;
```

Parameters

uiFbInQ

Actual numbr of function blocks in queue. Any +ve integer value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-52 describes the function block for MMC_GetFBDepth as applied within the IEC 61131 programming.

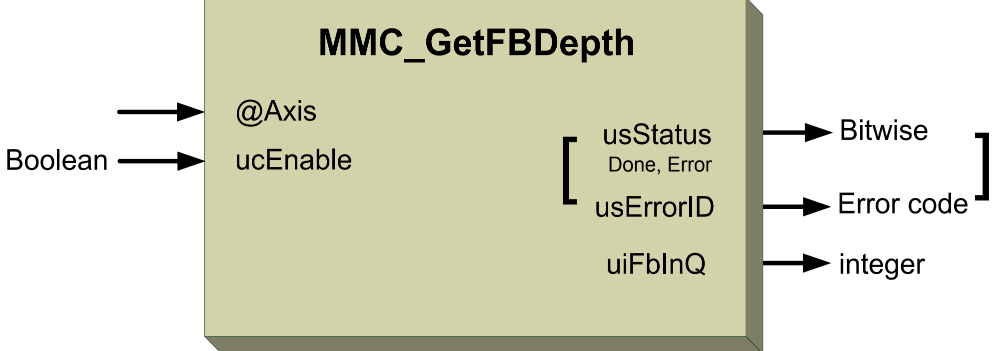


Figure 5-52: MMC_GetFBDepth function block

5.9.6.2 Function Block Code Example

```
int rc;
MMC_GETFBDEPTH_IN      stGetFBDepth_in;
MMC_GETFBDEPTH_OUT     stGetFBDepth_out;
//
// Inserting the structure parameters:
stGetFBDepth_in.uiHndl     = 100.0;      // Returned function block handle

//
rc = MMC_GetFbDepthCmd (hConn, iAxisRef, &stGetFBDepth_in, &stGetFBDepth_out);
printf("FB Depth[%ld] ErrId[%d]\n", (long int)stGetFBDepth_out.uiFbInQ,
(short)stGetFBDepth_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.9.7 MMC_GetTotalFbDepth

Sends a command to receive the total number of function blocks in the Node Queue, waiting to be activated, presently activated, and previously activated but not released to the function block pool.

```
MMC_LIB_API int MMC_GetTotalFbDepthCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETFBDEPTH_IN* pInParam,
OUT MMC_GETFBDEPTH_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETFBDEPTH_IN** input data structure using the MMC_GetFBDepth function.

pOutParam

Points to the **MMC_GETFBDEPTH_OUT** output structure receiving information as a result of calling the MMC_GetFBDepth function.

Remarks

None

Scope

All

MMC_GETFBDEPTH_IN Structure

```
typedef struct{
unsigned int uiHndl;
}MMC_GETFBDEPTH_IN;
```

Parameters

uiHndl

Returned function block handle. **Not in use at this moment.**

MMC_GETFBDEPTH_OUT Structure

```
typedef struct{
unsigned int  uiFbInQ;
unsigned short usStatus;
short sErrorID;
}MMC_GETFBDEPTH_OUT;
```

Parameters

uiFbInQ

Actual number of function blocks in queue. Any +ve integer value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-53 describes the function block for MMC_GetTotalFBDepth as applied within the IEC 61131 programming.

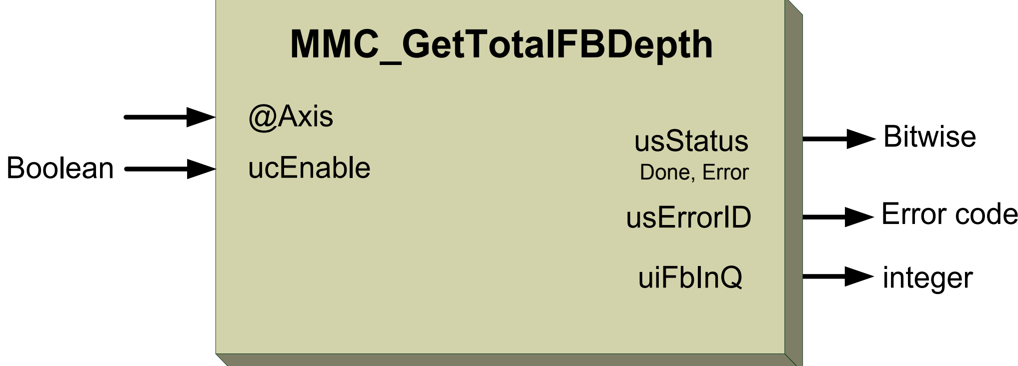


Figure 5-53: MMC_GetTotalFBDepth function block

5.9.7.2 Function Block Code Example

```
int rc;
MMC_GETFBDEPTH_IN      stGetFBDepth_in;
MMC_GETFBDEPTH_OUT     stGetFBDepth_out;
//
// Inserting the structure parameters:
stGetFBDepth_in.uiHndl      = 100.0;      // Returned function block handle

//
rc = MMC_GetFbDepthCmd (hConn, iAxisRef, &stGetFBDepth_in, &stGetFBDepth_out);
printf("FB Depth[%ld] ErrId[%d]\n", (long int)stGetFBDepth_out.uiFbInQ,
(short)stGetFBDepth_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.9.8 MMC_Power

Controls the power stage (On or Off).

```
MMC_LIB_API int MMC_PowerCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_POWER_IN* pInParam,
OUT MMC_POWER_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_POWER_IN** input data structure using the MMC_Power function.

pOutParam

Points to the **MMC_POWER_OUT** output structure receiving information as a result of calling the MMC_Power function.

Remarks

It is possible to set an error variable when the Command is TRUE for a while and the Status remains false with a Timer function block and an AND Function (with inverted Status input). It indicates that there is a hardware problem with the power stage.

If power fails (also during operation) it will generate a transition to the ErrorStop state

Enable_Positive and Enable_Negative are both level triggered

Scope

MMC_Power can be set to ON or OFF. If the MC_Power function block is called when Enable = TRUE, while being in Disabled, the axis state changes to StandStill.

MMC_POWER_IN Structure

```
typedef struct{
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucEnable;
unsigned char ucEnablePositive;
unsigned char ucEnableNegative;
unsigned char ucExecute;
}MMC_POWER_IN;
```

Parameters

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucEnable

Get the value of the parameter continuously while enabled. As long as Enable is true, power is enabled. Character values of 0, 1 integers.

ucEnablePositive

As long as Enable is true, permits motion in positive direction. Boolean values TRUE/FALSE accepted.

ucEnableNegative

As long as Enable is true, permits motion in negative direction (_Positive & _Negative can both be true). Boolean values TRUE/FALSE accepted.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_POWER_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_POWER_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-54 describes the function block for MMC_Power as applied within the IEC 61131 programming.

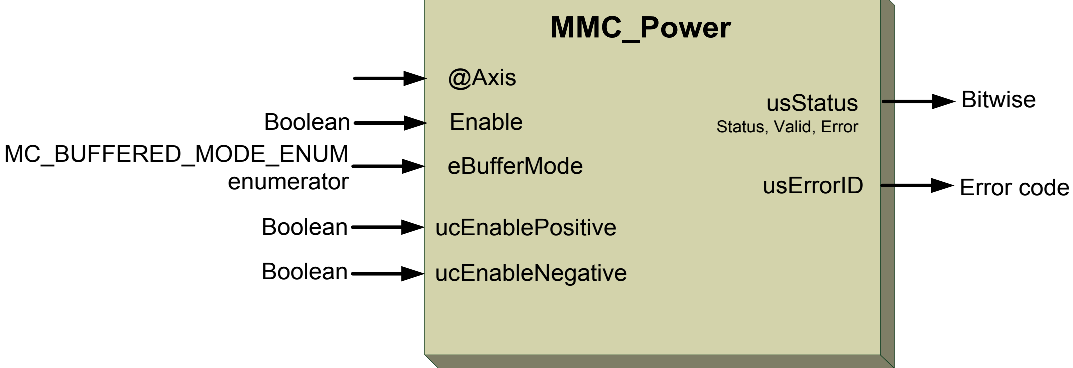


Figure 5-54: MMC_Power function block

5.9.8.2 Function Block Code Example

```
int rc;
MMC_POWER_IN      stPower_in;
MMC_POWER_OUT     stPower_out;
//
// Inserting the structure parameters:
stPower_in.ucEnable            = 1;              //Enabled
stPower_in.ucEnablePositive    = 1;              //Permits motion in positive direction
stPower_in.ucEnableNegative    = 0;              //Does not permit motion in negative
direction
stPower_in.eBufferMode         = MC_BUFFERED_MODE;  //MC_BUFFERED_MODE_ENUM Defines the
behavior of the axis
stPower_in.ucExecute           = 1;
//
rc = MMC_PowerCmd (hConn, iAxisRef, &stPower_in, &stPower_out);
if (rc != 0)
{
    HandleError();
}
```

5.9.9 MMC_PositionProfile

Describes the position profile of an axis.

```
MMC_LIB_API int MMC_PositionProfileCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_POSITIONPROFILE_IN* pInParam,
OUT MMC_POSITIONPROFILE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_POSITIONPROFILE_IN** input data structure using the MMC_PositionProfile function.

pOutParam

Points to the **MMC_POSITIONPROFILE_OUT** output structure receiving information as a result of calling the MMC_PositionProfile function.

Remarks

None

Scope

All

MMC_POSITIONPROFILE_IN Structure

```
typedef struct mmc_positionprofile_in{
MC_BUFFERED_MODE_ENUM eBufferMode;
MC_PATH_REF hMemHandle;
unsigned char ucExecute;
}MMC_POSITIONPROFILE_IN;
```

Parameters

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle can have integer values.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_POSITIONPROFILE_OUT Structure

```
typedef struct mmc_positionprofile_out{
unsigned short usStatus;
unsigned short sErrorID;
}MMC_POSITIONPROFILE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-55 describes the function block for MMC_PositionProfile.

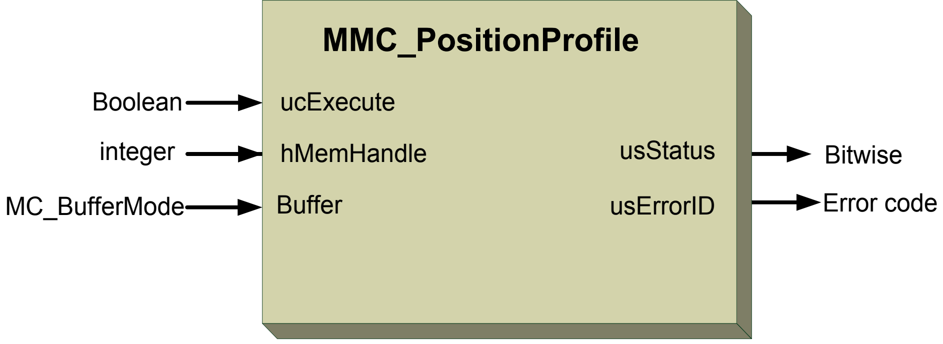


Figure 5-55: MMC_PositionProfile function block

5.9.9.2 Function Block Code Example

TBD

5.9.10 MMC_ReadActualPosition

Returns the actual position of the controlled axis.

```
MMC_LIB_API int MMC_ReadActualPositionCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READACTUALPOSITION_IN* pInParam,
OUT MMC_READACTUALPOSITION_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READACTUALPOSITION_IN** input data structure using the MMC_ReadActualPosition function.

pOutParam

Points to the **MMC_READACTUALPOSITION_OUT** output structure receiving information as a result of calling the MMC_ReadActualPosition function.

Remarks

For CAN only, depending on the mode, the position may not be updated. If Distributed motion mode is in operation, then this will depend whether the parameter was mapped via PDO mapping. Refer to the PDO3, 4 configuration mapping functions in section **12.7**.

For EtherCAT, the actual position must be part of the Input Group configured.

Scope

All

MMC_READACTUALPOSITION_IN Structure

```
typedef struct{
unsigned char ucEnable;
}MMC_READACTUALPOSITION_IN;
```

Parameters

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READACTUALPOSITION_OUT Structure

```
typedef struct{
double dbPosition;
unsigned short usStatus;
short sErrorID;
}MMC_READACTUALPOSITION_OUT;
```

Parameters

dbPosition

New absolute position. Any -ve or +ve double values in axis's unit [u]

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-56 and Figure 5-57 describe the function block for MMC_ReadActualPosition as applied within the IEC 61131 programming.

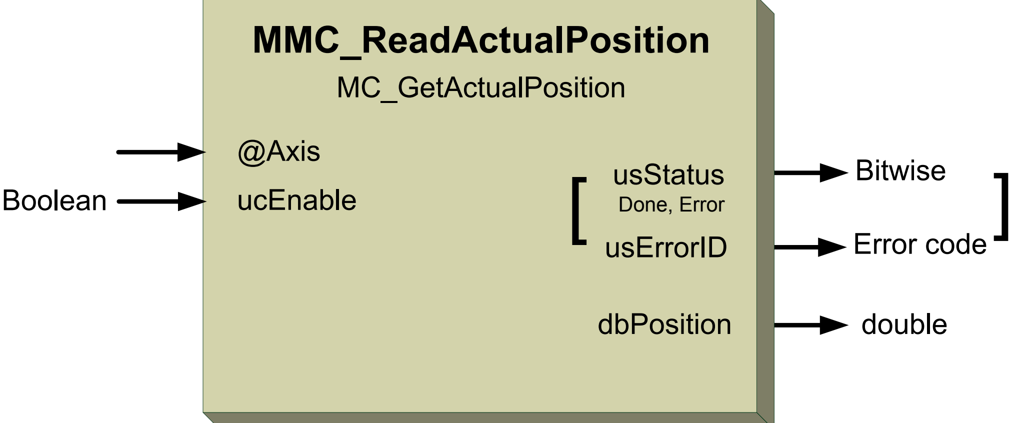


Figure 5-56: MMC_GetActualPosition function block

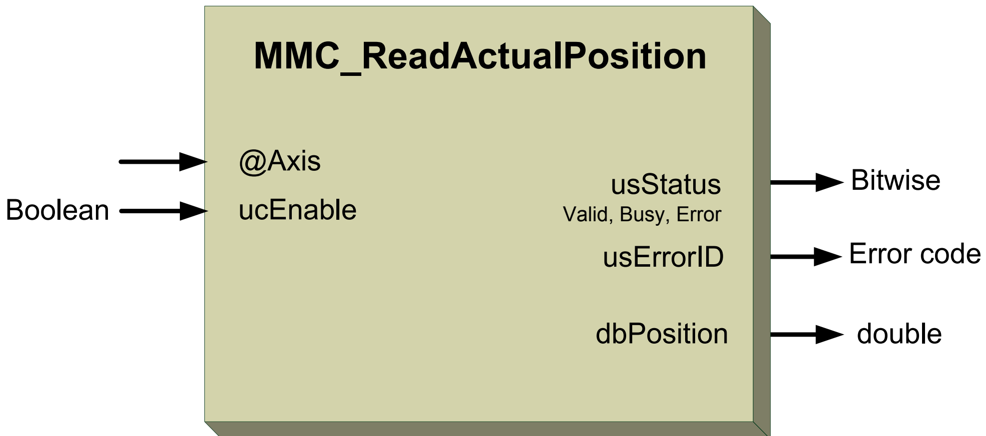


Figure 5-57: MMC_ReadActualPosition function block

5.9.10.2 Function Block Code Example

```
int rc ;
MMC_READACTUALPOSITION_IN       stReadActPos_in ;
MMC_READACTUALPOSITION_OUT      stReadActPos_out ;
//
// Inserting the structure parameters:
stReadActPos_in.ucEnable = 1; //Enabled
//
rc = MMC_ReadActualPositionCmd (hConn, iAxisRef,&stReadActPos_in, &stReadActPos_out);
printf("Actual Position[%ld] ErrId[%d]\n", (long int)stReadActPos_out.dbPosition,
(short)stReadActPos_out.sErrorID);
if (rc != 0)
{
    HandleError() ;
}
```

5.9.11 MMC_ReadActualTorque

Returns the actual torque value or force for a controlled axis, as long as Enable is set.

```
MMC_LIB_API int MMC_ReadActualTorqueCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READACTUALTORQUE_IN* pInParam,
OUT MMC_READACTUALTORQUE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h
GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READACTUALTORQUE_IN** input data structure using the MMC_ReadActualTorque function.

pOutParam

Points to the **MMC_READACTUALTORQUE_OUT** output structure receiving information as a result of calling the MMC_ReadActualTorque function.

Remarks

For CAN only, depending on the mode, the position may not be updated. The torque is mapped and not the position when in all opModes, not just in interpolated mode.

If Distributed motion mode is in operation, then this will depend whether the parameter was mapped via PDO mapping. Refer to the PDO3, 4 configuration mapping functions in section **16.4.3**.

For EtherCAT, the actual position must be part of the Input Group configured.

Scope

The usStatus mode Done is TRUE when the data-output Torque is valid. If Enable is Reset, the data loses its validity, and Done is also reset, no matter if new data is available.

The output dActualTorque can be a signed value.

MMC_READACTUALTORQUE_IN Structure

```
typedef struct{
unsigned char ucEnable;
}MMC_READACTUALTORQUE_IN;
```

Parameters

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READACTUALTORQUE_OUT Structure

```
typedef struct{
double dActualTorque;
unsigned short usStatus;
short sErrorID;
unsigned char ucValid;
}MMC_READACTUALTORQUE_OUT;
```

Parameters

dActualTorque

The value of the actual torque or force. Any -ve or +ve double values in torque units.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

ucValid

Values inputted are valid or not. Boolean values TRUE/FALSE accepted.

Figure 5-58 describes the function block for MMC_ReadActualTorque as applied within the IEC 61131 programming.

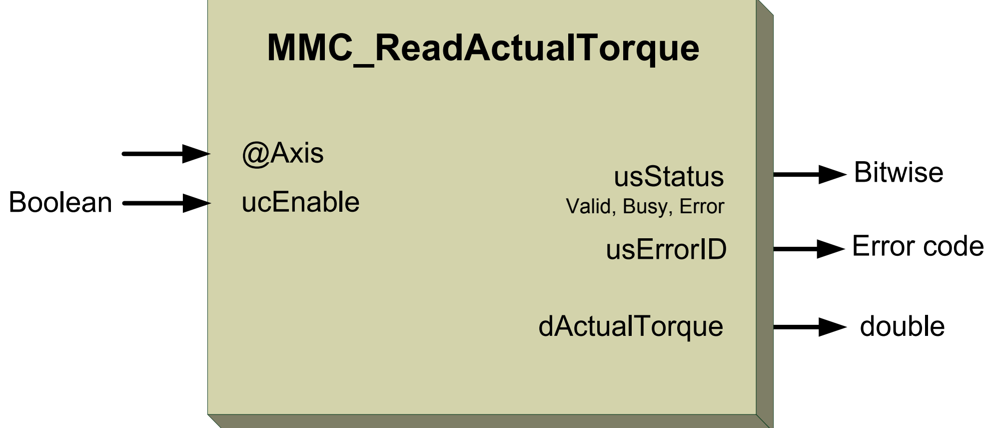


Figure 5-58: MMC_ReadActualTorque function block

5.9.11.2 Function Block Code Example

```
int rc;
MMC_READACTUALTORQUE_IN         stReadActTorque_in;
MMC_READACTUALTORQUE_OUT        stReadActTorque_out;
//
// Inserting the structure parameters:
stReadActTorque_in.ucEnable  = 1;   // Enabled
//
rc = MMC_ReadActualTorqueCmd (hConn, iAxisRef, &stReadActTorque_in, &stReadActTorque_out);
printf("Actual Torque[%ld] ErrId[%d]\n", (long int)stReadActTorque_out.dActualTorque,
(short)stReadActTorque_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.9.12 MMC_ReadActualVelocity

Returns the actual velocity value, as long as Enable is set.

```
MMC_LIB_API int MMC_ReadActualPositionCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READACTUALPOSITION_IN* pInParam,
OUT MMC_READACTUALPOSITION_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READACTUALVELOCITY_IN** input data structure using the MMC_ReadActualVelocity function.

pOutParam

Points to the **MMC_READACTUALVELOCITY_OUT** output structure receiving information as a result of calling the MMC_ReadActualVelocity function.

Remarks

For CAN only, depending on the mode, the position may not be updated. If Distributed motion mode is in operation, then this will depend whether the parameter was mapped via PDO mapping. Refer to the PDO3, 4 configuration mapping functions in section **12.7**.

For EtherCAT, the actual position must be part of the Input Group configured.

Scope

Valid is true when the data-output Velocity is valid. If Enable is Reset, the data loses its validity, and all outputs are reset, no matter if new data is available.

The output dVelocity can be a signed value.

MMC_READACTUALVELOCITY_IN Structure

```
typedef struct{
unsigned char ucEnable;
}MMC_READACTUALVELOCITY_IN;
```

Parameters

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READACTUALVELOCITY_OUT Structure

```
typedef struct{
double dVelocity;
unsigned short usStatus;
short sErrorID;
}MMC_READACTUALVELOCITY_OUT;
```

Parameters

dVelocity

The value of the actual velocity. Any positive double value in u/s units

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-59 describes the function block for MMC_ReadActualVelocity as applied within the IEC 61131 programming.

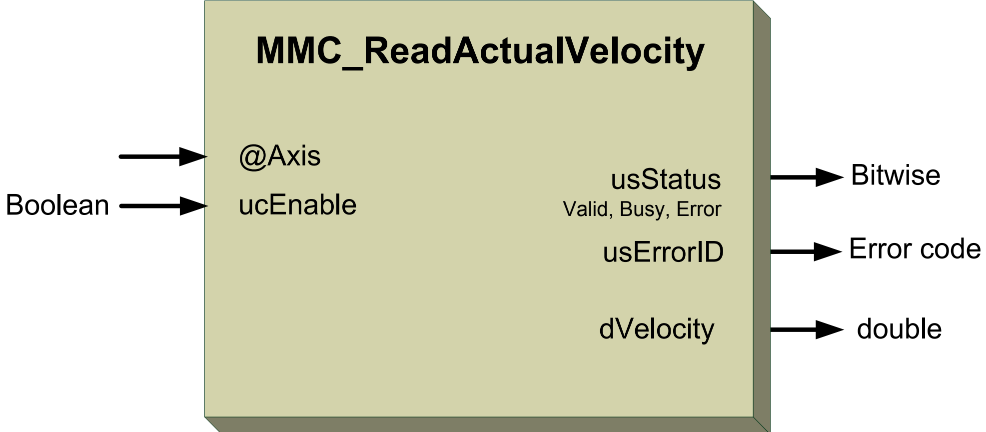


Figure 5-59: MMC_ReadActualVelocity function block

5.9.12.2 Function Block Code Example

```
int rc;
MMC_READACTUALVELOCITY_IN       stReadActVel_in;
MMC_READACTUALVELOCITY_OUT      stReadActVel_out;
//
// Inserting the structure parameters:
stReadActVel_in.ucEnable = 1;
//
rc = MMC_ReadActualVelocityCmd (hConn, iAxisRef, &stReadActVel_in, &stReadActVel_out);
printf("Actual Velocity[%ld] ErrId[%d]\n", (long int)stReadActVel_out.dVelocity,
(short)stReadActVel_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.9.13 MMC_ReadAxisError

Displays general axis errors not relating to the function blocks e.g. axis errors, drive errors, communication errors.		
MMC_LIB_API int MMC_ReadAxisError(IN MMC_CONNECT_HNDL hConn, IN MMC_AXIS_REF_HNDL hAxisRef, IN MMC_READAXISERROR_IN* pInParam, OUT MMC_READAXISERROR_OUT* pOutParam);		
Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_general_API.h GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	
Function Parameters		
hConn		
	[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.	
hAxisRef		
	Axis/group reference handle type returned by GetAxisRef command	
pInParam		
	Points to the **MMC_READAXISERROR_IN** input data structure using the MMC_ReadAxisError function.	
pOutParam		
	Points to the **MMC_READAXISERROR_OUT** output structure receiving information as a result of calling the MMC_ReadAxisError function.	
Remarks		
None		
Scope		
All		

MMC_READAXISERROR_IN Structure	
typedef struct{ unsigned char ucEnable; }MMC_READAXISERROR_IN;	
Parameters	
ucEnable	
	This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READAXISERROR_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned short usAxisErrorID;
unsigned short usLastEmergencyErrCode;
}MMC_READAXISERROR_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

usAxisErrorID

Returns the axis error bitwise ID defined by the following enumerators. Bitwise ID error code:

ID	Enumerator
0x1	MMC_ERR_TYPE_FAULT_BIT
0x2	MMC_ERR_TYPE_HEARTBEAT
0x4	MMC_ERR_TYPE_EMERGENCY
0x8	MMC_ERR_TYPE_COMM
0x10	MMC_ERR_TYPE_CFG_FILE
0x12	MMC_ERR_TYPE_UNEXPECTED
0x14	MMC_ERR_TYPE_AXIS_FAULT
0x16	MMC_ERR_TYPE_AL_ERROR

usLastEmergencyErrCode

Last emergency DS-402 error code that occurred in the system. Refer to the DS-402 Elmo emergency codes in the following:

- CANopen DS-301 Implementation Guide Chapter 6
- CANopen DSP-402 Implementation Guide
- SimplIQ Command Reference Guide or Gold Command Reference Guide

Figure 5-60 describes the function block for MMC_ReadAxisError as applied within the IEC 61131 programming. It should be noted that the parameter usLastEmergencyErrCode is not supported in IEC at this time.

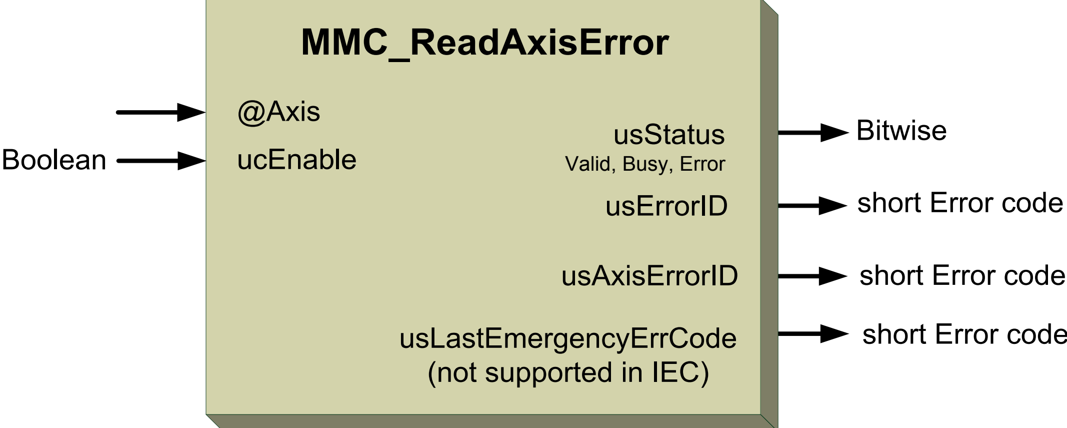


Figure 5-60: MMC_ReadAxisError function block

5.9.13.2 Function Block Code Example

```
int rc ;
MMC_READAXISERROR_IN   stReadAxisError_in;
MMC_READAXISERROR_OUT  stReadAxisError_out;
//
// Inserting the structure parameters:
stReadAxisError_in.ucEnable  = 1;        // Enabled
//
rc = MMC_ReadAxisError (hConn, iAxisRef, &stReadAxisError_in, &stReadAxisError_out) ;
printf("Axis Error[%ld] ErrId[%d]\n", (long int)stReadAxisError_out.usAxisErrorID,
(short)stReadAxisError_out.sErrorID, (short)stReadAxisError_out.usLastEmergencyErrCode);
if (rc != 0)
{
    HandleError() ;
}
```

5.9.14 MMC_ReadBoolParameter

Returns the value of a vendor specific with datatype unsigned long or un signed int.

```
MMC_LIB_API int MMC_ReadBoolParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READBOOLPARAMETER_IN* pInParam,
OUT MMC_READBOOLPARAMETER_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READBOOLPARAMETER_IN** input data structure using the MMC_ReadBoolParameter function.

pOutParam

Points to the **MMC_READBOOLPARAMETER_OUT** output structure receiving information as a result of calling the MMC_ReadBoolParameter function.

Remarks

None

Scope

All

MMC_READBOOLPARAMETER_IN Structure

```
typedef struct{
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_READBOOLPARAMETER_IN;
```

Parameters

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READBOOLPARAMETER_OUT Structure

```
typedef struct{
long lValue;
unsigned short usStatus;
short sErrorID;
}MMC_READBOOLPARAMETER_OUT;
```

Parameters

lValue

Boolean parameters integer value

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-61 describes the function block for MMC_ReadBoolParameter as applied within the IEC 61131 programming.

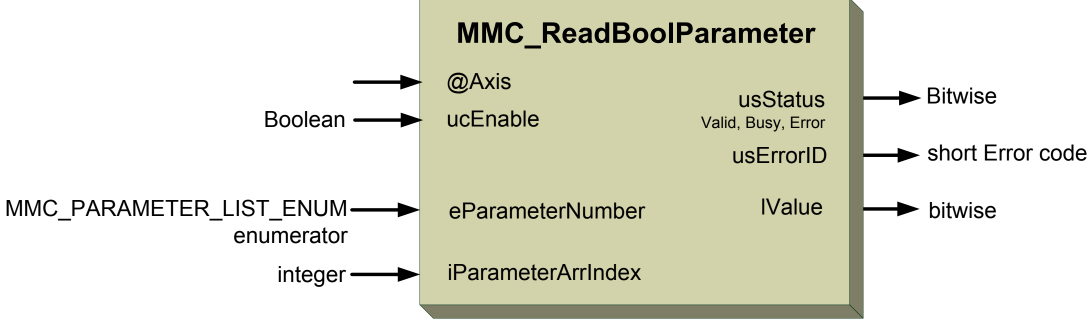


Figure 5-61: MMC_ReadBoolParameter function block

Another function in IEC that uses the same C function block base is MC_GetOpMode (Figure 5-62).

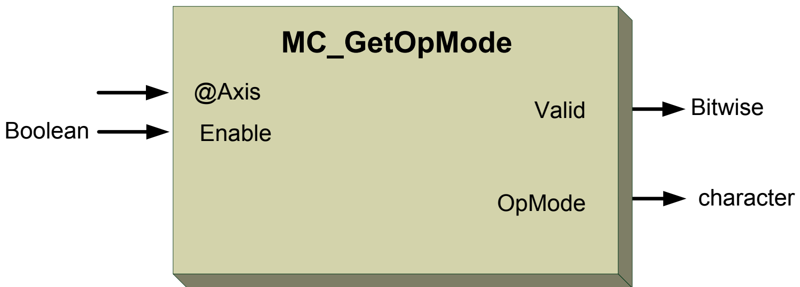


Figure 5-62: MC_GetOpMode function

5.9.14.2 Function Block Code Example

```
int rc ;
MMC_READBOOLPARAMETER_IN   stReadBoolParam_in;
MMC_READBOOLPARAMETER_OUT  stReadBoolParam_out;
//
// Inserting the structure parameters:
stReadBoolParam_in.eParameterNumber      = MMC_AXIS_MODE_PARAM; //MMC_PARAMETER_LIST_ENUM
enumerator
stReadBoolParam_in.iParameterArrIndex    = 0;                  //Array index parameter
stReadBoolParam_in.ucEnable              = 1;                  //Enabled
//
rc = MMC_ReadBoolParameter (hConn, iAxisRef, &stReadBoolParam_in, &stReadBoolParam_out) ;
printf("Boolean Parameter[%ld] ErrId[%d]\n", (long int)stReadBoolParam_out.lValue,
(short)stReadBoolParam_out.sErrorID);
if (rc != 0)
{
    HandleError() ;
}
```

5.9.15 MMC_GlobalReadBoolParameter

Returns the value of a vendor global parameter with datatype unsigned long or un signed int.

```
MMC_LIB_API int MMC_GlobalReadBoolParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_READBOOLPARAMETER_IN* pInParam,
OUT MMC_READBOOLPARAMETER_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_general_API.h

GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READBOOLPARAMETER_IN** input data structure using the MMC_GlobalReadBoolParameter function.

pOutParam

Points to the **MMC_READBOOLPARAMETER_OUT** output structure receiving information as a result of calling the MMC_GlobalReadBoolParameter function.

Remarks

None

Scope

All

MMC_READBOOLPARAMETER_IN Structure

```
typedef struct{
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_READBOOLPARAMETER_IN;
```

Parameters

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READBOOLPARAMETER_OUT Structure

```
typedef struct{
long lValue;
unsigned short usStatus;
short sErrorID;
}MMC_READBOOLPARAMETER_OUT;
```

Parameters

lValue

Boolean parameters integer value

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-63 describes the function block for MMC_GlobalReadBoolParameter as applied within the IEC 61131 programming.

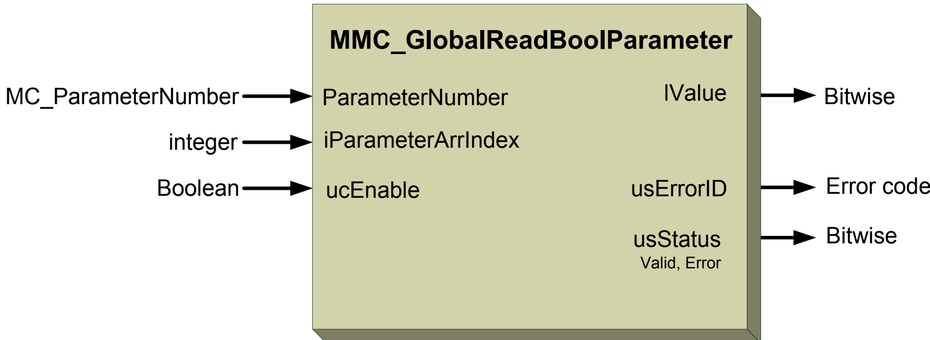


Figure 5-63: MMC_GlobalReadBoolParameter function block

5.9.15.2 Function Block Code Example

```
int rc ;
MMC_READBOOLPARAMETER_IN   stReadBoolParam_in;
MMC_READBOOLPARAMETER_OUT  stReadBoolParam_out;
//
// Inserting the structure parameters:
stReadBoolParam_in.eParameterNumber     = MMC_CYCLE_TIME_PARAM;//MMC_PARAMETER_LIST_ENUM
enumerator
stReadBoolParam_in.iParameterArrIndex   = 0;                  //Array index parameter
stReadBoolParam_in.ucEnable             = 1;                  //Enabled
//
rc = MMC_ReadBoolParameter (hConn, iAxisRef, &stReadBoolParam_in, &stReadBoolParam_out) ;
printf("Global Boolean Parameter[%ld] ErrId[%d]\n", (long int)stReadBoolParam_out.lValue,
(short)stReadBoolParam_out.sErrorID);
if (rc != 0)
{
    HandleError() ;
}
```

5.9.16 MMC_ReadDigitalInput(s)

Accesses the value of the input, referenced by the datatype MMC_Axis_Ref. It provides the Boolean value of the referenced input.

```
MMC_LIB_API int MMC_ReadDigitalInputCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READDIGITALINPUT_IN* pInParam,
OUT MMC_READDIGITALINPUT_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READDIGITALINPUT_IN** input data structure using the MMC_ReadDigitalInput function.

pOutParam

Points to the **MMC_READDIGITALINPUT_OUT** output structure receiving information as a result of calling the MMC_ReadDigitalInput function.

Remarks

For CAN, depending on the mode, the Input(s) may not be updated.

For EtherCAT, the actual Input(s) must be part of the Input Group configured.

This will depend whether the parameter was mapped via PDO mapping. Refer to the PDO3, 4 configuration mapping functions in section 12.7.

Scope

All

MMC_READDIGITALINPUT_IN Structure

```
typedef struct{
int iInputNumber;
unsigned char ucEnable;
}MMC_READDIGITALINPUT_IN;
```

Parameters

iInputNumber

Selects the input depending on the drive. Can be part of MMC_Axis_Ref, if only one single input is referenced. +ve integer value

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READDIGITALINPUT_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned char ucValue;
}MMC_READDIGITALINPUT_OUT;
```

Parameters

ucValue

Selected value of the input signal. +ve integer value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

MMC_ReadDigitalInputs

```
MMC_LIB_API int MMC_ReadDigitalInputsCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READDIGITALINPUTS_IN* pInParam,
OUT MMC_READDIGITALINPUTS_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READDIGITALINPUTS_IN** input data structure using the MMC_ReadDigitalInputs function.

pOutParam

Points to the **MMC_READDIGITALINPUTS_OUT** output structure receiving information as a result of calling the MMC_ReadDigitalInputs function.

MMC_READDIGITALINPUTS_IN Structure

```
typedef struct{
unsigned char ucEnable;
}MMC_READDIGITALINPUTS_IN;
```

Parameters

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READDIGITALINPUTS_OUT Structure

```
typedef struct{
unsigned long ulValue;
unsigned short usStatus;
short sErrorID;
}MMC_READDIGITALINPUTS_OUT;
```

Parameters

ulValue

Total value of all the inputs together. +ve 32 bit bitwise numeric value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-64 describes the function block for MMC_ReadDigitalInput as applied within the IEC 61131 programming.

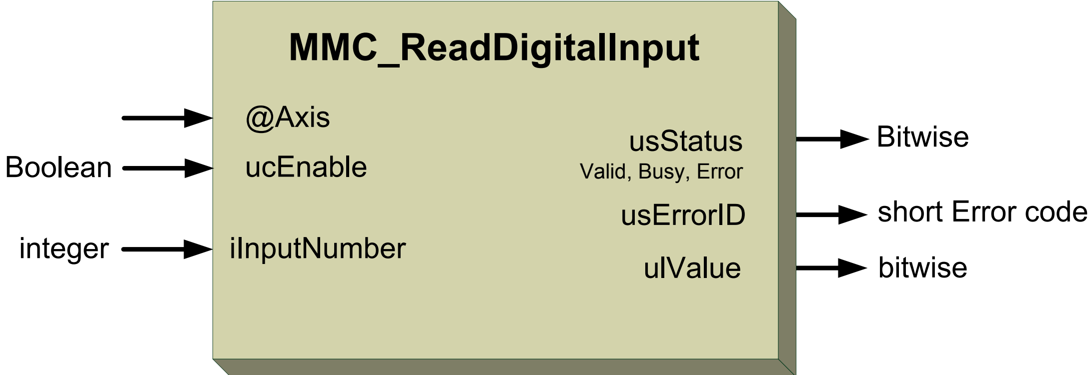


Figure 5-64: MMC_ReadDigitalInput function block

Figure 5-65 describes the function block for MMC_ReadDigitalInputs as applied within the IEC 61131 programming.

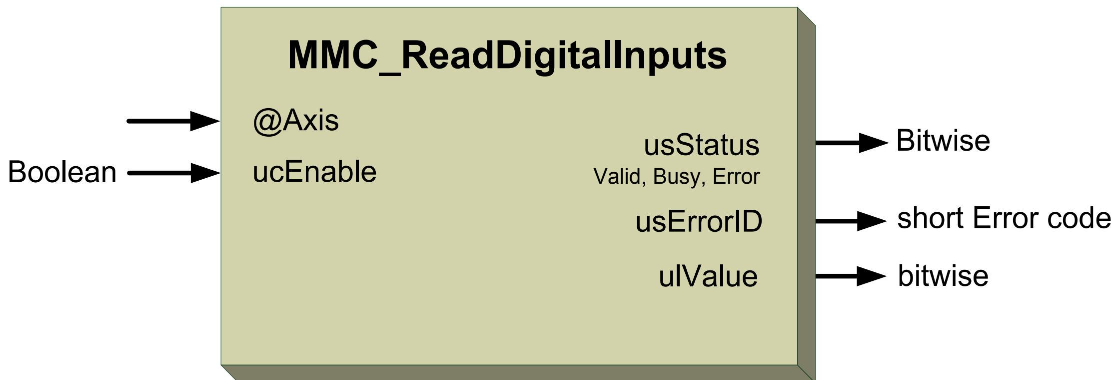


Figure 5-65: MMC_ReadDigitalInputs function block

5.9.16.2 Function Block Code Example

```
int rc;
MMC_READDIGITALINPUT_IN         stReadDigInput_in;
MMC_READDIGITALINPUT_OUT        stReadDigInput_out;
//
// Inserting the structure parameters:
stReadDigInput_in.iInputNumber = 16;    //Selects the input depending on the drive
stReadDigInput_in.ucEnable     = 1;     //Enabled
//
rc = MMC_ReadDigitalInputCmd (hConn, iAxisRef, &stReadDigInput_in, &stReadDigInput_out);
printf("Digital Input[%ld] ErrId[%d]\n", (long int)stReadDigInput_out.ucValue,
(short)stReadDigInput_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.9.17 MMC_ReadDigitalOutputs

Reads the actual digital outputs for the specific node.

```
MMC_LIB_API int MMC_ReadDigitalOutputs(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READDIGITALOUTPUT_IN* pInParam,
OUT MMC_READDIGITALOUTPUT_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READDIGITALOUTPUT_IN** input data structure using the MMC_ReadDigitalOutputs function.

pOutParam

Points to the **MMC_READDIGITALOUTPUT_OUT** output structure receiving information, as a result of calling the MMC_ReadDigitalOutputs function.

Remarks

Provides the value of the digital outputs, referenced by the datatype MC_OutputRef. It provides the Boolean value of the referenced outputs.

It is not guaranteed that the digital signal will be seen by the function block, as a short pulse on the digital output could be completed before the next function block cycle occurs.

Scope

All

MMC_READDIGITALOUTPUT_IN Structure

```
typedef struct MMC_READDIGITALOUTPUT_IN{
int iOutputNumber;
unsigned char ucEnable;
}MMC_READDIGITALOUTPUT_IN;
```

Parameters

iOutputNumber

Selects the output. Can be part of MC_OutputRef, if only one single output is referenced. +ve integer value.

Note: **This parameter is not in use at this time.**

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READDIGITALOUTPUT_OUT Structure

```
typedef struct MMC_READDIGITALOUTPUT_OUT{
unsigned short usStatus;
short sErrorID;
unsigned char ucValue;
}MMC_READDIGITALOUTPUT_OUT;
```

Parameters

ucValue

Selected value of the input signal. +ve integer value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-66 describes the function block for MMC_ReadDigitalOutput.

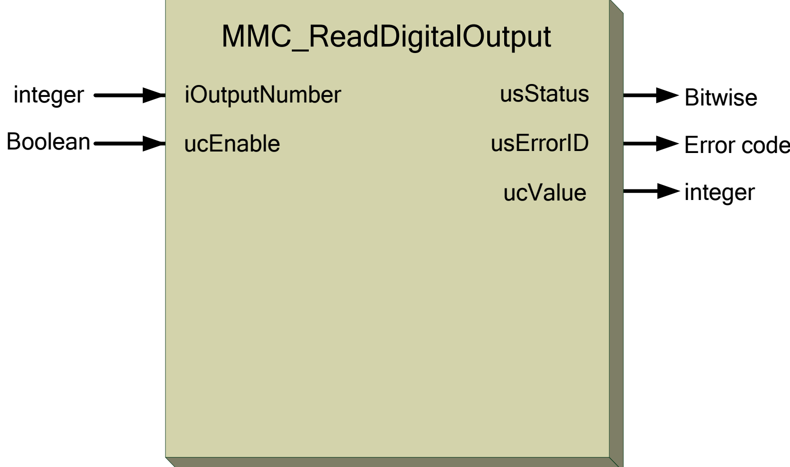


Figure 5-66: MMC_ReadDigitalOutput function block

5.9.17.2 Function Block Code Example

```
int rc;
MMC_READDIGITALOUTPUT_IN   stReadDigOutput_in;
MMC_READDIGITALOUTPUT_OUT  stReadDigOutput_out;
//
// Inserting the structure parameters:
stReadDigOutput_in.iOutputNumber     = 16;     //Selects the output depending on the drive
stReadDigOutput_in.ucEnable          = 1;      //Enabled
//
rc = MMC_ReadDigitalOutputs (hConn, iAxisRef, &stReadDigOutput_in, &stReadDigOutput_out);
printf("Digital Outputs[%ld] ErrId[%d]\n", (long int)stReadDigOutput_out.ucValue,
(short)stReadDigOutput_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.9.18 MMC_ReadDigitalOutputs32Bit

Reads the actual 32 bit digital output for the specific node

```
MMC_LIB_API int MMC_ReadDigitalOutputs32Bit(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READDIGITALOUTPUT32Bit_IN*pInParam,
OUT MMC_READDIGITALOUTPUT32Bit_OUT*pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READDIGITALOUTPUT32Bit_IN** input data structure using the MMC_ReadDigitalOutputs32Bit function.

pOutParam

Points to the **MMC_READDIGITALOUTPUT32Bit_OUT** output structure receiving information, as a result of calling the MMC_ReadDigitalOutputs32Bit function.

Remarks

Provides the value of the 32 bit digital outputs, referenced by the datatype MC_OutputRef. It provides the Boolean value of the referenced outputs.

The relevant function must be supported for digital outputs.

Scope

All

MMC_READDIGITALOUTPUT32Bit_IN Structure

```
typedef struct MMC_READDIGITALOUTPUT32Bit_IN{
int iOutputNumber;
unsigned char ucEnable;
}MMC_READDIGITALOUTPUT32Bit_IN;
```

Parameters

iOutputNumber

Selects the output. Can be part of MC_OutputRef, if only one single output is referenced. +ve integer value.

Note: **This parameter is not in use at this time.**

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READDIGITALOUTPUT32Bit_OUT Structure

```
typedef struct MMC_READDIGITALOUTPUT32Bit_OUT{
unsigned short usStatus;
unsigned short sErrorID;
unsigned long ulValue;
}MMC_READDIGITALOUTPUT32Bit_OUT;
```

Parameters

ulValue

Total value of all the inputs together. +ve 32 bit bitwise numeric value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-67 describes the function block for MMC_ReadDigitalOutputs32Bit

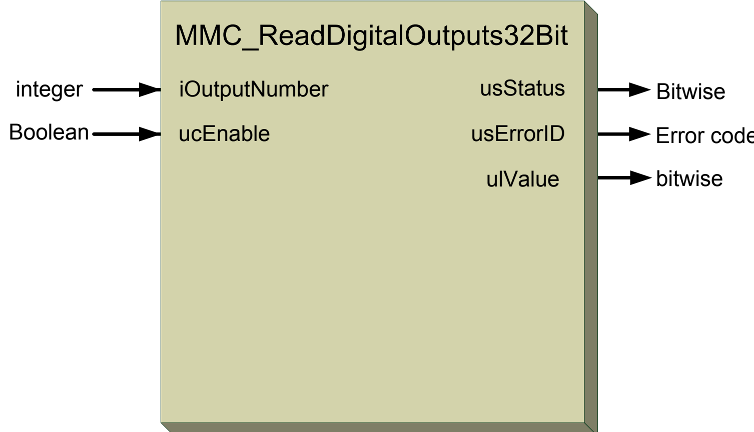


Figure 5-67: MMC_ReadDigitalOutputs32Bit function block

5.9.18.2 Function Block Code Example

```
int rc;
MMC_READDIGITALOUTPUT32Bit_IN       stReadDigOutput32Bit_in;
MMC_READDIGITALOUTPUT32Bit_OUT      stReadDigOutput32Bit_out;
//
// Inserting the structure parameters:
stReadDigOutput32Bit_in.iOutputNumber    = 16;    //Selects the output depending on the drive
stReadDigOutput32Bit_in.ucEnable         = 1;     //Enabled
//
rc = MMC_ReadDigitalOutputs32Bit (hConn, iAxisRef, &stReadDigOutput32Bit_in,
&stReadDigOutput32Bit_out);
printf("Digital Outputs 32Bit[%ld] ErrId[%d]\n", (long int)stReadDigOutput32Bit_out.ulValue,
(short)stReadDigOutput32Bit_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.9.19 MMC_ReadParameter

Returns the value of a vendor specific parameter.

```
MMC_LIB_API int MMC_ReadParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPARAMETER_IN* pInParam,
OUT MMC_READPARAMETER_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPARAMETER_IN** input data structure using the MMC_ReadParameter function.

pOutParam

Points to the **MMC_READPARAMETER_OUT** output structure receiving information as a result of calling the MMC_ReadParameter function.

Remarks

None

Scope

All

MMC_READPARAMETER_IN Structure

```
typedef struct{
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_READPARAMETER_IN;
```

Parameters

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READPARAMETER_OUT Structure

```
typedef struct{
double dbValue;
unsigned short usStatus;
short sErrorID;
}MMC_READPARAMETER_OUT;
```

Parameters

dbValue

Output of the specific parameter. Any Double value.

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-68 describes the function block for MMC_ReadParameter as applied within the IEC 61131 programming.

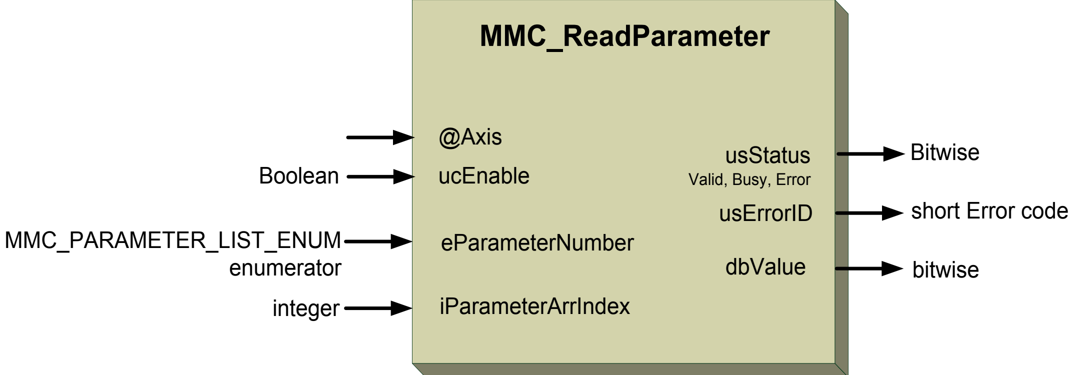


Figure 5-68: MMC_ReadParameter function block

5.9.19.2 Function Block Code Example

```
int rc;
MMC_READPARAMETER_IN   stReadParam_in;
MMC_READPARAMETER_OUT  stReadParam_out;
//
// Inserting the structure parameters:
stReadParam_in.ucEnable             = 1;                    //Enabled
stReadParam_in.iParameterArrIndex   = 0;                    //Parameter array index
stReadParam_in.eParameterNumber     = MMC_SET_POSITION_PARAM;  //Axis parameter value
//
rc = MMC_ReadParameter (hConn, iAxisRef, &stReadParam_in, &stReadParam_out);
printf("Read Parameter[%ld] ErrId[%d]\n", (long int)stReadParam_out.dbValue,
(short)stReadParam_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.9.20 MMC_GlobalReadParameter

Returns the value of a vendor global parameter.

```
MMC_LIB_API int MMC_GlobalReadParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_READPARAMETER_IN* pInParam,
OUT MMC_READPARAMETER_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_general_API.h GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoGlobal	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_READPARAMETER_IN** input data structure using the MMC_GlobalReadParameter function.

pOutParam

Points to the **MMC_READPARAMETER_OUT** output structure receiving information as a result of calling the MMC_GlobalReadParameter function.

Remarks

None

Scope

All

MMC_READPARAMETER_IN Structure

```
typedef struct{
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_READPARAMETER_IN;
```

Parameters

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READPARAMETER_OUT Structure

```
typedef struct{
double dbValue;
unsigned short usStatus;
short sErrorID;
}MMC_READPARAMETER_OUT;
```

Parameters

dbValue

Output of specific parameter. Any Double value.

usStatus

Bitwise returned command status with the following values:

Aborted, Done, CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-69 describes the function block for MMC_GlobalReadParameter as applied within the IEC 61131 programming.

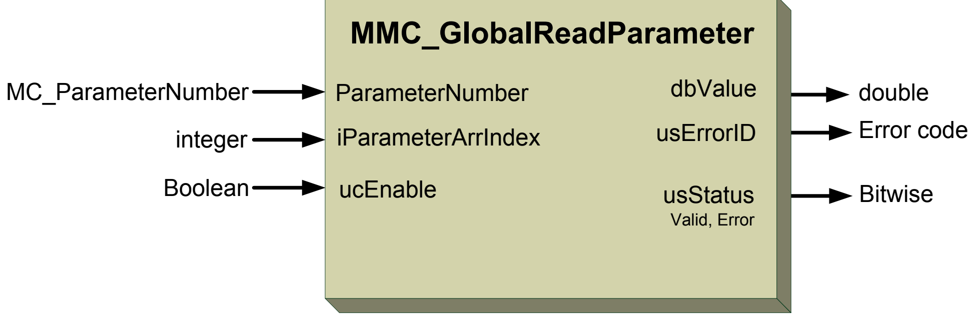


Figure 5-69: MMC_GlobalReadParameter function block

5.9.20.2 Function Block Code Example

```
int rc;
MMC_READPARAMETER_IN  stReadParam_in;
MMC_READPARAMETER_OUT stReadParam_out;
//
// Inserting the structure parameters:
stReadParam_in.ucEnable            = 1;                  //Enabled
stReadParam_in.iParameterArrIndex  = 0;                  //Parameter array index
stReadParam_in.eParameterNumber    = MMC_CYCLE_TIME_PARAM;//Axis parameter value
//
rc = MMC_ReadParameter (hConn, iAxisRef, &stReadParam_in, &stReadParam_out);
printf("Global Read Parameter[%ld] ErrId[%d]\n", (long int)stReadParam_out.dbValue,
(short)stReadParam_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.9.21 MMC_ReadStatus

Returns details of the state diagram status for the selected axis.

```
MMC_LIB_API int MMC_ReadStatusCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READSTATUS_IN* pInParam,
OUT MMC_READSTATUS_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READSTATUS_IN** input data structure using the MMC_ReadStatus function.

pOutParam

Points to the **MMC_READSTATUS_OUT** output structure receiving information as a result of calling the MMC_ReadStatus function.

Remarks

None

Scope

All

MMC_READSTATUS_IN Structure

```
typedef struct{
unsigned int  uiHndlr;
unsigned char ucEnable;
}MMC_READSTATUS_IN;
```

Parameters

uiHndlr

Requested axis handle integer. Any +ve integer number

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READSTATUS_OUT Structure

```
typedef struct{
unsigned long ulState;
unsigned short usStatus;
short sErrorID;
unsigned short usAxisErrorID;
unsigned short usStatusWord;
}MMC_READSTATUS_OUT;
```

Parameters

ulState

Returned command status. Refer to the sub-section **5.4 Axis Status** for the appropriate parameter value received and its explanation. Bitwise value returned.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

usAxisErrorID

Returns the axis error bitwise ID defined by the following enumerators. Bitwise ID error code:

ID	Enumerator
0x1	MMC_ERR_TYPE_FAULT_BIT
0x2	MMC_ERR_TYPE_HEARTBEAT
0x4	MMC_ERR_TYPE_EMERGENCY
0x8	MMC_ERR_TYPE_COMM
0x10	MMC_ERR_TYPE_CFG_FILE

usStatusWord

Drive Status text. Any text characters.

Figure 5-70 describes the function block for MMC_ReadStatus as applied within the IEC 61131 programming.

The parameters usAxisErrorID and usStatusWord are currently not supported by IEC.

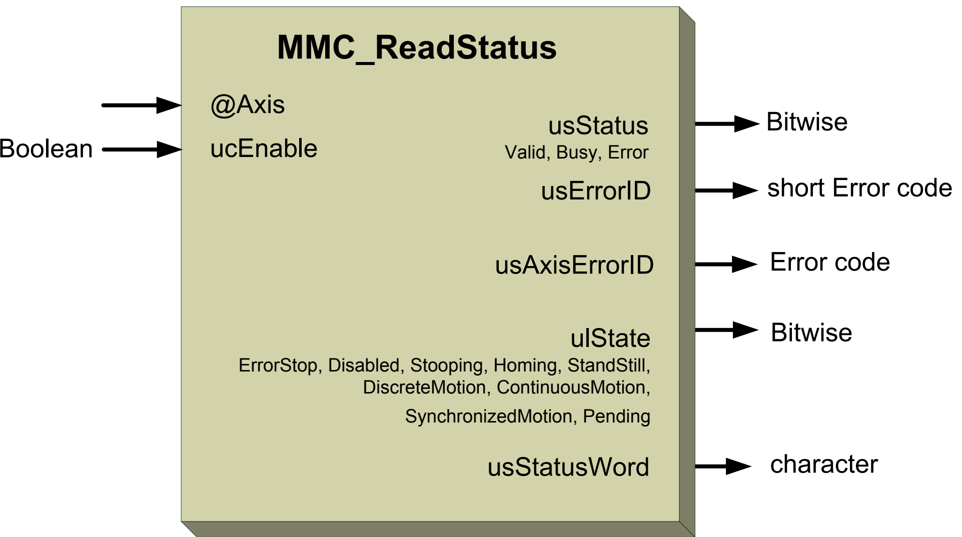


Figure 5-70: MMC_ReadStatus function block

5.9.21.2 Function Block Code Example

```
int rc ;
MMC_READSTATUS_IN      stReadStatus_in;
MMC_READSTATUS_OUT     stReadStatus_out;
//
// Inserting the structure parameters:
stReadStatus_in.uiHndlr     = 251;    //Requested axis handle integer
stReadStatus_in.ucEnable    = 1;      //Enabled
//
rc = MMC_ReadStatusCmd (hConn, iAxisRef, &stReadStatus_in, &stReadStatus_out);
printf("Read Status[%ld] ErrId[%d]\n", stReadStatus_out.ulState,
(short)stReadStatus_out.sErrorID);

if (rc != 0)
{
    HandleError() ;
}
```

5.9.22 MMC_Reset

Produces the transition from the state ErrorStop to StandStill or Disabled by resetting all internal axis-related errors, and returns immediately.

```
MMC_LIB_API int MMC_Reset(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_RESET_IN* pInParam,
OUT MMC_RESET_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_RESET_IN** input data structure using the MMC_Reset function.

pOutParam

Points to the **MMC_RESET_OUT** output structure receiving information as a result of calling the MMC_Reset function.

Remarks

However, it does not affect the output of the function block instances.

This function must be called in order to exit the ErrorStop state.

Scope

All

MMC_RESET_IN Structure

```
typedef struct{
unsigned char ucExecute;
}MMC_RESET_IN;
```

Parameters

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_RESET_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_RESET_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-71 describes the function block for MMC_Reset as applied within the IEC 61131 programming.

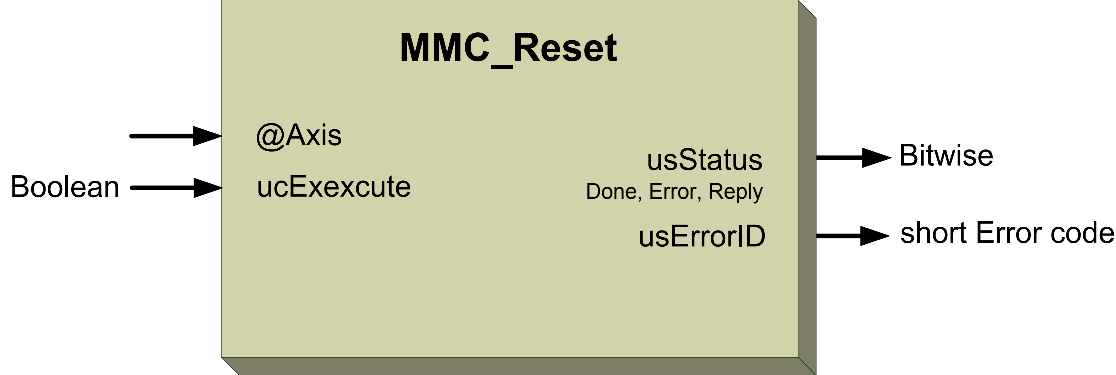


Figure 5-71: MMC_Reset function block

5.9.22.2 Function Block Code Example

```
int rc;
MMC_RESET_IN      stReset_in;
MMC_RESET_OUT     stReset_out;
//
// Inserting the structure parameters:
stReset_in.ucExecute   = 1;     // Start the execution command

//
rc = MMC_Reset (hConn, iAxisRef, &stReset_in, &stReset_out);
if (rc != 0)
{
    HandleError();
}
```

5.9.23 MMC_ResetAsync

Produces the transition from the state ErrorStop to StandStill or Disabled by resetting all internal axis-related errors. This function waits until the procedure is fully completed. An event is then sent once the reset is over.

```
MMC_LIB_API int MMC_Reset(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_RESET_IN* pInParam,
OUT MMC_RESET_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_RESET_IN** input data structure using the MMC_Reset function.

pOutParam

Points to the **MMC_RESET_OUT** output structure receiving information as a result of calling the MMC_Reset function.

Remarks

However, it does not affect the output of the function block instances.

This function must be called in order to exit the ErrorStop state.

Scope

All

MMC_RESET_IN Structure

```
typedef struct{
unsigned char ucExecute;
}MMC_RESET_IN;
```

Parameters

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_RESET_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_RESET_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-72 describes the function block for MMC_ResetAsync as applied within the IEC 61131 programming.

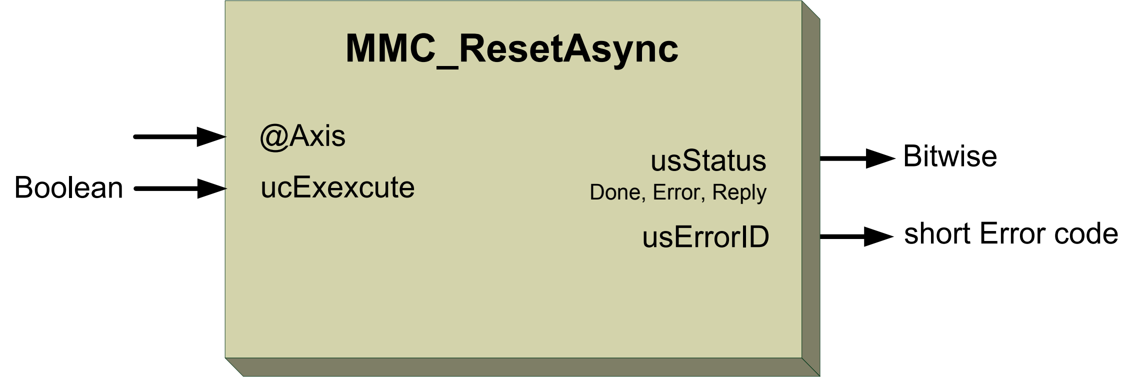


Figure 5-72: MMC_ResetAsync function block

5.9.24 MMC_SetOverride

Sets the values of override for the whole axis, including all functions that are operating on that axis.

```
MMC_LIB_API int MMC_SetOverrideCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SETOVERRIDE_IN* pInParam,
OUT MMC_SETOVERRIDE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source **Function**

GMAS\includes\MMC_PLCopen_single_API.h

Structures

GMAS\includes\MMC_PLCopen_group_API.h

IEC

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SETOVERRIDE_IN** input data structure using the MMC_SetOverride function.

pOutParam

Points to the **MMC_SETOVERRIDE_OUT** output structure receiving information as a result of calling the MMC_SetOverride function.

Remarks

The override parameters are applied to the commanded velocity, acceleration, deceleration, and jerk of the relevant Motion function block.

MMC_SetOverride does not influence the state diagram of the axis. The following should be noted:

- The Input *fAccFactor* acts on positive and negative acceleration (deceleration).
- This function block sets the override factor, which is valid until a new override is set.
- The default values of the override factor are 1.0. The value of the overrides can be between 0.0 and 1.0. The behavior of values > 1.0 is vendor specific. Values < 0.0 are not allowed. The value 0.0 is not allowed for *fAccFactor* and *fJerkFactor*.
- The value 0.0 set to the *fVelFactor* stops the axis without bringing it to the state Standstill.

- Override does not act on slave axes. (Axes in the state synchronized motion).
- *fVelFactor* can be changed at any time and acts directly on the ongoing motion.
- If in Discrete motion, reducing the fAccFactor and/or fJerkFactor can lead to a position overshoot – a possible cause of damage.

Activating this function block on an axis that is under control of MC_PositionProfile, MC_VelocityProfile, or MC_AccelerationProfile, is vendor specific.

Scope

All

MMC_SETOVERRIDE_IN Structure

```
typedef struct{
float fVelFactor;
float fAccFactor;
float fJerkFactor;
unsigned short usUpdateVelFactorIdx;
unsigned char ucEnable;
}MMC_SETOVERRIDE_IN;
```

Parameters

fVelFactor

New override factor for the velocity. Any +ve float value between [0 – 1].

fAccFactor

New override factor for the acceleration/deceleration. ACC/Jerk Factors are NOT supported at this time. For future compatibility, enter "1" in the function call.

fJerkFactor

New override factor for the jerk. ACC/Jerk Factors are NOT supported at this time. For future compatibility, enter "1" in the function call.

usUpdateVelFactorIdx

Index of changed velocity factor. Vendor defined. The default is 0. Has integer values of 0 - 2

This variable is not in use at this moment.

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_SETOVERRIDE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
} MMC_SETOVERRIDE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-73 describes the function block for MMC_SetOverride as applied within the IEC 61131 programming.

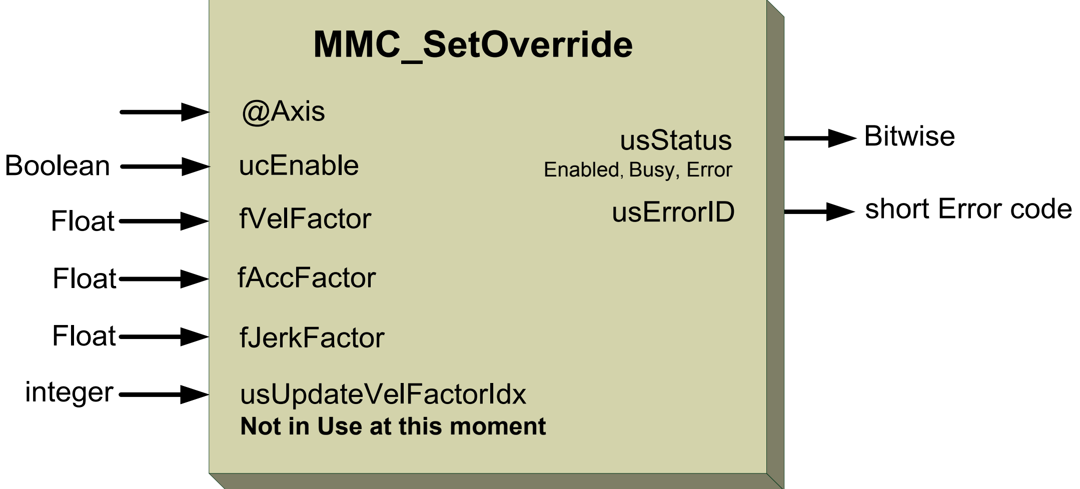


Figure 5-73: MMC_SetOverride function block

5.9.24.2 Function Block Code Example

```
int rc;
MMC_SETOVERRIDE_IN          stSetOverride_in;
MMC_SETOVERRIDE_OUT         stSetOverride_out;
//
// Inserting the structure parameters:
stSetOverride_in.fAccFactor              = 0.5; // New override factor for the
acceleration/deceleration
stSetOverride_in.fJerkFactor             = 0.4; // New override factor for the jerk
stSetOverride_in.usUpdateVelFactorIdx    = 0;   // Index of changed velocity factor
stSetOverride_in.fVelFactor              = 0.8; // New override factor for the velocity
stSetOverride_in.ucEnable                = 1;   // Enabled
//
rc = MMC_SetOverrideCmd (hConn, iAxisRef, &stSetOverride_in, &stSetOverride_out);
if (rc != 0)
{
    HandleError();
}
```

5.9.25 MMC_SetPosition

Sends the Set Position command to the Maestro for ac specific axis.

```
MMC_LIB_API int MMC_SetPositionCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SETPOSITION_IN* pInParam,
OUT MMC_SETPOSITION_OUT* pOutParam
);
```

Motion Mode NC – Not Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SETPOSITION_IN** input data structure using the MMC_SetPosition function.

pOutParam

Points to the **MMC_SETPOSITION_OUT** output structure receiving information, as a result of calling the MMC_SetPosition function.

Remarks

None

Scope

All

MMC_SETPOSITION_IN Structure

```
typedef struct{
double dbPosition;
double dbModulus;
unsigned char ucPosMode;
unsigned char ucExecute;
}MMC_SETPOSITION_IN;
```

Parameters

dbPosition

Target position for the motion of the axis when conditions are met. Any -ve or +ve double values in technical unit [u].

dbModulus

The relative modulus of the axis. Any -ve or +ve double values in technical unit [u].

ucPosMode

Boolean values for the position mode can be absolute mode = 0, or relative mode = 1

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_SETPOSITION_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETPOSITION_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-74 describes the function block for MMC_SetPosition as applied within the IEC 61131 programming.

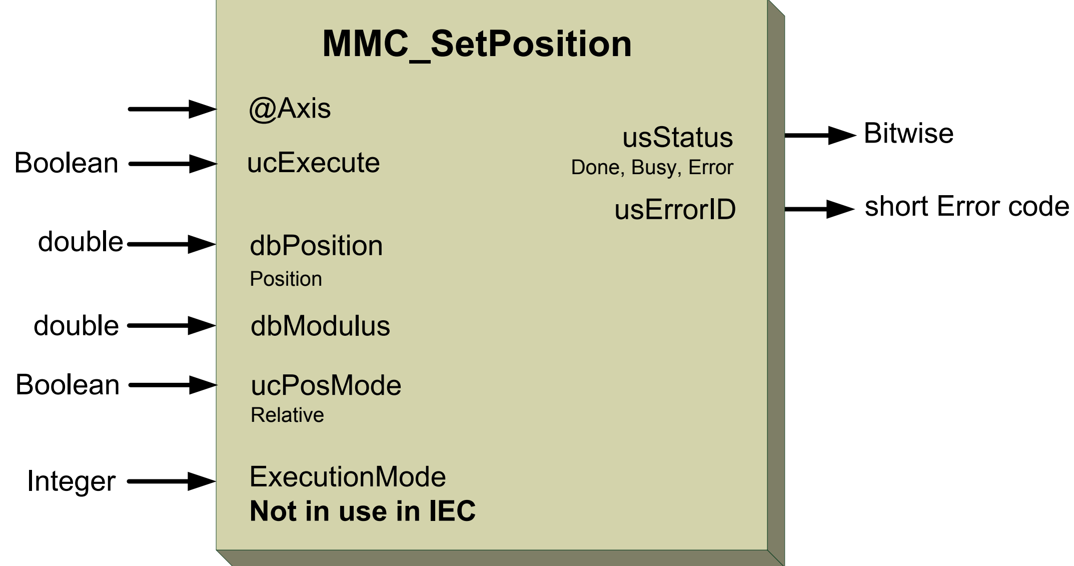


Figure 5-74: MMC_SetPosition function block

5.9.26 MMC_TouchProbeEnable

Enables the touch probe to record an axis position at a trigger event.

```
MMC_LIB_API int MMC_TouchProbeEnableCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_TOUCHPROBEENABLE_IN*pInParam,
OUT MMC_TOUCHPROBEENABLE_OUT*pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_TOUCHPROBEENABLE_IN** input data structure using the MMC_TouchProbeEnable function.

pOutParam

Points to the **MMC_TOUCHPROBEENABLE_OUT** output structure receiving information as a result of calling the MMC_TouchProbeEnable function.

Remarks

This function block is intended for single shot operation, that is, the first event after the rising edge at Execute is valid for recording only. Possible following events are ignored

One function block instance should represent exactly one probing command

In case of multiple instances on the same probe and axis, the elements of TRIGGER_REF should be extended with TouchProbeID - Identification of a unique probing command – this can be linked to MC_AbortTrigger.

Scope

Supported for Gold Servo Drives only.

User notification when touch probe is over via Callback and event.

MMC_TOUCHPROBEENABLE_IN Structure

```
typedef struct mmc_touchprobeenable_in{
unsigned char ucExecute;
unsigned char ucTriggerType;
}MMC_TOUCHPROBEENABLE_IN;
```

Parameters

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

ucTriggerType

Trigger enables the touch proble. Reference to the trigger signal source, where the trigger input may be specified by the AXIS_REF. Has the following enumerator values:

eMMC_TOUCHPROBE_POS_EDGE = 0,

eMMC_TOUCHPROBE_NEG_EDGE

MMC_TOUCHPROBEENABLE_OUT Structure

```
typedef struct mmc_touchprobeenable_out{
unsigned short usStatus;
short sErrorID;
}MMC_TOUCHPROBEENABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-75 describes the function block for MMC_TouchProbeEnable as applied within the IEC 61131 programming.

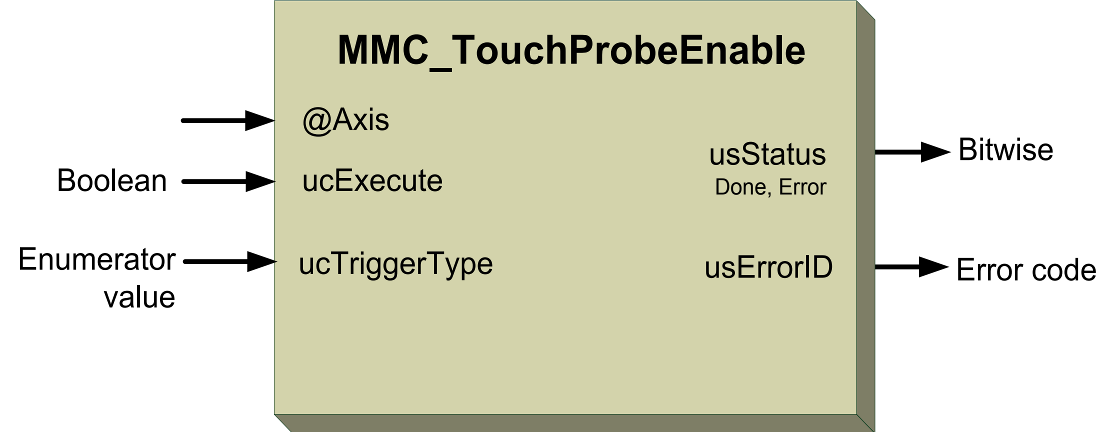


Figure 5-75: MMC_TouchProbeEnable function block

5.9.27 MMC_TouchProbeDisable

Disables the touch probe to record an axis position at a trigger event.

```
MMC_LIB_API int MMC_TouchProbeDisableCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_TOUCHPROBEDISABLE_IN* pInParam,
OUT MMC_TOUCHPROBEDISABLE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_TOUCHPROBEDISABLE_IN** input data structure using the MMC_TouchProbeDisable function.

pOutParam

Points to the **MMC_TOUCHPROBEDISABLE_OUT** output structure receiving information as a result of calling the MMC_TouchProbeDisable function.

Remarks

This function block is intended for single shot operation, that is, the first event after the rising edge at Execute is valid for recording only. Possible following events are ignored.

One function block instance should represent exactly one probing command.

In case of multiple instances on the same probe and axis, the elements of TRIGGER_REF should be extended with TouchProbeID - Identification of a unique probing command – this can be linked to MC_AbortTrigger.

Scope

Supported for Gold Servo Drives only.

User notification when touch probe is over via Callback and event.

MMC_TOUCHPROBEDISABLE_IN Structure

```
typedef struct mmc_touchprobedisable_in{
unsigned char ucExecute;
}MMC_TOUCHPROBEDISABLE_IN;
```

Parameters

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_TOUCHPROBEDISABLE_OUT Structure

```
typedef struct mmc_touchprobedisable_out{
unsigned short usStatus;
short sErrorID;
}MMC_TOUCHPROBEDISABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-76 describes the function block for MMC_TouchProbeDisable as applied within the IEC 61131 programming.

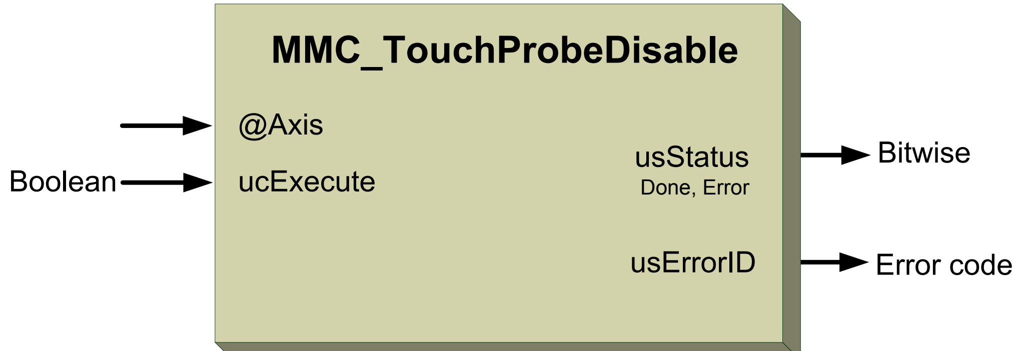


Figure 5-76: MMC_TouchProbeDisable function block

5.9.28 MMC_WriteBoolParameter

Modifies the value of a vendor specific parameter of type BOOL.

```
MMC_LIB_API int MMC_WriteBoolParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEBOOLPARAMETER_IN* pInParam,
OUT MMC_WRITEBOOLPARAMETER_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEBOOLPARAMETER_IN** input data structure using the MMC_WriteBoolParameter function.

pOutParam

Points to the **MMC_WRITEBOOLPARAMETER_OUT** output structure receiving information, as a result of calling the MMC_WriteBoolParameter function.

Remarks

The parameters are defined in the table in section 5.9.18 above for the function block MMC_ReadParameter.

Scope

All

MMC_WRITEBOOLPARAMETER_IN Structure

```
typedef struct{
long lValue;
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_WRITEBOOLPARAMETER_IN;
```

Parameters

lValue

Input value. Any integer.

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_WRITEBOOLPARAMETER_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEBOOLPARAMETER_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-77 describes the function block for MMC_WriteBoolParameter as applied within the IEC 61131 programming.

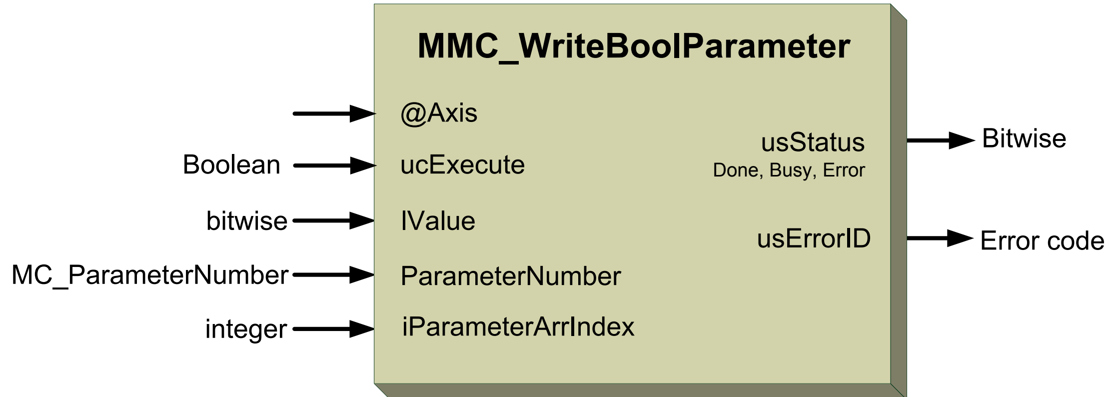


Figure 5-77: MMC_WriteBoolParameter function block

5.9.28.2 Function Block Code Example

```
int rc;
MMC_WRITEBOOLPARAMETER_IN       stWriteBoolParam_in;
MMC_WRITEBOOLPARAMETER_OUT      stWriteBoolParam_out;
//
// Inserting the structure parameters:
stWriteBoolParam_in.lValue          = 1;              // Axis status value of the axis 
parameter
stWriteBoolParam_in.eParameterNumber     = MMC_AXIS_STATE_PARAM;// Enumerator value of the 
parameter
stWriteBoolParam_in.iParameterArrIndex   = 2;              // Array index parameter
stWriteBoolParam_in.ucEnable             = 1;              // Enabled
//
rc = MMC_WriteBoolParameter (hConn, iAxisRef, &stWriteBoolParam_in, &stWriteBoolParam_out);
if (rc != 0)
{
    HandleError();
}
```

5.9.29 MMC_GlobalWriteBoolParameter

Modifies the value of a vendor global parameter of type BOOL.

```
MMC_LIB_API int MMC_GlobalWriteBoolParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_WRITEBOOLPARAMETER_IN* pInParam,
OUT MMC_WRITEBOOLPARAMETER_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_WRITEBOOLPARAMETER_IN** input data structure using the MMC_GlobalWriteBoolParameter function.

pOutParam

Points to the **MMC_WRITEBOOLPARAMETER_OUT** output structure receiving information, as a result of calling the MMC_GlobalWriteBoolParameter function.

Remarks

The parameters are defined in the table in section 5.9.18 above for the function block MMC_ReadParameter.

Scope

All

MMC_WRITEBOOLPARAMETER_IN Structure

```
typedef struct{
long lValue;
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_WRITEBOOLPARAMETER_IN;
```

Parameters

lValue

Input parameter. Any integer value.

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_WRITEBOOLPARAMETER_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEBOOLPARAMETER_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-78 describes the function block for MMC_GlobalWriteBoolParameter as applied within the IEC 61131 programming.

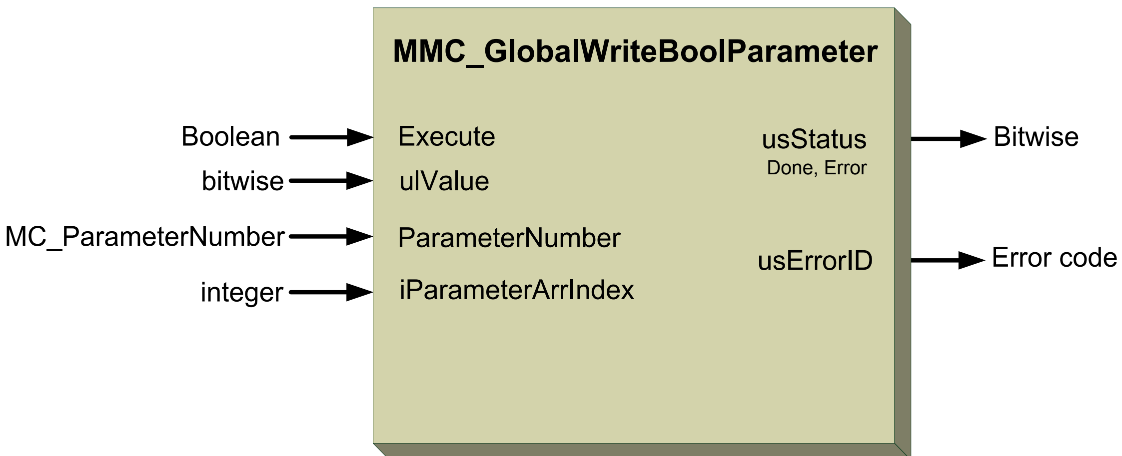


Figure 5-78: MMC_GlobalWriteBoolParameter function block

5.9.29.2 Function Block Code Example

```
int rc;
MMC_WRITEBOOLPARAMETER_IN       stWriteBoolParam_in;
MMC_WRITEBOOLPARAMETER_OUT      stWriteBoolParam_out;
//
// Inserting the structure parameters:
stWriteBoolParam_in.lValue           = 1;              // Axis status value of the axis
parameter
stWriteBoolParam_in.eParameterNumber     = MMC_MOVEMENT_TYPE_PARAM;// Enumerator value of the
parameter
stWriteBoolParam_in.iParameterArrIndex   = 2;              // Array index parameter
stWriteBoolParam_in.ucEnable             = 1;              // Enabled
//
rc = MMC_WriteBoolParameter (hConn, iAxisRef, &stWriteBoolParam_in, &stWriteBoolParam_out);
if (rc != 0)
{
    HandleError();
}
```

5.9.30 MMC_WriteDigitalOutputs

Writes a value to the digital output referenced by the single argument Output (with rising edge of Execute).

```
MMC_LIB_API int MMC_WriteDigitalOutputs(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEDIGITALOUTPUT_IN* pInParam,
OUT MMC_WRITEDIGITALOUTPUT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEDIGITALOUTPUT_IN** input data structure using the MMC_WriteDigitalOutputs function.

pOutParam

Points to the **MMC_WRITEDIGITALOUTPUT_OUT** output structure receiving information, as a result of calling the MMC_WriteDigitalOutputs function.

Remarks

This function must be supported for digital output.

Scope

All

MMC_WRITEDIGITALOUTPUT_IN Structure

```
typedef struct MMC_WRITEDIGITALOUTPUT_IN{
int iOutputNumber;
unsigned char ucEnable;
unsigned char ucValue;
}MMC_WRITEDIGITALOUTPUT_IN;
```

Parameters

iOutputNumber

Selects the output. Can be part of MC_OutputRef, if only one single input is referenced. +ve integer value.

Note: **This parameter is not in use at this time.**

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

ucValue

Whether the selected value input signal is True. Boolean values of TRUE/FALSE.

MMC_WRITEDIGITALOUTPUT_OUT Structure

```
typedef struct MMC_WRITEDIGITALOUTPUT_OUT{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEDIGITALOUTPUT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-79 describes the function block for MMC_WriteigitalOutput

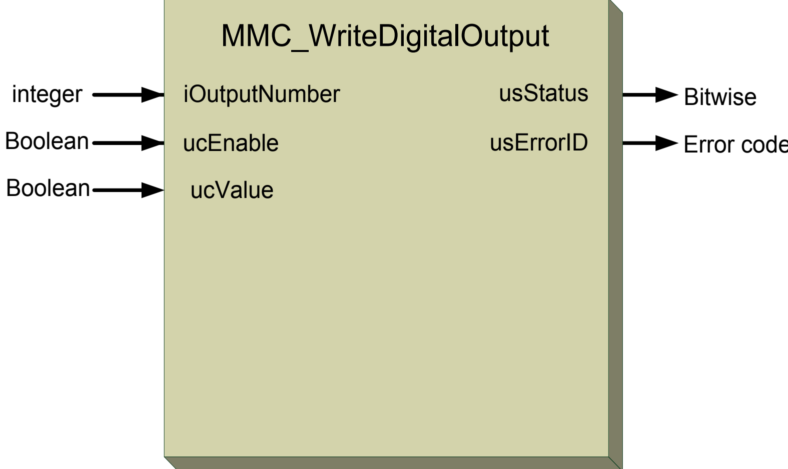


Figure 5-79: MMC_WriteDigitalOutput function block

5.9.30.2 Function Block Code Example

```
int rc;
MMC_WRITEDIGITALOUTPUT_IN       stWriteDigOutput_in;
MMC_WRITEDIGITALOUTPUT_OUT      stWriteDigOutput_out;
//
// Inserting the structure parameters:
stWriteDigOutput_in.iOutputNumber    = 16;    //Selects the output depending on the drive
stWriteDigOutput_in.ucEnable         = 1;     //Enabled
//
rc = MMC_WriteDigitalOutputs (hConn, iAxisRef, &stWriteDigOutput_in, &stWriteDigOutput_out);
if (rc != 0)
{
    HandleError();
}
```

5.9.31 MMC_WriteDigitalOutputs32Bit

Writes a value to the 32-bit digital output referenced by the single argument Output (with rising edge of Execute).

```
MMC_LIB_API int MMC_WriteDigitalOutputs32Bit(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEDIGITALOUTPUT32Bit_IN* pInParam,
OUT MMC_WRITEDIGITALOUTPUT32Bit_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEDIGITALOUTPUT32Bit_IN** input data structure using the MMC_WriteDigitalOutputs32Bit function.

pOutParam

Points to the **MMC_WRITEDIGITALOUTPUT32Bit_OUT** output structure receiving information, as a result of calling the MMC_WriteDigitalOutputs32Bit function.

Remarks

This function must be supported for digital output.

Scope

All

MMC_WRITEDIGITALOUTPUT32Bit_IN Structure

```
typedef struct MMC_WRITEDIGITALOUTPUT32BIT_IN{
int iOutputNumber;
unsigned long ulValue;
unsigned char ucEnable;
}MMC_WRITEDIGITALOUTPUT32Bit_IN;
```

Parameters

iOutputNumber

Selects the output. Can be part of MC_OutputRef, if only one single input is referenced. +ve integer value.

Note: **This parameter is not in use at this time.**

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

ulValue

Total value of all the inputs together. 32 bit bitwise +ve numeric value.

MMC_WRITEDIGITALOUTPUT32Bit_OUT Structure

```
typedef struct MMC_WRITEDIGITALOUTPUT32BIT_OUT{
unsigned short usStatus;
unsigned short sErrorID;
}MMC_WRITEDIGITALOUTPUT32Bit_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-80 describes the function block for MMC_WriteDigitalOutputs32Bit s applied within the IEC 61131 programming.

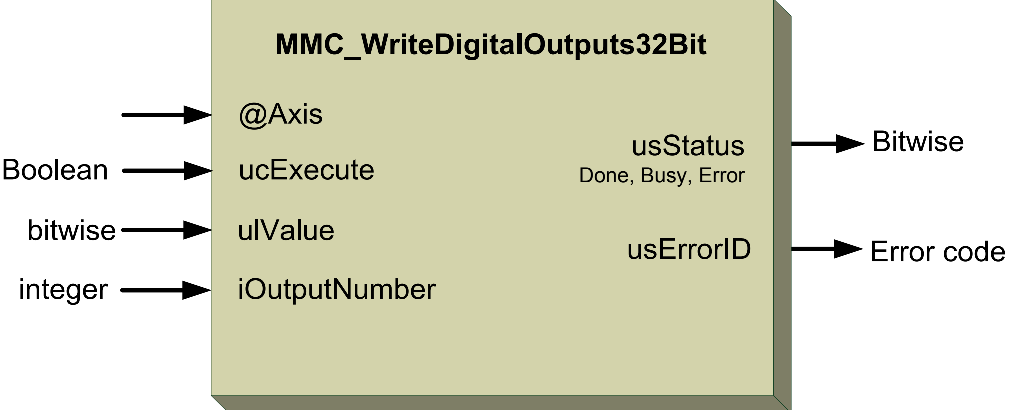


Figure 5-80: MMC_WriteDigitalOutputs32Bit function block

5.9.31.2 Function Block Code Example

```
int rc;
MMC_WRITEDIGITALOUTPUT32Bit_IN      stWriteDigOutput32Bit_in;
MMC_WRITEDIGITALOUTPUT32Bit_OUT     stWriteDigOutput32Bit_out;
//
// Inserting the structure parameters:
stWriteDigOutput32Bit_in.iOutputNumber   = 16;    //Selects the output depending on the drive
stWriteDigOutput32Bit_in.ucEnable        = 1;     //Enabled
//
rc = MMC_WriteDigitalOutputs32Bit (hConn, iAxisRef, &stWriteDigOutput32Bit_in,
&stWriteDigOutput32Bit_out);
if (rc != 0)
{
    HandleError();
}
```

5.9.32 MMC_WriteParameter

Modifies the value of a vendor specific parameter.

```
MMC_LIB_API int MMC_WriteParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPARAMETER_IN* pInParam,
OUT MMC_WRITEPARAMETER_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_PLCopen_single_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPARAMETER_IN** input data structure using the MMC_WriteParameter function.

pOutParam

Points to the **MMC_WRITEPARAMETER_OUT** output structure receiving information as a result of calling the MMC_WriteParameter function.

Remarks

Refer to the full explanation in section **5.4.1 Blended Behavior Mechanism on page 178**.

Scope

All

MMC_WRITEPARAMETER_IN Structure

```
typedef struct{
double dbValue;
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_WRITEPARAMETER_IN;
```

Parameters

dbValue

Parameter value. Any double value.

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_WRITEPARAMETER_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEPARAMETER_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-81 describes the function block for MMC_WriteParameter as applied within the IEC 61131 programming.

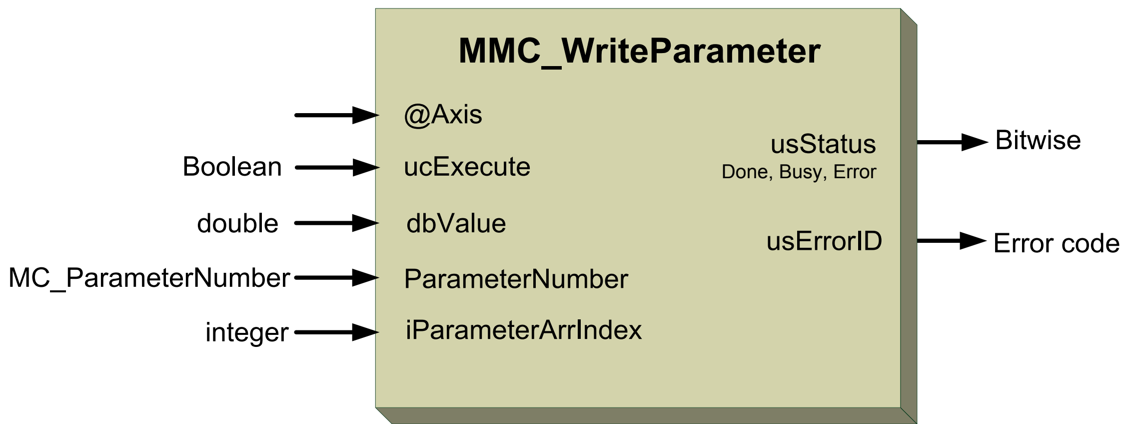


Figure 5-81: MMC_WriteParameter function block

5.9.32.2 Function Block Code Example

```
int rc;
MMC_WRITEPARAMETER_IN       stWriteParam_in;
MMC_WRITEPARAMETER_OUT      stWriteParam_out;
//
// Inserting the structure parameters:
stWriteParam_in.dbValue             = 1000;              //Parameter value
stWriteParam_in.ucEnable            = 1;                 //Enabled
stWriteParam_in.iParameterArrIndex  = 1;                 //Parameter array index
stWriteParam_in.eParameterNumber    = MMC_SET_POSITION_PARAM;//Axis parameter value
//
rc = MMC_WriteParameter (hConn, iAxisRef, &stWriteParam_in, &stWriteParam_out);
if (rc != 0)
{
    HandleError();
}
```

5.9.33 MMC_GlobalWriteParameter

Modifies the value of a vendor global parameter.

```
MMC_LIB_API int MMC_GlobalWriteParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_WRITEPARAMETER_IN* pInParam,
OUT MMC_WRITEPARAMETER_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_general_API.h

GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_WRITEPARAMETER_IN** input data structure using the MMC_GlobalWriteParameter function.

pOutParam

Points to the **MMC_WRITEPARAMETER_OUT** output structure receiving information as a result of calling the MMC_GlobalWriteParameter function.

Remarks

For a detailed explanation of this function, refer to the section **5.4.1 Blended Behavior Mechanism on page 178**.

Scope

All

MMC_WRITEPARAMETER_IN Structure

```
typedef struct{
double dbValue;
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_WRITEPARAMETER_IN;
```

Parameters

dbValue

Parameter value. Any double (2 bytes)value.

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_WRITEPARAMETER_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEPARAMETER_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-82 describes the function block for MMC_GlobalWriteParameter as applied within the IEC 61131 programming.

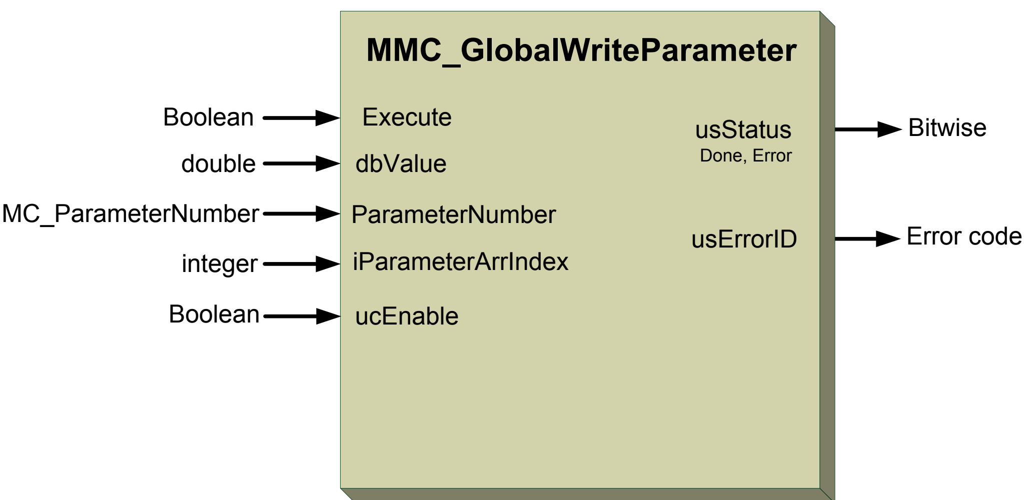


Figure 5-82: MMC_GlobalWriteParameter function block

5.9.33.2 Function Block Code Example

```
int rc;
MMC_WRITEPARAMETER_IN      stWriteParam_in;
MMC_WRITEPARAMETER_OUT     stWriteParam_out;
//
// Inserting the structure parameters:
stWriteParam_in.dbValue            = 1000;             //Parameter value
stWriteParam_in.ucEnable           = 1;                //Enabled
stWriteParam_in.iParameterArrIndex  = 1;               //Parameter array index
stWriteParam_in.eParameterNumber    = MMC_POSITION_PERIOD_PARAM;//Axis parameter value
//
rc = MMC_GlobalWriteParameter(hConn, iAxisRef, &stWriteParam_in, &stWriteParam_out);
if (rc != 0)
{
    HandleError();
}
```

5.10 Multiple Axes Motion Control - Introduction

Motion Control describes the motion of the multiple axes as either NC or Distributed according to the definitions described in section **2.2 Maestro Operation Modes** and **2.2.3 Maestro Axes and Node Definitions** above. The following multiple axes function blocks are described:

Multiple Axes
MMC_GroupStop
MMC_GroupHalt
MMC_MoveCircularAbsolute
MMC_MoveCircularAbsoluteCenter
MMC_MoveCircularAbsoluteBorder
MMC_MoveCircularAbsoluteRadius
MMC_MoveCircularAbsoluteAngle
MMC_MoveLinearAbsolute
MMC_MoveLinearRelative
MMC_MoveLinearAdditive
MMC_MoveLinearAdditiveEx
MMC_MoveLinearAbsoluteRepetitive
MMC_MoveLinearRelativeRepetitive
MMC_MovePath
MMC_PathSelect
MMC_PathUnselect

The structured definitions MMC_MCS_Info_Struct and MMC_MCS_Kin_Ref_Struct are excluded from the above list as they are not function blocks.

5.10.1 Coordinate System and kinematic transformation

The essence of a trajectory is the coordinated motion of one or more axes from a starting point to a target point via a defined path with specified kinematic properties. As for path, one can think of a straight line, a circular movement, or via an input data array. The execution of a path– or any position information - in space requires a coordinate system definition. Within this specification, three coordinate systems are defined; ACS, MCS, and PCS.

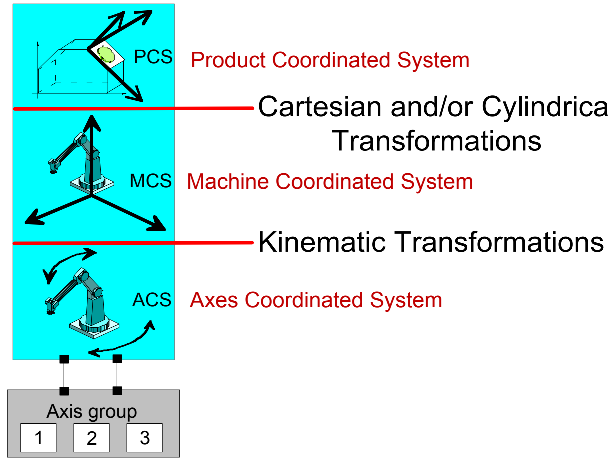


Figure 5-83: Overview of the coordinate systems and transformations

5.10.1.1 Coordinated Motion

When a function block is inserted for operation to a group, the kinematic mode is defined. However, each function block can be inserted in either ACS, MCS, or PCS kinematic mode. When operating in ACS mode no transformation occurs, but to operate in MCS or PCS mode, the kinematic system must be predefined using MMC_SetKinTransformEx and a transformation applied. **Each kinematic is defined per specific vector**.

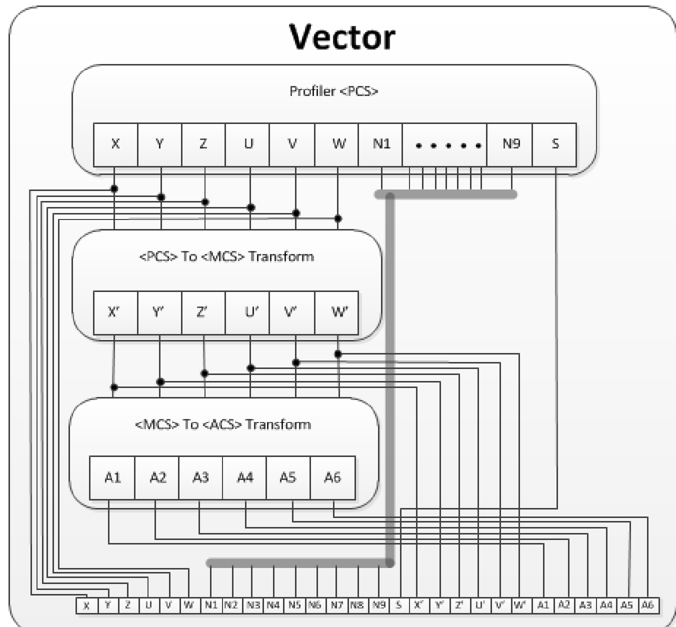


Figure 5-84: Vector kinematic transformations

The profiler of the Maestro generates up to 16 outputs. Actually, the profiler operates in the PCS environment. MCS and ACS have an additional 12 outputs generated by different kinematic transformations.

5.10.2 ACS - Axes Coordinate System

ACS represents the actual position of the axis in user units. **ACS** is the mode where all group member kinematic values are downloaded to the axes, without being transformed, or manipulated from the kinematic view point. Each axis in a group moves as a separate axis (point to point), but all axes' movement start and ends simultaneously. Linear, PVT, and Spline movements are permitted, and only one type of transition mode is acceptable, MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES. Any other transition will produce an error. The kinematic limitations refer to the group parameters:

- Max Vector Velocity
- Max Vector AC, DC
- Max Vector Jerk

However, the position values downloaded to the target drive are real values in count units, and the actual position displayed in user units. A maximum of 16 axes are allowed in ACS.

5.10.3 MCS - Machine Coordinate System

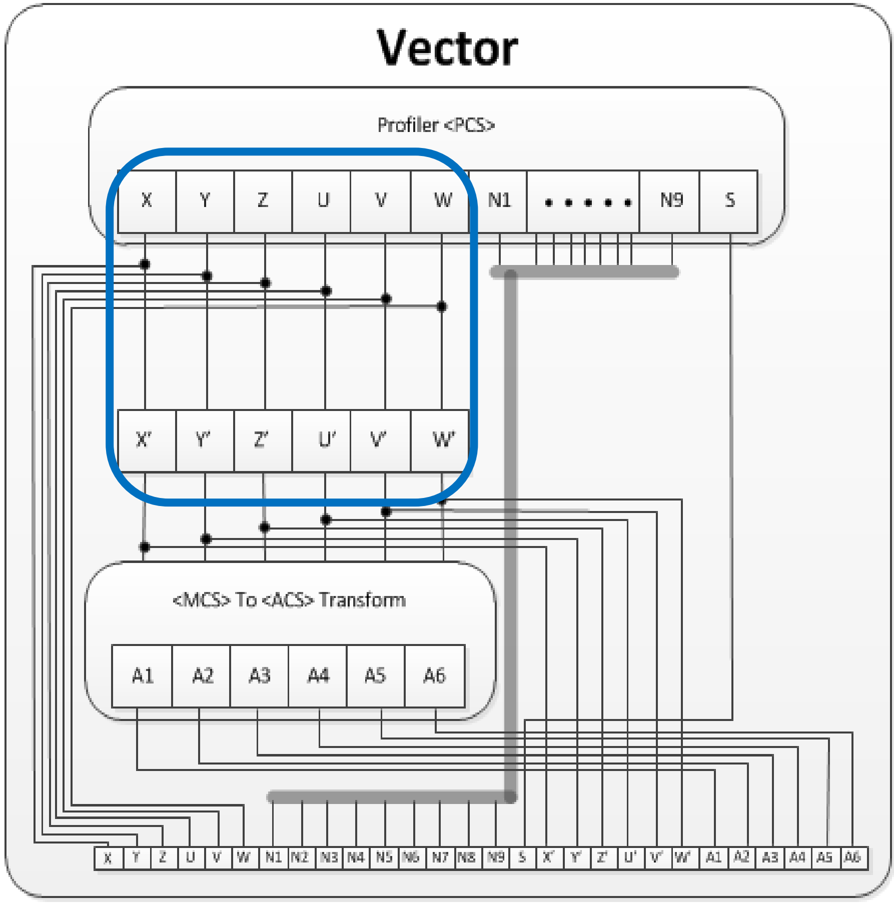
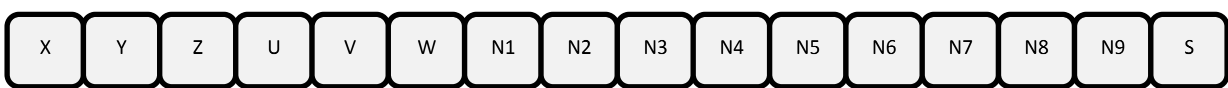
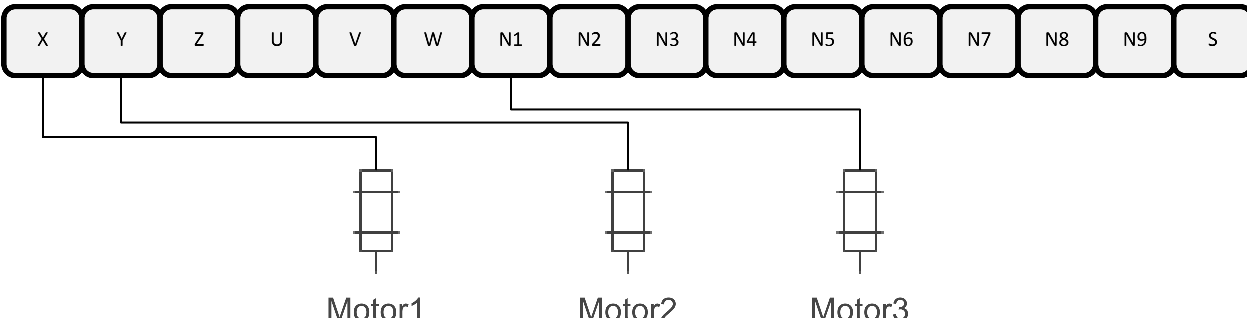


Figure 5-85 Direct Transformation from Profiler to MCS

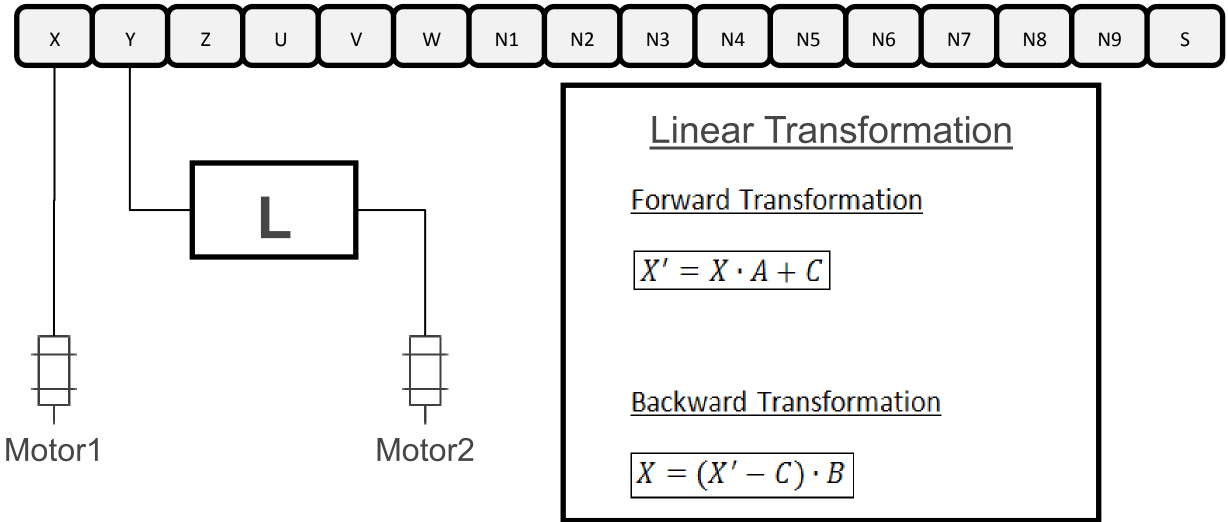
In this mode, the machine coordinated system is manipulated to allow transformation of kinematic values between the MCS and ACS coordinate systems. All the axes are grouped to some specific coordinate system, and start and end together (in terms of time). Each group can generate a profile with up to 16 kinematic directions.



Each axis in a group can be assigned to a different kinematic direction.



Each assigned axis can obtain the target position from the kinematic direction using a linear transformation (optional).



The motion is fully synchronized between the axes, and any type of vector movement; Linear, Circular, Polynomial, Spline, and PVT is permitted. To be able to work in MCS ↔ PCS modes the kinematic system should be predefined using the MMC_SetKinTransformEx() function. The kinematic limitations refer to the parameters per group:

- Max Vector Velocity
- Max Vector AC, DC
- Max Vector Jerk

When the linear transformation is defined, the transformation is performed in each cycle before download the target position to the drive as described in **Figure 5-86 (objects 0X607A, 0X6064, are based on the DS402 protocol).**

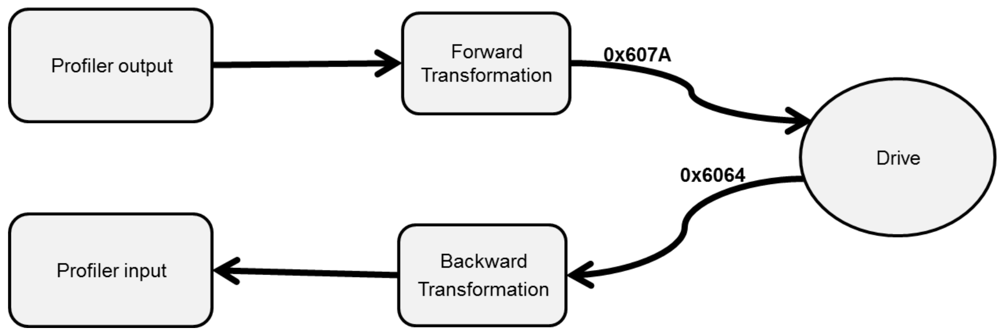
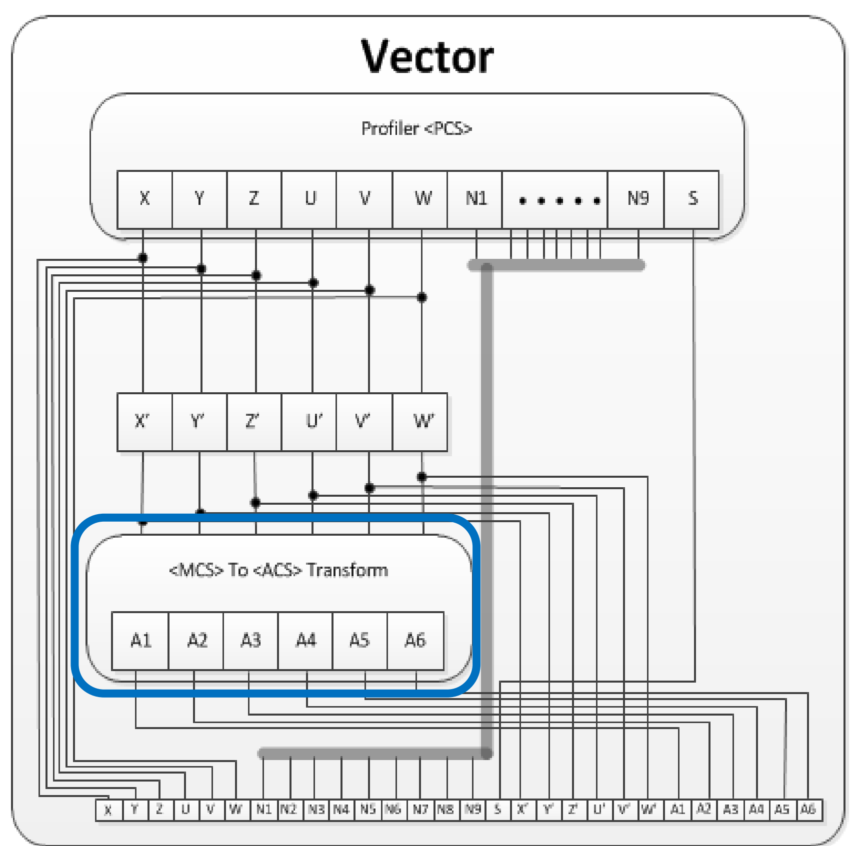


Figure 5-86: Position values transformed

MCS allows a maximum of 22 output kinematic directions, and offers the following types of coordinate system:

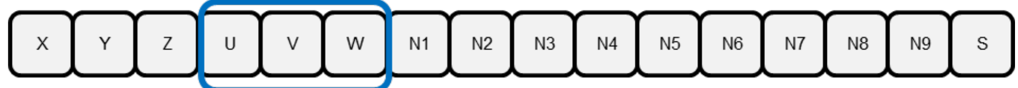
Cartesian The Cartesian coordinate system defines the origin as a fixed position relative to the machine. There are three outputs in this mode; X position, Y position, and Z position.

For some machines built when using the Cartesian system, the MCS may be identical to ACS, or mapped via a trivial transformation, the special robot transformation provide an addition six kinematic directions; A1,A2,A3,A4,A5,A6.

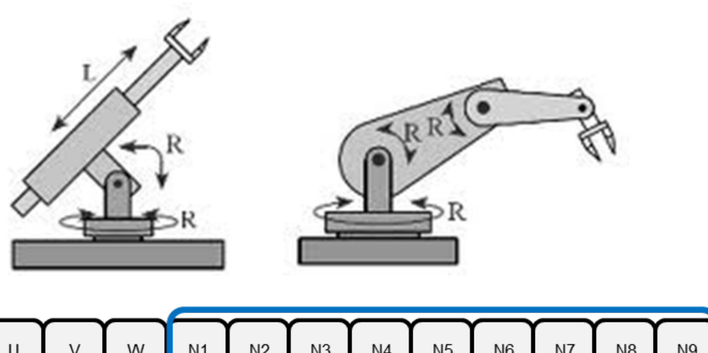


The physical axes are linked to the kinematic directions via a kinematic definition. If a linear transformation is defined for an axis, it will also be calculated for the forward and backward paths.

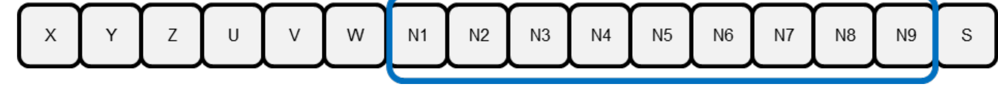
Polar



For the Polar mode, there are three outputs in this mode; U angle, V angle, and W angle. Currently this mode is not supported.



N Axes



Standalone axes (Service axes). In the N Axes mode, there are nine outputs; N1 – N9 positions.

S Direction

The output for S direction **during motion** is generated by other kinematic directions currently used. The velocity of the S direction is always non-negative, and there are two ways to change the value of S direction:

- Using parameter mechanism – in every PLC state MMC_MCS_S_DIRECTION
- Using other kinematic directions – when in motion

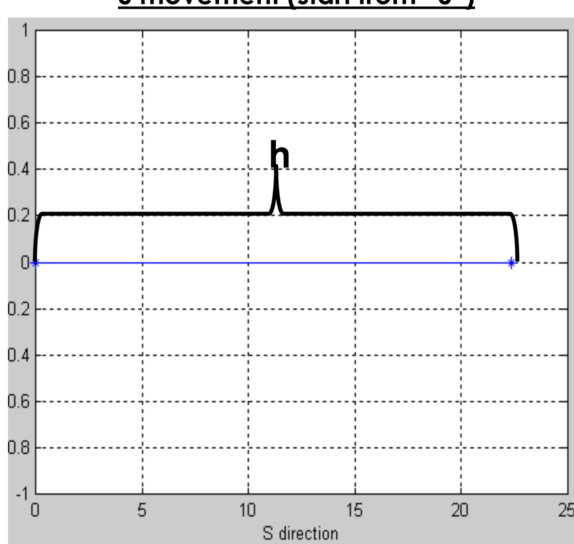
The Update of the **S Direction** during motion may be of three different types:

- **1st , Cartesian**
  When using only Cartesian kinematic directions, the output of the S is a spatial path vector size (without direction).

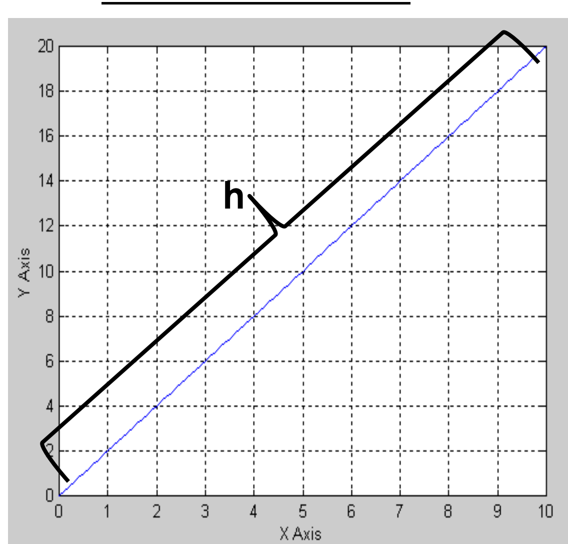
$$\Delta Pos_s = \sqrt{\Delta Pos_X{}^2 + \Delta Pos_Y{}^2 + \Delta Pos_Z{}^2}$$

Example: $\Delta S = 22.36 = \sqrt{10^2 + 20^2}$

S movement (start from "0")



2D cartesian movement



- **2nd , N axes**
  When using only N axes kinematic, the input is a position vector size of all active N axes (without direction)

$$\Delta Pos_s = \sqrt{\Delta Pos_{N1}{}^2 + \Delta Pos_{N2}{}^2 + \ldots + \Delta Pos_{N8}{}^2 + \Delta Pos_{N9}{}^2}$$

- **3rd , Cartesian and N axes**
  Behave as in Cartesian case

Another method to set the S direction, is using a Parameter mechanism using the **MMC_WriteParameter** function with the MMC_MCS_S_DIRECTION enumerator. This can be **applied for all PLC states**. In contrast, the velocity of the S direction is always non-negative.

5.10.3.1 Transition in MCS

When the coordinate system is mixed with Cartesian and N axes, only the blended motion is permitted, on the condition that the N axes are not moving, as shown in the table below.

No.	N axes moving	Cartesian axes moving	Transition acceptable
1	YES	YES	X
2	YES	NO	X
3	NO	YES	√

5.10.4 PCS - Product Coordinate System

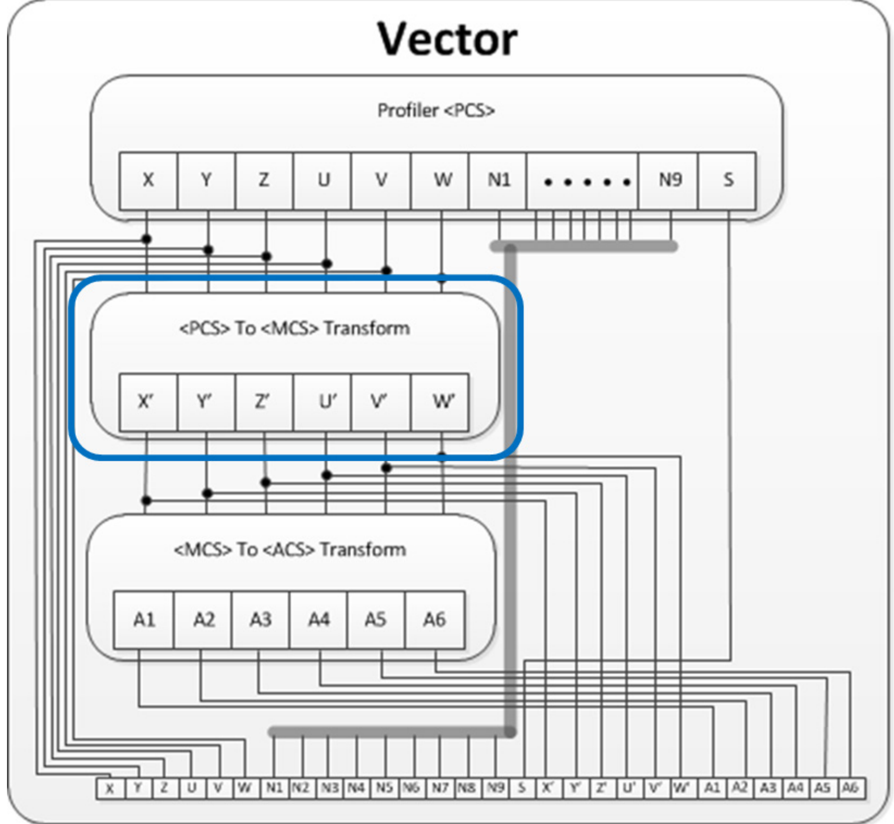


Figure 5-87 Indirect Transformation from PCS to MCS

Figure 5-92 describes the transformation from Profiler to MCS; X', Y', Z', U', V', W'. The PCS type offers the same facilities as MCS, i.e.:

- Kinematic directions
- Motion types
- Transitions
- Linear transformation
- Special robot transformation

Therefore all axes starts and end together (time wise). The motion is fully synchronized between the axes and allows a maximum of 28 output kinematic directions. However the PCS transformation is defined using the MMC_SetCartesianTransform function.

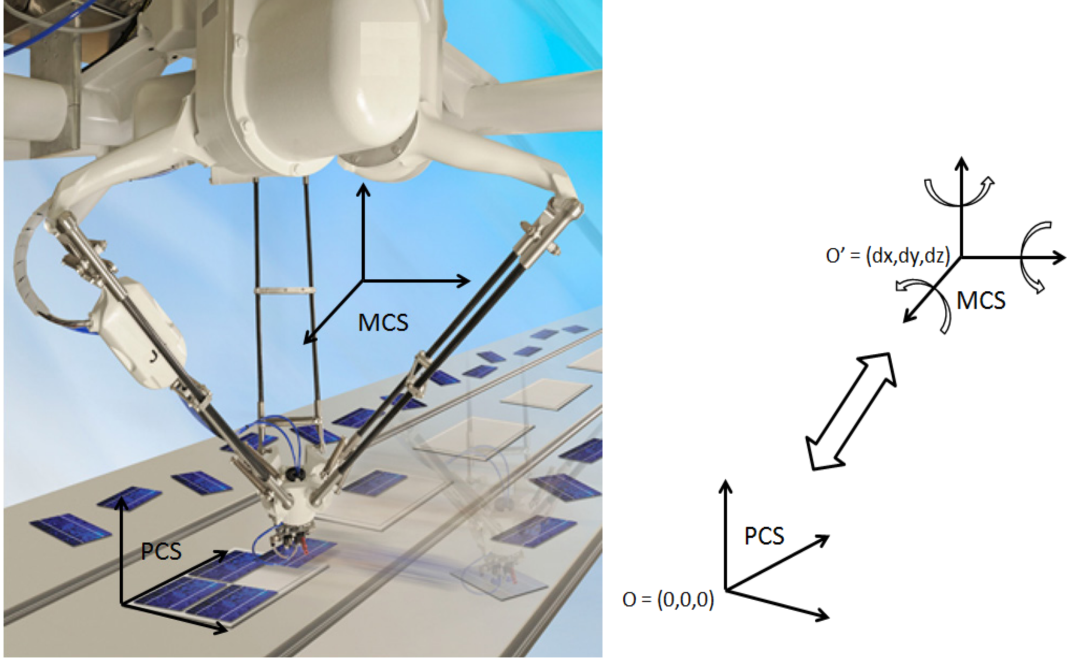
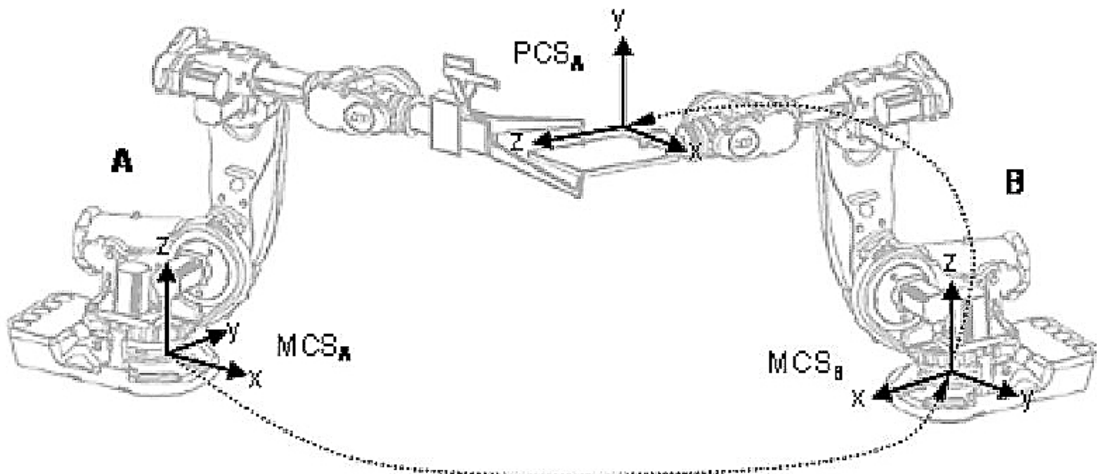


Figure 5-88 PCS to MCS conversion via transformation

The PCS to MCS transformation defines the conversion from product coordinate system to machine coordinate system and vice versa (Figure 5-88). In order to perform this transformation, the origin of the MCS coordinate system should be re-located and the orientation of the MCS coordinate system defined relative to the PCS coordinate system.

5.10.4.1 Tracking in Dynamic Coordinate Transformations

Tracking is characterized by an axis group's (A) movement that follows the movement of an individual axis or other axis group (B). During the ensuing coordinated tracking motion, A is performing a movement relative to the movement of B. In the diagram below, robot B is clasping a work section, while robot A is welding parts of the section simultaneously, while B orientates the section.



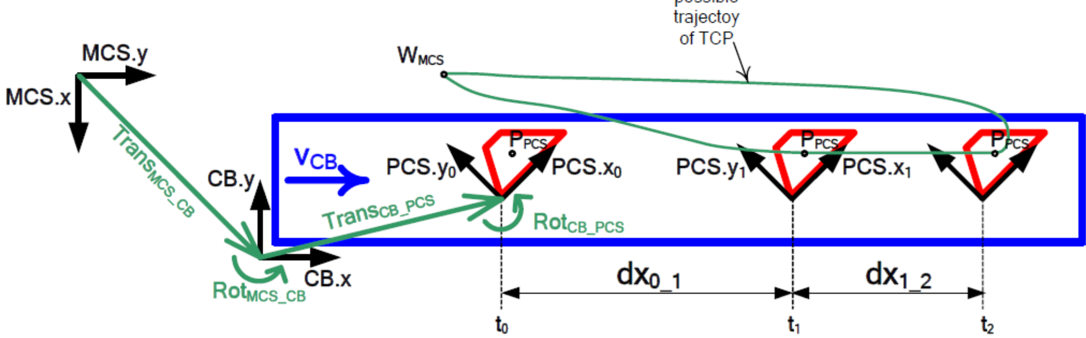
Tracking can be seen as a superposition of two independent movements:

1. The section movement - moving PCS. The position of the PCS and therefore also the movement of the PCS relative to MCS is described by the dynamic coordinate transformation MCS to PCS.

2. The movement of the Tool Center Point (TCP) that would be executed, if the product is standing still. This movement has to be defined in PCS.

The activation of a dynamic transformation is performed by activating **MMC_SetDynCoordTransform** in a general case or by activating of **MMC_TrackConveyorBelt** and **MMC_TrackRotaryTable** for specific applications.

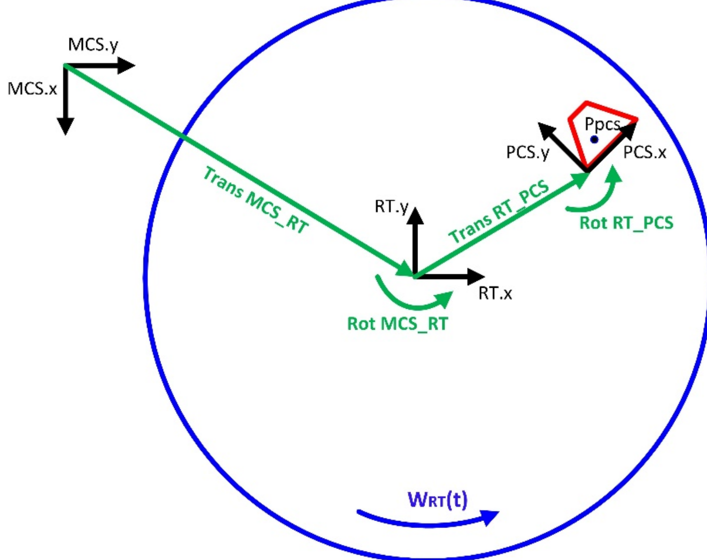
5.10.4.2 Tracking Workpiece Processing on a Conveyor Belt



To process a workpiece, starting from a certain position P_{PCS} (X_{PCS}; Y_{PCS}; 0) on the conveyor belt:

- The workpiece is lying on a conveyor belt (CB) which is moving with speed V_{CB}. The axes group is in its waiting position W_{MCS}.
- **t_0**: the workpiece is detected by a digital camera, then:
  - The camera provides the position (and orientation) of the workpiece relative to the conveyor belt.
  - The MC_TouchProbe provides the absolute position of the conveyor belt.
  - The function block **MMC_TrackConveyorBelt** is executed. The automatic calculation of the coordinate transformation MCS ↔ PCS is activated.
  - Activating MC_TrackConveyorBelt can simultaneously activate execution of the function block's commanding motion to P_{PCS}.
- **t_0 - t_1**: the axes group moves from its waiting position W_{MCS} towards P_{PCS}. This includes:
  - Synchronizing the axes group with the conveyor motion (GearIn with ratio 1).
  - Execute group coordinated motion to P_{PCS}.
- **t_1**: the axes group has reached P_{PCS} (in the meantime the conveyor belt has moved the distance dx0_1).
- **t_1 - t_2**: after having reached P_{PCS} the process on workpiece can be started. All positions within the process have to be related to PCS.
- **t_2**: the process is finished and the axes group is commanded to move back to its waiting position W_{MCS}.
  - Release synchronization of the axes group ($Gear_{In}$ with ratio 0).
  - Execute group coordinated motion to W_{MCS}.

5.10.4.3 Tracking Workpiece Processing on a Rotary Table



The work piece is lying on the rotary table, and being processed similar to the workpiece lying on the conveyor belt. Refer to to the section 5.10.4.2 above. However, in this case, the workpiece is lying on a rotary table (RT) which is rotating with angular speed $\dot{W}_{RT}(t)$.

5.10.4.4 Using the function MMC_SetKinTransformEx

The function definitions are only relevant for MCS/PCS modes, and can only be operated when the group is in non-motion state. The function received a structure that define two types of parameters:

- General
- Axis related

The general parameters involve, the number of axes used in the motion, the type of the kinematic and any data that related to a specific robot transformation.

The axis related parameters involve, assigning each axis to a specific kinematic direction, deciding whether to use linear transformation or not, and if linear transformation is selected, set the coefficients. Additionally, the parameters involve any data related to a specific robot transformation.

The function defines the following parameters:

<table>
<tr><th>Parameter</th><th>Explanation</th></tr>
<tr><td colspan="2">General Inputs</td></tr>
<tr><td>Number of axes to be used in motion (iNumAxes)</td><td>This number defines how many axes are part of the kinematic. This parameter is limited by the total number of axes in the group. It can be smaller than the total number in the group (currently not supported)

<code>MC_KIN_REF_DELTA stDelta;
stDelta.iNumAxes = 3;</code>

<code>MC_KIN_REF_CARTESIAN stCart;
stCart.iNumAxes = 3;</code></td></tr>
<tr><td>Kinematic (eKinType)</td><td>This is the kinematic system type. If specific robot is not used, set Cartesian type:

<code>MMC_SETKINTRANSFORMEX_IN SetKinTransformInput;
SetKinTransformInput.eKinType = NC_CARTESIAN_TYPE;</code></td></tr>
</table>

<table>
<tr><th>Parameter</th><th>Explanation</th></tr>
<tr><td></td><td>If specific robot is used, set a proper type, for example:
<code>MMC_SETKINTRANSFORMEX_IN SetKinTransformInput;</code>
<code>SetKinTransformInput.eKinType = NC_DELTA_ROBOT_TYPE;</code></td></tr>
<tr><td colspan="2">Axis related inputs
All axis related parameters are defined in the MC_KIN_NODE_DEF structure.</td></tr>
<tr><td>Axis reference (hNode)</td><td>This is the axis reference of a given axis. This parameter received from MMC_AxisByName function. Refer to the explanation in section 5.15.11 MMC_SetKinTransformEx for this parameter.</td></tr>
<tr><td>Axis type (eType)</td><td>Set a proper kinematic direction for each axis .</td></tr>
</table>

Parameter	Explanation
Linear Transformation (*iMcsToAcsFuncID*)	Currently Elmo supports only two transformation types: • Without Transformation • Linear Transformation The transformation is between the kinematic direction and drive. Set whether to use linear transformation (for each axis).

X Y Z U V W N1 N2 N3 N4 N5 N6 N7 N8 N9 S

L

Forward Transformation

$X' = X \cdot 10 + 3$

Backward Transformation

$X = (X' - 3) \cdot 0.1$

Motor1 Motor2

```
MC_KIN_REF_CARTESIAN stCart;

// First axis
stCart.sNode[0].hNode = AxisReferenceA;
stCart.sNode[0].eType = NC_PROFILER_X_AXIS_TYPE;
stCart.sNode[0].iMcsToAcsFuncID = NC_TR_NONE_FUNC;
// no needed to set coefficients

// Second axis
stCart.sNode[1].hNode = AxisReferenceB;
stCart.sNode[1].eType = NC_PROFILER_Y_AXIS_TYPE;
stCart.sNode[1].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;
// set coefficients
stCart.sNode[1].ulTrCoef[0] = 10;  // A
stCart.sNode[1].ulTrCoef[1] = 0.1; // B
stCart.sNode[1].ulTrCoef[2] = 3;   // C
```

Linear Transformation

The Inputs coefficients should be:

- Forward Ratio (FR)
- Backward Ratio (BR)
- Shift (SH)

<table>
<tr><th>Parameter</th><th>Explanation</th></tr>
<tr><td></td><td>The Transition functions are defined as:
Forward: Position = Position * FR + SH
Backward: Position = Position * BD – SH
Without Transformation
The Inputs are not in use:
The Transition functions are defined as:
Forward: Position = Position
Backward: Position = Position</td></tr>
<tr><td>Linear Transformation Coefficients (ulTrCoef[…])</td><td>When selecting to set a linear transformation for a specific axis, define the coefficients of the transformation.
Example:<pre>
MC_KIN_REF_CARTESIAN stCart;

// First axis
stCart.sNode[0].hNode = AxisReferenceA;
stCart.sNode[0].eType = NC_PROFILER_X_AXIS_TYPE;
stCart.sNode[0].iMcsToAcsFuncID = NC_TR_NONE_FUNC;
// no needed to set coefficients

// Second axis
stCart.sNode[1].hNode = AxisReferenceB;
stCart.sNode[1].eType = NC_PROFILER_Y_AXIS_TYPE;
stCart.sNode[1].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;
// set coefficients
stCart.sNode[1].ulTrCoef[0] = 10; // A
stCart.sNode[1].ulTrCoef[1] = 0.1; // B
stCart.sNode[1].ulTrCoef[2] = 3;   // C
</pre>An important point when entering the coefficients:
In order to avoid problematic position adjustments (when using linear transformation) always maintain the following ratio between the coefficients:
NC_BACK_TR_RATIO_COEF.NC_BACK_SHIFT_COEF =1; (A.B = 1)</td></tr>
</table>

5.10.4.5 Implementation

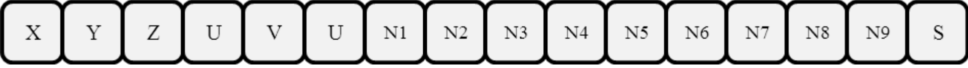
Each Multiaxis function has an MC_COORD_SYSTEM_ENUM input which define the coordinate system of the current function block:

- ACS
- MCS
- PCS

The user can change the Coordinate system every function block, but the Group should be in non-motion state. Each Multi axis motion function has 16 Position inputs. Position inputs have a different meaning in each coordinate system.

Position Input Array – ACS (each number is indexed inside group)

Position Input Array – MCS



Position Input Array – PCS

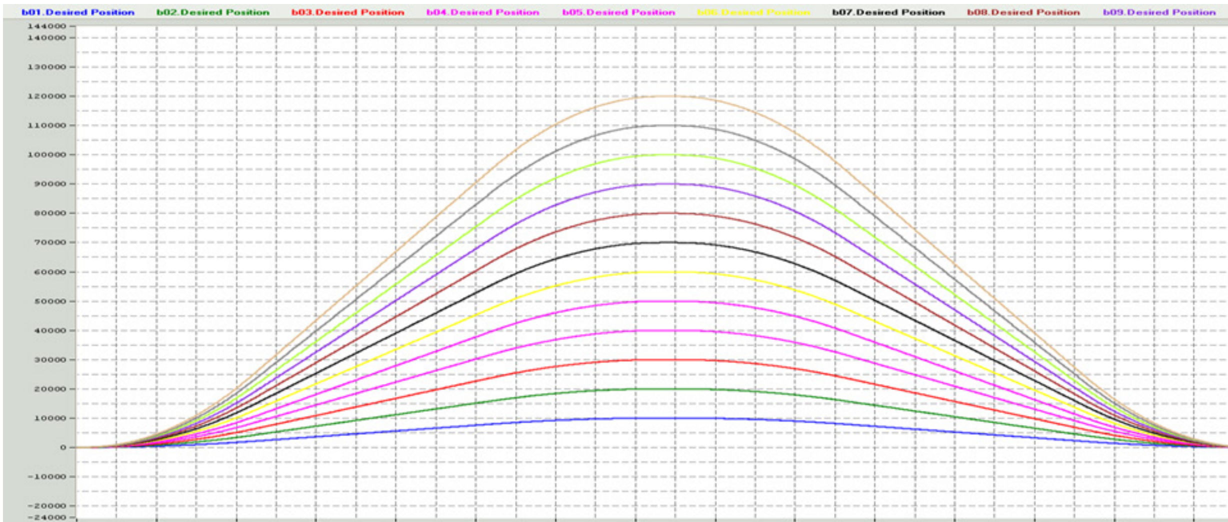
5.10.5 General Example

These sections describe a simple examples of ACS and MCS linear motion.

ACS – Two FB's (Move Linear), Start position – all axes in 0.

FB1 – Position[10,20,30,40,50,60,70,80,90,100,110,120,0,0,0,0]

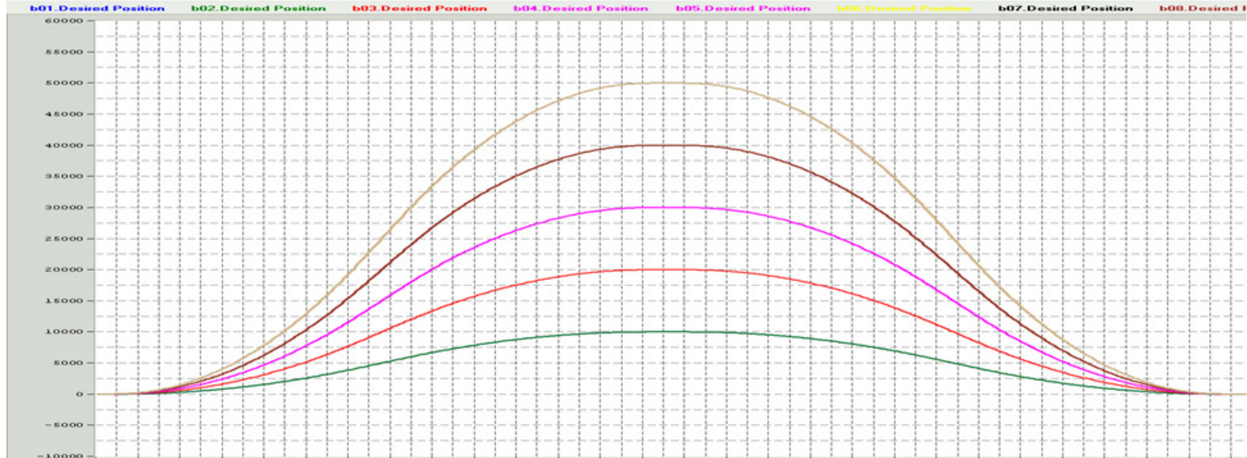
FB2 – Position[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]



MCS - Two FB’s (Move Linear), Kinematic: (2-X),(1-Y),(2-Z),(3-N1),(4-N2), Start position – all axes in 0.

FB1 – Position[10, 20, 30, 0, 0, 0, 40, 50, 0, 0, 0, 0, 0, 0, 0, 0]

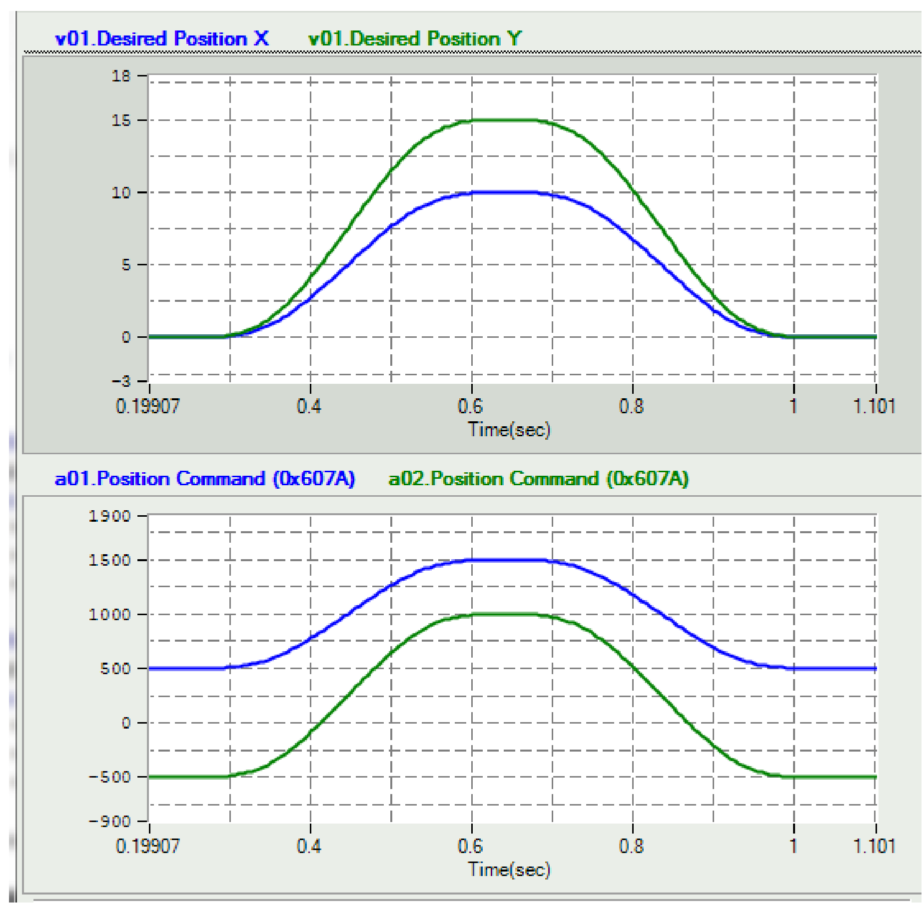
FB2 – Position[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]



5.10.6 Example of Set Kinematic

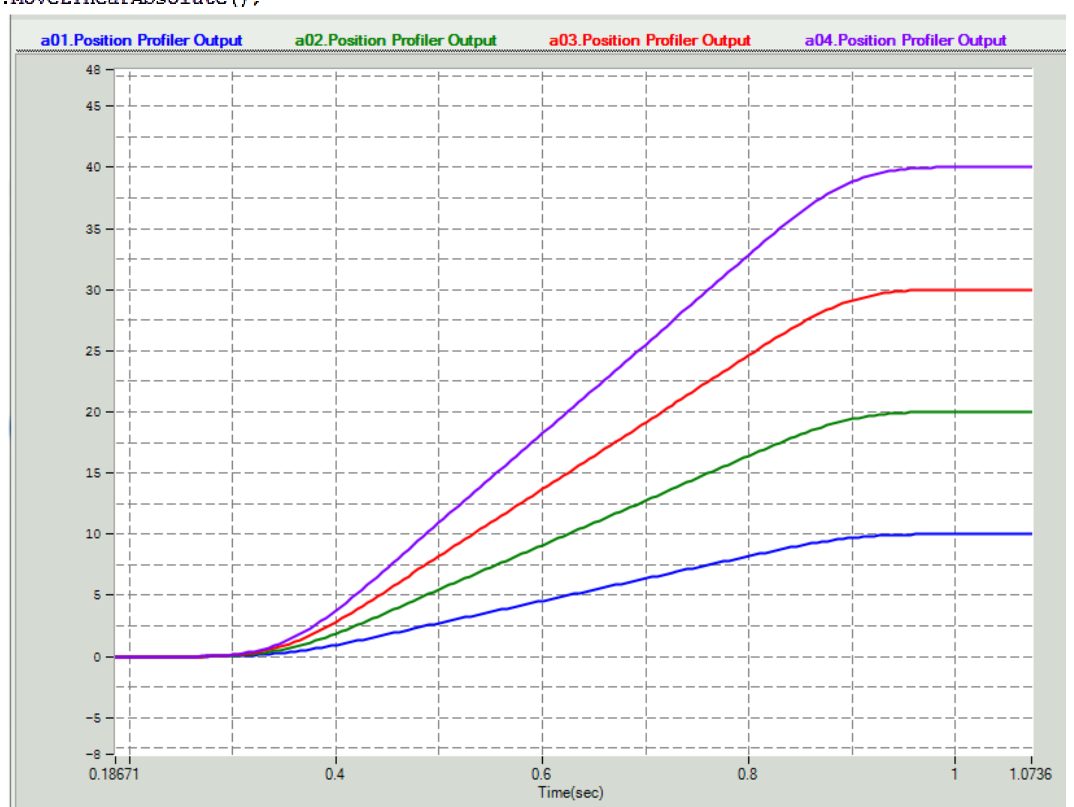
```
CMMCConnection Connection;
CMMCSingleAxis AxisA,AxisB;
CMMCGroupAxis GROUP;
//
// Create connection
ui_conn_hndl = Connection.ConnectIPCEx(0,NULL);
//
// Define the relevant axes and group
AxisA.InitAxisData("a01",ui_conn_hndl);
AxisB.InitAxisData("a02",ui_conn_hndl);
GROUP.InitAxisData("v01",ui_conn_hndl);
//
MC_KIN_REF_CARTESIAN stCartesianKinematic;
//
stCartesianKinematic.iNumAxes = 2;
stCartesianKinematic.sNode[0].eType = NC_PROFILER_X_AXIS_TYPE;
stCartesianKinematic.sNode[0].hNode = AxisA.GetRef();
stCartesianKinematic.sNode[0].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;
stCartesianKinematic.sNode[0].ulTrCoef[0] = 100;
stCartesianKinematic.sNode[0].ulTrCoef[1] = 0.01;
stCartesianKinematic.sNode[0].ulTrCoef[2] = 500;
//
stCartesianKinematic.sNode[1].eType = NC_PROFILER_Y_AXIS_TYPE;
stCartesianKinematic.sNode[1].hNode = AxisB.GetRef();
stCartesianKinematic.sNode[1].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;
stCartesianKinematic.sNode[1].ulTrCoef[0] = 100;
stCartesianKinematic.sNode[1].ulTrCoef[1] = 0.01;
stCartesianKinematic.sNode[1].ulTrCoef[2] = -500;
//
GROUP.SetCartesianKinematic(stCartesianKinematic);
```

Start position – (0,0), End position1 – (10,15), End position2 – (0,0)



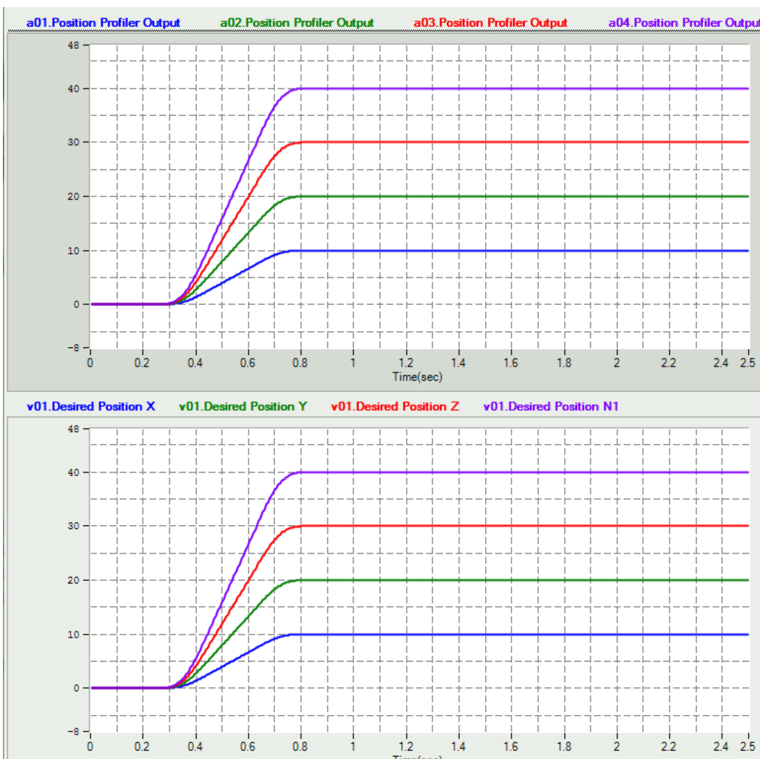
5.10.7 Example - ACS Motion

```
CMMCGroupAxis GROUP;

// Set kinematic values
GROUP.m_fVelocity = 100;
GROUP.m_fAcceleration = 100;
GROUP.m_fDeceleration = 100;
GROUP.m_fJerk = 1000;

// Set end position
GROUP.m_dEndPoint[0] = 10;
GROUP.m_dEndPoint[1] = 20;
GROUP.m_dEndPoint[2] = 30;
GROUP.m_dEndPoint[3] = 40;

// Set coordinate system
GROUP.m_eCoordSystem = MC_ACS_COORD;

// Run motion
GROUP.MoveLinearAbsolute();
```



5.10.8 Example - MCS Motion

```
CMMCGroupAxis GROUP;

// Set kinematic values
GROUP.m_fVelocity = 100;
GROUP.m_fAcceleration = 100;
GROUP.m_fDeceleration = 100;
GROUP.m_fJerk = 1000;

// Set end position
GROUP.m_dEndPoint[NC_PROFILER_X_AXIS_TYPE] = 10;
GROUP.m_dEndPoint[NC_PROFILER_Y_AXIS_TYPE] = 20;
GROUP.m_dEndPoint[NC_PROFILER_Z_AXIS_TYPE] = 30;
GROUP.m_dEndPoint[NC_PROFILER_N1_AXIS_TYPE] = 40;

// Set coordinate system
GROUP.m_eCoordSystem = MC_MCS_COORD;

// Run motion
GROUP.MoveLinearAbsolute();
```



5.10.9 Special Robot Transformations

This section describes special robot transformations of which currently Elmo supports the Delta Robot. The special transformation converts the MCS kinematic directions to ACS kinematic directions as shown in **Figure 5-89**:

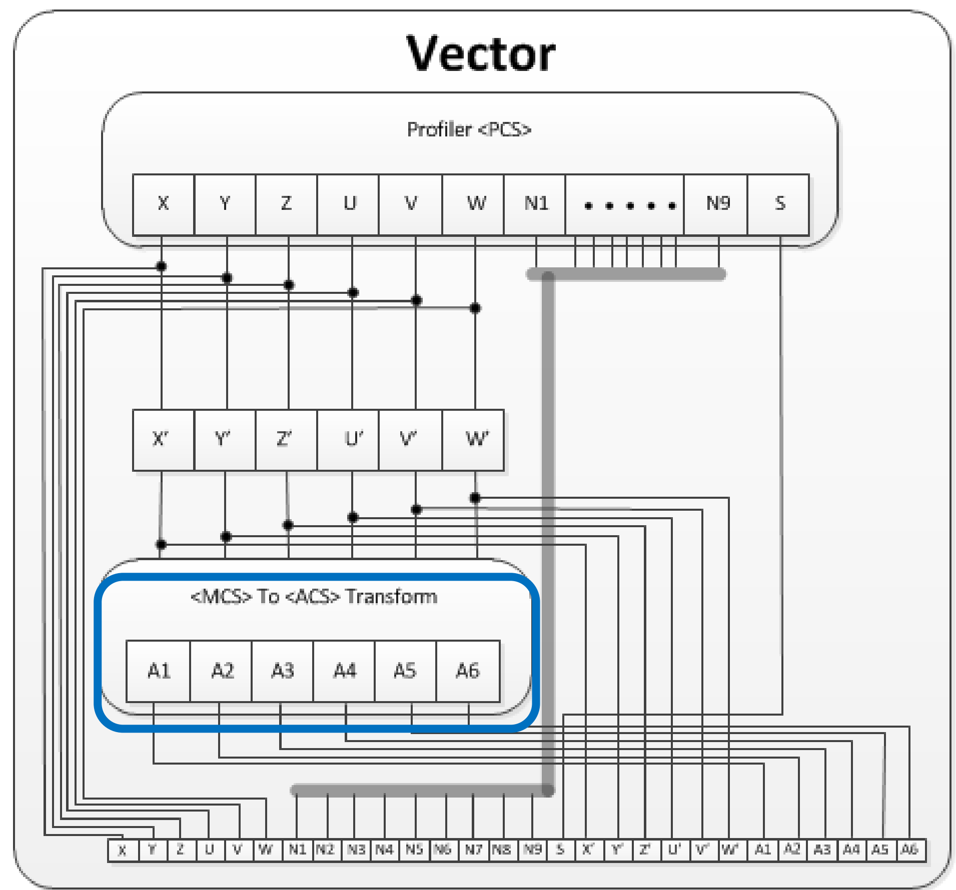


Figure 5-89 Indirect Transformation from MCS to ACS

5.10.9.1 Delta Robot kinematic

5.10.9.1.1 General

The Delta Robot is a type of Parallel Robot, with three parallel serial chains providing three degrees of freedom to the end effector. The benefits of the robot are:

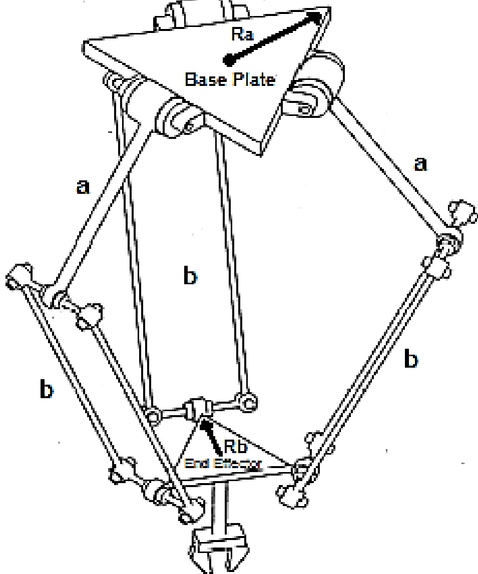
- High accuracy
- High speed dynamics
- Low inertia

The weakness is the complex calculations in real time, and the limited working space.

The Special Kinematic can be set in the Maestro, for both MCS and PCS coordinate systems, and all types of MultiAxis motions and transitions are supported. The Limit handling mechanism also supports ACS and MCS limits.

The Maestro can run a single Delta Robot in each vector, in up to 16 vectors, meaning that up to 16 Delta Robots can be operated at one time, using the ReadGroupActualPosition function. Any mix of kinematic

directions are supported. The units of the target position (only in the A1-A6 directions) is the same as the units of the mechanic inputs of the robot.



The kinematic transformations on the Maestro convert the position and other kinematics from Cartesian space to Joint space and vice versa. The transformations can then be defined as:

- Inverse Kinematic
- Direct Kinematic

In order to clearly define the Inverse and Direct kinematics, it is necessary to define the Cartesian and Joint spaces (Figure 5-91), the origin and orientation of the Cartesian space, and the direction of the Theta (Figure 5-90) in the joint space.

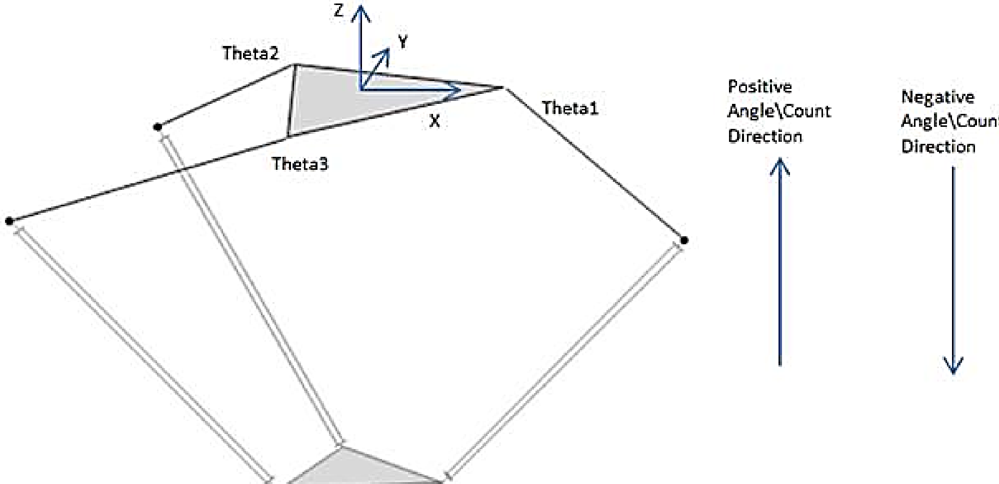


Figure 5-90: Defining Inverse and Direct Kinematics

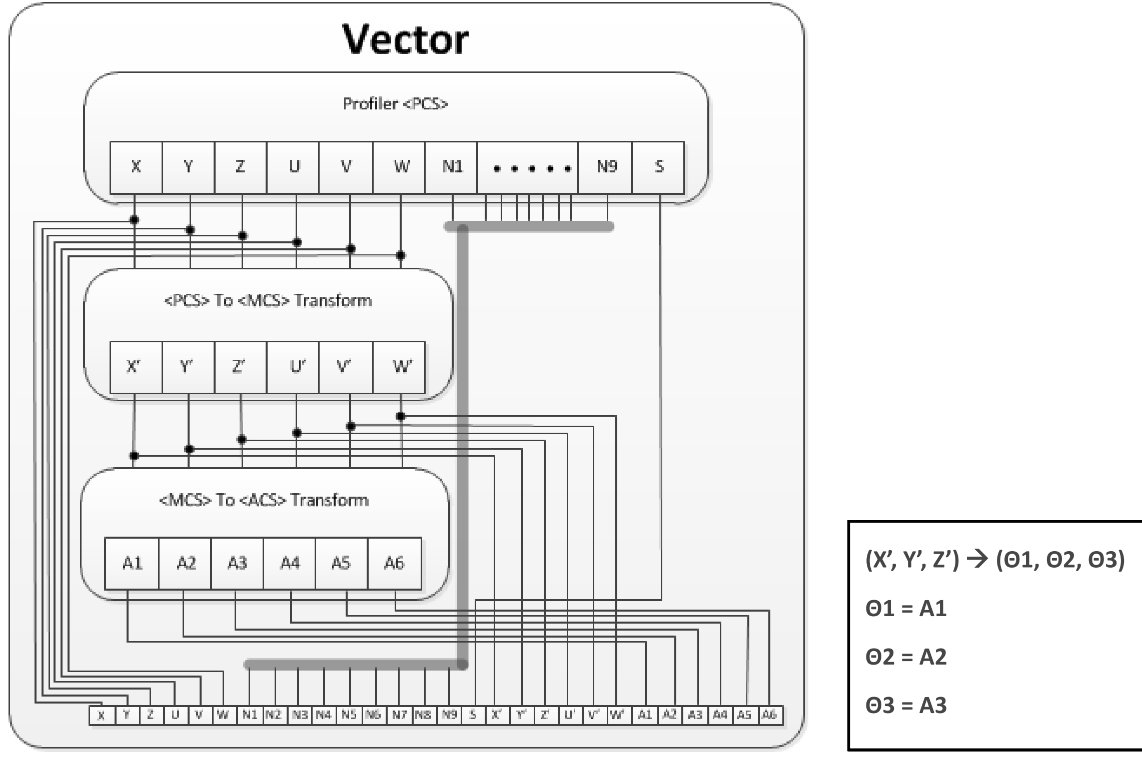
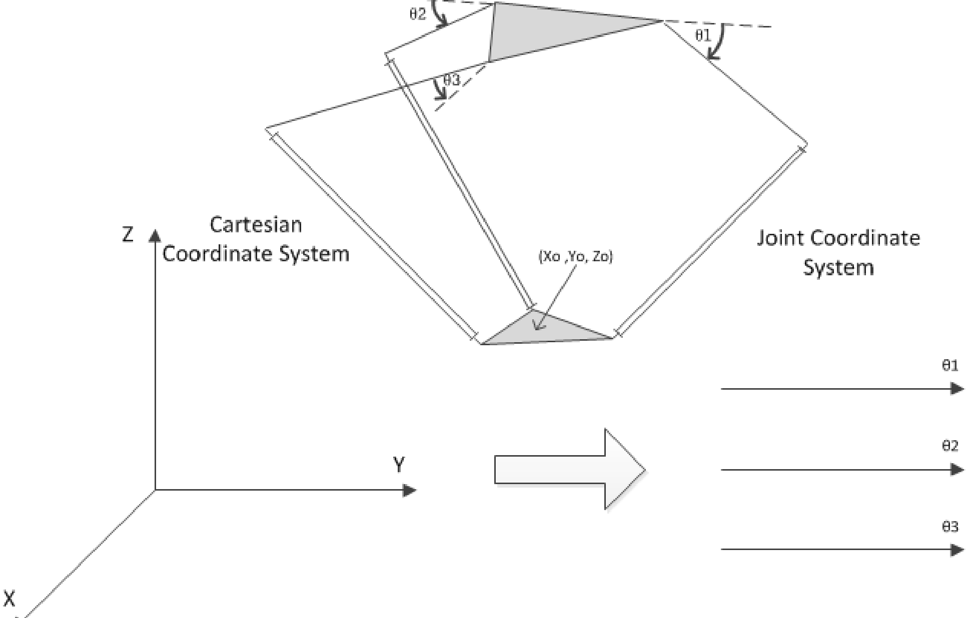


Figure 5-91: Defining Cartesian and Joint spaces

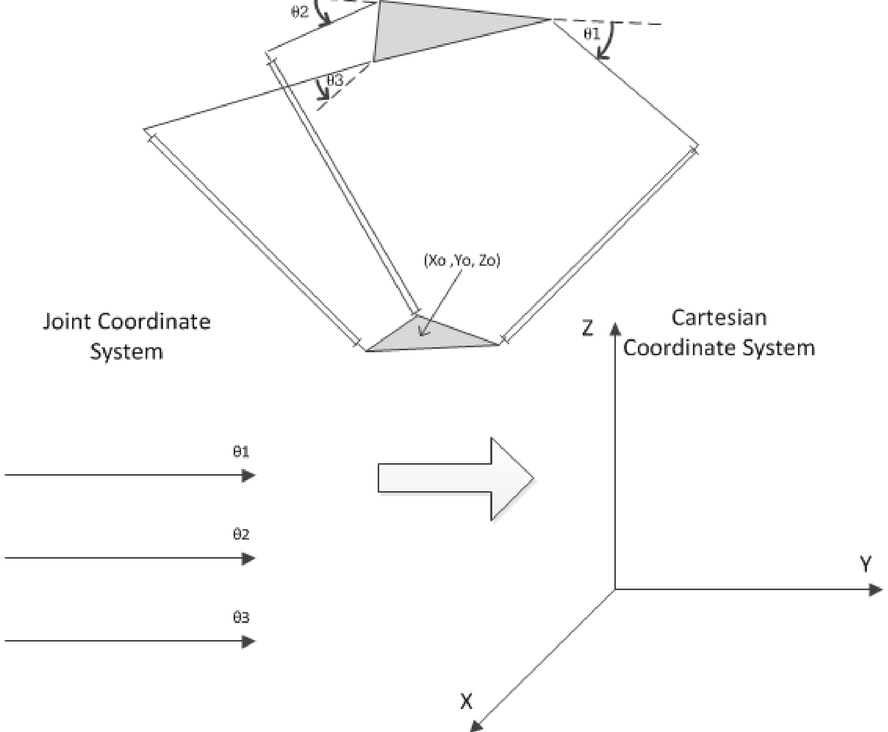
5.10.9.1.2 Inverse Kinematic

Inverse kinematic can convert the target position in Cartesian space to position in Joint space of the Delta Robot. The inverse kinematic is performed in each real time cycle, during motion.



5.10.9.1.3 Direct Kinematic

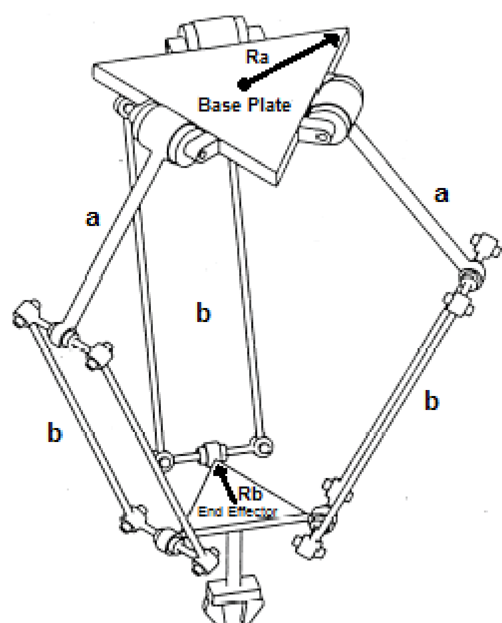
Direct kinematic converts the target position in joint Delta Robot space to a position in Cartesian space. The transformation is performed when the actual position of the end effector is read.



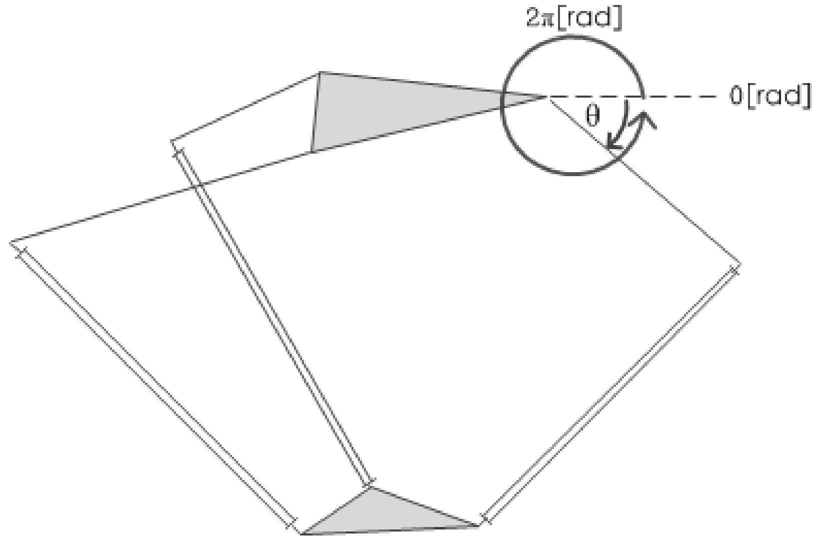
5.10.9.1.4 Interfaces

To use the Delta Robot transformation the user should define the following parameters:

- Mechanical inputs of the robot; Arm length (a), Forearm length (b), Base Radius (Ra), and End Effector Radius (Rb)

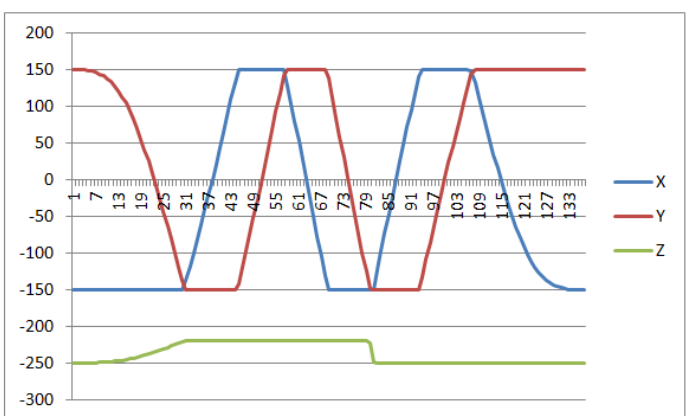


- Teaching the Maestro the “Homing” position of the drive encoder; including the parameters Count to Angle ratio. The number of counts per one Arm revolution (2π). The count offset at zero angle.

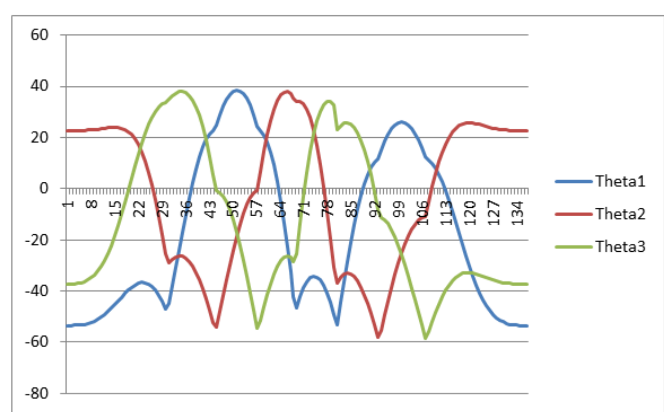


5.10.9.1.5 Transformation between spaces - example 1

Cartesian - space

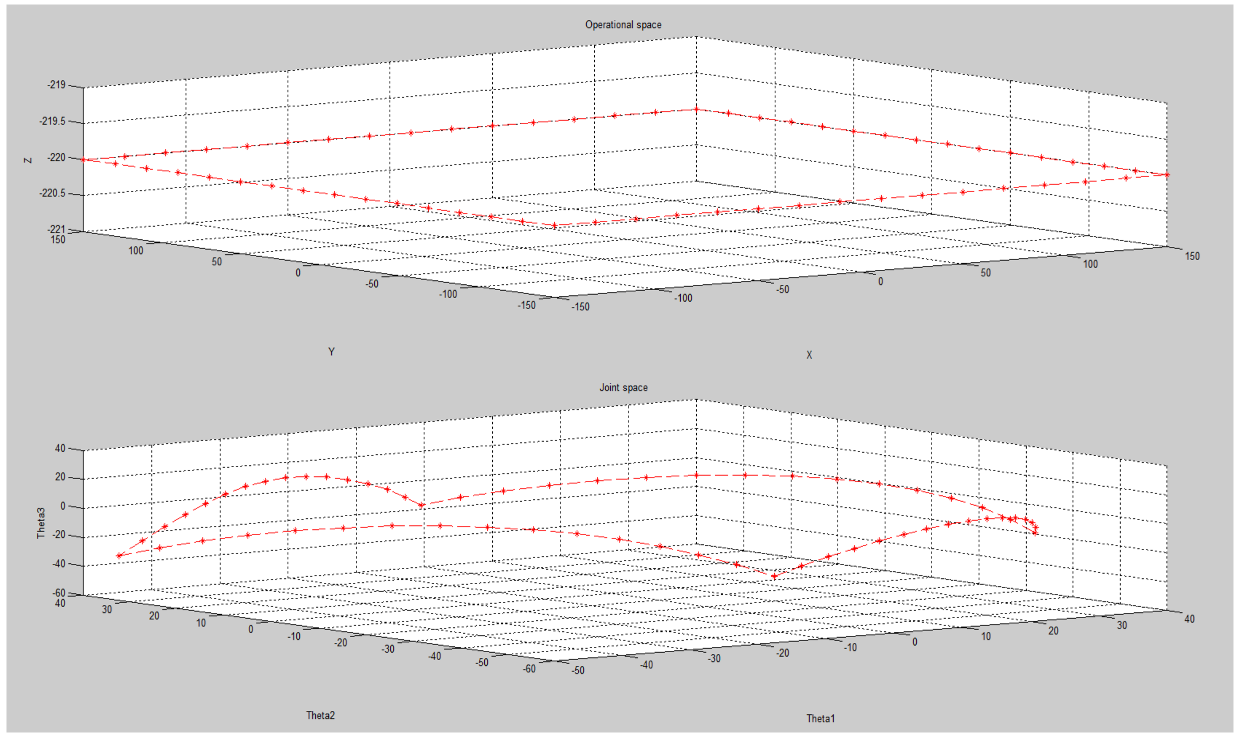


Joint - space



5.10.9.1.6 Transformation between spaces - example 2

This example shows the same motion in two different spaces, one in operational space (Cartesian) and the other in joint space (angle displacement of each leg).

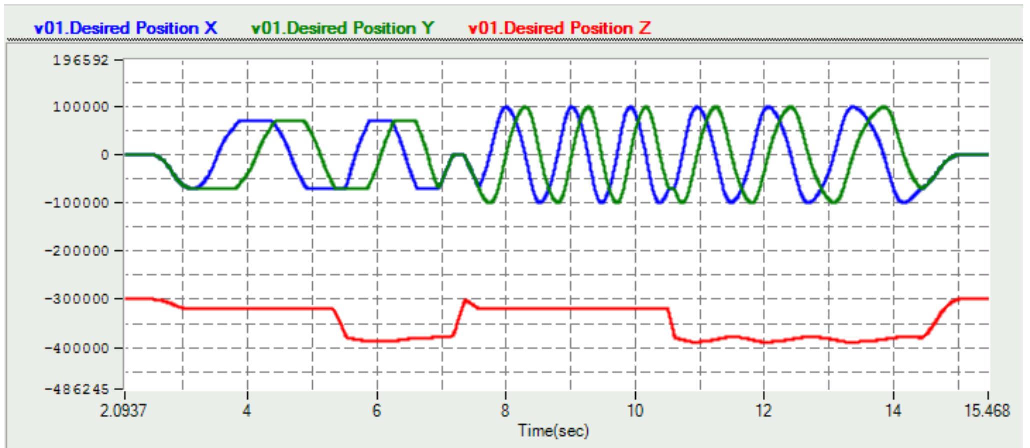


5.10.9.1.7 Example 1 - Set kinematic of delta robot (only 3 delta robot axes)

```
MC_KIN_REF_DELTA DeltaRobotKin;

// Set Mechanic parameters (in mm)
DeltaRobotKin.dbArm = 160;
DeltaRobotKin.dbForeArm = 320;
DeltaRobotKin.dbBaseRadius = 50;
DeltaRobotKin.dbEndEffectorRadius = 40;

// Set the number of axes
DeltaRobotKin.iNumAxes = 3;

// Assign the axes to kinematic directions

// Theta 1
DeltaRobotKin.sNode[0].eType = NC_ACS_A1_AXIS_TYPE;
DeltaRobotKin.sNode[0].hNode = AxisA.GetRef();
DeltaRobotKin.sNode[0].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;

// Theta 2
DeltaRobotKin.sNode[1].eType = NC_ACS_A2_AXIS_TYPE;
DeltaRobotKin.sNode[1].hNode = AxisB.GetRef();
DeltaRobotKin.sNode[1].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;

// Theta 3
DeltaRobotKin.sNode[2].eType = NC_ACS_A3_AXIS_TYPE;
DeltaRobotKin.sNode[2].hNode = AxisC.GetRef();
DeltaRobotKin.sNode[2].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;
```

```
// Set the encoder inputs

// Encoder to Angle ratio
// (encoder with 17 bits per one revolution)
DeltaRobotKin.sNode[0].ulTrCoef[0] = 131072.0;
DeltaRobotKin.sNode[0].ulTrCoef[1] = 1.0/(131072.0);
DeltaRobotKin.sNode[1].ulTrCoef[0] = 131072.0;
DeltaRobotKin.sNode[1].ulTrCoef[1] = 1.0/(131072.0);
DeltaRobotKin.sNode[2].ulTrCoef[0] = 131072.0;
DeltaRobotKin.sNode[2].ulTrCoef[1] = 1.0/(131072.0);

// Encoder offset - the caounts value at 0 angle
DeltaRobotKin.sNode[0].ulTrCoef[2] = Theta1OffsetCnt;
DeltaRobotKin.sNode[1].ulTrCoef[2] = Theta2OffsetCnt;
DeltaRobotKin.sNode[2].ulTrCoef[2] = Theta3OffsetCnt;

GROUP.SetDeltaRobotKinematic(DeltaRobotKin);
```

Results are:



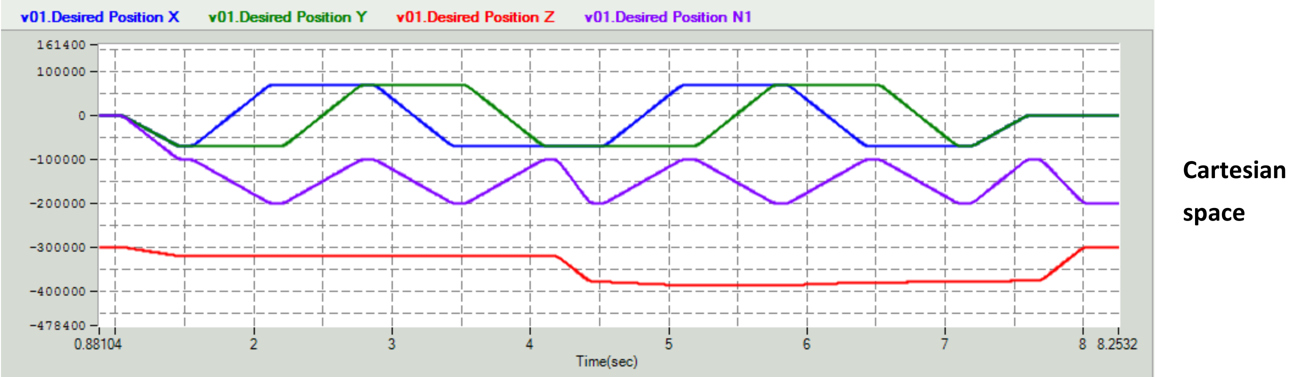
Cartesian space

Joint space

5.10.9.1.8 Example 2 - Set kinematic of delta robot with one service axis

```
MC_KIN_REF_DELTA DeltaRobotKin;

// Set Mechanic parameters (in mm)
DeltaRobotKin.dbArm = 160;
DeltaRobotKin.dbForeArm = 320;
DeltaRobotKin.dbBaseRadius = 50;
DeltaRobotKin.dbEndEffectorRadius = 40;

// Set the number of axes
DeltaRobotKin.iNumAxes = 4;

// Assign the axes to kinematic directions

// Theta 1
DeltaRobotKin.sNode[0].eType = NC_ACS_A1_AXIS_TYPE;
DeltaRobotKin.sNode[0].hNode = AxisA.GetRef();
DeltaRobotKin.sNode[0].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;

// Theta 2
DeltaRobotKin.sNode[1].eType = NC_ACS_A2_AXIS_TYPE;
DeltaRobotKin.sNode[1].hNode = AxisB.GetRef();
DeltaRobotKin.sNode[1].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;

// Theta 3
DeltaRobotKin.sNode[2].eType = NC_ACS_A3_AXIS_TYPE;
DeltaRobotKin.sNode[2].hNode = AxisC.GetRef();
DeltaRobotKin.sNode[2].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;

// N axis - with linear transformation
DeltaRobotKin.sNode[2].eType = NC_PROFILER_N1_AXIS_TYPE;
DeltaRobotKin.sNode[2].hNode = AxisD.GetRef();
DeltaRobotKin.sNode[2].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC;

// Set the encoder inputs

// Encoder to Angle ratio
// (encoder with 17 bits per one revolution)
DeltaRobotKin.sNode[0].ulTrCoef[0] = 131072.0;
DeltaRobotKin.sNode[0].ulTrCoef[1] = 1.0/(131072.0);
DeltaRobotKin.sNode[1].ulTrCoef[0] = 131072.0;
DeltaRobotKin.sNode[1].ulTrCoef[1] = 1.0/(131072.0);
DeltaRobotKin.sNode[2].ulTrCoef[0] = 131072.0;
DeltaRobotKin.sNode[2].ulTrCoef[1] = 1.0/(131072.0);

// Encoder offset - the caounts value at 0 angle
DeltaRobotKin.sNode[0].ulTrCoef[2] = Theta1OffsetCnt;
DeltaRobotKin.sNode[1].ulTrCoef[2] = Theta2OffsetCnt;
DeltaRobotKin.sNode[2].ulTrCoef[2] = Theta3OffsetCnt;

// Set parameters of the linear transformation of the
// N axis
DeltaRobotKin.sNode[3].ulTrCoef[0] = 10.0;      // A
DeltaRobotKin.sNode[3].ulTrCoef[1] = 1.0/(10.0);// B
DeltaRobotKin.sNode[3].ulTrCoef[2] = 5;         // C

GROUP.SetDeltaRobotKinematic(DeltaRobotKin);
```

Results are:



Cartesian space

Joint space

5.10.9.2 SCARA Robot kinematic

5.10.9.2.1 General

The SCARA acronym stands for Selective Compliance Assembly Robot.

In general, SCARAs are 4-axis robot arms. They are designed to imitate the action of a human arm and commonly used in pick-in-place, assembly and packaging applications.

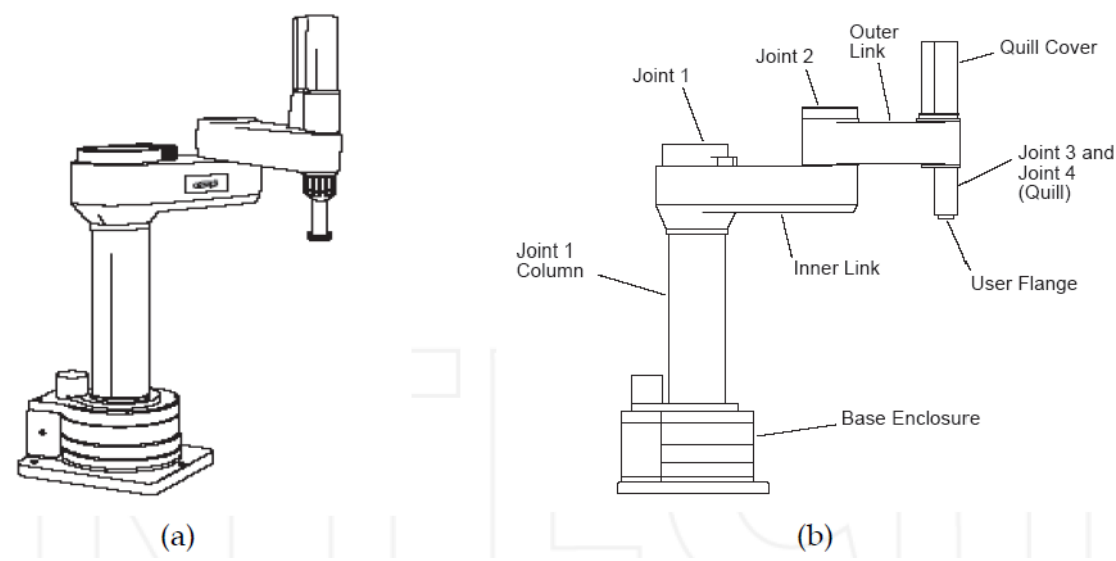


Figure 5-92 (a) SCARA robot and (b) Joints and links names

SCARA robot has 4 joints which are linked to the robot. Joint 3 is a translational joint which can move along Z-axis while joints 1, 2 and 4 are rotational joints.

First joint is the base joint and it is also called "the shoulder" as its function looks like a human shoulder. In this joint, the rotational movement of the inner link is provided.

Second joint is called "the elbow" as its function looks like a human elbow. In this joint, the outer link and the inner link are linked. The robot can be programmed to move like a human left or right arm ("Lefty" or "Righty").

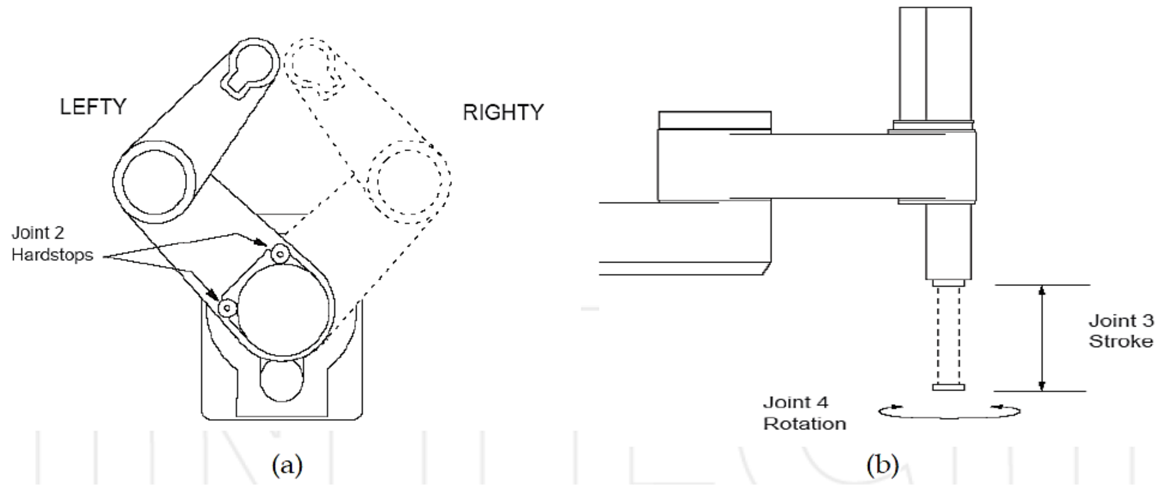


Figure 5-93 (a) 2nd joint, (b) 3rd and 4th joint movements

Third joint is a linear joint. It is placed at the end of the outer link.

Forth joint is also called "the wrist". Its function is similar to a human wrist and it can be rotated to tighten a bolt or unscrew a screw.

5.10.9.2.2 Inverse and Direct Kinematic

Kinematics problem consists of forward and inverse kinematics:

- Inverse kinematic is used to find joint variables (angles) of a robot end-effector from profiler position and orientation commands (X/Y/Z/W).
- Forward kinematics is used to find the position and orientation of a robot end effector as a function of its joint coordinates (A1/A2/A3/A4).

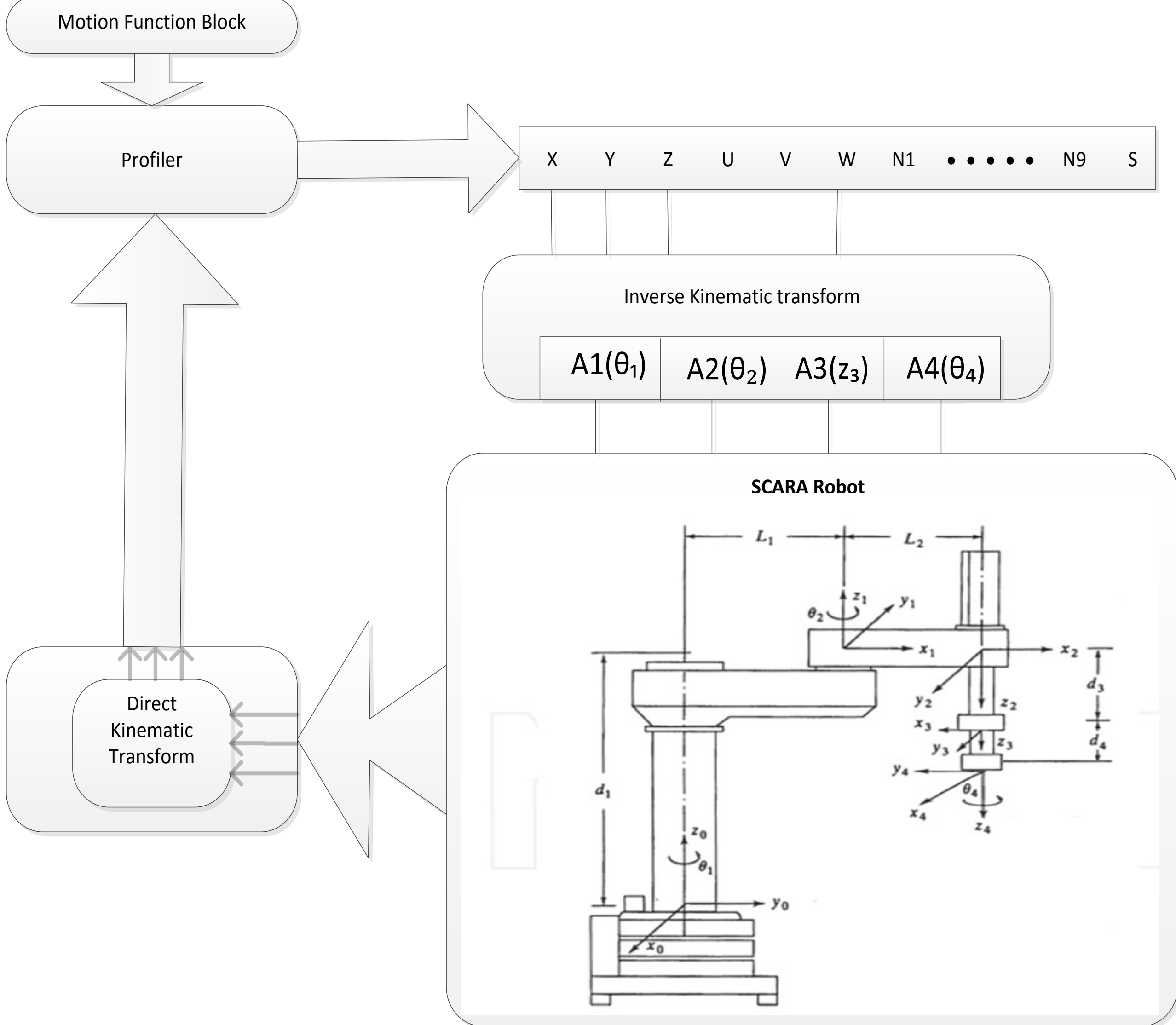


Figure 5-94: Defining Cartesian and Joint spaces

Maestro ACS coordinates must be defined as follows (see Figure 5-94):

- A1 = rotational joint #1 ("the shoulder");
- A2 = rotational joint #2 ("the elbow");
- A3 = linear joint #3;

- A4 = rotational joint #4 ("the wrist").

5.10.9.2.3 Interfaces

Here are the mechanical parameters which are used for the transformations (see Figure 6):

- Inner link length l_1 [mm];
- Outer link length l_2 [mm];
- Shoulder offset (column height) d_1 [mm];
- Offset between axis 3 an 4 in Z direction (wrist offset) d_3 [mm];
- Coefficient defining Z axis displacement as a result of the joint 4 rotation [mm/rad];
- Elbow sign (+1 for right-hand direction, -1 for left-hand direction).

5.10.9.2.4 Example - Set kinematic of SCARA robot

```
MMC_KINTRANSFORM_SCARA_IN stScara;
MMC_KINTRANSFORM_SCARA_OUT stOutParam;

// Set mechanical parameters [mm]
stScara.stParams.dInnerLinkLength = 430;
stScara.stParams.dOuterLinkLength =370;
stScara.stParams.dShoulderOffset = 800;
stScara.stParams.dWristOffset = 380 ;
stScara.stParams.dWristTheta2OffsetCoef = 0;
stScara.stParams.cElbowSign = 1 ;
// Set the number of axes
stScara.stParams.iNumAxes = 4 ;
//
stScara.eBufferMode = MC_BUFFERED_MODE;
stScara.ucExecute = 1;

// Assign the axes to kinematic directions

// Theta 1
stScara.stParams.sNode[0].eType = NC_ACS_A1_AXIS_TYPE;
stScara.stParams.sNode[0].hNode=0 ; //axis a01
stScara.stParams.sNode[0].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC ;

// Theta 2
stScara.stParams.sNode[1].eType = NC_ACS_A2_AXIS_TYPE;
stScara.stParams.sNode[1].hNode=1 ; //axis a2
stScara.stParams.sNode[1].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC ;

// Z 3
stScara.stParams.sNode[2].eType = NC_ACS_A3_AXIS_TYPE;
stScara.stParams.sNode[2].hNode=2 ; //axis a03
stScara.stParams.sNode[2].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC ;

// Theta 4
stScara.stParams.sNode[3].eType = NC_ACS_A4_AXIS_TYPE;
stScara.stParams.sNode[3].hNode=3 ; //axis a04
stScara.stParams.sNode[3].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC ;
```

```
//Set the encoder inputs

// Encoder to angle ratio
stScara.stParams.sNode[0].ulTrCoef[0]=102400.0 * 64 ;
stScara.stParams.sNode[0].ulTrCoef[1]=1.0 / (102400.0 * 64) ;
stScara.stParams.sNode[0].ulTrCoef[2]=0.0 ;
stScara.stParams.sNode[1].ulTrCoef[0]=81920.0 * 64 ;
stScara.stParams.sNode[1].ulTrCoef[1]=1.0 / (81920.0 * 64);
stScara.stParams.sNode[1].ulTrCoef[2]=0.0 ;
stScara.stParams.sNode[2].ulTrCoef[0]=10000.0 * 64 / 111.0 ;
stScara.stParams.sNode[2].ulTrCoef[1]=111.0 / (10000.0 * 64) ;
stScara.stParams.sNode[2].ulTrCoef[2]=0.0 ;
stScara.stParams.sNode[3].ulTrCoef[0]=30720.0 * 64 ;
stScara.stParams.sNode[3].ulTrCoef[1]=1.0 / (30720.0 * 64) ;
stScara.stParams.sNode[3].ulTrCoef[2]=0.0 ;

MMC_SetKinTransformScara(g_pHndl,256,&stScara,&stOutParam) ;
```

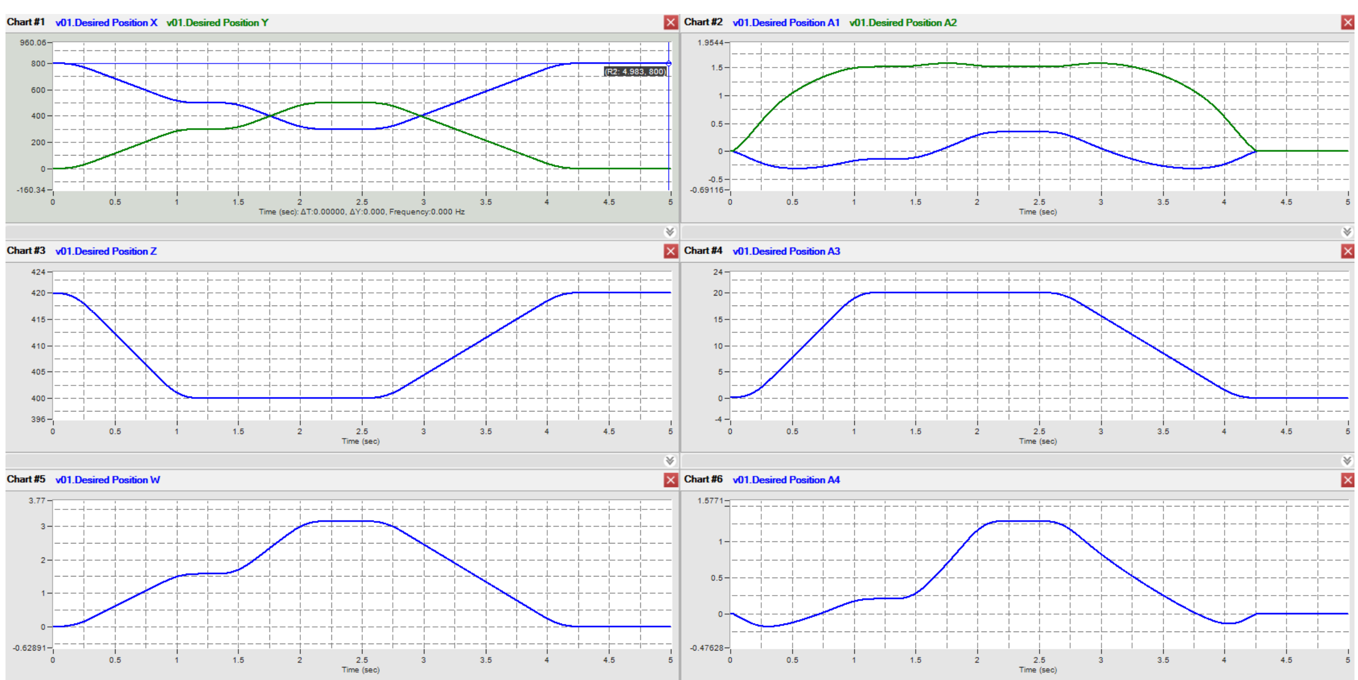


Figure 5-95 Set kinematic of SCARA robot Results

5.10.9.3 Three Link Robot Kinematic

5.10.9.3.1 General

The three link robot is similar to the SCARA robot. The difference is that its arm has additional link:

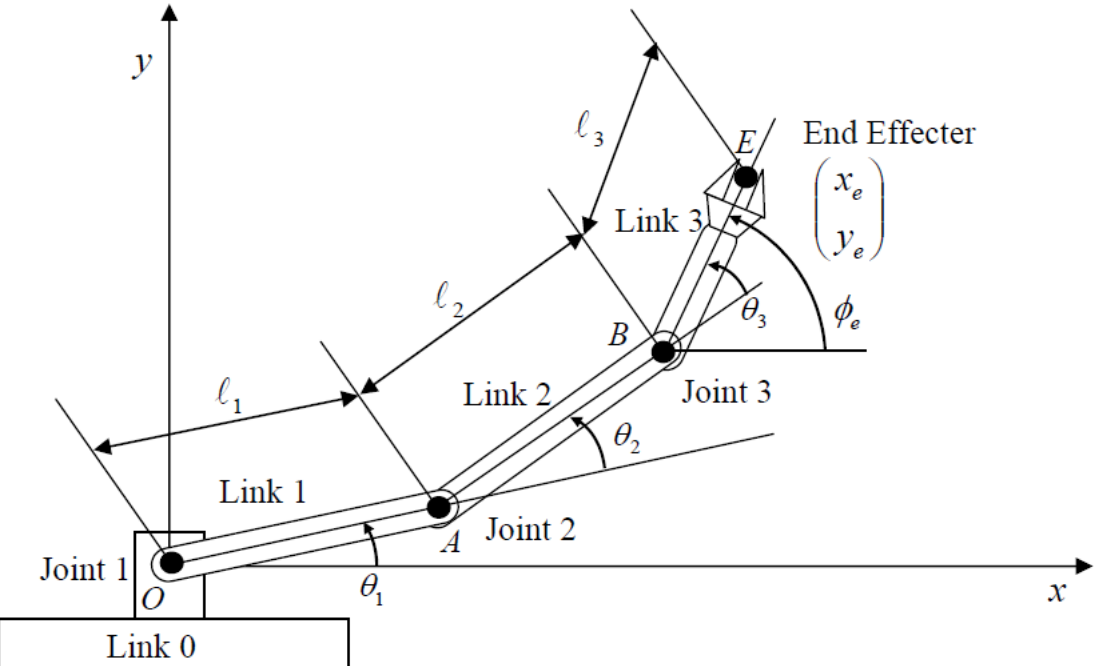


Figure 5-96 Three Link Robot

5.10.9.3.2 Inverse and Direct Kinematic

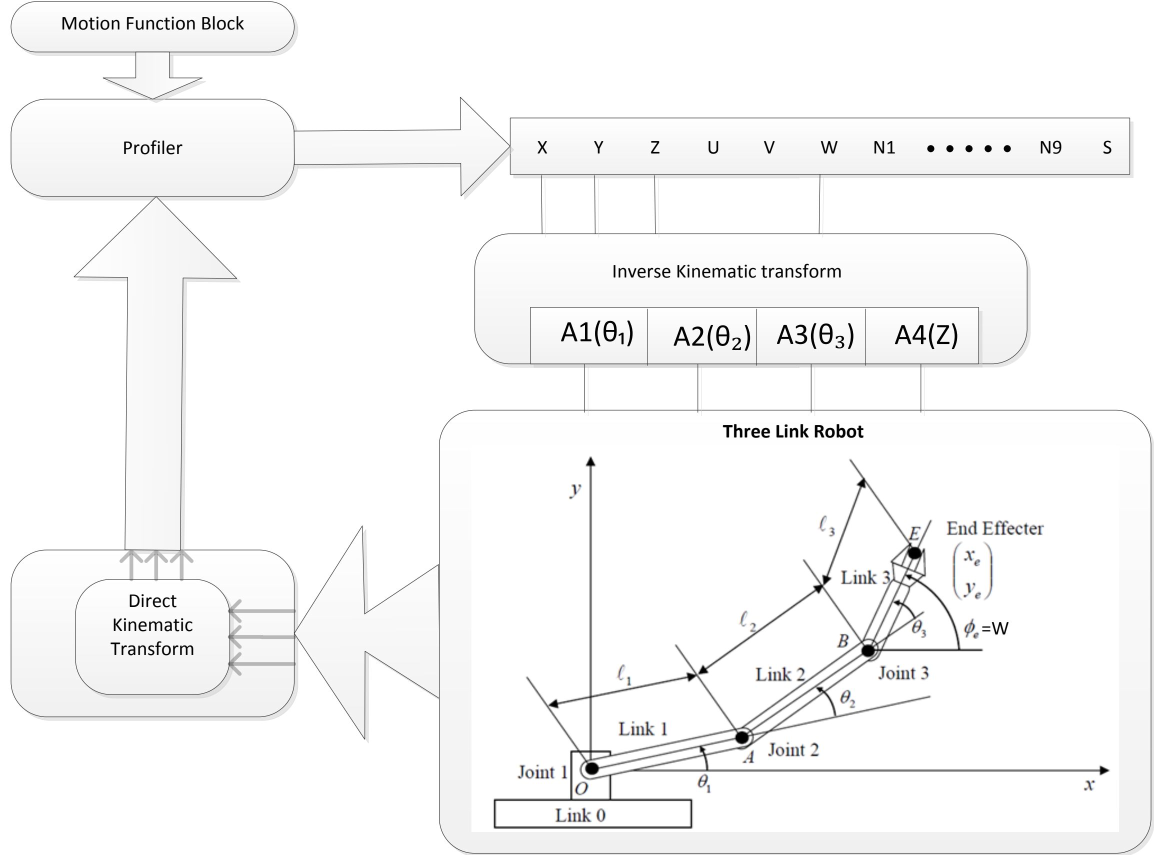


Figure 5-97 Three Link Robot kinematic

Maestro ACS coordinates must be defined as follows (see Figure 5-97):

- A1 = rotational joint #1 ("the shoulder");
- A2 = rotational joint #2 ("the elbow");
- A3 = rotational joint #3 ("the wrist") ;
- A4 = linear joint Z.

5.10.9.3.3 Interfaces

Here are the mechanical parameters which are used for the transformations (see Figure 5-97):

- Inner link length l_1 [mm];
- Central link length l_2 [mm];
- Outer link length l_3 [mm];
- Shoulder offset (column height) d_1 [mm];
- Offset between axis 3 an 4 in Z direction (wrist offset) d_3 [mm];
- Coefficient defining Z axis displacement as a result of the joint 4 rotation [mm/rad];
- Elbow sign (+1 for right-hand direction, -1 for left-hand direction).

5.10.9.3.4 Example - Set kinematic of Three Link robot

```
MMC_KINTRANSFORM_THREELINK_IN stThreeLink;
MMC_KINTRANSFORM_THREELINK_OUT stOutParam;

// Set mechanical parameters [mm]
stThreeLink.stParams.dInnerLinkLength = 430;
stThreeLink.stParams.dMediumLinkLength = 370;
stThreeLink.stParams.dOuterLinkLength = 100;
stThreeLink.stParams.dShoulderOffset = 800;
stThreeLink.stParams.dWristOffset = 380 ;
stThreeLink.stParams.dWristTheta2OffsetCoef = 0;
stThreeLink.stParams.cElbowSign = 1 ;
// Set the number of axes
stThreeLink.stParams.iNumAxes = 4 ;
//
stThreeLink.eBufferMode = MC_BUFFERED_MODE;
stThreeLink.ucExecute = 1;

// Assign the axes to kinematic directions

// Theta 1
stThreeLink.stParams.sNode[0].eType = NC_ACS_A1_AXIS_TYPE;
stThreeLink.stParams.sNode[0].hNode=0 ; //axis a01
stThreeLink.stParams.sNode[0].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC ;

// Theta 2
stThreeLink.stParams.sNode[1].eType = NC_ACS_A2_AXIS_TYPE;
stThreeLink.stParams.sNode[1].hNode=1 ; //axis a2
stThreeLink.stParams.sNode[1].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC ;

// Theta 3
stThreeLink.stParams.sNode[2].eType = NC_ACS_A4_AXIS_TYPE;
stThreeLink.stParams.sNode[2].hNode=2 ; //axis a03
stThreeLink.stParams.sNode[2].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC ;

// Z
stThreeLink.stParams.sNode[3].eType = NC_ACS_A3_AXIS_TYPE;
stThreeLink.stParams.sNode[3].hNode=3 ; //axis a04
stThreeLink.stParams.sNode[3].iMcsToAcsFuncID = NC_TR_SHIFT_FUNC ;
```

```
//Set the encoder inputs

// Encoder to angle ratio
stThreeLink.stParams.sNode[0].ulTrCoef[0]=102400.0 * 64 ;
stThreeLink.stParams.sNode[0].ulTrCoef[1]=1.0 / (102400.0 * 64) ;
stThreeLink.stParams.sNode[0].ulTrCoef[2]=0.0 ;
stThreeLink.stParams.sNode[1].ulTrCoef[0]=81920.0 * 64 ;
stThreeLink.stParams.sNode[1].ulTrCoef[1]=1.0 / (81920.0 * 64);
stThreeLink.stParams.sNode[1].ulTrCoef[2]=0.0 ;
stThreeLink.stParams.sNode[2].ulTrCoef[0]=30720.0 * 64 ;
stThreeLink.stParams.sNode[2].ulTrCoef[1]=1.0 / (30720.0 * 64) ;
stThreeLink.stParams.sNode[2].ulTrCoef[2]=0.0 ;
stThreeLink.stParams.sNode[3].ulTrCoef[0]=10000.0 * 64 / 111.0 ;
stThreeLink.stParams.sNode[3].ulTrCoef[1]=111.0 / (10000.0 * 64) ;
stThreeLink.stParams.sNode[3].ulTrCoef[2]=0.0 ;

MMC_SetKinTransformThreeLink(g_pHndl,256,&stThreeLink,&stOutParam) ;
```

TCP (tool center point) motion example:

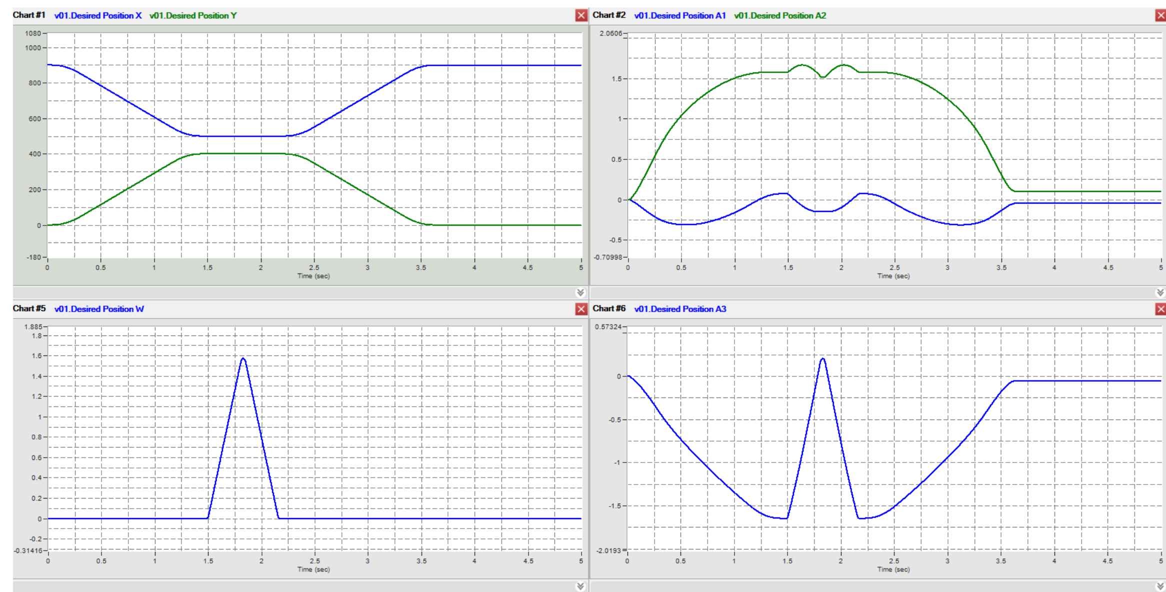


Figure 5-98 Set Kinematic of Three Link Robot Results

5.10.10 Error IDs

-330	Kinematic type out of range	When you set kinematic transformation using "MMC_SetKinTransformex" function, you can set one of the following kinematic types: * NC_CARTESIAN_TYPE = 0 * NC_DELTA_ROBOT_TYPE = 1 * NC_SCARA_ROBOT_TYPE = 2 * NC_THREE_LINK_ROBOT_TYPE = 3 The kinematic types can be found in the enum definition "NC_KIN_TYPE" at "MMC_PLCopen_group_API.h" file.
-479	SCARA Robot: inverse kinematics failed	Please ensure that mechanical inputs of the SCARA robot configuration are properly set and target position is inside the working envelope. Note that the mechanical inputs and the target position should be inserted in [mm] units.
-480	SCARA Robot: direct kinematics failed	Please ensure that the homing procedure of the SCARA robot is properly done. Note that the initial angles of the "Shoulder" and "Elbow" joints must be in the range +/-PI radians.
-481	SCARA Robot: direct kinematics failed	Please ensure that the homing procedure of the SCARA robot is properly done. Note that the initial angle of the "Elbow" joint should match the cElbowDirection input of the SCARA robot configuration.
-482	SCARA Robot: one of the kinematic directions is missing	When you set configuration for SCARA robot kinematics, you should map all kinematic directions to the real axes: * "the shoulder" theta - NC_ACS_A1_AXIS_TYPE * "the elbow" theta - NC_ACS_A2_AXIS_TYPE * "the wrist" stroke - NC_ACS_A3_AXIS_TYPE * "the wrist" theta - NC_ACS_A4_AXIS_TYPE
-483	SCARA Robot: one of mechanical inputs is non-legal	When you set configuration for delta robot kinematics, you should enter the mechanical parameters of the robot. Inner and outer link lengths should be positive (>0). Note that the mechanical inputs and the target position should be inserted in [mm] units.
-484	Three Link Robot: inverse kinematics failed	Please ensure that mechanical inputs of the Three Link robot are properly set and target position is inside the working envelope. Note that the mechanical inputs and the target position should be inserted in [mm] units.

-485	Three Link Robot: direct kinematics failed	Please ensure that the homing procedure of the Three Link robot is properly done. Note that the initial angles of the robot joints must be in the range +/-PI radians.
-486	Three Link Robot: direct kinematics failed	Please ensure that the homing procedure of the Three Link robot is properly done. Note that the initial angle of the "Elbow" joint should match the cElbowDirection input of the Three Link robot.
-487	Three Link Robot: one of the kinematic directions is missing	When you set configuration for Three Link robot kinematics, you should map all kinematic directions to the real axes: * "the shoulder" theta - NC_ACS_A1_AXIS_TYPE * "the elbow" theta - NC_ACS_A2_AXIS_TYPE * "the wrist" theta - NC_ACS_A3_AXIS_TYPE * "the wrist" stroke - NC_ACS_A4_AXIS_TYPE
-488	Three Link Robot: one of mechanical inputs is non-legal	When you set configuration for Three Link robot kinematics, you should enter the mechanical parameters of the robot. Inner, medium and outer link lengths should be positive (>0). Note that the mechanical inputs and the target position should be inserted in [mm] units.
-489	Inappropriate state for MMC_SetCartesianTransform function	The MMC_SetCartesianTransform function can be operated only in one of the following states: 1. GROUP_DISABLED. 2. GROUP_STANDBY.
-490	Inappropriate state for MMC_TrackConveyorBelt function	The MMC_TrackConveyorBelt function can be operated only in one of the following states: 1. GROUP_DISABLED. 2. GROUP_STANDBY.
-491	Inappropriate state for MMC_TrackRotaryTable function	The MMC_TrackRotaryTable function can be operated only in one of the following states: 1. GROUP_DISABLED. 2. GROUP_STANDBY.
-492	MCS to PCS transformation type is out of range or not defined	The following transformation types are supported: * NC_PCS_MCS_TRANSFORM_STATIC = 1, * NC_PCS_MCS_CONVEYOR_BELT = 2, * NC_PCS_MCS_ROTARY_TABLE = 3

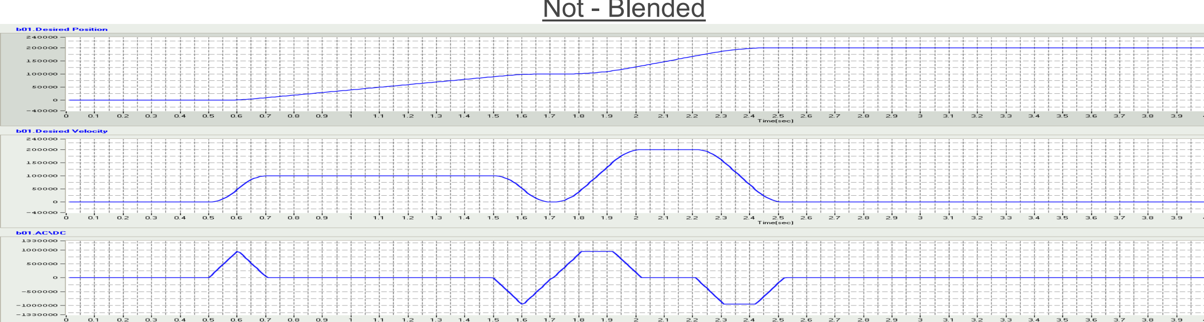
5.11 Multiple Axes Motion Control - Transition and Buffer Modes

The basic transition modes are defined as follows. Other modes as well as supplier specific modes can be added.

No.	MC_TransitionMode	Description
0	MC_TM_NONE_MODE	Insert no transition curve (default mode)
1	MC_TM_MAX_VELOCITY_MODE	Transition with given start velocity. This is not supported at this time.
2	MC_TM_DEFINED_VELOCITY_MODE	Transition with given constant velocity
3	MC_TM_CORNER_DISTANCE_MODE	Transition with given corner distance
4	MC_TM_MAX_CORNER_DEVIATION_MODE	Transition with given maximum corner deviation
5	MC_TM_SWITCH_RADIUS_MODE	Transition with given radius of the circle arc
7	MC_TM_CORNER_DIST_CV_POLYNOM3	Refer to the sections below, for a detailed explanation.
8	MC_TM_CORNER_DIST_CV_POLYNOM5	Refer to the sections below, for a detailed explanation.
9	MC_TM_CORNER_DEVIATION_MODE_PLN6	Refer to the sections below, for a detailed explanation.
10	MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES	Refer to the sections below, for a detailed explanation.
11--...	Supplier specific modes	

5.11.1 Single Axis Buffer Modes

These transitions can only be applied in NC mode with Blending in a forward direction. The Input parameters are set using a Buffer mode, but no blending occurs when the previous function block is presently active.



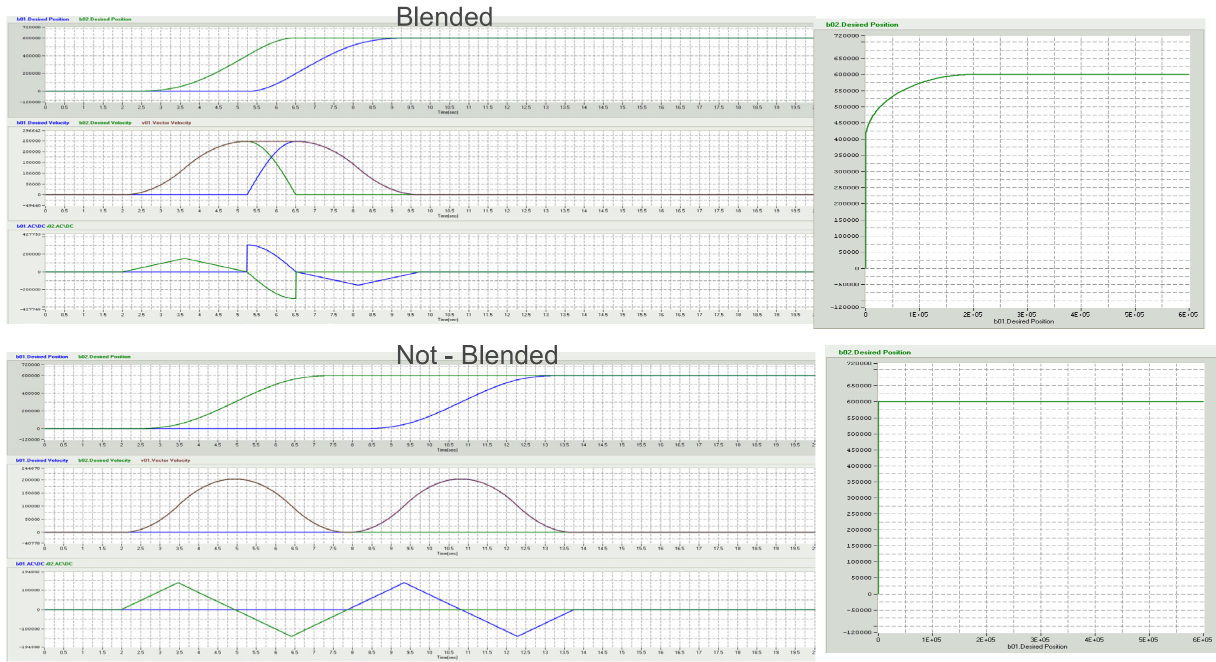
5.11.2 Multi-Axes Transitions

Transitions for the multi-axes are only supported in the following kinematic modes:

- MCS and PCS, Cartesian system – all types.
- MCS and PCS, N axes – all types.
- ACS – only one type "MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES".

The transition mode is not supported when the geometry of the motion is opposite (180°). The Input is set in Buffer mode, and the Transition mode uses the appropriate special Transition parameter. When using a special polynomial transitions, the “S” parameter is used.

Similar to the Single axis transitions, no blending is allowed with transitions when the previous function block is active. However, in Multi-axes transitions, the geometry of the transition curve is defined by the Transition mode. The geometry of the transition involves the use of another function block whose kinematics (velocity) is defined by a Buffer mode, and as a pre-condition, the use of MMC_SetKinTransform.



5.11.3 Matrix of available Transition Modes

The matrix in the table below describes the available transition modes for the different buffer modes, and can be used by the supplier to document its supported transition modes.

The matrix of the available transition modes applies to the following transitions:

Line-line — Circle-line

Line-circle — Circle-circle

TransitionMode > All BufferModes	0	1	2	3	4	5	7	8	9	10
Aborting	A	N	N	N	N	N	N	N	N	N
Buffered	A	N	N	N	N	A	N	N	N	N
Blending Low	N	N	A	A	A	A	A	A	A	A
Blending Previous	N	N	A	A	A	A	A	A	A	A
Blending Next	N	N	A	A	A	A	A	A	A	A
Blending High	N	N	A	A	A	A	A	A	A	A

Legend:

Transition Modes 0 – 10 Refer to the table in section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes.

A = Available, N = Not supported

The Input parameter Transition parameter is a double data type with various connotations for each transition type. When buffered or in Abortion mode, the value of this parameter is not used. It is also limited by 45% of the shortest segment between two connected consecutive function blocks.

5.11.3.1 Transition Parameter Definition

When **Defined_Velocity_Mode** is used, the Transition Parameter (TP) is measured in % of the function block's defined velocity. The radius of the Arc is calculated using speed and vector acceleration of the current vector according to:

R = V^2/Acc

Where **V** is the velocity defined by the Blended mode x % of TP

And **Acc** is the maximum vector acceleration, defind by **MAX_AC.**

When the **Switch_Radius_Mode** is used, the TP defines the radius of the Arc that connects the two function blocks as shown in **Figure 5-92**.

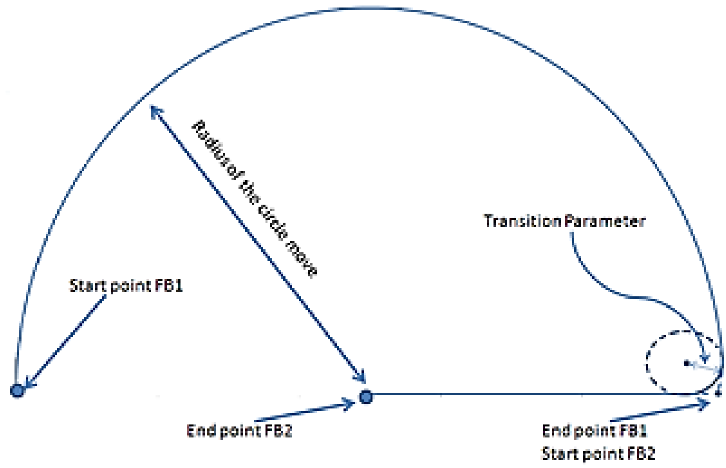


Figure 5-99: Transition Parameter defined for Switch Radius

When the **Corner_Deviation_Mode /Polynom6**_Mode is used, the TP defines the minimum distance between the Arc and the buffered transition point as shown in **Figure 5-93** (End point of FB1 and Start point of FB2 in case of Buffered mode).

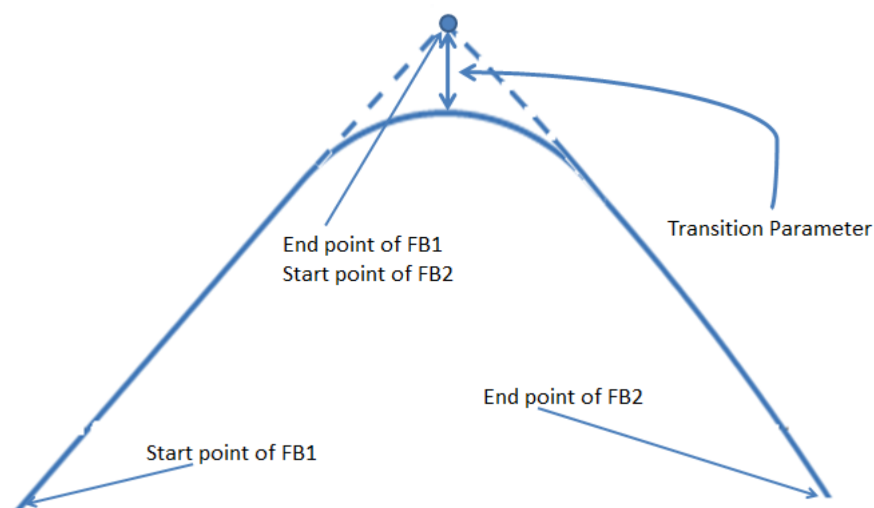


Figure 5-100: Transition Parameter defined for Corner_Deviation_Mode/Polynom6_Mode

When the **Corner_Distance_Mode\Polynom3_Mode\Polynom5_[NAXES]Mode** is used, the TP defines the trajectory distance between the start of the Arc and the transition point (buffered point) as shown in **Figure 5-94** (End point of FB1 and Start point of FB2 in case of Buffered mode).

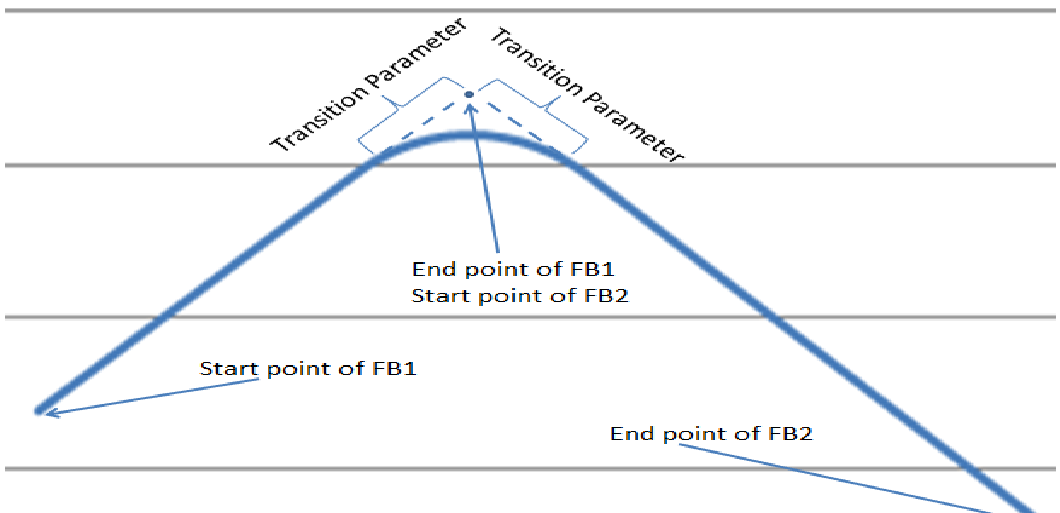


Figure 5-101: Transition Parameter defined for Corner_Distance_Mode\Polynom3_Mode\Polynom5_[NAXES]Mode

5.11.3.2 Transition in 2D and 3D

- In 2D space all transition modes can be used.
- In 3D space modes 7-10 can be used by default, whereas modes 1-5 can be used only in the situations where two neighboring shapes are located on same 2D shape. Otherwice the transition mode will be changed internally to mode "8".

5.11.3.3 Transition parameter limitation

No matter what transition mode is finally used, it predefines some corner distance. To avoid transition curves intersecting, the Maestro limits the corner distance to 40% of the original segment length for the transition modes 1-5 and to 45% for the modes 6 - 10.

5.11.3.4 Arc Transitions - Transition modes (1 – 5)

This modes describes a transition curve built as a circle arc. The constant velocity is originally defined by the blend mode but can be decreased if cannot be achieved due to trajectory parameters. It can be also decreased before the transition curve if the predefined velocity causes at some points of the transition curve acceleration greater than the one defined by the parameter MMC_MAX_ACCELERATION_PARAM. This acceleration is calculated as $a = v^2/\rho$ where ρ is a radius of curvature.

This transition mode guarantees the transition is smoothed by the position, but produces a minor jump via the velocity and acceleration at start and end points of the transition curve.

5.11.3.5 Polynomial Transitions

5.11.3.5.1 Transition mode MC_TM_CORNER_DIST_CV_POLYNOM3 = 7

Note: **The Transition parameter must be defined as a corner distance, where the transition is executed with the constant velocity**.

This mode describes a transition curve built as a cubic polynomial of some parameter "S". Initial estimation of the parameter variation is based on the distance between start and end points with the objective to approach the desired curve length.

The constant velocity is originally defined by the blend mode but can be decreased if cannot be achieved due to trajectory parameters. It can be also decreased before the transition curve if the predefined velocity causes at some points of the transition curve acceleration greater than the one defined by the parameter MMC_MAX_ACCELERATION_PARAM. This acceleration is calculated as $a = v^2/\rho$ where ρ is a radius of curvature.

This transition mode guarantees the transition is smoothed by the velocity, but produces a minor jump via acceleration at start and end points of the transition curve. If the acceleration achieved at some points of the transition curve is less than one predefined by the parameter MMC_MAX_ACCELERATION_PARAM but greater than desirable, the user can decrease the value of the parameter, causing a decrease of the transition velocity by the formula
$V = [MMC_MAX_ACCELERATION_PARAM * \rho_{min}]^{1/2}$
where ρ_{min} is the minimal radius of curvature on the transition curve.

This mode is recommended in some issues with the mode 8, e.g. detrimental geometry.

5.11.3.5.2 Transition mode MC_TM_CORNER_DIST_CV_POLYNOM5 = 8

Note: **The Transition parameter must be defined as a corner distance, where the Transition is executed with the constant velocity.**

This mode the transition curve is constructed as a quintic polynomial of some parameter "S". The initial estimation of the parameter variation is based on the distance between start and end-points with the aim to approach the desired curve length.

This velocity originally defined by the blend mode, can be decreased if due to the trajectory parameters, it cannot be achieved. It may also be decreased before the transition curve is built, if the predefined velocity at some points of the transition curve, causes acceleration greater than the one defined by the parameter MMC_MAX_ACCELERATION_PARAM. This acceleration is calculated as $a = v^2/\rho$ where ρ is a radius of curvature.

This transition mode guarantees transition smoothed by the velocity and acceleration. If the acceleration achieved at some points of the transition curve is less than one predefined by the parameter MMC_MAX_ACCELERATION_PARAM but greater than desirable, the user can decrease the value of the max acceleration parameter causing a decrease of the transition velocity by the formula
$V = [MMC_MAX_ACCELERATION_PARAM * \rho_{min}]^{1/2}$
where ρ_{min} is the minimal radius of curvature on the transition curve.

This mode is recommended for 3D transitions, and for 2D transitions if the inevitable acceleration jumps for the transition by the circle arc are unacceptable.

5.11.3.5.3 Transition mode MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9

Note: The Transition parameter must be defined as a corner deviation, where the maximum distance between the intersection point and the nearest point belongs to the transition curve.

This mode transition curve is constructed as a sextic polynomial of some parameter “S”. Initial estimation of the parameter variation is based on the distance between start and end points with the aim to approach the desirable curve length.

The Transition is executed with constant velocity. This velocity is initially defined by the blend mode but can be decreased if due to the trajectory parameters, it cannot be achieved. It may also be decreased before the transition curve is built, if the predefined velocity at some points of the transition curve, causes acceleration greater than the one defined by the parameter MMC_MAX_ACCELERATION_PARAM. This acceleration is calculated as $a = V^2/\rho$ where ρ is a radius of curvature.

In this mode, the user also defines the corner distance with the use of the parameter MMC_S_FACTOR. The Corner distance is calculated as
(1 + MMC_S_FACTOR)*TRANSITION_PARAM

where TRANSITION_PARAM defined in the function block, must be equal to the required corner deviation. These two parameters must be reasonably coordinated to achieve desired geometry. In some cases, this may require experimentation.

This transition mode guarantees a transition smoothed by the velocity and acceleration. If, at some points, the acceleration achieved is less than one predefined by the parameter MMC_MAX_ACCELERATION_PARAM but greater than desirable, the user can decrease the value of the maximum acceleration parameter causing a decrease of the transition velocity, using the formula
$V = [MMC_MAX_ACCELERATION_PARAM * \rho_{min}]^{1/2}$
where ρ_{min} is the minimal radius of curvature of the transition curve.

This mode is recommended for 3D transitions, and for 2D transitions if the inevitable acceleration jumps for the transition by the circle arc are unacceptable. It should be noted that this is the only possible mode for the corner deviation in case of a transition between two shapes that do not belong to the same coordinate plane or some plane parallel to coordinate planes XY, XZ, or YZ.

On the other hand, transition mode MC_TM_MAX_CORNER_DEVIATION_MODE = 4 can only be used for two shapes belonging to the same coordinate plane or any two lines intersection in 3D.

5.11.3.5.4 Transition mode MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10

Until now, in order to perform blended motions, they had to be part of an X-Y-Z Cartesian system, and the user had to work in MCS, and select the Kinematic direction for each axis. However, for systems with no Kinematic directions using the ACS coordinate system, only one blended mode can be performed.

A new transition type was introduced; MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10. In this mode the TR is defined as in Corner Distance Mode, and fifth order polynom are used, similarly to the mode MC_TM_CORNER_DIST_CV_POLYNOM5.

5.11.4 Polynomial transition curve vs circle arc

Transition by the circle arc produces the most natural geometry but has two critical drawbacks:

- Such transition is only possible when two neighbor shapes belong to the same 2D plane.
- Acceleration discontinuity. At start point, there is an acceleration jump V^2/R (where R is the radius of the circle) when moving along the line with constant velocity. The same jump can occur at the end-point of the switch arc.

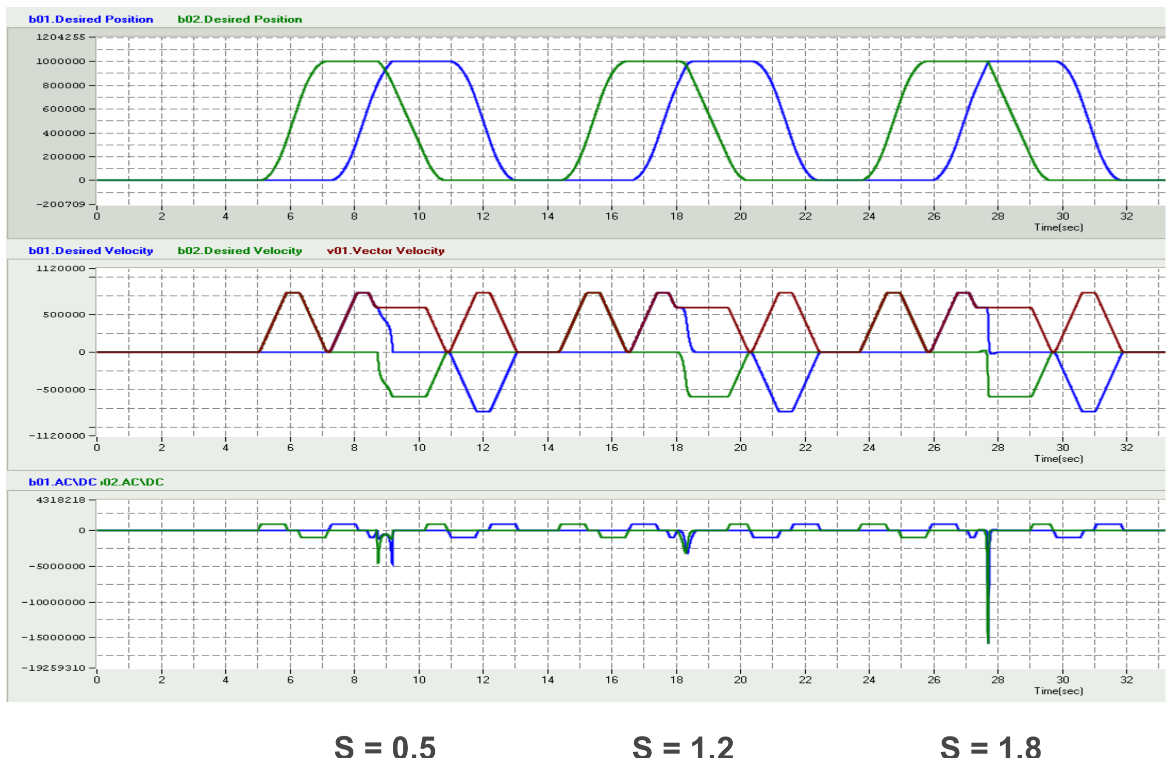
The disadvantage of the polynomial is a less predictable geometry. In the cases of modes 7 and 8, they can be influenced by the parameter MMC_S_FACTOR. When this parameter increases, the curve length increases and approaches the initial intersection geometry, causing a small radius of curvature and high acceleration. When this parameter decreases, we approach the line connecting the start and end-points of the transition curve, with the resulting low acceleration. The default value of the MMC_S_FACTOR parameter is 1.4. The user can define some value of MMC_S_FACTOR different from the default if the original transition curve does not satisfy his requirements.

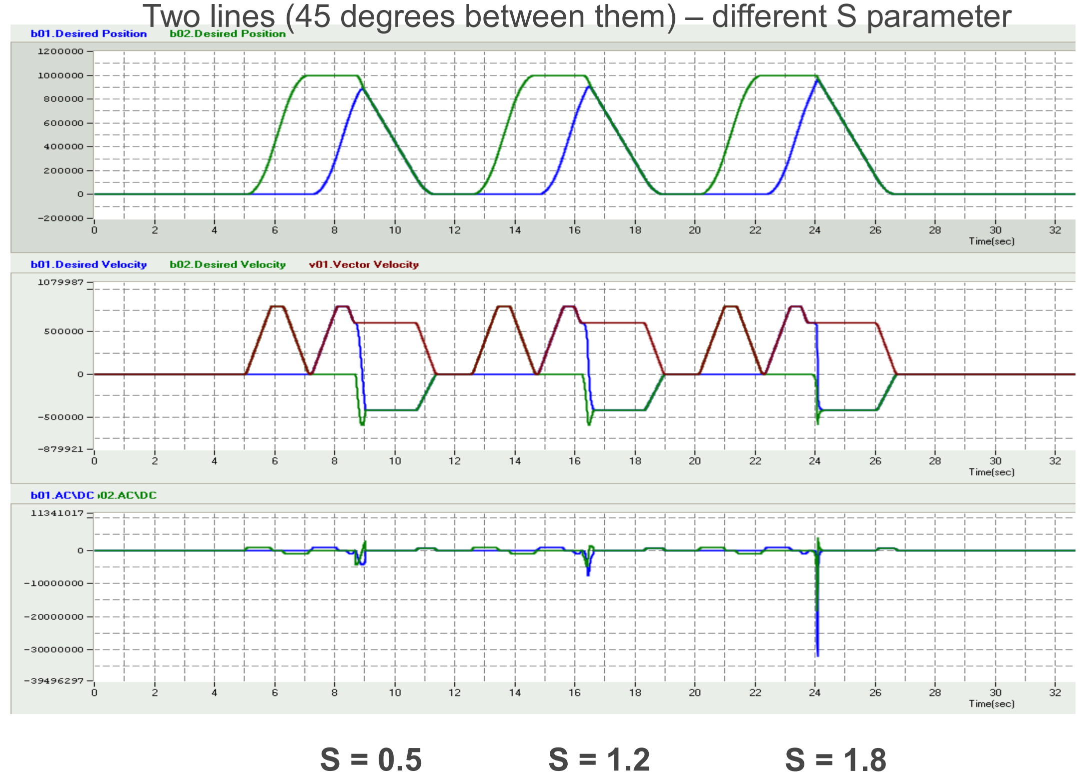
Therefore, in the case of a polynomial, we can use two parameters to define the intersection geometry TRANSITION_PARAMETER, which in this polynomial situation refers to the corner distance, and MMC_S_FACTOR that influences the length of the transition curve. In case of transition by the circle arc, the TRANSITION_PARAMETER (corner distance, corner deviation, switch radius) uniquely defines the intersection geometry.

5.11.5 The MMC_S_FACTOR Parameter

The MMC_S_FACTOR parameter is a Maestro group related parameter which is only relevant for polynomial transitions. The Transition length is proportional to the MMC_S_FACTOR Parameter (CurveLength ~ S) with a range [0…2]. The optimal MMC_S_FACTOR parameter causes a minimum AC\DC on transition, does not cross the buffering point, and varies according to the angle at the connection buffered point.

Two lines (90 degrees between them) – different S parameter





5.11.6 Transition Inputs

The following table displays an application of the Buffered mode with the Transition mode.

Buffer mode	Transition Mode application
Aborted Buffered	None
Blending Low Blending Previous Blending Next Blending High	Defined velocity Switch Radius Corner deviation Corner deviation (Polynom 6) Corner distance Corner distance (Polynom 3) Corner distance (Polynom 5) Corner distance (Polynom 5) NAXES

5.11.7 Buffer modes

The MC_BufferMode defines the velocity at the transition point. When **Aborting** is applied:

- All existing function blocks in the node queue are aborted
- The Transition velocity changes to zero, which in turn stops all current motion, and executes the new motion from zero velocity

When **Buffering** is applied, the current function block is inserted to the function block queue in the Maestro, and on completion of the axis\vector execution of all function blocks in the queue (with the except of the current function block), then the current function block will start its execution. In addition, the Transition velocity is zero.

When **Blending Low** is applied, the current function block is inserted to the function block queue in the Maestro, and on completion of the axis\vector execution of all function blocks in the queue (with the except of the current function block), then the current function block will start its execution. However, the Transition velocity is the smaller value between the previous and current function blocks.

When **Blending Previous** is applied, the current function block is inserted to the function block queue in the Maestro, and on completion of the axis\vector execution of all function blocks in the queue (with the except of the current function block), then the current function block will start its execution. However, the Transition velocity is the velocity of the previous function block.

When **Blending Next** is applied, the current function block is inserted to the function block queue in the Maestro, and on completion of the axis\vector execution of all function blocks in the queue (with the except of the current function block), then the current function block will start its execution. However, the Transition velocity is the velocity of the current function block.

When **Blending High** is applied, the current function block is inserted to the function block queue in the Maestro, and on completion of the axis\vector execution of all function blocks in the queue (with the except of the current function block), then the current function block will start its execution. However, the Transition velocity is the higher between the previous and the current function blocks.

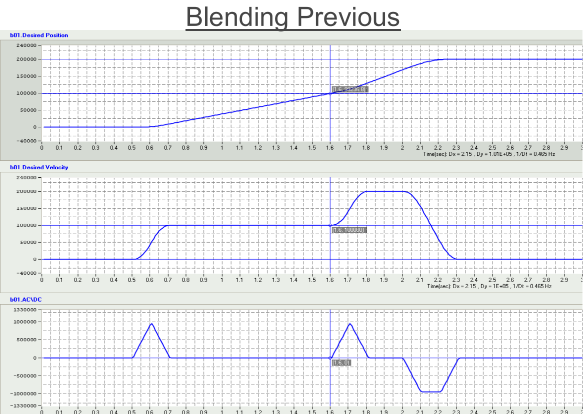
Example (Single Axis)

FB1 – velocity = 100K

FB2 – velocity = 200K

Blending High

Blending Previous

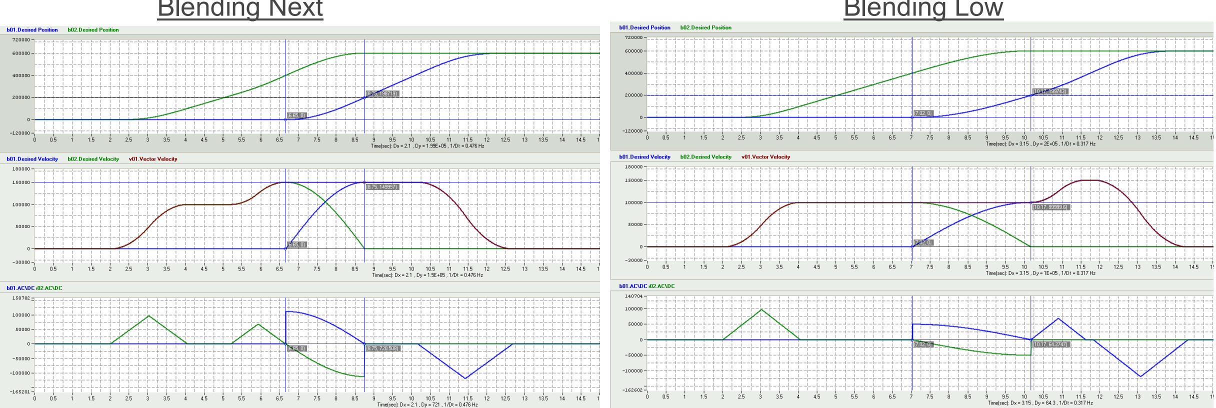


Example (Multi Axis)

FB1 – velocity = 100K

FB2 – velocity = 150K

Blending Next　　　　Blending Low



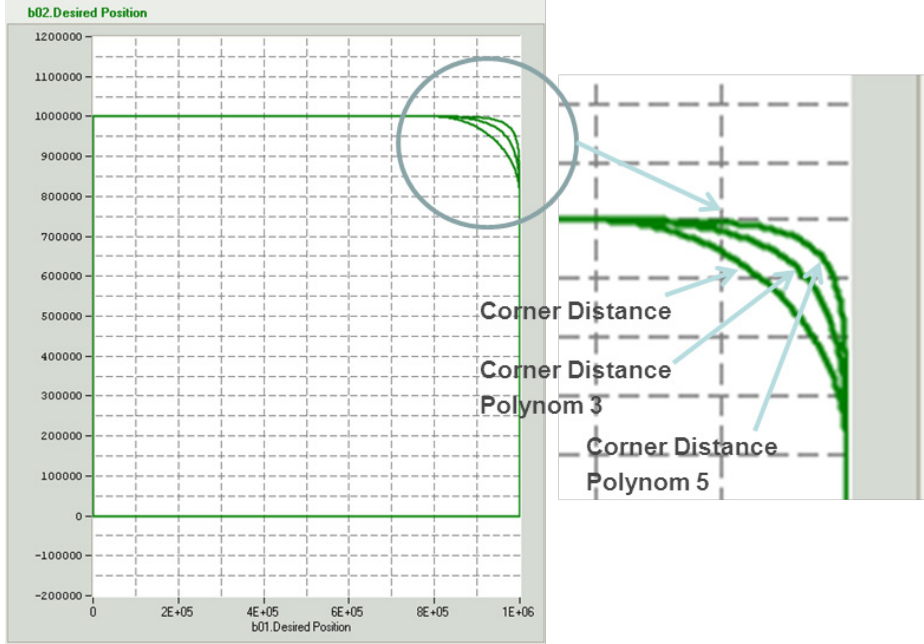
5.11.8 Selecting Polynomial or Simple Transition

The following table describes the advantages and disadvantages of each transition mode.

Transition	Advantage	Disadvantage
Simple	Smooth Position Strongly defined geometry (circle shape) Low AC\DC on transition	Uneven velocity on transition point Uneven AC\DC (current) on transition point
Corner distance Polynom3 – third order	Smooth Position Smooth Velocity	Uneven AC\DC (current) on transition point. Vaguely defined geometry (S parameter), shape limit. High AC\DC on transition
Corner distance Polynom5 – fifth order	Smooth Position Smooth Velocity Smooth AC\DC (current)	Vaguely defined geometry (S parameter), shape limit. Very high AC\DC on transition
Corner deviation Polynom6 – sixth order	Smooth Position Smooth Velocity Smooth AC\DC (current) The deviation point is the closest points to buffered point	Vaguely defined geometry (S parameter) start point of transition. Very high AC\DC on transition.

5.11.9 2D and 3D Transition Examples

The following is a 2D simple example, containing two lines with 90degrees between them:



The sequence is:

1. Start from 0,0
2. Go to 0,1M
3. Go to 1M,1M
4. Go to 1M,0 (Blended)
5. Go to 0,0

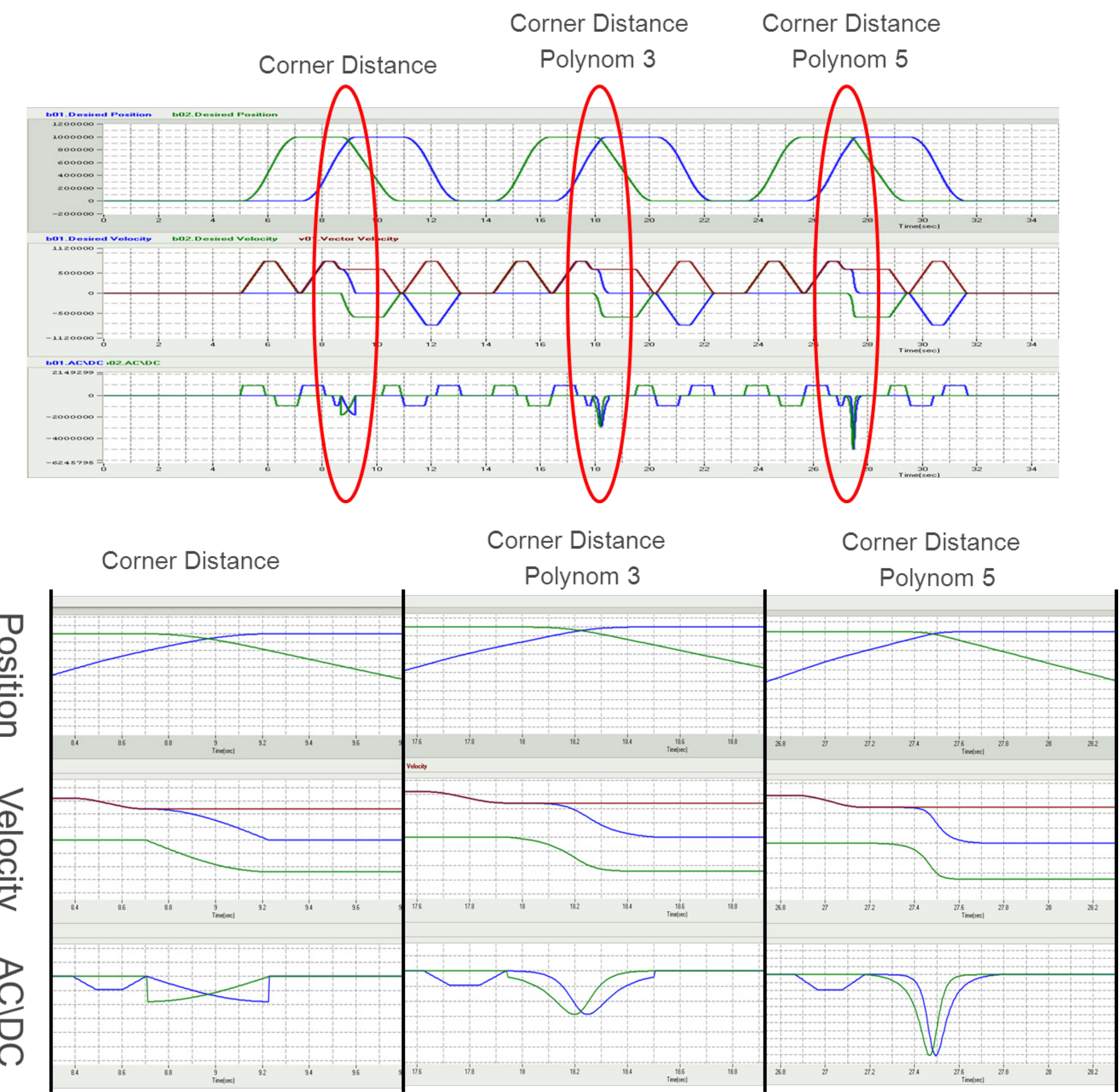
This sequence performed three times.

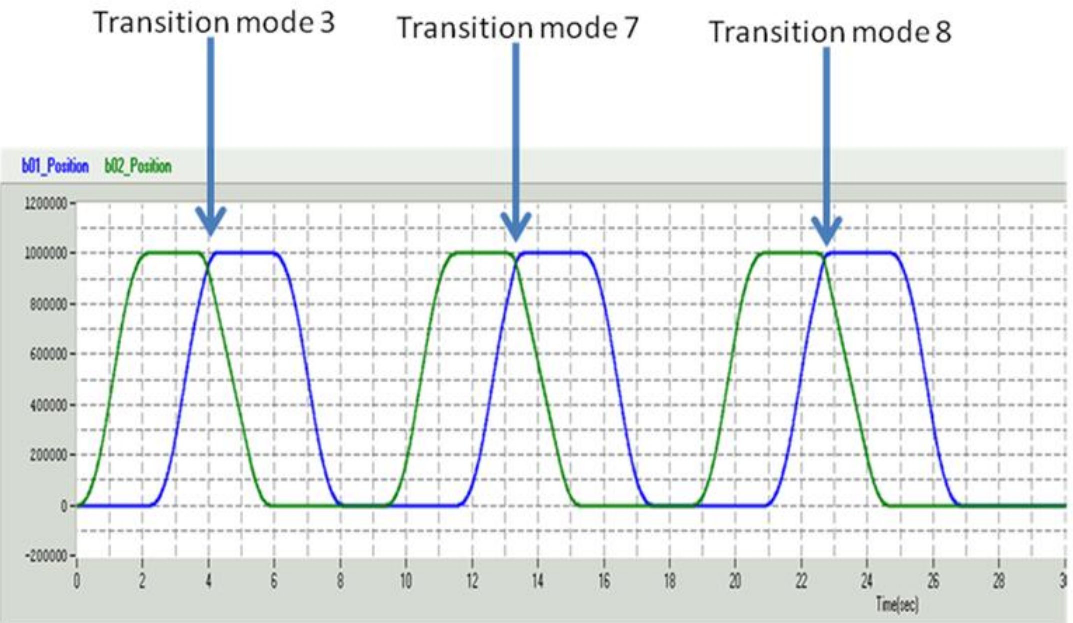
- First time with Transition mode “3”.
- Second time with Transition mode “7”.
- Third time with Transition mode “8”.

NOTE: **All the sequences are performed with the same transition parameter (S=1.4).**

By viewing the position plots alone, it is not possible to indicate any significant difference between them. It is only possible to say that at different corner distance mode (simple, Polynom 3 and 5) the shape of the curve varies.

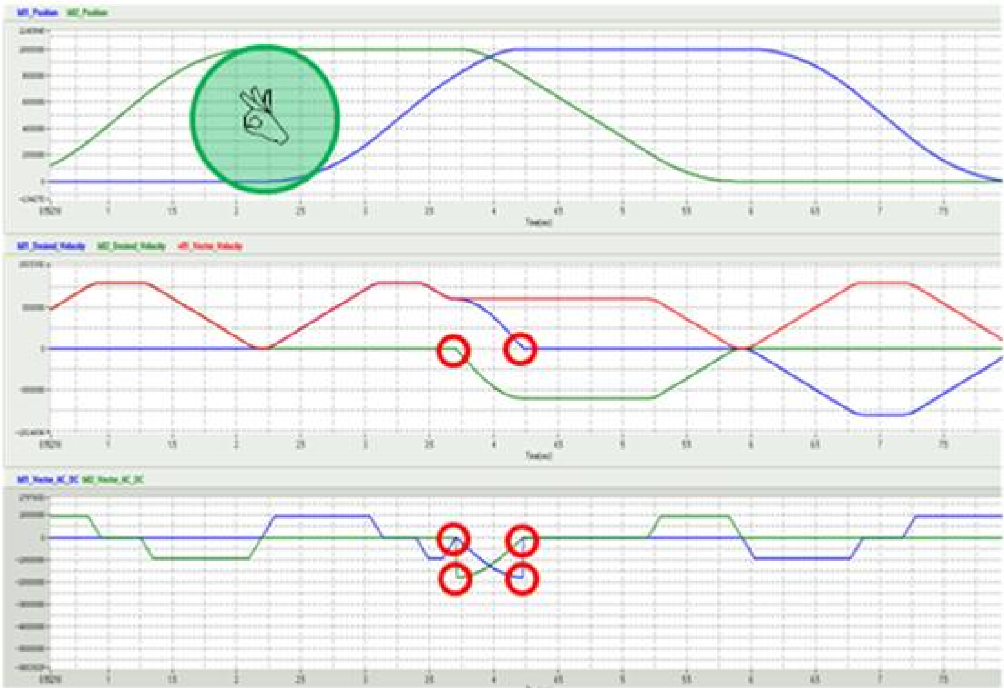
The following is a comparison between three optional methods to apply the corner distance transition modes, and involves a transition between two linear X,Y motions on the plane.



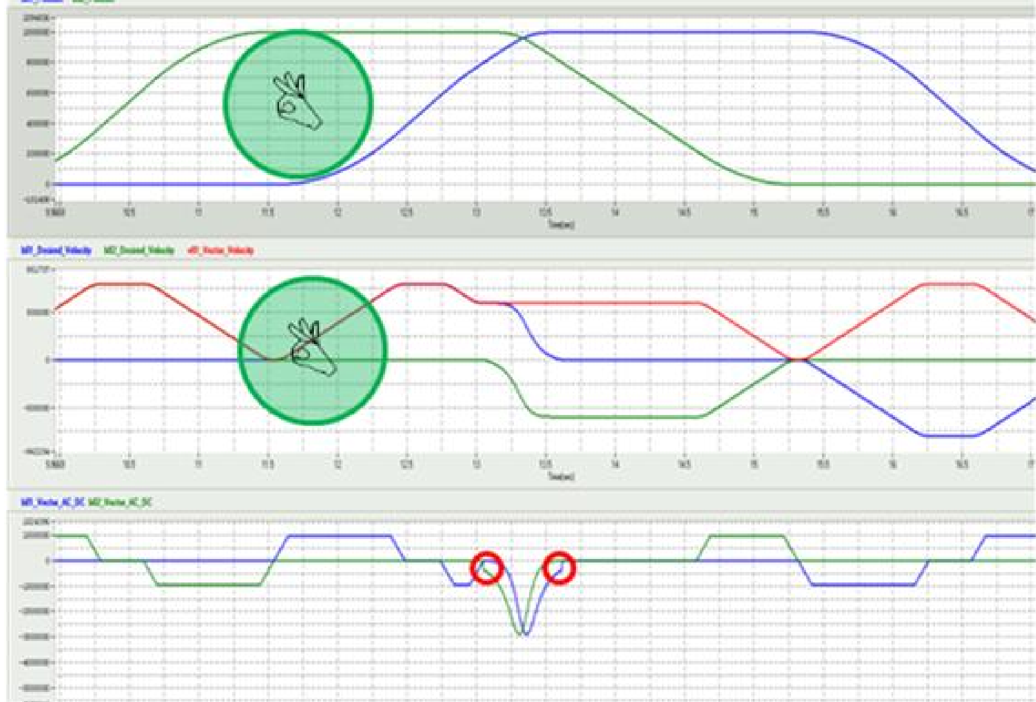


To view and understand the real difference between transition modes simple, Polynom 3, and 5, it is necessary to look at the Velocity and AC\DC graphs. However, it is necessary to define the differences between them in mathematical terms. The keyword of the differences is “continuity”, as the transition mode increases, the continuity is “better” (higher order).

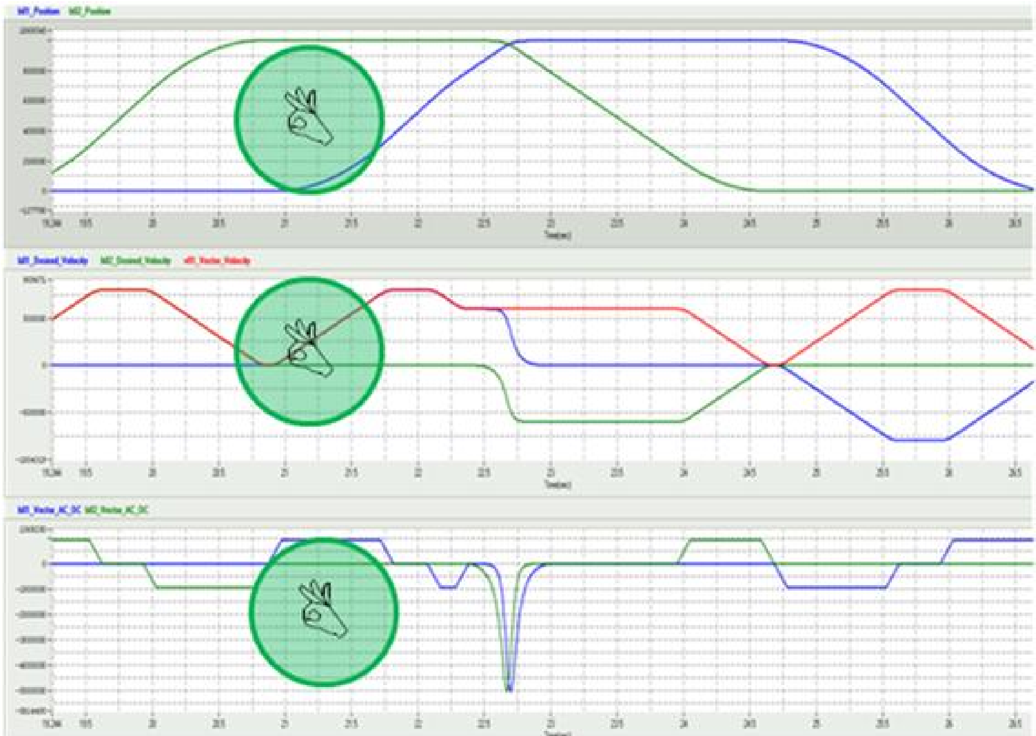
In Transition mode 3 (Simple), the continuity is only in position. In all cases where the continuity is in position, there are no unexpected jumps or sharp transitions.



In Transition mode 7(Polynom 3), the continuity is in position and velocity.



In Transition mode 8 (Polynom 5), the continuity is in position velocity and AC\DC.



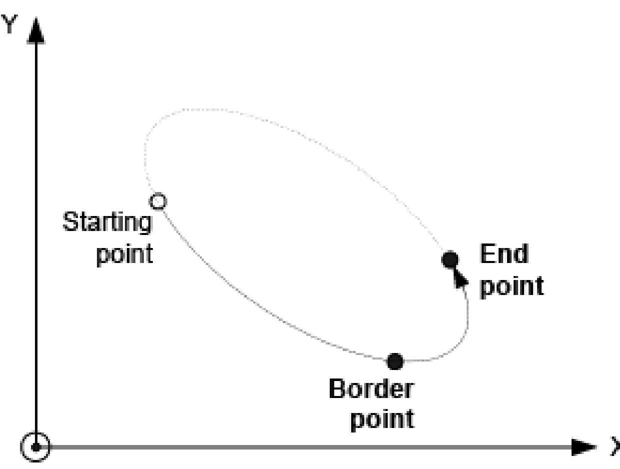
Looking at the graphs for each Transition mode (Position, Velocity and AC\DC), the red circles indicate the points of NON-continuity.

5.12 Multiple Axes Motion Control - Circular Modes

Consists of three modes:

- Border
- Center
- Radius

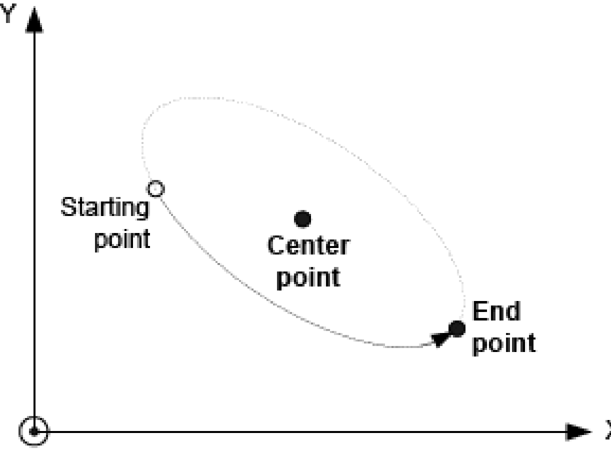
5.12.1 Border mode



CircMode = BORDER

Defines the end point (*dEndPoint*) and a border point (*dBorderPoint*) on the sector of the circle travelled by the machine (equivalent to the AuxPoint in the PLCopen standard). The border point usually can be reached by the machine, i.e. it can be taught. However, this parameter is restricted to angles < 2π in one single command.

5.12.2 Center mode

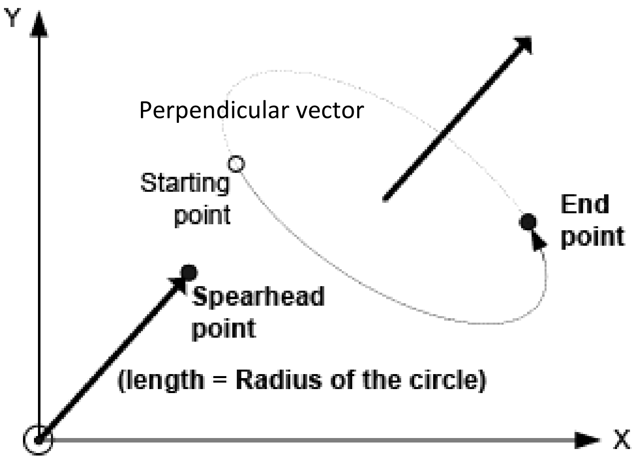


CircMode = CENTER

Defines the endpoint and the center point (*dCenterPoint*) of the circle (equivalent to the AuxPoint in the PLCopen standard). When using this mode, the input *PathChoice* defines whether the Short or the Long sector is travelled by the machine.

This parameter is restricted to angles < 2π and ≠ π in one single command.

5.12.3 Radius mode



CircMode = RADIUS

Defines the end point and the perpendicular vector of the circle plane. The length of the vector corresponds to the radius of the circle. The spearhead point of the vector is the input signal *dSpearHeadPoint* in absolute coordinates (equivalent to the AuxPoint in the PLCopen standard), i.e. referring to the origin of the coordinate system specified in CoordSystem.

If the diameter is larger than the distance between starting and end point, two different circles have to be considered. When using this mode, the user defines a choice of *Pathchoice*; clockwise or counterclockwise, and *ArcShortLong*; Short or Long segment. Clockwise or counterclockwise direction is defined as seen from the end of the perpendicular vector described by the user.

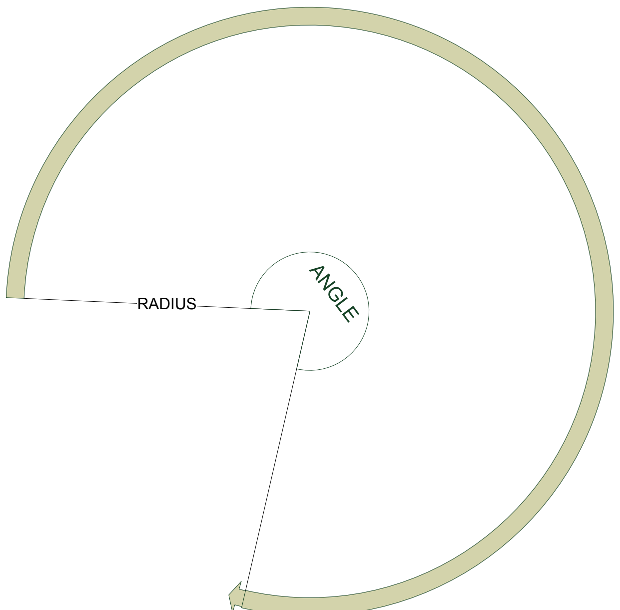
This parameter is restricted to angles < 2π in one single command and the perpendicular vector has to be computed.

Example:

dSpearHeadPoint = (50,0,0) → Circle in plane parallel to yz plane with radius 50 and rotation around axis parallel to x-axis (CoordSystem = MCS).

5.12.4 Angle Mode

The angle is the relative angular position for the coordinate system specified by the input signal CoordSystem, measured in degrees.



The angle is defined by:

$$|Angle| > 0$$

Its range may be any +ve or –ve number (e.g. 1012 is quite acceptable). In principle, an angle larger than 360°, means multiple rotations of an object in the direction defined.

5.12.5 PathChoice Data verification in MoveCircular functions

Path choice and arc short long parameters data verifications are now added in the MoveCircular functions. These verifications involve a check of the validity of the path choice and arc short long parameter according to the circle mode for the following parameters (in each mode):

MC_BORDER_CIRC_MODE	None
MC_CENTER_CIRC_MODE	eArcShortLong
MC_RADIUS_CIRC_MODE	eArcShortLong and ePathChoice
MC_ANGLE_CIRC_MODE	None

5.12.6 Move Polynomial Function Block

Polynomial Function Blocks were introduced to improve the ability to build motion sequences with a smooth velocity and AC\DC while performing transitions between motion blocks, for the Linear and Circular motions.

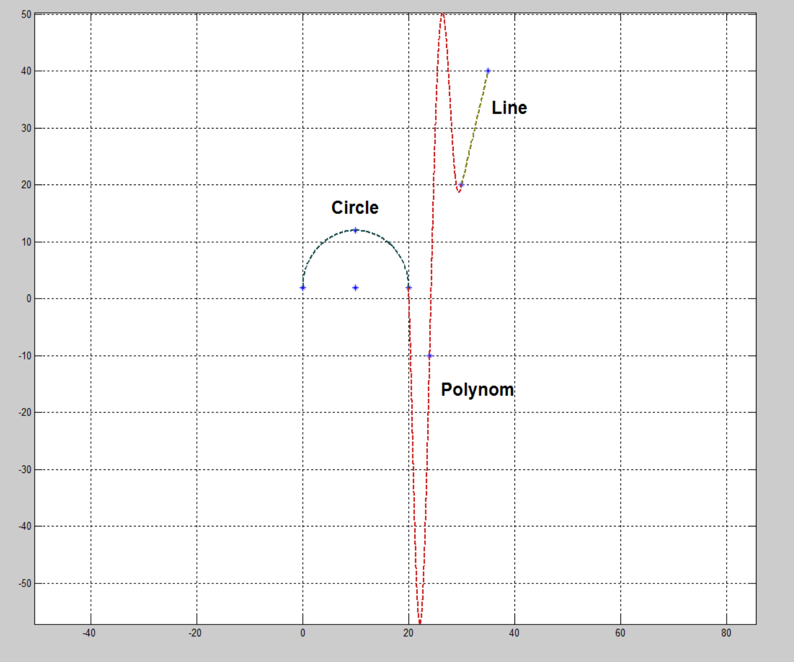


Figure 5-102 Polynomial Function Motion

The motion shape is a 6th order Polynom which crosses three given points:

- Start point
- Auxiliary point
- End point

The Position, Velocity, and AC/DC are controlled to provide the smoothness throughout the path as shown in the following examples.

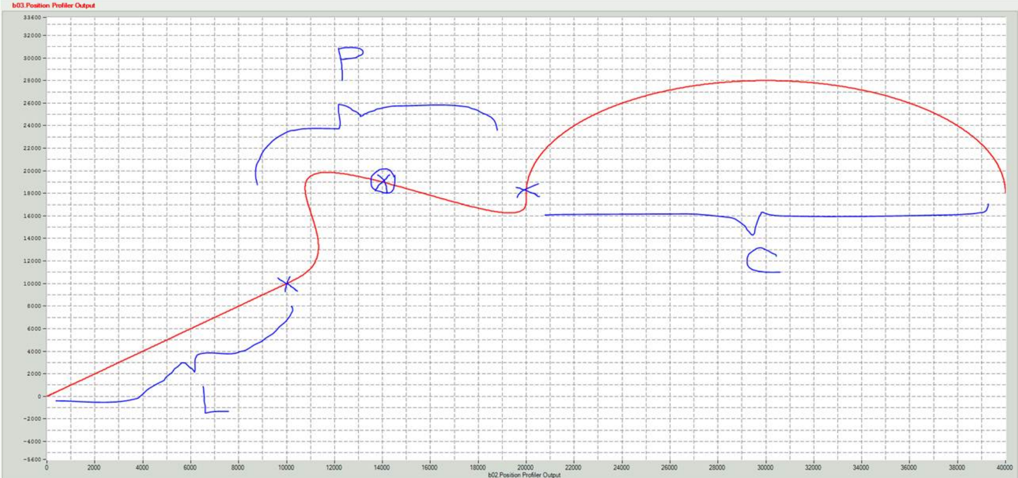


Figure 5-103 Example of Polynomial Function Motion

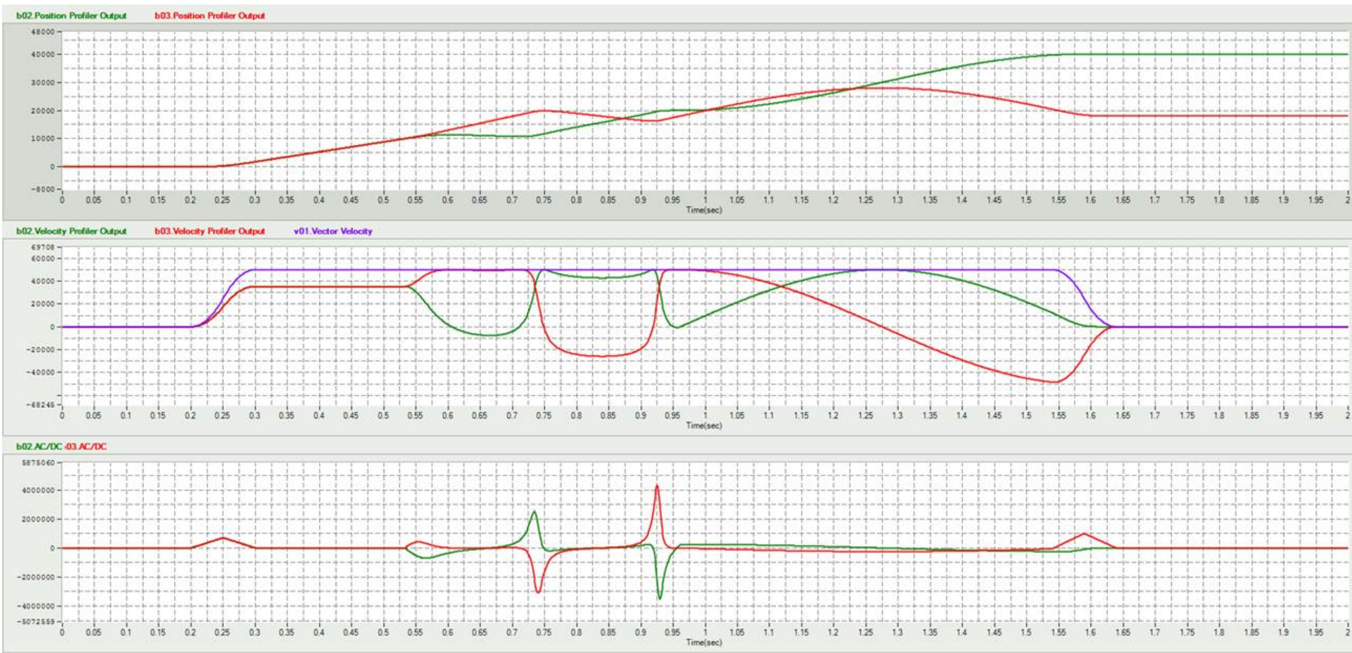


Figure 5-104 Example of Polynomial Function Motion within EAS Application

5.13 Multiple Axes Motion Control - Splines

Splines can be defined for the motion of a vector/system, designed where all the potential trajectory points are known in advance. The user is interested in the smoothest trajectory obtainable. Therefore using Splines may be the solution.

Splines are defined as n dimensional position points (1 < n<=16), which describe the geometry of the trajectory. The kinematics of the trajectory can be defined in two ways:

a) By the time interval between each point

b) By the kinematic parameters: Max Velocity, Max Acceleration, Max Deceleration, and Jerk.

Either way, the spline is designed as a sequence of cubic polynomials continuous by the first and second derivatives. The following Spline modes define kinematic options:

<table>
<tr><th>Spline mode</th><th>Explanation</th></tr>
<tr><td>MC_SPLINE_MODE_FT = 1</td><td rowspan="2">In modes MC_SPLINE_MODE_FT and MC_SPLINE_MODE_V_AC_DC the Spline Velocity is a parabolic function of time.</td></tr>
<tr><td>MC_SPLINE_MODE_V_AC_DC = 2</td></tr>
<tr><td>MC_SPLINE_MODE_CV = 7</td><td rowspan="2">In modes MC_SPLINE_MODE_CV and MC_SPLINE_MODE_NP the Spline is executed with Constant Velocity with an Acceleration segment at the start and Deceleration segment at the end.</td></tr>
<tr><td>MC_SPLINE_MODE_NP = 3</td></tr>
<tr><td>MC_ONLINE_SPLINE_MODE_CV = 9</td><td></td></tr>
</table>

When would a user prefer one mode to another?

If the user requires a minimal time for the whole trajectory but constant velocity is not required, the Spline mode V_AC_DC should be selected.

If on the other hand, the user knows the precise time intervals between each two points and a constant velocity is not required, the Spline mode FT should be selected.

If the trajectory must be executed with constant velocity, then the user should select the Spline modes CV or NP.

Having selected the appropriate Spline mode, a Spline data table (or series of Spline tables) is created (in Excel or otherwise) according to the definitions and explanations described in the next sub-sections. This Spline table is saved as a text file, and downloaded to the Maestro. Its possible file path may then be, for example: /mnt/jffs/usr/Spline_2D.txt

5.13.1 Spline Mode MC_SPLINE_MODE_FT = 1

In spline mode FT the user defines the spline mode, dimension (number of axes 1-16), number of points N and a table of points. Each row of the table has the same structure and contains information for the point i. The first column of the row i contains dT_i, the time interval (in seconds) required to arrive at point i from point i-1. The next row contains the position coordinates for the point i.

The example below is part of the spline table (first five points) for the butterfly spline.

Spline data			
Spline Mode:			1
Spline Dimension:			2
Spline number of points:			433
Spline data start			
	2.000000	35914.091423	0.000000
	2.000000	56390.497961	9943.166246
	2.000000	103925.224490	37825.688304
	2.000000	146247.550991	84436.062933
	2.000000	154381.879874	129541.778462

Spline in mode FT comprises of a function of time in continuous velocity and acceleration. Each segment i is a cubic polynomial of time with parameter deviation dT_i. This polynomial is built for each coordinate j. Velocities and accelerations are predefined by the parameters dT_i, i = 1,2,….N, but are limited by the parameters MMC_MAX_VELOCITY_PARAM and MMC_MAX_ACCELERATION_PARAM. Therefore, the initial values of dT_i can be increased to satisfy these constraints.

For the user to decrease the max. velocity or acceleration on some segment i, he can increase dT_i. To prevent undesirable deviations from the original polygon, some position control is implemented for the short splines (N < 50). This is performed by inserting additional internal points. If the user is dissatisfied with some interval on the spline curve, he can manually add a number of internal points.

Spline mode MC_SPLINE_MODE_FT should be used if the user can define precise time intervals for each segment and is not interested in constant velocity along the spline.

5.13.2 Spline mode MC_SPLINE_MODE_V_AC_DC = 2

In the spline mode MC_SPLINE_MODE_V_AC_DC the user defines the following:

- Spline mode
- Dimension(number of axes 1-16)
- Number of points N, max. velocity
- Max. acceleration (AC)
- Max. deceleration (DC)
- Max. jerk (or alternatively smooth factors for AC and DC)
- Table of points

Each table row has the same structure and contains the data for the point i. All columns contain position coordinates for the point i (1-16).

The example below is part of the spline table (first five points) for the spline.

Spline data	
Lissajous	
Spline mode:	2
Spline dimension:	2
Spline number of points:	145
Max Velocity:	32000.000000
Max AC:	1000000.000000
Max DC:	1000000.000000
Max Jerk:	20000000.000000
SF AC:	0.000000
SF DC:	0.000000
Splines data start	
50000.000000	0.000000
49572.243069	10821.980697
48296.291314	21130.913087
46193.976626	30438.071450
43301.270189	38302.222156

Spline in mode V/AC/DC comprises of a function of time in continuous velocity and acceleration. Each segment i is a cubic polynomial of time, specifically built for each coordinate j (1-16). Velocities and accelerations are limited by user defined parameters.

To prevent undesirable deviations from the original polygon, some position control is implemented for the short splines (N < 50). This is performed by inserting additional internal points. If the user is dissatisfied by the some interval of the spline curve, he can manually add a number of internal points.

Spline mode MC_SPLINE_MODE_V_AC_DC should be used if the user cannot define precise time intervals for each segment and is not interested in constant velocity along the spline curve but can define limitations on max velocity and AC/DC along the path.

5.13.3 Spline mode MC_SPLINE_MODE_NP = 3

In the spline mode MC_SPLINE_MODE_NP the user defines the following:

- Spline mode
- Dimension(number of axes 1-16),
- Number of points
- Max velocity
- Max AC
- Max DC
- Max jerk (or alternatively smooth factors for AC and DC)
- Table of points.

Each table row has the same structure and contains the data for point i. All columns contain position coordinates for the point i (1-16). The example below is part of the spline table (first five points) for the butterfly spline.

Spline data	
Spline mode:	8
Spline dimension:	2
Number of spline points:	433
Max Velocity:	100000.000000
Max AC:	10000000.000000
Max DC:	10000000.000000
Max Jerk:	20000000.000000
SF AC:	0.000000
SF DC:	0.000000
Splines data start	
35914.091423	0.000000
56390.497961	9943.166246
103925.224490	37825.688304
146247.550991	84436.062933
154381.879874	129541.778462

Splines in mode CV comprise of a function of some continuous parameter "**S**". Each segment i is a cubic polynomial of **s,** specifically built for each coordinate j (1-16), with velocities and accelerations limited by user defined parameters. The Spline is executed with a constant velocity that is less or equal to the max velocity defined by the user.

The difference with the mode CV is in definition of the parameter "S", whereas the NP modes is close to the natural parameter. In reality, for short splines (N < 50) and smoothed curves, the parameter is completely

natural. The advantage of the natural parameter is that the resultant spline does not require any additional calculations in real-time. On the other hand, to build such a spline without using the natural parameter can employ considerable off-line calculations (a couple of minutes for the long and unsmoothed spline). Smoothed splines like ellipse and helix do not need long iterative calculations.

The Spline is executed as a normal Maestro trajectory with the acceleration limited by jerk, constant velocity stage, and deceleration.

If the user is dissatisfied with some interval of the spline curve, he can manually add a number of internal points.

The Spline mode MC_SPLINE_MODE_NP should be used if the user is interested in constant velocity along the spline curve. This constant velocity is limited by the user defined V_{max} and
V_{ρ} = [MMC_MAX_ACCELERATION_PARAM *ρ_{min}]$^{1/2}$
where ρ_{min} is the minimal radius of curvature on the spline curve.

So $V_{const} = \min(V_{max}, V_{\rho})$.

This limitation usually does not influence smoothed trajectories like ellipse or helixes. This mode is recommended in situations where there are problems with the CV mode.

5.13.4 Spline mode MC_SPLINE_MODE_CV = 7

In the spline mode MC_SPLINE_MODE_CV the user defines the following:

- Spline mode
- Dimension(number of axes 1-16)
- Number of points
- Max velocity
- Max AC, max DC
- Max jerk (or alternatively smooth factors for AC and DC)
- Table of points

Each table row has the same structure and contains the data for the point i. All columns contain position coordinates for the point i (1-16).

The example below is part of the spline table (first five points) for the butterfly spline.

Spline data	
Spline mode:	7
Spline dimension:	2
Number of spline points:	433
Max Velocity:	100000.000000
Max AC:	10000000.000000
Max DC:	10000000.000000
Max Jerk:	20000000.000000
SF AC:	0.000000
SF DC:	0.000000
Splines data start	
35914.091423	0.000000
56390.497961	9943.166246
103925.224490	37825.688304
146247.550991	84436.062933
154381.879874	129541.778462

Splines in mode CV comprises of a function of some continuous velocity and acceleration dependent parameter “**S**”. Each segment i is a cubic polynomial of “**S**”, specifically built for each coordinate j (1-16), where the velocities and accelerations are limited by user defined parameters. The Spline is executed with constant velocity that is less or equal to the maximum velocity defined by the user, as a normal Maestro trajectory with the acceleration limited by the jerk, constant velocity stage and deceleration.

If the user is dissatisfied with the some interval of the spline curve, he can manually add a number of internal points.

The Spline mode MC_SPLINE_MODE_CV should be used where the user is interested in constant velocity along the spline curve. This constant velocity is limited by the user defined
V_{max} and V_{ρ} = [MMC_MAX_ACCELERATION_PARAM *ρ_{min}]$^{1/2}$
where ρ_{min} is the minimal radius of curvature on the spline curve.

So $V_{constant}$ = min(V_{max}, V_{ρ}).

This limitation usually does not influence smoothed trajectories like ellipse or helix.

5.13.5 Online Splines

5.13.5.1 Online Spline Of Type PT

This online spline is defined by the set of pairs (P_1T_1), (P_2T_2), (P_3T_3), where P_i is an n-dimensional position point and T_i is a time needed for the motion from P_{i-1} to P_i.

For the process to begin, the online spline profiler requires the three first pairs (P,T) and on completing execution of the current segment [i-1,i], requires one additional pair (P_{i+2}, T_{i+2}). The pairs (P,T) can be generated by the user program on the fly or taken from a file.

5.13.5.1.1 Polynomial Interpolation Functions

Quintic polynomial – Polynomial Order 5
(MC_QUINTIC_ON_PARAB_VT_DWELL and MC_QUINTIC_ON_PARAB_VT_DWELL)

Quintic polynomial guarantees continuity by position, velocity and acceleration. Its drawback is discontinuity by the jerk at every connection point.

In the plot below the position, velocity acceleration and jerk profiles for the online XY spline are shown with six points and non-constant time intervals (spline mode MC_QUINTIC_ON_PARAB_VT_DWELL).

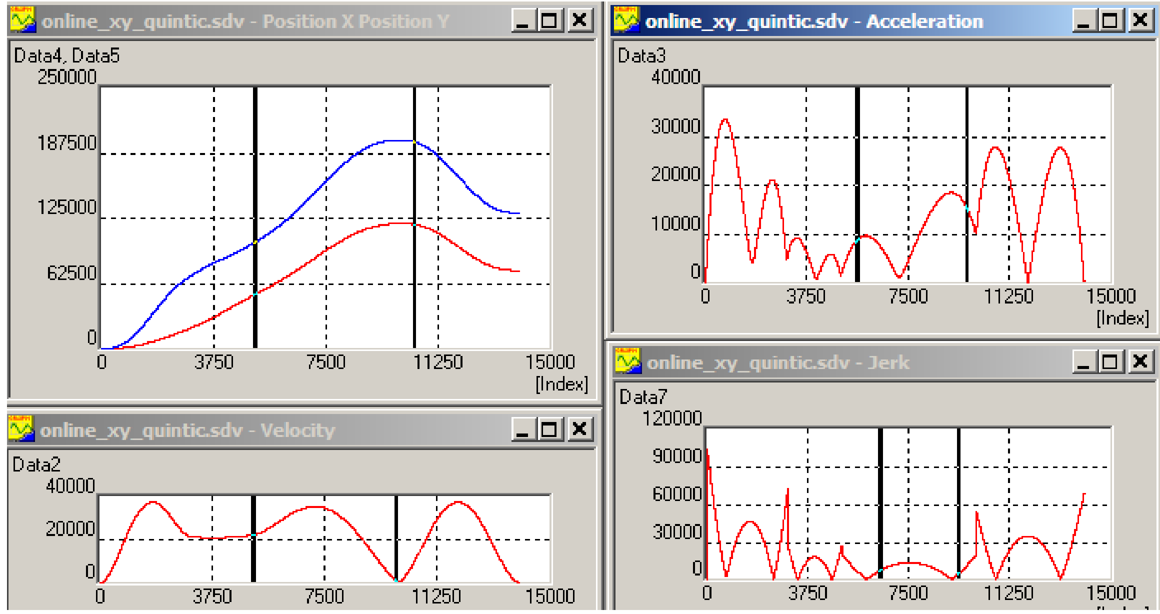


Figure 5-105 spline mode MC_QUINTIC_ON_PARAB_VT_DWELL

Septic polynomial - Polynomial Order 7
(MC_SEPTIC_ON_PARAB_VT_DWELL and MC_SEPTIC_ON_PARAB_FT_DWELL)

Septic polynomial guarantees continuity by position, velocity, acceleration and jerk. Its disadvantage is higher variability. It is the most universal interpolation mode that can be recommended in most cases. In the plot shown below, the same online 2D spline with six points.

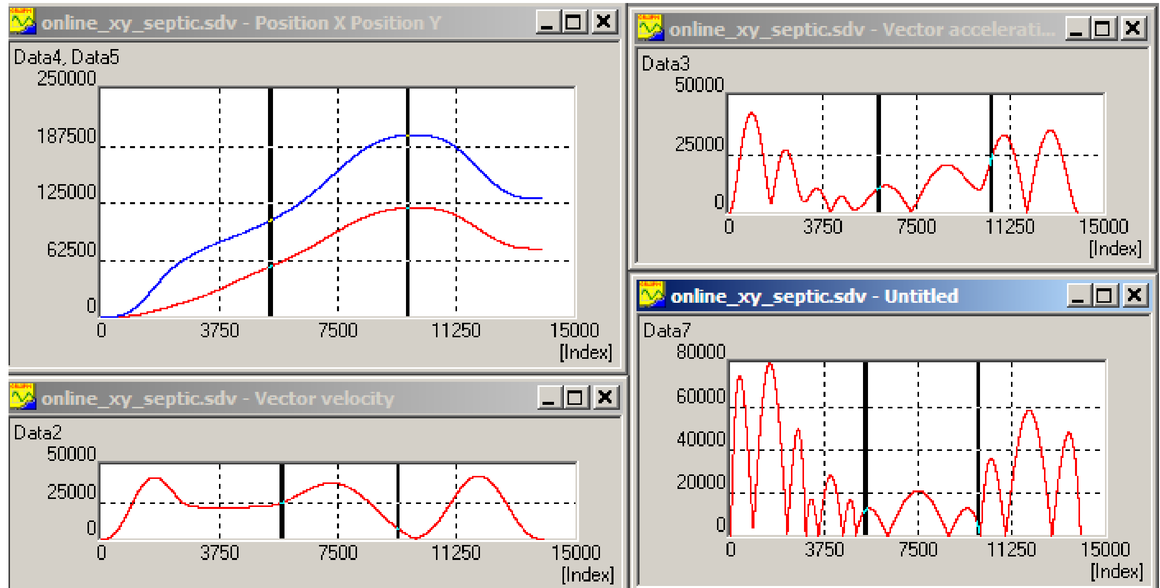


Figure 5-106 Septic polynomial - Polynomial Order 7

Segments with the septic interpolation can be smoothly connected to any other segments.

5.13.5.2 Sinusoidal Interpolation Functions

An ECAM profiler can also support a number of sinusoidal interpolation options.

Maestro supports two interpolation modes with the sinusoidal acceleration and cyclodal velocity profiles as well.

5.13.5.2.1 Triangle sinusoidal acceleration (MC_CYCLOID_VELOCITY_MODIFIED1)

Triangle sinusoidal acceleration with AC(t) and DC(t) increasing by the sinusoid to some maximum value and then decreasing to zero. It guarantees continuity by position, velocity, acceleration and jerk. Six points with the non-constant ΔT.

In the plots shown below the same online spline with six points.

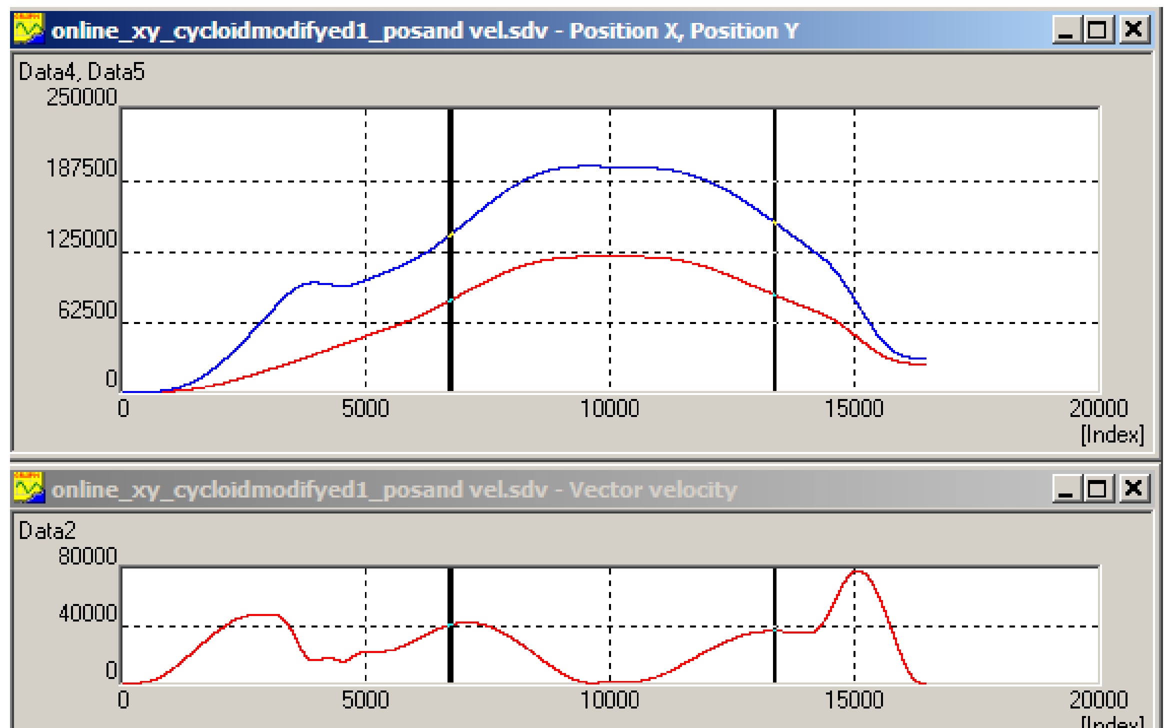


Figure 5-107 Position and velocity - Triangle sinusoidal acceleration

Figure 5-108 Acceleration and jerk plots - Triangle sinusoidal acceleration

5.13.5.2.2 Trapezoidal sinusoidal acceleration (MC_CYCLOID_VELOCITY_MODIFIED2)

Trapezoidal sinusoidal acceleration or modified trapezoidal acceleration usually consists of two parts:

- a movement with an acceleration increasing by the sinusoid from zero to AC_{max}, movement with AC_{max} (parabolic position, linear velocity profiles) and acceleration decreasing by the sinusoid to zero.
- a movement with a deceleration decreasing by the sinusoid from zero to -AC_{max}, movement with -AC_{max} (parabolic position profile, linear velocity profiles) and deceleration increasing by the sinusoid to zero.

It guarantees continuity by position, velocity, acceleration and jerk (if all the segments defined with this interpolation mode). An example can be seen in Figures 5-6.

In the plots shown below, the same online spline with six points.

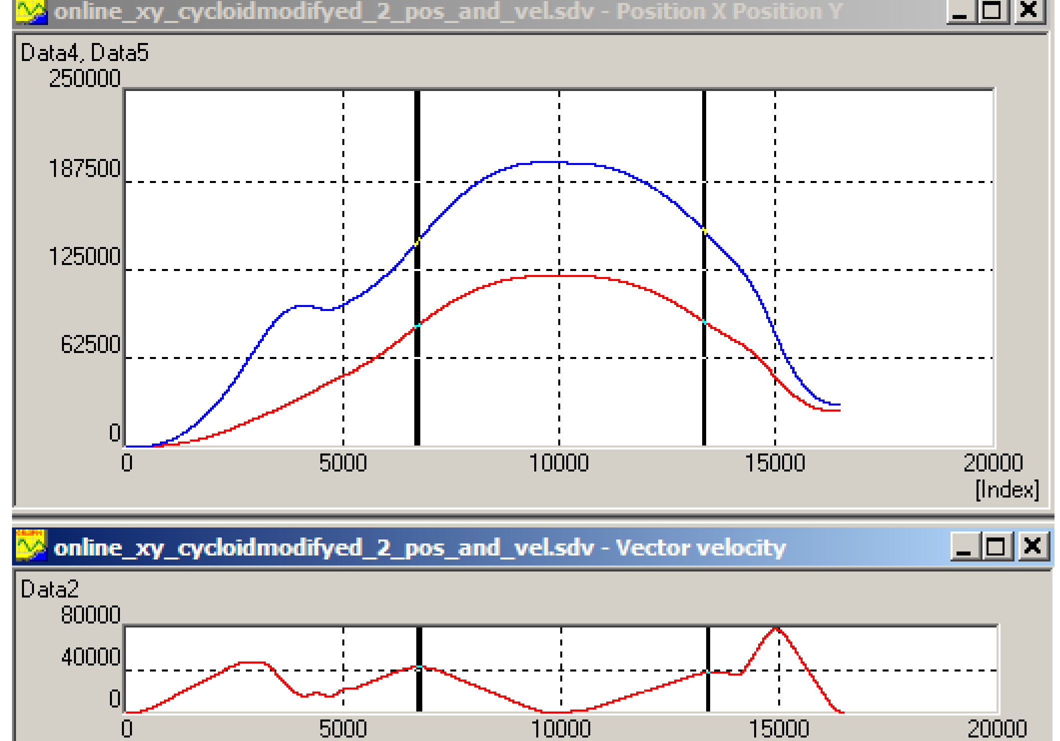


Figure 5-109 Position and velocity plots

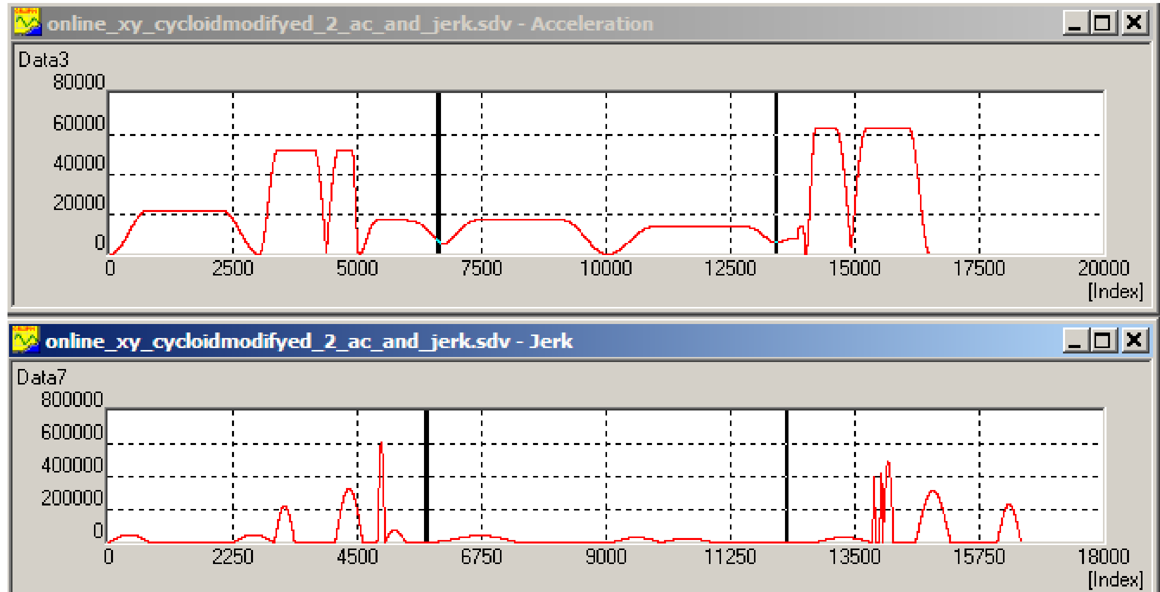


Figure 5-110 Acceleration and jerk

5.13.5.2.3 Online spline defined by a set of n-dimensional points P_1, P_2, and maximum vector velocity V_{max}

This online spline is defined by the set of n-dimensional points P_1, P_2, P_3, and maximum vector velocity V_{max}. For the process to begin, the online spline profiler requires 3 initial three points and on finishing execution of the current segment $[i-1,i]$, requires one additional point P_{i+2}. The points P_i can be generated by the user program on the fly or taken from a file. Time definition is not relevant with the exception of dwell segments – segments between two equal points

$P_i = P_{i+1}$. In this case time defined for the point i+1 defines the time for the system to be in stand still without motion. When the time T_{i+1} is elapsed a new point is requested. So in fact as we use pairs (P_i,T_i) but in most cases parameter T_i is not used.

The maximum vector velocity is achieved within one segment and motion is executed with the V_{max} to the last or dwell segment. While the last segment (or a segment that precedes dwell) the vector velocity decreases from V_{max} to zero.

Acceleration/deceleration segments are built by n-dimensional sinusoids and constant velocity segments as n-dimensional quintic polynomials (for the multi axis motion).

Two-dimensional online spline can be defined by ten points with a motion of XY axes and maximum vector velocity 50000 count/sec.

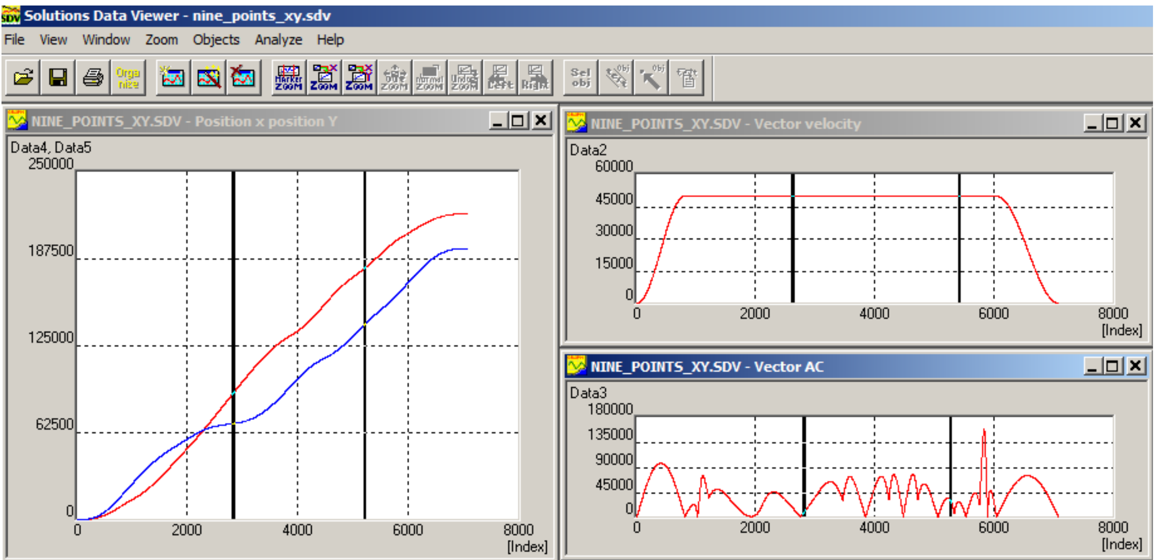


Figure 5-111 Two-dimensional online spline (XY axes + Max Vector Velocity)

Two-dimensional online spline can also be defined by ten points with a motion of XY axes and two dwell segments. The maximum vector velocity 50000 count/sec.

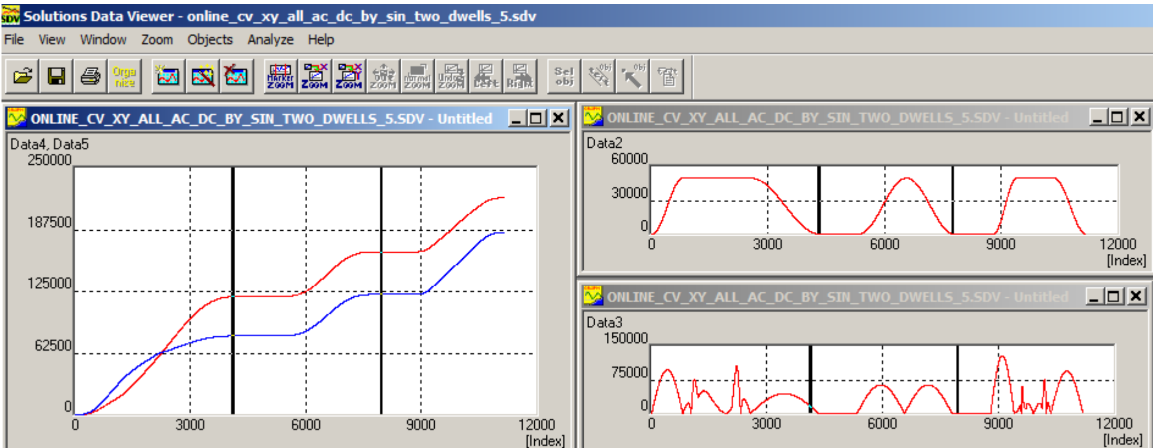


Figure 5-112 Two-dimensional online spline (XY axes + Two Dwell Segments)

In the plot position, velocity and acceleration profiles are shown below for the single axis online spline with the max velocity 50000 count/sec.

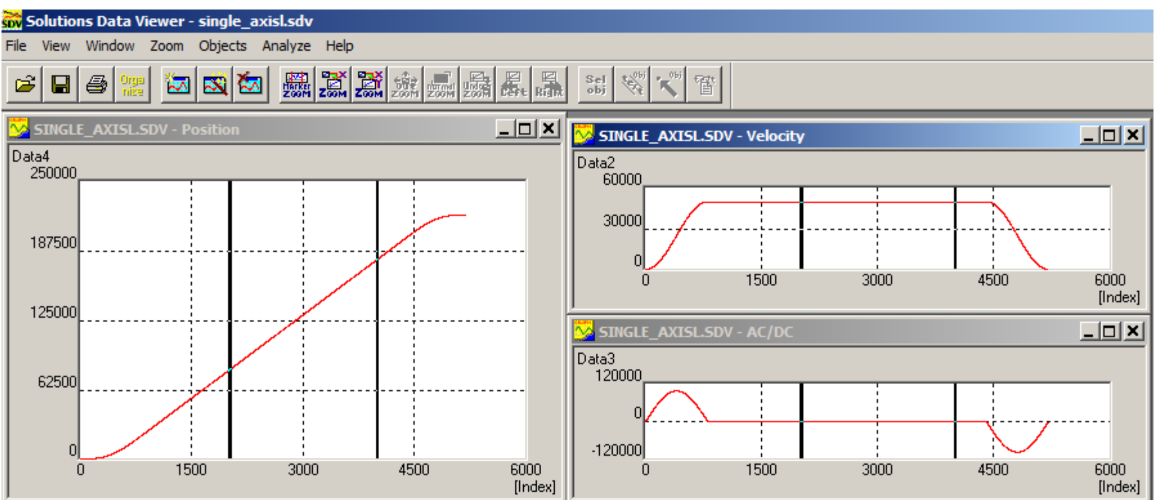


Figure 5-113 Single Axis Online Spline (Velocity and Acceleration Profiles)

Figure 10 shows a single axis online spline with the max velocity 50000 and a dwell segment.

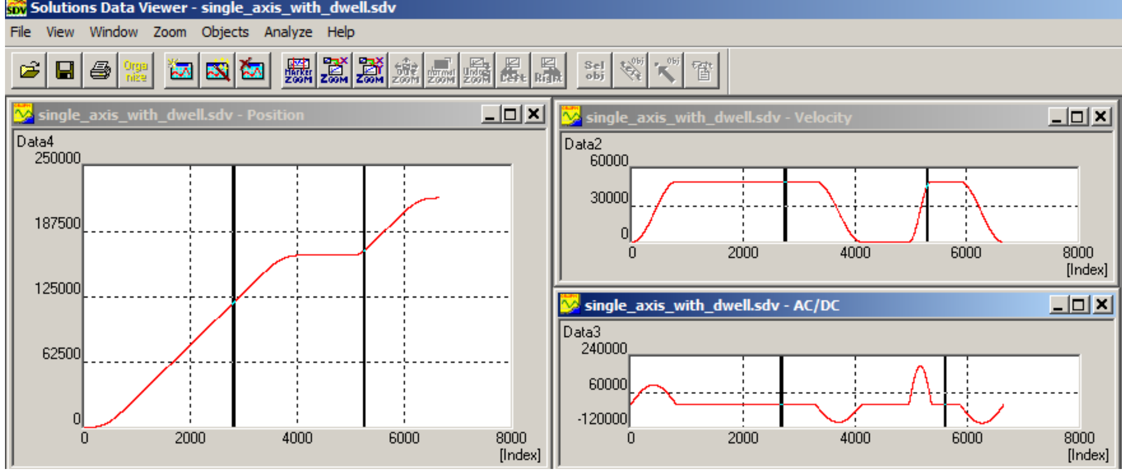


Figure 5-114 Single Axis Online Spline (Max Velocity + Dwell Segment)

5.13.6 Implementing the Spline Motion

The final Spline table(s) is now downloaded to the Maestro to reside in a known directory with a specific filepath. The functions associated with Spline are used to access the Spline table data. These are:

- MMC_PathSelect
- MMC_MovePath
- MMC_PathUnselect

The function MMC_PathSelect uses the parameter pPathToSplineFile to select and calculate the trajectory by defining the path to the Spline table. The function will then create an index for each Spline table, in the Maestro memory. The function MovePath then moves along the trajectory path using the hMemHandle index to indicate the path to move the vector/system.

5.14 Multiple Axes Motion Control – Functions

5.14.1 Function Block Status Bit Masks

The following table list the function block status bit masks for Single axis motion.

Bitwise value	Parameter	Explanation of parameter
0x00020000	NC_GROUP_STANDBY_MASK	Group Standby State Bit Mask
0x00010000	NC_GROUP_DISABLED_MASK	Group Disabled State Bit Mask
0x00008000	NC_GROUP_HOMING_MASK	Group Homing State Bit Mask
0x00004000	NC_GROUP_ERROR_STOP_MASK	Group Error State Bit Mask
0x00002000	NC_GROUP_MOVING_MASK	Group Moving State Bit Mask
0x00001000	NC_GROUP_STOPPING_MASK	Group Stopping State Bit Mask
	NC_MAX_SPLINES_FILE_PATH_LENGTH	100

The following multiple axes function blocks are described:

Multiple Axes
MMC_GroupStop
MMC_GroupHalt
MMC_MoveCircularAbsolute
MMC_MoveCircularAbsoluteCenter
MMC_MoveCircularAbsoluteBorder
MMC_MoveCircularAbsoluteRadius
MMC_MoveCircularAbsoluteAngle
MMC_MoveLinearAbsolute
MMC_MoveLinearRelative
MMC_MoveLinearAdditive
MMC_MoveLinearAdditiveEx
MMC_MoveLinearAbsoluteRepetitive
MMC_MoveLinearRelativeRepetitive
MMC_MovePath
MMC_PathSelect
MMC_PathUnselect

5.14.2 MMC_GroupStop

For multi-axis systems. Brings a group of axes to stop status.

```
MMC_LIB_API int MMC_GroupStopCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPSTOP_IN* pInParam,
OUT MMC_GROUPSTOP_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Not Supported
Source	GMAS\includes\MMC_PLCopen_group_API.h GMAS Programming(IEC 61331 Program)\ElmoGroupAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPSTOP_IN** group axes input structure that receives the MMC_GroupStop command.

pOutParam

Points to the **MMC_GROUPSTOP_OUT** group axes output structure receiving information, as a result of calling the MMC_GroupStop function.

Remarks

None

Scope

Only MCS coordinate system is supported.

MMC_GROUPSTOP_IN Structure

```
typedef struct MMC_Group_Stop_in{
float fDeceleration;
float fJerk;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
}MMC_GROUPSTOP_IN;
```

Parameters

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_GROUPSTOP_OUT Structure

```
typedef struct MMC_Group_Stop_out{
unsigned int uiHndl;
unsigned short usStatus;
unsigned short sErrorID;
}MMC_GROUPSTOP_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-98 describes the function block for MMC_GroupStop as applied within the IEC 61131 programming.

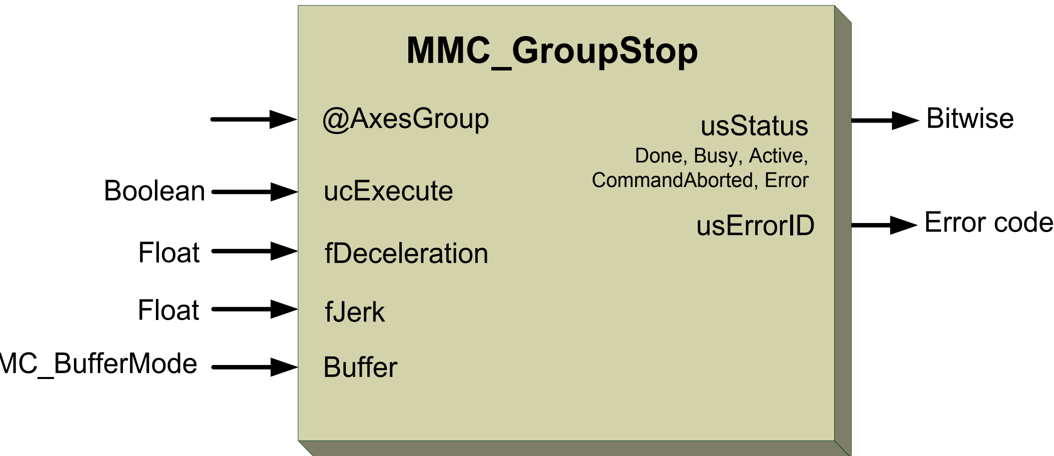


Figure 5-115: MMC_GroupStop function block

5.14.2.2 Function Block Code Example

```
int rc;
MMC_GROUPSTOP_IN        stGroupStop_in;
MMC_GROUPSTOP_OUT       stGroupStop_out;
//
// Inserting the structure parameters:
stGroupStop_in.fDeceleration     = 100000.0;             //Value of the deceleration
stGroupStop_in.fJerk             = 20000000.0;           //Maximum value of the Jerk
stGroupStop_in.eBufferMode       = MC_BUFFERED_MODE;     //Defines the behavior of the axis
stGroupStop_in.ucExecute         = 1;
//
rc = MMC_GroupStopCmd (hConn, iAxisRef, &stGroupStop_in, &stGroupStop_out);
if (rc != 0)
{
    HandleError();
}
```

5.14.3 MMC_GroupHalt

For multi-axis systems. Brings a group of axes to Halt status.

```
MMC_LIB_API int MMC_GroupHaltCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPHALT_IN* pInParam,
OUT MMC_GROUPHALT_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Not Supported
Source	GMAS\includes\MMC_PLCopen_group_API.h GMAS Programming(IEC 61331 Program)\ElmoGroupAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPHALT_IN** group axes input structure that receives the MMC_GroupHalt command.

pOutParam

Points to the **MMC_GROUPHALT_OUT** group axes output structure receiving information, as a result of calling the MMC_GroupHalt function.

Remarks

None

Scope

Only MCS coordinate system is supported.

MMC_GROUPHALT_IN Structure

```
typedef struct MMC_Group_Halt_in{
float fDeceleration;
float fJerk;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
}MMC_GROUPHALT_IN;
```

Parameters

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eBufferMode

MC_BUFFERED_MODE_ENUM defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_GROUPHALT_OUT Structure

```
typedef struct MMC_Group_Halt_out{
unsigned int uiHndl;
unsigned short usStatus;
unsigned short sErrorID;
}MMC_GROUPHALT_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-99 describes the function block for MMC_GroupHalt as applied within the IEC 61131 programming.

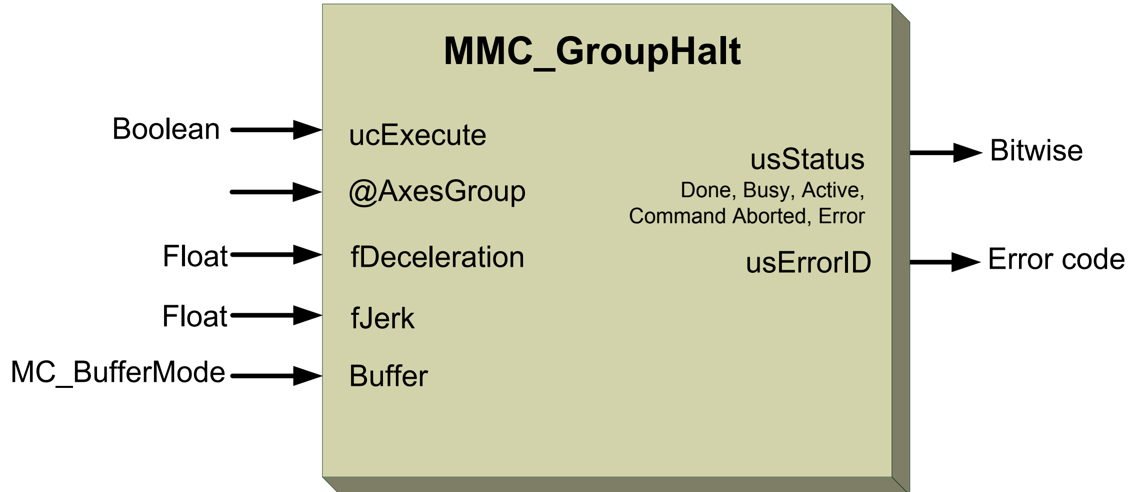


Figure 5-116: MMC_GroupHalt function block

5.14.3.2 Function Block Code Example

```
int rc;
MMC_GROUPHALT_IN        stGroupHalt_in;
MMC_GROUPHALT_OUT       stGroupHalt_out;
//
// Inserting the structure parameters:
stGroupHalt_in.fDeceleration    = 100000.0;            //Value of the deceleration
stGroupHalt_in.fJerk            = 20000000.0;          //Maximum value of the Jerk
stGroupHalt_in.eBufferMode      = MC_BUFFERED_MODE;    //Defines the behavior of the axis
stGroupHalt_in.ucExecute        = 1;
//
rc = MMC_GroupHaltCmd (hConn, iAxisRef, &stGroupHalt_in, &stGroupHalt_out);
if (rc != 0)
{
    HandleError();
}
```

5.14.4 MMC_MoveCircularAbsolute

For multi-axis systems. Commands an interpolated circular movement on an axes group from the actual position of the TCP.

```
MMC_LIB_API int MMC_MoveCircularAbsoluteCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVECIRCULARABSOLUTE_IN* pInParam,
OUT MMC_MOVECIRCULARABSOLUTE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Not Supported
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVECIRCULARABSOLUTE_IN** group axes input structure that receives the MMC_MoveCircularAbsolute command.

pOutParam

Points to the **MMC_MOVECIRCULARABSOLUTE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveCircularAbsolute function.

Remarks

The end point and auxiliary point, depending on the applied mode, are defined absolutely in the specified coordinate system.

Scope

Only MCS coordinate system is supported.

MMC_MOVECIRCULARABSOLUTE_IN Structure

```
typedef struct{
double dAuxPoint[NC_MAX_NUM_AXES_IN_NODE];
double dEndPoint[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
NC_PATH_CHOICE_ENUM ePathChoice;
NC_ARC_SHORT_LONG_ENUM eArcShortLong;
NC_CIRC_MODE_ENUM eCircleMode;
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucSuperimposed;
unsigned char ucExecute;
}MMC_MOVECIRCULARABSOLUTE_IN;
```

Parameters

dAuxPoint [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dAuxPoint can have vector array [1….3] double values in a technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dEndPoint [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute end point positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dEndPoint is a 2D or 3D double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

NC_PATH_CHOICE_ENUM ePathChoice

Defines the NC_PATH_CHOICE_ENUM enumerator types of supported path choice. The option are:

MC_NONE_PATH_CHOICE = 0

MC_CLOCKWISE = 1

MC_COUNTERCLOCKWISE = 2

eArcShortLong

Defines the types of supported arc length. The NC_ARC_SHORT_LONG_ENUM enumerator options are:

MC_NONE_ARC_CHOICE = 0

MC_SHORT = 1

MC_LONG = 2

eCircleMode

Defines the types of supported circular modes in 2D. Refer to the section **5.10.1 Coordinate System and kinematic transformation**. The NC_CIRC_MODE_ENUM enumerator options are:

MC_NONE_CIRC_MODE = 0

MC_BORDER_CIRC_MODE= 1

MC_CENTER_CIRC_MODE = 2

MC_RADIUS_CIRC_MODE = 3

MC_ANGLE_CIRC_MODE = 4

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control - Transition and Buffer Modes** and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

MC_BLENDING_HIGH_MODE = 6

Aborting Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered The next function block affects the axis as soon as the previous movement is completed.

BlendingLow The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).

BlendingPrevious Blending with the velocity of function block 1 at the end-position of this block

BlendingNext Blending with the velocity of function block 2 at end-position of function block1

BlendingHigh Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVECIRCULARABSOLUTE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVECIRCULARABSOLUTE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-100 describes the function block for MMC_MoveCircularAbsolute as applied within the IEC 61131 programming.

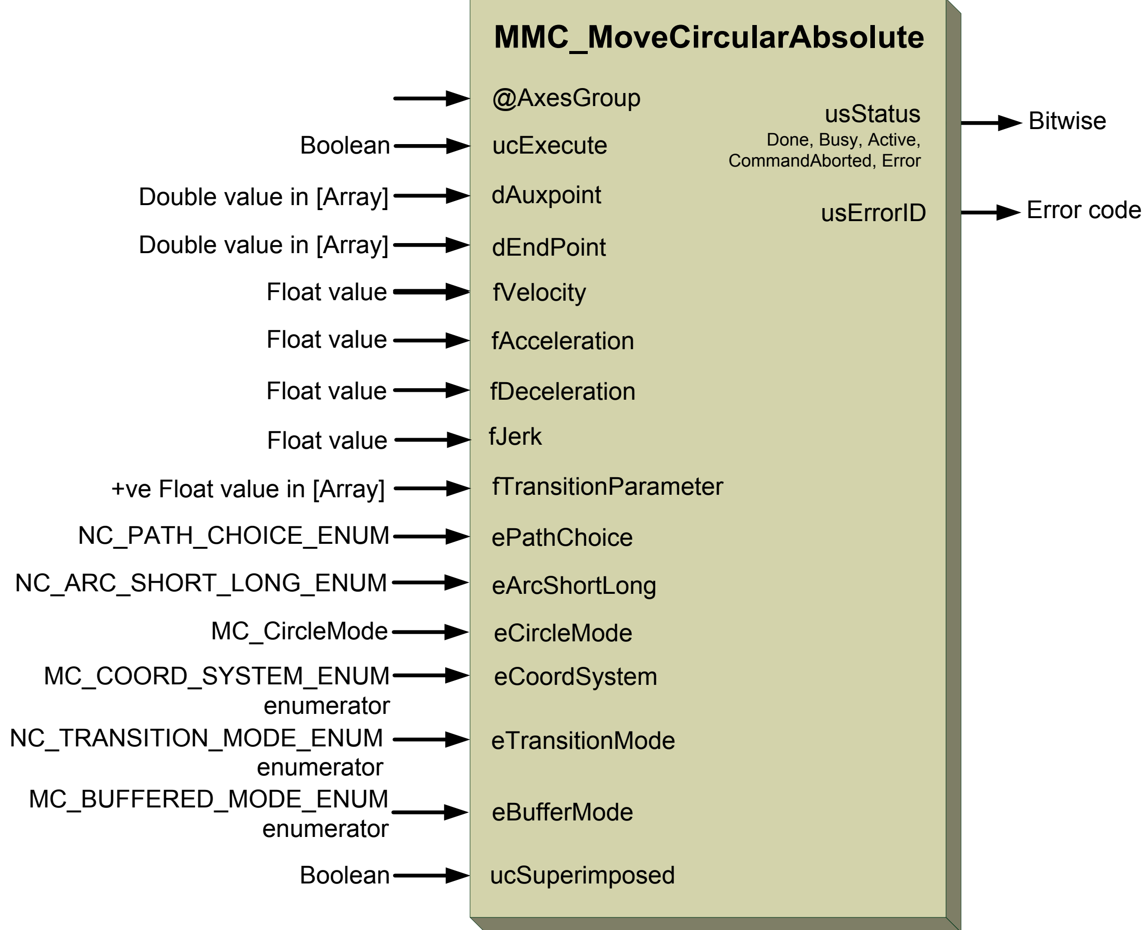


Figure 5-117: MMC_MoveCircularAbsolute function block

5.14.4.2 Function Block Code Example

```
int rc;
MMC_MOVECIRCULARABSOLUTE_IN         stMoveCircularAbs_in;
MMC_MOVECIRCULARABSOLUTE_OUT        stMoveCircularAbs_out;
//
// Inserting the structure parameters:
stMoveCircularAbs_in.fAcceleration          = 100000.0;           //Value of the acceleration
stMoveCircularAbs_in.fDeceleration          = 100000.0;           //Value of the deceleration
stMoveCircularAbs_in.fJerk                  = 20000000.0;         //Maximum value of the Jerk
stMoveCircularAbs_in.dAuxPoint[0]           = 5000;  //Absolute center positions for each
dimension
stMoveCircularAbs_in.dAuxPoint[1]           = 5000;  //Absolute center positions for each
dimension
stMoveCircularAbs_in.fTransitionParameter[0] = 10000; //Transition parameters which
characterize the contour curve
stMoveCircularAbs_in.fTransitionParameter[1] = 10000;             //Transition parameters
which characterize the contour curve
stMoveCircularAbs_in.eArcShortLong          = MC_SHORT;   //Defines the types of supported arc
length
stMoveCircularAbs_in.eCoordSystem           = MC_MCS_COORD;   //Supported coordinate system
stMoveCircularAbs_in.eTransitionMode        = MC_TM_DEFINED_VELOCITY_MODE;//Supported
transition modes
stMoveCircularAbs_in.eBufferMode            = MC_BUFFERED_MODE; //Defines the behavior of the
axis
stMoveCircularAbs_in.dEndPoint[0]           = 5000.0; //Absolute end point positions for each
dimension
stMoveCircularAbs_in.dEndPoint[1]           = 5000.0;//Absolute end point positions for each
dimension
```

```
stMoveCircularAbs_in.fVelocity              = 5000.0;           //Maximum Velocity of the
path
stMoveCircularAbs_in.ePathChoice            = MC_CLOCKWISE;     //Supported path choice
stMoveCircularAbs_in.eCircleMode            = MC_BORDER_CIRC_MODE; //Supported circular modes
in 2D
stMoveCircularAbs_in.ucSuperimposed         = 1;        //Option to superimpose is operated
stMoveCircularAbs_in.ucExecute              = 1;
//
rc = MMC_MoveCircularAbsoluteCmd (hConn, iAxisRef, &stMoveCircularAbs_in,
&stMoveCircularAbs_out);
if (rc != 0)
{
    HandleError();
}
```

5.14.4.3 Implementation Example

The example below shows two MMC_MoveCircular Absolute function blocks operation in conjunction. Their timing diagram displays dots on the red line, based on the same timing differences and representing the velocity.

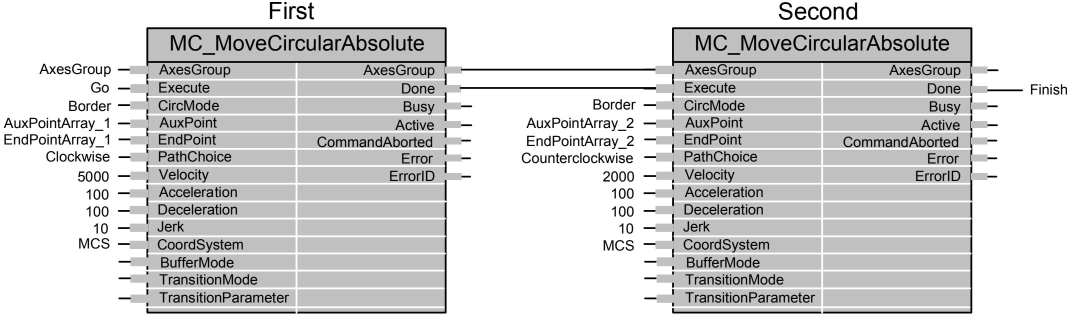


Figure 5-118: MMC_MoveCircularAbsolute - Example

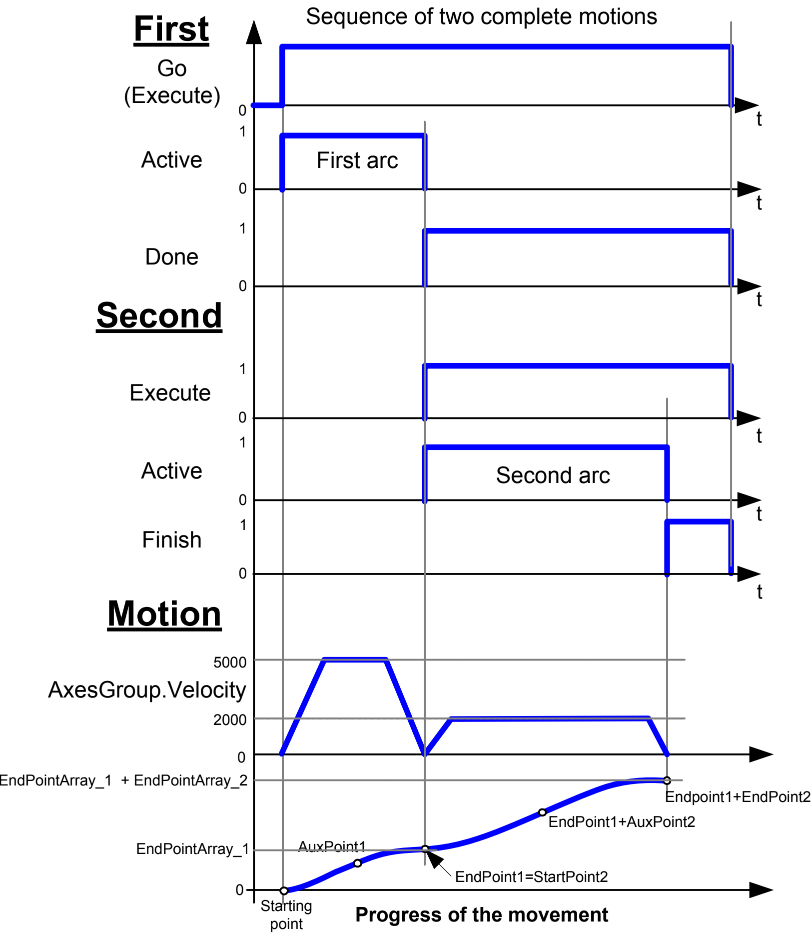


Figure 5-119: MMC_MoveCircularAbsolute timing diagram - Example

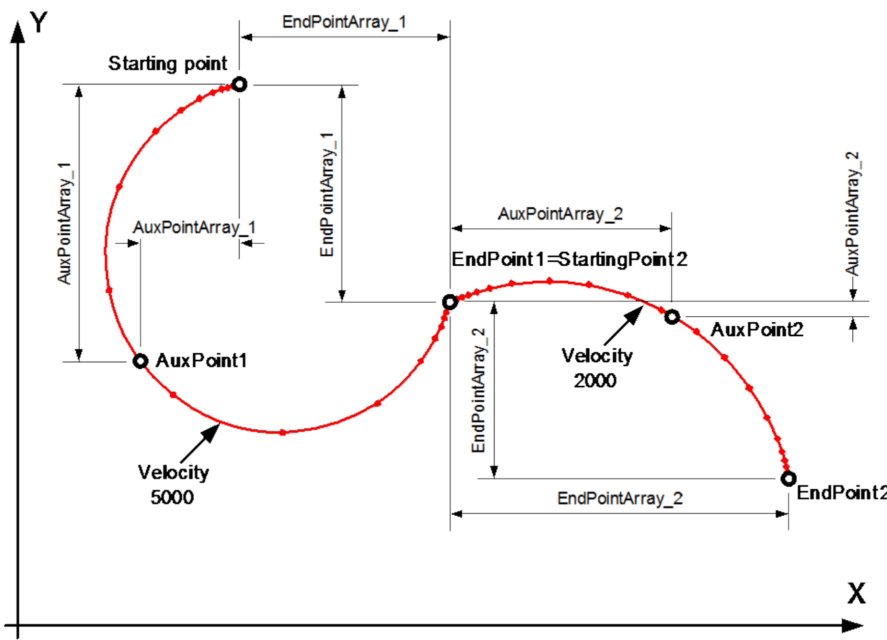


Figure 5-120: Timing diagram for MMC_MoveCircularAbsolute – Example

5.14.5 MMC_MoveCircularAbsoluteCenter

For multi-axis systems. Commands an interpolated circular center movement on an axes group from the actual position of the TCP.

```
MMC_LIB_API int MMC_MoveCircularAbsoluteCenterCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVECIRCULARABSOLUTECENTER_IN* pInParam,
OUT MMC_MOVECIRCULARABSOLUTE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVECIRCULARABSOLUTECENTER_IN** group axes input structure that receives the MMC_MoveCircularAbsoluteCenter command.

pOutParam

Points to the **MMC_MOVECIRCULARABSOLUTE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveCircularAbsoluteCenter function.

Remarks

The end point and auxiliary point, depending on the applied mode, are defined absolutely in the specified coordinate system.

Scope

Only MCS coordinate system is supported.

MMC_MOVECIRCULARABSOLUTECENTER_IN Structure

```
typedef struct{
double dCenterPoint[NC_MAX_NUM_AXES_IN_NODE];
double dEndPoint[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
NC_ARC_SHORT_LONG_ENUM eArcShortLong;
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucSuperimposed;
unsigned char ucExecute;
}MMC_MOVECIRCULARABSOLUTECENTER_IN;
```

Parameters

dCenterPoint [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute center positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dCenterPoint can have vector array [1….3] double values in a technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dEndPoint [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute end point positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dEndPoint is a 2D or 3D double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eArcShortLong

Defines the types of supported arc length. The NC_ARC_SHORT_LONG_ENUM enumerator options are:

MC_NONE_ARC_CHOICE = 0

MC_SHORT = 1

MC_LONG = 2

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

MC_BLENDING_HIGH_MODE = 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVECIRCULARABSOLUTE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVECIRCULARABSOLUTE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-104 describes the function block for MMC_MoveCircularAbsoluteCenter as applied within the IEC 61131 programming.

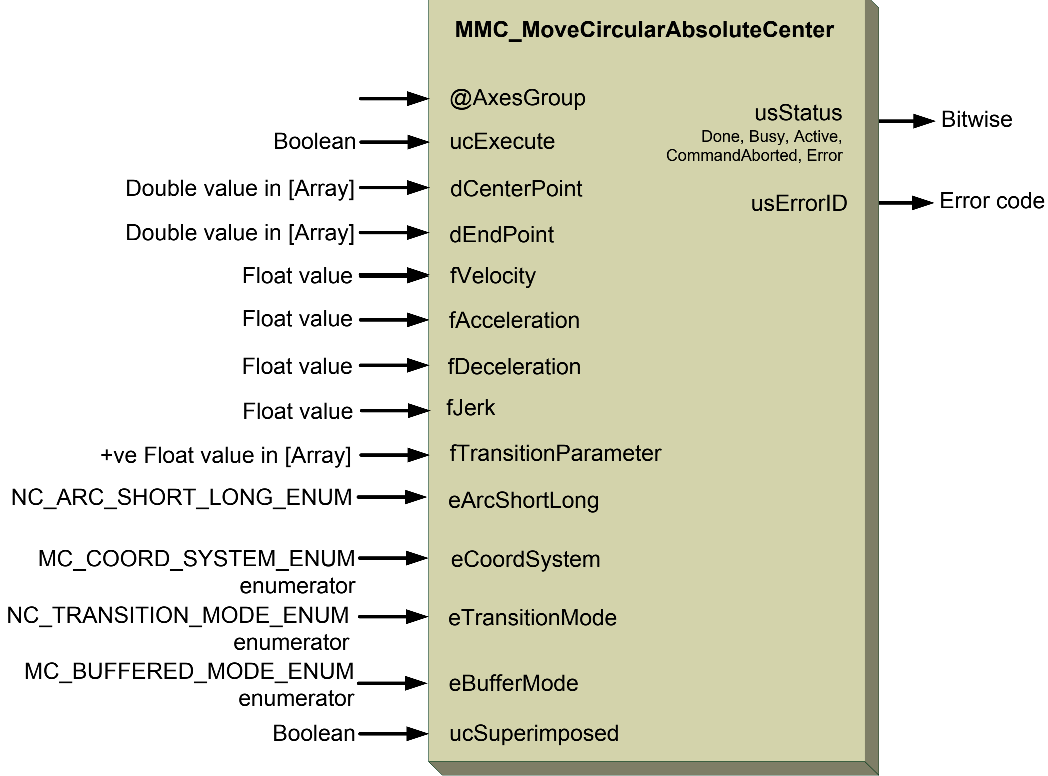


Figure 5-121: MMC_MoveCircularAbsoluteCenter function block

5.14.5.2 Function Block Code Example

```
int rc;
MMC_MOVECIRCULARABSOLUTECENTER_IN   stMoveCircularAbsCenter_in;
MMC_MOVECIRCULARABSOLUTE_OUT        stMoveCircularAbs_out;
//
// Inserting the structure parameters:
stMoveCircularAbsCenter_in.fAcceleration              = 100000.0;       //Value of the
acceleration
stMoveCircularAbsCenter_in.fDeceleration              = 100000.0;       //Value of the
deceleration
stMoveCircularAbsCenter_in.fJerk                      = 20000000.0;     //Maximum value of the
Jerk
stMoveCircularAbsCenter_in.dCenterPoint[0]            = 5000;//Absolute center positions for
each dimension
stMoveCircularAbsCenter_in.dCenterPoint[1]            = 5000;//Absolute center positions for
each dimension
stMoveCircularAbsCenter_in.fTransitionParameter[0]    = 10000;          //Transition
parameters which characterize the contour curve
stMoveCircularAbsCenter_in.fTransitionParameter[1]    = 10000;          //Transition
parameters which characterize the contour curve
stMoveCircularAbsCenter_in.eArcShortLong              = MC_SHORT;//Defines the types of
supported arc length
stMoveCircularAbsCenter_in.eCoordSystem               = MC_MCS_COORD; //Supported coordinate
system
stMoveCircularAbsCenter_in.eTransitionMode            = MC_TM_DEFINED_VELOCITY_MODE;
     //Supported transition modes
stMoveCircularAbsCenter_in.eBufferMode                = MC_BUFFERED_MODE;//Defines the
behavior of the axis
stMoveCircularAbsCenter_in.dEndPoint[0]               = 5000.0;     //Absolute end point
positions for each dimension
stMoveCircularAbsCenter_in.dEndPoint[1]               = 5000.0;     //Absolute end point
positions for each dimension
stMoveCircularAbsCenter_in.fVelocity                  = 5000.0;     //Maximum Velocity of the
path
stMoveCircularAbsCenter_in.ucSuperimposed             = 1;          //Option to superimpose is
operated
stMoveCircularAbsCenter_in.ucExecute                  = 1;
//
rc = MMC_MoveCircularAbsoluteCenterCmd (hConn, iAxisRef, &stMoveCircularAbsCenter_in,
&stMoveCircularAbs_out);
if (rc != 0)
{
    HandleError();
}
```

5.14.6 MMC_MoveCircularAbsoluteBorder

For multi-axis systems. Commands an interpolated circular border movement on an axes group from the actual position of the TCP.

```
MMC_LIB_API int MMC_MoveCircularAbsoluteBorderCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVECIRCULARABSOLUTEBORDER_IN* pInParam,
OUT MMC_MOVECIRCULARABSOLUTE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Not Supported
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVECIRCULARABSOLUTEBORDER_IN** group axes input structure that receives the MMC_MoveCircularAbsoluteBorder command.

pOutParam

Points to the **MMC_MOVECIRCULARABSOLUTE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveCircularAbsoluteBorder function.

Remarks

The end-point and auxiliary point, depending on the applied mode, are defined absolutely in the specified coordinate system.

Scope

Only MCS coordinate system is supported.

MMC_MOVECIRCULARABSOLUTEBORDER_IN Structure

```
typedef struct{
double dBorderPoint[NC_MAX_NUM_AXES_IN_NODE];
double dEndPoint[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucSuperimposed;
unsigned char ucExecute;
}MMC_MOVECIRCULARABSOLUTEBORDER_IN;
```

Parameters

dBorderPoint [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute border positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dBorderPoint can have vector array [1….3] double values in a technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dEndPoint [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute end point positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dEndPoint is a 2D or 3D double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVECIRCULARABSOLUTE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVECIRCULARABSOLUTE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-105 describes the function block for MMC_MoveCircularAbsoluteBorder as applied within the IEC 61131 programming.

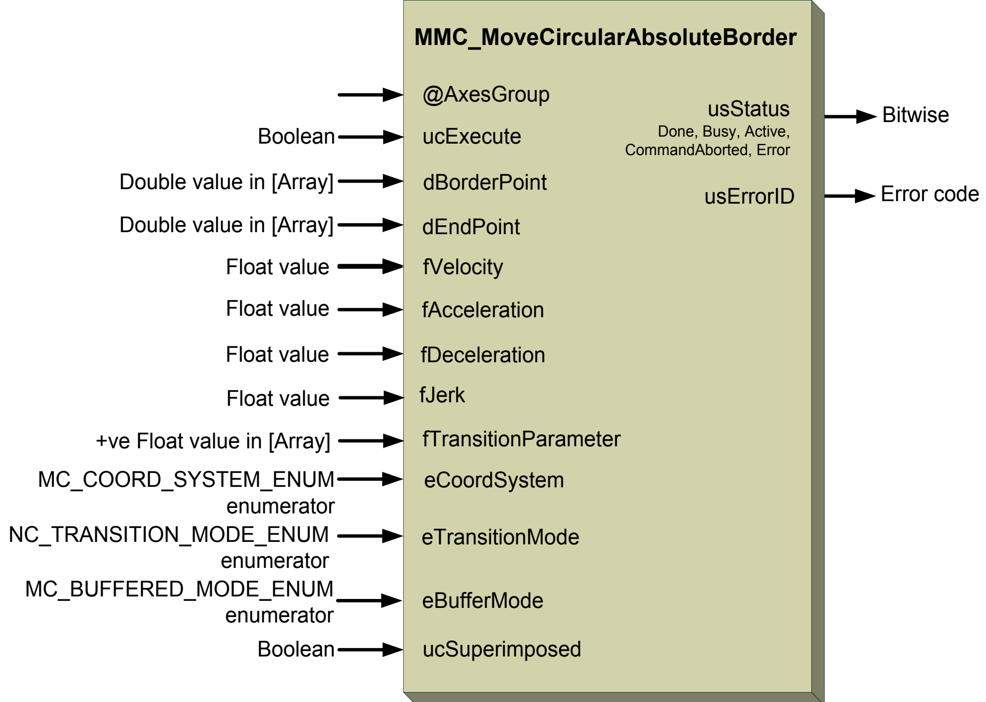


Figure 5-122: MMC_MoveCircularAbsoluteBorder function block

5.14.6.2 Function Block Code Example

```
int rc;
MMC_MOVECIRCULARABSOLUTEBORDER_IN   stMoveCircularAbsBorder_in;
MMC_MOVECIRCULARABSOLUTE_OUT        stMoveCircularAbs_out;
//
// Inserting the structure parameters:
stMoveCircularAbsBorder_in.fAcceleration            = 100000.0;       //Value of the 
acceleration
stMoveCircularAbsBorder_in.fDeceleration            = 100000.0;       //Value of the 
deceleration
stMoveCircularAbsBorder_in.fJerk                    = 20000000.0;     //Maximum value of the 
Jerk
stMoveCircularAbsBorder_in.dBorderPoint[0]          = 10000;          //Absolute border 
positions for each dimension
stMoveCircularAbsBorder_in.dBorderPoint[1]          = 10000; //Absolute border positions for 
each dimension
stMoveCircularAbsBorder_in.fTransitionParameter[0]  = 10000;          //Transition 
parameters which characterize the contour curve
stMoveCircularAbsBorder_in.fTransitionParameter[1]  = 10000;          //Transition 
parameters which characterize the contour curve
stMoveCircularAbsBorder_in.eCoordSystem             = MC_MCS_COORD;   //Supported coordinate 
system
stMoveCircularAbsBorder_in.eTransitionMode          = MC_TM_DEFINED_VELOCITY_MODE;
     //Supported transition modes
stMoveCircularAbsBorder_in.eBufferMode              = MC_BUFFERED_MODE;//Defines the 
behavior of the axis
stMoveCircularAbsBorder_in.dEndPoint[0]             = 10000; //Absolute end point positions 
for each dimension
stMoveCircularAbsBorder_in.dEndPoint[1]             = 10000;          //Absolute end point 
positions for each dimension
stMoveCircularAbsBorder_in.fVelocity                = 5000.0;         //Maximum Velocity of 
the path
stMoveCircularAbsBorder_in.ucSuperimposed           = 1;              //Option to 
superimpose is operated
stMoveCircularAbsBorder_in.ucExecute                = 1;
//
rc = MMC_MoveCircularAbsoluteBorderCmd (hConn, iAxisRef, &stMoveCircularAbsBorder_in, 
&stMoveCircularAbs_out);
if (rc != 0)
{
    HandleError();
}
```

5.14.7 MMC_MoveCircularAbsoluteRadius

For multi-axis systems. Commands an interpolated circular movement radius on an axes group from the actual position of the TCP.

```
MMC_LIB_API int MMC_MoveCircularAbsoluteRadiusCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVECIRCULARABSOLUTERADIUS_IN* pInParam,
OUT MMC_MOVECIRCULARABSOLUTE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MOVECIRCULARABSOLUTERADIUS_IN** group axes input structure that receives the MMC_MoveCircularAbsoluteRadius command.

pOutParam

Points to the **MMC_MOVECIRCULARABSOLUTE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveCircularAbsoluteRadius function.

Remarks

The end-point and auxiliary point, depending on the applied mode, are defined absolutely in the specified coordinate system.

Scope

Only MCS coordinate system is supported.

MOVECIRCULARABSOLUTERADIUS_IN Structure

```
typedef struct{
double dSpearHeadPoint[NC_MAX_NUM_AXES_IN_NODE];
double dEndPoint[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
NC_PATH_CHOICE_ENUM ePathChoice;
NC_ARC_SHORT_LONG_ENUM eArcShortLong;
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucSuperimposed;
unsigned char ucExecute;
}MMC_MOVECIRCULARABSOLUTERADIUS_IN;
```

Parameters

dSpearHeadPoint [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute radius positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dSpearHeadPoint can have vector array [1….3] double values in a technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dEndPoint [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute end point positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dEndPoint is a 2D or 3D double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

ePathChoice

Defines the NC_PATH_CHOICE_ENUM enumerator types of supported path choice. The option are:

MC_NONE_PATH_CHOICE = 0

MC_CLOCKWISE = 1

MC_COUNTERCLOCKWISE = 2

eArcShortLong

Defines the types of supported arc length. The NC_ARC_SHORT_LONG_ENUM enumerator options are:

MC_NONE_ARC_CHOICE = 0

MC_SHORT = 1

MC_LONG = 2

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

MC_BLENDING_HIGH_MODE = 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVECIRCULARABSOLUTE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVECIRCULARABSOLUTE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-106 describes the function block for MMC_MoveCircularAbsoluteRadius as applied within the IEC 61131 programming.

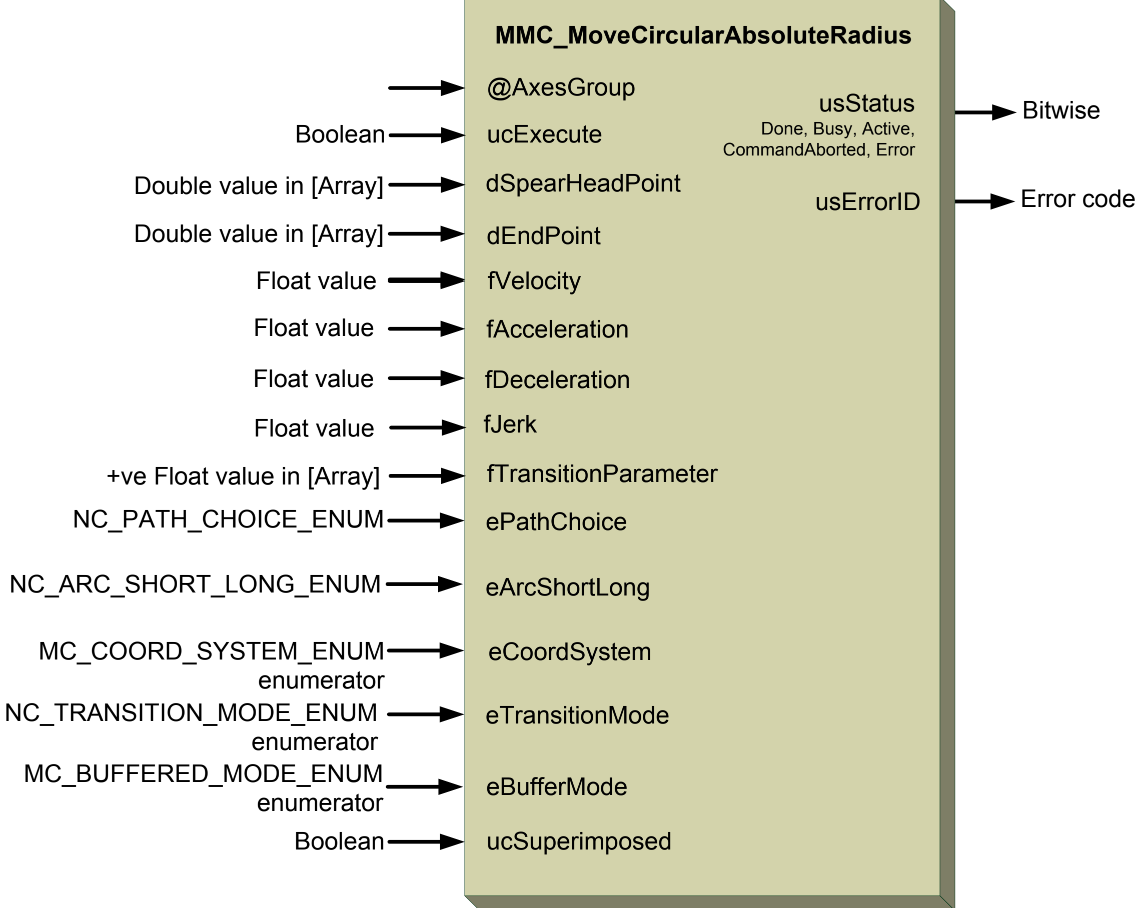


Figure 5-123: MMC_MoveCircularAbsoluteRadius function block

5.14.7.2 Function Block Code Example

```
int rc;
MMC_MOVECIRCULARABSOLUTERADIUS_IN   stMoveCircularAbsRadius_in;
MMC_MOVECIRCULARABSOLUTE_OUT        stMoveCircularAbs_out;
//
// Inserting the structure parameters:
stMoveCircularAbsRadius_in.fAcceleration            = 100000.0;        //Value of the 
acceleration
stMoveCircularAbsRadius_in.fDeceleration            = 100000.0;        //Value of the 
deceleration
stMoveCircularAbsRadius_in.fJerk                    = 20000000.0;      //Maximum value of the 
Jerk
stMoveCircularAbsRadius_in.dSpearHeadPoint[0]       = 5000;//Absolute radius positions for 
each dimension
stMoveCircularAbsRadius_in.dSpearHeadPoint[1]       = 5000;//Absolute radius positions for 
each dimension
stMoveCircularAbsRadius_in.fTransitionParameter[0]  = 10000;           //Transition 
parameters which characterize the contour curve
stMoveCircularAbsRadius_in.fTransitionParameter[1]  = 10000;           //Transition 
parameters which characterize the contour curve
stMoveCircularAbsRadius_in.ePathChoice              = MC_CLOCKWISE;    //Supported path 
choice
stMoveCircularAbsRadius_in.eArcShortLong            = MC_SHORT;//Defines the types of 
supported arc length
stMoveCircularAbsRadius_in.eCoordSystem             = MC_MCS_COORD;//Supported coordinate 
system
stMoveCircularAbsRadius_in.eTransitionMode          = 
MC_TM_DEFINED_VELOCITY_MODE;//Supported transition modes
```

```
stMoveCircularAbsRadius_in.eBufferMode                = MC_BUFFERED_MODE;//Defines the
behavior of the axis
stMoveCircularAbsRadius_in.dEndPoint[0]               = 1000.0;//Absolute end point positions
stMoveCircularAbsRadius_in.dEndPoint[1]               = 1000.0;//Absolute end point positions
stMoveCircularAbsRadius_in.fVelocity                  = 5000.0;        //Maximum Velocity of
the path
stMoveCircularAbsRadius_in.ucSuperimposed             = 1;             //Option to
superimpose is operated
stMoveCircularAbsRadius_in.ucExecute                  = 1;
//
rc = MMC_MoveCircularAbsoluteRadiusCmd (hConn, iAxisRef, &stMoveCircularAbsRadius_in,
&stMoveCircularAbs_out);
if (rc != 0)
{
    HandleError() ;
}
```

5.14.8 MMC_MoveCircularAbsoluteAngle

For multi-axis systems. Commands an interpolated circular angle movement on an axes group from the actual position of the TCP. The movement may be in either +ve or –ve direction without restriction.

```
MMC_LIB_API int MMC_MoveCircularAbsoluteAngleCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVECIRCULARABSOLUTEANGLE_IN* pInParam,
OUT MMC_MOVECIRCULARABSOLUTE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MOVECIRCULARABSOLUTEANGLE_IN** group axes input structure that receives the MMC_MoveCircularAbsoluteAngle command.

pOutParam

Points to the **MMC_MOVECIRCULARABSOLUTE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveCircularAbsoluteAngle function.

Remarks

The end-point and auxiliary point, depending on the applied mode, are defined absolutely in the specified coordinate system.

Scope

Motion is only permitted in two dimensions. Only MCS coordinate system is supported.

MOVECIRCULARABSOLUTEANGLE_IN Structure

```
typedef struct{
double dCenterPoint[NC_MAX_NUM_AXES_IN_NODE];
double dAngle;
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucSuperimposed;
unsigned char ucExecute;
}MMC_MOVECIRCULARABSOLUTEANGLE_IN;
```

Parameters

dCenterPoint [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dCenterPoint can have vector array [1….3] double values in a technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dAngle

Relative angular position for the coordinate system specified by the input signal CoordSystem. Angular double value in degrees [u], which may be +ve or –ve without restriction.

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^{2}.

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control - Transition and Buffer Modes** and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting — Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered — The next function block affects the axis as soon as the previous movement is completed.

BlendingLow — The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).

BlendingPrevious — Blending with the velocity of function block 1 at the end-position of this block

BlendingNext — Blending with the velocity of function block 2 at end-position of function block1

BlendingHigh — Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVECIRCULARABSOLUTE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVECIRCULARABSOLUTE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-107 describes the function block for MMC_MoveCircularAbsoluteAngle

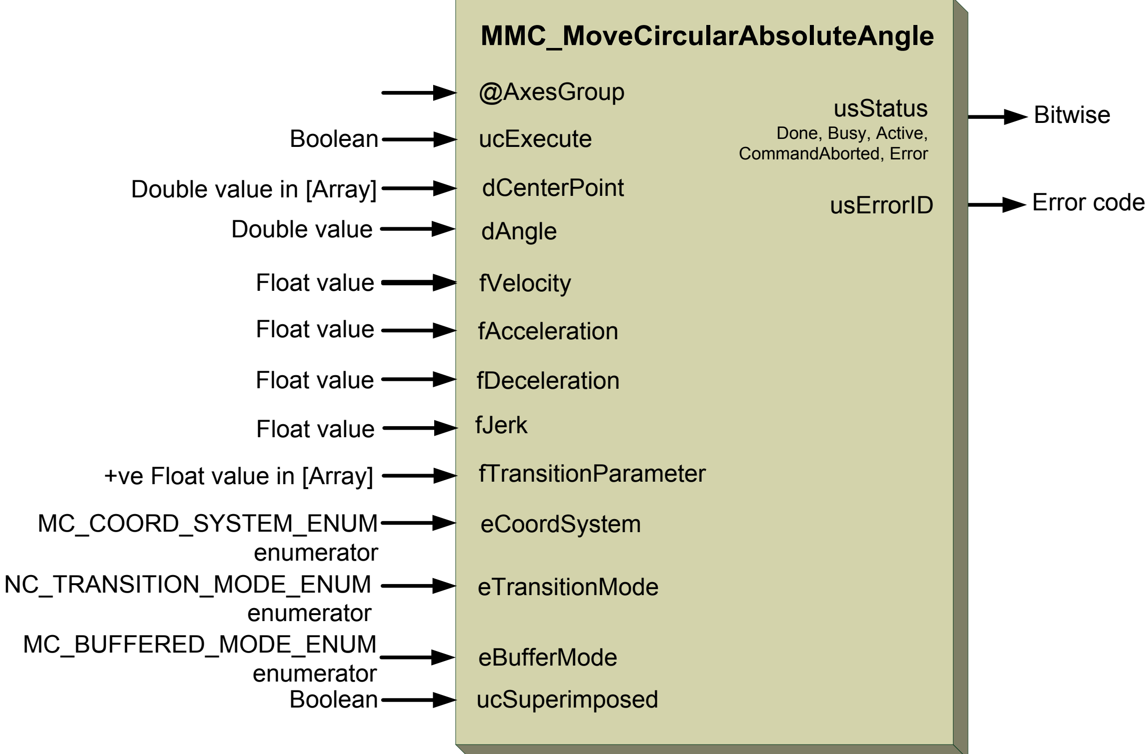


Figure 5-124: MMC_MoveCircularAbsoluteAngle function block

5.14.8.2 Function Block Code Example

```
int rc;
MMC_MOVECIRCULARABSOLUTEANGLE_IN    stMoveCircularAbsAngle_in;
MMC_MOVECIRCULARABSOLUTE_OUT        stMoveCircularAbs_out;
//
// Inserting the structure parameters:
stMoveCircularAbsAngle_in.fAcceleration          = 100000.0;   //Value of the acceleration
stMoveCircularAbsAngle_in.fDeceleration          = 100000.0;   //Value of the deceleration
stMoveCircularAbsAngle_in.fJerk                  = 20000000.0; //Maximum value of the Jerk
stMoveCircularAbsAngle_in.dCenterPoint[0]        = 5000;//Absolute center positions for each
dimension
stMoveCircularAbsAngle_in.dCenterPoint[1]        = 5000;//Absolute center positions for each
dimension
stMoveCircularAbsAngle_in.fTransitionParameter[0] = 10000; //Transition parameters which
characterize the contour curve
stMoveCircularAbsAngle_in.fTransitionParameter[1] = 10000; //Transition parameters which
characterize the contour curve
stMoveCircularAbsAngle_in.dAngle                 = 165;   //absolute positions for each
dimension in the coordinate system
stMoveCircularAbsAngle_in.eCoordSystem           = MC_MCS_COORD;    //Supported coordinate
system
stMoveCircularAbsAngle_in.eTransitionMode        = MC_TM_DEFINED_VELOCITY_MODE;  //Supported
transition modes
stMoveCircularAbsAngle_in.eBufferMode            = MC_BUFFERED_MODE; //Defines the behavior of
the axis
stMoveCircularAbsAngle_in.fVelocity              = 5000.0;         //Maximum Velocity of the
path
stMoveCircularAbsAngle_in.ucSuperimposed         = 1;          //Option to superimpose is
operated
stMoveCircularAbsAngle_in.ucExecute              = 1;
//
rc = MMC_MoveCircularAbsoluteAngleCmd (hConn, iAxisRef, &stMoveCircularAbsAngle_in,
&stMoveCircularAbs_out);
if (rc != 0)
{
    HandleError();
}
```

5.14.9 MMC_MoveLinearAbsolute

For multi-axis systems. Commands an interpolated linear movement on an axes group from the actual position of the TCP to an absolute position in the specified coordinate system.

```
MMC_LIB_API int MMC_MoveLinearAbsoluteCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVELINEARABSOLUTE_IN* pInParam,
OUT MMC_MOVELINEARABSOLUTE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVELINEARABSOLUTE_IN** group axes input structure that receives the MMC_MoveLinearAbsolute command.

pOutParam

Points to the **MMC_MOVELINEARABSOLUTE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveLinearAbsolute function.

Remarks

None

Scope

All

MMC_MOVELINEARABSOLUTE_IN Structure

```
typedef struct{
double dbPosition[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucSuperimposed;
unsigned char ucExecute;
}MMC_MOVELINEARABSOLUTE_IN;
```

Parameters

dbPosition [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute end positions for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbPosition is a vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes** Motion Control - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

	MC_BLENDING_HIGH_MODE = 6
Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVELINEARABSOLUTE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVELINEARABSOLUTE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-108 describes the function block for MMC_MoveLinearAbsolute as applied within the IEC 61131 programming.

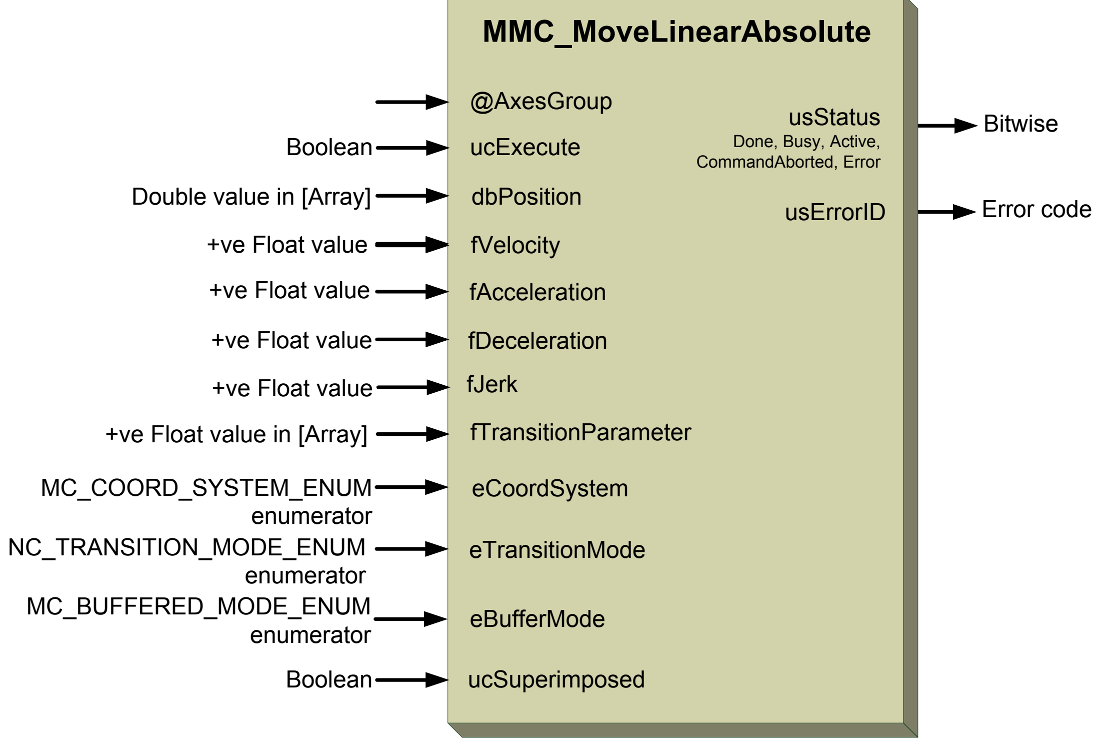


Figure 5-125: MMC_MoveLinearAbsolute function block

5.14.9.2 Function Block Code Example

```
int rc;
MMC_MOVELINEARABSOLUTE_IN      stMoveLinearAbs_in;
MMC_MOVELINEARABSOLUTE_OUT     stMoveLinearAbs_out;
//
// Inserting the structure parameters:
stMoveLinearAbs_in.fAcceleration        = 100000.0;       //Value of the acceleration
stMoveLinearAbs_in.fDeceleration        = 100000.0;       //Value of the deceleration
stMoveLinearAbs_in.fJerk                = 20000000.0;     //Maximum value of the Jerk
stMoveLinearAbs_in.dbPosition[0]        = 5000;           //Absolute position for each
dimension
stMoveLinearAbs_in.dbPosition[1]        = 5000;           //Absolute position for each
dimension
stMoveLinearAbs_in.fTransitionParameter[0]= 10000;            //Transition parameters which
characterize the contour curve
stMoveLinearAbs_in.fTransitionParameter[1]= 10000;            //Transition parameters which
characterize the contour curve
stMoveLinearAbs_in.eCoordSystem         = MC_MCS_COORD;   //Supported coordinate system
stMoveLinearAbs_in.eTransitionMode      = MC_TM_DEFINED_VELOCITY_MODE;//Supported transition
modes
stMoveLinearAbs_in.eBufferMode          = MC_BUFFERED_MODE;//Defines the behavior of the axis
stMoveLinearAbs_in.fVelocity            = 5000.0;         //Maximum Velocity of the path
stMoveLinearAbs_in.ucSuperimposed       = 1;              //Option to superimpose is operated
stMoveLinearAbs_in.ucExecute            = 1;
//
rc = MMC_MoveLinearAbsoluteCmd (hConn, iAxisRef, &stMoveLinearAbs_in, &stMoveLinearAbs_out);
if (rc != 0)
{
    HandleError();
}
```

5.14.9.3 Implementation Example

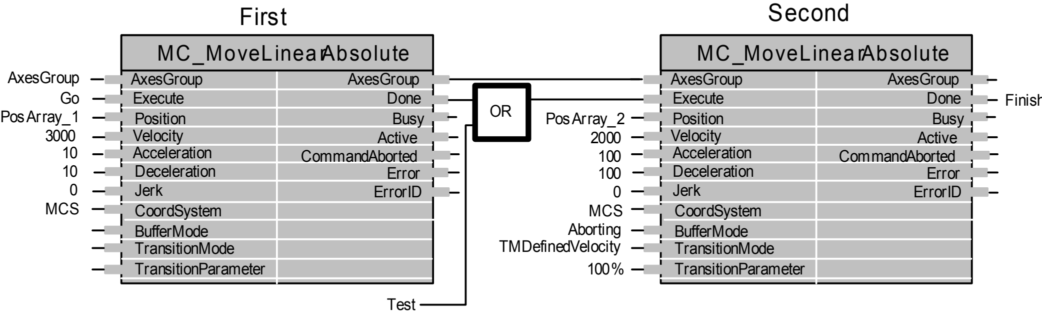


Figure 5-126: MMC_MoveLinearAbsolute - Example

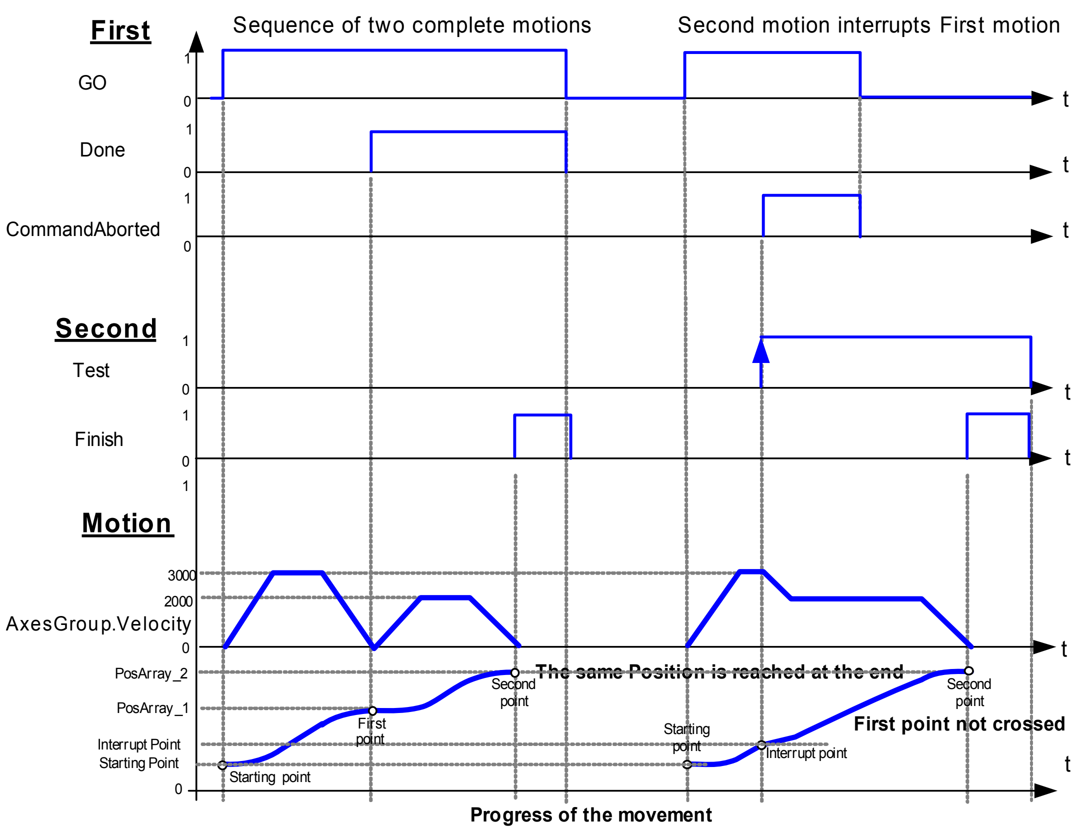


Figure 5-127: MMC_MoveLinearAbsolute timing diagram - Example

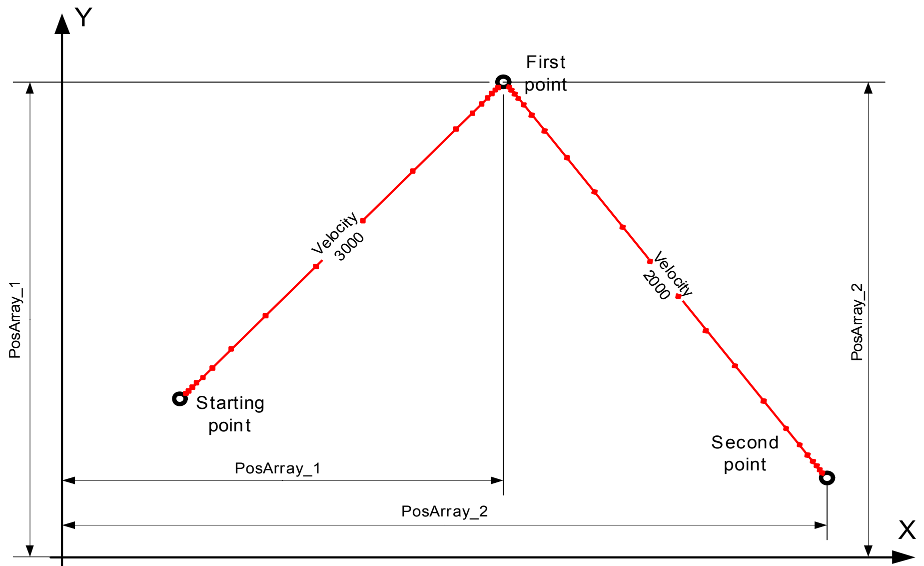


Figure 5-128: Sequence of two complete motions (Done>Execute)

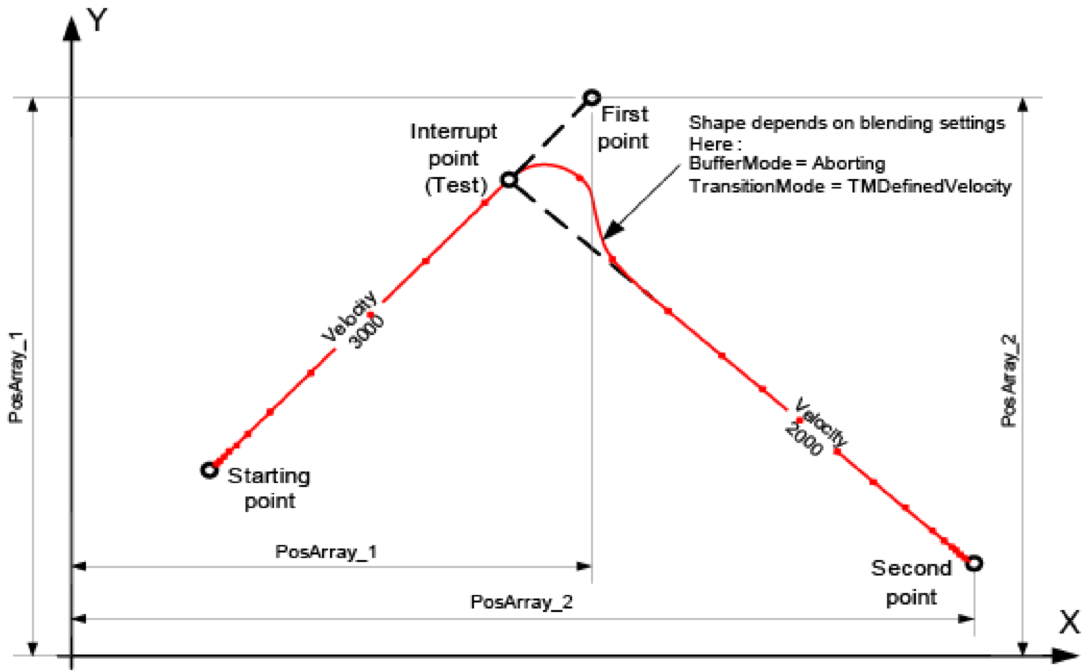


Figure 5-129: Second motion interrupts first motion

5.14.10 MMC_MoveLinearRelative

For multi-axis systems. Commands an interpolated linear movement on an axes group from the actual position of the TCP to a relative distance in the specified coordinate system.

```
MMC_LIB_API int MMC_MoveLinearRelativeCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVELINEARRELATIVE_IN* pInParam,
OUT MMC_MOVELINEARRELATIVE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVELINEARRELATIVE_IN** group axes input structure that receives the MMC_MoveLinearRelative command.

pOutParam

Points to the **MMC_MOVELINEARRELATIVE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveLinearRelative function.

Remarks

None

Scope

All

MMC_MOVELINEARRELATIVE_IN Structure

```
typedef struct{
double dbDistance[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucSuperimposed;
unsigned char ucExecute;
}MMC_MOVELINEARRELATIVE_IN;
```

Parameters

dbDistance [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation.**

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

	MC_BLENDING_HIGH_MODE = 6
Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVELINEARRELATIVE_OUT Structure

```
typedef struct mmc_movealinearrelative_out{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVELINEARRELATIVE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-113 describes the function block for MMC_MoveLinearRelative as applied within the IEC 61131 programming.

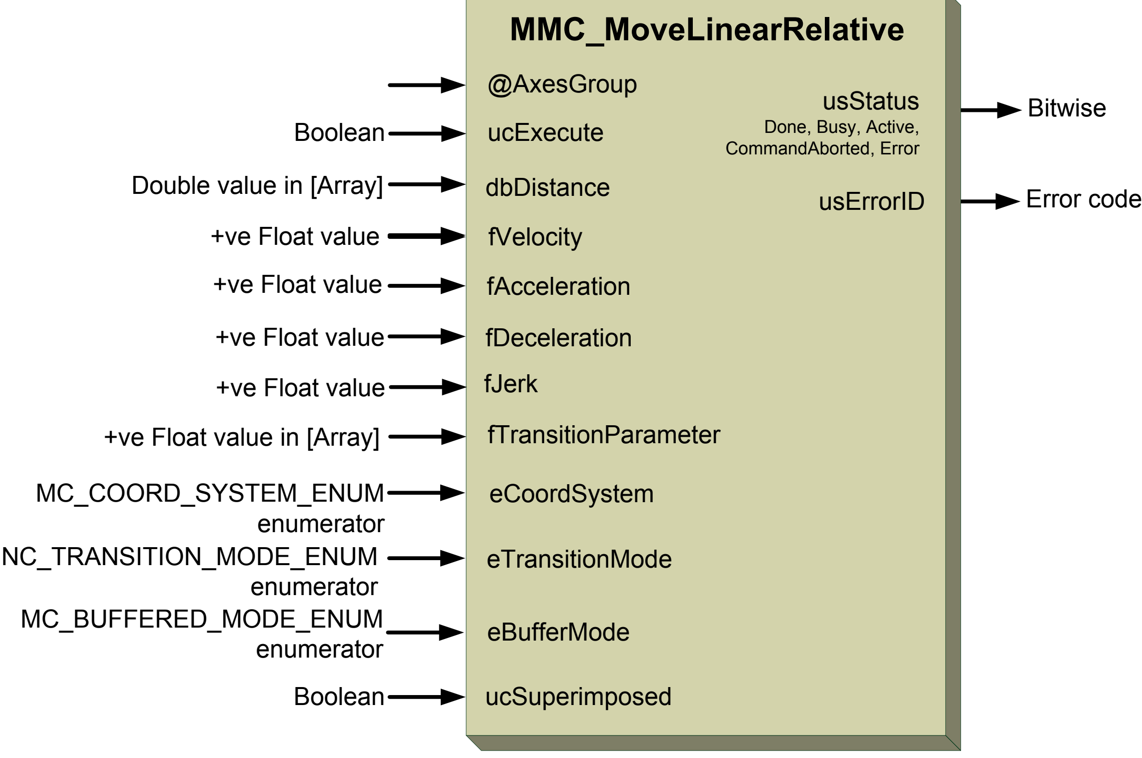


Figure 5-130: MMC_MoveLinearRelative function block

5.14.10.2 Function Block Code Example

```
int rc;
MMC_MOVELINEARRELATIVE_IN       stMoveLinearRel_in;
MMC_MOVELINEARRELATIVE_OUT      stMoveLinearRel_out;
//
// Inserting the structure parameters:
stMoveLinearRel_in.fAcceleration            = 100000.0;          //Value of the acceleration
stMoveLinearRel_in.fDeceleration            = 100000.0;          //Value of the deceleration
stMoveLinearRel_in.fJerk                    = 20000000.0;        //Maximum value of the Jerk
stMoveLinearRel_in.dbDistance[0]            = 10000;      //Absolute position for each
dimension
stMoveLinearRel_in.dbDistance[1]            = 10000;      //Absolute position for each
dimension
stMoveLinearRel_in.fTransitionParameter[0]  = 10000;             //Transition parameters
which characterize the contour curve
MoveLinearRel_in.fTransitionParameter[1]    = 10000;             //Transition parameters
which characterize the contour curve
stMoveLinearRel_in.eCoordSystem             = MC_MCS_COORD;      //Supported coordinate
system
stMoveLinearRel_in.eTransitionMode          = MC_TM_DEFINED_VELOCITY_MODE;  //Supported
transition modes
stMoveLinearRel_in.eBufferMode              = MC_BUFFERED_MODE;  //Defines the behavior of
the axis
stMoveLinearRel_in.fVelocity                = 5000.0;            //Maximum Velocity of the
path
stMoveLinearRel_in.ucSuperimposed           = 1;             //Option to superimpose is
operated
stMoveLinearRel_in.ucExecute                = 1;
//
rc = MMC_MoveLinearRelativeCmd (hConn, iAxisRef, &stMoveLinearRel_in, &stMoveLinearRel_out);
if (rc != 0)
{
    HandleError();
}
```

5.14.10.3 Implementation Example

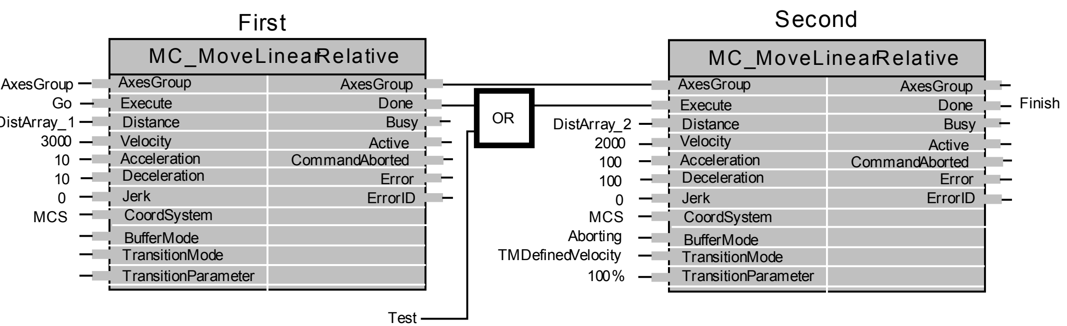


Figure 5-131: MMC_MoveLinearRelative - Example

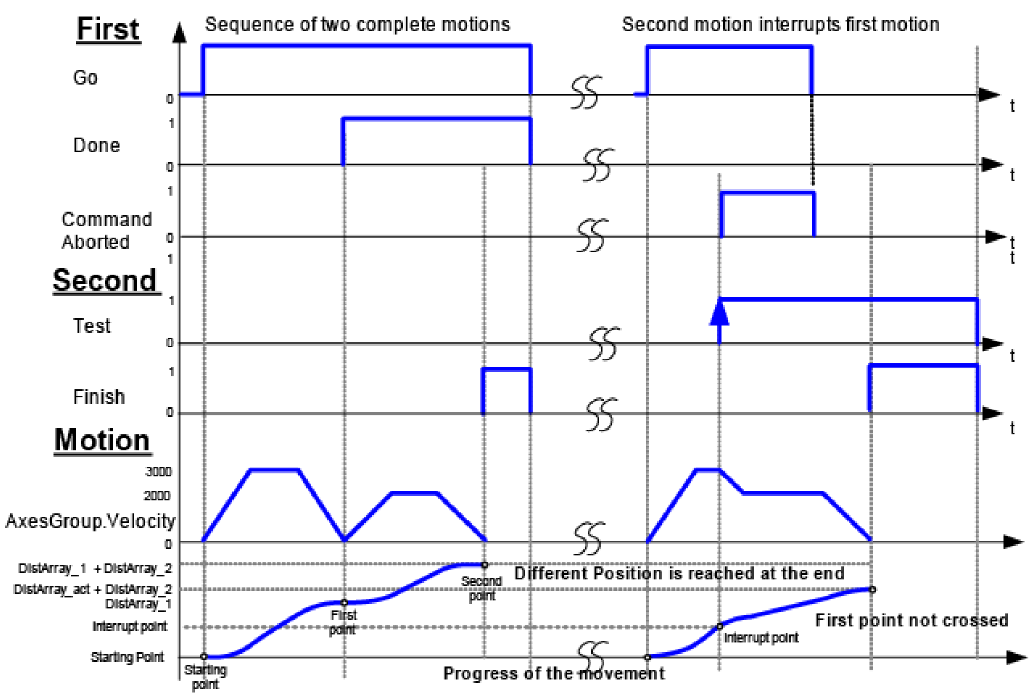


Figure 5-132: Timing diagram for MMC_MoveLinearRelative – Example

Timing diagram for example above (the dots on the red line are based on the same timing difference and representing the velocity)

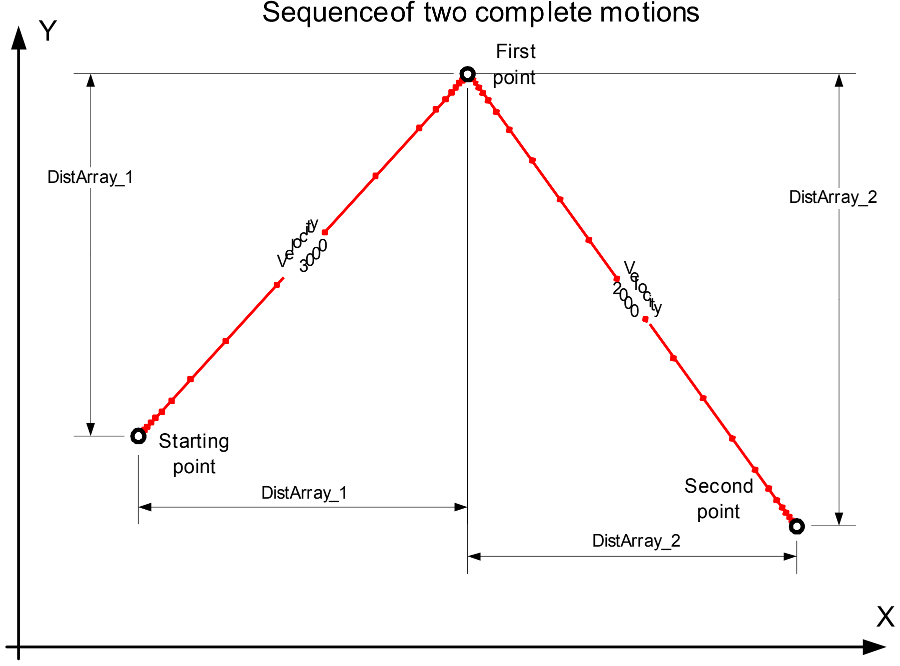


Figure 5-133: Sequence of two complete motions

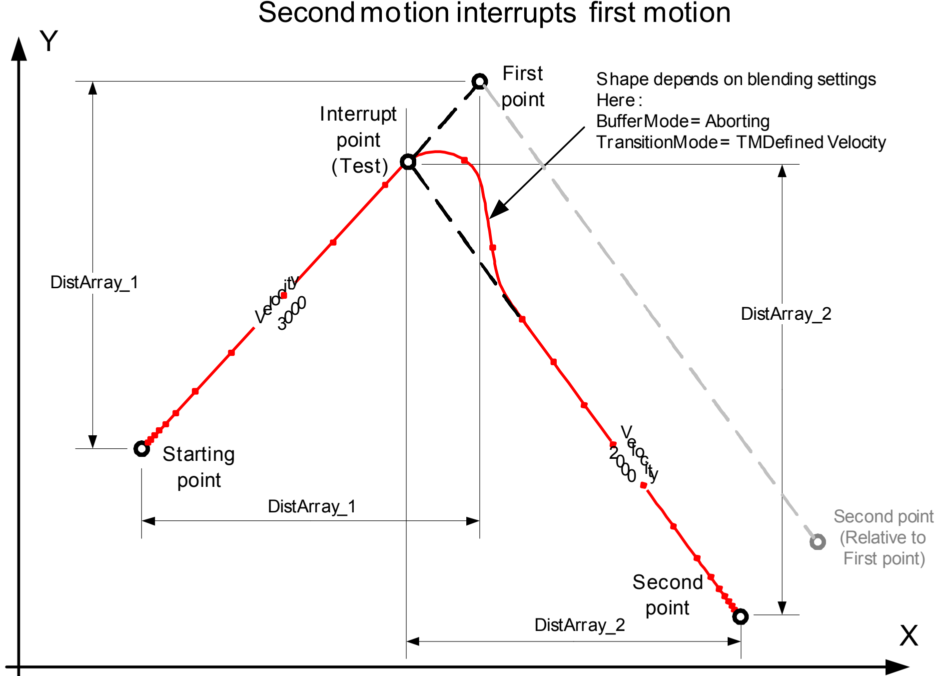


Figure 5-134: Sequence where second motion interrupts first motion

5.14.11 MMC_MoveLinearAdditive

For multi-axis systems. Commands an interpolated linear movement on an axes group from the actual position of the TCP to an additive position in the specified coordinate system.

```
MMC_LIB_API int MMC_MoveLinearAdditiveCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVELINEARADDITIVE_IN* pInParam,
OUT MMC_MOVELINEARADDITIVE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Not Supported
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVELINEARADDITIVE_IN** group axes input structure that receives the MMC_MoveLinearAdditive command.

pOutParam

Points to the **MMC_MOVELINEARADDITIVE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveLinearAdditive function.

Remarks

None

Scope

All

MMC_MOVELINEARADDITIVE_IN Structure

```
typedef struct{
double dbDistance[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity
float fAcceleration
float fDeceleration
float fJerk
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE]
MC_COORD_SYSTEM_ENUM eCoordSystem
NC_TRANSITION_MODE_ENUM eTransitionMode
MC_BUFFERED_MODE_ENUM eBufferMode
unsigned char ucSuperimposed
unsigned char ucExecute
}MMC_MOVELINEARADDITIVE_IN;
```

Parameters

dbDistance [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units accuracy to 4 bytes only, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

MC_BLENDING_HIGH_MODE = 6

Aborting — Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered — The next function block affects the axis as soon as the previous movement is completed.

BlendingLow — The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with

	the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVELINEARADDITIVE_OUT Structure

```
typedef struct mmc_movealinearadditive_out{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVELINEARADDITIVE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-118 describes the function block for MMC_MoveLinearAdditive as applied within the IEC 61131 programming.

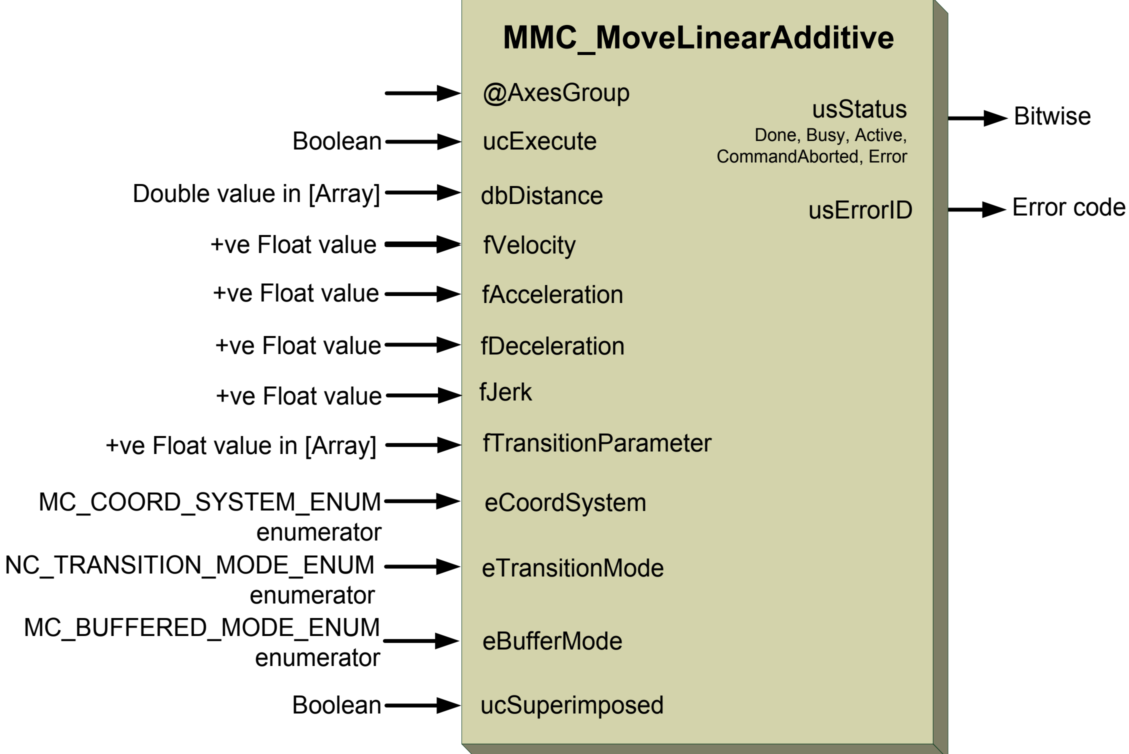


Figure 5-135: MMC_MoveLinearAdditive function block

5.14.12 MMC_MoveLinearAdditiveEx

For multi-axis systems. Commands an extended interpolated linear movement on an axes group from the actual position of the TCP to an additive accurate position in the specified coordinate system.

```
MMC_LIB_API int MMC_MoveLinearAdditiveExCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVELINEARADDITIVEEX_IN* pInParam,
OUT MMC_MOVELINEARADDITIVEEX_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Not Supported
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVELINEARADDITIVEEX_IN** group axes input structure that receives the MMC_MoveLinearAdditiveEx command.

pOutParam

Points to the **MMC_MOVELINEARADDITIVEEX_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveLinearAdditiveEx function.

Remarks

None

Scope

All

MMC_MOVELINEARADDITIVEEX_IN Structure

```
typedef struct{
double dbDistance[NC_MAX_NUM_AXES_IN_NODE
double dVelocity
double dAcceleration;
double dDeceleration;
double dJerk
double dTransitionParameter[NC_MAX_NUM_AXES_IN_NODE
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode
unsigned char ucSuperimposed
unsigned char ucExecute;
}MMC_MOVELINEARADDITIVEEX_IN;
```

Parameters

dbDistance [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive double value in u/s

dAcceleration

Value of the acceleration (increasing energy of the motor). Any positive double value in $u/s^{2.}$

dDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive double value in u/s^2

dJerk

Maximum double value of the Jerk. Any positive value in u/s^3

dTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dTransitionParameter can have any positive double value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes** Motion Control - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

MC_BLENDING_HIGH_MODE = 6

Aborting Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered The next function block affects the axis as soon as the previous movement is completed.

BlendingLow The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis

	will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVELINEARADDITIVEEX_OUT Structure

```
typedef struct mmc_movealinearadditiveex_out{
unsigned int uiHndl
unsigned short usStatus;
short sErrorID;
}MMC_MOVELINEARADDITIVEEX_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-119 describes the function block for MMC_MoveLinearAdditiveEx.

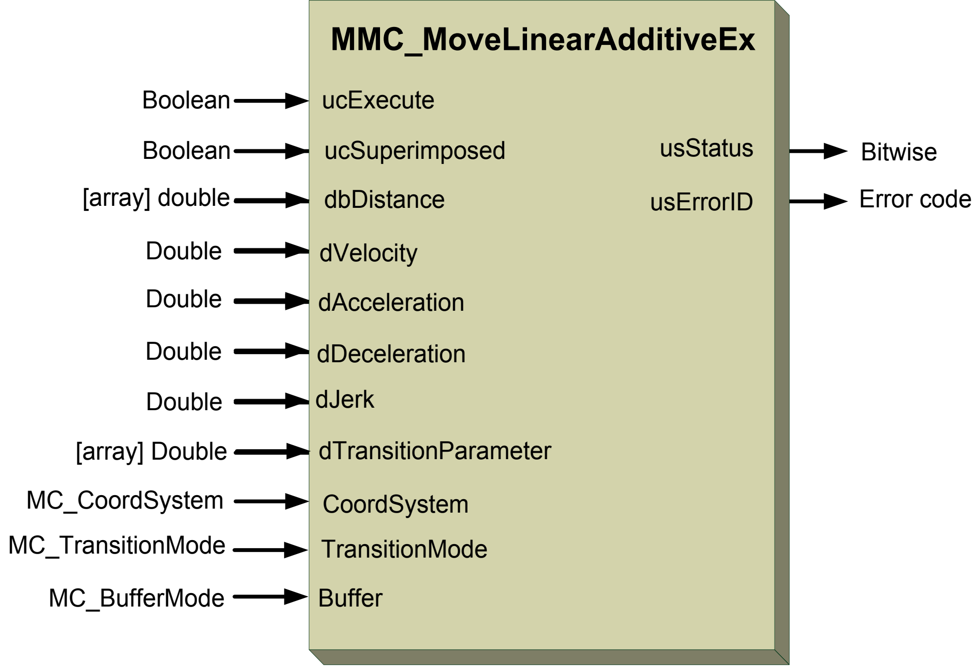


Figure 5-136: MMC_MoveLinearAdditiveEx function block

5.14.13 MMC_MoveLinearAbsoluteRepetitive

For multi-axis systems. Commands an interpolated repetitive linear movement on an axes group vector to an absolute point given as an input in the specified coordinate system.

```
MMC_LIB_API int MMC_MoveLinearAbsoluteRepetitiveCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVELINEARABSOLUTEREPETITIVE_IN* pInParam,
OUT MMC_MOVELINEARABSOLUTEREPETITIVE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Not Supported
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVELINEARABSOLUTEREPETITIVE_IN** group axes input structure that receives the MMC_MoveLinearAbsoluteRepetitive command.

pOutParam

Points to the **MMC_MOVELINEARRELATIVEREPETITIVE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveLinearAbsoluteRepetitive function.

Remarks

It should be noted that all repetitive mode motion will not operate in Abort Mode.

Scope

This function only operates when the vector state is in Standby mode. Calling MMC_Stop, MMC_Halt in aborting mode will stop the motion.

MMC_MOVELINEARABSOLUTEREPETITIVE_IN Structure

```
typedef struct mmc_movelinearabsoluterepetitive_in{
double dbPosition[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned int uiExecDelayMs;
unsigned char ucSuperImposed;
unsigned char ucExecute;
}MMC_MOVELINEARABSOLUTEREPETITIVE_IN;
```

Parameters

dbPosition [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute end positions for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbPosition is a vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant

on the TransitionMode parameter. Refer to the section ***5.10.1 Coordinate System and kinematic transformation*.**

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes** Motion Control - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

MC_BLENDING_HIGH_MODE = 6

Aborting Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered The next function block affects the axis as soon as the previous movement is completed.

BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

uiExecDelayMs

The delay in execution of the next action (in msecs). Any +ve integer value.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVELINEARABSOLUTEREPETITIVE_OUT Structure

```
typedef struct mmc_movelinearabsoluterepetitive_out{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVELINEARABSOLUTEREPETITIVE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-120 describes the function block for MMC_MoveLinearAbsoluteRepetitive

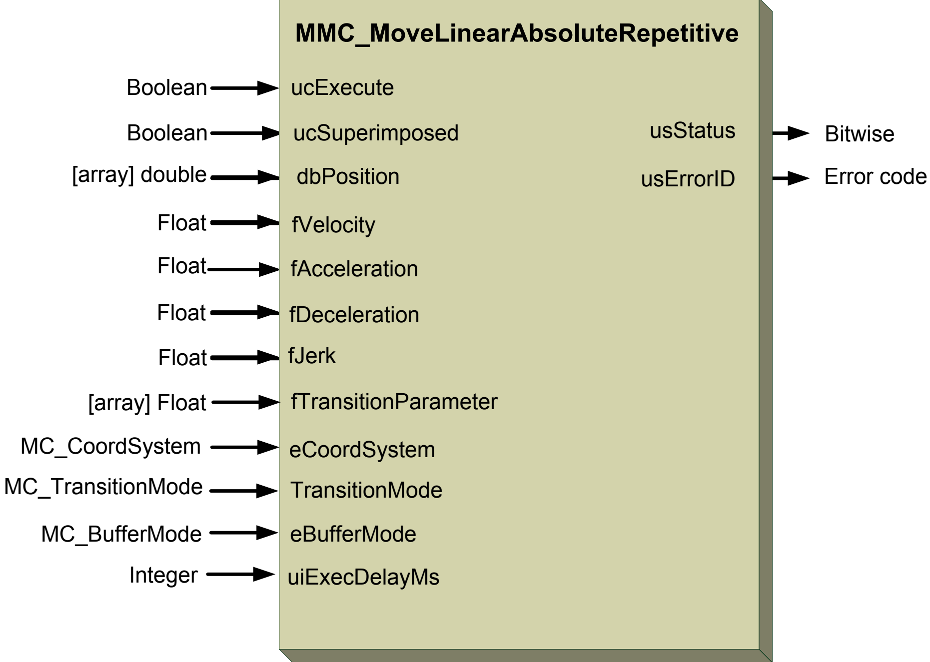


Figure 5-137: MMC_MoveLinearAbsoluteRepetitive function block

5.14.13.2 Function Block Code Example

```
int rc;
MMC_MOVELINEARABSOLUTEREPETITIVE_IN      stMoveLinAbsRepIn;
MMC_MOVELINEARABSOLUTEREPETITIVE_OUT     stMoveLinAbsRepOut;
// Suppose we are working in 3D
stMoveLinAbsRepIn.dbPosition[0]              = 100000.0;
stMoveLinAbsRepIn.dbPosition[1]              = 100000.0;
stMoveLinAbsRepIn.dbPosition[2]              = 100000.0;
stMoveLinAbsRepIn.eBufferMode                = MC_BUFFERED_MODE_ENUM;
stMoveLinAbsRepIn.eTransitionSystem          = MC_TM_NONE_MODE;
stMoveLinAbsRepIn.fTransitionParameter[0]    = 0.0;
stMoveLinAbsRepIn.fAcceleration              = 1000000.0;
stMoveLinAbsRepIn.fDeceleration              = 1000000.0;
stMoveLinAbsRepIn.fJerk                      = 10000000.0;
stMoveLinAbsRepIn.fVelocity                  = 100000.0;
stMoveLinAbsRepIn.ucExecute                  = 1;
stMoveLinAbsRepIn.uiExecDelayMs              = 0;
//
rc = MMC_MoveLinearAbsoluteRepetitiveCmd (hConn, iAxisRef, &stMoveLinAbsRepIn,
&stMoveLinAbsRepOut);
if (rc != 0)
{
HandleError();
}
```

5.14.14 MMC_MoveLinearRelativeRepetitive

For multi-axis systems. Commands an interpolated repetitive linear movement on an axes group vector to a relative distance from actual position given as an input in the specified coordinate system.

```
MMC_LIB_API int MMC_MoveLinearRelativeRepetitiveCmd
(IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVELINEARRELATIVEREPETITIVE_IN* pInParam,
OUT MMC_MOVELINEARRELATIVEREPETITIVE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_MOVELINEARRELATIVEREPETITIVE_IN** group axes input structure that receives the MMC_MoveLinearRelativeRepetitive command.

pOutParam

Points to the **MMC_MOVELINEARRELATIVEREPETITIVE_OUT** group axes output structure receiving information, as a result of calling the MMC_MoveLinearRelativeRepetitive function.

Remarks

It should be noted that all repetitive mode motion will not operate in Abort Mode.

Scope

This function only operates when the vector state is in Standby mode. Calling MMC_Stop, MMC_Halt in aborting mode will stop the motion.

MMC_MOVELINEARRELATIVEREPETITIVE_IN Structure

```
typedef struct mmc_movelinearrelativerepetitive_in{
double dbDistance[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_COORD_SYSTEM_ENUM eCoordSystem;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned int uiExecDelayMs;
unsigned char ucSuperImposed;
unsigned char ucExecute;
}MMC_MOVELINEARRELATIVEREPETITIVE_IN;
```

Parameters

dbDistance[NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^{2}.

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD	= 0
MC_ACS_COORD	= 1
MC_MCS_COORD	= 2
MC_PCS_COORD	= 3

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

uiExecDelayMs

The delay in execution of the next action (in msecs). Any +ve integer value.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVELINEARRELATIVEREPETITIVE_OUT Structure

```
typedef struct mmc_movelinearrelativerepetitive_out{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVELINEARRELATIVEREPETITIVE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-121 describes the function block for MMC_MoveLinearRelativeRepetitive

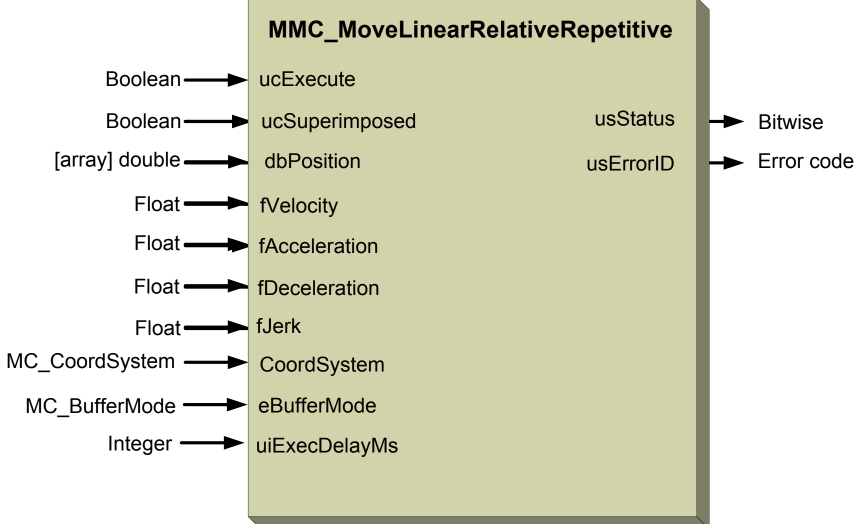


Figure 5-138: MMC_MoveLinearRelativeRepetitive function block

5.14.14.2 Function Block Code Example

```
int rc;
MMC_MOVELINEARRELATIVEREPETITIVE_IN      stMoveLinRelRepIn;
MMC_MOVELINEARRELATIVEREPETITIVE_OUT     stMoveLinAbsRepOut;
// Suppose we are working in 3D
stMoveLinRelRepIn.dbDistance[0]          = 100000.0;
stMoveLinRelRepIn.dbDistance[1]          = 100000.0;
stMoveLinRelRepIn.dbDistance[2]          = 100000.0;
stMoveLinRelRepIn.eBufferMode            = MC_BUFFERED_MODE_ENUM;
stMoveLinRelRepIn.eTransitionSystem      = MC_TM_NONE_MODE;
stMoveLinRelRepIn.fTransitionParameter[0]= 0.0;
stMoveLinRelRepIn.fAcceleration          = 1000000.0;
stMoveLinRelRepIn.fDeceleration          = 1000000.0;
stMoveLinRelRepIn.fJerk                  = 10000000.0;
stMoveLinRelRepIn.fVelocity              = 100000.0;
stMoveLinRelRepIn.ucExecute              = 1;
stMoveLinRelRepIn.uiExecDelayMs          = 0;
//
rc = MMC_MoveLinearRelativeRepetitiveCmd (hConn, iAxisRef, &stMoveLinRelRepIn,
&stMoveLinRelRepOut);
if (rc != 0)
{
HandleError();
}
```

5.14.15 MMC_MovePolynomAbsolute

For multi-axis systems and complex motion sequences where the Polynomial expression is relevant. This function sends a Move Polynom Absolute command to the MMC server for specific Vector. Refer to the section **5.12.6 Move Polynomial Function Block**.

```
MMC_LIB_API int MMC_MovePolynomAbsoluteCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVEPOLYNOMABSOLUTE_IN* pInParam,
OUT MMC_MOVEPOLYNOMABSOLUTE_OUT* pOutParam
);
```

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command.

hAxisRef

Vector reference handle type returned by GetVectorRef command

pInParam

Points to the **MMC_MOVEPOLYNOMABSOLUTE_IN** group axes input structure that receives the MMC_MovePolynomAbsolute command.

pOutParam

Points to the **MMC_MOVEPOLYNOMABSOLUTE_OUT** group axes output structure receiving information, as a result of calling the MMC_MovePolynomAbsolute function.

Remarks

Scope

MMC_MOVEPOLYNOMABSOLUTE_IN Structure

```
typedef struct{
double dbAuxPoint[NC_MAX_NUM_AXES_IN_NODE];
double dbEndPoint[NC_MAX_NUM_AXES_IN_NODE];
double dVelocity;
double dAcceleration;
double dDeceleration;
double dJerk;
MC_COORD_SYSTEM_ENUM eCoordSystem;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucSuperimposed
unsigned char ucExecute;
}MMC_MOVEPOLYNOMABSOLUTE_IN;
```

Parameters

dbAuxPoint[NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dbEndPoint[NC_MAX_NUM_AXES_IN_NODE]

Array [1..16] of absolute target positions for each dimension in the specified coordinate system.

dVelocity

Maximum Velocity [u/s] for the path for the coordinate system in which the path is defined. Always positive. Not necessarily reached.

dAcceleration

Maximum acceleration. Always positive. Not necessarily reached

dDeceleration

Maximum deceleration. Always positive. Not necessarily reached

dJerk

Maximum jerk. Always positive. Not necessarily reached

eCoordSystem

Define the type of supported coordinate system. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the blended velocity with previous function block. It defines the chronological sequence of the FB relative to the previous block. Refer to section **5.11 Multiple Axes Motion Control - Transition and Buffer Modes**.

ucSuperimposed

Not in use.

ucExecute

Refer to the section **2.2 Maestro Operation Modes**. Boolean TRUE/FALSE values.

MMC_MOVEPOLYNOMABSOLUTE_OUT Structure

```
typedef struct{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_MOVEPOLYNOMABSOLUTE_OUT;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-122 describes the function block for MMC_MovePolynomAbsolute as applied within the IEC 61131 programming.

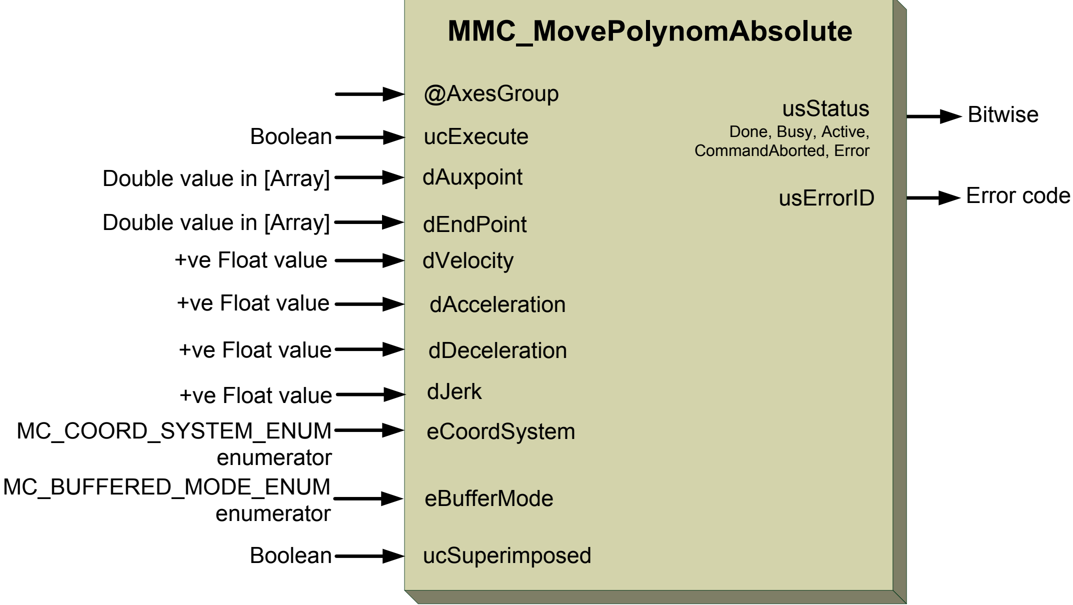


Figure 5-139: MMC_MovePolynomAbsolute function block

5.14.16 MMC_PathSelect

For multi-axis systems. Reads splines data from a file and calculates the optimal path.

```
MMC_LIB_API int MMC_PathSelectCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_PATHSELECT_IN* pInParam,
OUT MMC_PATHSELECT_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Not Supported
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Group reference handle type returned by GetAxisRef command in the vector motion.

pInParam

Points to the **MMC_PATHSELECT_IN** input data structure using the MMC_PathSelect function.

pOutParam

Points to the **MMC_PATHSELECT_OUT** output structure receiving information, as a result of calling the MMC_PathSelect function.

Remarks

This function obtains the spline file from the user, calculates the trajectory and stores it in a specially reserved memory. In other words, this function prepares the trajectory for MMC_MovePathCmd().

Scope

All

MMC_PATHSELECT_IN Structure

```
typedef struct mmc_pathselect_in{
MC_COORD_SYSTEM_ENUM eCoordSystem;
MC_PATH_DATA_REF pPathToSplineFile;
unsigned char ucExecute;
}MMC_PATHSELECT_IN;
```

Parameters

eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator. The options are:

MC_NONE_COORD	= 0
MC_ACS_COORD	= 1
MC_MCS_COORD	= 2
MC_PCS_COORD	= 3

pPathToSplineFile

This string describes where the splines data file is located, the absolute path. Values accepted are any characters describing a file path, for example: /mnt/jffs/usr/Spline_2D.txt

MC_PATH_DATA_REF	Where MC_PATH_DATA_REF is a string of NC_MAX_SPLINES_FILE_PATH_LENGTH that defines the maximum length of the splines file path data. MC_PATH_DATA_REF is an alias for unsigned integer and can have values that are allowed by Linux for a file name. NC_MAX_SPLINES_FILE_PATH_LENGTH is a constant of value 100.

ucExecute

Start the execution command. Boolean TRUE/FALSE values. Currently not in use, and set to 1.

MMC_PATHSELECT_OUT Structure

```
typedef struct mmc_pathselect_out{
unsigned short usStatus;
short sErrorID;
MC_PATH_REF hMemHandle;
}MMC_PATHSELECT_OUT;
```

Parameters

hMemHandle

MC_PATH_REF alias for unsigned integer with values and handle to a journal entry where the pointer to the shared memory is located, and indicates the memory area where the spline is allocated. In fact this is the unique identifier between PathSelect, MovePath and PathUnselect commands. MC_PATH_REF is the journal entry path reference.

hMemHandle produces +ve Integer values.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-123 describes the function block for MMC_PathSelect as applied within the IEC 61131 programming.

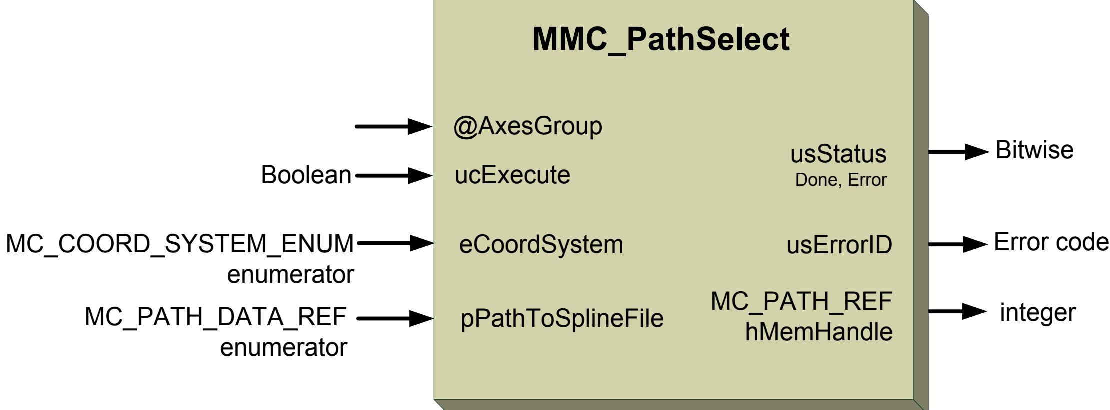


Figure 5-140: MMC_PathSelect function block

5.14.16.2 Function Block Code Example

```
int rc;
MMC_PATHSELECT_IN      stPathSelectIn;
MMC_PATHSELECT_OUT     stPathSelectOut;
// Inserting the structure parameters:
stPathSelectIn.eCoordSystem = MC_MCS_COORD;
stPathSelectOut.sErrorID = 0;
strcpy(stPathSelectIn.pPathToSplineFile, "/mnt/jffs/usr/Helix.txt");
stPathSelectIn.ucExecute = 1;
retval = 0;
retval = MMC_PathSelectCmd(g_conn_hndl, g_vect_ref[vect_idx], &stPathSelectIn,
&stPathSelectOut);
if (0 > retval)
{
printf("Error ID: %d", (short )stPathSelectOut.sErrorID);
return -1;
}
```

5.14.17 MMC_MovePath

For multi-axis systems. Moves a group of drives along a previously defined spline path.

```
MMC_LIB_API int MMC_MovePathCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_MOVEPATH_IN* pInParam,
OUT MMC_MOVEPATH_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

GMAS Programming(IEC 61331 Program)\ElmoGroupAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Group reference handle type returned by GetAxisRef command in the vector motion.

pInParam

Points to the **MMC_MOVEPATH_IN** input data structure using the MMC_MovePath function.

pOutParam

Points to the **MMC_MOVEPATH_OUT** output structure receiving information, as a result of calling the MMC_MovePath function.

Remarks

Note that only buffered mode, motion from STANDBY state, with no transitions, is currently supported.

Scope

All

MMC_MOVEPATH_IN Structure

```
typedef struct mmc_movepath_in{
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
MC_PATH_REF hMemHandle;
unsigned char ucSuperImposed;
unsigned char ucExecute;
} MMC_MOVEPATH_IN;
```

Parameters

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator. The options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes** Motion Control - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows, but only the Buffered Mode is supported:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

MC_BLENDING_HIGH_MODE = 6

Aborting	The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

hMemHandle

MC_PATH_REF index handle to a journal entry where the pointer to the shared memory is located obtained from the PathSelect command. MC_PATH_REF is the journal entry path reference.

hMemHandle has Integer values.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE. Not currently in use.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVEPATH_OUT Structure

```
typedef struct mmc_movepath_out{
unsigned int uiHandle;
unsigned short usStatus;
short sErrorID;
}MMC_MOVEPATH_OUT;
```

Parameters

uiHandle

Returned function block handle. Integer with any +ve value

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-124 describes the function block for MMC_MovePath as applied within the IEC 61131 programming.

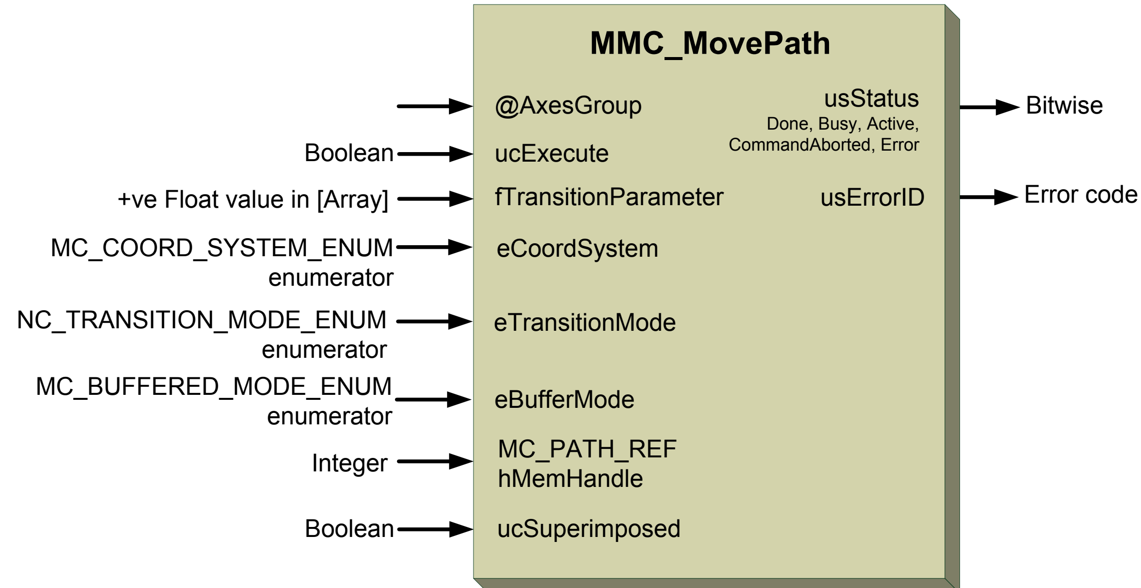


Figure 5-141: MMC_MovePath function block

5.14.17.2 Function Block Code Example

```
int rc;
MMC_MOVEPATH_IN   stMovePathIn;
MMC_MOVEPATH_OUT  stMovePathOut;
// Inserting the structure parameters:
stMovePathIn.fTransitionParameter[0] = 0.0;
stMovePathIn.eCoordSystem = MC_MCS_COORD;
stMovePathIn.eTransitionMode = MC_TM_NONE_MODE;
stMovePathIn.eBufferMode = MC_BUFFERED_MODE;
stMovePathIn.hMemHandle = stPathSelectOut.hMemHandle;
stMovePathIn.ucSuperImposed = 0;
stMovePathIn.ucExecute = 1;
retval = MMC_MovePathCmd(g_conn_hndl, g_vect_ref[vect_idx], &stMovePathIn, &stMovePathOut);
if (0 > retval)
{
printf("Error ID: %d", (short )stMovePathOut.sErrorID);
return -1;
}
```

5.14.17.3 Implementation Example

```
MMC_PATHSELECT_IN stPathSelectIn;
MMC_PATHSELECT_OUT stPathSelectOut;

MMC_MOVEPATH_IN stMovePathIn;
MMC_MOVEPATH_OUT stMovePathOut;

MMC_PATHUNSELECT_IN stPathUnselectIn;
MMC_PATHUNSELECT_OUT stPathUnselectOut;

int rc = 0;
stPathSelectIn.eCoordSystem = MC_MCS_COORD;
strcpy(stPathSelectIn.pPathToSplineFile, "/mnt/jffs/usr/Helix_CL_CV.txt");
stPathSelectIn.ucExecute = 1;

rc = MMC_PathSelectCmd(g_hConnectHandle, g_hVectorRef, &stPathSelectIn, &stPathSelectOut);
if (NC_OK != rc)
{
 HandleError();
 return -1;
}
stMovePathIn.eBufferMode = MC_BUFFERED_MODE;
stMovePathIn.eCoordSystem = MC_MCS_COORD;
stMovePathIn.eTransitionMode = MC_TM_NONE_MODE;
stMovePathIn.fTransitionParameter[0] = 0;
stMovePathIn.hMemHandle = stPathSelectOut.hMemHandle;
stMovePathIn.ucExecute = 1;
stMovePathIn.ucSuperImposed = 0;

rc = MMC_MovePathCmd(g_hConnectHandle, g_hVectorRef, &stMovePathIn, &stMovePathOut);
if (NC_OK != rc)
{
HandleError();
 return -1;
}
stPathUnselectIn.hMemHandle = stPathSelectOut.hMemHandle;
stPathUnselectIn.ucExecute = 1;

rc = MMC_PathUnselectCmd(g_hConnectHandle, g_hVectorRef, &stPathUnselectIn,
&stPathUnselectOut);
if (NC_OK != rc)
{
HandleError();
 return -1;
}
```

5.14.18 MMC_PathUnselect

For multi-axis systems. This function unloads the spline data table from the Maestro.

```
MMC_LIB_API int MMC_PathUnselectCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_PATHUNSELECT_IN* pInParam,
OUT MMC_PATHUNSELECT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Group reference handle type returned by GetAxisRef command in the vector motion.

pInParam

Points to the **MMC_PATHUNSELECT_IN** input data structure using the MMC_PathUnselect function.

pOutParam

Points to the **MMC_PATHUNSELECT_OUT** output structure receiving information, as a result of calling the MMC_PathUnselect function.

Remarks

None

Scope

All

MMC_PATHUNSELECT_IN Structure

typedef struct mmc_pathunselect_in{
MC_PATH_REF hMemHandle;
unsigned char ucExecute;
}MMC_PATHUNSELECT_IN

Parameters

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located, obtained from the PathSelect command. MC_PATH_REF is the journal entry path reference.

hMemHandle can have Integer values.

ucExecute

Start the execution command. Boolean TRUE/FALSE values. Currently not in use and set to 1.

MMC_PATHUNSELECT_OUT Structure

typedef struct mmc_pathunselect_out{
unsigned short usStatus;
short sErrorID;
}MMC_PATHUNSELECT_OUT;

Parameters

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-125 describes the function block for MMC_PathUnselect as applied within the IEC 61131 programming.

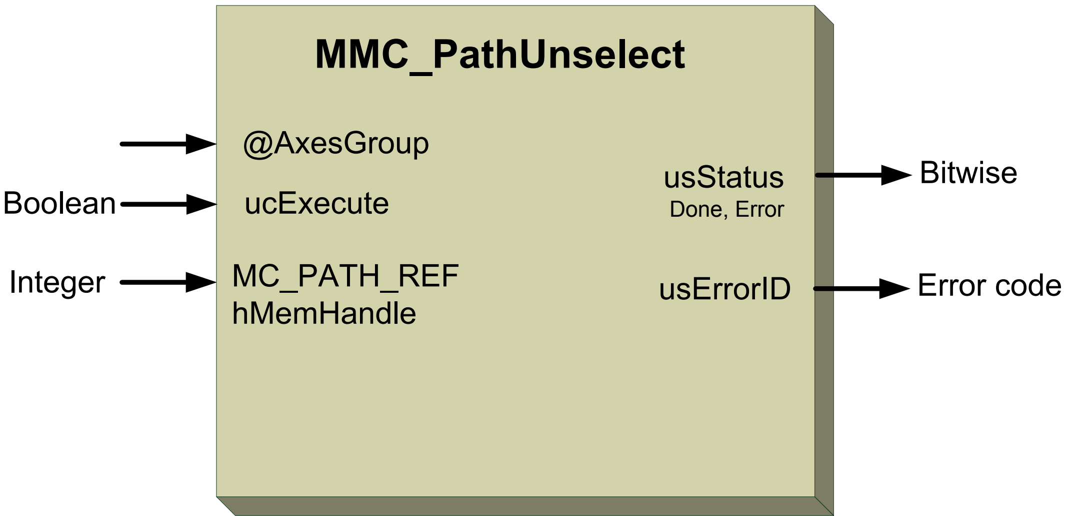


Figure 5-142: MMC_PathUnselect function block

5.14.18.2 Function Block Code Example

```
int rc;
MMC_PATHUNSELECT_IN    stPathUnselectIn;
MMC_PATHUNSELECT_OUT   stPathUnselectOut;
// Inserting the structure parameters:
stPathUnselectIn.hMemHandle = stPathSelectOut.hMemHandle;
stPathUnselectIn.ucExecute = 1;
retval = MMC_PathUnselectCmd(g_conn_hndl, g_vect_ref[vect_idx], &stPathUnselectIn,
&stPathUnselectOut);
if (0 > retval)
{
printf("Error ID: %d", (short )stPathUnselectOut.sErrorID);
return -1;
}
```

5.15 Multiple Axes Administrative Control

Administrative function blocks are blocks where no driving motion is involved. The following multiple axes administrative function blocks are described:

Multiple Axes
MMC_SetKinTransform
MMC_SetKinTransformEx
MMC_SetCartesianTransform
MMC_TrackConveyorBelt
MMC_TrackRotaryTable
MMC_SetKinTransformDelta
MMC_SetKinTransformCartesian
MMC_SetKinTransformScara
MMC_SetKinTransformThreeLink
MMC_AddAxisToGroup
MMC_GroupDisable
MMC_GroupEnable
MMC_GroupReadActualPosition
MMC_GroupReadActualVelocity
MMC_GroupReadError
MMC_GroupReadStatus
MMC_GroupReset
MMC_GroupSetOverride
MMC_GroupSetPosition
MMC_RemoveAxisFromGroup
MMC_SetKinTransform
MMC_GroupReadParameter
MMC_GroupReadBoolParameter
MMC_GroupWriteParameter
MMC_GroupWriteBoolParameter

The structured definitions from sections **5.15.2 - 5.15.9** are excluded from the above list as they are not function blocks.

5.15.1 Coordinated System and Kinematic Trnsformation Definitions

As described in section **5.10.1 Coordinate System and kinematic transformation** and onwards, the execution of a path– or any position information - in space is described by the three coordinate systems; ACS, MCS, and PCS. As a result, an enumerator and certain structures are defined which perform these inter-coordinate actions.

The enumerator NC_KIN_TYPE_ENUM defines a Cartesian and number of standard robot types, of transformations.

```
typedef enum
{
    NC_CARTESIAN_TYPE        = 0,
    NC_DELTA_ROBOT_TYPE      = 1,
    NC_SCARA_ROBOT_TYPE      = 2,
    NC_THREE_LINK_ROBOT_TYPE = 3,
    NC_LAST_KIN_TYPE         = 4,

}NC_KIN_TYPE;
```

In order to simplify the kinematic transformation process by using less complex and more standardize functions, the following wrappers for kinematics's transformation functions are introduced. These will eventually be replaced by MMC_SETKINTRANSFORMEX_IN.

```
typedef struct _mc_kintransform_delta {
    MC_KIN_REF_DELTA     stParams;
    MC_BUFFERED_MODE_ENUM eBufferMode;
    unsigned char ucExecute;
} MMC_KINTRANSFORM_DELTA_IN;
typedef MMC_SETKINTRANSFORMEX_OUT MMC_KINTRANSFORM_DELTA_OUT;
typedef struct _mc_kintransform_cartesian {
    MC_KIN_REF_CARTESIAN stParams;
    MC_BUFFERED_MODE_ENUM eBufferMode;
    unsigned char ucExecute;
} MMC_KINTRANSFORM_CARTESIAN_IN;
typedef MMC_SETKINTRANSFORMEX_OUT MMC_KINTRANSFORM_CARTESIAN_OUT;
typedef struct _mc_kintransform_scara {
    MC_KIN_REF_SCARA     stParams;
    MC_BUFFERED_MODE_ENUM eBufferMode;
    unsigned char ucExecute;
} MMC_KINTRANSFORM_SCARA_IN;
typedef MMC_SETKINTRANSFORMEX_OUT MMC_KINTRANSFORM_SCARA_OUT;
typedef struct _mc_kintransform_threelink {
    MC_KIN_REF_THREE_LINK    stParams;
    MC_BUFFERED_MODE_ENUM eBufferMode;
    unsigned char ucExecute;
} MMC_KINTRANSFORM_THREELINK_IN;

typedef MMC_SETKINTRANSFORMEX_OUT MMC_KINTRANSFORM_THREELINK_OUT;
```

Refer to the following new functions:

- MMC_SetKinTransformDelta
- MMC_SetKinTransformCartesian
- MMC_SetKinTransformScara
- MMC_SetKinTransformThreeLink

5.15.2 MC_KIN_NODE_DEF

MC_KIN_NODE_DEF Structure

```
typedef struct {
double ulTrCoef [NC_MAX_NUM_COEF];
NC_TR_FUNC_ID_ENUM iMcsToAcsFuncID;
NC_NODE_HNDL_T hNode;
NC_AXIS_IN_GROUP_TYPE_ENUM_EX eType;
} MC_KIN_NODE_DEF;
```

Description Kinetic node definition structure. This structure is not a function block.

Source GMAS\includes\MMC_PLCopen_group_API.h

Parameters

ulTrCoef [NC_MAX_NUM_COEF]

Defined by the type of supported MCS transform coefficients according to the definition of the enumerator NC_TR_COEF_TYPE_ENUM:

NC_BACK_TR_RATIO_COEF = 0

NC_FRWD_TR_RATIO_COEF = 1

NC_BACK_SHIFT_COEF = 2

The parameter ulTrCoef can have values of any +ve double values.

[NC_MAX_NUM_COEF] can have array values of [1….n].

NC_TR_FUNC_ID_ENUM iMcsToAcsFuncID

Definition of the array of [NC_TR_FUNC_ID_ENUM] enumerators defined by the linear transformation iMcsToAcsFuncID to change between the MCS and ACS modes.

i*McsToAcsFuncID* can have one of the NC_TR_FUNC_ID_ENUM enumerator values 0 – 1. The size of the i*McsToAcsFuncID* is 16, the maximum number of members in the group.

Presently, Elmo supports only two functions defined by the following enumerator NC_TR_FUNC_ID_ENUM:

NC_TR_NONE_FUNC = 0
NC_TR_SHIFT_FUNC = 1
NC_TR_LAST_ FUNC = 2

NC_NODE_HNDL_T hNode

Node index integer. Any positive numeric value.

NC_AXIS_IN_GROUP_TYPE_ENUM eType

NC_AXIS_IN_GROUP_TYPE_ENUM define the types of supported axis in the group. The are 16 possible values:

NC_TR_NONE_FUNC = 0,
NC_TR_SHIFT_FUNC,

NC_TR_LAST_FUNC

NC_NODE_1_ID = 0,

NC_NODE_2_ID = 1,

NC_NODE_3_ID = 2,

NC_NODE_4_ID = 3,

NC_NODE_5_ID = 4,

NC_NODE_6_ID = 5,

NC_NODE_7_ID = 6,

NC_NODE_8_ID = 7,

NC_NODE_9_ID = 8,

NC_NODE_10_ID = 9,

NC_NODE_11_ID = 10,

NC_NODE_12_ID = 11,

NC_NODE_13_ID = 12,

NC_NODE_14_ID = 13,

NC_NODE_15_ID = 14,

NC_NODE_16_ID = 15

5.15.3 MC_KIN_REF_CARTESIAN

MC_KIN_REF_CARTESIAN Structure

```
typedef struct {
MC_KIN_NODE_DEF sNode[NC_MAX_NUM_AXES_IN_NODE];
int iNumAxes;
} MC_KIN_REF_CARTESIAN;
```

Description Cartesian kinetic node definition structure. This structure is not a function block.

Source GMAS\includes\MMC_PLCopen_group_API.h

Parameters

MC_KIN_NODE_DEF sNode

Refer to the previous sub-structure node **5.15.2 MC_KIN_NODE_DEF** for details.

iNumAxes

Number of axes. *iNumAxes* can have values of any positive integer [1….16].

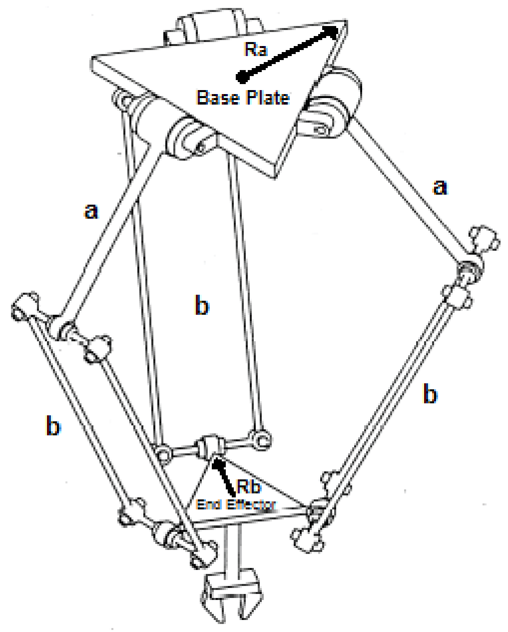
5.15.4 MC_KIN_REF_DELTA

MC_KIN_REF_DELTA Structure

```
typedef struct {
double dbArm;
double dbForeArm;
double dbBaseRadius;
double dbEndEffectorRadius;
MC_KIN_NODE_DEF sNode [NC_MAX_NUM_AXES_IN_NODE];
int iNumAxes;
}MC_KIN_REF_DELTA;
```

Description Delta robot definition structure. This structure is not a function block. Refer to section **5.10.9.1 Delta Robot kinematic** for more details.

Mechanical inputs of the robot; Arm length (a), Forearm length (b), Base Radius (Ra), and End Effector Radius (Rb)



Source GMAS\includes\MMC_PLCopen_group_API.h

Parameters

dbArm

Position of the arm length. Any +ve value.

dbForeArm

Position of the forearm length. Any +ve value.

dbBaseRadius

Base Radius value. Any +ve value.

dbEndEffectorRadius

End Effector radius value. Any +ve value.

MC_KIN_NODE_DEF sNode

Refer to the previous sub-structure node **5.15.2 MC_KIN_NODE_DEF** for details.

iNumAxes

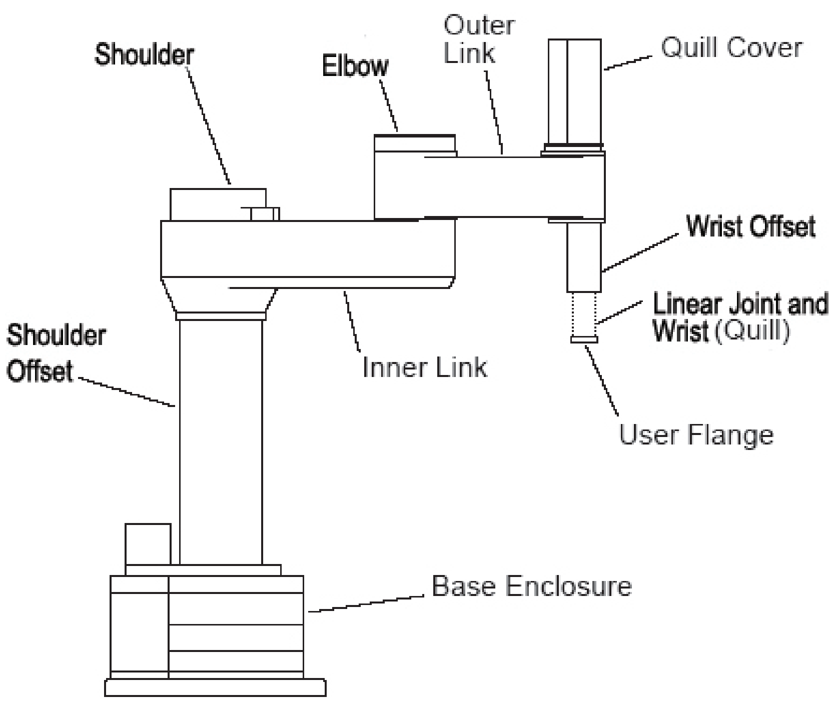
Number of axes. *iNumAxes* can have values of any positive integer [1….16].

5.15.5 MC_KIN_REF_SCARA

MC_KIN_REF_SCARA Structure

```
typedef struct {
double dInnerLinkLength;
double dOuterLinkLength;
double dShoulderOffset;
double dWristOffset;
double dWristTheta2OffsetCoef;
MC_KIN_NODE_DEF sNode[NC_MAX_NUM_AXES_IN_NODE];
int iNumAxes;
char  cElbowSign;
char  cPadding1 ;
char  cPadding2 ;
char  cPadding3 ;
} MC_KIN_REF_SCARA;
```

Description Scara definition structure. This structure is not a function block.



Source GMAS\includes\MMC_PLCopen_group_API.h

Parameters

dInnerLinkLength

Inner link length. Any +ve value.

dOuterLinkLength

Outer link length. Any +ve value.

dShoulderOffset

Shoulder offset (column height) value. Any +ve value

dWristOffset

Fixed value offset of the wrist height. Any +ve value.

dWristTheta2OffsetCoef

Value of the length of the Linear Joint offset relative to the Wrist rotation.

As the Wrist rotates measured in mm/radians, the offset raises and lowers.

Any +ve or –ve value.

MC_KIN_NODE_DEF sNode

Refer to the previous sub-structure node **5.15.2 MC_KIN_NODE_DEF** for details.

Dependant on the array NC_MAX_NUM_AXES_IN_NODE.

The array size [NC_MAX_NUM_AXES_IN_NODE] is equal to 16, as the maximum number of axes in a group.

iNumAxes

Number of axes. *iNumAxes* has a value of 4.

cElbowSign

The direction of the movement of the Elbow. Is either +1 (Right-hand direction) or -1 (Left-hand direction)

cPadding1

Alignment padding of data. Unsigned +ve character values.

cPadding2

Alignment padding of data. Unsigned +ve character values.

cPadding3

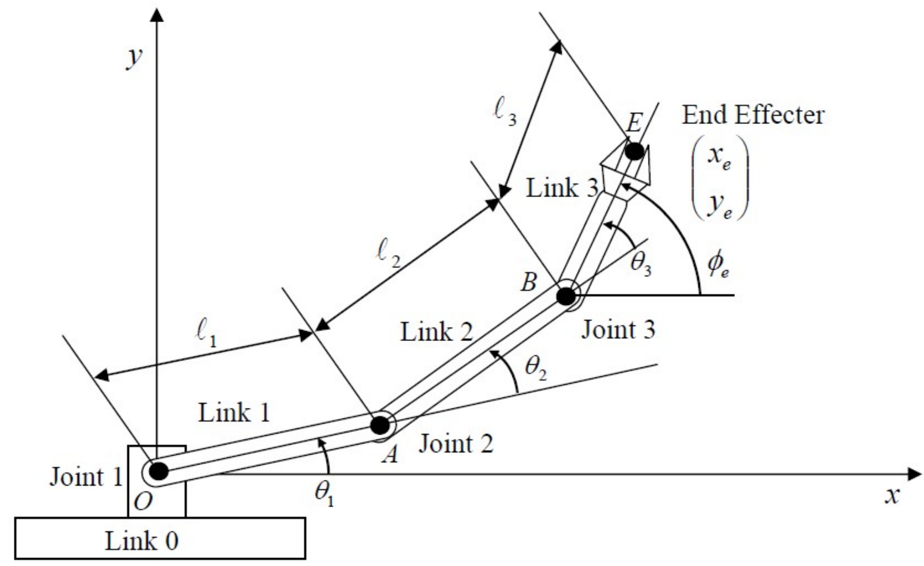
Alignment padding of data. Unsigned +ve character values.

5.15.6 MC_KIN_REF_THREE_LINK

MC_KIN_REF_THREE_LINK Structure

```
typedef struct {
double dInnerLinkLength ;
double dMediumLinkLength;
double dOuterLinkLength;
double dShoulderOffset;
double dWristOffset;
double  dWristTheta2OffsetCoef;
MC_KIN_NODE_DEF sNode[NC_MAX_NUM_AXES_IN_NODE];
int iNumAxes;
char cElbowSign ;
char cPadding1 ;
char cPadding2 ;
char cPadding3 ;
}MC_KIN_REF_THREE_LINK;
```

Description The diagram below defines the Three Link Robot Arm structure. This structure is not a function block. φ_e is the profile vector defined by axis W.



Source GMAS\includes\MMC_PLCopen_group_API.h

Parameters

dInnerLinkLength

Inner link length (Link 1). Any +ve value.

dMediumLinkLength

Central link length (Link 2). Any +ve value.

dOuterLinkLength

Outer link length (Link 3). Any +ve value.

dShoulderOffset

Shoulder offset (Joint 1 column height) value. Any +ve value

dWristOffset

Fixed value offset of the wrist height. Any +ve value.

dWristTheta2OffsetCoef

Value of the length of the Linear Joint offset relative to the Wrist rotation.

As the Wrist rotates measured in mm/radians, the offset raises and lowers.

Any +ve or –ve value.

MC_KIN_NODE_DEF sNode

Refer to the previous sub-structure node **5.15.2 MC_KIN_NODE_DEF** for details.

Dependant on the array NC_MAX_NUM_AXES_IN_NODE.

The array size [NC_MAX_NUM_AXES_IN_NODE] is equal to 16, as the maximum number of axes in a group.

iNumAxes

Number of axes. *iNumAxes* has a value of 4.

cElbowSign

The direction of the movement of the Elbow. Is either +1 (Right-hand direction) or -1 (Left-hand direction)

cPadding1

Alignment padding of data. Unsigned +ve character values.

cPadding2

Alignment padding of data. Unsigned +ve character values.

cPadding3

Alignment padding of data. Unsigned +ve character values.

5.15.7 MC_KIN_REF UNION

5.15.7 MC_KIN_REF UNION Structure

```
typedef union {
MC_KIN_REF_DELTA stDelta;
MC_KIN_REF_CARTESIAN stCart;
MC_KIN_REF_SCARA stScara;
MC_KIN_REF_THREE_LINK stThreeLink;
unsigned char ucMaxSize[1300];
}MC_KIN_REF;
```

Description Union of Kinetic node structures. This structure is not a function block.

Source GMAS\includes\MMC_PLCopen_group_API.h

Parameters

MC_KIN_REF_DELTA stDelta

Refer to the structure **5.15.4 MC_KIN_REF_DELTA** for further details

MC_KIN_REF_CARTESIAN stCart

Refer to the structure **5.15.3 MC_KIN_REF_CARTESIAN** for further details

MC_KIN_REF_SCARA stScara

Refer to the structure **5.15.5 MC_KIN_REF_SCARA** for further details

MC_KIN_REF_THREE_LINK stThreeLink

Refer to the structure **5.15.6 MC_KIN_REF_THREE_LINK** for further details

ucMaxSize[1300]

Text parameter with a maximum of 1300 bit

5.15.8 NC_MCS_Info_Struct

NC_MCS_INFO_STRUCT Structure

```
typedef struct{
double ulTrCoef[NC_MAX_NUM_COEF];
unsigned int hNode;
NC_AXIS_IN_GROUP_TYPE_ENUM eType;
NC_TR_FUNC_ID_ENUM eMcsToAcsFuncID;
}NC_MCS_INFO_STRUCT;
```

Description Info structural parameters. This structure is not a function block.

This structure is deprecated, and therefore will be removed within the near future.

Source GMAS\includes\MMC_PLCopen_group_API.h

Parameters

ulTrCoef [NC_MAX_NUM_COEF]

Defined by the type of supported MCS transform coefficients according to the definition of the enumerator NC_TR_COEF_TYPE_ENUM:

NC_BACK_TR_RATIO_COEF = 0

NC_FRWD_TR_RATIO_COEF = 1

NC_BACK_SHIFT_COEF = 2

The parameter ulTrCoef can have values of any +ve double values.

[NC_MAX_NUM_COEF] can have array values of [1….n].

hNode

Node index integer. Any positive numeric value.

NC_AXIS_IN_GROUP_TYPE_ENUM eType

NC_AXIS_IN_GROUP_TYPE_ENUM define the types of supported axis in the group. The are 16 possible values:

NC_TR_NONE_FUNC = 0,
NC_TR_SHIFT_FUNC,

NC_TR_LAST_FUNC

NC_NODE_1_ID = 0,

NC_NODE_2_ID = 1,

NC_NODE_3_ID = 2,

NC_NODE_4_ID = 3,

NC_NODE_5_ID = 4,

NC_NODE_6_ID = 5,

NC_NODE_7_ID = 6,

NC_NODE_8_ID = 7,

NC_NODE_9_ID = 8,

NC_NODE_10_ID = 9,

NC_NODE_11_ID = 10,

NC_NODE_12_ID = 11,

NC_NODE_13_ID = 12,

NC_NODE_14_ID = 13,

NC_NODE_15_ID = 14,

NC_NODE_16_ID = 15

eMcsToAcsFuncID

MC_TR_FUNC_ID_ENUM can have IDs 0…15 integer values for the 16 Nodes:

NC_NODE_1_ID = 0

…………………

NC_NODE_16_ID = 15

5.15.9 NC_MCS_Kin_Ref_Struct

NC_MCS_Kin_Ref_Struct Structure

```
typedef struct{
int iNumAxes;
NC_MCS_INFO_STRUCT sMcsInfo[NC_MAX_NUM_AXES_IN_NODE];
}NC_MCS_KIN_REF_STRUCT;
```

Description MCS Info structural parameters. This structure is not a function block.

This structure is deprecated, and therefore will be removed within the near future.

Source GMAS\includes\MMC_PLCopen_group_API.h

Parameters

iNumAxes

Number of axes in group. Any positive integer.

NC_MCS_INFO_STRUCT sMcsInfo[NC_MAX_NUM_AXES_IN_NODE]

Refer to the previous structure **5.15.8 NC_MCS_Info_Struct** for a detailed explanation of the parameter.

The array size [NC_MAX_NUM_AXES_IN_NODE] is equal to 16, as the maximum number of axes in a group. The parameter sMcsInfo is a structure, whereas the array parmeter has values of [2….15].

5.15.10 MMC_SetKinTransform

Sets a kinematic transformation between the ACS and MCS based on the predefined kinematic model for multi-axes. Refer to the section **5.10.1 Coordinate System and kinematic transformation** for a further detailed explanation. Refer to sections **5.15.1 - 5.15.9** for details of the structures used within this function.

This structure is deprecated, and therefore will be removed within the near future.

```
MMC_LIB_API int MMC_SetKinTransform(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SETKINTRANSFORM_IN* pInParam,
OUT MMC_SETKINTRANSFORM_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed – Not supported
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Group reference handle type returned by GetAxisRef command

pInParam

Points to the kinematic transformation **MMC_SETKINTRANSFORM_IN** input data structure using the MMC_SetKinTransform function.

pOutParam

Points to the kinematic transformation **MMC_SETKINTRANSFORM_OUT** output structure receiving information, as a result of calling the MMC_SetKinTransform function.

Remarks

A kinematic transformation is a representation of the machine construction. The input MMC_SETKINTRANSFORM_IN refers to a kinematic model including the parameters.

This function block always behaves on a pre-defined AxesGroup. Since a kinematic transformation always has to fit to an appropriate *AxesGroup*, a call to some group function block in MCS mode will lead to an error unless an appropriate *Kinematic* is defined. For a further detailed explanation, refer to the section **5.10.1 Coordinate System and kinematic transformation on page 368.**

Scope

The SetKinTransform function can only be called in the following states:

- Group_Standby
- Group_Disabled

Calling the function in any other results in the Error **-182**.

MMC_SETKINTRANSFORM_IN Structure

```
typedef struct{
double ulTrCoef[NC_MAX_NUM_AXES_IN_NODE][NC_MAX_NUM_COEF];
int iNumAxes;
MC_BUFFERED_MODE_ENUM eBufferMode;
NC_TR_FUNC_ID_ENUM iMcsToAcsFuncID[NC_MAX_NUM_AXES_IN_NODE];
NC_NODE_HNDL_T hNode[NC_MAX_NUM_AXES_IN_NODE];
NC_AXIS_IN_GROUP_TYPE_ENUM eType[NC_MAX_NUM_AXES_IN_NODE];
unsigned char ucExecute;
}MMC_SETKINTRANSFORM_IN;
```

Parameters

ulTrCoef

The parameter *ulTrCoef* is defined by the type of supported MCS transform function according to the definition of the array of [NC_TR_FUNC_ID_ENUM] enumerators defined by the linear transformation iMcsToAcsFuncID. The value of *ulTrCoef* is 8 bytes (double) and ranges between any ±15 digit value.

These coefficients depend on the transition function type. Presently, Elmo supports only two functions defined by the following enumerator NC_TR_FUNC_ID_ENUM:

NC_TR_NONE_FUNC = 0
NC_TR_SHIFT_FUNC = 1
NC_TR_LAST_ FUNC = 2

Each of the above enumerations has a different connotation dependant on the transition function type. The coefficients are a two dimensional double array 16 x 3, with the following definitions:

16 – the maximum number of axes

3 – the three inputs for each axis

An important point when entering the coefficients:

In order to avoid problematic position adjustments (when using MCS mode) always maintain the following ratio between the coefficients:

NC_BACK_TR_RATIO_COEF x NC_FRWD_TR_RATIO_COEF =1

[NC_MAX_NUM_COEF] is defined as 3.

For further details refer to the section **5.10.4 PCS - Product Coordinate System**.

iNumAxes

Number of axes. *iNumAxes* can have values of any positive integer [1….16].

eBufferMode

MC_BufferMode defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3

	MC_BLENDING_PREVIOUS_MODE = 4
	MC_BLENDING_NEXT_MODE = 5
	MC_BLENDING_HIGH_MODE = 6
Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

iMcsToAcsFuncID

Linear transformation Function ID to change between the MCS and ACS modes.

i*McsToAcsFuncID* can have one of the NC_TR_FUNC_ID_ENUM enumerator values 0 – 1. The size of the i*McsToAcsFuncID* is 16, the maximum number of members in the group.

The array parameter NC_MAX_NUM_AXES_IN_NODE is equal to 16, and defined as the maximum number of axes in a group.

hNode

NC_NODE_HNDL_T defines the Node references array. The array parameter NC_MAX_NUM_AXES_IN_NODE is equal to 16, and defined as the maximum number of axes in a group. Since this is in essence the axis reference, refer to the function in section **8.1.16 MMC_GetAxisByName.**

NC_AXIS_IN_GROUP_TYPE_ENUM eType

NC_AXIS_IN_GROUP_TYPE_ENUM_EX define the types of supported kinematic directions in the vector. There are 16 possible values for the parameter *eType*:

typedef enum{
NC_PROFILER_X_AXIS_TYPE = 0,
NC_PROFILER_Y_AXIS_TYPE,
NC_PROFILER_Z_AXIS_TYPE,
NC_PROFILER_U_AXIS_TYPE,
NC_PROFILER_V_AXIS_TYPE,
NC_PROFILER_W_AXIS_TYPE,
NC_PROFILER_N1_AXIS_TYPE,
NC_PROFILER_N2_AXIS_TYPE,

NC_PROFILER_N3_AXIS_TYPE,
NC_PROFILER_N4_AXIS_TYPE,
NC_PROFILER_N5_AXIS_TYPE,
NC_PROFILER_N6_AXIS_TYPE,
NC_PROFILER_N7_AXIS_TYPE,
NC_PROFILER_N8_AXIS_TYPE,
NC_PROFILER_N9_AXIS_TYPE,
NC_PROFILER_S_AXIS_TYPE,
}NC_AXIS_IN_GROUP_TYPE_ENUM_EX;

ucExecute

For this function, this parameter is not relevant.

MMC_SETKINTRANSFORM_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETKINTRANSFORM_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-126 describes the function block for MMC_SetKinTransform as applied within the IEC 61131 programming.

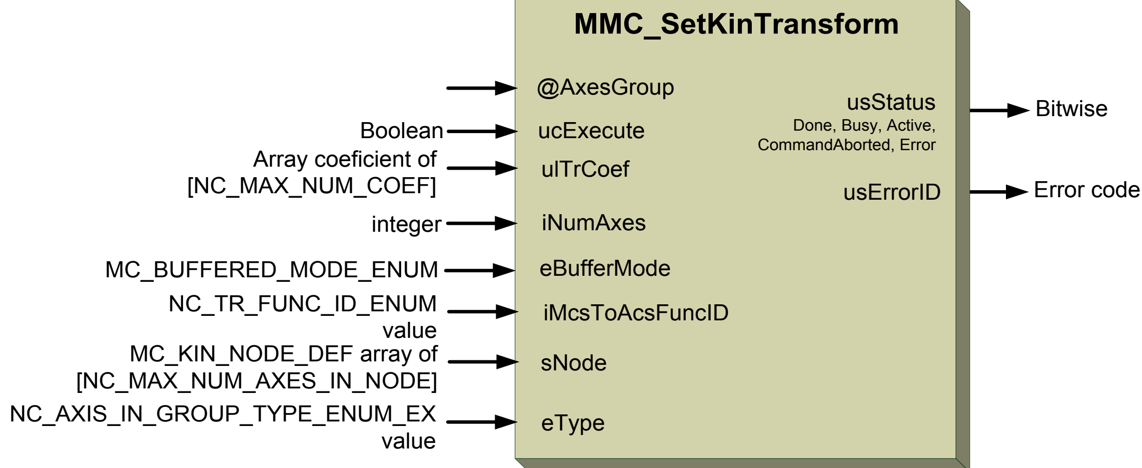


Figure 5-143: MMC_SetKinTransform function block

5.15.10.2 Function Block Code Example

```
int rc;
MMC_SETKINTRANSFORM_IN      stSetKinTransform_in;
MMC_SETKINTRANSFORM_OUT     stSetKinTransform_out;
//
// Inserting the structure parameters:
stSetKinTransform_in.eBufferMode         = MC_BUFFERED_MODE;// MC_BUFFERED_MODE_ENUM Defines
the behavior of the axis
stSetKinTransform_in.ulTrCoef[2][5]      = 1; // Array of coordinates, incl. positions and
orientations
stSetKinTransform_in.ulTrCoef[3][5]      = 1;
stSetKinTransform_in.ulTrCoef[4][5]      = 1;
stSetKinTransform_in.iMcsToAcsFuncID[2]  = 2; //MCS to ACS function ID
stSetKinTransform_in.iMcsToAcsFuncID[3]  = 3;
stSetKinTransform_in.iMcsToAcsFuncID[4]  = 4;
stSetKinTransform_in.iNumAxes            = 6;// Position is added to the actual position value
of the axis
stSetKinTransform_in.ucExecute           = 1;
//
rc = MMC_SetKinTransform (hConn, iAxisRef, &stSetKinTransform_in, &stSetKinTransform_out) ;
if (rc != 0)
{
    HandleError();
}
```

5.15.11 MMC_SetKinTransformEx

Sets a kinematic transformation between the ACS and MCS based on the predefined kinematic model for group multi-axes. Refer to the section **5.10.1 Coordinate System and kinematic transformation** for a further detailed explanation. Refer to sections **5.15.1 - 5.15.9** for details of the structures used within this function.

```
MMC_LIB_API int MMC_SetKinTransformex(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SETKINTRANSFORMEX_IN* pInParam,
OUT MMC_SETKINTRANSFORMEX_OUT* pOutParam
);
```

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command.

hAxisRef

Vector reference handle type returned by GetVectorRef command

pInParam

Points to set the kinematic transformation **MMC_SETKINTRANSFORMEX_IN** input data structure using the MMC_SetKinTransformEx function.

pOutParam

Points to set the kinematic transformation **MMC_SETKINTRANSFORMEX_OUT** output structure receiving information, as a result of calling the MMC_SetKinTransformEx function.

Remarks

A kinematic transformation is a representation of the machine construction. The input MMC_SETKINTRANSFORMEX_IN refers to a kinematic model including the parameters.

This function always behaves on a pre-defined Vector. Since a kinematic transformation always has to fit to an appropriate *Vector*, a call to some vector function block in MCS mode will lead to an error unless an appropriate *Kinematic* is defined. For a further detailed explanation, refer to the section **5.10.1 Coordinate System and kinematic transformation.**

Scope

The SetKinTransformEx function can only be called in the following states:

- Group_Standby
- Group_Disabled

Calling the function in any other results in the Error **-182**.

MMC_SETKINTRANSFORMEX_IN Structure

```
typedef struct{
MC_KIN_REF stInput;
NC_KIN_TYPE eKinType;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
}MMC_SETKINTRANSFORMEX_IN;
```

Parameters

MC_KIN_REF stInput

```
typedef union{
MC_KIN_REF_DELTA stDelta;
MC_KIN_REF_CARTESIAN stCart;
unsigned char ucMaxSize[1300];
}MC_KIN_REF;
```

Where the user can select to use either the parameters:

MC_KIN_REF_DELTA stDelta

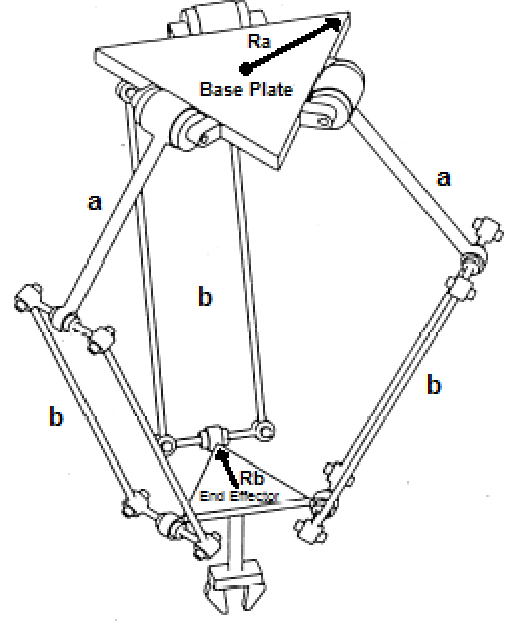
Or

MC_KIN_REF_CARTESIAN stCart

MC_KIN_REF_DELTA stDelta

```
typedef struct{
double dbArm;
double dbForeArm;
double dbBaseRadius;
double dbEndEffectorRadius;
MC_KIN_NODE_DEF sNode[NC_MAX_NUM_AXES_IN_NODE];
int iNumAxes;
}MC_KIN_REF_DELTA;
```

Each of these preset parameters have the following definitions based on the drawing below:



dbArm

Arm length (a). Any +ve or –ve values

dbForeArm

ForeArm length (b). Any +ve or –ve values

dbBaseRadius

Base Radius (Ra). Any +ve or –ve values

dbEndEffectorRadius

End Effector Radius (Rb). Any +ve or –ve values

MC_KIN_NODE_DEF sNode[NC_MAX_NUM_AXES_IN_NODE]

The array parameter NC_MAX_NUM_AXES_IN_NODE is equal to 16, and defined as the maximum number of axes in a vector.

MC_KIN_NODE_DEF

```
typedef struct{
double ulTrCoef[NC_MAX_NUM_COEF];
NC_TR_FUNC_ID_ENUM iMcsToAcsFuncID;
NC_NODE_HNDL_T hNode;
NC_AXIS_IN_GROUP_TYPE_ENUM_EX eType;
}MC_KIN_NODE_DEF;
```

ulTrCoef

The parameter *ulTrCoef* is defined by the type of supported define transformation function at the iMcsToAcsFuncID variable. Thevalue of *ulTrCoef* is a double type array with three

items.

An important point when entering the coefficients:

In order to avoid problematic position adjustments (when using linear transformation) always maintain the following ratio between the coefficients:

NC_BACK_TR_RATIO_COEF x
NC_FRWD_TR_RATIO_COEF =1

[NC_MAX_NUM_COEF] is defined as 3.

For further details refer to the section **5.10.1 Coordinate System and kinematic transformation.**

iMcsToAcsFuncID

i*McsToAcsFuncID* can have one of the NC_TR_FUNC_ID_ENUM enumerator values:

NC_TR_NONE_FUNC = 0
NC_TR_SHIFT_FUNC = 1
NC_TR_LAST_ FUNC = 2

hNode

NC_NODE_HNDL_T defines the Node reference of current axis (member of the vector). Since this is in essence the axis reference, refer to the function in section **8.1.16 MMC_GetAxisByName.**

NC_AXIS_IN_GROUP_TYPE_ENUM_EX eType

NC_AXIS_IN_GROUP_TYPE_ENUM_EX define the types of supported kinematic directions in the vector. There are 28 possible values for the parameter *eType*:

typedef enum{
NC_PROFILER_X_AXIS_TYPE = 0,
NC_PROFILER_Y_AXIS_TYPE,
NC_PROFILER_Z_AXIS_TYPE,
NC_PROFILER_U_AXIS_TYPE,
NC_PROFILER_V_AXIS_TYPE,
NC_PROFILER_W_AXIS_TYPE,
NC_PROFILER_N1_AXIS_TYPE,
NC_PROFILER_N2_AXIS_TYPE,
NC_PROFILER_N3_AXIS_TYPE,
NC_PROFILER_N4_AXIS_TYPE,
NC_PROFILER_N5_AXIS_TYPE,
NC_PROFILER_N6_AXIS_TYPE,
NC_PROFILER_N7_AXIS_TYPE,
NC_PROFILER_N8_AXIS_TYPE,

NC_PROFILER_N9_AXIS_TYPE,
NC_PROFILER_S_AXIS_TYPE,
}NC_AXIS_IN_GROUP_TYPE_ENUM_EX;

iNumAxes

Number of axes in group. *iNumAxes* can have values of any positive integer [1….16]. Integer value.

MC_KIN_REF_CARTESIAN stCart

typedef struct{
MC_KIN_NODE_DEF sNode[NC_MAX_NUM_AXES_IN_NODE];
int iNumAxes;
}MC_KIN_REF_CARTESIAN;

Each of these preset parameters have the following definitions:

MC_KIN_NODE_DEF sNode[NC_MAX_NUM_AXES_IN_NODE]

The array parameter NC_MAX_NUM_AXES_IN_NODE is equal to 16, and defined as the maximum number of axes in a vector.

MC_KIN_NODE_DEF

typedef struct{
double ulTrCoef[NC_MAX_NUM_COEF];
NC_TR_FUNC_ID_ENUM iMcsToAcsFuncID;
NC_NODE_HNDL_T hNode;
NC_AXIS_IN_GROUP_TYPE_ENUM_EX eType;
}MC_KIN_NODE_DEF;

Each of these preset parameters have the following definitions:

ulTrCoef[NC_MAX_NUM_COEF]

The parameter *ulTrCoef* is defined by the type of supported define transformation function at iMcsToAcsFuncID variable. Thevalue of *ulTrCoef* is a double type array with three items.

An important point when entering the coefficients:

n order to avoid problematic position adjustments (when using linear transformation) always maintain the following ratio between the coefficients:

NC_BACK_TR_RATIO_COEF x NC_FRWD_TR_RATIO_COEF =1

[NC_MAX_NUM_COEF] is defined as 3.

For further details refer to the section **5.10.1 Coordinate System and kinematic transformation.**

NC_TR_FUNC_ID_ENUM iMcsToAcsFuncID

i*McsToAcsFuncID* can have one of the NC_TR_FUNC_ID_ENUM enumerator values:

NC_TR_NONE_FUNC = 0

NC_TR_SHIFT_FUNC = 1
NC_TR_LAST_ FUNC = 2

NC_NODE_HNDL_T hNode

NC_NODE_HNDL_T defines the Node reference of current axis (member of the vector). Since this is in essence the axis reference, refer to the function in section **8.1.16 MMC_GetAxisByName.**

NC_AXIS_IN_GROUP_TYPE_ENUM_EX eType

NC_AXIS_IN_GROUP_TYPE_ENUM_EX define the types of supported axis in the group. There are 28 possible values for the parameter *eType*:

typedef enum{
NC_PROFILER_X_AXIS_TYPE = 0,
NC_PROFILER_Y_AXIS_TYPE,
NC_PROFILER_Z_AXIS_TYPE,
NC_PROFILER_U_AXIS_TYPE,
NC_PROFILER_V_AXIS_TYPE,
NC_PROFILER_W_AXIS_TYPE,
NC_PROFILER_N1_AXIS_TYPE,
NC_PROFILER_N2_AXIS_TYPE,
NC_PROFILER_N3_AXIS_TYPE,
NC_PROFILER_N4_AXIS_TYPE,
NC_PROFILER_N5_AXIS_TYPE,
NC_PROFILER_N6_AXIS_TYPE,
NC_PROFILER_N7_AXIS_TYPE,
NC_PROFILER_N8_AXIS_TYPE,
NC_PROFILER_N9_AXIS_TYPE,
NC_PROFILER_S_AXIS_TYPE,
}NC_AXIS_IN_GROUP_TYPE_ENUM_EX;

iNumAxes

Number of axes in group. *iNumAxes* can have values of any positive integer [1….16]. Integer value.

ucMaxSize[1300]

The maximum size of the parameters together is defined to 1300 digits

NC_KIN_TYPE eKinType

This parameter define the type of the kinematic:

NC_CARTESIAN_TYPE when the user work with Cartesian robot.

NC_DELTA_ROBOT_TYPE when the user work with Delta robot.

MC_BUFFERED_MODE_ENUM eBufferMode

Not in use at this moment.

ucExecute

For this function, this parameter is not relevant.

MMC_SETKINTRANSFORMEX_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETKINTRANSFORMEX_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-127 describes the function block for MMC_SetKinTransformEx.

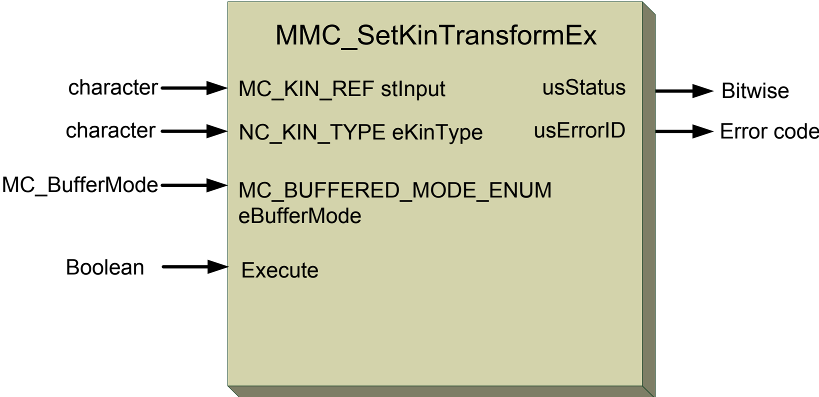


Figure 5-144: MMC_SetKinTransformEx function block

Figure 5-128 describes the function block for MMC_SetDeltaRobotKinematics as applied within the IEC 61131 programming.

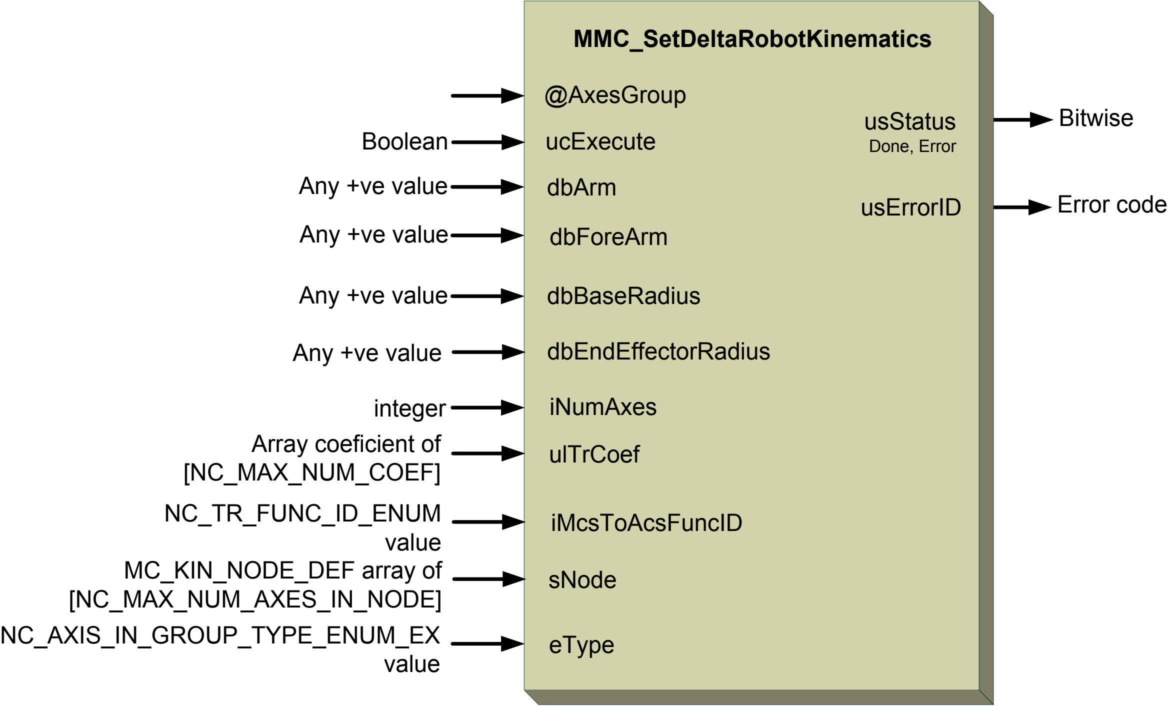


Figure 5-145: MMC_SetDeltaRobotKinematics function block

5.15.12 MMC_SetCartesianTransform

Sets a group's cartesian transformation between MCS and PCS parameters based on the predefined kinematic model for group multi-axes. Refer to the section **5.10.4 PCS - Product Coordinate System** for a further detailed explanation. Refer to sections **5.15.1 - 5.15.9** for details of the structures used within this function.

```
MMC_LIB_API int MMC_SetCartesianTransform(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SETCARTESIANTRANSFORM_IN* pInParam,
OUT MMC_SETCARTESIANTRANSFORM_OUT* pOutParam
);
```

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command.

hAxisRef

Vector reference handle type returned by GetVectorRef command

pInParam

Points to set the cartesian transformation **MMC_SETCARTESIANTRANSFORM_IN** input data structure using the MMC_SetCartesianTransform function.

pOutParam

Points to set the cartesian transformation **MMC_SETCARTESIANTRANSFORM_OUT** output structure receiving information, as a result of calling the MMC_SetCartesianTransform function.

Remarks

For a further detailed explanation, refer to the section **5.10.4 PCS - Product Coordinate System.**

Scope

MMC_SETCARTESIANTRANSFORM_IN Structure

```
typedef struct {
double dOffset[3];
double dRotAngle[3];
double dPadding[5] ;
PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits;
MC_BUFFERED_MODE_ENUM eBufferMode;
MC_EXECUTION_MODE eExecutionMode;
unsigned char ucExecute;
} MMC_SETCARTESIANTRANSFORM_IN;
```

Parameters

dOffset[3]

X,Y,Z translation components' offsets. Any +ve or –ve values

dRotAngle[3]

U,V,W rotation angles. Any +ve or –ve values

dPadding[5]

Alignment padding of data, up to 5 bits. Any +ve or –ve values

PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits

The rotational units used to defined the angle units used in PCS to MCS transformations

The enumerator PCS_ROTATION_ANGLE_UNITS_ENUM is defined by the following:

```
PCS_DEGREE   = 0
PCS_RADIAN   = 1
```

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest

velocity of both commands (1 and 2) at the first end-position (1).

BlendingPrevious Blending with the velocity of function block 1 at the end-position of this block

BlendingNext Blending with the velocity of function block 2 at end-position of function block1

BlendingHigh Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

MC_EXECUTION_MODE eExecutionMode

Execution mode enumerator defining whether the execution is immediate or queued, with the following values:

eMMC_EXECUTION_MODE_IMMEDIATE = 0,

eMMC_EXECUTION_MODE_QUEUED

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_SETCARTESIANTRANSFORM_OUT Structure

```
typedef struct {
unsigned short usStatus;
short sErrorID;
} MMC_SETCARTESIANTRANSFORM_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-129 describes the function block for MMC_SetCartesianTransform.

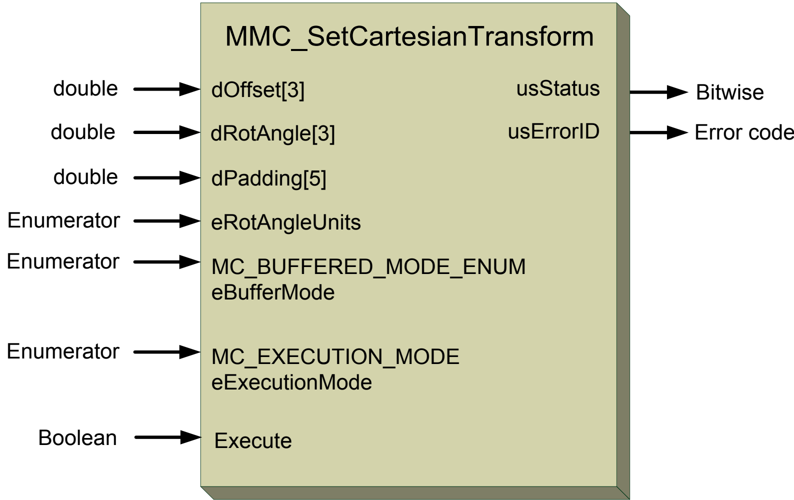


Figure 5-146: MMC_SetCartesianTransform function block

5.15.13 MMC_TrackConveyorBelt

Sets a group's conveyor belt track transformation parameters based on the predefined kinematic model for group multi-axes. Refer to the section **5.10.4 PCS - Product Coordinate System** for a further detailed explanation. Refer to sections **5.15.1 - 5.15.9** for details of the structures used within this function.

```
MMC_LIB_API int MMC_TrackConveyorBelt(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_TRACKCONVEYORBELT_IN* pInParam,
OUT MMC_TRACKCONVEYORBELT_OUT* pOutParam
);
```

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command.

hAxisRef

Vector reference handle type returned by GetVectorRef command

pInParam

Points to set the kinematic transformation **MMC_TRACKCONVEYORBELT_IN** input data structure using the MMC_TrackConveyorBelt function.

pOutParam

Points to set the kinematic transformation **MMC_TRACKCONVEYORBELT_OUT** output structure receiving information, as a result of calling the MMC_TrackConveyorBelt function.

Remarks

For a further detailed explanation, refer to the section **5.10.4 PCS - Product Coordinate System.**

Scope

MMC_TRACKCONVEYORBELT_IN Structure

```
typedef struct mmc_trackconveyorbelt_in {
double dConveyorBeltOrigin[6];
double dInitialObjectPosition[6];
double dInitialRefAxisPosition;
double dRefAxisScaling;
double dPadding[5];
unsigned short usRefAxis;
PCS_REF_AXIS_SRC_ENUM eRefAxisSourceType;
PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits;
MC_BUFFERED_MODE_ENUM eBufferMode;
MC_COORD_SYSTEM_ENUM eCoordSystem;
MC_EXECUTION_MODE eExecutionMode;
unsigned char ucExecute;
} MMC_TRACKCONVEYORBELT_IN;
```

Parameters

dConveyorBeltOrigin[6]

X,Y,Z,U,V,W conveyor belt origin. Any +ve –ve values

dInitialObjectPosition[6]

X,Y,Z,U,V,W initial object's position on the conveyor belt. Any +ve or –ve values

dInitialRefAxisPosition

Initial reference axis position of the conveyor belt. Any +ve or –ve values

dRefAxisScaling

The scaling of the referenece axis relative to the position of the object on the conveyor belt and the length of the conveyor belt. Any +ve or –ve values

dPadding[5]

Alignment padding of data, up to 5 bits. Any +ve or –ve values

usRefAxis

The selected axis reference. Any +ve value.

PCS_REF_AXIS_SRC_ENUM eRefAxisSourceType

Defines the reference axis source as either:

DTargetPosUU *or*

dActualPosUU

PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits

The rotational units used to defined the angle units used in PCS to MCS transformations

The enumerator PCS_ROTATION_ANGLE_UNITS_ENUM is defined by the following:

PCS_DEGREE = 0

PCS_RADIAN = 1

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator. The options are:

MC_NONE_COORD	= 0
MC_ACS_COORD	= 1
MC_MCS_COORD	= 2
MC_PCS_COORD	= 3

MC_EXECUTION_MODE eExecutionMode

Execution mode enumerator defining whether the execution is immediate or queued, with the following values:

eMMC_EXECUTION_MODE_IMMEDIATE = 0,

eMMC_EXECUTION_MODE_QUEUED

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_TRACKCONVEYORBELT_OUT Structure

typedef struct {
unsigned short usStatus;
short sErrorID;
} MMC_SETCARTESIANTRANSFORM_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-130 describes the function block for MMC_TrackConveyorBelt.

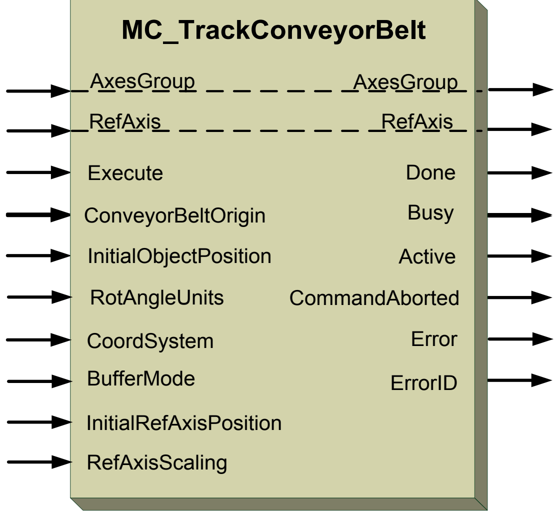


Figure 5-147: MMC_TrackConveyorBelt function block

5.15.14 MMC_TrackRotaryTable

Sets a group's rotary table track transformation parameters based on the predefined kinematic model for group multi-axes. Refer to the section **5.10.4 PCS - Product Coordinate System** for a further detailed explanation. Refer to sections **5.15.1 - 5.15.9** for details of the structures used within this function.

```
MMC_LIB_API int MMC_TrackRotaryTable(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_TRACKROTARYTABLE_IN* pInParam,
OUT MMC_TRACKROTARYTABLE_OUT* pOutParam
);
```

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command.

hAxisRef

Vector reference handle type returned by GetVectorRef command

pInParam

Points to set the kinematic transformation MMC_TRACKROTARYTABLE_IN input data structure using the MMC_TrackRotaryTable function.

pOutParam

Points to set the kinematic transformation MMC_TRACKROTARYTABLE_OUT output structure receiving information, as a result of calling the MMC_TrackRotaryTable function.

Remarks

For a further detailed explanation, refer to the section **5.10.4 PCS - Product Coordinate System.**

Scope

MMC_TRACKROTARYTABLE_IN Structure

```
typedef struct mmc_trackrotarytable_in {
double dRotaryTableOrigin[6];
double dInitialObjectPosition[6];
double dInitialRefAxisPosition;
double dRefAxisScaling;
double dPadding[5] ;
unsigned short usRefAxis;
PCS_REF_AXIS_SRC_ENUM eRefAxisSourceType;
PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits;
MC_BUFFERED_MODE_ENUM eBufferMode;
MC_COORD_SYSTEM_ENUM eCoordSystem;
MC_EXECUTION_MODE eExecutionMode;
unsigned char ucExecute;
} MMC_TRACKROTARYTABLE_IN;
```

Parameters

dRotaryTableOrigin[6]

X,Y,Z,U,V,W rotary table origin. Any +ve –ve values

dInitialObjectPosition[6]

X,Y,Z,U,V,W initial object's position on the conveyor belt. Any +ve or –ve values

dInitialRefAxisPosition

Initial reference axis position of the rotary table. Any +ve or –ve values

dRefAxisScaling

The scaling of the referenece axis relative to the position of the object on the rotary table and the diameter of the table. Any +ve or –ve values

dPadding[5]

Alignment padding of data, up to 5 bits. Any +ve or –ve values

usRefAxis

The selected axis reference. Any +ve value.

PCS_REF_AXIS_SRC_ENUM eRefAxisSourceType

Defines the reference axis source as either:

DTargetPosUU *or*

dActualPosUU

PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits

The rotational units used to defined the angle units used in PCS to MCS transformations

The enumerator PCS_ROTATION_ANGLE_UNITS_ENUM is defined by the following:

```
PCS_DEGREE  = 0
PCS_RADIAN  = 1
```

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator. The options are:

MC_NONE_COORD	= 0
MC_ACS_COORD	= 1
MC_MCS_COORD	= 2
MC_PCS_COORD	= 3

MC_EXECUTION_MODE eExecutionMode

Execution mode enumerator defining whether the execution is immediate or queued, with the following values:

eMMC_EXECUTION_MODE_IMMEDIATE = 0,

eMMC_EXECUTION_MODE_QUEUED

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_TRACKROTARYTABLE_OUT Structure

```
typedef struct mmc_trackrotarytable_out {
unsigned short usStatus;
short sErrorID;
} MMC_TRACKROTARYTABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-131 describes the function block for MMC_TrackRotaryTable.

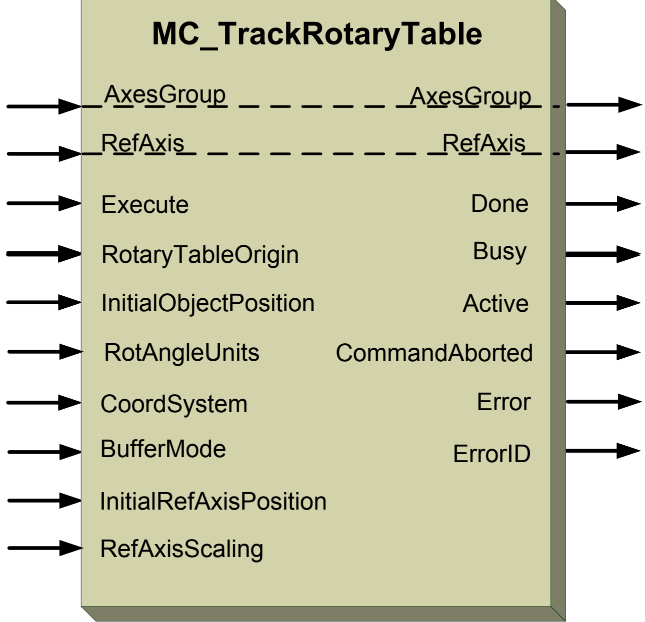


Figure 5-148: MMC_TrackRotaryTable function block

5.15.15 MMC_SetKinTransformDelta

Sets a kinematic transformation between the ACS and MCS based on the predefined kinematic model for group multi-axes with the Delta robot. Refer to the section **5.10.4 PCS - Product Coordinate System** for a further detailed explanation. Refer to sections **5.15.1 - 5.15.9** for details of the structures used within this function.

```
MMC_LIB_API int MMC_SetKinTransformDelta(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_KINTRANSFORM_DELTA_IN* pInParam,
OUT MMC_KINTRANSFORM_DELTA_OUT* pOutParam
);
```

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command.

hAxisRef

Vector reference handle type returned by GetVectorRef command

pInParam

Points to set the kinematic transformation **MMC_KINTRANSFORM_DELTA_IN** input data structure using the MMC_SetKinTransformDelta function.

pOutParam

Points to set the kinematic transformation **MMC_KINTRANSFORM_DELTA_OUT** output structure receiving information, as a result of calling the MMC_SetKinTransformDelta function.

Remarks

For a further detailed explanation, refer to the section **5.10.4 PCS - Product Coordinate System.**

Scope

MMC_KINTRANSFORM_DELTA_IN Structure

```
typedef struct _mc_kintransform_delta_in {
MC_KIN_REF_DELTA stParams;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
} MMC_KINTRANSFORM_DELTA_IN;
```

Parameters

MC_KIN_REF_DELTA stParams

Refer to the MC_KIN_REF_DELTA structure in section **5.15.4** for details, and implementation.

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_KINTRANSFORM_DELTA_OUT Structure

typedef MMC_SETKINTRANSFORMEX_OUT MMC_KINTRANSFORM_DELTA_OUT;

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETKINTRANSFORMEX_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-132 describes the function block for MMC_SetKinTransformDelta.

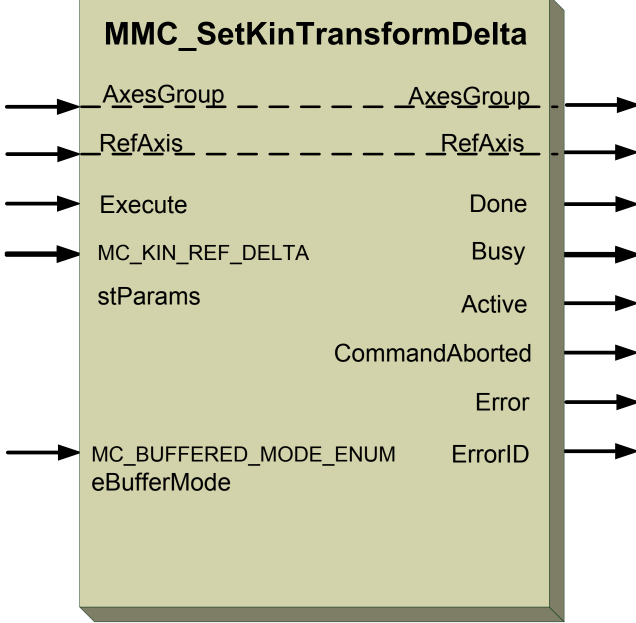


Figure 5-149: MMC_SetKinTransformDelta function block

5.15.16 MMC_SetKinTransformCartesian

Sets the parameters kinematic transformation (MSC to ACS) for Cartesian system based on the predefined kinematic model for group multi-axes. Refer to the section **5.10.4 PCS - Product Coordinate System** for a further detailed explanation. Refer to sections **5.15.1 - 5.15.9** for details of the structures used within this function.

```
MMC_LIB_API int MMC_SetKinTransformCartesian(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_KINTRANSFORM_CARTESIAN_IN* pInParam,
OUT MMC_KINTRANSFORM_CARTESIAN_OUT* pOutParam
);
```

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command.

hAxisRef

Vector reference handle type returned by GetVectorRef command

pInParam

Points to set the kinematic transformation **MMC_KINTRANSFORM_CARTESIAN_IN** input data structure using the MMC_SetKinTransformCartesian function.

pOutParam

Points to set the kinematic transformation **MMC_KINTRANSFORM_CARTESIAN_OUT** output structure receiving information, as a result of calling the MMC_SetKinTransformCartesian function.

Remarks

For a further detailed explanation, refer to the section **5.10.4 PCS - Product Coordinate System.**

Scope

MMC_KINTRANSFORM_CARTESIAN_IN Structure

```
typedef struct _mc_kintransform_cartesian_in {
MC_KIN_REF_CARTESIAN stParams;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
} MMC_KINTRANSFORM_CARTESIAN_IN;
```

Parameters

MC_KIN_REF_CARTESIAN stParams

Refer to the MC_KIN_REF_CARTESIAN structure in section **5.15.4** for details, and implementation.

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_KINTRANSFORM_CARTESIAN_OUT Structure

typedef MMC_SETKINTRANSFORMEX_OUT MMC_KINTRANSFORM_CARTESIAN_OUT;

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETKINTRANSFORMEX_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-133 describes the function block for MMC_SetKinTransformCartesian.

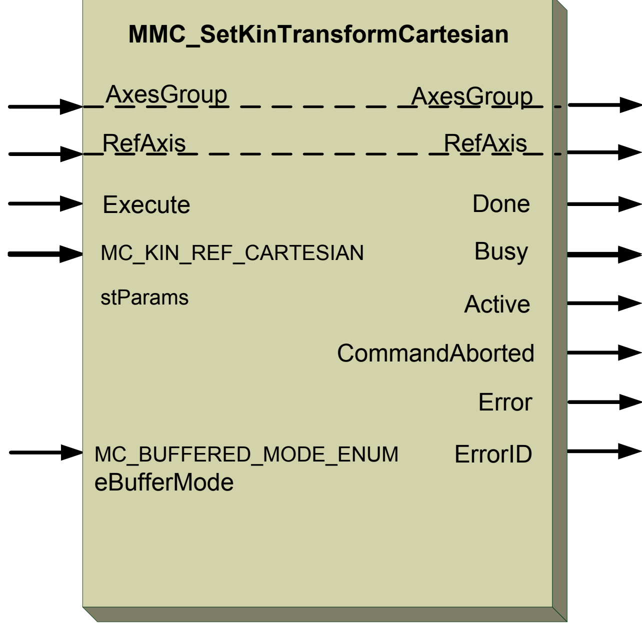


Figure 5-150: MMC_SetKinTransformCartesian function block

5.15.17 MMC_SetKinTransformScara

Sets the parameters kinematic transformation (MSC to ACS) for SCARA robot based on the predefined kinematic model for group multi-axes. Refer to the section **5.10.4 PCS - Product Coordinate System** for a further detailed explanation. Refer to sections **5.15.1 - 5.15.9** for details of the structures used within this function.

```
MMC_LIB_API int MMC_SetKinTransformScara(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_KINTRANSFORM_SCARA_IN* pInParam,
OUT MMC_KINTRANSFORM_SCARA_OUT* pOutParam
);
```

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command.

hAxisRef

Vector reference handle type returned by GetVectorRef command

pInParam

Points to set the kinematic transformation **MMC_KINTRANSFORM_SCARA_IN** input data structure using the MMC_SetKinTransformScara function.

pOutParam

Points to set the kinematic transformation **MMC_KINTRANSFORM_SCARA_OUT** output structure receiving information, as a result of calling the MMC_SetKinTransformScara function.

Remarks

For a further detailed explanation, refer to the section **5.10.4 PCS - Product Coordinate System.**

Scope

MMC_KINTRANSFORM_SCARA_IN Structure

```
typedef struct _mc_kintransform_scara_in {
MC_KIN_REF_SCARA stParams;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
} MMC_KINTRANSFORM_SCARA_IN;
```

Parameters

MC_KIN_REF_SCARA stParams

Refer to the MC_KIN_REF_SCARA structure in section **5.15.4** for details, and implementation.

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_KINTRANSFORM_SCARA_OUT Structure

typedef MMC_SETKINTRANSFORMEX_OUT MMC_KINTRANSFORM_SCARA_OUT;

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETKINTRANSFORMEX_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-134 describes the function block for MMC_SetKinTransformScara.

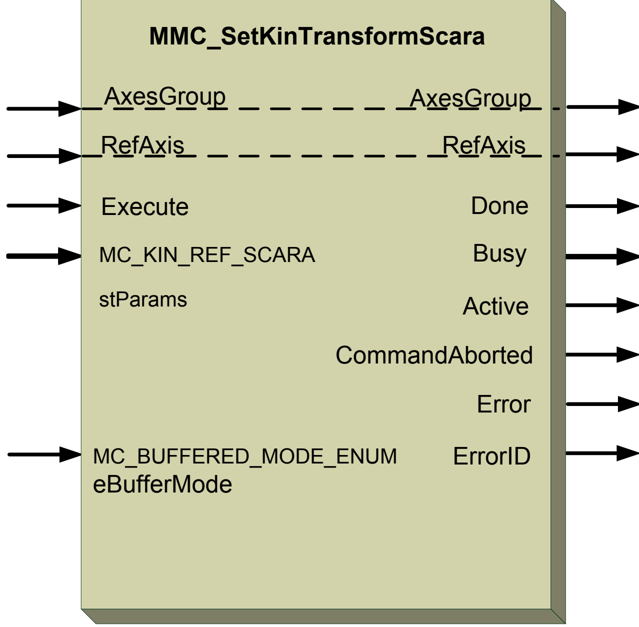


Figure 5-151: MMC_SetKinTransformScara function block

5.15.18 MMC_SetKinTransformThreeLink

Sets the parameters kinematic transformation (MSC to ACS) for THREELINK robot based on the predefined kinematic model for group multi-axes. Refer to the section **5.10.4 PCS - Product Coordinate System** for a further detailed explanation. Refer to sections **5.15.1 - 5.15.9** for details of the structures used within this function.

```
MMC_LIB_API int MMC_SetKinTransformThreeLink(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_KINTRANSFORM_THREELINK_IN* pInParam,
OUT MMC_KINTRANSFORM_THREELINK_OUT* pOutParam
);
```

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command.

hAxisRef

Vector reference handle type returned by GetVectorRef command

pInParam

Points to set the kinematic transformation **MMC_KINTRANSFORM_THREELINK_IN** input data structure using the MMC_SetKinTransformThreeLink function.

pOutParam

Points to set the kinematic transformation **MMC_KINTRANSFORM_THREELINK_OUT** output structure receiving information, as a result of calling the MMC_SetKinTransformThreeLink function.

Remarks

For a further detailed explanation, refer to the section **5.10.4 PCS - Product Coordinate System.**

Scope

MMC_KINTRANSFORM_THREELINK_IN Structure

```
typedef struct _mc_kintransform_threelink_in {
MC_KIN_REF_THREE_LINK stParams;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
} MMC_KINTRANSFORM_THREELINK_IN;
```

Parameters

MC_KIN_REF_THREE_LINK stParams

Refer to the MC_KIN_REF_THREE_LINK structure in section **5.15.6** for details, and implementation.

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_KINTRANSFORM_THREELINK_OUT Structure

typedef MMC_SETKINTRANSFORMEX_OUT MMC_KINTRANSFORM_THREELINK_OUT;

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETKINTRANSFORMEX_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-135 describes the function block for MMC_SetKinTransformThreeLink.

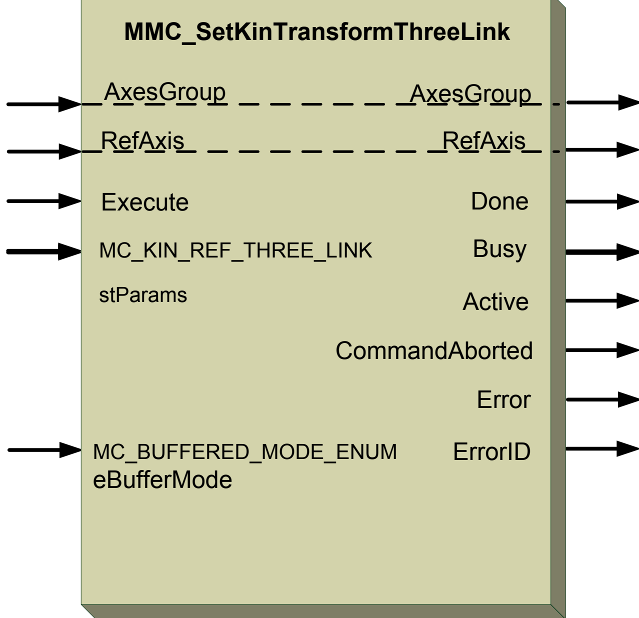


Figure 5-152: MMC_SetKinTransformThreeLink function block

5.15.19 MMC_AddAxisToGroup

For multiple Axes. Adds one axis to a group in a structure *AxesGroup*.

```
MMC_LIB_API int MMC_AddAxisToGroup(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_ADDAXISTOGROUP_IN* pInParam,
OUT MMC_ADDAXISTOGROUP_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

GMAS Programming(IEC 61331 Program)\ElmoGroupAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_ADDAXISTOGROUP_IN** input data structure using the MMC_AddAxisToGroup function.

pOutParam

Points to the **MMC_ADDAXISTOGROUP_OUT** output structure receiving information, as a result of calling the MMC_AddAxisToGroup function.

Remarks

This is an administrative function block, since no movement is generated. The command cannot be buffered.

Each *IdentInGroup* can be used only once, otherwise it leads to an error.

Scope

An axis can be added to a group in GroupStandby or GroupDisabled, but not in GroupHoming, GroupStopping, or GroupErrorStop.

If an axis group is previously defined with nodes, in the resource file, do not call this function. Calling this function under this circumstance will result in the error -69.

MMC_ADDAXISTOGROUP_IN Structure

```
typedef struct{
NC_NODE_HNDL_T hNode;
NC_IDENT_IN_GROUP_ENUM eIdentInGroup;
}MMC_ADDAXISTOGROUP_IN;
```

Parameters

hNode

The NC_NODE_HNDL_T enumerator defines the Node handle transition. The axis ref parameter

hNode can have any positive numeric value.

eIdentInGroup

The NC_IDENT_IN_GROUP_ENUM enumerator identifies the order and Nodes in the group of the added axis. Performed via an enumerator to give the different axes a name in the order, which can be coupled to the names in the kinematic model. The options are:

NC_NODE_1_ID = 0

NC_NODE_2_ID = 1,

NC_NODE_3_ID = 2

NC_NODE_4_ID = 3

NC_NODE_5_ID = 4

NC_NODE_6_ID = 5

NC_NODE_7_ID = 6

NC_NODE_8_ID = 7

NC_NODE_9_ID = 8

NC_NODE_10_ID = 9

NC_NODE_11_ID = 10

NC_NODE_12_ID = 11

NC_NODE_13_ID = 12

NC_NODE_14_ID = 13

NC_NODE_15_ID = 14

NC_NODE_16_ID = 15

MMC_ADDAXISTOGROUP_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_ADDAXISTOGROUP_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-136 describes the function block for MMC_AddAxisToGroup as applied within the IEC 61131 programming.

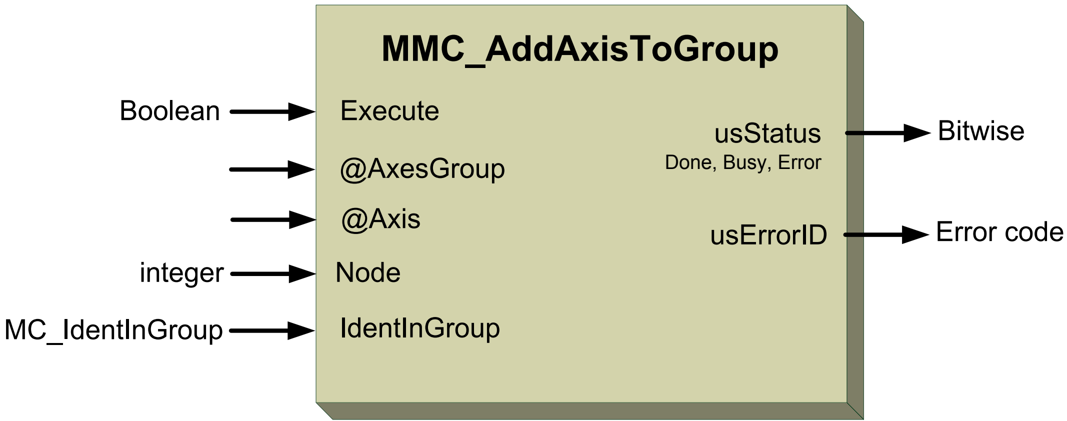


Figure 5-153: MMC_AddAxisToGroup function block

5.15.19.2 Function Block Code Example

```
int rc;
MMC_ADDAXISTOGROUP_IN       stAddAxisToGroup_in;
MMC_ADDAXISTOGROUP_OUT      stAddAxisToGroup_out;
//
// Inserting the structure parameters:
stAddAxisToGroup_in.hNode          = 2;            //Defines the Node handle transition
stAddAxisToGroup_in.eIdentInGroup  = NC_NODE_4_ID;  //Identifies the order and Nodes in the
group of the added axis
//
rc = MMC_AddAxisToGroup (hConn, iAxisRef, &stAddAxisToGroup_in, &stAddAxisToGroup_out);
if (rc != 0)
{
    HandleError();
}
```

5.15.20 MMC_GroupDisable

For multiple Axes. Changes the state for a group to GroupDisabled, although it is an administrative function block, since no movement is generated.

```
MMC_LIB_API int MMC_GroupDisableCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPDISABLE_IN* pInParam,
OUT MMC_GROUPDISABLE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

GMAS Programming(IEC 61331 Program)\ElmoGroupAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPDISABLE_IN** input data structure using the MMC_GroupDisable function.

pOutParam

Points to the **MMC_GROUPDISABLE_OUT** output structure receiving information, as a result of calling the MMC_GroupDisable function.

Remarks

If the axes are not static while issuing this command, the application should take the necessary precautions.

Scope

All

MMC_GROUPDISABLE_IN Structure

```
typedef struct{
unsigned char ucExecute;
}MMC_GROUPDISABLE_IN;
```

Parameters

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_GROUPDISABLE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_GROUPDISABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-137 describes the function block for MMC_GroupDisable as applied within the IEC 61131 programming.

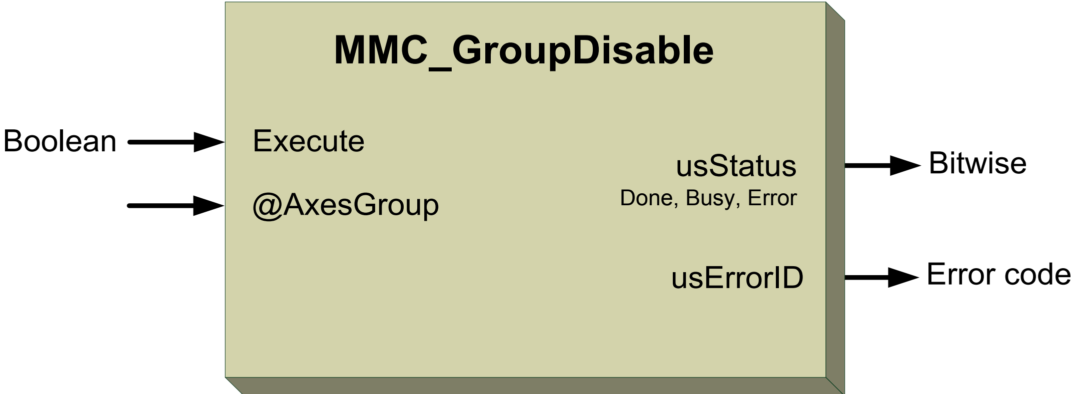


Figure 5-154: MMC_GroupDisable function block

5.15.20.2 Function Block Code Example

```
int rc;
MMC_GROUPDISABLE_IN        stGroupDisable_in;
MMC_GROUPDISABLE_OUT       stGroupDisable_out;
//
// Inserting the structure parameters:
stGroupDisable_in.ucExecute     = 0;     // Start the execution command

//
rc = MMC_GroupDisableCmd (hConn, iAxisRef, &stGroupDisable_in, &stGroupDisable_out);
if (rc != 0)
{
    HandleError();
}
```

5.15.21 MMC_GroupEnable

For multi-axis systems. Changes the state for a group from GroupDisabled to GroupStandby. This is an administrative function block, since no movement is generated.

```
MMC_LIB_API int MMC_GroupEnableCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPENABLE_IN* pInParam,
OUT MMC_GROUPENABLE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

GMAS Programming(IEC 61331 Program)\ElmoGroupAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPENABLE_IN** input data structure using the MMC_GroupEnable function.

pOutParam

Points to the **MMC_GROUPENABLE_OUT** output structure receiving information, as a result of calling the MMC_GroupEnable function.

Remarks

The command does not influence the power state of any of the single axes in the group.

Scope

All

MMC_GROUPENABLE_IN Structure

```
typedef struct{
unsigned char ucExecute;
}MMC_GROUPENABLE_IN;
```

Parameters

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_GROUPENABLE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_GROUPENABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-138 describes the function block for MMC_GroupEnable as applied within the IEC 61131 programming.

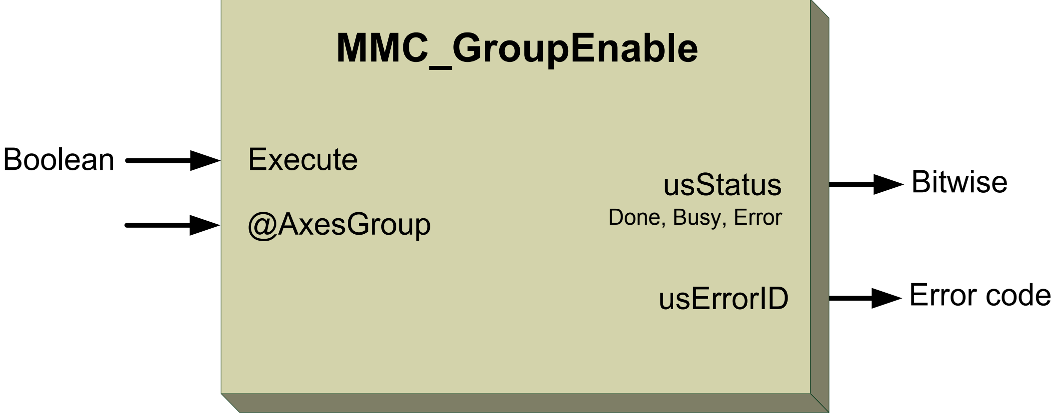


Figure 5-155: MMC_GroupEnable function block

5.15.21.2 Function Block Code Example

```
int rc;
MMC_GROUPENABLE_IN          stGroupEnable_in;
MMC_GROUPENABLE_OUT         stGroupEnable_out;
//
// Inserting the structure parameters:
stGroupEnable_in.ucExecute      = 1;     // Start the execution command

//
rc = MMC_GroupEnableCmd (hConn, iAxisRef, &stGroupEnable_in, &stGroupEnable_out);
if (rc != 0)
{
    HandleError();
}
```

5.15.22 MMC_GroupReadActualPosition

For multi-axis systems. Returns the actual position in the selected coordinate system of an axes group. This is an administrative function block, since no movement is generated.

```
MMC_LIB_API int MMC_GroupReadActualPosition(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPREADACTUALPOSITION_IN* pInParam,
OUT MMC_GROUPREADACTUALPOSITION_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

GMAS Programming(IEC 61331 Program)\ElmoGroupAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPREADACTUALPOSITION_IN** input data structure using the MMC_GroupReadActualPosition function.

pOutParam

Points to the **MMC_GROUPREADACTUALPOSITION_OUT** output structure receiving information, as a result of calling the MMC_GroupReadActualPosition function.

Remarks

None

Scope

All

MMC_GROUPREADACTUALPOSITION_IN Structure

```
typedef struct{
MC_COORD_SYSTEM_ENUM eCoordSystem;
unsigned char ucEnable;
}MMC_GROUPREADACTUALPOSITION_IN;
```

Parameters

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator. The options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_GROUPREADACTUALPOSITION_OUT Structure

```
typedef struct{
double dbPosition[NC_MAX_NUM_AXES_IN_NODE];
unsigned short usStatus;
short sErrorID;
}MMC_GROUPREADACTUALPOSITION_OUT;
```

Parameters

dbPosition [NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute end positions for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbPosition is a vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-139 describes the function block for MMC_GroupReadActualPosition as applied within the IEC 61131 programming.

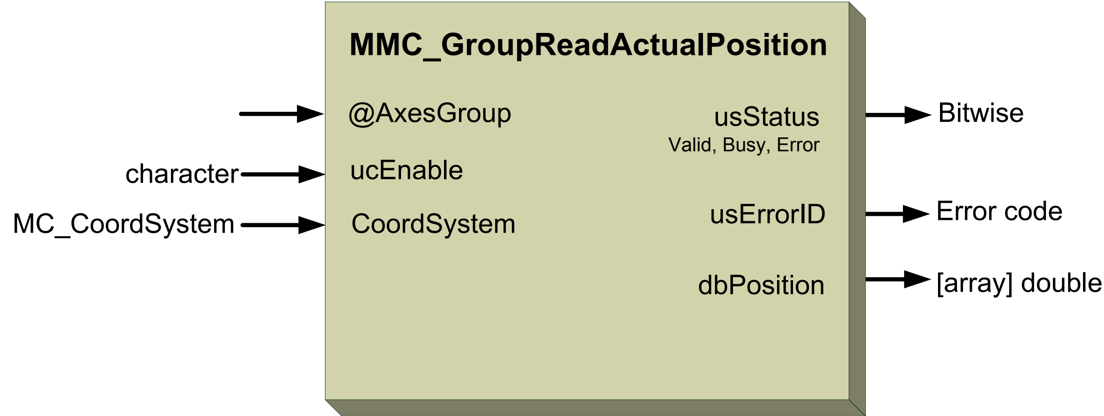


Figure 5-156: MMC_GroupReadActualPosition function block

5.15.22.2 Function Block Code Example

```
int rc;
MMC_GROUPREADACTUALPOSITION_IN      stGroupReadActualPos_in;
MMC_GROUPREADACTUALPOSITION_OUT     stGroupReadActualPos_out;
//
// Inserting the structure parameters:
stGroupReadActualPos_in.eCoordSystem     = MC_MCS_COORD; //Define types of supported coordinate
systems
stGroupReadActualPos_in.ucEnable         = 1;            //Get the actual position in the
selected coordinate system of the axes group
//
rc = MMC_GroupReadActualPosition (hConn, iAxisRef, &stGroupReadActualPos_in,
&stGroupReadActualPos_out);
printf("Group Actual Position[%ld] ErrId[%d]\n", (long
int)stGroupReadActualPos_out.dbPosition, (short)stGroupReadActualPos_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.15.23 MMC_GroupReadActualVelocity

Returns the actual velocity in the selected coordinate system of an axes group. This is an administrative function block, since no movement is generated.

```
MMC_LIB_API int MMC_GroupReadActualVelocity(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPREADACTUALVELOCITY_IN* pInParam,
OUT MMC_GROUPREADACTUALVELOCITY_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPREADACTUALVELOCITY_IN** input data structure using the MMC_GroupReadActualVelocity function.

pOutParam

Points to the **MMC_GROUPREADACTUALVELOCITY_OUT** output structure receiving information, as a result of calling the MMC_GroupReadActualVelocity function.

Remarks

None

Scope

Relevant for ACS coordinate system only and not MCS. Conditional on the values of the CANbus PDO3 and PDO4.

MMC_GROUPREADACTUALVELOCITY_IN Structure

```
typedef struct{
MC_COORD_SYSTEM_ENUM eCoordSystem;
unsigned char ucEnable;
}MMC_GROUPREADACTUALVELOCITY_IN;
```

Parameters

eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator. The options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_GROUPREADACTUALVELOCITY_OUT Structure

```
typedef struct{
double dVelocity[NC_MAX_NUM_AXES_IN_NODE];
double dPathVelocity;
unsigned short usStatus;
short sErrorID;
}MMC_GROUPREADACTUALVELOCITY_OUT;
```

Parameters

dVelocity [NC_MAX_NUM_AXES_IN_NODE]

Current velocity of the group:

- For ACS the velocities of the different axes
- For MCS it provides the velocity of the TCP

The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dVelocity Any -ve or +ve array double value in the axis's unit [u/s]

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dPathVelocity

Current path velocity (speed, combined result) of the TCP. *dPathVelocity* can have values of any -ve or +ve double values in axis's unit [u].

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-140 describes the function block for MMC_GroupReadActualVelocity

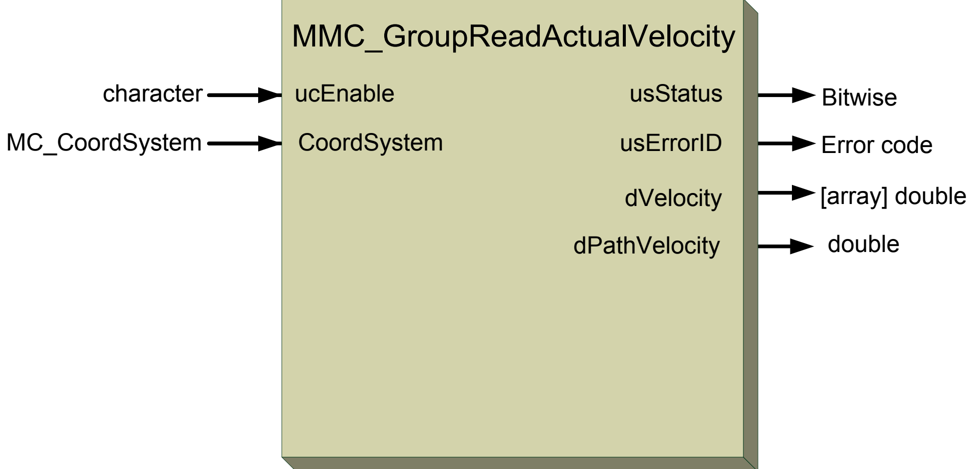


Figure 5-157: MMC_GroupReadActualVelocity function block

5.15.23.2 Function Block Code Example

```
int rc;
MMC_GROUPREADACTUALVELOCITY_IN       stGroupReadActualVel_in;
MMC_GROUPREADACTUALVELOCITY_OUT      stGroupReadActualVel_out;
//
// Inserting the structure parameters:
stGroupReadActualVel_in.eCoordSystem     = MC_MCS_COORD; //Define types of supported coordinate
systems
stGroupReadActualVel_in.ucEnable         = 1;            //Get the actual position in the
selected coordinate system of the axes group
//
rc = MMC_GroupReadActualVelocity (hConn, iAxisRef, &stGroupReadActualVel_in,
&stGroupReadActualVel_out);
printf("Group Actual Velocity[%ld] ErrId[%d]\n", (long int)stGroupReadActualVel_out.dVelocity,
(short)stGroupReadActualVel_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.15.24 MMC_GroupReadError

Describes general axes group errors not relating to the function blocks. This is an administrative function block, since no movement is generated. This function is not in use at this moment.

```
MMC_LIB_API int MMC_GroupReadError(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPREADERROR_IN* pInParam,
OUT MMC_GROUPREADERROR_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPREADERROR_IN** input data structure using the MMC_GroupReadError function.

pOutParam

Points to the **MMC_GROUPREADERROR_OUT** output structure receiving information, as a result of calling the MMC_GroupReadError function.

Remarks

Examples are (software) limit switch exceeded or single axis error in GroupStandby state. This function is not in use at this moment and will always produce a usGroupErrorID of 256.

Scope

MMC_GROUPREADERROR_IN Structure

```
typedef struct{
unsigned char ucEnable;
}MMC_GROUPREADERROR_IN;
```

Parameters

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_GROUPREADERROR_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned short usGroupErrorID;
}MMC_GROUPREADERROR_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

usGroupErrorID

Returned command group error ID. Signals where an group error has occurred within function block. These values are vendor specific. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Displays an error code integer.

Figure 5-141 describes the function block for MMC_GroupReadError

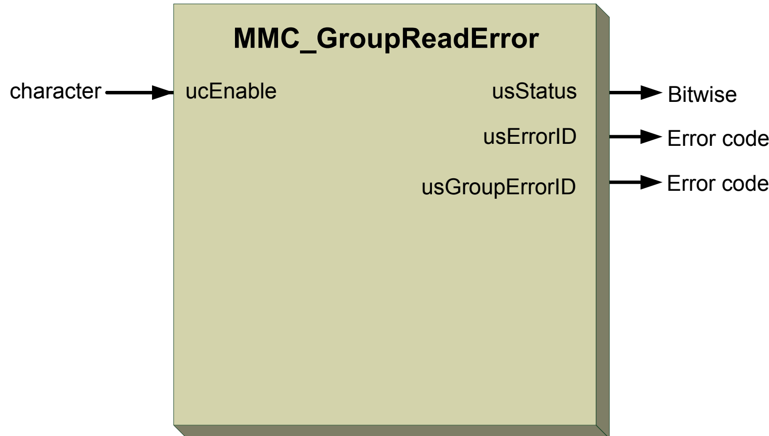


Figure 5-158: MMC_GroupReadError function block

5.15.24.2 Function Block Code Example

```
int rc;
MMC_GROUPREADERROR_IN      stGroupReadError_in;
MMC_GROUPREADERROR_OUT     stGroupReadError_out;
//
// Inserting the structure parameters:
stGroupReadError_in.ucEnable    = 1;      //Enabled
//
rc = MMC_GroupReadError (hConn, iAxisRef, &stGroupReadError_in, &stGroupReadError_out);
printf("Group Error[%ld] ErrId[%d]\n", (long int)stGroupReadError_out.usGroupErrorID,
(short)stGroupReadError_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

5.15.25 MMC_GroupReadStatus

For multiple Axes. Returns the status of an axes group according to the active Group function block. This is an administrative function block, since no movement is generated.

```
MMC_LIB_API int MMC_GroupReadStatusCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPREADSTATUS_IN* pInParam,
OUT MMC_GROUPREADSTATUS_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h

GMAS Programming(IEC 61331 Program)\ElmoGroupAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPREADSTATUS_IN** input data structure using the MMC_GroupReadStatus function.

pOutParam

Points to the **MMC_GROUPREADSTATUS_OUT** output structure receiving information, as a result of calling the MMC_GroupReadStatus function.

Remarks

The outputs reflect the commanded state of the group.

Scope

All

MMC_GROUPREADSTATUS_IN Structure

```
typedef struct{
unsigned int uiHndlr;
unsigned char ucEnable;
}MMC_GROUPREADSTATUS_IN;
```

Parameters

uiHndlr

Requested group handle. Values of any integer accepted.

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_GROUPREADSTATUS_OUT Structure

```
typedef struct{
unsigned long ulState;
unsigned short usStatus;
short sErrorID;
unsigned short usGroupErrorID;
}MMC_GROUPREADSTATUS_OUT;
```

Parameters

ulState

Group current state. Returned command status.

Refer to the sub-section **5.4 Axis Status** for the range of axis status bit masks with bitwise values.

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

usGroupErrorID

Returned command group error ID. Signals where an group error has occurred within function block. These values are vendor specific. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Displays an error code integer.

Figure 5-142 describes the function block for MMC_GroupReadStatus as applied within the IEC 61131 programming.

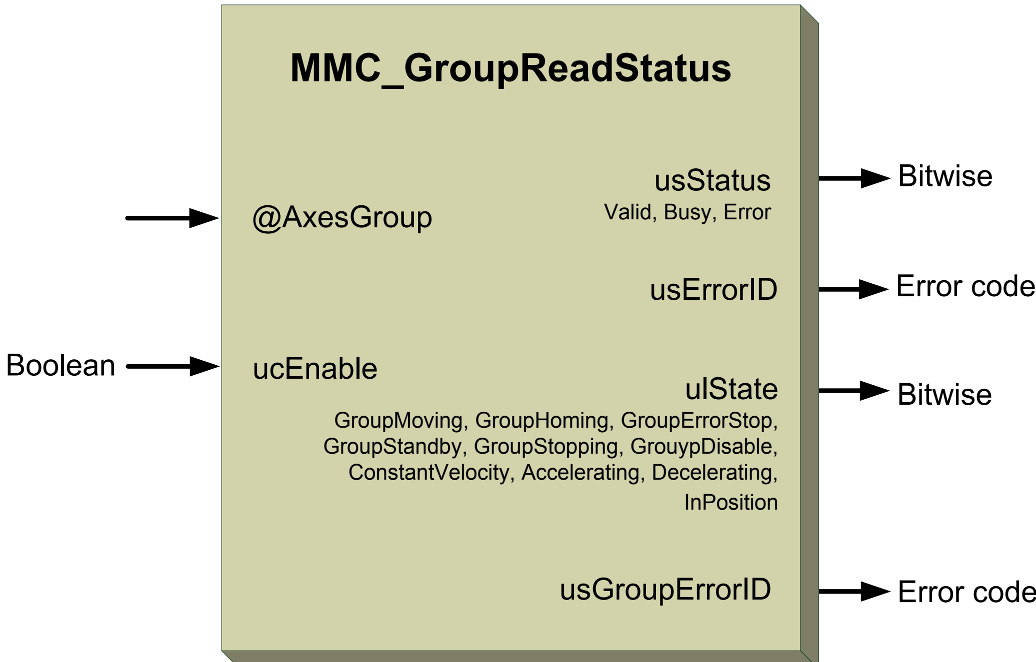


Figure 5-159: MMC_GroupReadStatus function block

5.15.25.2 Function Block Code Example

```
int rc;
MMC_GROUPREADSTATUS_IN      stGroupReadStatus_in;
MMC_GROUPREADSTATUS_OUT     stGroupReadStatus_out;
//
// Inserting the structure parameters:
stGroupReadStatus_in.uiHndlr    = 35;     // Returned function block handle
stGroupReadStatus_in.ucEnable   = 1;      //Enabled
//
rc = MMC_GroupReadStatusCmd (hConn, iAxisRef, &stGroupReadStatus_in, &stGroupReadStatus_out);
printf("Group Status[%ld] ErrId[%d]\n", (long int)stGroupReadStatus_out.usGroupErrorID,
(short)stGroupReadStatus_out.sErrorID);
printf("Group Status[%ld]\n", (long int)stGroupReadStatus_out.ulState);
if (rc != 0)
{
    HandleError();
}
```

5.15.26 MMC_GroupReset

For multiple Axes. Makes the transition from the state GroupErrorStop to GroupDisabled by resetting all internal group-related errors – it does not affect the output of the function block instances.

```
MMC_LIB_API int MMC_GroupResetCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPRESET_IN* pInParam,
OUT MMC_GROUPRESET_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PLCopen_group_API.h
GMAS Programming(IEC 61331 Program)\ElmoGroupAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPRESET_IN** input data structure using the MMC_GroupReset function.

pOutParam

Points to the **MMC_GROUPRESET_OUT** output structure receiving information, as a result of calling the MMC_GroupReset function.

Remarks

This function block also resets all axes in this group like MMC_Reset.

Scope

All

MMC_GROUPRESET_IN Structure

```
typedef struct{
unsigned char ucExecute;
}MMC_GROUPRESET_IN;
```

Parameters

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_GROUPRESET_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_GROUPRESET_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-143 describes the function block for MMC_GroupReset as applied within the IEC 61131 programming.

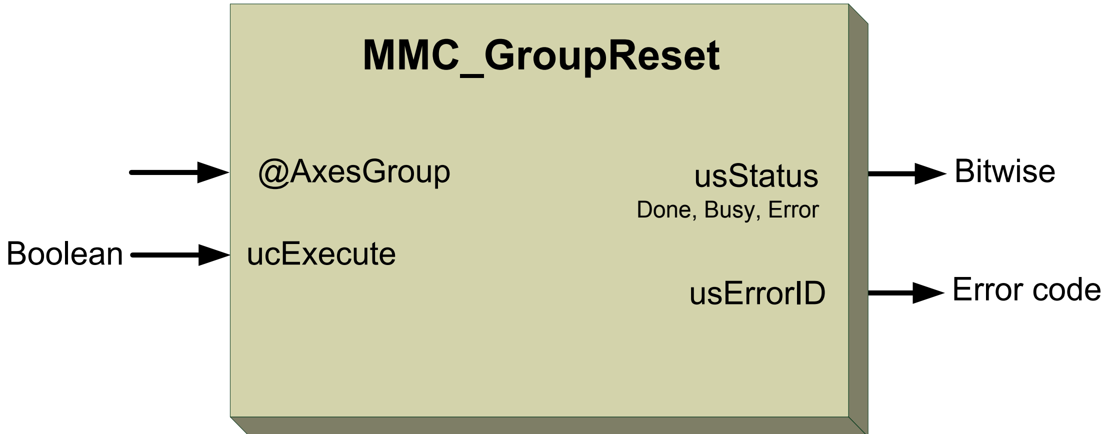


Figure 5-160: MMC_GroupReset function block

5.15.26.2 Function Block Code Example

```
int rc;
MMC_GROUPRESET_IN      stGroupReset_in;
MMC_GROUPRESET_OUT     stGroupReset_out;
//
// Inserting the structure parameters:
stGroupReset_in.ucExecute  = 1;    // Start the execution command

//
rc = MMC_GroupResetCmd (hConn, iAxisRef, &stGroupReset_in, &stGroupReset_out);
if (rc != 0)
{
    HandleError();
}
```

5.15.27 MMC_GroupSetOverride

For multi-axes. Sets the values of override for the coordinated motion of several axes, and all functions operating on that axes group.

```
MMC_LIB_API int MMC_GroupSetOverrideCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SETOVERRIDE_IN* pInParam,
OUT MMC_SETOVERRIDE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed – Not supported

Source **Function**
GMAS\includes\MMC_PLCopen_single_API.h
Structures
GMAS\includes\MMC_PLCopen_group_API.h
IEC
GMAS Programming(IEC 61331 Program)\ElmoGroupAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SETOVERRIDE_IN** input data structure using the MMC_SetOverride function.

pOutParam

Points to the **MMC_SETOVERRIDE_OUT** output structure receiving information as a result of calling the MMC_SetOverride function.

Remarks

The override parameters act as a factor that is multiplied to the commanded vector velocity, acceleration, deceleration, and jerk of all axes group function blocks. MMC_GroupSetOverride is not applicable to master-slave group synchronized motion commands.

MMC_GroupSetOverride does not influence the state diagram of the axis group. The following should be noted:

- The Input *fAccFactor* acts on positive and negative acceleration (deceleration).
- This function block sets the override factor, which is valid until a new override is set.
- The default values of the override factor are 1.0. The value of the overrides can be between 0.0 and 1.0. The behavior of values > 1.0 is vendor specific. Values < 0.0 are not allowed. The value 0.0 is not

allowed for both *fAccFactor* and *fJerkFactor*, and generates an error.

- The value 0.0 set to the *fVelFactor* stops the axis without bringing it to the state GroupStandby.
- Override does not act on slave axes groups in the state Group Synchronized motion.
- *fVelFactor* can be changed at any time and acts directly on the ongoing motion.
- Reducing the *fAccFactor* and/or *fJerkFactor* can lead to a position overshoot – a possible cause of damage.

Scope

All

MMC_SETOVERRIDE_IN Structure

```
typedef struct{
float fVelFactor;
float fAccFactor;
float fJerkFactor;
unsigned short usUpdateVelFactorIdx;
unsigned char ucEnable;
}MMC_SETOVERRIDE_IN;
```

Parameters

fVelFactor

New override factor for the velocity. Any +ve float value between [0 – 1].

fAccFactor

New override factor for the acceleration/deceleration. ACC/Jerk Factors are NOT supported at this time. For future compatibility, enter "1" in the function call.

fJerkFactor

New override factor for the jerk. ACC/Jerk Factors are NOT supported at this time. For future compatibility, enter "1" in the function call.

usUpdateVelFactorIdx

Index of changed velocity factor. Vendor defined. The default is 0. Has integer values of 0 - 2

This variable is not in use at this moment.

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_SETOVERRIDE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
} MMC_SETOVERRIDE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-144 describes the function block for MMC_GroupSetOverride as applied within the IEC 61131 programming.

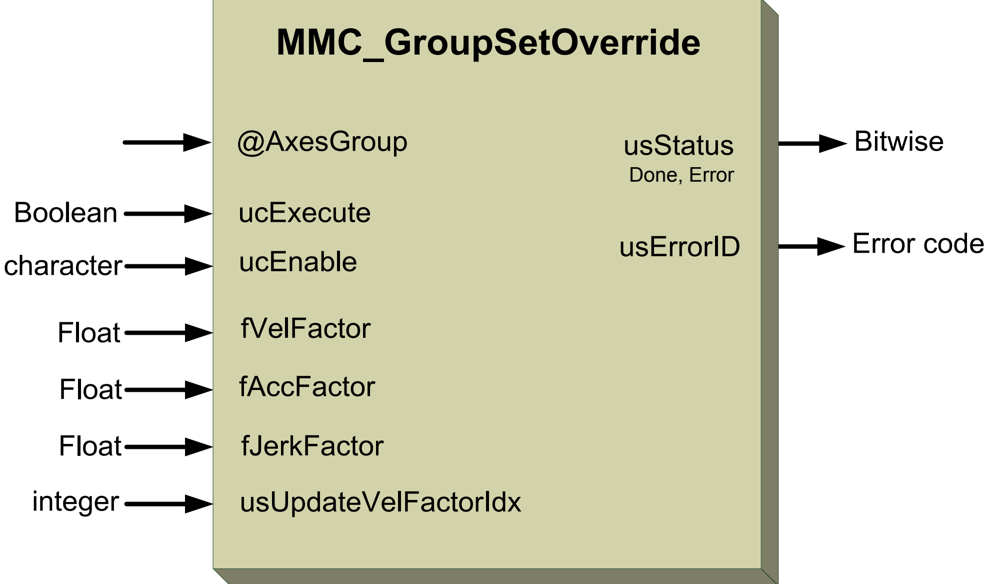


Figure 5-161: MMC_GroupSetOverride function block

5.15.27.2 Function Block Code Example

```
int rc;
MMC_SETOVERRIDE_IN          stGSetOverride_in;
MMC_SETOVERRIDE_OUT         stGSetOverride_out;
//
// Inserting the structure parameters:
stGSetOverride_in.fAccFactor            = 0.4;    //New override factor for the
acceleration/deceleration
stGSetOverride_in.fJerkFactor       = 0.1;    //New override factor for the jerk
stGSetOverride_in.usUpdateVelFactorIdx  = 1;      //Index of changed velocity factor
stGSetOverride_in.fVelFactor            = 0.4;    //New override factor for the velocity
stGSetOverride_in.ucEnable              = 1;      //Enabled
//
rc = MMC_GroupSetOverrideCmd (hConn, iAxisRef, &stGSetOverride_in, &stGSetOverride_out);
if (rc != 0)
{
    HandleError();
}
```

5.15.27.3 Implementation Example

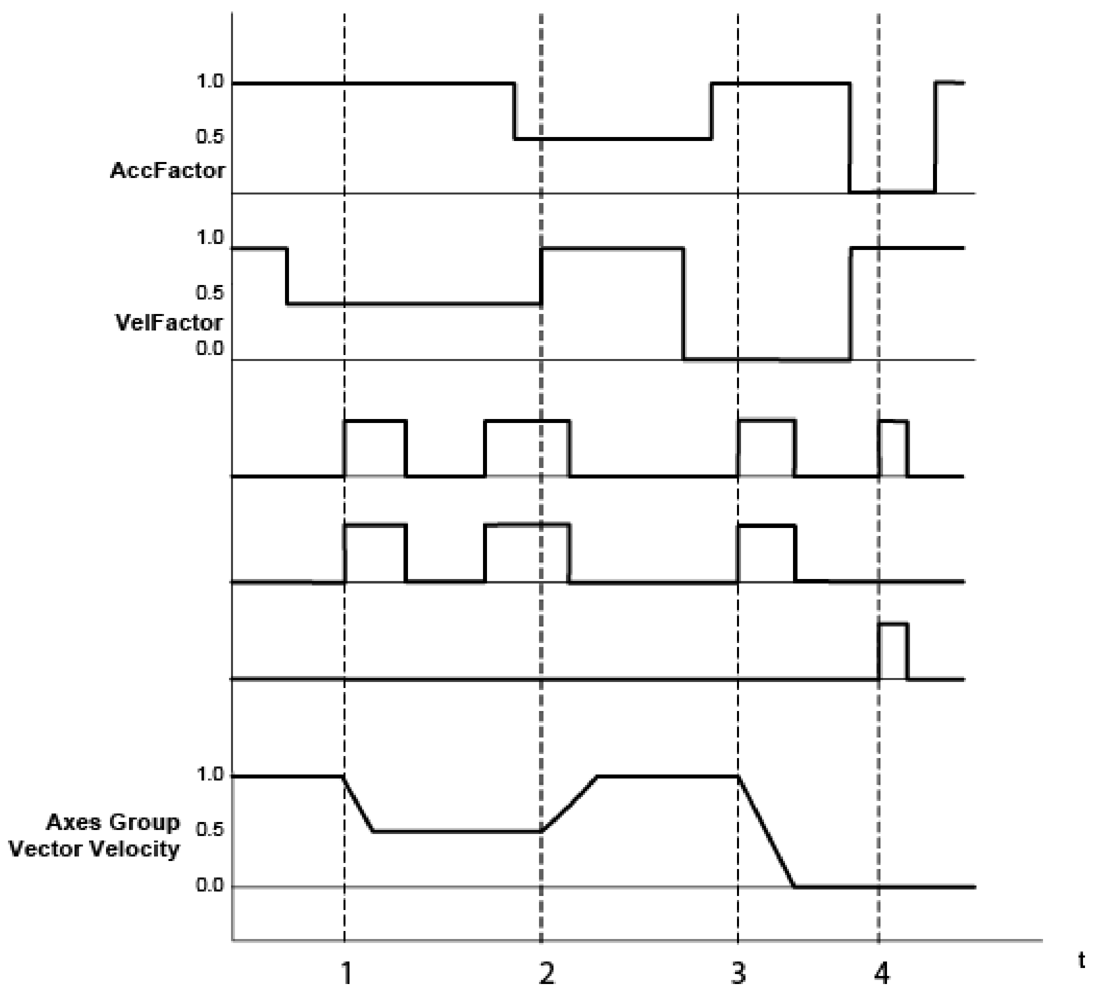


Figure 5-162: Timing diagram for MMC_GroupSetOverride – Example

1. Axes Group Vector Velocity changes to 50% with 100% of deceleration
2. Axes Group Vector Velocity back to 100% with 50% acceleration
3. Axes Group Vector Velocity moves to 0% with 100% deceleration
4. No Change, because *fAccFactor* 0.0 is not allowed; Error is set

5.15.28 MMC_GroupSetPosition

For multiple Axes. Sets the Position of all axes in a group without moving the axes.

```
MMC_LIB_API int MMC_GroupSetPositionCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GROUPSETPOSITION_IN* pInParam,
OUT MMC_GROUPSETPOSITION_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed – Not supported
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GROUPSETPOSITION_IN** input data structure using the MMC_GroupSetPosition function.

pOutParam

Points to the **MMC_GROUPSETPOSITION_OUT** output structure receiving information, as a result of calling the MMC_GroupSetPosition function.

Remarks

The new coordinates are described in an array. Within the coordinate system input, the respective coordinate system is selected as a method of referencing or a transformation. MMC_GroupSetPosition shifts the position of the addressed coordinate system and affects the higher level coordinate systems (so if ACS is selected, MCS and PCS are affected).

Scope

All

MMC_GROUPSETPOSITION_IN Structure

```
typedef struct{
double dbPosition[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
unsigned char ucMode;
}MMC_GROUPSETPOSITION_IN;
```

Parameters

dbPosition

Target position for the motion of the axis when conditions are met. Any -ve or +ve double values in technical unit [u].

Array of coordinates, incl. positions and orientations (Distance if Mode = RELATIVE). The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2....15].

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD	= 0
MC_ACS_COORD	= 1
MC_MCS_COORD	= 2
MC_PCS_COORD	= 3

MC_BUFFERED_MODE_ENUM eBufferMode

MC_BufferMode defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting — Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered — The next function block affects the axis as soon as the previous movement is completed.

BlendingLow — The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-

position (1).

BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucMode

RELATIVE =True, ABSOLUTE = False (Default)

RELATIVE means that Position is added to the actual position value of the axis at the time of execution. This results in a recalibration by a specified distance. ABSOLUTE means that the actual position value of the axis is set to the value specified in the Position parameter.

Values accepted are Boolean, TRUE/FALSE.

ucPosMode

Boolean values for the position mode can be absolute mode = 0, or relative mode = 1

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_GROUPSETPOSITION_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_GROUPSETPOSITION_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-146 describes the function block for MMC_GroupSetPosition

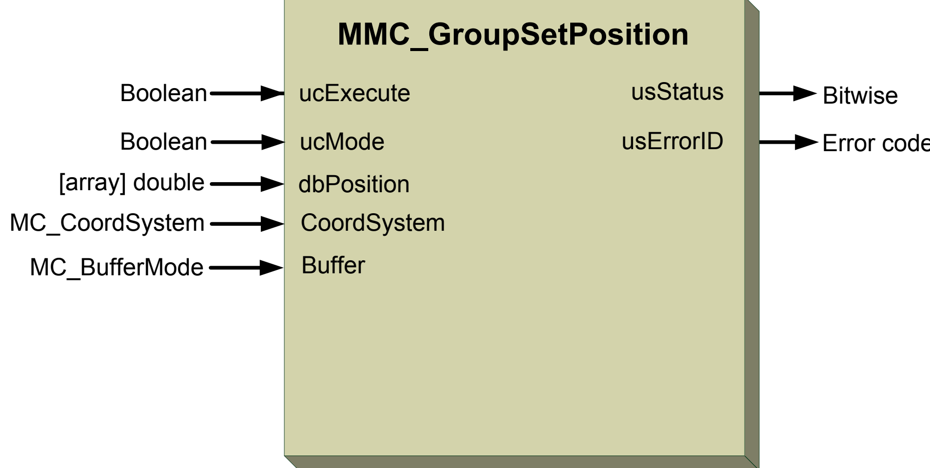


Figure 5-163: MMC_GroupSetPosition function block

5.15.28.2 Function Block Code Example

```
int rc;
MMC_GROUPSETPOSITION_IN    stGroupSetPos_in;
MMC_GROUPSETPOSITION_OUT   stGroupSetPos_out;
//
// Inserting the structure parameters:
stGroupSetPos_in.eCordSystem    = MC_MCS_COORD; //Define types of supported coordinate systems
stGroupSetPos_in.eBufferMode    = MC_BUFFERED_MODE;//MC_BUFFERED_MODE_ENUM Defines the behavior
of the axis
stGroupSetPos_in.dbPosition[2]  = 80000.0;//Array of coordinates, positions and orientations
stGroupSetPos_in.dbPosition[3]  = 50000.0;
stGroupSetPos_in.dbPosition[4]  = 60000.0;
stGroupSetPos_in.ucMode         = 1;//Position is added to the actual position value of the
axis
stGroupSetPos_in.ucExecute      = 1;
//
rc = MMC_GroupSetPositionCmd (hConn, iAxisRef, &stGroupSetPos_in, &stGroupSetPos_out);
if (rc != 0)
{
    HandleError();
}
```

5.15.29 MMC_RemoveAxisFromGroup

For multiple Axes. Removes one axis from the group AxesGroup. This is an administrative function block, since no movement is generated.

```
MMC_LIB_API int MMC_RemoveAxisFromGroup(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_REMOVEAXISFROMGROUP_IN* pInParam,
OUT MMC_REMOVEAXISFROMGROUP_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Not Supported
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_REMOVEAXISFROMGROUP_IN** input data structure using the MMC_RemoveAxisFromGroup function.

pOutParam

Points to the **MMC_REMOVEAXISFROMGROUP_OUT** output structure receiving information, as a result of calling the MMC_RemoveAxisFromGroup function.

Remarks

The command cannot be buffered. If there is no axis remaining in the group, the state automatically changes to GroupDisabled.

Scope

Only available if the group is **disabled**. If issued on a group that is in GroupEnabled, GroupStandby, or GroupErrorStop state, it generates an error and the function block is not executed. If GroupEnabled, an explicit error is output:

NC_UNSUITABLE_NODE_STATE

MMC_REMOVEAXISFROMGROUP_IN Structure

```
typedef struct{
NC_IDENT_IN_GROUP_ENUM eIdentInGroup;
}MMC_REMOVEAXISFROMGROUP_IN;
```

Parameters

eIdentInGroup

The NC_IDENT_IN_GROUP_ENUM enumerator identifies the order and Nodes in the group of the added axis. Performed via an enumerator to give the different axes a name in the order, which can be coupled to the names in the kinematic model. The options are:

NC_NODE_1_ID = 0

……………

NC_NODE_16_ID = 15

MMC_REMOVEAXISFROMGROUP_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_REMOVEAXISFROMGROUP_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-147 describes the function block for MMC_RemoveAxisFromGroup as applied within the IEC 61131 programming.

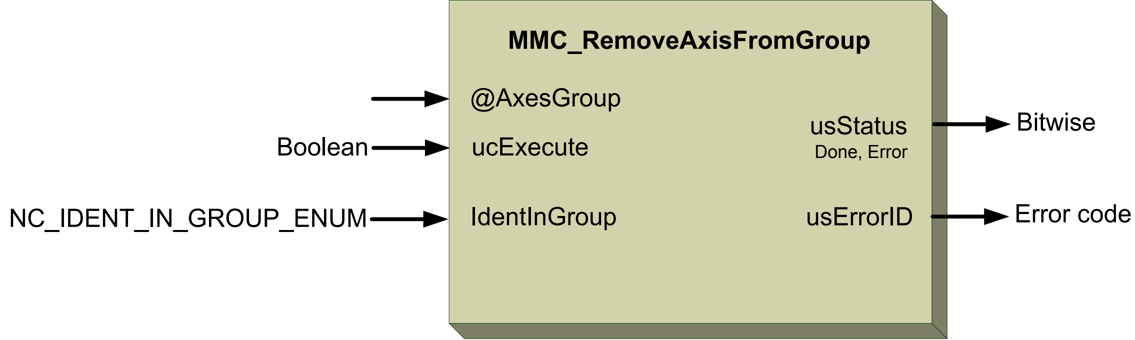


Figure 5-164: MMC_RemoveAxisFromGroup function block

5.15.29.2 Function Block Code Example

```
int rc;
MMC_REMOVEAXISFROMGROUP_IN      stRemoveAxisFromGrp_in;
MMC_REMOVEAXISFROMGROUP_OUT     stRemoveAxisFromGrp_out;
//
// Inserting the structure parameters:
stRemoveAxisFromGrp_in.eIdentInGroup     = NC_NODE_4_ID;//Identifies the supported axes (nodes)
in a group
//
rc = MMC_RemoveAxisFromGroup (hConn, iAxisRef, &stRemoveAxisFromGrp_in,
&stRemoveAxisFromGrp_out);
if (rc != 0)
{
    HandleError();
}
```

5.15.30 MMC_GroupReadParameter

Reads a specific group axes parameter.

MMC_LIB_API int MMC_GroupReadParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPARAMETER_IN* pInParam,
OUT MMC_READPARAMETER_OUT* pOutParam
);

Motion Mode	NC - Irrelevant	Distributed – Irrelevant
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPARAMETER_IN** input data structure using the MMC_GroupReadParameter function.

pOutParam

Points to the **MMC_READPARAMETER_OUT** output structure receiving information as a result of calling the MMC_GroupReadParameter function.

Remarks

N/A

Scope

None

MMC_READPARAMETER_IN Structure

```
typedef struct{
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_READPARAMETER_IN;
```

Parameters

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READPARAMETER_OUT Structure

```
typedef struct{
double dbValue;
unsigned short usStatus;
short sErrorID;
}MMC_READPARAMETER_OUT;
```

Parameters

dbValue

Output of the specific parameter. Any Double value.

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-148 describes the function block for MMC_GroupReadParameter

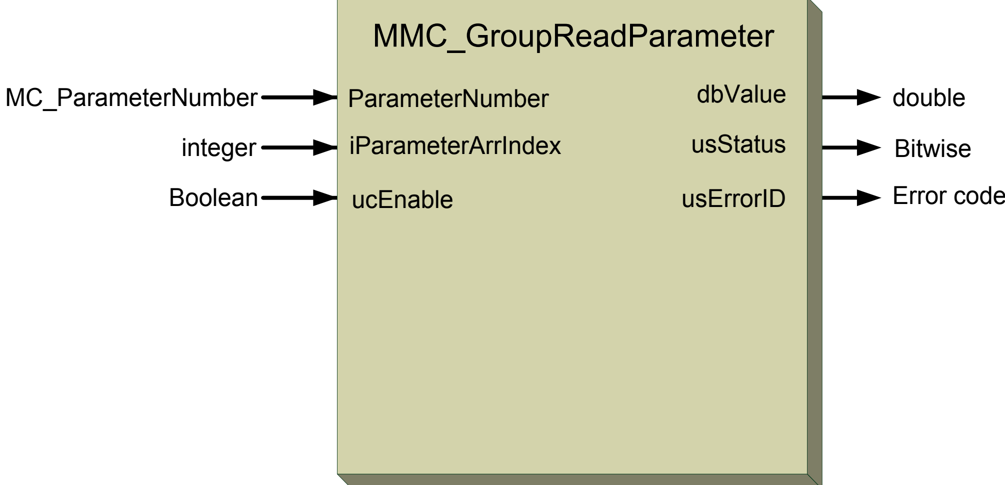


Figure 5-165: MMC_GroupReadParameter function block

5.15.31 MMC_GroupReadBoolParameter

Reads a specific group axes Boolean parameter.

MMC_LIB_API int MMC_GroupWriteParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READBOOLPARAMETER_IN* pInParam,
OUT MMC_READBOOLPARAMETER_OUT* pOutParam
);

Motion Mode	NC - Irrelevant	Distributed – Irrelevant
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READBOOLPARAMETER_IN** input data structure using the MMC_GroupReadBoolParameter function.

pOutParam

Points to the **MMC_READBOOLPARAMETER_OUT** output structure receiving information as a result of calling the MMC_GroupReadBoolParameter function.

Remarks

N/A

Scope

None

MMC_READBOOLPARAMETER_IN Structure

```
typedef struct{
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_READBOOLPARAMETER_IN;
```

Parameters

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_READBOOLPARAMETER_OUT Structure

```
typedef struct{
long lValue;
unsigned short usStatus;
short sErrorID;
}MMC_READBOOLPARAMETER_OUT;
```

Parameters

lValue

Boolean parameters integer value

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-149 describes the function block for MMC_GroupReadBoolParameter

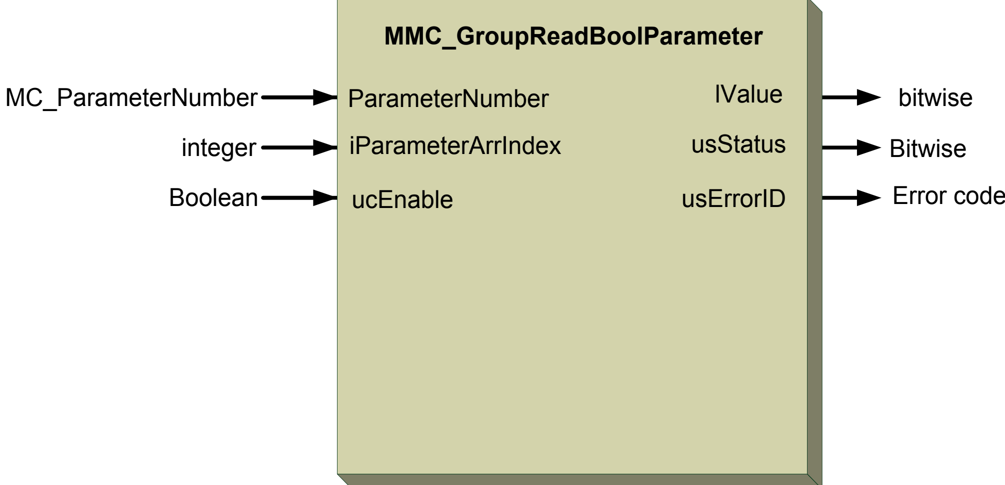


Figure 5-166: MMC_GroupReadBoolParameter function block

5.15.32 MMC_GroupWriteParameter

Modifies the value of a specific group axes parameter.

```
MMC_LIB_API int MMC_GroupWriteParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPARAMETER_IN* pInParam,
OUT MMC_WRITEPARAMETER_OUT* pOutParam
);
```

Motion Mode	NC - Irrelevant	Distributed – Irrelevant
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPARAMETER_IN** input data structure using the MMC_GroupWriteParameter function.

pOutParam

Points to the **MMC_WRITEPARAMETER_OUT** output structure receiving information, as a result of calling the MMC_GroupWriteParameter function.

Remarks

N/A

Scope

None

MMC_WRITEPARAMETER_IN Structure

```
typedef struct{
double dbValue;
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_WRITEPARAMETER_IN;
```

Parameters

dbValue

Parameter value. Any positive double (8 bytes)value up to 15 digits.

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_WRITEPARAMETER_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEPARAMETER_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-150 describes the function block for MMC_GroupWriteParameter

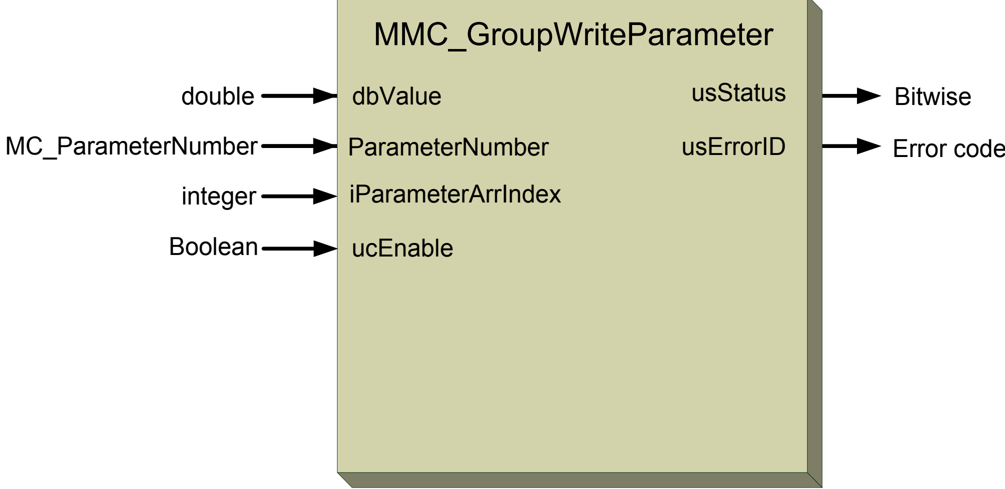


Figure 5-167: MMC_GroupWriteParameter function block

5.15.33 MMC_GroupWriteBoolParameter

Modifies the value of a specific group axes Boolean parameter.

```
MMC_LIB_API int MMC_GroupWriteBoolParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEBOOLPARAMETER_IN* pInParam,
OUT MMC_WRITEBOOLPARAMETER_OUT* pOutParam
);
```

Motion Mode	NC - Irrelevant	Distributed – Irrelevant
Source	GMAS\includes\MMC_PLCopen_group_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEBOOLPARAMETER_IN** input data structure using the MMC_GroupWriteBoolParameter function.

pOutParam

Points to the **MMC_WRITEBOOLPARAMETER_OUT** output structure receiving information, as a result of calling the MMC_GroupWriteBoolParameter function.

Remarks

N/A

Scope

None

MMC_WRITEBOOLPARAMETER_IN Structure

```
typedef struct{
long lValue;
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterArrIndex;
unsigned char ucEnable;
}MMC_WRITEBOOLPARAMETER_IN;
```

Parameters

lValue

Input value. Any +ve integer.

eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterArrIndex

Array index parameter. Any +ve integer values

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_WRITEBOOLPARAMETER_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEBOOLPARAMETER_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 5-151 describes the function block for MMC_GroupWriteBoolParameter

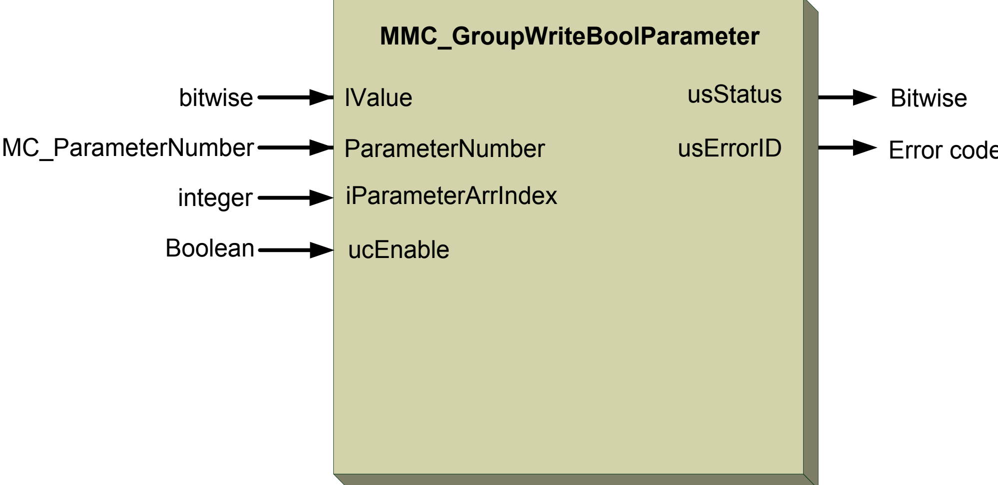


Figure 5-168: MMC_GroupWriteBoolParameter function block

5.15.34 MMC_GetGroupMembersInfo

Returns information about a specific group and its members.

```
MMC_LIB_API int MMC_GetGroupMembersInfo(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETGROUPMEMBERSINFO_IN * pInParam,
OUT MMC_GETGROUPMEMBERSINFO_OUT * pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h GMAS Programming(IEC 61331 Program)\ElmoGroupAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETGROUPMEMBERSINFO_IN** input data structure using the MMC_GetGroupMembersInfo function.

pOutParam

Points to the **MMC_GETGROUPMEMBERSINFO_OUT** output structure receiving information, as a result of calling the MMC_GetGroupMembersInfo function.

Remarks

None

Scope

All

MMC_GETGROUPMEMBERSINFO_IN Structure

t typedef struct mmc_getgroupmembersinfo_in{
unsigned char ucDummy;
} MMC_GETGROUPMEMBERSINFO_IN;

Parameters

ucDummy

Any dummy value accepted.

MMC_GETGROUPMEMBERSINFO_OUT Structure

typedef struct mmc_getgroupmembersinfo_out{
char pAxesNames[NC_MAX_NUM_AXES_IN_NODE][NODE_NAME_MAX_LENGTH];
unsigned short pAxesReferences[NC_MAX_NUM_AXES_IN_NODE];
unsigned short pDeviceID[NC_MAX_NUM_AXES_IN_NODE];
unsigned short usStatus;
short sErrorID;
unsigned char ucNumOfAxes;
} MMC_GETGROUPMEMBERSINFO_OUT;

Parameters

pAxesNames[NC_MAX_NUM_AXES_IN_NODE][NODE_NAME_MAX_LENGTH]

Name recorded in the Resource file of the Maestro. Value is a set of characters.

Array of axes names. The array size is equal to 16, as the maximum number of axes in a group.

[NODE_NAME_MAX_LENGTH] is the maximum name lengths = 80 characters.

pAxesReferences[NC_MAX_NUM_AXES_IN_NODE]

Reference to each axis handle in the group. Has values of any +ve number.

Array of axes references. The array size is equal to 16, as the maximum number of axes in a group.

pDeviceID[NC_MAX_NUM_AXES_IN_NODE]

Bus device ID. Has values of any +ve number.

Array of device IDs. The array size is equal to 16, as the maximum number of axes in a group.

ucNumOfAxes

The number of axes in the group. Any number set [0….16] is accepted.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 5-152 describes the function block for MMC_GetGroupMembersInfo as applied within the IEC 61131 programming.

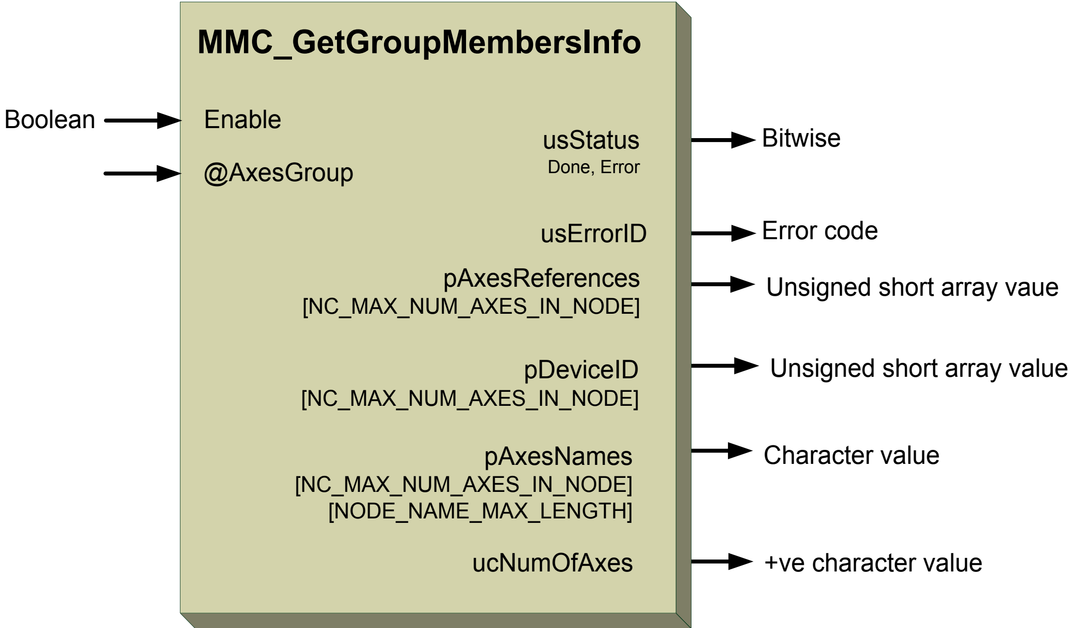


Figure 5-169: MMC_GetGroupMembersInfo function block

5.15.34.2 Function Block Code Example

```
// Declarate variables for connection
MMC_CONNECT_HNDL hConn;
MMC_IPC_CONNECTION_PARAM_STRUCT sInitConnectionInParam;
sInitConnectionInParam.uiParams = 1;
//
// Create connection (assume that there are not errors in this function)
MMC_IPCInitConnection(sInitConnectionInParam,NULL,&hConn);
//
//
MMC_AXISBYNAME_IN  sGroupByNameInParam;
MMC_AXISBYNAME_OUT sGroupByNameOutParam;
strcpy(sGroupByNameInParam.cAxisName,"v01");
//
// Get group reference (assume that there are not errors in this function)
MMC_GetGroupByNameCmd(hConn,&sGroupByNameInParam,&sGroupByNameOutParam);
//
// Create input and output structures
MMC_GETGROUPMEMBERSINFO_IN  sMembersInfoInParam;
MMC_GETGROUPMEMBERSINFO_OUT sMembersInfoOutParam;
//
// There are no neccessary inputs in the input structure (only dummy variable)
```

```
sMembersInfoInParam.ucDummy = 0;
//
// call GetGroupMembersInfo function (assume that there are not errors in this function)
MMC_GetGroupMembersInfo(hConn,sGroupByNameOutParam.usAxisIdx,&sMembersInfoInParam,&sMembersInf
oOutParam);
//
// Read the outputs
printf("The Vector [%s] has %d
axes.\n\n",sGroupByNameInParam.cAxisName,sMembersInfoOutParam.ucNumOfAxes);
for(int i = 0; i < sMembersInfoOutParam.ucNumOfAxes; i++)
{
// Print axes data
printf("Axis - [%s]\n",sMembersInfoOutParam.pAxesNames[i]);
printf("\tReference = [%d]\n",sMembersInfoOutParam.pAxesReferences[i]);
printf("\tDevice ID = [%d]\n\n",sMembersInfoOutParam.pDeviceID[i]);
}
//
```

5.15.34.3 Implementation Example

Run example in real system and get the following results. In order to obtain these results, this example should be run with a specific resource file:

The Vector v01 has 5 axes.

Axis - [b01]
  Reference = [0]
  Device ID = [1]
Axis - [b02]
  Reference = [1]
  Device ID = [2]
Axis - [b03]
  Reference = [2]
  Device ID = [3]
Axis - [b04]
  Reference = [3]
  Device ID = [4]
Axis - [b05]
  Reference = [4]
  Device ID = [5]

Chapter 6: Position, Velocity, Time (PVT) Motion

6.1 Overview

The PV/PVT special motion class describes the path given by position-velocity pairs per axis, and an optional time interval given per system (expressing as time per axis is unnecessary), and is applicable both for single and multi-axis motion.

The input data for PV motion is stored as a table in the Maestro RAM using a function. The PV motion can operate in "normal" or cyclic modes. In cyclic mode, the profiler reaches the end of table and returns to the beginning.

6.2 PV, PVT Profiler

Generally, PV/PVT motion is defined by a set of points, and if a number of points are provided in advance of the present position, the profiler can build a 3rd degree polynomial by which to calculate the next position to be downloaded to the drive. The path is calculated on-the-fly, and therefore all polynomial coefficient calculations occur in the real-time module. The PV/PVT motion does not require full data to execute, and only requires a minimal number of points.

Similar to splines, the input table is stored in the shared memory, but unlike splines, only the coordinates are stored, whereas the polynomial coefficients are calculated in the real-time module. The tables can be loaded via file, or via an N x M user-provided array. Optionally, a row or a few rows can be appended to the tables, but within reasonable constraints; for example, data cannot be appended to the currently segment, the profiler is operating on. The PV/PVT function blocks are only applicable in NC cyclic/interpolated modes.

6.3 PVT Interpolation Mode

This online spline is defined by the PVT table. Each row of this table has the form:

T_i P_{ix} V_{ix} P_{iy} V_{iy} P_{iz} V_{iz} …

Where:

i is a number of the PVT segment

T_i time to come to the point **T_i.**

There are two options for **T_i**; it may be the time interval for the motion from the previous point **(P_{i-1x}, P_{i-1y}, P_{i-1z},..)** to (**P_{ix}, P_{iy}, P_{iz},…**) in relative time,

or an absolute time from the start point **(P_{0x}, P_{0y}, P_{0z},..)**, where (**P_{ix}, P_{iy}, P_{iz},…**) is the position of the axes at point **i**, and **(V_{ix} V_{iy} V_{iz} …)** is the axes velocities at point **i**. The number of the axes is not limited by three coordinate axes and can be up to 16 axes.

In the following sections, given plots are presented for the same six point table with the different interpolation methods.

Time	Position	Velocity
0.0000000	0.000000	0.000000
1.000000	50000.000000	50000.000000
1.000000	90000.000000	40000.000000
1.000000	110000.000000	0.000000
1.000000	75000.000000	-70000.000000
1.000000	20000.000000	0.000000

6.3.1 Polynomial Interpolation Functions

6.3.1.1 Cubic polynomial – Polynomial Order 3 (eCUBIC_POLYNOM)

Cubic polynomial guarantees continuity by position and velocity. Its drawback is a discontinuity by acceleration and jerk at every connection point. An example can be seen in Figure 1-1. At each table point we see discontinuity by the acceleration.

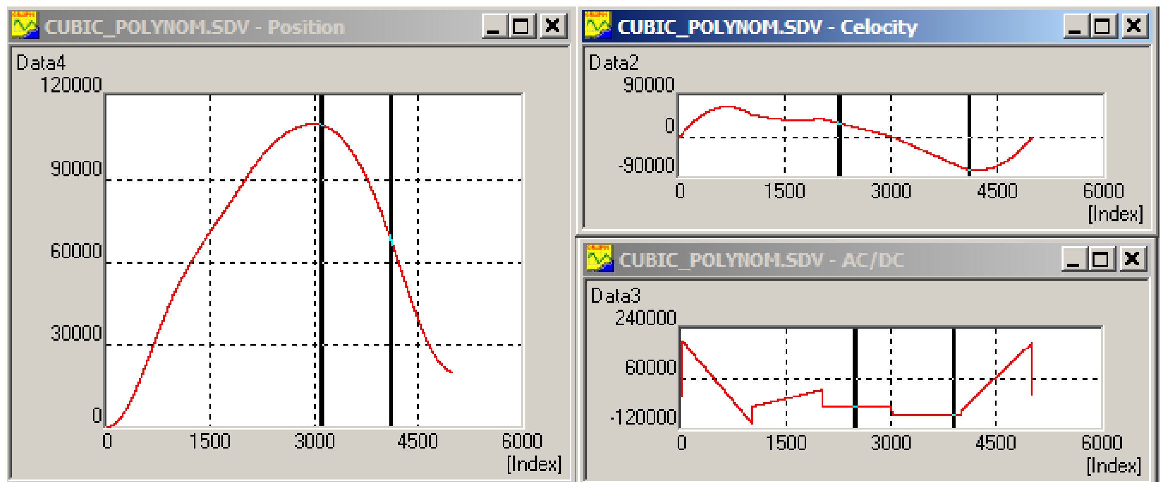


Figure 1-1 Cubic polynomial example

6.3.1.2 Quintic polynomial – Polynomial Order 5 (eQUINTIC_ON_CUBIC)

Quintic polynomial guarantees continuity by position, velocity and acceleration. Its drawback is discontinuity by jerk at every connection point (Figure 1-2).

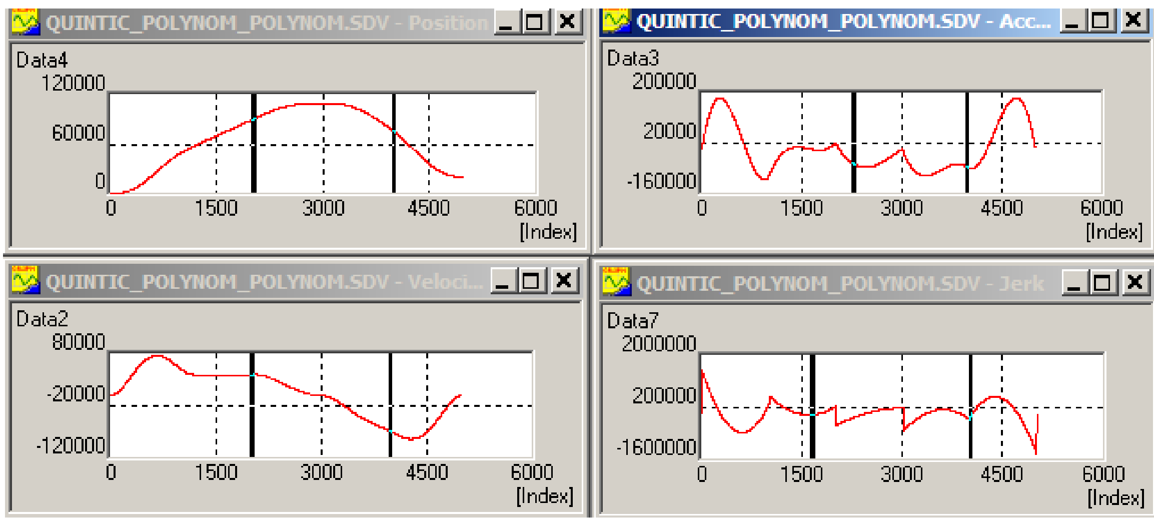


Figure 1-2 Quintic polynomial example

6.3.1.3 Septic polynomial - Polynomial Order 7 (eSEPTIC_ON_CUBIC)

Septic polynomial guarantees continuity by position, velocity, acceleration and jerk. Its disadvantage is a higher amplitude variation than we see for the lower degree polynomials. It is the most universal interpolation mode that can be recommended in most cases. Example of the septic interpolation is presented below (Figure 1-3).

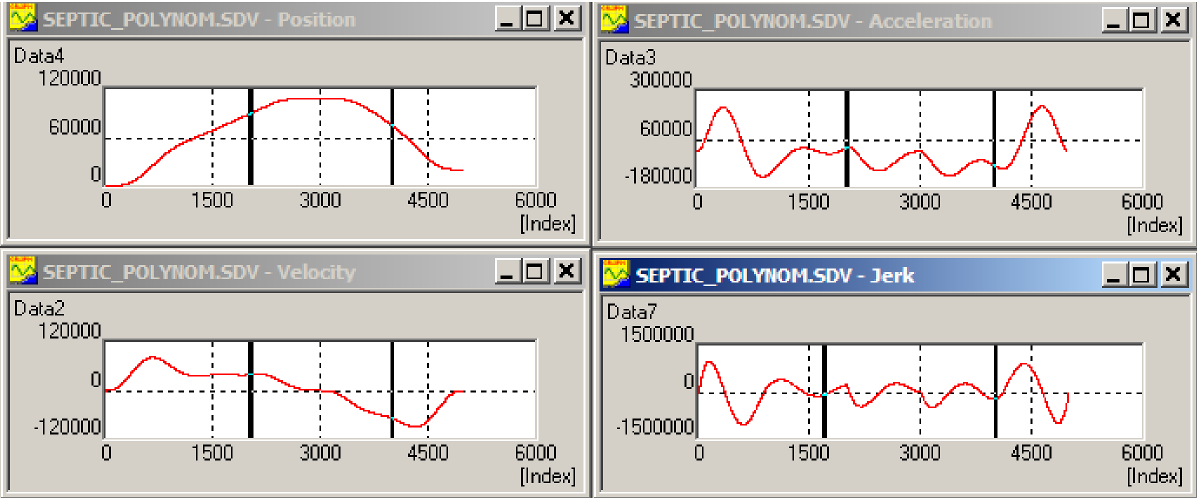


Figure 1-3 Septic polynomial example

6.3.2 Sinusoidal interpolation functions

A PVT profiler also supports a number of sinusoidal interpolation options.

6.3.2.1 Triangle sinusoidal acceleration (eCYCLOID_VELOCITY_MODIFIED1)

Triangle sinusoidal acceleration - modified triangle acceleration with AC(t) and DC(t) increasing by the sinusoid to some maximum value and then decreasing to zero. It guarantees continuity by position, velocity, acceleration and jerk. Result of execution for the same PVT table below: in Figure 1-4 – position and velocity, on Figure 1-5 – acceleration and jerk.

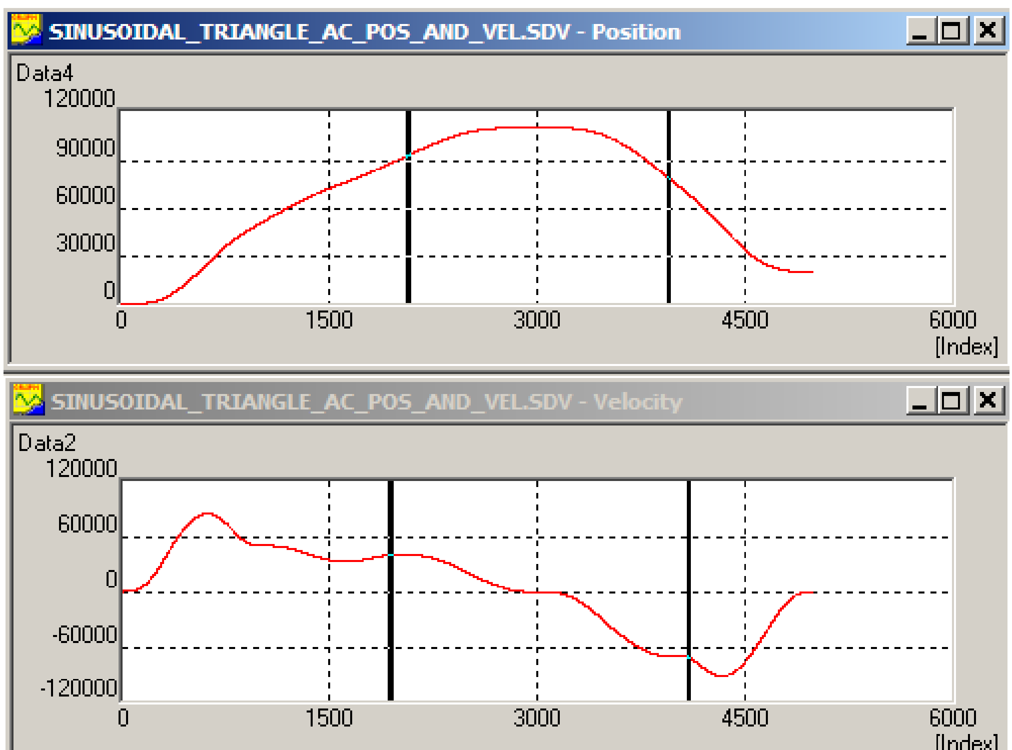


Figure 1-4 Triangle sinusoidal acceleration - position and velocity

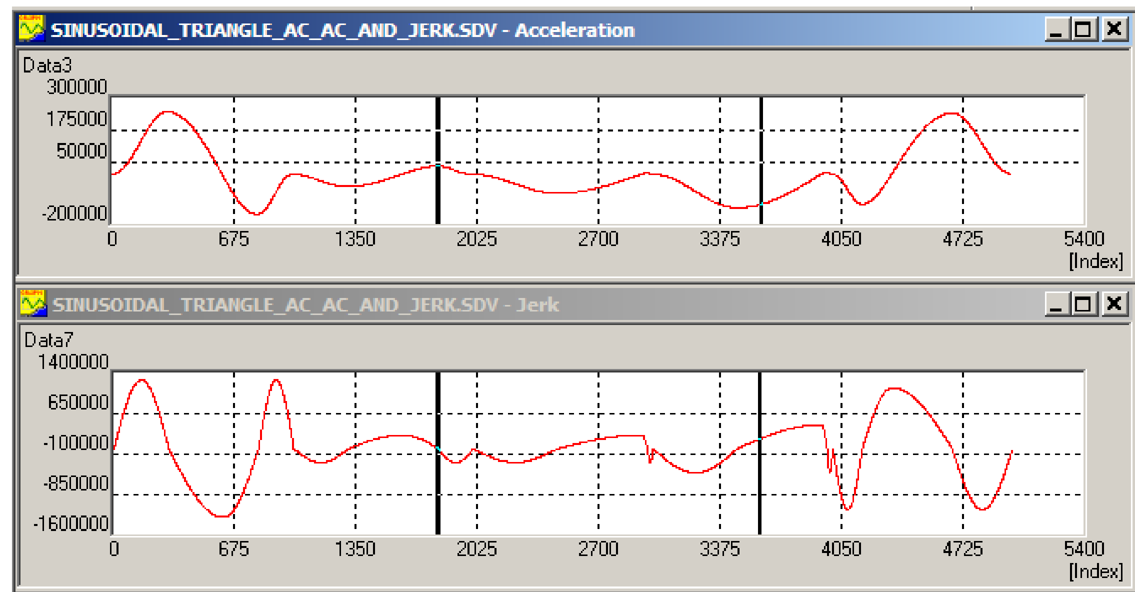


Figure 1-5 Triangle sinusoidal acceleration - acceleration and jerk

6.3.2.2 Trapezoidal sinusoidal acceleration (eCYCLOID_VELOCITY_MODIFIED2)

Trapezoidal sinusoidal acceleration or modified trapezoidal acceleration usually consists of two parts:

- A movement with acceleration increasing by the sinusoid from zero to AC_{max}, movement with AC_{max} (parabolic position, linear velocity profiles) and acceleration decreasing by the sinusoid to zero.
- A movement with deceleration decreasing by the sinusoid from zero to $-AC_{max}$, movement with $-AC_{max}$ (parabolic position profile, linear velocity profiles) and deceleration increasing by the sinusoid to zero.

It guarantees continuity by position, velocity, acceleration and jerk (if all the segments defined with this interpolation mode). An example can be seen in Figure 1-6 (position and velocity) and Figure 1-7 (acceleration and jerk).

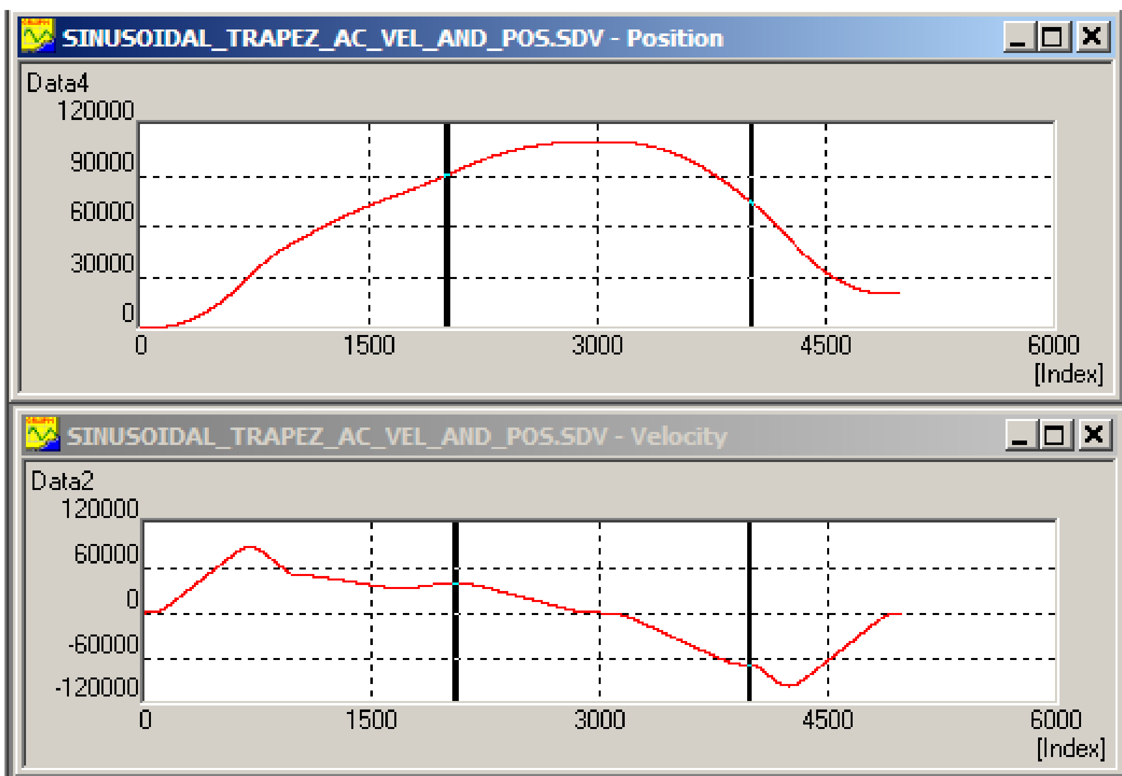


Figure 1-6 Trapezoidal sinusoidal acceleration - position and velocity

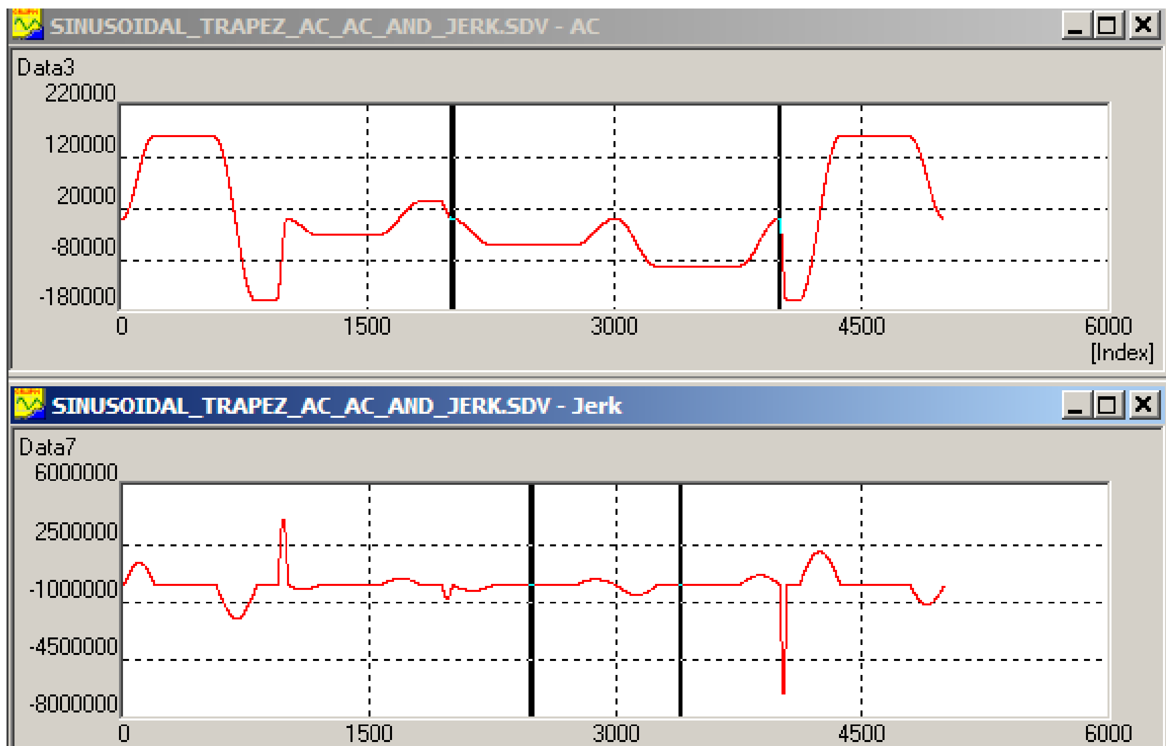


Figure 1-7 Trapezoidal sinusoidal acceleration - acceleration and jerk

6.3.2.3 Sinusoidal velocity (eCYCLOID_POSITION)

This interpolation mode produces cycloidal position and sinusoidal velocity interpolation. Sinusoidal velocity is an advantage of this interpolation type, but it has two significant drawbacks. The first is that it starts and ends with a maximum jerk. The second one is that in the general case with $\dot{Y}_i \neq 0$ and $\dot{Y}_{i+1} \neq 0$ these start and end derivatives must obey a requirement $\Delta Y_i = 0.5(\dot{Y}_i + \dot{Y}_{i+1})\Delta X_i$. For this reason, this interpolation mode can be recommended mainly for the ramp segments with $\dot{Y}_i \neq 0$ (ramp up) or $\dot{Y}_{i+1} = 0$ (ramp down).

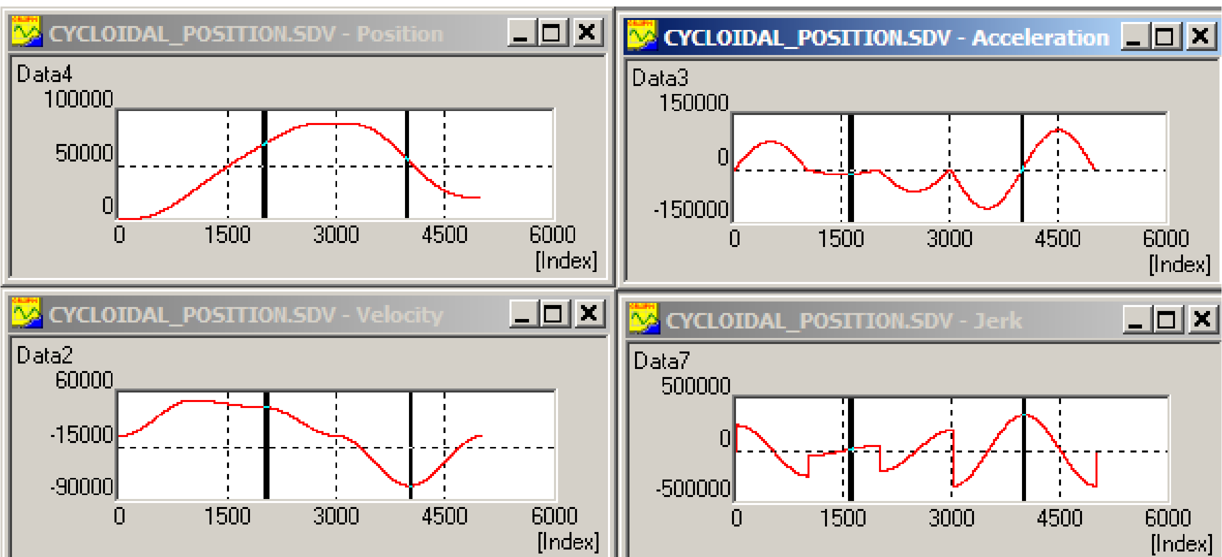


Figure 1-8 Sinusoidal velocity example

Maestro also supports two interpolation modes with the sinusoidal acceleration and cycloidal velocity profiles.

6.4 Data loading

Each row of input data stored as a table in shared memory, represents the time, position, velocity and vector position. For example, if we have an m-axes setup, the nth row in the table will be generated according to the following format:

$T_n \quad P_{n1} \; V_{n1} \; P_{n2} \; V_{n2} \; \dots \; P_{nm} \; V_{nm}$

As shown, T is similar for all axes, and therefore can be used only once.

The user can provide the data in several ways:

- Provide a file with a table containing the data points
- Provide an array containing data points, which can also be appended to an existing path within a given index

When the user decides to append points to an existing path, he has to be aware that:

- The table size is limited, and its maximum size should be known
- For dynamic Append only, when the number of points exceeds the table upper limit, the points will be appended to the beginning of the table.
- It is not possible to append to the current segment (presently, if the current index is X, the user can only append from X + 3).

During data loading, the points are read from the file. However, unlike splines, they are inserted into shared memory without pre-calculations.

Note: **Appending data to a table loaded from file is forbidden.**
If T equals the cycle time, a simpler profiler code should be applied.

6.5 PVT motion

The motion is performed using a special type of function block. When this function block type is inserted, a special profiler is applied that calculates polynomial coefficients and moves along the calculated trajectory.

Every cycle the next segment is calculated forward. The function block includes a pointer to the current selected path data start, and includes all path-constant data, i.e. dimension, number of points, etc. In addition, the function block contains the current working index, so that if the user wishes to append points, the system will avoid appending the points at the current working index. When the PVT function block is in motion, no other function block can be inserted in any buffer mode, only via the STOP command (similar to spline behavior).

6.6 On-The-Fly Mode

The Maestro also supports loading data to the Maestro on-the-fly, i.e. the user can start the motion based on existing data, and the remaining data can be appended later. This concept allows the user to change the path on-the-fly, taking the software constraints into consideration. At this moment, there are two appending sub-modes implemented; automatic and manual.

In automatic mode, the Maestro will remember the last index where data was appended, and will append to this index next time.

In the manual mode, the user will have to provide the index, where he wishes to append points.

The number of points calculated is used in the real-time module to check that the end of path has not been reached, i.e. if 2000 points is allocated, and the user inserts just 1000 points, the 1000 points limit should not be traversed. In addition, in dynamic mode only, an underflow event can be sent when the index is close to the end of path.

6.6.1 Initializing Table

Before appending points to a table, the user should initialize it. This can be achieved in two ways; by loading a table from file, or invoke a dedicated function, which will initialize the "constant" path parameters, like dimensions, maximal number of points, etc.

In addition, the user should select whether the appending is to be done statically or dynamically. If dynamic appending is selected, the user should choose an underflow threshold.

6.6.2 Loading Data

There are two modes of load data from array – Static and Dynamic. Static mode is pretty similar to load from file mode, the table is loaded and no appending is allowed, when the table is filled with maximum points. Dynamic mode is cyclic by default, and real-time events will be generated upon crossing the underflow threshold. Each time the number of inserted points drops below the predefined threshold – an event is generated to the user space. Using the existing events mechanism – the user handles the overflow as necessary.

The user has control of two parameters during appending data to table; Whether appending in automatic mode? If no, then the second parameter is the index to append from. Based on these two parameters, the algorithm inserts the data safely.

If the axis/vector is in motion, an index delta is applied to maintain the calculated path (index delta = 3). If the difference between the current index and the start index to append is less than the delta, insertion is forbidden.

In cyclic mode the data is appended to the given index (automatically or manually), and when the data reaches the end of PVT segment, it is automatically appended to the beginning. In non-cyclic mode, where the data reaches the end of PVT segment, an error is returned.

6.6.3 Cyclic Mode

In order to support cyclic mode, a cyclic buffer is managed. The main constraint is that the size of the buffer appended must not exceed the table size, as then the end will run over the beginning. In addition, if the appending index is after the current index, the user should maintain a minimal delta:

Abs(current – start) < 3

If the index of appending is prior to the current index, the following should be maintained:

start + block size < current

6.7 File example

The points are read as doubles. In practice each row will take

((M*2 + 1) * size of(double)) bytes where the 1 stands for time column.

The total memory "allocated" will be (N * (M*2 + 1) * size of(double)).

```
PV mode:    X
Dimension:  M
Number of points: N
Time Absolute:  1/0
Position Absolute: 1/0
Cyclic: 1/0

### Data start ###
T1   P11 V11 P12 V12 … P1m V1m
T2   P21 V21 P22 V22 … P2m V2m
.
..
Tn   Pn1 Vn1 Pn2 Vn2 … Pnm Vnm
### Data end ###
```

6.8 Table Functions

The following definitions are maximum values for the relevant parameter of the Table functions.

MMC_MAX_JOURNAL_ENTRIES 20

The max length for a table file name (exluded from path) is:

MMC_TABLE_FILE_LENGTH 20

The following Table functions are generic and are applicable to PVT, PT (static and dynamic spline), and ECAM.

PVT functions
MMC_GetTableList
MMC_GetTableInfo

6.8.1 MMC_TABLE_LIST_OUT

O_TABLE_LIST Structure

```
typedef struct o_table_list {
unsigned int _uiHandlers[MMC_MAX_JOURNAL_ENTRIES];
int _iNumber;
unsigned short usStatus;
shortsErrorID;
} MMC_TABLE_LIST_OUT;
```

Description Output table list. This structure is not a function block.

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Parameters

_uiHandlers [MMC_MAX_JOURNAL_ENTRIES]

Defines the list of table handlers with a maximum limited to the number of MMC_MAX_JOURNAL_ENTRIES, i.e. 20.

_iNumber

Defines the number of tables found. Positive integer value.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

6.8.2 MMC_TABLE_LIST_IN

I_TABLE_LIST Structure

```
typedef struct i_table_list {
NC_MOTION_TABLE_TYPE_ENUM eTableType;
} MMC_TABLE_LIST_IN;
```

Description Input table list. This structure is not a function block.

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Parameters

NC_MOTION_TABLE_TYPE_ENUM eTableType

The Table type defined according to the enumerator parameter NC_MOTION_TABLE_TYPE_ENUM defined by:

eNC_TABLE_NONE	= 0
eNC_TABLE_SPLINE	= 1
eNC_TABLE_PVT_FILE	= 2
eNC_TABLE_PVT_ARRAY	= 3
eNC_TABLE_PVT_FILE_QUINTIC_CUB	= 4
eNC_TABLE_PVT_ARRAY_QUINTIC_CUB	= 5
eNC_TABLE_ECAM_FILE	= 6
eNC_TABLE_ECAM_ARRAY	= 7
eNC_TABLE_OLSPLN_FILE	= 8
eNC_TABLE_OLSPLN_ARRAY	= 9
eNC_TABLE_MAX	= 10

This enumeration is used as the input for these functions, to distinguish between ECAM and PVT. Currently only eNC_TABLE_PVT_FILE and eNC_TABLE_PVT_ARRAY can be applied.

6.8.3 MMC_TABLE_DATA_OUT

O_TABLE_DATA Structure

```
typedef struct O_TABLE_DATA {
char _name[MMC_TABLE_FILE_LENGTH+1];
unsigned short usStatus;
shortsErrorID;
} MMC_TABLE_DATA_OUT;
```

Description Output table data. This structure is not a function block.

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Parameters

_name[MMC_TABLE_FILE_LENGTH+1]

Defines the name of the table with a max length for the table file name (exluded from path) of 20+1 i.e. MMC_TABLE_FILE_LENGTH+1

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

6.8.4 MMC_TABLE_DATA_IN

I_TABLE_LIST_DATA Structure

```
typedef struct I_TABLE_LIST_DATA {
unsigned int uiHandler;
} MMC_TABLE_DATA_IN;
```

Description Input table data. This structure is not a function block.

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Parameters

uiHandle

Table access handler.

6.8.5 MMC_GetTableList

This function provides a list of tables for a given table type.

```
MMC_LIB_API int MMC_GetTableList(
IN MMC_CONNECT_HNDL hConn,
IN MMC_TABLE_LIST_IN* pInParam,
OUT MMC_TABLE_LIST_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_TABLE_LIST_IN** input data structure using the MMC_GetTableList.

pOutParam

Points to the **MMC_TABLE_LIST_OUT** output structure receiving information, as a result of calling the MMC_GetTableList.

Remarks

Scope

All

MMC_TABLE_LIST_IN Structure

typedef struct MMC_TABLE_LIST_IN {
} MMC_TABLE_LIST_IN;

Parameters

All MMC_TABLE_LIST_IN parameters

Refer to the MMC_TABLE_LIST_IN Structure described in section **6.7.2 MMC_TABLE_LIST_IN**

MMC_TABLE_LIST_OUT Structure

typedef struct MMC_TABLE_LIST_OUT {
} MMC_TABLE_LIST_OUT;

Parameters

All MMC_TABLE_LIST_OUT parameters

Refer to the MMC_TABLE_LIST_OUT Structure described in section **6.7.1 MMC_TABLE_LIST_OUT**

Figure 6-1 describes the function block for MMC_GetTableList as applied within the IEC 61131 programming.

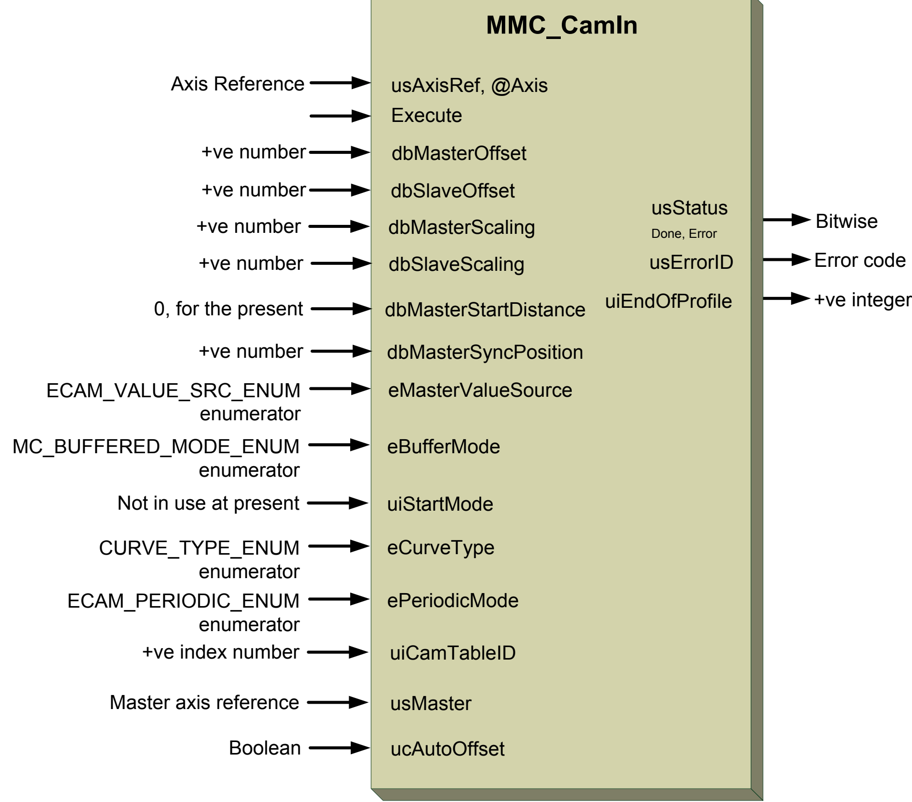


Figure 6-9: MMC_GetTableList function

6.8.5.2 Function Code Example

To be provided.

6.8.6 MMC_GetTableInfo

This function provides the table info (currently name only) for a given table handler.

```
MMC_LIB_API int MMC_GetTableInfo(
IN MMC_CONNECT_HNDL hConn,
IN MMC_TABLE_DATA_IN* pInParam,
OUT MMC_TABLE_DATA_OUT* pOutParam
);
```

Motion Mode	NC – Supported	Distributed – Not supported
Source	GMAS\includes\MMC_PVT_ECAM_API.h GMAS Programming(IEC 61331 Program)\ElmoGenAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_TABLE_DATA_IN** input data structure using the MMC_GetTableInfo.

pOutParam

Points to the **MMC_TABLE_DATA_OUT** output structure receiving information, as a result of calling the MMC_GetTableInfo.

Remarks

Scope

All

MMC_TABLE_DATA_IN Structure

typedef struct MMC_TABLE_DATA_IN {
} MMC_TABLE_DATA_IN;

Parameters

All MMC_TABLE_DATA_IN parameters

Refer to the MMC_TABLE_DATA_IN Structure described in section **6.7.4 MMC_TABLE_DATA_IN**

MMC_TABLE_DATA_OUT Structure

typedef struct MMC_TABLE_DATA_OUT {
} MMC_TABLE_DATA_OUT;

Parameters

All MMC_TABLE_DATA_OUT parameters

Refer to the MMC_TABLE_DATA_OUT Structure described in section **6.7.3 MMC_TABLE_DATA_OUT**

Note: The length of file's name refers only to file name and not the length of the whole path. If file name is longer than twenty only the first twenty characters shall be returned.

Figure 6-2 describes the function block for MMC_GetTableInfo as applied within the IEC 61131 programming.

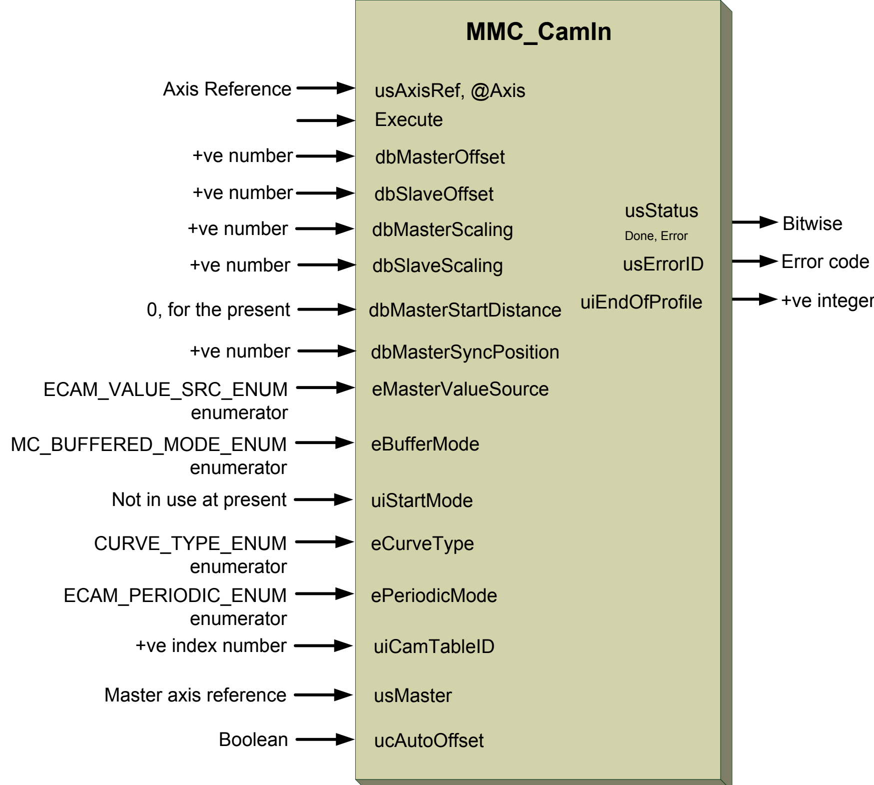


Figure 6-10: MMC_GetTableInfo function

6.8.6.2 Function Code Example

To be provided.

6.9 PVT Functions

The following PVT functions are described:

PVT functions
MMC_InitTable
MMC_LoadTableFromFile
MMC_AppendPointsToTable
MMC_UnloadTable
MMC_MoveTable
MMC_GetTableIndex

6.9.1 MMC_InitTable

This function allocates a memory segment in the shared memory according to the dimension and number of points.

```
int MMC_InitTableCmd(
MMC_CONNECT_HNDL hConn,
MMC_INITTABLE_IN* pInParam,
MMC_INITTABLE_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_INITTABLE_IN** input data structure using the MMC_InitTableCmd.

pOutParam

Points to the **MMC_INITTABLE_OUT** output structure receiving information, as a result of calling the MMC_InitTableCmd.

Remarks

It should be noted that when using this function, no data is loaded. The function returns hMemHandle, which is the unique path ID (similar to splines).

Scope

All

MMC_INITTABLE_IN Structure

```
typedef struct mmc_inittable_in{
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
NC_MOTION_TABLE_TYPE_ENUM eTableType;
MC_COORD_SYSTEM_ENUM eCoordSystem;
unsigned long ulMaxNumberOfPoints;
unsigned long ulUnderflowThreshold;
unsigned short usAxisRef;
unsigned short usDimension;
unsigned char ucIsDynamicMode;
unsigned char ucIsPosAbsolute;
unsigned char ucIsCyclic;
unsigned char ucSuperimposed;
unsigned char ucExecute;
} MMC_INITTABLE_IN
```

Parameters

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.11 Multiple Axes Motion Control - Transition and Buffer Modes**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,
MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time
MC_TM_DEFINED_VELOCITY_MODE = 2,
MC_TM_CORNER_DISTANCE_MODE = 3,
MC_TM_MAX_CORNER_DEVIATION_MODE = 4,
MC_TM_SWITCH_RADIUS_MODE = 5,
MC_TM_CORNER_DIST_TC_POLYNOM = 6,
MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,
MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,
MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

eBufferMode

MC_BufferMode defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting — Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered — The next function block affects the axis as soon as the previous movement is completed.

BlendingLow — The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).

BlendingPrevious — Blending with the velocity of function block 1 at the end-position of this block

BlendingNext — Blending with the velocity of function block 2 at end-position of function block1

BlendingHigh — Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

NC_MOTION_TABLE_TYPE_ENUM eTableType

The Table type defined according to the enumerator parameter NC_MOTION_TABLE_TYPE_ENUM defined by:

eNC_TABLE_NONE	= 0	
eNC_TABLE_SPLINE	= 1	
eNC_TABLE_PVT_FILE	= 2	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_ARRAY	= 3	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_FILE_QUINTIC_CUB	= 4	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_PVT_ARRAY_QUINTIC_CUB	= 5	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_ECAM_FILE	= 6	
eNC_TABLE_ECAM_ARRAY	= 7	
eNC_TABLE_OLSPLN_FILE	= 8	
eNC_TABLE_OLSPLN_ARRAY	= 9	
eNC_TABLE_MAX	= 10	

This enumeration is used as the input for these functions, to distinguish between ECAM and PVT. Currently only eNC_TABLE_PVT_FILE and eNC_TABLE_PVT_ARRAY can be applied.

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

ulMaxNumberOfPoints

The maximum number of points allocated. Theoretical maximum value of $4.294E^{+9}$, however in practice will be limited well below this value.

ulUnderflowThreshold

The underflow limit. An event will be generated if the number of points between the current index, and the end index falls below this value. Value cannot be greater that the ulMaxNumberOfPoints value. Any +ve values accepted.

usAxisRef

Axis/group reference handle type returned by GetAxisRef command.

usDimension

The dimensions of the table. A +ve number

ucIsDynamicMode

Boolean result to question, Is the mode Dynamic or not? 0, or 1

ucIsPosAbsolute

Boolean result to question, Is the position Absolute or not? 0, or 1.

This parameters is for future use.

ucIsCyclic

Boolean result to question, Is the mode Cyclic or not? 0, or 1

MMC_INITTABLE_OUT Structure

```
typedef struct mmc_inittable_out{
MC_PATH_REF hMemHandle;
unsigned short usStatus;
short sErrorID;
} MMC_INITTABLE_OUT;
```

Parameters

hMemHandle

MC_PATH_REF alias for unsigned integer with values and handle to a journal entry where the pointer to the shared memory is located, and indicates the memory area where the spline is allocated. In fact this is the unique identifier between PathSelect, MovePath and PathUnselect commands. MC_PATH_REF is the journal entry path reference.

hMemHandle produces +ve Integer values.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 6-3 describes the function for MMC_InitTable as applied within the IEC 61131 programming.

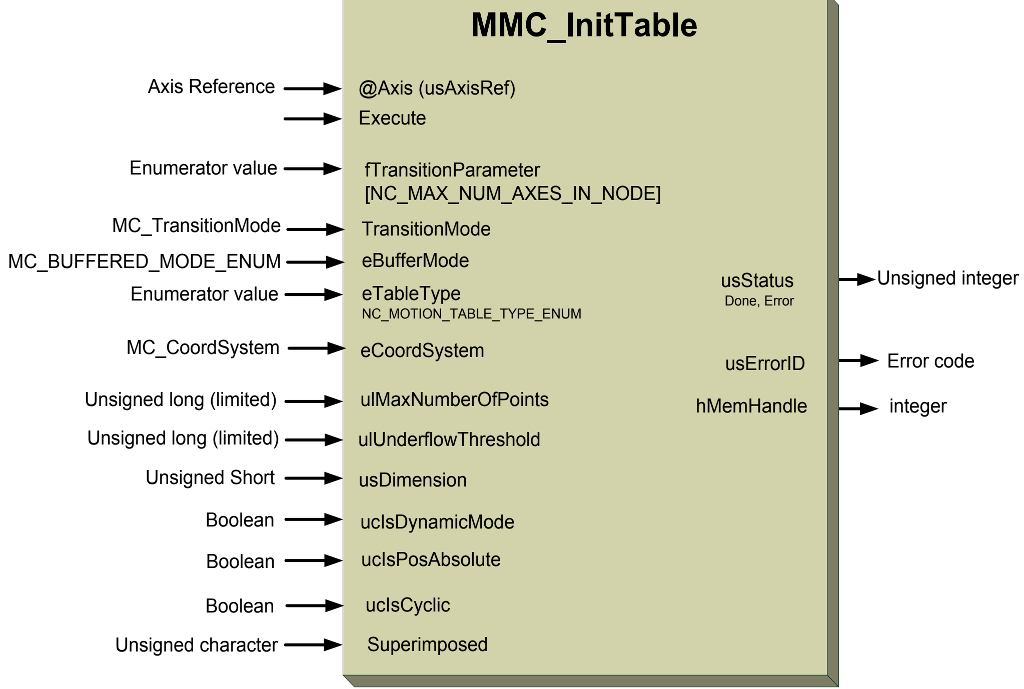


Figure 6-11: MMC_InitTable function

6.9.1.2 Function Code Example

Refer to the example in section 6.8.7.2

6.9.2 MMC_InitTableEx

This function allocates unlimted memory segments in the shared memory according to the dimension and number of points.

```
MMC_LIB_API int MMC_InitTableExCmd(
MMC_CONNECT_HNDL hConn,
MMC_INITTABLEEX_IN* pInParam,
MMC_INITTABLEEX_OUT* pOutParam);
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_INITTABLEEX_IN** input data structure using the MMC_InitTableExCmd.

pOutParam

Points to the **MMC_INITTABLEEX_OUT** output structure receiving information, as a result of calling the MMC_InitTableExCmd.

Remarks

It should be noted that when using this function, no data is loaded. The function returns hMemHandle, which is the unique path ID (similar to splines).

Scope

All

MMC_INITTABLEEX_IN Structure

```
typedef struct mmc_inittableex_in {
double dbConstVelocity;
double dbConstTime;
MC_BUFFERED_MODE_ENUM eBufferMode;
NC_MOTION_TABLE_TYPE_ENUM eTableType;
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_ONLINE_SPLINE_MODE_ENUM eSplineMode;
unsigned long ulMaxNumberOfPoints;
unsigned long ulUnderflowThreshold;
unsigned short usAxisRef;
unsigned short usDimension;
unsigned char ucIsDynamicMode;
unsigned char ucIsPosAbsolute;
unsigned char ucIsCyclic;
unsigned char ucSuperimposed;
unsigned char ucExecute;
unsigned char ucSpare[35];
} MMC_INITTABLEEX_IN;
```

Parameters

dbConstVelocity

Set the velocity to be constant. 8 byte +ve value.

dbConstTime

Set the time to be constant. 8 byte +ve value.

MC_BUFFERED_MODE_ENUM eBufferMode

MC_BufferMode defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting — Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered — The next function block affects the axis as soon as the previous movement is completed.

BlendingLow — The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis

	will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

NC_MOTION_TABLE_TYPE_ENUM eTableType

The Table type defined according to the enumerator parameter NC_MOTION_TABLE_TYPE_ENUM defined by:

eNC_TABLE_NONE	= 0	
eNC_TABLE_SPLINE	= 1	
eNC_TABLE_PVT_FILE	= 2	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_ARRAY	= 3	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_FILE_QUINTIC_CUB	= 4	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_PVT_ARRAY_QUINTIC_CUB	= 5	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_ECAM_FILE	= 6	
eNC_TABLE_ECAM_ARRAY	= 7	
eNC_TABLE_OLSPLN_FILE	= 8	
eNC_TABLE_OLSPLN_ARRAY	= 9	
eNC_TABLE_MAX	= 10	

This enumeration is used as the input for these functions, to distinguish between ECAM and PVT. Currently only eNC_TABLE_PVT_FILE and eNC_TABLE_PVT_ARRAY can be applied.

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD	= 0
MC_ACS_COORD	= 1
MC_MCS_COORD	= 2
MC_PCS_COORD	= 3

NC_ONLINE_SPLINE_MODE_ENUM eSplineMode

Define the types of spline mode. The NC_ONLINE_SPLINE_MODE_ENUM enumerator options are:

MC_NONE_ONLINE_SPLINE_MODE	= 0	
MC_QUINTIC_ON_PARAB_FT_DWELL	= 1	//defines fixed time
MC_QUINTIC_ON_PARAB_VT_DWELL	= 2	//defines variable time
MC_QUINTIC_ON_PARAB_CV_DWELL	= 3	//defines constant velocity

ulMaxNumberOfPoints

The maximum number of points allocated. Theoretical maximum value of $4.294E^{+9}$, however in practice will be limited well below this value.

ulUnderflowThreshold

The underflow limit. An event will be generated if the number of points between the current index, and the end index falls below this value. Value cannot be greater that the ulMaxNumberOfPoints value. Any +ve values accepted.

usAxisRef

Axis/group reference handle type returned by GetAxisRef command.

usDimension

The dimensions of the table. A +ve number

ucIsDynamicMode

Boolean result to question, Is the mode Dynamic or not? 0, or 1

ucIsPosAbsolute

Boolean result to question, Is the position Absolute or not? 0, or 1.

This parameters is for future use.

ucIsCyclic

Boolean result to question, Is the mode Cyclic or not? 0, or 1

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE. Not currently in use.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

ucSpare[35]

Spare description for future use. +ve char value with maximum of 35 chars.

MMC_INITTABLEEX_OUT Structure

```
typedef struct mmc_inittableex_out {
MC_PATH_REF hMemHandle;
unsigned short usStatus;
short usErrorID;
} MMC_INITTABLEEX_OUT;
```

Parameters

hMemHandle

MC_PATH_REF alias for unsigned integer with values and handle to a journal entry where the pointer to the shared memory is located, and indicates the memory area where the spline is allocated. In fact this is the unique identifier between PathSelect, MovePath and PathUnselect commands. MC_PATH_REF is the journal entry path reference.

hMemHandle produces +ve Integer values.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 6-4 describes the function for MMC_InitTableEx as applied within the IEC 61131 programming.

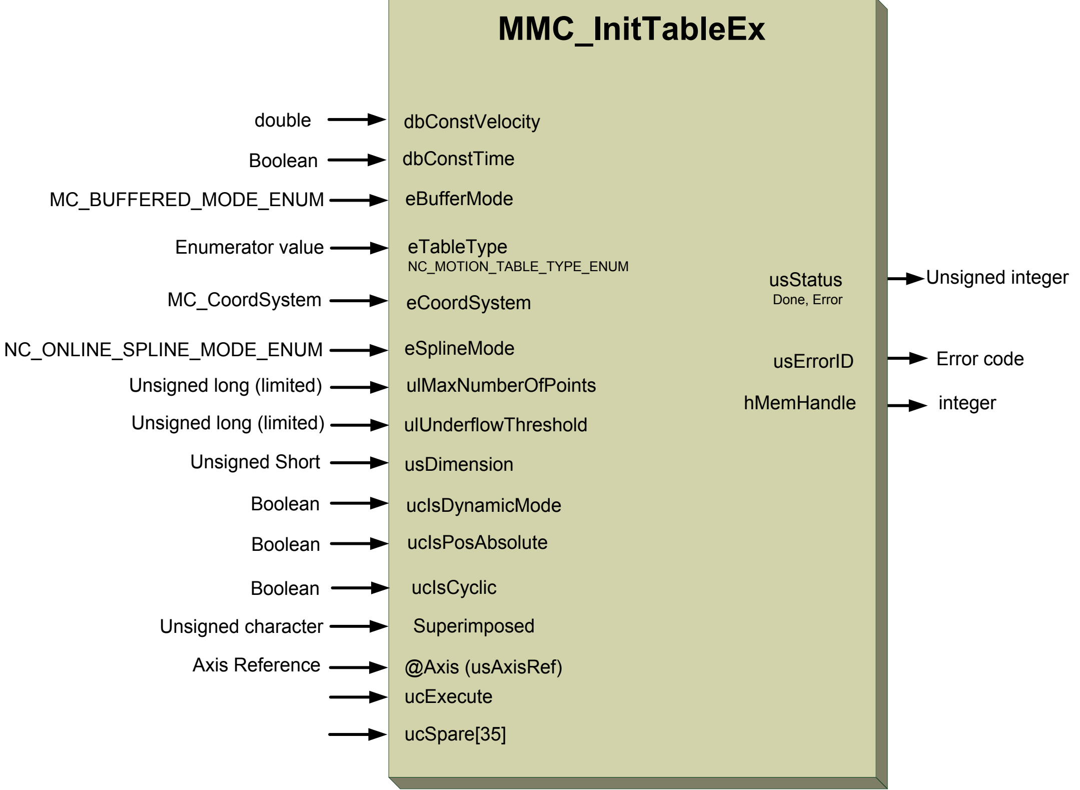


Figure 6-12: MMC_InitTableEx function

6.9.2.2 Function Code Example

Refer to the example in section 6.8.7.2

6.9.3 MMC_LoadTableFromFile

The function allocates a memory segment in the Maestro shared memory according to the dimension and number of points given in a file.

```
MMC_LIB_API int MMC_LoadTableFromFileCmd(
MMC_CONNECT_HNDL hConn,
MMC_LOADTABLEFROMFILE_IN* pInParam,
MMC_LOADTABLE_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_general_API.h
GMAS\includes\MMC_PVT_ECAM_API.h
GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_LOADTABLEFROMFILE_IN** input data structure using the MMC_LoadTableFromFileCmd.

pOutParam

Points to the **MMC_LOADTABLE_OUT** output structure receiving information, as a result of calling the MMC_LoadTableFromFileCmd.

Remarks

The file always contains data.

Scope

All

MMC_LOADTABLEFROMFILE_IN Structure

```
typedef struct mmc_loadtablefromfile_in{
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
NC_MOTION_TABLE_TYPE_ENUM eTableType;
MC_COORD_SYSTEM_ENUM eCoordSystem;
unsigned short usAxisRef;
MC_PATH_DATA_REF pPathToTableFile;
} MMC_LOADTABLEFROMFILE_IN;
```

Parameters

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE	= 0,
MC_TM_MAX_VELOCITY_MODE	= 1, Not supported at this time
MC_TM_DEFINED_VELOCITY_MODE	= 2,
MC_TM_CORNER_DISTANCE_MODE	= 3,
MC_TM_MAX_CORNER_DEVIATION_MODE	= 4,
MC_TM_SWITCH_RADIUS_MODE	= 5,
MC_TM_CORNER_DIST_TC_POLYNOM	= 6,
MC_TM_CORNER_DIST_CV_POLYNOM3	= 7,
MC_TM_CORNER_DIST_CV_POLYNOM5	= 8,
MC_TM_CORNER_DEVIATION_MODE_PLN6	= 9,
MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES	= 10,
MC_TM_LAST_MODE	

eBufferMode

MC_BufferMode defines the behavior of the axis. Enumerator modes are as follows:

MC_ABORTING_MODE	= 1

MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

eTableType

The Table type defined according to the enumerator parameter NC_MOTION_TABLE_TYPE_ENUM defined by:

eNC_TABLE_NONE	= 0	
eNC_TABLE_SPLINE	= 1	
eNC_TABLE_PVT_FILE	= 2	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_ARRAY	= 3	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_FILE_QUINTIC_CUB	= 4	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_PVT_ARRAY_QUINTIC_CUB	= 5	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_ECAM_FILE	= 6	
eNC_TABLE_ECAM_ARRAY	= 7	
eNC_TABLE_OLSPLN_FILE	= 8	
eNC_TABLE_OLSPLN_ARRAY	= 9	
eNC_TABLE_MAX	= 10	

This enumeration is used as the input for these functions, to distinguish between ECAM and PVT. Currently only eNC_TABLE_PVT_FILE and eNC_TABLE_PVT_ARRAY can be applied.

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

usAxisRef

Axis/group reference handle type returned by GetAxisRef command. Any +ve value.

pPathToTableFile

Unix path to the file that should be loaded

MC_PATH_DATA_REF — Where the enumerator MC_PATH_DATA_REF describes the I/O definition of the path data reference using the array [NC_PVT_ECAM_MAX_ARRAY_SIZE] that defines the maximum array size file path data.

MC_PATH_DATA_REF can have values of any characters.

NC_PVT_ECAM_MAX_ARRAY_SIZE is 170.

MMC_LOADTABLE_OUT Structure

```
typedef struct MMC_LOADTABLE_OUT {
unsigned short usStatus;
short sErrorID;
MC_PATH_REF hMemHandle;
} MMC_LOADTABLE_OUT;
```

Parameters

hMemHandle

MC_PATH_REF alias for unsigned integer with values and handle to a journal entry where the pointer to the shared memory is located, and indicates the memory area where the spline is allocated. In fact this is the unique identifier between PathSelect, MovePath and PathUnselect commands. MC_PATH_REF is the journal entry path reference.

hMemHandle produces +ve Integer values.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 6-5 describes the function for MMC_LoadTableFromFile as applied within the IEC 61131 programming.

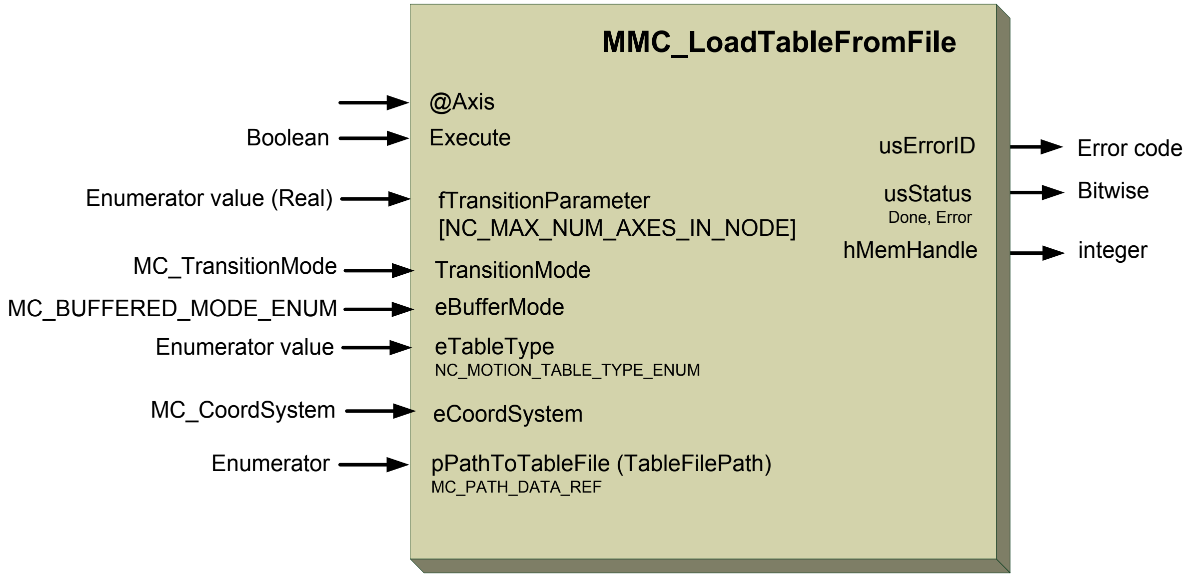


Figure 6-13: MMC_LoadTableFromFile function

6.9.3.2 Function Code Example

Refer to the example in section 6.8.7.2

6.9.4 MMC_UnloadTable

The function unloads a table from the Maestro and frees a memory segment in the Maestro shared memory according to the dimension and number of points given in a file.

```
MMC_LIB_API int MMC_UnloadTableCmd(
MMC_CONNECT_HNDL hConn,
MMC_UNLOADTABLE_IN* pInParam,
MMC_UNLOADTABLE_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_general_API.h

GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_UNLOADTABLE_IN** input data structure using the MMC_UnloadTableFromFileCmd.

pOutParam

Points to the **MMC_UNLOADTABLE_OUT** output structure receiving information, as a result of calling the MMC_UnloadTableFromFileCmd.

Remarks

The file always contains data.

Scope

All

MMC_UNLOADTABLE_IN Structure

```
typedef struct mmc_unloadtable_in{
MC_PATH_REF hMemHandle;
unsigned char ucExecute;
}MMC_UNLOADTABLE_IN
```

Parameters

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located, obtained from the PathSelect command. MC_PATH_REF is the journal entry path reference.

hMemHandle can have Integer values.

ucExecute

Start the execution command. Boolean TRUE/FALSE values. Currently not in use and set to 1.

MMC_UNLOADTABLE_OUT Structure

```
typedef struct mmc_unloadtable_out{
unsigned short usStatus;
short sErrorID;
}MMC_UNLOADTABLE_OUT
```

Parameters

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 6-5 describes the function for MMC_UnloadTable as applied within the IEC 61131 programming.

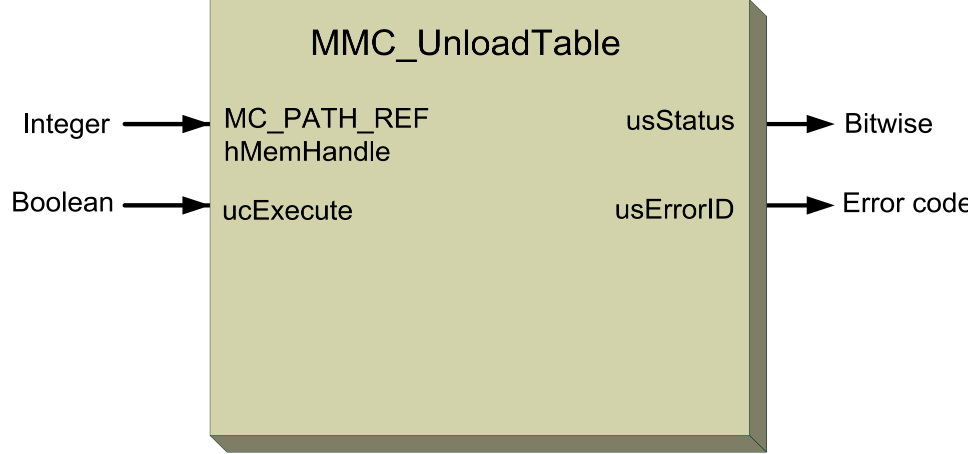


Figure 6-14: MMC_UnloadTable function

6.9.4.2 Function Code Example

Refer to the example in section 6.8.7.2

6.9.5 MMC_MoveTable

This function moves the Table along a selected path.

```
MMC_LIB_API int MMC_MoveTableCmd(
MMC_CONNECT_HNDL hConn,
MMC_AXIS_REF_HNDL hAxisRef,
MMC_MOVETABLE_IN* pInParam,
MMC_MOVETABLE_OUT* pOutParam
);
```

Motion Mode	NC – Supported	Distributed – Not supported
Source	GMAS\includes\MMC_general_API.h GMAS\includes\MMC_PVT_ECAM_API.h GMAS Programming(IEC 61331 Program)\ElmoGenAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_MOVETABLE_IN** input data structure using the MMC_MoveTableCmd.

pOutParam

Points to the **MMC_MOVETABLE_OUT** output structure receiving information, as a result of calling the MMC_MoveTableCmd.

Remarks

When applying to a single-axis PVT, the multi-axes values of the parameter MMC_MOVETABLE_OUT are not relevant, and therefore any value can be provided.

Scope

All

MMC_MOVETABLE_IN Structure

```
typedef struct MMC_MOVETABLE_IN{
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
MC_PATH_REF hMemHandle;
unsigned char ucSuperImposed;
unsigned char ucExecute;
} MMC_MOVETABLE_IN;
```

Parameters

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.9 Special Robot Transformations.**

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator value. The options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

NC_TRANSITION_MODE_ENUM eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control** - Transition and Buffer Modes and options below. The options are:

MC_TM_NONE_MODE = 0,

MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time

MC_TM_DEFINED_VELOCITY_MODE = 2,

MC_TM_CORNER_DISTANCE_MODE = 3,

MC_TM_MAX_CORNER_DEVIATION_MODE = 4,

MC_TM_SWITCH_RADIUS_MODE = 5,

MC_TM_CORNER_DIST_TC_POLYNOM = 6,

MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,

MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,

MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,

MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,

MC_TM_LAST_MODE

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows, but only the Buffered Mode is supported:

MC_ABORTING_MODE = 1

MC_BUFFERED_MODE = 2

MC_BLENDING_LOW_MODE = 3

MC_BLENDING_PREVIOUS_MODE = 4

MC_BLENDING_NEXT_MODE = 5

MC_BLENDING_HIGH_MODE = 6

Aborting Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered The next function block affects the axis as soon as the previous movement is completed.

BlendingLow The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).

BlendingPrevious Blending with the velocity of function block 1 at the end-position of this block

BlendingNext Blending with the velocity of function block 2 at end-position of function block1

BlendingHigh Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located obtained from the PathSelect command. MC_PATH_REF is the journal entry path reference.

hMemHandle has Integer values.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE. Not currently in use.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_MOVETABLE_OUT Structure

```
typedef struct MMC_MOVETABLE_OUT {
unsigned int uiHandle;
unsigned short usStatus;
short sErrorID;
} MMC_MOVETABLE_OUT;
```

Parameters

uiHandle

Handle to a journal entry where the pointer to the shared memory is located, and indicates the memory area where the spline is allocated. In fact this is the unique identifier between PathSelect, MovePath and PathUnselect commands. MC_PATH_REF is the journal entry path reference.

uiHandle produces +ve Integer values.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 6-7 describes the function for MMC_MoveTable as applied within the IEC 61131 programming.

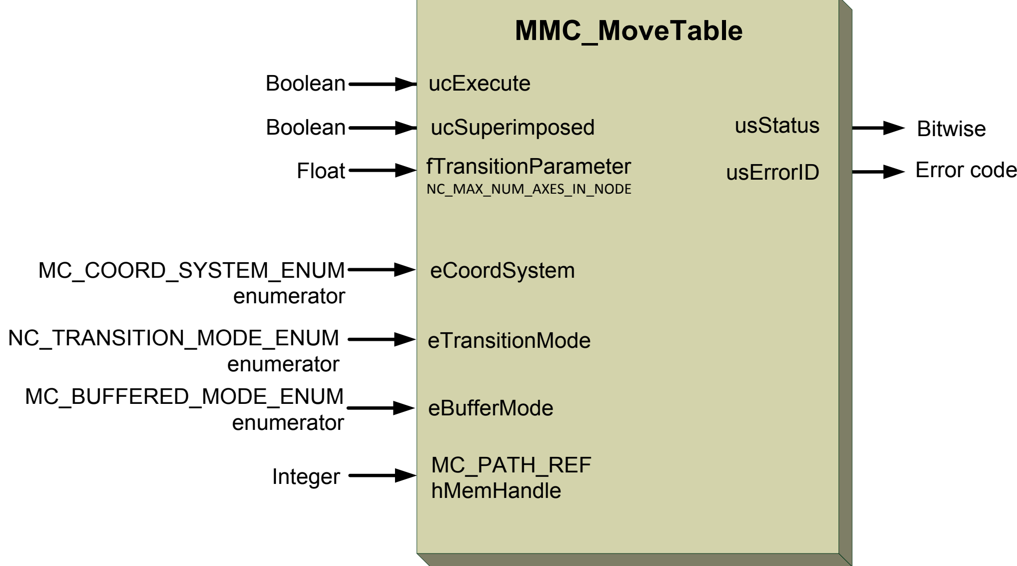


Figure 6-15: MMC_MoveTable function

6.9.5.2 Function Code Example

Refer to the example in section **6.8.7.2**

6.9.6 MMC_AppendPointsToTable

This function appends points to an existing table.

```
int MMC_AppendPointsToTableCmd(
MMC_CONNECT_HNDL hConn,
MMC_APPENDPOINTSTOTABLE_IN* pInParam,
MMC_APPENDPOINTSTOTABLE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_general_API.h

GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_APPENDPOINTSTOTABLE_IN** input data structure using the MMC_AppendPointsToTableCmd function.

pOutParam

Points to the **MMC_UNLOADTABLE_OUT** output structure receiving information, as a result of calling the MMC_AppendPointsToTableCmd function.

Remarks

None

Scope

All

MMC_APPENDPOINTSTOTABLE_IN Structure

```
typedef struct mmc_appendpointstotable_in{
double dTable[NC_PVT_ECAM_MAX_ARRAY_SIZE];
NC_MOTION_TABLE_TYPE_ENUM eTableType;
MC_PATH_REF hMemHandle;
unsigned long ulStartIndex;
unsigned long ulNumberOfPoints;
unsigned short usAxisRef;
unsigned char ucIsTimeAbsolute;
unsigned char ucIsAutoAppend;
} MMC_APPENDPOINTSTOTABLE_IN;
```

Parameters

dTable

Range of Points to be added to the Table, with 8 byte ± values accepted.

The array consists of a range of points in columns, where each three cells represents one point described by its Position, Velocity, and Time.

The maximum value of the NC_PVT_ECAM_MAX_ARRAY_SIZE array variable is 170. This is the maximum number of point allowed in the table.

eTableType

The Table type defined according to the enumerator parameter NC_MOTION_TABLE_TYPE_ENUM defined by:

```
eNC_TABLE_NONE                  = 0
eNC_TABLE_SPLINE                = 1
eNC_TABLE_PVT_FILE              = 2   //old method eCUBIC_POLYNOM
eNC_TABLE_PVT_ARRAY             = 3   //old method eCUBIC_POLYNOM
eNC_TABLE_PVT_FILE_QUINTIC_CUB  = 4   //new method eQUINTIC_ON_CUBIC
eNC_TABLE_PVT_ARRAY_QUINTIC_CUB = 5   //new method eQUINTIC_ON_CUBIC
eNC_TABLE_ECAM_FILE             = 6
eNC_TABLE_ECAM_ARRAY            = 7
eNC_TABLE_OLSPLN_FILE           = 8
eNC_TABLE_OLSPLN_ARRAY          = 9
eNC_TABLE_MAX                   = 10
```

This enumeration is used as the input for these functions, to distinguish between ECAM and PVT. Currently only eNC_TABLE_PVT_FILE and eNC_TABLE_PVT_ARRAY can be applied.

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located, obtained from the PathSelect command. MC_PATH_REF is the journal entry path reference.

hMemHandle can have Integer values.

ulStartIndex

Integer value of the index start. Any +ve value.

ulNumberOfPoints

Number of points to be appended to table. Any +ve number. The value cannot exceed MaxNumberOfPoints used in the function MMC_InitTable.

usAxisRef

Axis/group reference handle type returned by GetAxisRef command. Any +ve value.

ucIsTimeAbsolute

Boolean result to the question, Is the Time Absolute or not? Select the mode of the parameter:

0 - absolute mode
Every "time" input is used as absolute time when the current point will end the motion and reach the desired position.

1 - relative mode
Every "time" input is used as the time that will take to move from previous point to the end of current point.

ucIsAutoAppend

Boolean result to the question, Is the Append operation to be automatic or not? 0, or 1.

MMC_APPENDPOINTSTOTABLE_OUT Structure

```
typedef struct mmc_appendpointstotable_out{
unsigned short usStatus;
short sErrorID;
} MMC_APPENDPOINTSTOTABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 6-8 describes the function for MMC_AppendPointsToTable as applied within the IEC 61131 programming.

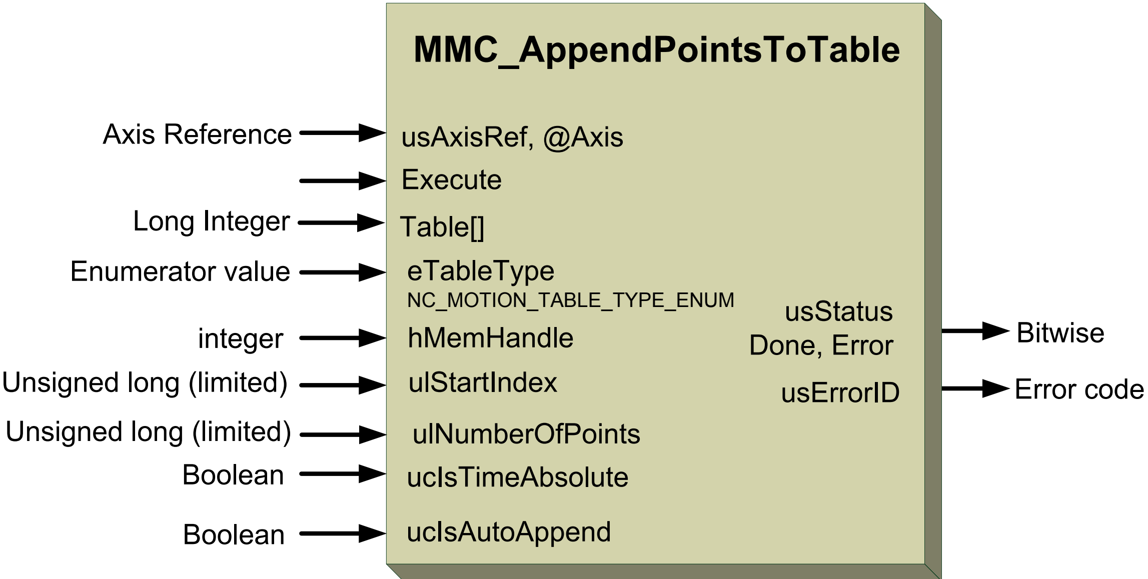


Figure 6-16: MMC_AppendPointsToTable function

6.9.6.2 Function Code Example

Refer to the example in section 6.8.7.2

6.9.7 MMC_GetTableIndex

This function obtains the PVT index.

int MMC_GetTableIndexCmd(
MMC_CONNECT_HNDL hConn,
MMC_GETTABLEINDEX_IN* pInParam,
MMC_GETTABLEINDEX_OUT* pOutParam
);

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_general_API.h
GMAS\includes\MMC_PVT_ECAM_API.h
GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_GETTABLEINDEX_IN** input data structure using the MMC_GetTableIndexCmd.

pOutParam

Points to the **MMC_GETTABLEINDEX_OUT** output structure receiving information, as a result of calling the MMC_GetTableIndexCmd.

Remarks

When applying to a single-axis PVT, the multi-axes values of the parameter MMC_MOVETABLE_OUT are not relevant, and therefore any value can be provided.

Scope

All

MMC_GETTABLEINDEX_IN Structure

```
typedef struct mmc_gettableindex_in{
init MC_PATH_REF hMemHandle;
} MMC_GETTABLEINDEX_IN;
```

Parameters

MC_PATH_REF hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle can have integer values.

MMC_GETTABLEINDEX_OUT Structure

```
typedef struct mmc_gettableindex_out{
unsigned long ulCurrentIndex;
unsigned short usStatus;
short sErrorID;
} MMC_GETTABLEINDEX_OUT;
```

Parameters

ulCurrentIndex

Obtain the current index. Any +ve number.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 6-9 describes the function block for MMC_GetTableIndex as applied within the IEC 61131 programming.

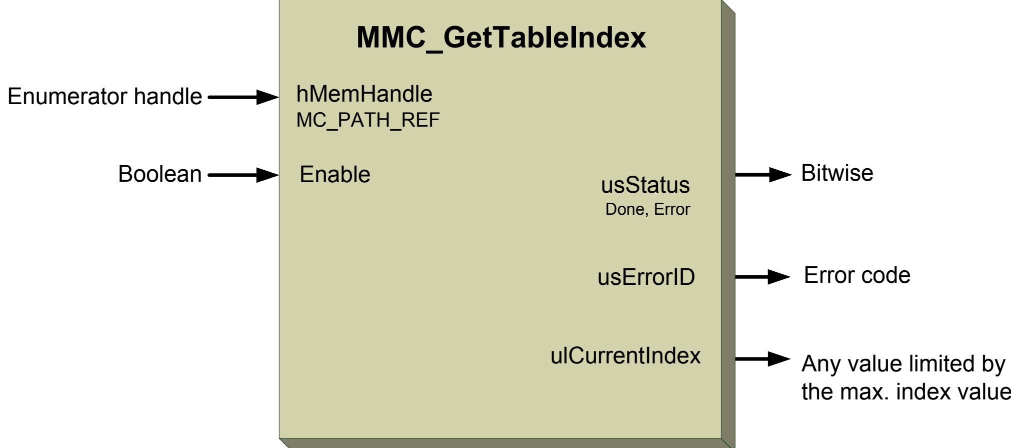


Figure 6-17: MMC_GetTableIndex function

6.9.7.2 Function Code Full Example

```
MMC_INITTABLE_IN      stInitTableIn;
MMC_INITTABLE_OUT     stInitTableOut;

MMC_LOADTABLEFROMFILE_IN stLoadTableFromFileIn;
MMC_LOADTABLE_OUT stLoadTableOut;

MMC_APPENDPOINTSTOTABLE_IN stAppendPointsIn;
MMC_APPENDPOINTSTOTABLE_OUT stAppendPointsOut;

MMC_GETTABLEINDEX_IN stGetTableIndexIn;
MMC_GETTABLEINDEX_OUT stGetTableIndexOut;

MMC_MOVETABLE_IN stMoveTableIn;
MMC_MOVETABLE_OUT stMoveTableOut;

stInitTableIn.eTableType = eNC_TABLE_PVT_ARRAY; //The table is defined as a PVT array
stInitTableIn.hAxisRef = aRef; //Group axes reference returned by the GatAxisRef command
stInitTableIn.ucIsCyclic = 1; //Mode is cyclic
stInitTableIn.ucIsPosAbsolute = 1; //Position is Absolute
stInitTableIn.ucIsTimeAbsolute = 0; //Time is not Absolute - This parameter is not available
for the function
stInitTableIn.usDimension = 2; //Dimensions of the table
stInitTableIn.usMaxNumberOfPoints = 50; //Maximum points allocated to the table

rc = MMC_InitTableCmd(hConn, &stInitTableIn, &stInitTableOut);
if (NC_OK != rc)
{
 HandleError();
}


stAppendPointsIn.eTableType = eNC_TABLE_PVT_ARRAY; //The table is defined as a PVT array
stAppendPointsIn.hMemHandle = stInitTableOut.hMemHandle; //Pointer to a journal entry where
the pointer is stored
stAppendPointsIn.usStartIndex = 0; //Value of the start index

FillTable(stLoadTableFromArrayIn.dTable);

rc = MMC_AppendPointsToTableCmd(hConn, &stAppendPointsIn, &stAppendPointsOut);
if (NC_OK != rc)
{
 HandleError();
}
```

```
stLoadTableFromFileIn.eTableType = eNC_TABLE_PVT_FILE; //The table is defined as a PVT file
stLoadTableFromFileIn.hAxisRef = aRef; //Axis reference returned by the GatAxisRef command
strcpy(stLoadTableFromFileIn.pPathToTableFile, pFileName); //Unix path to the file to be
loaded

rc = MMC_LoadTableFromFileCmd(hConn, &stLoadTableFromFileIn, &stLoadTableOut);
if (NC_OK != rc)
{
 HandleError();
}

stMoveTableIn.eBufferMode = MC_BUFFERED_MODE; //standard buffered mode supported
stMoveTableIn.eCoordSystem = MC_MCS_COORD; // MC_COORD_SYSTEM_ENUM enumerator value of MCS
coord system
stMoveTableIn.eTransitionMode = MC_TM_NONE_MODE; // only transition mode supported
stMoveTableIn.fTransitionParameter[0] = 0; // 0 Array value for specific transition
stMoveTableIn.hMemHandle = stLoadTableOut.hMemHandle; // MC_PATH_REF enum handle
stMoveTableIn.ucExecute = 1;  //
stMoveTableIn.ucSuperImposed = 0;  //

rc = MMC_MoveTableCmd(hConn, aRef, &stMoveTableIn, &stMoveTableOut);
if (NC_OK != rc)
{
 HandleError();
}

stGetTableIndexIn.hMemHandle = stInitTableOut.hMemHandle;

rc = MMC_GetTableIndexCmd(hConn, &stGetTableIndexIn, &stGetTableIndexOut);
if (NC_OK != rc)
{
 HandleError();
}
```

Chapter 7: Electronic CAM

7.1 Overview

Cam links a master to one or more slaves in a position / position mode (Figure 7-1). With motors and drives one can create the same position / position relationship, but in this case, via the so called CAM table listing the positions. In this way the relationship is converted to software and control.

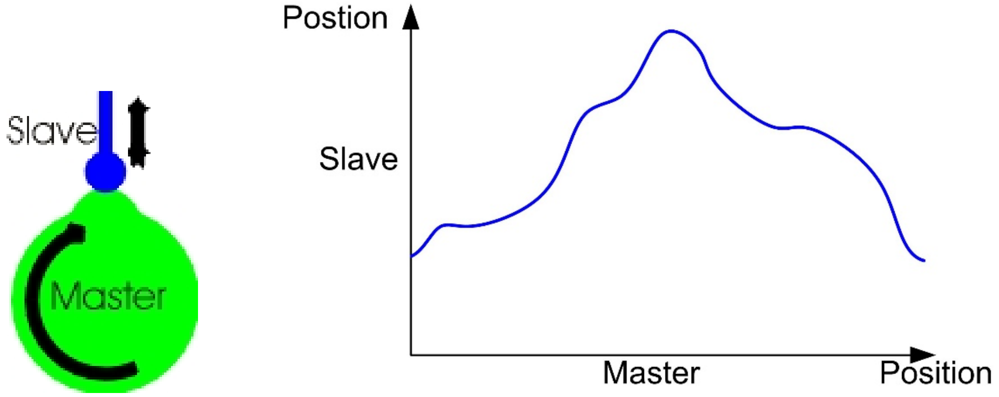


Figure 7-1: CAM profile illustration

Basically, one can differentiate between two types of Camming for both modulo and linear (or finite) master axes:

Periodic mode Repeats the execution of the CAM profile on a continuous basis, even if the CAM profile does not match modulo. This means that for modulo axis with a modulo of 360 degrees, where the CAM profile is specified for 90 degrees, will be executed 4 times in a modulo. In reverse mode the profile is executed the inverse way.

Non-periodic mode The CAM profile is run only once. If the master position is outside of the CAM profile, the slave axis stays in synchronized motion and maintains the last position. In reverse mode, the CAM profile is not executed after having reached the EndOfProfile position. The 90 degrees example above will only be run once.

Camming may be done with several combined CAM tables which are executed sequentially, like a ramp-in, a production cycle, and a ramp-out. Between the different CAM curves may be a gap (wait for trigger) in the execution. However, one could use the buffered mode or use the output *EndOfProfile* to start the next profile.

7.2 CAM Table

Camming is performed using a single table (two dimensional – describing master and slave positions together). The table should be strictly monotonic rising or falling, going both reverse and forward with the master. It is allowed and possible to change tables while CAM is running and to change elements in the table while the CAM is running.

The generation and filling of the CAM table (master, slave) is performed by an external tool, which is supplier specific. The coupling of the FB MC_CamIn to the table is also supplier-specific.

7.2.1 CAM Table File Format

CAM table follows the familiar convention of other tables like PVT, Spline etc… It consists of two parts, header and data. The header (see brown below) defines:

Version: optional

Table mode: ECAM file on this case

Dimension: number of slaves. Group is not supported at this phase therefore it must be one for now.

Number of points: number of rows within data section.

Cyclic: Not in use, but present for historical reasons.

Position absolute: Not in use, but present for historical reasons.

Master gap: fixed or varies

Note: Parameters 'Cycle' and 'Position absolute' in header are mandatory for historic reasons but the CAM loader does not care about them.

Each row is TAB delimited, ended with 'CRLF'. TAB character separates between text and numbers and between numbers in header (see red example below). Text must not contain TAB characters. TAB is also separates between columns in data part. The data part defines the the CAM table as lists of master/slave pairs. If master gap defined as fixed (*ECAM fixed gap 1*) then master column is missing. On that case master position relates to the MasterStartPosition, (set by MC_ CamTableSelect) and calculated by MC_CamIn FB for each segment at runtime. The third column refers to the curve type.

If the third column is missing then Polynomic 5 (1) is set as default value. Possible values for curve type in a file are:

ENUM	Value
eTableDefInterp	0
eLinearInterp	1
ePolynom5Interp	2
ePolynom7Interp	3
eCycloidPositionInterp	4
eCycloidVelocityModified1Interp	5
eCycloidVelocityModified2Interp	6

Below is a table example:

ECAM version	*1*	*0*	
ECAM mode	*46*		
ECAM dimension	*1*		
ECAM num of pts	*10*		
ECAM cyclic	*0*		
ECAM pos absolute	*1*		
ECAM fixed gap	*0*		
ECAM data start			
	0.0000000	0.000000	2
	20000.000000	35000.000000	
	50000.000000	60000.000000	
	75000.000000	50000.000000	
	120000.000000	80000.000000	
	144000.000000	110000.000000	
	165000.000000	122000.000000	
	185000.000000	145000.000000	
	200000.000000	160000.000000	
	220000.000000	185000.000000	3
ECAM data end			

Note: If ECAM fixed gap is 1, then first column, which is the master column, does not appear in CAM table.

7.2.2 CAM Table Loading

Cam loading is performed by the function MC_CamTableSelect. One may either load CAM table from file or use an array from user program for that matter. In any case a call to MC_CamTableSelect is mandatory. One cannot run MC_CamIn if MC_CamTableSelect was not previously called.

7.3 Interpolation Options (Curve Types)

The enumerator for the curve type consists of six optional interpolations. These are:

Description	Name	ENUM
Linear Interpolation	Linear	eLinearInterp
5th Order Polynomial	Polynom5	ePolynom5Interp
7th Order Polynomial	Polynom7	ePolynom7Interp
Cycloid Position Sinusoidal velocity	Cycloid Position	eCycloidPositionInterp
Cycloid Sinusoidal velocity Modified 1	Cycloid Velocity 1	eCycloidVelocityModified1Interp
Cycloid Sinusoidal velocity Modified 2	Cycloid Velocity 2	eCycloidVelocityModified2Interp

7.3.1 Polynomial interpolation functions

7.3.1.1 Linear Interpolation - Linear (eLinearInterp)

The linear interpolation produces continuity by the position and an exact correspondence between the master and slave by the simple formula:

$$Y(x) = x(\Delta Y/\Delta X)$$

where $Y(x)$ – slave position, X – master position, ΔY – slave increment for the whole segment, ΔX – slave increment for the whole segment.

Its drawback is a discontinuity by the velocity and higher order derivatives (acceleration and jerk) at every connection point between two linear segments.

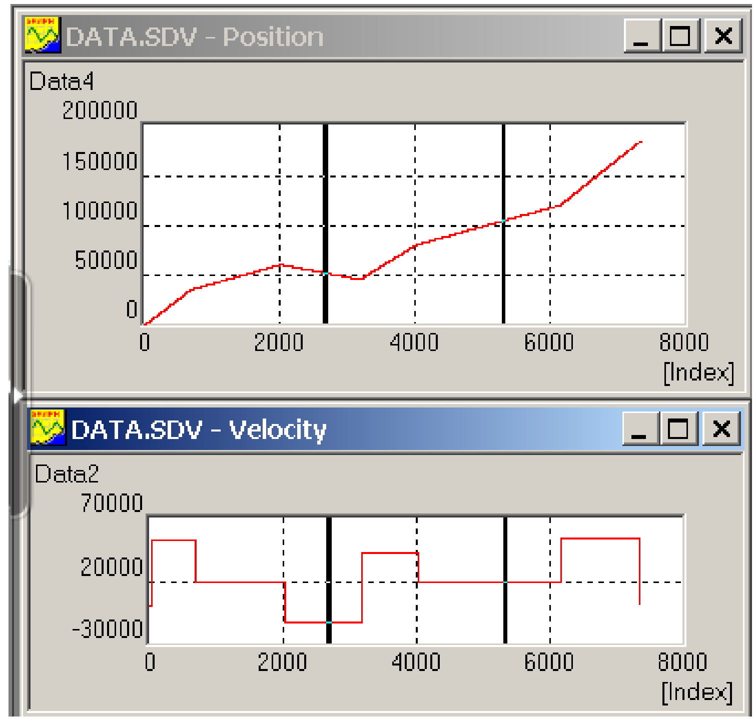


Figure 7-2: Linear interpolation Example

7.3.1.2 5th Order Polynomial – Polynomic 5 (ePolynom5Interp)

Quintic polynomial guarantees continuity by position, velocity and acceleration. Its drawback is discontinuity by jerk at every connection point.

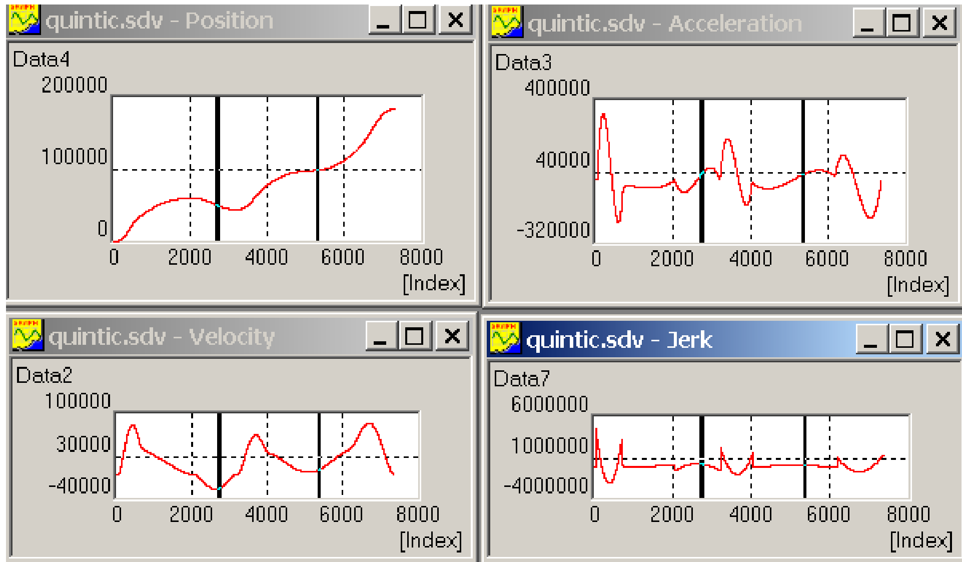


Figure 7-3: 5th Order Polynomial Example

7.3.1.3 7th Order Polynomial - Polynom7 (ePolynom7Interp)

Septic polynomial guarantees continuity by position, velocity, acceleration and jerk. Its disadvantage is a higher variability. It's the most universal interpolation mode that can be recommended in most cases.

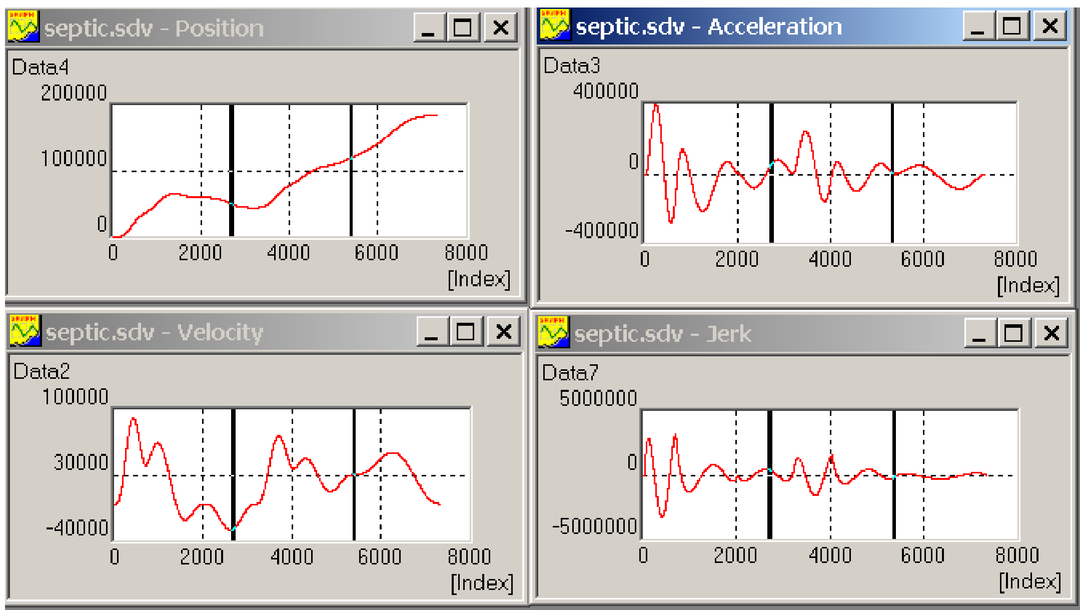


Figure 7-4: 7th Order Polynomial Example

Segments with the septic interpolation can be smoothly connected to any other segments including linear that can be seen in Figure 7-5

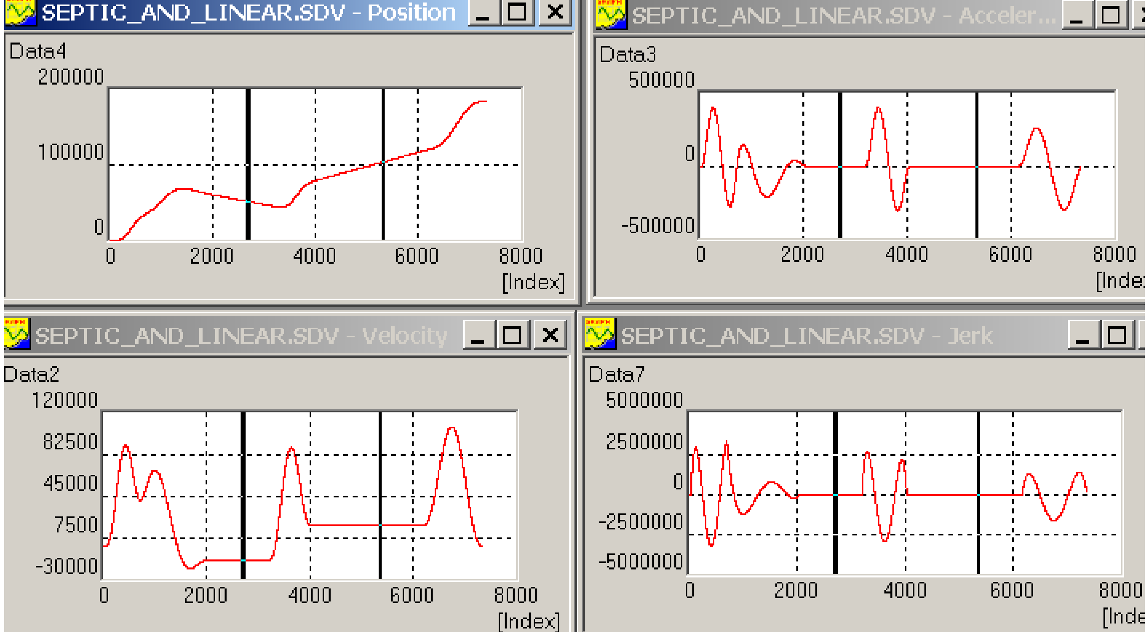


Figure 7-5: 7th Order & Linear Polynomial Example

7.3.2 Sinusoidal interpolation functions

An ECAM profiler also supports a number of sinusoidal interpolation options.

Cycloidal interpolation guarantees continuity by position and velocity. It produces a segment profile that starts and ends with the zero acceleration with cycloidal position and sinusoidal velocity functions. It starts and ends the segment with zero acceleration.

7.3.2.1 Cycloid Position Sinusoidal Velocity - Cycloidal Position (eCycloidPositionInterp)

This interpolation mode produces cycloidal position and sinusoidal velocity interpolation. Sinusoidal velocity is an advantage of this interpolation type but it has two significant drawbacks. The first is that it starts and ends with a maximum jerk. The second is that in the general case where Yx_start ≠ 0, Yx_end ≠ 0, these start and end derivatives must obey the requirement

$$\Delta Y = 0.5(\dot{Y}_i + \dot{Y}_{i+1})\Delta X_i.$$

For this reason, this interpolation mode can be recommended mainly for the dwell-rise-dwell case with Yx_start = 0, Yx_end = 0. An example of the cycloid segment between two dwells can be seen in Figure 7-6.

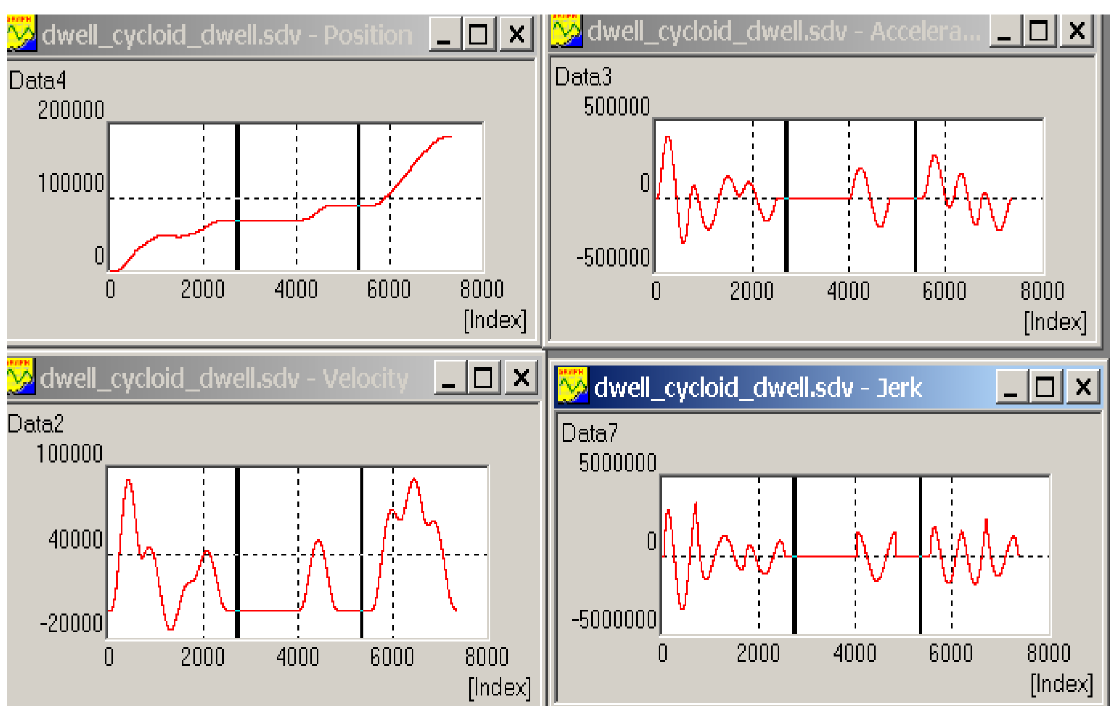


Figure 7-6: Cycloid Position Sinusoidal Velocity Example

Another case that this mode can be effectively used, is a Ramp From Standstill to the Linear Segment and a ramp to zero velocity after the segment with the linear interpolation. The requirement mentioned above must be fulfilled.

An example of such a ramp can be seen in Figure 7-7. It was designed for the ECAM table

Master Position	Slave position	Interpolation Mode
0	2000	
2000	4000	eCycloidPositionInterp
10000	20000	eLinearInterp
12000	22000	eCycloidPositionInterp

For the first cycloidal segment $\dot{Y}_i$ = 0 and $\dot{Y}_{i+1}$ = (20000-4000)/(10000-2000) = 2

Then for the first segment 0.5(0 + 2)2000 = (4000-2000).

For the last segment $\dot{Y}_i$ = (20000-4000)/(10000-2000) and

$\dot{Y}_{i+1}$ = 0 than 0.5(2 + 0)2000 = (22000-20000).

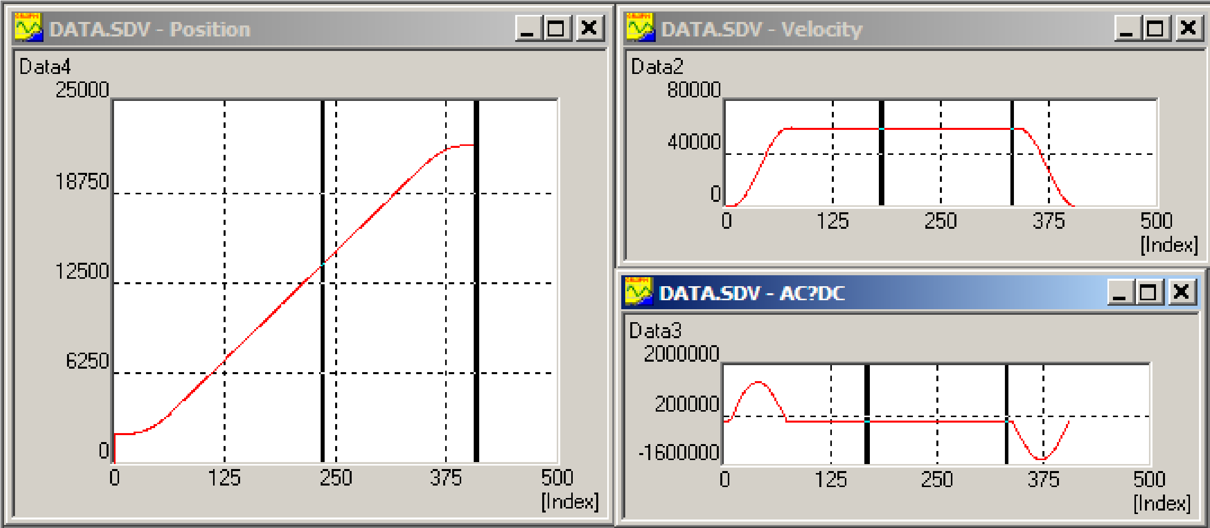


Figure 7-7: Cycloid Position Sinusoidal Velocity Example For Ramp

The Maestro supports two interpolation modes with the sinusoidal acceleration and cyclodal velocity profiles as well.

7.3.2.2 Cycloid Sinusoidal velocity Modified 1 - Cycloid Velocity 1 (eCycloidVelocityModified1Interp)

Triangle sinusoidal acceleration - modified triangle acceleration with AC(t) and DC(t) increasing by the sinusoid to some maximum value and then decreasing to zero. It guarantees continuity by position, velocity, acceleration and jerk.

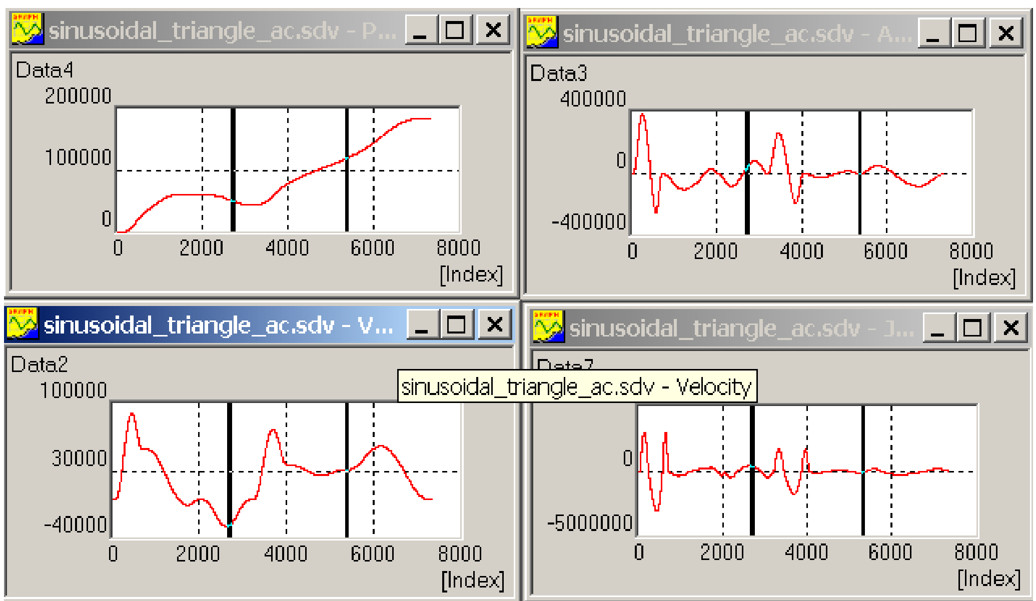


Figure 7-8: Cycloid Sinusoidal velocity Modified 1 example

7.3.2.3 Cycloid Sinusoidal velocity Modified 2 - Cycloid Velocity 2 (eCycloidVelocityModified2Interp)

Cycloid Sinusoidal velocity Modified 2 usually consists of two parts:

- A movement with an acceleration increasing by the sinusoid from zero to ACmax, movement with AC(max) (parabolic position, linear velocity profile) and acceleration decreasing by the sinusoid to zero.
- A movement with a deceleration decreasing by the sinusoid from zero to –ACmax, movement with –ACmax (parabolic position profile, linear velocity profile) and deceleration increasing by the sinusoid to zero.
- It guarantees continuity by position, velocity, acceleration and jerk (if all the segments defined with this interpolation mode). An example can be seen in Figure 7-9.

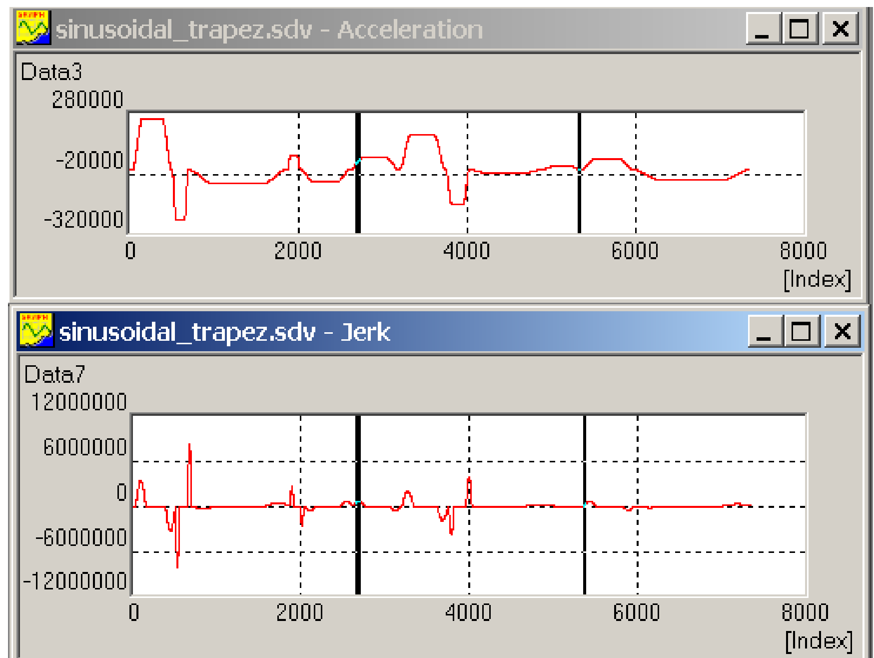


Figure 7-9: Cycloid Sinusoidal Velocity Modified 2 Example

In case of the dwell-rise-dwell movement an acceleration is evidently precedes deceleration. In a general case Yx_start ≠ 0, Yx_end ≠ 0 the sequence of acceleration and deceleration parts may be reverse.

For the dwell-rise-dwell case, it can be defined as a skew parameter that defines the relation of the maximum acceleration and maximum deceleration that in this case are not equal. If this parameter is not defined, then AC(max) = DC(max) and the length of the acceleration part is equal to the length of the deceleration part.

In a special case where $\Delta Y_i = 0.5(\dot{Y}_i + \dot{Y}_{i+1})\Delta X_i$, the segment profile contains only acceleration or only deceleration part. This case can be seen in Figure 7-10. On the first and the last segments is used interpolation with the modified trapezoidal acceleration and only AC at start segment and only DC at the end segment (Figure 7-10).

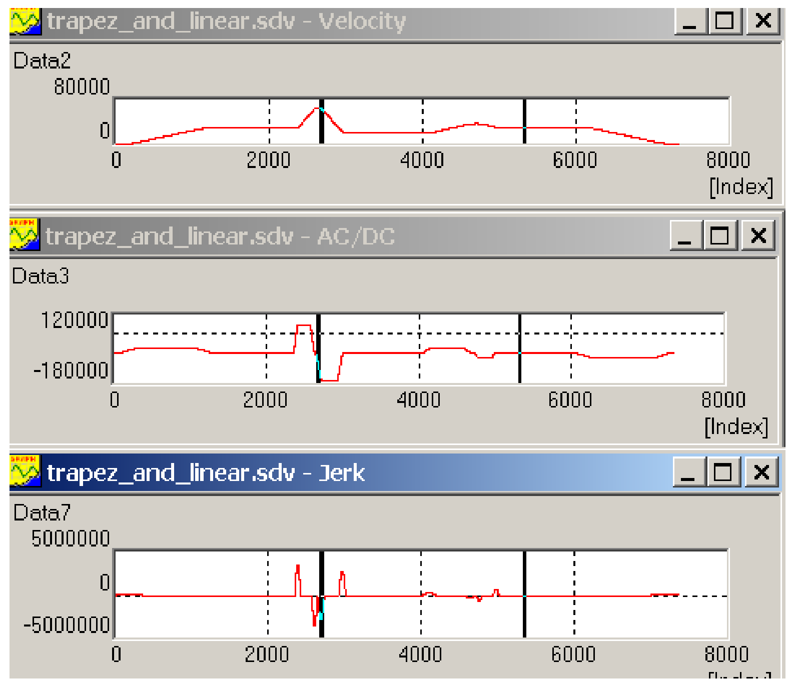


Figure 7-10: Cycloid Sinusoidal Velocity Modified 2 Linear Acceleration Example

Both interpolation modes with the sinusoidal acceleration start and end segment profile with the zero acceleration and jerk.

7.4 Basic flow for CAM process

1. Load CAM table (from user program or file)
2. Call to MC_CamIn with input parameter as desired.
3. Call MC_CamOut (or MC_Stop) to end the CAM process.

7.4.1 Using an Array in User Program

In order to use an array from the user program, three functions should be used in the following order:

1. Call MC_CamTableInit to allocate memory in GMAS for a certain CAM table.
2. Call MC_CamTableSelect using the table handler, which was retrieved by the call above to MC_CamTableInit.
3. Call MC_CamTableAdd, using the table handler above, to load data into table.

7.4.2 Using a File

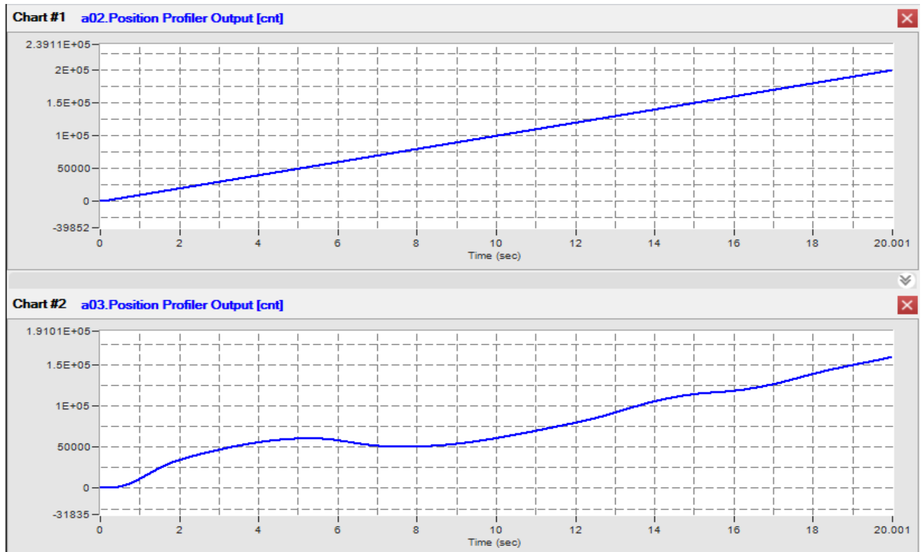
Call the function MC_CamTableSelect with table handler set to (-1) and file path set as appropriate. The file format should be compatible to the definitions for CAM table format.

7.5 Examples

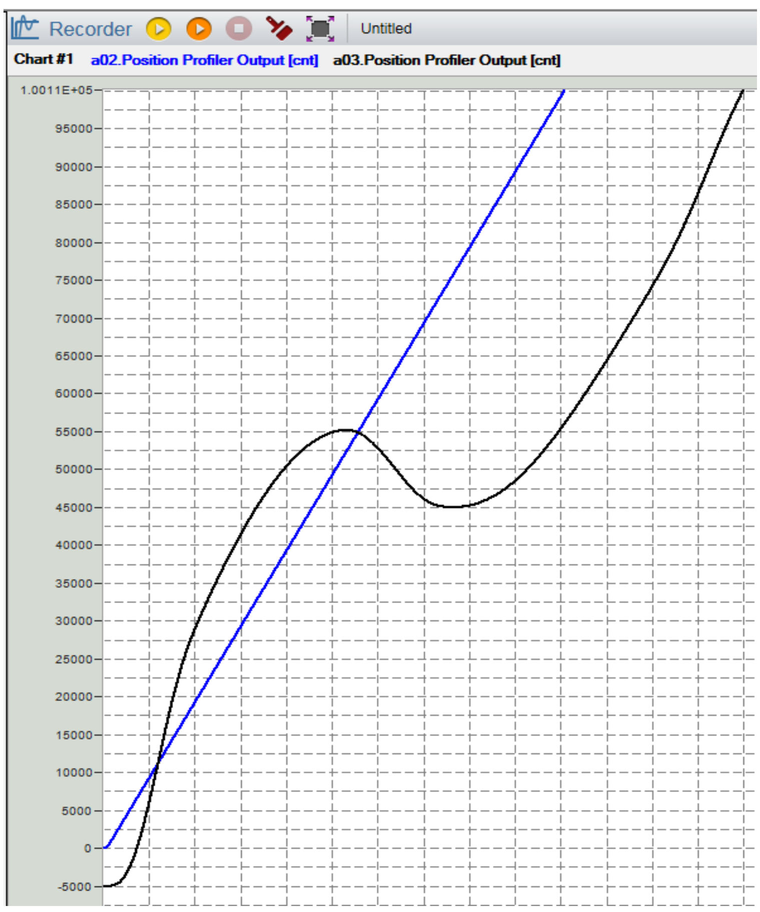
Consider the CAM table described above, the following charts describes the way of using relative/absolute tables.

7.5.1 Slave Relative

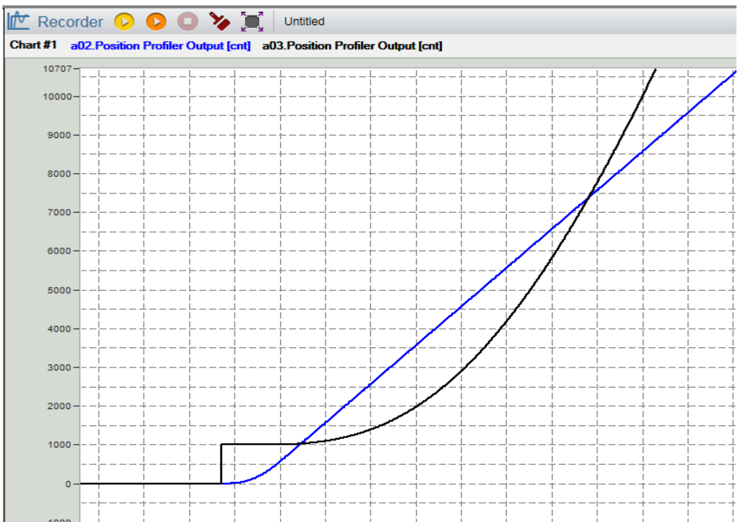
Input parameter ucIsSlavePosAbsolute is set to false. The participant axes are Master Axis a02 (the upper chart), and the slave axis a03 (the lower chart).



Slave axis (a03) position on CAM activation is zero, therefore the slave position on each row relates to zero and there is no jump of the slave position. The relative offset refers to slave position on CAM activation. If the slave position on CAM activation is changed to -5000.



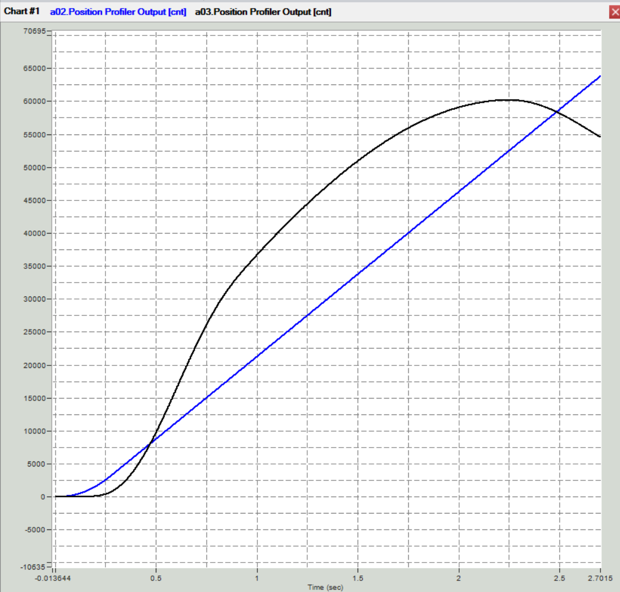
The chart produces no jump since the relative slave position in the first row is zero. So, what happen if CAM process starts where the slave position in table is not zero? Let's assume we change the slave position in the first row to 1000 and slave axis position on CAM activation is zero.



The jump in this case is noticeable obviously.

7.5.2 Slave Absolute

In general, setup of absolute values in slave column, requires that slave axis position when CAM is activated will match to the specified position of the corresponding row. Please note that CAM process may start in first row as well as other rows within the table. If this determining is not fulfilled a jump in slave position is expected. Auto offset mechanism comes to solve the problem of possible jumps but we get to this option later when we discuss the parameters of MC_CamIn. We shall use the same table of the relative table example (see above) but Input parameter ucIsSlavePosAbsolute is set true. Participant axes: master Axis a02 (blue in chart below), slave axis a03 (black in chart below) and slave axis a03 was moved to zero before CAM activation:



One can see the slave motion is smooth. Obviously, if slave axis position was 1000 on CAM activation, a jump in position would be expected.

7.6 ECAM Functions

The following ECAM functions are described:

ECAM functions
MMC_CAMTableInit
MMC_CAMTableSelect
MMC_CAMRef
MMC_CAMTableUnload
MMC_CAMTableAdd
MMC_CAMTableSet
MMC_CAMIn
MMC_CAMOut
MMC_CAMGetStatus

7.6.1 MMC_CamTableInit

This function allocates memory for the ECAM table, prepares and initializes the function block in journal. In general, it is similar to MC_TableInit without an option for dynamic append.

```
MMC_LIB_API int MMC_CamTableInitCmd(
MMC_CONNECT_HNDL hConn,
MMC_CAMTABLEINIT_IN* pInParam,
MMC_INITTABLE_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_INITTABLE_IN** input data structure using the MMC_InitTableCmd.

pOutParam

Points to the **MMC_INITTABLE_OUT** output structure receiving information, as a result of calling the MMC_InitTableCmd.

Remarks

It should be noted that when using this function, no data is loaded. The function returns hMemHandle, which is the unique path ID (similar to splines).

Online modification of slave(s) column(s) is perfromed via MC_CamTableSet.

Scope

All

MMC_CAMTABLEINIT_IN Structure

```
typedef struct _mmc_camtableinit_in{
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned long ulMaxPoints;
unsigned long ulUnderflowThreshold;
unsigned short usDimension;
unsigned char ucSuperimposed;
unsigned char ucIsFixedGap;
CURVE_TYPE_ENUM eCurveType
unsigned char ucExecute;
unsigned char ucSpare[32];
} MMC_CAMTABLEINIT_IN;
```

Parameters

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows, but only the Buffered Mode is supported:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ulMaxNumberOfPoints

The maximum number of points allocated. Theoretical maximum value of $4.294E^{+9}$, however in practice will be limited well below this value.

ulUnderflowThreshold

The underflow limit. An event will be generated if the number of points between the current index, and the end index falls below this value. Value cannot be greater that the ulMaxNumberOfPoints value. Any +ve values accepted.

usDimension

The dimensions of the table, i.e. the number of slaves. Since group is not supported at this phase, dimension is always 1. A +ve number

ucIsFixedGap

Mandatory factor here for memory allocation. Boolean result to question, Is the Gap variable (0) or fixed (1)? 0, or 1

CURVE_TYPE_ENUM eCurveType

Interpolation type. User defined type allows different type for each segment. Any other type sets that type (same type) for all segments. *

CurveType of each segment defines the curve type between current segment and its predecessor.

CurveType defines interpolation type. If CurveType is 'UserDefineInterp', then CurveType must be supplied by user array (via MC_CamTableAdd/Set) or table's file on the last column of each row. Otherwise the supplied CurveType will be used for all segments and CurveType must not be supplied by user array (via MC_CamTableAdd/Set). File loader ignores CurveType column if CurveType is not 'UserDefineInterp'.

CurveType of each segment defines the curve type between current segment and its predecessor. Defined by the Interpolation type enumerator CURVE_TYPE_ENUM with values:

eTableDefInterp = 0

eLinearInterp = 1

ePolynom5Interp = 2

ePolynom7Interp = 3,

eCycloidPositionInterp = 4

eCycloidVelocityModified1Interp = 5

eCycloidVelocityModified2Interp = 6

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE. Not currently in use.

ucSpare[32]

Spare for future use

ucExecute

Boolean result to question, Is the function running on a raising edge or not?

MMC_INITTABLE_OUT Structure

typedef struct mmc_inittable_out{
MC_PATH_REF hMemHandle;
unsigned short usStatus;
short sErrorID;
} MMC_INITTABLE_OUT;

Parameters

hMemHandle

MC_PATH_REF alias for unsigned integer with values and handle to a journal entry where the pointer to the shared memory is located, and indicates the memory area where the spline is allocated. In fact this is the unique identifier between PathSelect, MovePath and PathUnselect commands. MC_PATH_REF is the journal entry path reference.

hMemHandle produces +ve Integer values.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-10 describes the function for MMC_CamTableInit as applied within the IEC 61131 programming.

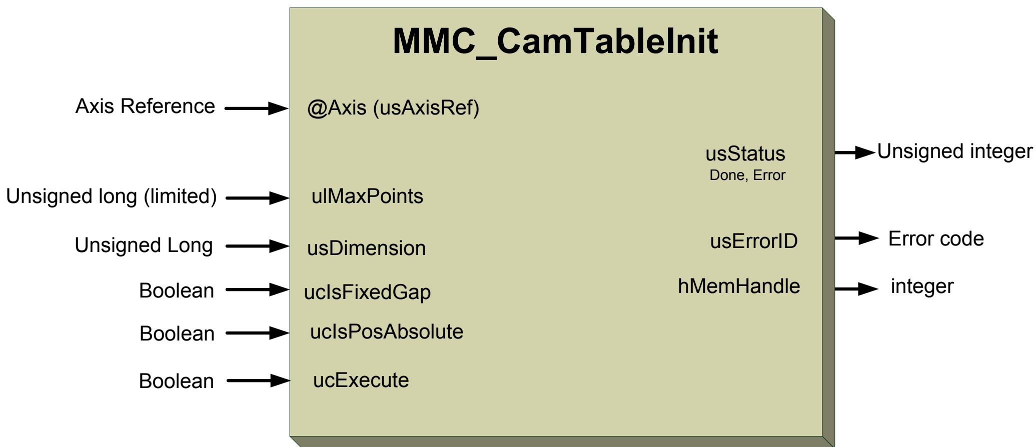


Figure 7-11: MMC_CamTableInit function

7.6.1.2 Function Code Example

Refer to the example in section **7.7 Application Example**.

7.6.2 MMC_CamTableSelect

This function selects a table by input handler.

MMC_LIB_API int MMC_CamTableSelectCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_CAMTABLESELECT_IN* pInParam,
OUT MMC_CAMTABLESELECT_OUT* pOutParam
);

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CAMTABLESELECT_IN** input data structure using the MMC_CamTableSelectCmd.

pOutParam

Points to the **MMC_CAMTABLESELECT_OUT** output structure receiving information, as a result of calling the MMC_CamTableSelectCmd.

Remarks

MC_CamTableSelect is mandatory invocation before MC_CamIn execution, whether the table is loaded from file or from memory. If MC_CamTableInit was executed then one can use the table handler as input for MC_CamTableSelect. Otherwise MC_CamTableSelect loads the table from file and retrieves the table handler by itself. Invocation of MC_CamIn will fail if MC_CamTableSelect was not executed for the corresponding table at least once.

Scope

If the table is missing (handler < 0 for instance) then a table is loaded from file (file path is given). Otherwise table is selected and input parameters are set (or override) in function block structure.

MMC_CAMTABLESELECT_IN Structure

```
typedef struct MMC_CAMTABLESELECT_IN {
MC_CAMREF eCamTableDescr;
unsigned int uiStartMode;
unsigned char ucIsMasterPosAbsolute;
unsigned char cIsSlavePosAbsolute;
unsigned char ucSpare[32];
unsigned char ucExecute;
} MMC_CAMTABLESELECT_IN;
```

Parameters

CamTableDescr

MC_CAMREF

MC_CamRef Input/output parameter of MC_CamTableSelect is an Elmo specific data type, namely:

```
typedef struct_mmc_camtable_ref{
double dbGap;
double dbMasterStartPosition;
MC_COORD_SYSTEM_ENUM eCoordSystem;
int iCamTableID;
CAM_TABLE_MODE_ENUM eTableMode;
MC_PATH_DATA_REF pPathToTableFile;
unsigned char ucSpare[32];
} MC_CamRef;
```

dbGap

Difference between rows. Relevant only if gap is fixed.
Double.

dbMasterStartPosition

Master position in CAM table first entry. Must be specified in case of fixed gap (no master column) if master position is absolute.
Double value.

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator value. This is currently irrelevant because group is not supported.The options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

Relevant only on file loader.

pPathToTableFile

MC_PATH_DATA_REF path to file

iCamTableID

iCamTableID <0 means - load new table from file. Otherwise it is a handle of existing table, which was either retrieved by MMC_CamTableInit or previous call to MMC_CamTableSelect.

Integer values

CAM_TABLE_MODE_ENUM eTableMode

Modifiable table or not. Defines whether the table will be read only or adjustable during the course of operation. Dependant on the enumerator CAM_TABLE_MODE_ENUM with values:

eCAMT_RWMode = 0

eCAMT_ROMode = 1

where eCAMT_ROMode defines that on-the-fly changes are forbidden.

ucSpare[32]

8 longs spare

uiStartMode

Reserved for future use of Ramp-In and other options.

ucIsMasterPosAbsolute

Boolean parameter value. If 1 GMAS refers to master column as absolute values, otherwise 0 as relative values.

cIsSlavePosAbsolute

Boolean parameter value. If 1 GMAS refers to slave column as absolute values, otherwise 0 as relative values.

ucSpare[32]

8 longs spare

ucExecute

Boolean result to question, Is the function running on a raising edge or not?

MMC_CAMTABLESELECT_OUT Structure

typedef struct MMC_CAMTABLESELECT_OUT {
unsigned short usStatus;
short sErrorID;
MC_PATH_REF hMemHandle;
} MMC_CAMTABLESELECT_OUT;

Parameters

hMemHandle

MC_PATH_REF alias for unsigned integer with values and handle to a journal entry where the pointer to the shared memory is located, and indicates the memory area where the spline is allocated. In fact this is the unique identifier between PathSelect, MovePath and PathUnselect commands. MC_PATH_REF is the journal entry path reference.

hMemHandle produces +ve Integer values.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-11 describes the function for MMC_CamTableSelectCmd as applied within the IEC 61131 programming.

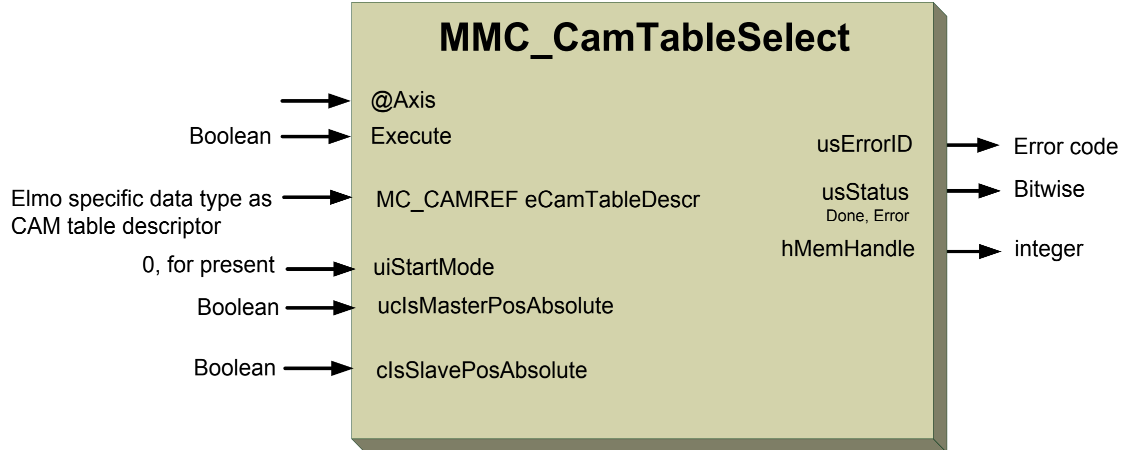


Figure 7-12: MMC_CamTableSelectCmd function

7.6.2.2 Function Code Example

Refer to the example in section **7.7 Application Example**.

7.6.3 MMC_CamTableUnload

The function unloads a ECAM table from the Maestro and frees a memory segment in the Maestro shared memory according to the dimension and number of points given in a file.

```
MMC_LIB_API int MMC_UnloadTableCmd(
MMC_CONNECT_HNDL hConn,
MMC_UNLOADTABLE_IN* pInParam,
MMC_UNLOADTABLE_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_UNLOADTABLE_IN** input data structure using the MMC_UnloadTableCmd.

pOutParam

Points to the **MMC_UNLOADTABLE_OUT** output structure receiving information, as a result of calling the MMC_UnloadTableCmd.

Remarks

The file always contains data.

Scope

All

MMC_UNLOADTABLE_IN Structure

```
typedef struct mmc_unloadtable_in{
MC_PATH_REF hMemHandle;
unsigned char ucExecute;
}MMC_UNLOADTABLE_IN
```

Parameters

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located, obtained from the PathSelect command. MC_PATH_REF is the journal entry path reference.

hMemHandle can have Integer values.

ucExecute

Start the execution command. Boolean TRUE/FALSE values. Currently not in use and set to 1.

MMC_UNLOADTABLE_OUT Structure

```
typedef struct mmc_unloadtable_out{
unsigned short usStatus;
short sErrorID;
}MMC_UNLOADTABLE_OUT
```

Parameters

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-12 describes the function for MMC_UnloadTable as applied within the IEC 61131 programming.

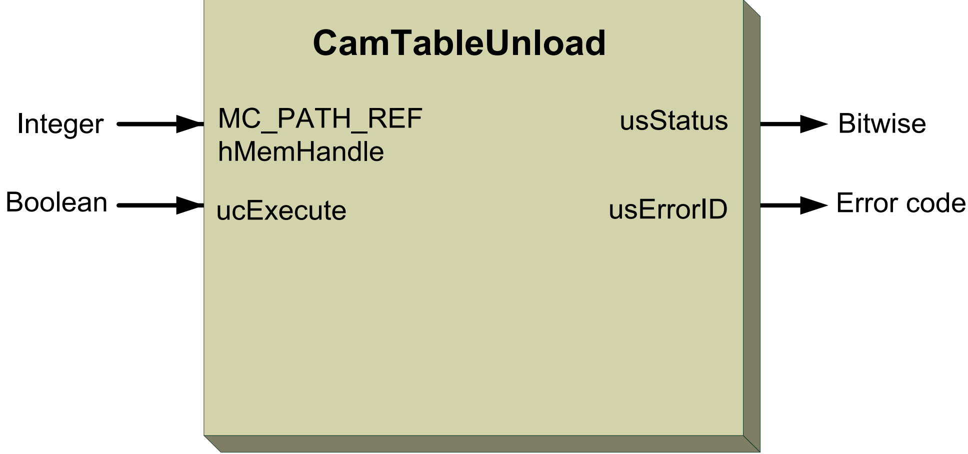


Figure 7-13: MMC_UnloadTable function

7.6.3.2 Function Code Example

Refer to the example in section **7.7 Application Example**.

7.6.4 MMC_CamTableAdd

This function appends points to an existing table.

```
MMC_LIB_API int MMC_CamTableAddCmd(
MMC_CONNECT_HNDL hConn,
MMC_CAMTABLESET_IN* pInParam,
MMC_CAMTABLESET_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_CAMTABLESET_IN** input data structure using the MMC_CamTableAddCmd.

pOutParam

Points to the **MMC_CAMTABLESET_OUT** output structure receiving information, as a result of calling the MMC_CamTableAddCmd.

Remarks

The prerequisite to using this function is a call to MC_CamTableInit.

Scope

Loads CAM tables from an array in a user program into the Maestro. The user should be aware of the amount of columns used for each row (a point).

- Use an array of type double.
- The array must contain a sequence of rows (points), one by one.
- The columns order must be as follows: master, slave, curve type.
- Each row must contain the slave position. Master position and curve type are optional.
- If the Master gap is fixed then a row contains no master column, otherwise it does.
- If the curve type parameter is defined by user then a special column for curve type must be supplied, otherwise it must not.

MMC_CAMTABLESET_IN Structure

```
typedef struct MMC_CAMTABLESET_IN{
double dbTable[NC_ECAM_MAX_ARRAY_SIZE];
unsigned long ulStartIndex;
unsigned long ulNumberOfPoints;
MC_PATH_REF hMemHandle;
unsigned char ucSpare[32];
unsigned char ucExecute;
} MMC_CAMTABLESET_IN;
```

Parameters

dbTable[NC_ECAM_MAX_ARRAY_SIZE]

Range of Points to be added to the ECAM Table. The array consists of a range of points in columns.

The maximum value of the NC_ECAM_MAX_ARRAY_SIZE array variable is 164. This is the maximum number of point allowed in the table.

ulStartIndex

Row index to start from. +ve number.

ulNumberOfPoints

Number of rows to add. +ve number

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located obtained from the PathSelect command. MC_PATH_REF is the journal entry path reference.

hMemHandle has Integer values.

ucSpare[32]

8 longs spare

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_CAMTABLESET_OUT Structure

```
typedef struct _mmc_camtableset_out{
usigned short usStatus;
unsigned short usErrorID;
} MMC_CAMTABLESET_OUT;
```

Parameters

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-13 describes the function for MMC_CamTableAddCmd as applied within the IEC 61131 programming.

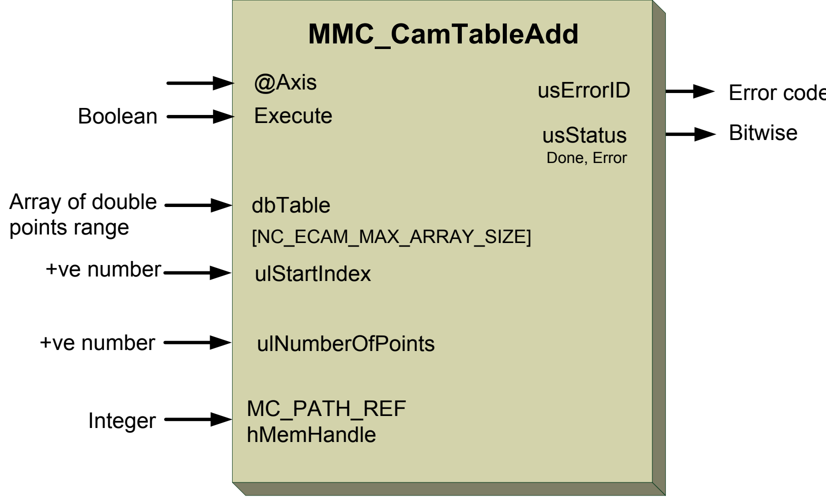


Figure 7-14: MMC_CamTableAddCmd function

7.6.4.2 Function Code Example

Refer to the example in section **7.7 Application Example**.

7.6.5 MMC_CamTableAddEx

This function allows appending an unlimited number of rows to an existing table present in memory.

```
MMC_LIB_API int MMC_CamTableAddEx(
MMC_CONNECT_HNDL hConn,
MMC_CAMTABLESET_INEx* pInParam,
MMC_CAMTABLESET_OUTEx* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_CAMTABLESET_INEx** input data structure using the MMC_CamTableAddCmd.

pOutParam

Points to the **MMC_CAMTABLESET_OUTEx** output structure receiving information, as a result of calling the MMC_CamTableAddCmd.

Remarks

The prerequisite to using this function is a call to MC_CamTableInit.

Scope

Allows adding an unlimited number of rows to the table.

MMC_CAMTABLESET_INEx Structure

typedef struct MMC_CAMTABLESET_INEx{
double dbTable[];
unsigned short usColumns;
unsigned long ulNumberOfPoints;
MC_PATH_REF hMemHandle;
} MMC_CAMTABLESET_INEx;

Parameters

dbTable[]

Range of rows to be added to the ECAM Table without limit.

ulNumberOfPoints

Number of rows to add. +ve number

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located obtained from the PathSelect command. MC_PATH_REF is the journal entry path reference.

hMemHandle has Integer values.

MMC_CAMTABLESET_OUTEx Structure

typedef struct _mmc_camtableset_out{
usigned short usStatus;
unsigned short usErrorID;
} MMC_CAMTABLESET_OUT;

Parameters

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-14 describes the function for MMC_CamTableAddEx as applied within the IEC 61131 programming.

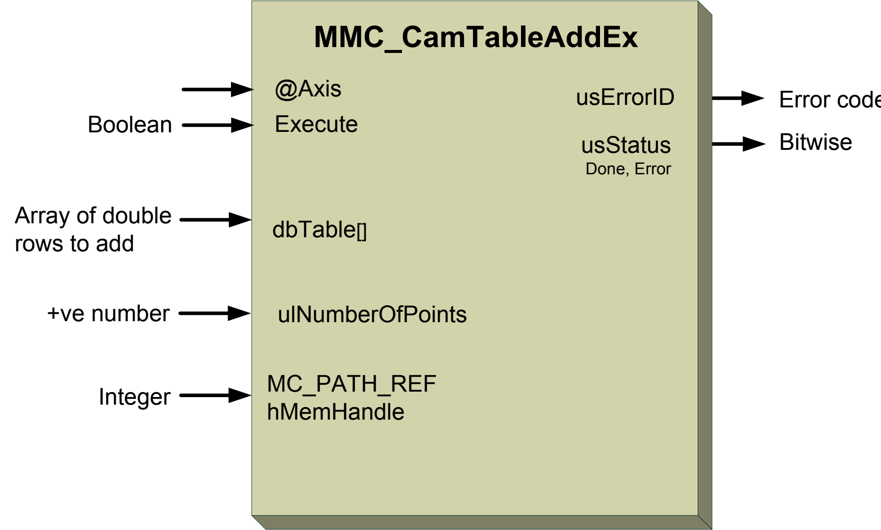


Figure 7-15: MMC_CamTableAddEx function

7.6.5.2 Function Code Example

Refer to the example in section 7.7 Application Example.

7.6.6 MC_CamTableSet

This function MMC_CamTableAdd is used for loading a table from memory.

```
MMC_LIB_API int MMC_CamTableSetCmd(
MMC_CONNECT_HNDL hConn,
MMC_CAMTABLESET_IN* pInParam,
MMC_CAMTABLESET_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Not Supported

Source GMAS\includes\MMC_PVT_ECAM_API.h
GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CAMTABLESET_IN** input data structure using the MMC_CamTableSetCmd.

pOutParam

Points to the **MMC_CAMTABLESET_OUT** output structure receiving information, as a result of calling the MMC_CamTableSetCmd.

Remarks

None

Scope

CAM table online modification is enabled. This capability refers to modification of a specific CAM table while CAM process is active (or not). The user should be aware of the amount of columns used for each row (a point).

- Use an array of type double.
- The array must contain a sequence of rows (points), one by one.
- The columns order must be as follows: master, slave, curve type.
- Each row must contain the slave position. Master position and curve type are optional.
- If the Master gap is fixed then a row contains no master column, otherwise it does.
- If the curve type parameter is defined by user then a special column for curve type must be supplied, otherwise it must not.
- User responsible when modifying rows (segments in table), which are not processed at that time.

MMC_CAMTABLESET_IN Structure

```
typedef struct MMC_CAMTABLESET_IN{
double dbTable[NC_ECAM_MAX_ARRAY_SIZE];
unsigned long ulStartIndex;
unsigned long ulNumberOfPoints;
MC_PATH_REF hMemHandle;
unsigned char ucExecute;
} MMC_CAMTABLESET_IN;
```

Parameters

dbTable[NC_ECAM_MAX_ARRAY_SIZE]

Range of Points to be added to the ECAM Table. The array consists of a range of points in rows.

The maximum value of the NC_ECAM_MAX_ARRAY_SIZE array variable is 164. This is the maximum number of point allowed in the table.

ulStartIndex

Row index to start from. +ve number.

ulNumberOfPoints

Number of rows to add. +ve number.

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located obtained from the PathSelect command. MC_PATH_REF is the journal entry path reference. The table access handler.

hMemHandle has Integer values.

ucExecute

Start the execution command. Run on raising edge. Boolean TRUE/FALSE values.

MMC_CAMTABLESET_OUT Structure

```
typedef struct _mmc_camtableset_out{
usigned short usStatus;
unsigned short usErrorID;
} MMC_CAMTABLESET_OUT;
```

Parameters

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-15 describes the function for MMC_CamTableSetCmd as applied within the IEC 61131 programming.

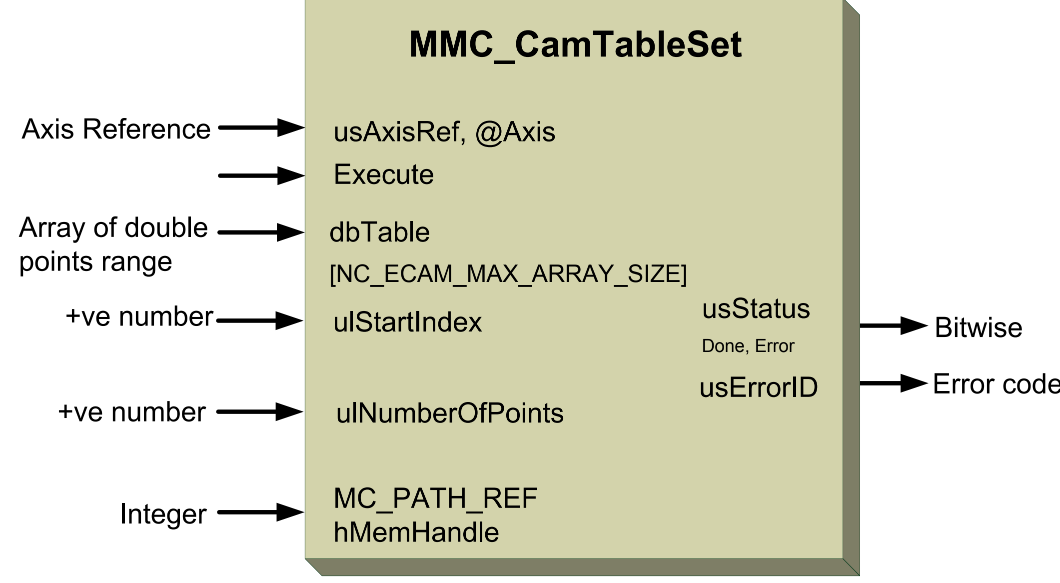


Figure 7-16: MMC_CamTableSetCmd function

7.6.6.2 Function Code Example

Refer to the example in section **7.7 Application Example**.

7.6.7 MMC_CamIn

MC_CamIn executes the CAM process.

```
MMC_LIB_API int MMC_CamInCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CAMIN_IN* pInParam,
OUT MMC_CAMIN_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_CAMIN_IN** input data structure using the MMC_CamInCmd.

pOutParam

Points to the **MMC_CAMIN_OUT** output structure receiving information, as a result of calling the MMC_CamInCmd.

Remarks

Scope

All

MMC_CAMIN_IN Structure

```
typedef struct _mmc_camin_in{
double dbMasterOffset;
double dbSlaveOffset;
double dbMasterScaling;
double dbSlaveScaling;
double dbMasterStartDistance;
double dbMasterSyncPosition;
ECAM_VALUE_SRC_ENUM eMasterValueSource;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned int uiStartMode;
CURVE_TYPE_ENUM eCurveType;
ECAM_PERIODIC_ENUM ePeriodicMode;
unsigned int uiCamTableID;
unsigned short usMaster;
unsigned char ucAutoOffset;
unsigned char ucExecute;
unsigned char ucSpare[32];
} MMC_CAMIN_IN;
```

Parameters

dbMasterOffset

Master offset from master definition in CAM table. +ve number

dbSlaveOffset

Slave offset from slave definition in CAM table. +ve number

dbMasterScaling

Master scaling of the master definition in the CAM table. +ve number

dbSlaveScaling

Slave scaling of the master definition in the CAM table. +ve number

dbMasterStartDistance

Backward distance from dbMasterSyncPosition to allow Ramp-In. When Ram-In implemented this parameter will have a value. Now it is always zero.

MasterStartDistance defines backward distance from MasterSyncPosition. Together with MasterSyncPosition they both defines the position in which coupling (engagement) with slave starts. That is to say, the position in which the CAM process starts. How this CAM process starts depends on the "StartMod" input parameter (see below). In any case Master Start Position must be within table boundaries, otherwise CAM process should report an error.

dbMasterSyncPosition

Defined as relative to first phase of master in the CAM table. If the table is relative then it is defined as relative master position just like any other phase in table.

The Master Start Position is defined as MasterSyncPosition. For now it only defines the position in which CamIn output flag is turned on. The Master Start Position must be within the range of the CAM table.

MasterSyncPosition is always relative to table's first phase. It defines the master position within the table in which the function block is in sync state. There will be no additional scaling nor offset calculations on MasterSyncPosition. This is an absolute value, which relates to the first position after scaling and offset supplement, based on the type of the table (absolute/relative).

ECAM_VALUE_SRC_ENUM eMasterValueSource

The Master source value is defined by the eMasterValueSource input parameter. It may be a Maestro parameter for target position, actual position (integer) or some kind of auxiliary. If the master axis operates in modulo mode, then the target position uses the Maestro parameter as a source for the target modulated position (UU).

The Master value source dependant on whether set, actual or based on another value, according the ECAM_VALUE_SRC_ENUM enumerator.

eECAM_SET_VALUE	= 0
eECAM_ACTUAL_VALUE	= 1
eECAM_AUX_VALUE	= 2

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows, but only the Buffered Mode is supported:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block

BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

uiStartMode

Overrides uiStartMode of MC_CamTableSelect. Reserved for future use of Ramp-In and other options.

CURVE_TYPE_ENUM eCurveType

Interpolation type, according to the options:

eTableDefInterp = 0
eLinearInterp = 1
ePolynom5Interp = 2
ePolynom7Interp = 3,
eCycloidPositionInterp = 4
eCycloidVelocityModified1Interp = 5
eCycloidVelocityModified2Interp = 6

ECAM_PERIODIC_ENUM ePeriodicMode

Describes the perodicity mode of the function, according to the following options:

eCAM_NON_PERIODIC	= 0	One shot
eCAM_PERIODIC	= 1	periodic
eCAM_PERIODIC_LINEAR	= 2	periodic-linear

uiCamTableID

Table index which is retrieved by the function MMC_CamtableSelect.

usMaster

The master axis reference. +ve short value.

ucAutoOffset

Boolean question whether the Offset is user defined or automatically defined. False (0) for user defined offsets, true (1) for Auto offset. Auto offset is used to avoid any jumps by the slave axis. It means that at the engagement stage, the slave position in table is adjusted to the slave's actual position. All slave positions in all other rows are adjusted by the same offset. Auto offset parameter overrides the user definition for slave offset.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

ucSpare[32]

Spare for future use

MMC_CAMIN_OUT Structure

```
typedef struct _mmc_camin_out{
unsigned int uiHandle;
unsigned int uiEndOfProfile;
unsigned short usStatus;
short sErrorID;
} MMC_CAMIN_OUT;
```

Parameters

uiHandle

Table access handler.

uiEndOfProfile

Variable counter increases each time the master meets the end of the table. Boolean for IEC. Count variable increases each time the master meets the end of the table.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-16 describes the function block for MMC_CamInCmd as applied within the IEC 61131 programming.

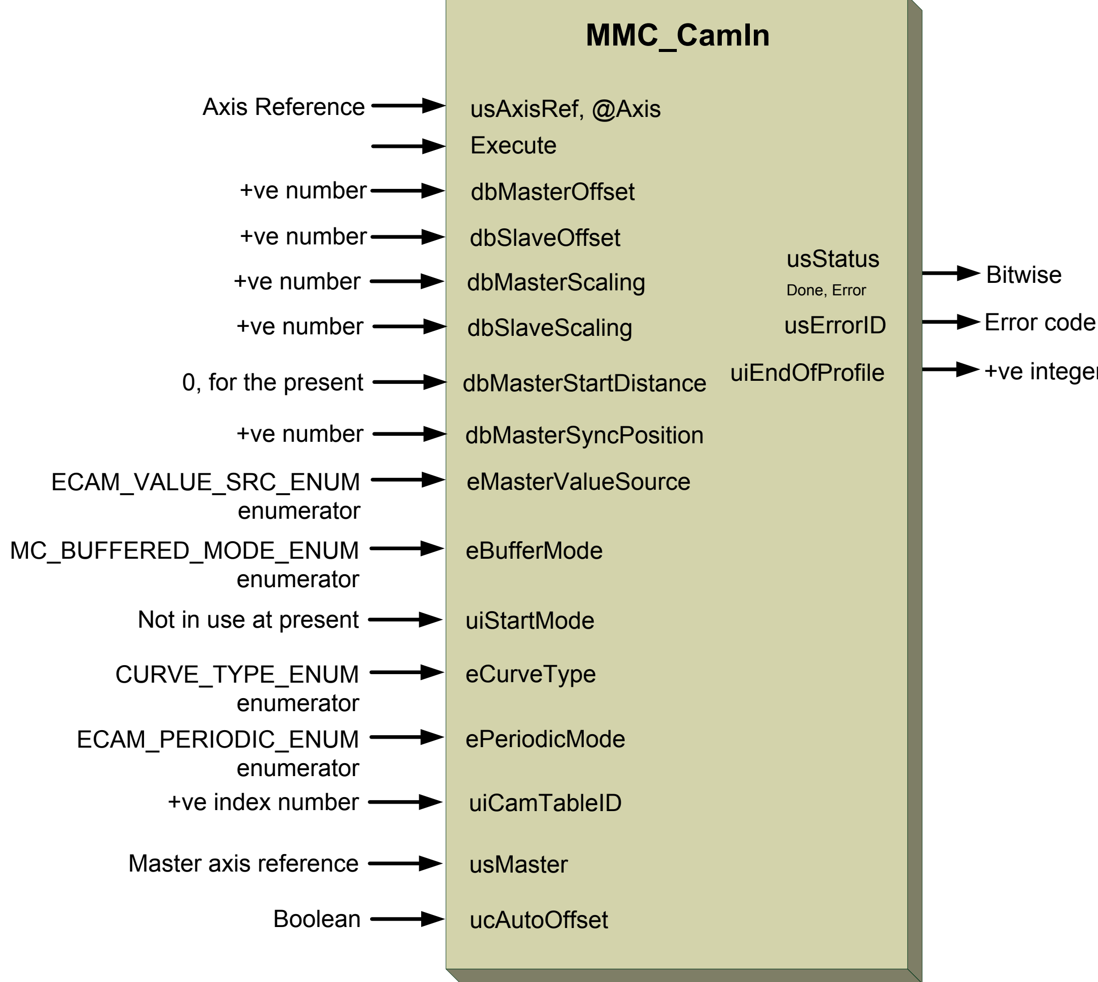


Figure 7-17: MMC_CamInCmd function

7.6.7.2 Function Code Full Example

Refer to the example in section 7.7 Application Example.

7.6.8 MMC_CamOut

Performs a MC_Stop on the slave axis to disengage the CAM process.

```
MMC_LIB_API int MMC_CamOutCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CAMOUT_IN* pInParam,
OUT MMC_CAMOUT_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CAMOUT_IN** input data structure using the MMC_CamOutCmd.

pOutParam

Points to the **MMC_CAMOUT_OUT** output structure receiving information, as a result of calling the MMC_CamOutCmd.

Remarks

Scope

All

MMC_CAMOUT_IN Structure

```
typedef struct _mmc_camout_in{
unsigned char ucExecute;
} MMC_CAMOUT_IN;
```

Parameters

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

MMC_CAMOUT_OUT Structure

```
typedef struct _mmc_camout_out{
unsigned int uiHandle;
unsigned short usStatus;
short sErrorID;
} MMC_CAMOUT_OUT;
```

Parameters

uiHandle

Table access handler.

usStatus

Bitwise returned command status with th values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-17 describes the function for MMC_CamOutCmd as applied within the IEC 61131 programming.

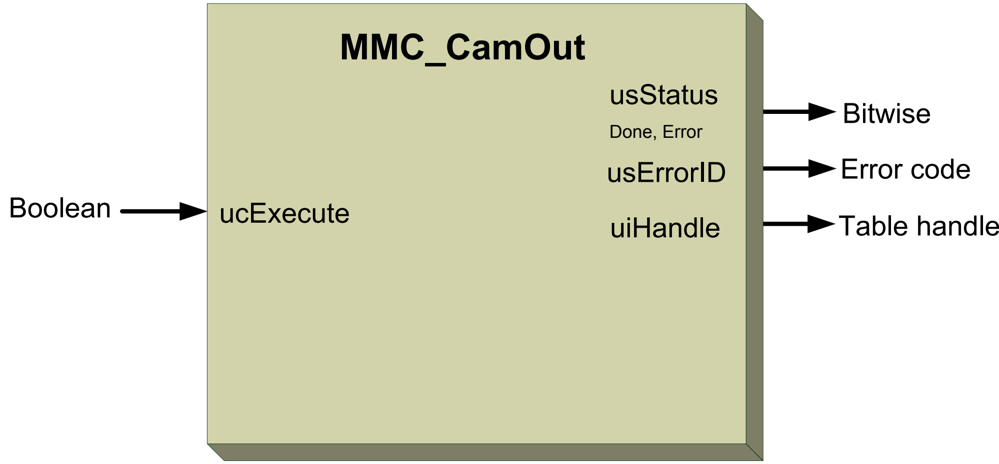


Figure 7-18: MMC_CamOutCmd function

7.6.8.2 Function Code Example

Refer to the example in section 7.7 Application Example.

7.6.9 MMC_CamStatus

MC_CamStatus retrieves the significant parameters of the CAM process.

```
MMC_LIB_API int MMC_CamStatusCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CAMSTATUS_IN* pInParam,
OUT MMC_CAMSTATUS_OUT* pOutParam);
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CAMSTATUS_IN** input data structure using the MMC_CamStatusCmd.

pOutParam

Points to the **MMC_CAMSTATUS_OUT** output structure receiving information, as a result of calling the MMC_CamStatusCmd.

Remarks

Be aware of the fact that this function operates on user space scheduling, while the CAM process is active as part of the internal schedule. This may cause some kind of uncertainty regarding data reliability.

Scope

All

MMC_CAMSTATUS_IN Structure

```
typedef struct _mmc_camstatus_in{
unsigned char ucDummy;
} MMC_CAMSTATUS_IN;
```

Parameters

ucDummy

Dummy parameters

MMC_CAMSTATUS_OUT Structure

```
typedef struct _mmc_camstatus_out{
unsigned long ulEndOfProfile;
unsigned long ulCurrentIndex;
unsigned long ulCycle;
unsigned short usStatus;
short sErrorID;
unsigned char ucSpare[32];
} MMC_CAMSTATUS_OUT;
```

Parameters

ulEndOfProfile

Counts the number of exits from CAM table. +ve number

ulCurrentIndex

Segment index in CAM table, which is currently processed. +ve number

ulCycle

Maestro cycle counter of the ECAM profiler. +ve number

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-18 describes the function for MMC_CamStatusCmd as applied within the IEC 61131 programming.

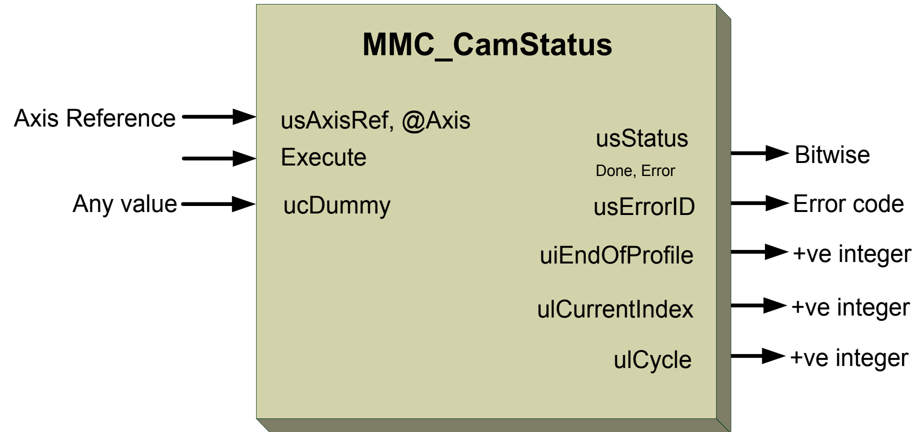


Figure 7-19: MMC_CamStatusCmd function

7.6.9.2 Function Code Example

Refer to the example in section 7.7 Application Example.

7.6.10 MMC_CamSetProperty

This function sets specific properties of the CAM function. It was created for a specific situation whereby the ECAM periodic motion is to be stopped using a non-periodic motion.

```
MMC_LIB_API int MMC_CamSetPropertyCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CAMSETPROP_IN* pInParam,
OUT MMC_CAMSETPROP_OUT* pOutParam);
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CAMSETPROP_IN** input data structure using the MMC_CamSetPropertyCmd.

pOutParam

Points to the **MMC_CAMSETPROP_OUT** output structure receiving information, as a result of calling the MMC_CamSetPropertyCmd.

Remarks

This function is comparible to the motion of wipers on the windscreen of a motor car. The motion is periodic. The driver then wishes to stop the motion of the wipers by introducing a non-periodic motion.

Scope

All

MMC_CAMSETPROP_IN Structure

```
typedef struct _mmc_camsetprop_in {
ECAM_PROPERTIES_ENUM eProperty;
ECAM_PROPERTY_VALUE value;
unsigned char ucExecute;
unsigned char ucSpare[32];
} MMC_CAMSETPROP_IN;
```

Parameters

ECAM_PROPERTIES_ENUM eProperty

Defines a non-periodic or periodic property of the ECAM. At this time, the only property to be defined is eCAM_PeriodicMode This is limited by the start modes:

ECAM_STARTMODE_BASE 0x00000000

ECAM_STARTMODE_RISING_EDGE 0x00000001

ECAM_PROPERTY_VALUE value

Defines the periodicity of the ECAM. This function is a union of several parameters defined by:

```
typedef union {
double dbVal;
ECAM_PERIODIC_ENUM ePeriodicMode;
CURVE_TYPE_ENUM eCurveType;
int iVal;
float fVal;
short sVal;
char cVal;
} ECAM_PROPERTY_VALUE;
```

dbVal

The value of the eProperty parameter, in this case, eCAM_PeriodicMode, as defined as a double 8 byte unlimited value

ECAM_PERIODIC_ENUM ePeriodicMode

The periodic or non-periodic mode is defined by the following enumerator ECAM_PERIODIC_ENUM:

eCAM_NON_PERIODIC = 0

eCAM_PERIODIC = 1

eCAM_LINEAR = 2

CURVE_TYPE_ENUM eCurveType

This enumerator ECAM_PERIODIC_ENUM defines the curve type according to:

eTableDefInterp = 0

eLinearInterp = 1

ePolynom5Interp = 2

ePolynom7Interp = 3

eCycloidPositionInterp = 4

eCycloidVelocityModified1Interp = 5

eCycloidVelocityModified2Interp = 6

iVal

The value of the eProperty parameter, in this case, eCAM_PeriodicMode, as defined as an integer 4 byte value

fVal

The value of the eProperty parameter, in this case, eCAM_PeriodicMode, as defined as a float 4 byte value

sVal

The value of the eProperty parameter, in this case, eCAM_PeriodicMode, as defined as a short 2 byte value

cVal

The value of the eProperty parameter, in this case, eCAM_PeriodicMode, as defined as a character 1 byte value

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

ucSpare[32]

Spare for future use

ucSpare[32]

Spare description for future use. +ve char value with maximum of 32 chars.

MMC_CAMSETPROP_OUT Structure

typedef struct _mmc_camsetprop_out {
unsigned short usStatus;
short sErrorID;
} MMC_CAMSETPROP_OUT;

Parameters

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-19 describes the function for MMC_CamSetPropertyCmd.

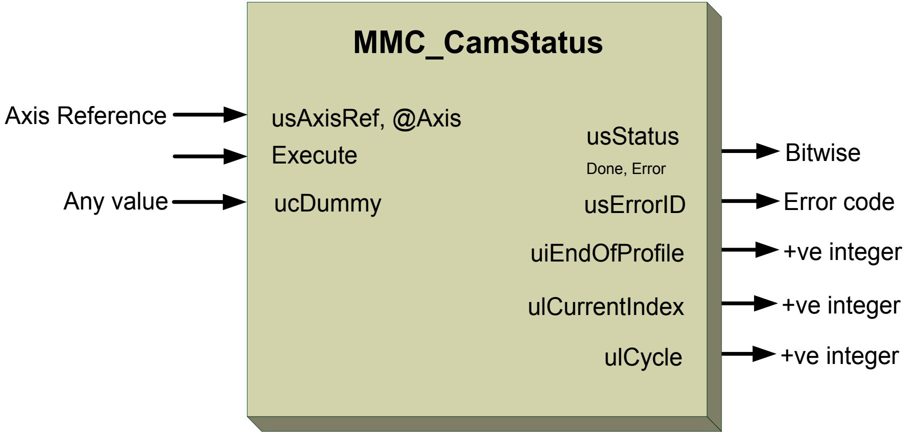


Figure 7-20: MMC_CamSetPropertyCmd function

7.6.10.2 Function Code Example

To be provided.

7.6.11 MMC_GearIn

Provides a command a to define the ratio between the Velocity of the slave and master axes.

This function is not supported at this moment.

```
MMC_LIB_API int MMC_GearInCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GEARIN_IN* pInParam,
OUT MMC_GEARIN_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not supported

Source GMAS\includes\MMC_PVT_ECAM_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_GEARIN_IN** input data structure using the MMC_GearInCmd.

pOutParam

Points to the **MMC_GEARIN_OUT** output structure receiving information, as a result of calling the MMC_GearInCmd.

Remarks

Scope

All

MMC_GEARIN_IN Structure

```
typedef struct _mmc_gearin_in {
unsigned short usMaster;
int iRatioNumerator;
int iRatioDenominator;
ECAM_VALUE_SRC_ENUM eMasterValueSource;
unsigned int uiSyncMode;
double dbAcceleration;
double dbDeceleration;
double dbJerk;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute
unsigned char ucSpare[32];
} MMC_GEARIN_IN;
```

Parameters

usMaster

Axis reference of master. 2 byte +ve value.

iRatioNumerator

Gear ratio numerator. +ve integer

iRatioDenominator

Gear ratio denominator. +ve integer

ECAM_VALUE_SRC_ENUM eMasterValueSource

Defines the enumerator ECAM_VALUE_SRC_ENUM source for synchronization:

mcSetValue - Synchronization on master set value

mcActualValue - Synchronization on master actual value

uiSyncMode

Defines the method of synchronization, e.g. mcShortest, mcCatchUp, or mcSlowDown. Is vendor specific - for future use

dbAcceleration

Acceleration for the gearing in. 8 byte unlimited value.

dbDeceleration

Deceleration for the gearing in. 8 byte unlimited value.

dbJerk

Jerk of the Gearing in. 8 byte unlimited value.

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows, but only the Buffered Mode is supported:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command by starting the gearing process at the rising edge. Boolean TRUE/FALSE values.

ucSpare[32]

Spare for future use

MMC_GEARIN_OUT Structure

```
typedef struct _mmc_gearin_out {
unsigned int uiHandle;
unsigned short usStatus;
short sErrorID;
} MMC_GEARIN_OUT;
```

Parameters

uiHandle

Table access handler.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-20 describes the function block for MMC_GearInCmd.

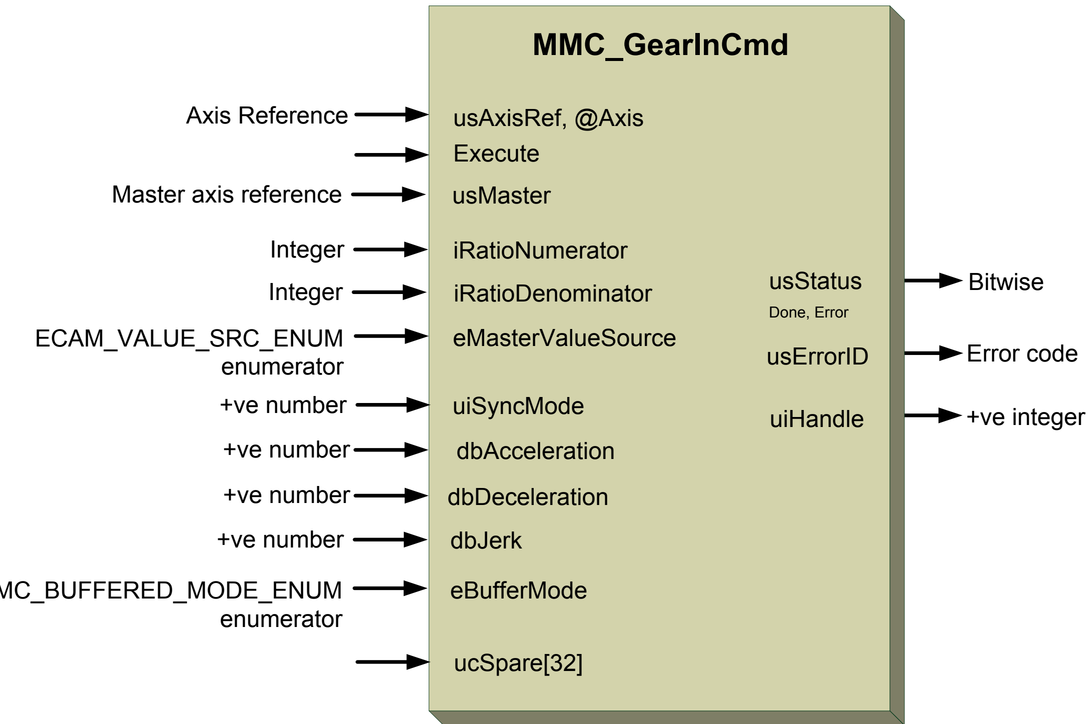


Figure 7-21: MMC_GearInCmd function

7.6.11.2 Function Code Example

To be provided.

7.6.12 MMC_GearInPos

Provides a command to define the gear ratio between the position of the slave and master axes from synchronization point onwards.

This function is not supported at this moment.

```
MMC_LIB_API int MMC_GearInPosCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GEARINPOS_IN* pInParam,
OUT MMC_GEARINPOS_OUT* pOutParam
);
```

Motion Mode	NC – Supported	Distributed – Not supported
Source	GMAS\includes\MMC_PVT_ECAM_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_GEARINPOS_IN** input data structure using the MMC_GearInPosCmd.

pOutParam

Points to the **MMC_GEARINPOS_OUT** output structure receiving information, as a result of calling the MMC_GearInPosCmd.

Remarks

Scope

All

MMC_GEARINPOS_IN Structure

```
typedef struct _mmc_gearinpos_in {
unsigned short usMaster;
double dbMasterStartDistance;
double dbMasterSyncPosition;
double dbSlaveSyncPosition;
int iRatioNumerator;
int iRatioDenominator;
ECAM_VALUE_SRC_ENUM eMasterValueSource;
unsigned int uiSyncMode;
double dbVelocity;
double dbAcceleration;
double dbDeceleration;
double dbJerk;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
unsigned char ucSpare[32];
} MMC_GEARINPOS_IN;
```

Parameters

usMaster

Axis reference of master. 2 byte +ve value.

dbMasterStartDistance

Master distance for the gear in procedure i.e. distance for ramp-in when the slave axis starts to become synchronized. +ve number

dbMasterSyncPosition

Position of the master when the slave is synchronized with the master.

dbSlaveSyncPosition

Slave position when the axes are running in synchronization. +ve number

iRatioNumerator

Gear ratio numerator. +ve integer

iRatioDenominator

Gear ratio denominator. +ve integer

ECAM_VALUE_SRC_ENUM eMasterValueSource

Defines the enumerator ECAM_VALUE_SRC_ENUM source for synchronization:

mcSetValue - Synchronization on master set value

mcActualValue - Synchronization on master actual value

uiSyncMode

Defines the method of synchronization, e.g. mcShortest, mcCatchUp, or mcSlowDown. Is vendor specific - for future use

dbVelocity

Velocity for the gearing in. 8 byte unlimited value.

dbAcceleration

Acceleration for the gearing in. 8 byte unlimited value.

dbDeceleration

Deceleration for the gearing in. 8 byte unlimited value.

dbJerk

Jerk of Gearing. 8 byte unlimited value.

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows, but only the Buffered Mode is supported:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

ucSpare[32]

Spare for future use

MMC_GEARINPOS_OUT Structure

```
typedef struct _mmc_gearinpos_out {
unsigned int uiHandle;
unsigned short usStatus;
short sErrorID;
} MMC_GEARINPOS_OUT;
```

Parameters

uiHandle

Table access handler.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-21 describes the function block for MMC_GearInPosCmd.

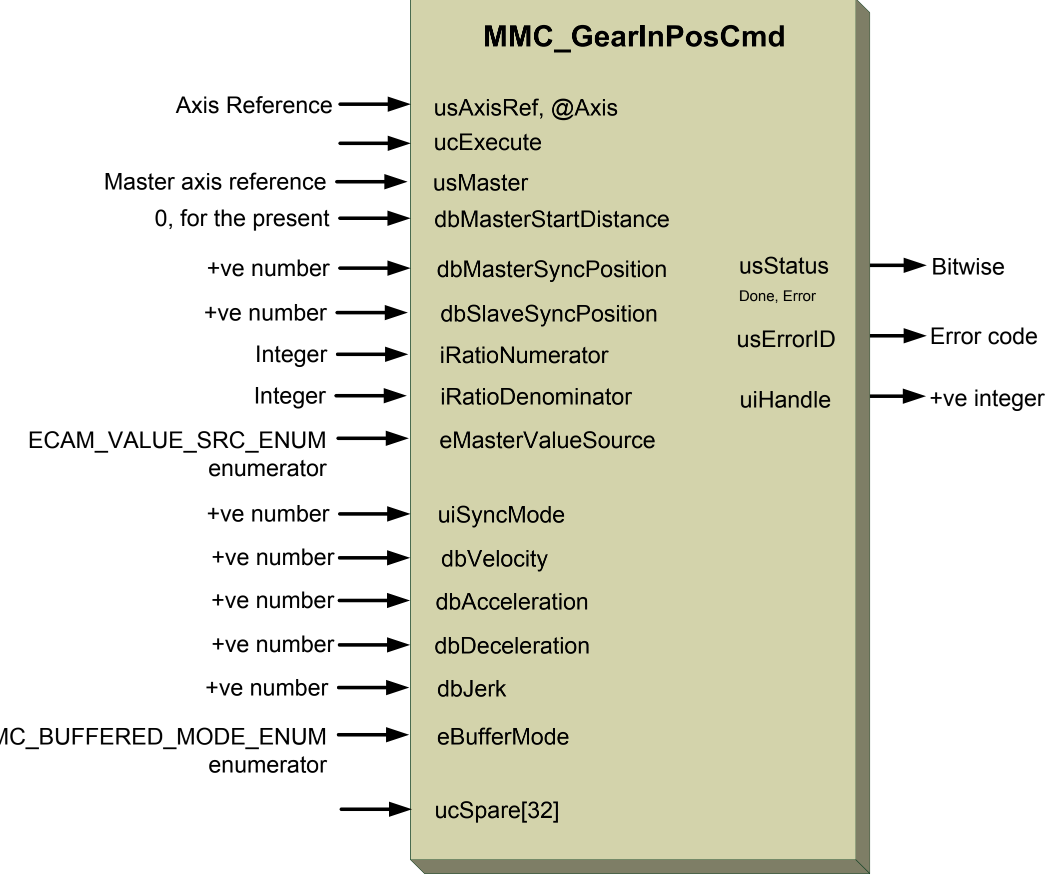


Figure 7-22: MMC_GearInPosCmd function

7.6.12.2 Function Code Example

To be provided.

7.6.13 MMC_GearOut

Provides a command to disengage a gear between slave and master axes, actually MC_Stop at this stage.

This function is not supported at this moment.

```
MMC_LIB_API int MMC_GearOutCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GEAROUT_IN* pInParam,
OUT MMC_GEAROUT_OUT* pOutParam
);
```

Motion Mode	NC – Supported	Distributed – Not supported
Source	GMAS\includes\MMC_PVT_ECAM_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command.

pInParam

Points to the **MMC_GEAROUT_IN** input data structure using the MMC_GearOutCmd.

pOutParam

Points to the **MMC_GEAROUT_OUT** output structure receiving information, as a result of calling the MMC_GearOutCmd.

Remarks

Scope

All

MMC_GEAROUT_IN Structure

typedef struct _mmc_gearout_in {
unsigned char ucExecute
} MMC_GEAROUT_IN;

Parameters

ucExecute

Start the execution command by starting the disengaging process at the rising edge. Boolean TRUE/FALSE values.

MMC_GEAROUT_OUT Structure

typedef struct _mmc_gearout_out {
unsigned int uiHandle;
unsigned short usStatus;
short sErrorID;
} MMC_GEAROUT_OUT;

Parameters

uiHandle

Table access handler.

uiEndOfProfile

Variable counter increases each time the master meets the end of the table. Boolean for IEC. Count variable increases each time the master meets the end of the table.

usStatus

Bitwise returned command status with the values MMC_REMOTE_FUNC_STATUS_BIT_ERROR or 0 (OK).

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 7-22 describes the function block for MMC_GearOutCmd.

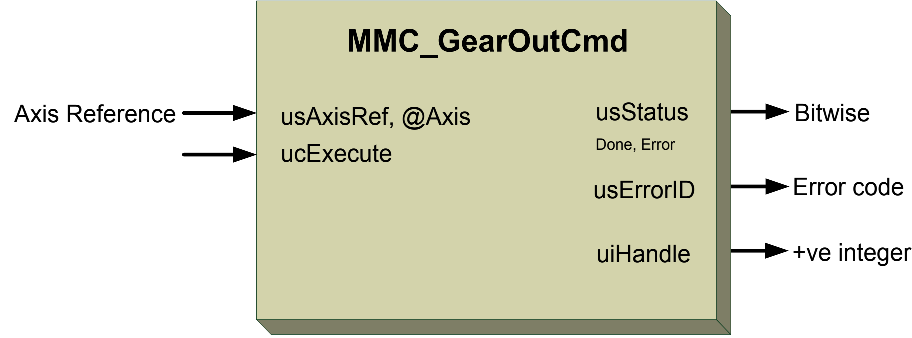


Figure 7-23: MMC_GearOutCmd function

7.6.13.2 Function Code Full Example

To be provided.

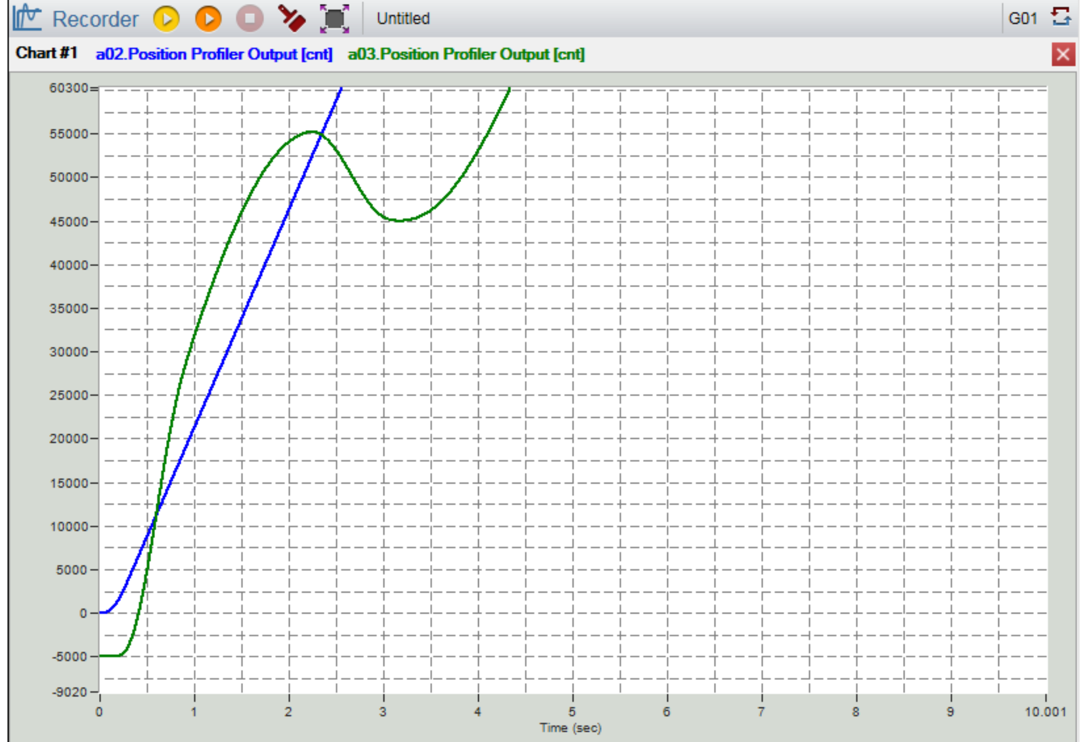
7.7 Application Example

For the follows examples consider CAM table as follows:

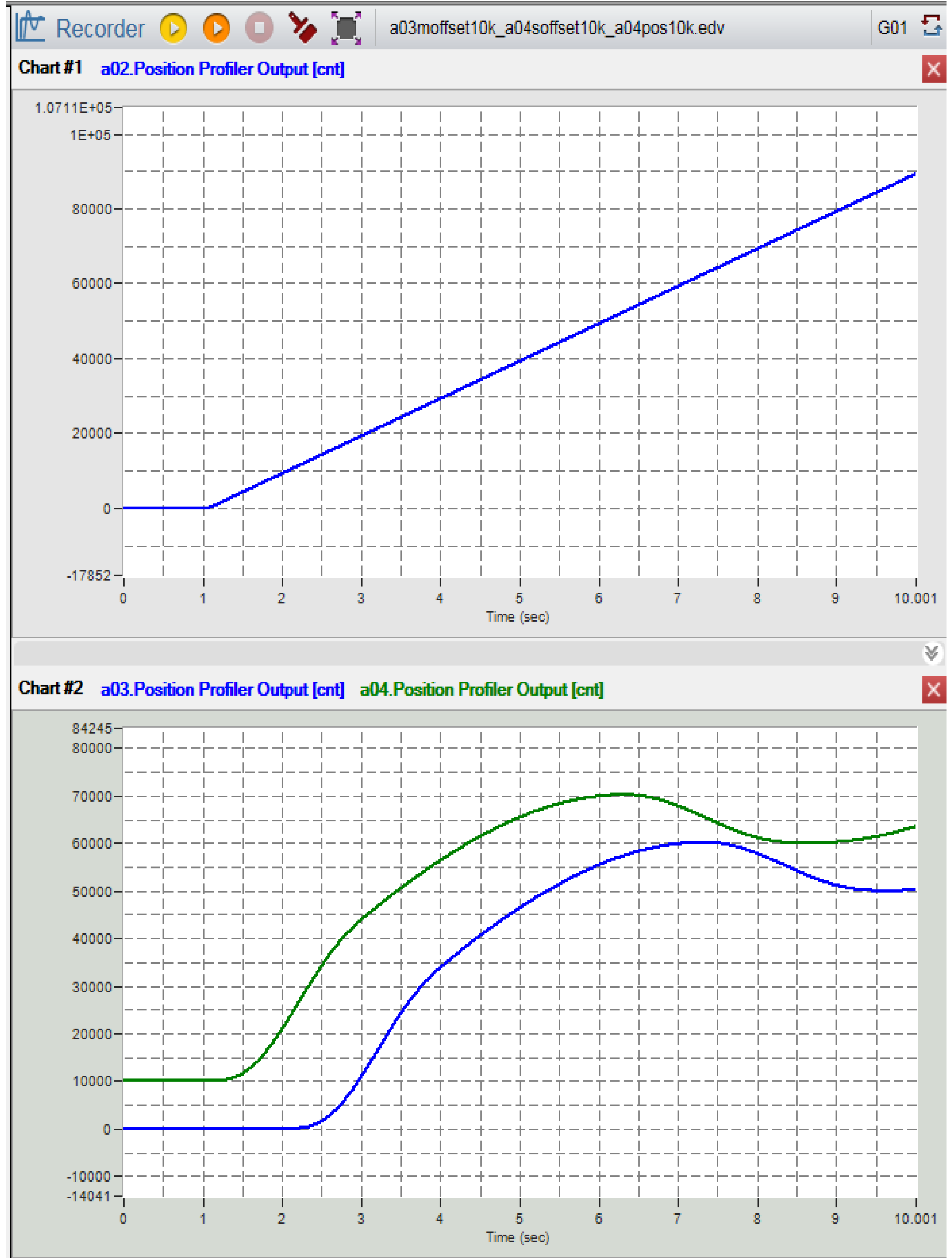
ECAM version 1 0
ECAM mode 46
ECAM dimension 1
ECAM num of pts 10
ECAM cyclic 0
ECAM pos absolute 1
ECAM fixed gap 0
ECAM data start
0.0000000, 0.000000,
20000.000000, 35000.000000,
50000.000000, 60000.000000,
75000.000000, 50000.000000,
120000.000000, 80000.000000,
144000.000000, 110000.000000,
165000.000000, 122000.000000,
185000.000000, 145000.000000,
200000.000000, 160000.000000,
220000.000000, 185000.000000
ECAM data end

7.7.1 Master/Slave Offset

The User may define an extra offset for master or slave positions in the CAM table. The CAM process uses the same offset as specified for each master (or slave) position respectively (see the examples below). The Auto-offset overrides the user defined slave offset.



The participant axes are the master axis a02 (blue line in chart) and slave axis a03 (black line in chart).
The above CAM Table is defined as absolute (by MC_CamTableSelect). The CAM process in this example runs with slave offset -5000 and slave position in first row is zero. So the slave was moved in advance to -5000 in order to avoid the jump, producing a smooth curve of the CAM process.
Another example is the following.

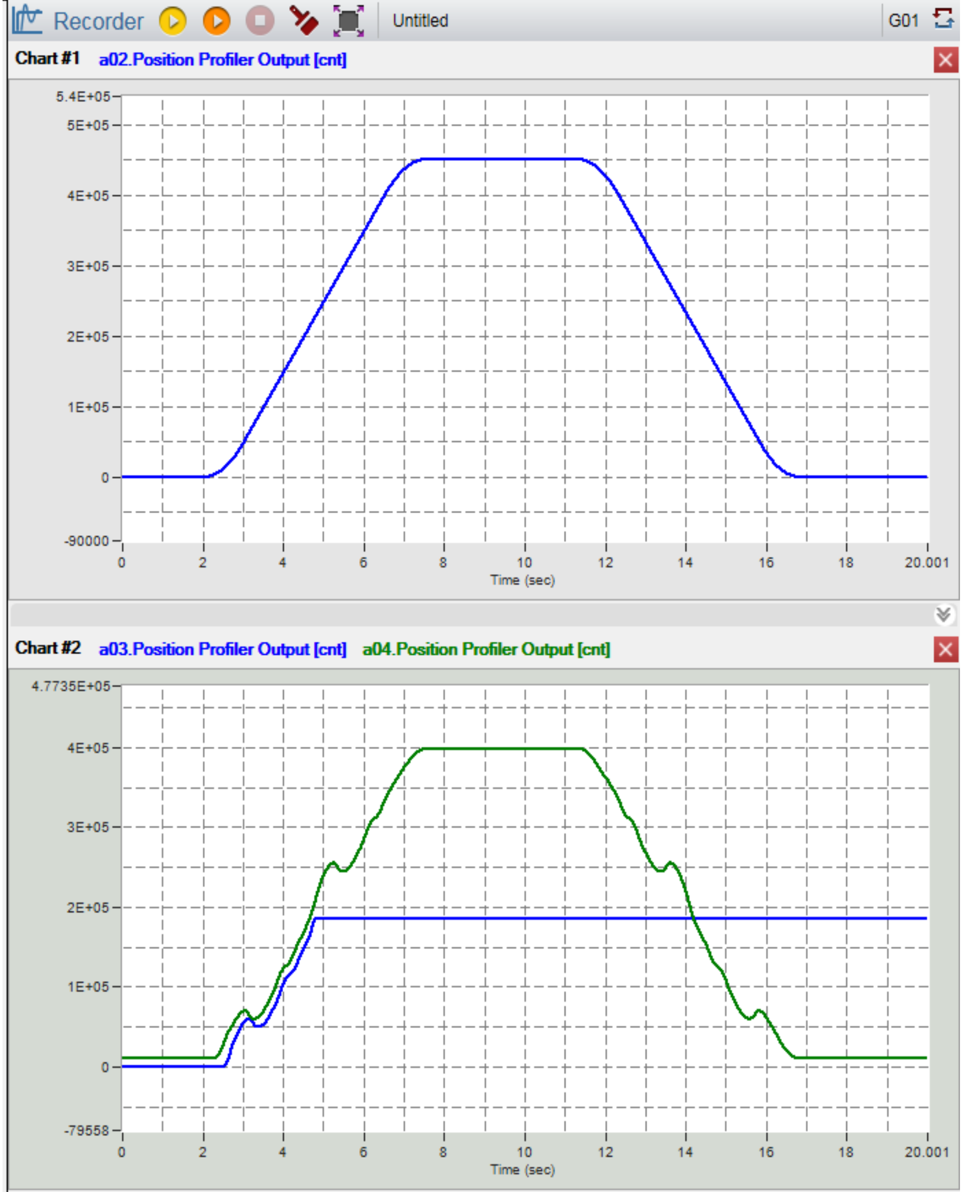


The participant axes are the master axis a02 (the upper chart), slave axis a03, and slave axis a04 (the lower chart).
The CAM table is an absolute table and the first phase is 0 (see above). The function MC_CamIn for the Slave a03 (blue) is run with the *Master* offset at 10000. The Slave a04 (green) at engagement time, is in position 10000 and therefore MC_CamIn on that slave will run with a slave offset 10000 in order to avoid the leap.

7.7.2 Periodicity Modes of Operation

7.7.2.1 None Periodic

The non-periodic mode is also known as one shot execution. The CAM process starts when the master is inserted from the CAM table and ends the first time it exits the table (no matter from which direction).



In the above example the participant axes arethe master axis a02 (the upper chart), slave axis a03 and slave axis a04 (the lower chart).

The CAM process on a03 (blue) runs as one-shot (none periodic) and on a04 as periodic process. Master a02 of this example moves from 0 to 450000, then stops momentory and then moves back to zero. From the graph slave a03 appears to be engaged as long as the master passes along the CAM table and is disengaged when it exits the table for the first time. The other CAM process on a04 proceeds for ever until it is disengaged by MC_CamOut or MC_Stop.

7.7.2.2 Periodic

The Periodic mode is a continuous process. The Master may enter or exit the table range frequently in both directions. Disengagement occurs when CAM process is aborted by MC_Stop or MC_CamOut. On periodic mode master position is modulated as if it never exits the table range. In order to achieve a smooth curve each cycle of the master, an additional full range slave column is added to the slave calculated position.

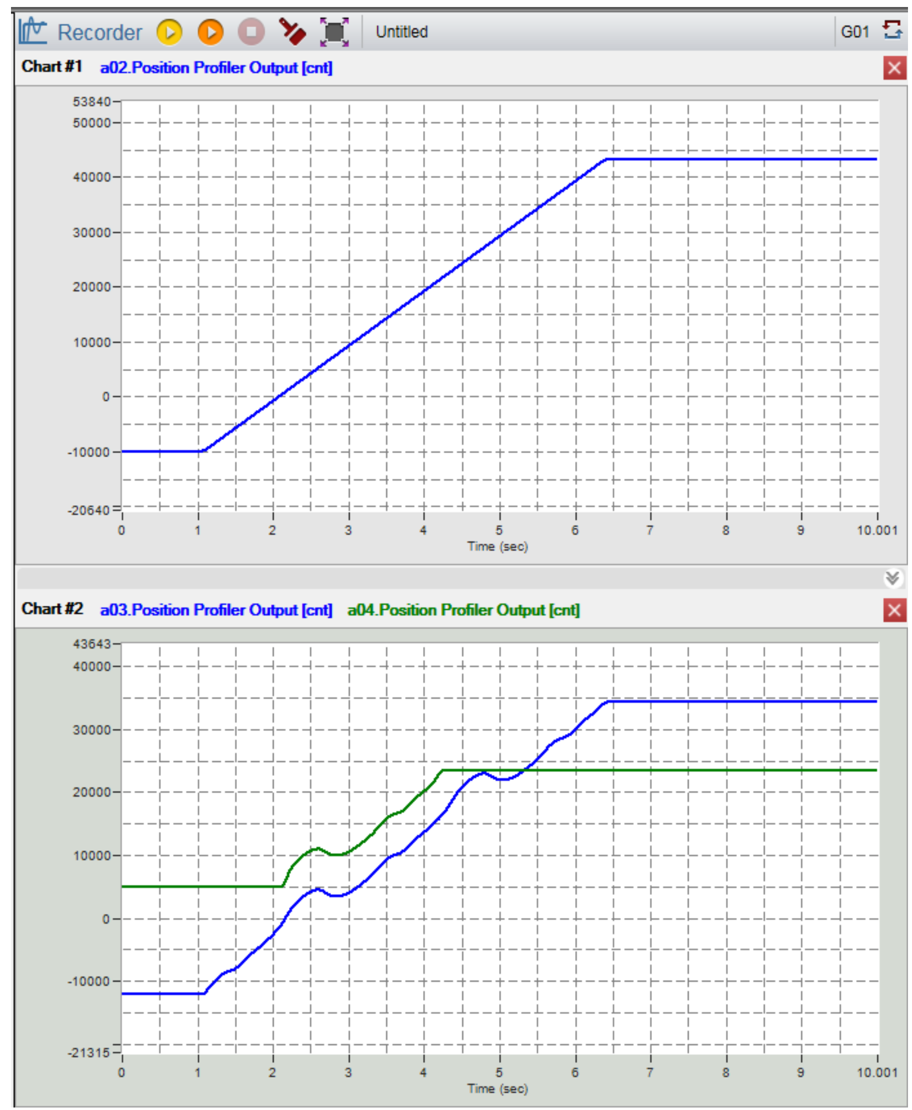
7.7.2.3 Periodic Linear

Periodic linear is an endless CAM process. Engagement starts in normal conditions. Decoupling occurs when the master exits the table. The CAM process is not completed on decoupling and proceeds until aborted by MC_Stop (or MC_CamOut). The Master may enter and exit the CAM table in both directions, so that coupling and decoupling may occur many times as part of this CAM process (see examples below).

7.7.2.4 Periodic vs. Periodic Linear

The graph below uses the following data to demonstrate the difference between periodic CAM process and periodic-linear.

Axis	Role	Start Position	End Position	Periodic	Velocity	Auto Offset	Scaling	Relative
a02	Master	-10000	43200	-	10000	-	1	0
a03	Slave	-1500	-	Periodic	-	1	0.1	0
a04	Slave	5000	-	Periodic Linear	-	1	0.1	0



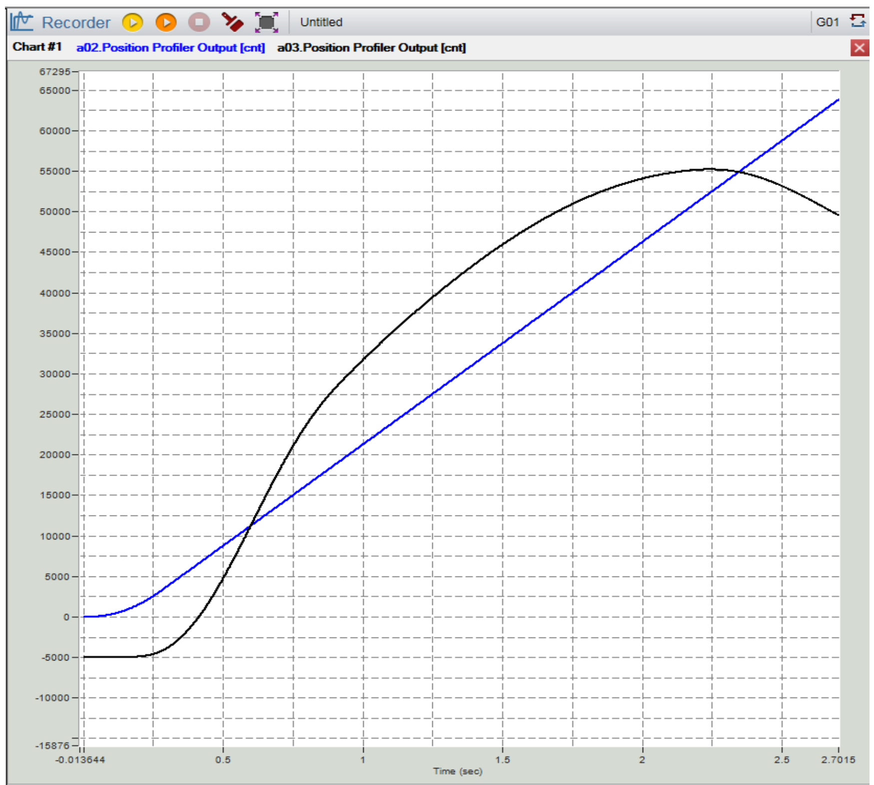
It should be noted that both slaves engage the same master (a02) and process the same table. Shown in the graph is a04 which is coupled with a02 (master) only within the CAM table range, while a03 behaves as if it never exits the CAM table and continues motion until the master stops.

7.7.2.5 Auto Offset

Auto offset mode refers to slave position and makes sure that slave axis does not jump when engaged. At the time when master reaches Master Sync Position (see definition below) the offset between slave axis position and slave position in the table is determined automatically. From that moment onwards, all entries will be adjusted in accordance with this offset.

Auto OFF adjust slave offsets with respect to user define offset only.

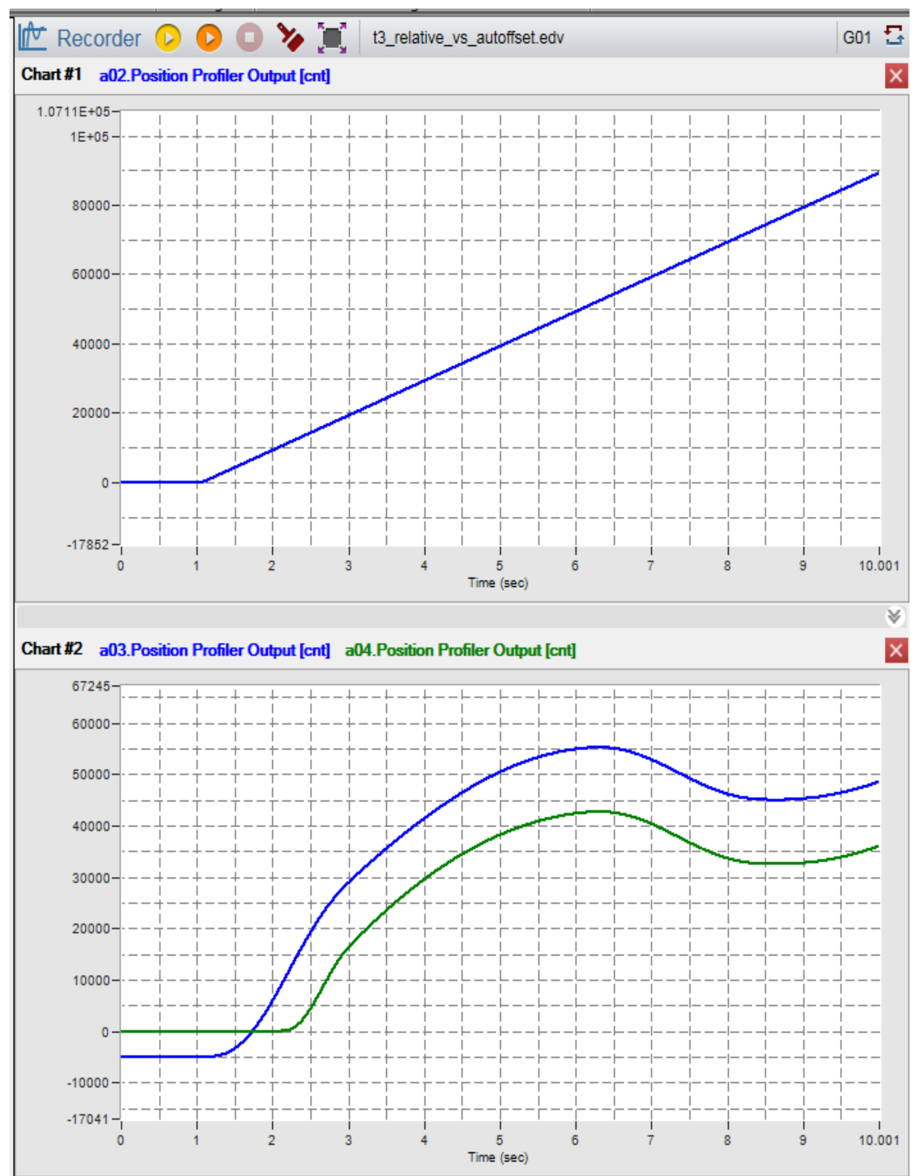
Auto ON adjust slave offsets automatically with respect to slave axis position on sync position. This mode overrides and replaces user definition for slave offset.



The participating axes are the master axis a02 (blue in chart), slave axes a03 (black in chart). It should be noted that the slave axis position on CAM activation is -5000, Sync Position is 0 (first row of CAM table). In spite of the gap between slave axis position (-5000) and table position in first row (0) the slave motion is smooth starting from -5000.

7.7.2.6 Auto Offset vs. Relative Table

Auto Offset parameter is relevant to the slave column only while the relative column can be defined for master as well as for slave. When MC_CamIn operates on relative table it adds the row position in table to the axis position on CAM activation (activation time, not necessary the coupling position). In other words, the table is shifted relatively to the axis position. When Auto Offset parameter is on, MC_CamIn adjusts the table position at the coupling point to the slave position at that time and all other slave positions in table are adjusted by the corresponding offset (see the charts below).



The participating axes are the master axis a02 (the upper chart), slave axes a03, and a04 (the lower chart). For both slaves operating in the relative table.

- Maser a02 moves from 0 to 450000 by speed of 10000.
- Slave a03: coupling point defined as the first phase of the table (default behavior), which is zero in this case. Axis position on CAM activation is -5000. The plot graph shows the slave a03 which is simply the table position minus 5000.
- Slave a04:
  - Coupling point defined as 10000 (somewhere within the first segment) by dbMasterSyncPosition
  - Slave position according to CAM table is positive value (somewhere on the line between 0 to 135000) high above zero
  - Slave axis position at engagement time is zero
  - Auto offset is turned on
  - All points in table are adjusted automatically by an offset, which is set at the coupling point when it processed. Slave axis motion is smooth right from its position at the coupling point.

Chapter 8:API Services and Operations

This chapter describes the API services and operations for the Maestro, and involves the following:

- Main configuration variables
- Maestro Preoperational Mode. Refer to **Chapter 2: Maestro Overview** section for further details
- EtherCAT Configuration Mode. Refer to **Chapter 2: Maestro Overview**, section for further details
- Data Recording. Refer to section **9.9 PI Functions and Implementation Examples**
- for further details
- Resource file uploading and downloading
- Download new firmware version

The following main configuration function blocks are described, where MMC_Connection_Param_Struct is an administrative function only:

Function Block	Services and Operation
MMC_InitConnection	Main configuration variables
MMC_IPCInitConnection	
MMC_RpcInitConnection	
MMC_CloseConnection	
MMC_CmdStatus	
MMC_Config	
MMC_Exit	
MMC_FreeFbStat	
MMC_GetAxisByName	
MMC_GetGroupByName	
MMC_GetVersion	
MMC_ResetMultiAxisControl	
MMC_SaveParam	
MMC_ShowNodeStat	
MMC_ClearNodeFbList	
MMC_CloseConnection	
MMC_CreateSYNCTimer	
MMC_DestroySYNCTimer	
MMC_DownloadFoE	
MMC_Dwell	

MMC_GetActiveVectorsNum	
MMC_GetErrorCodeDescriptionByID	
MMC_GetFoEStatus	
MMC_GetEthercatCommStatistics	
MMC_GetEnquireFbStatus	
MMC_GetGroupMembersInfo	
MMC_GetStatusRegister	
MMC_GetVersionEx	
MMC_LoadParam	
MMC_RpcInitConnectionEx	
MMC_SetEnquireFbStatus	
MMC_SetDefaultParameters	
MMC_SetDefaultParametersGlobal	
MMC_SetIsToLoadGlobalParams	
MMC_GetActiveAxesNum	
MMC_ToggleConsoleOutput	
MMC_GetCyclesCounter	
MMC_WriteGroupOfParameters	
MMC_ReadGroupOfParameters	
MMC_WaitUntilConditionFB	
MMC_ChangeToPreOPMode	Maestro Preoperational Mode
MMC_ChangeToOperationMode	
GetGMASOperationMode	
MMC_GetResList	Resource file variables involving list, snapshot, export, and import.
MMC_GetResSnapshot	
MMC_ResExportFile	
MMC_ResImportFile	
MMC_GetVerPath	
MMC_DownloadVersion	
MMC_ReadDownloadVersionStatus	
MMC_SetVerPath	

8.1 Main Configuration Function Blocks

The following main configuration function blocks are described:

Main Configuration
MMC_ChangeToPreOPMode
MMC_ChangeToOperationMode
MMC_ClearNodeFbList
MMC_CmdStatus
MMC_CloseConnection
MMC_Config
MMC_CreateSYNCTimer
MMC_DestroySYNCTimer
MMC_DownloadFoE
MMC_Exit
MMC_FreeFbStat
MMC_GetActiveVectorsNum
MMC_GetErrorCodeDescriptionByID
MMC_GetFoEStatus
MMC_GetEnquireFbStatus
MMC_GetAxisByName
MMC_GetGroupByName
MMC_GetGroupMembersInfo
MMC_GetGMASOperationMode
MMC_GetStatusRegister
MMC_GetResList
MMC_GetResSnapshot
MMC_GetVersion
MMC_GetVersionEx
MMC_InitConnection
MMC_IPCInitConnection
MMC_LoadParam
MMC_RpcInitConnection
MMC_RpcInitConnectionEx

Main Configuration
MMC_ResetMultiAxisControl
MMC_ResExportFile
MMC_ResImportFile
MMC_SaveParam
MMC_SetEnquireFbStatus
MMC_SetDefaultParameters
MMC_SetDefaultParametersGlobal
MMC_SetIsToLoadGlobalParams
MMC_ShowNodeStat
MMC_GetActiveAxesNum
MMC_ToggleConsoleOutput
MMC_GetCyclesCounter
MMC_WriteGroupOfParameters
MMC_WriteGroupOfParametersEx
MMC_ReadGroupOfParameters
MMC_WaitUntilConditionFB
MMC_WaitUntilConditionFBEx
MMC_WriteMemoryRange
MMC_ReadMemoryRange
MMC_SetDefaultResources
MMC_KillRepetitive
MMC_UserCommandControl
MMC_GetVerPath
MMC_DownloadVersion
MMC_ReadDownloadVersionStatus
MMC_SetVerPath

8.1.1 MMC_ChangeToPreOPMode

Changes the Maestro to preoperational mode.

```
MMC_LIB_API int MMC_ChangeToPreOPMode(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SET_GMAS_PREOP_IN* pInParam
OUT MMC_SET_GMAS_PREOP_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SET_GMAS_PREOP_IN** input data structure using the MMC_ChangeToPreOPMode function.

pOutParam

Points to the **MMC_SET_GMAS_PREOP_OUT** output structure receiving information, as a result of calling the MMC_ChangeToPreOPMode function.

Remarks

None

Scope

All

MMC_SET_GMAS_PREOP_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_SET_GMAS_PREOP_IN;
```

Parameters

dummy

Dummy input. Any +ve character value.

MMC_SET_GMAS_PREOP_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SET_GMAS_PREOP_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-1 describes the function block for MMC_ChangeToPreOPMode.

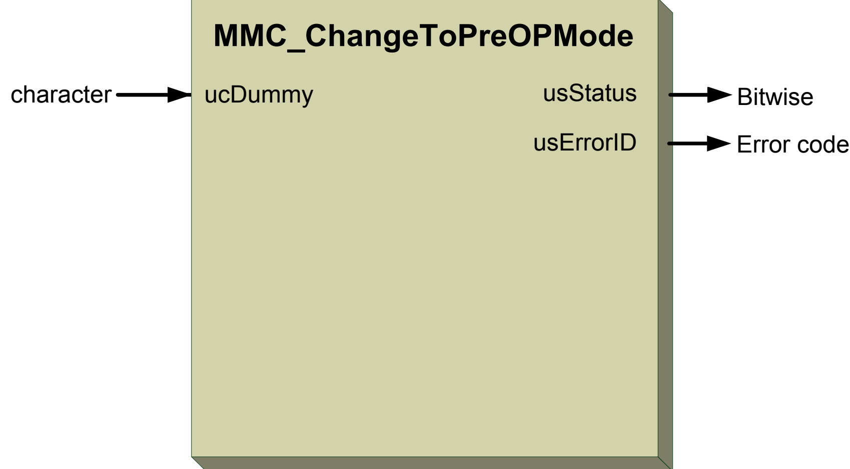


Figure 8-1: MMC_ChangeToPreOPMode function block

8.1.1.2 Function Block Code Example

```
int rc;
MMC_SET_GMAS_PREOP_IN      stSetGMASPreOp_in;
MMC_SET_GMAS_PREOP_OUT     stSetGMASPreOp_out;
//
// Inserting the structure parameters:
stSetGMASPreOp_in.ucDummy       = 1;      // Dummy input
//
rc = MMC_ChangeToPreOPMode (hConn, &stSetGMASPreOp_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.2 MMC_ChangeToOperationMode

Changes the Maestro to operational mode.

```
MMC_LIB_API int MMC_ChangeToOperationMode(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SET_GMAS_OP_IN* pInParam
OUT MMC_SET_GMAS_OP_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SET_GMAS_OP_IN** input data structure using the MMC_ChangeToOperationMode function.

pOutParam

Points to the **MMC_SET_GMAS_OP_OUT** output structure receiving information, as a result of calling the MMC_ChangeToOperationMode function.

Remarks

None

Scope

All

MMC_SET_GMAS_OP_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_SET_GMAS_OP_IN;
```

Parameters

dummy

Dummy input. Any +ve character value.

MMC_SET_GMAS_OP_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SET_GMAS_OP_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-2 describes the function block for MMC_ChangeToOperationMode.

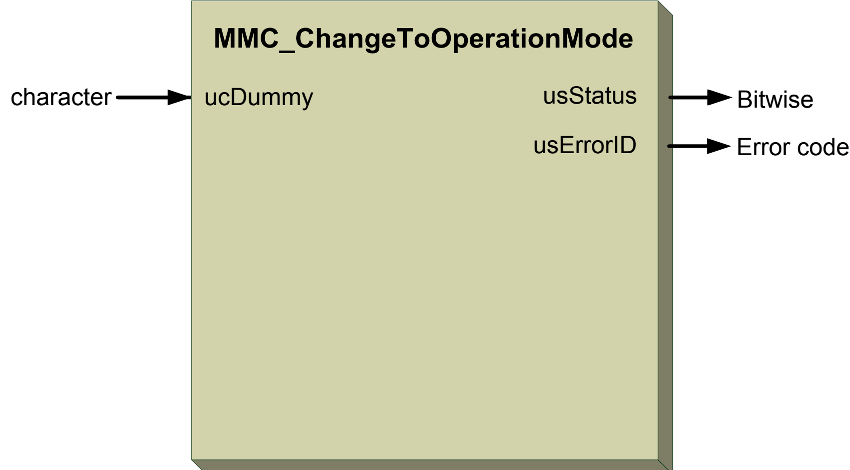


Figure 8-2: MMC_ChangeToOperationMode function block

8.1.2.2 Function Block Code Example

```
int rc;
MMC_SET_GMAS_OP_IN          stSetGMASOp_in;
MMC_SET_GMAS_OP_OUT         stSetGMASOp_out;
//
// Inserting the structure parameters:
stSetGMASOp_in.ucDummy      = 1;      // Dummy input
//
rc = MMC_ChangeToOperationMode (hConn, &stSetGMASOp_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.3 MMC_ClearNodeFbList

This adds the ability to clear the function block list of a specific node, i.e. either Axis or Group.This can only be performed if the node is not in a moving state.

```
int MMC_ClearNodeFbListCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_CLEARFBLIST_IN* pInParam,
OUT MMC_CLEARFBLIST_OUT* pOutParam
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CLEARFBLIST_IN** input data structure using the MMC_ClearNodeFbList function.

pOutParam

Points to the **MMC_CLEARFBLIST_OUT** output structure receiving information, as a result of calling the MMC_ClearNodeFbList function.

Remarks

Refer to the use of the function in section **8.1.12 MMC_GetActiveVectorsNum on page 800**.

Scope

All

MMC_CLEARFBLIST_IN Structure

```
typedef struct mmc_clearfblist_in{
unsigned short usAxisRef;
}MMC_CLEARFBLIST_IN;
```

Parameters

usAxisRef

Axis reference. Any +ve bitwise integer.

MMC_CLEARFBLIST_OUT Structure

```
typedef struct mmc_clearfblist_out{
unsigned short usStatus;
short sErrorID;
}MMC_CLEARFBLIST_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-3 describes the function block for MMC_ClearNodeFbList

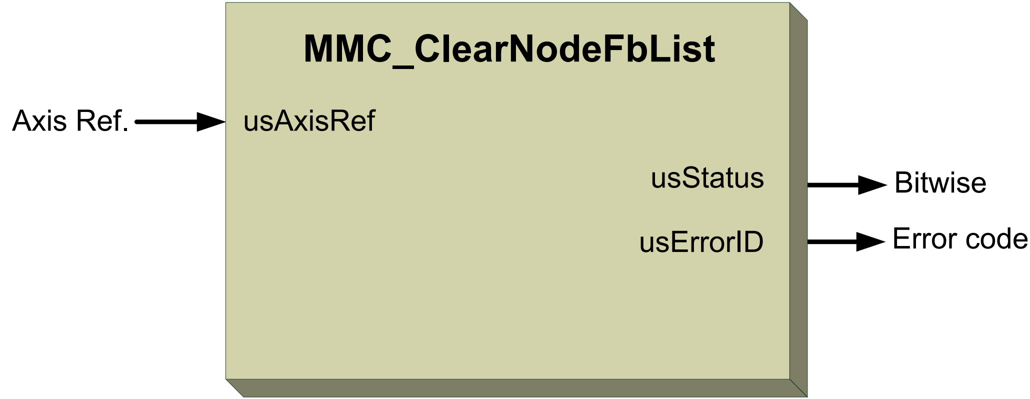


Figure 8-3: MMC_ClearNodeFbList function block

8.1.4 MMC_CmdStatus

Sends a Read Function Block Status command to the Maestro server for specific Axis/Group and receive status back.

```
MMC_LIB_API int MMC_CmdStatus(
IN MMC_CONNECT_HNDL hConn,
IN MMC_FBSTATUS_IN* pInParam,
OUT MMC_FBSTATUS_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_FBSTATUS_IN** input data structure using the MMC_CmdStatus function.

pOutParam

Points to the **MMC_FBSTATUS_OUT** output structure receiving information, as a result of calling the MMC_CmdStatus function.

Remarks

None

Scope

All

MMC_FBSTATUS_IN Structure

```
typedef struct{
unsigned int uiHndl;
}MMC_FBSTATUS_IN;
```

Parameters

uiHndl

Function block handle. Any +ve integer value

MMC_FBSTATUS_OUT Structure

```
typedef struct{
unsigned int uiFbStatus;
unsigned short usStatus;
short sErrorID;
unsigned short usFbErrorID;
}MMC_FBSTATUS_OUT;
```

Parameters

uiFbStatus

Returned function block status. Any +ve integer bitwise value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

usFbErrorID

Returned function block error ID. Signals where a function block error occurs. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-4 describes the function block for MMC_CmdStatus

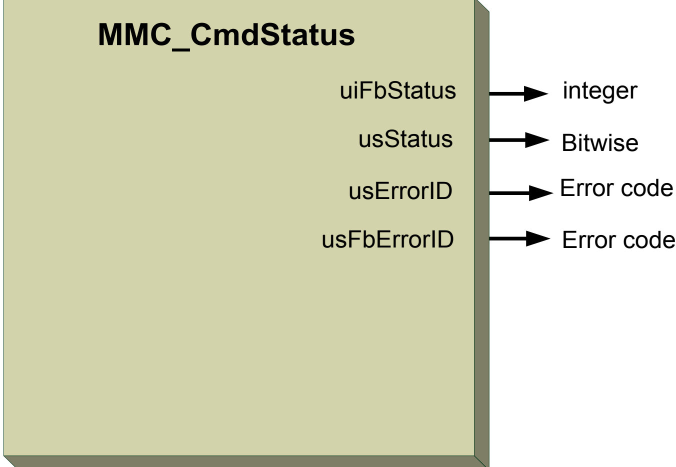


Figure 8-4: MMC_CmdStatus function block

8.1.4.2 Function Block Code Example

```
int rc;
MMC_FBSTATUS_IN        stFBStatus_in;
MMC_FBSTATUS_OUT       stFBStatus_out;
//
// Inserting the structure parameters:
stFBStatus_in.uiHndl   = 1;          //Function block handle
//
rc = MMC_CmdStatus (hConn, &stFBStatus_in, &stFBStatus_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.5 MMC_CloseConnection

Closes the connection to the Maestro.

```
MMC_LIB_API int MMC_CloseConnection(
IN MMC_CONNECT_HNDL hConn
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

Remarks

None

Scope

The input parameter *hConn* is dependent on the connection handle value created when performing the InitConnection function. This value should therefore be retained with other connection handle thread values for use when a connection is to be closed.

Figure 8-5 describes the function block for MMC_CloseConnection

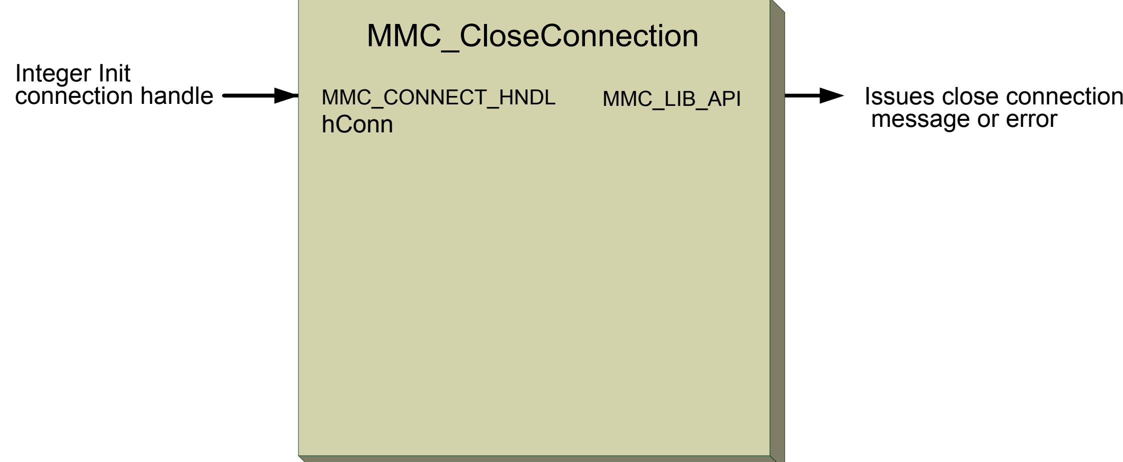


Figure 8-5: MMC_CloseConnection function block

8.1.5.1 Function Block Code Example

```
int rc;
//
// Inserting the structure parameters:
hConn         = 1 ;         // Connection Handle Type. Number from the connection handle
//
rc = MMC_CloseConnection (hConn);
printf("Connection State[%ld]\n", (long int)(MMC_CONNECT_HNDL) hConn);
if (rc != 0)
printf("ERROR:%s: MMC_CloseConnection fail\n", __func__);
{
    HandleError();
}
```

8.1.6 MMC_Config

Set the Maestro to configuration mode and allow changes to any configuration parameters.

```
MMC_LIB_API int MMC_ConfigCmd
(IN MMC_CONNECT_HNDL hConn,
IN MMC_CONFIG_IN* pInParam,
OUT MMC_CONFIG_OUT* pOutParam
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CONFIG_IN** input data structure using the MMC_Config function.

pOutParam

Points to the **MMC_CONFIG_OUT** output structure receiving information, as a result of calling the MMC_Config function.

Remarks

There are two Maestro operational modes:

- Normal
- Configuration

When any communication configuration parameters (network IP etc.) are changed using the Set command, this function is invoked to exit the configuration mode to the normal operational mode of the Maestro.

Scope

All

MMC_CONFIG_IN Structure

```
typedef struct mmc_config_in{
unsigned char dummy;
}MMC_CONFIG_IN;
```

Parameters

dummy

Dummy character. Any +ve value accepted.

MMC_CONFIG_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIG_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-6 describes the function block for MMC_Config.

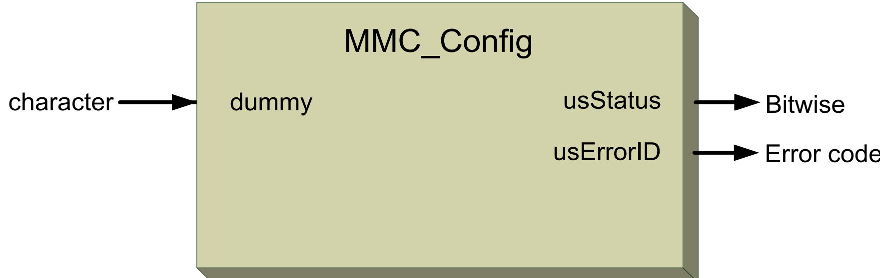


Figure 8-6: MMC_Config function block

8.1.6.2 Function Block Code Example

```
int rc;
MMC_CONFIG_IN      stConfig_in;
MMC_CONFIG_OUT     stConfig_out;
//
// Inserting the structure parameters:
stConfig_in.dummy = 1;      //dummy value
//
rc = MMC_ConfigCmd (hConn, &stConfig_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.7 MMC_CreateSYNCTimer

Creates a SYNC timer to synchronize servo-drive, Maestro movements using the connection handle operator.

```
MMC_LIB_API int MMC_CreateSYNCTimer(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SYNC_TIMER_CB_FUNC func,
IN unsigned short usSYNCTimerTime
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

func

[IN] Points to the callback function MMC_SYNC_TIMER_CB_FUNC using MMC_CreateSYNCTimer.

usSYNCTimerTime

[IN] Defines the time between which a synchronization message is sent as an event.

Remarks

None

Scope

All

Figure 8-7 describes the function for MMC_CreateSYNCTimer.

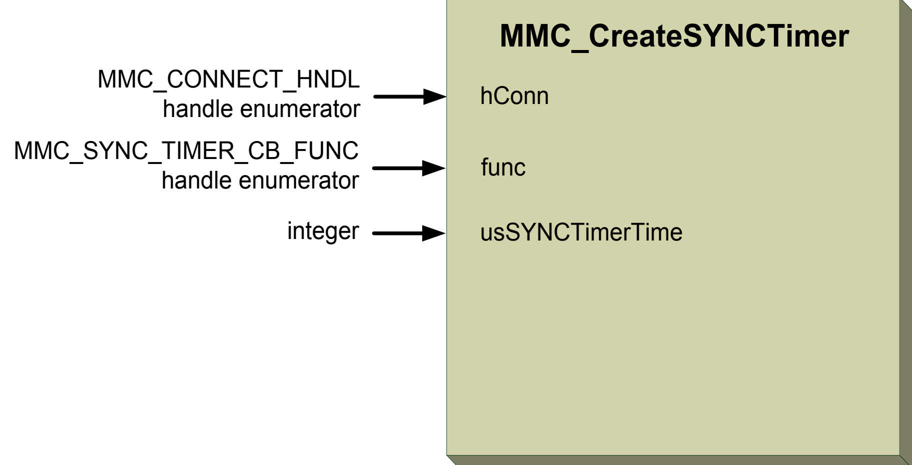


Figure 8-7: MMC_CreateSYNCTimer function

8.1.8 MMC_DestroySYNCTimer

Removes the SYNC timer to synchronize servo-drive, Maestro movements using the connection handle operator.

```
MMC_LIB_API int MMC_DestroySYNCTimer(
IN MMC_CONNECT_HNDL hConn
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

Remarks

None

Scope

All

Figure 8-8 describes the function for MMC_DestroySYNCTimer.

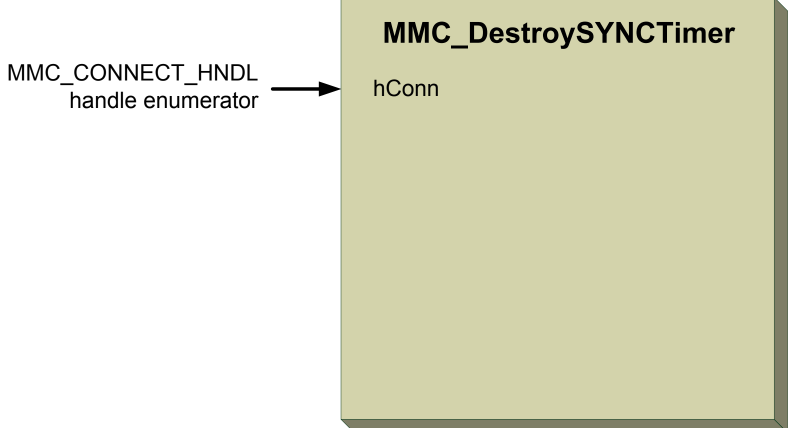


Figure 8-8: MMC_DestroySYNCTimer function

8.1.9 MMC_DownloadFoE

Manages downloads of a fileor files over EtherCAT to the Maestro.

Important: To use this function refer to Elmo for support.

```
MMC_LIB_API int MMC_DownloadFoE(
IN MMC_CONNECT_HNDL hConn,
IN MMC_DOWNLOADFOE_IN* pInParam,
OUT MMC_DOWNLOADFOE_OUT* pOutParam);
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_DOWNLOADFOE_IN** input data structure using the MMC_DownloadFoE function.

pOutParam

Points to the **MMC_DOWNLOADFOE_OUT** output structure receiving information, as a result of calling the MMC_DownloadFoE function.

Remarks

The FoE download procedure in the GMAS, is performed for N slaves on the bus in parallel. The master moves the slaves to Bootstrap mode and sends periodically at each Mailbox cycle, a full Mailbox message to each slave containing relevant sections of the file, until the complete file is downloaded to all N slaves.

Scope

EtherCAT

MMC_DOWNLOADFOE_IN Structure

```
t typedef struct mmc_downloadfoe_in{
unsigned short pwSlaveId[NC_NODES_SING_AXIS_NUM];
char pcFileName[256];
unsigned char pucServer[4];
unsigned char ucSlavesNum;
}MMC_DOWNLOADFOE_IN;
```

Parameters

pwSlaveId[NC_NODES_SING_AXIS_NUM]

Slave ID to which the download is sent, with a limit of 3 characters with any +ve value, dependant on the array [NC_NODES_SING_AXIS_NUM], the number of single axis nodes.

pcFileName[256]

Full location and filename to be downloaded to the Maestro from the host, with a limit of 256 characters

pucServer[4]

Serverhost IP with a limit of four characters (32 bit)

ucSlavesNum

Number of slaves to download the files with a maximum of 76 slaves.

MMC_DOWNLOADFOE_OUT Structure

```
typedef struct mmc_downloadfoe_out{
unsigned short usStatus;
unsigned short sErrorID;
}MMC_DOWNLOADFOE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-9 describes the function for MMC_DownloadFoE.

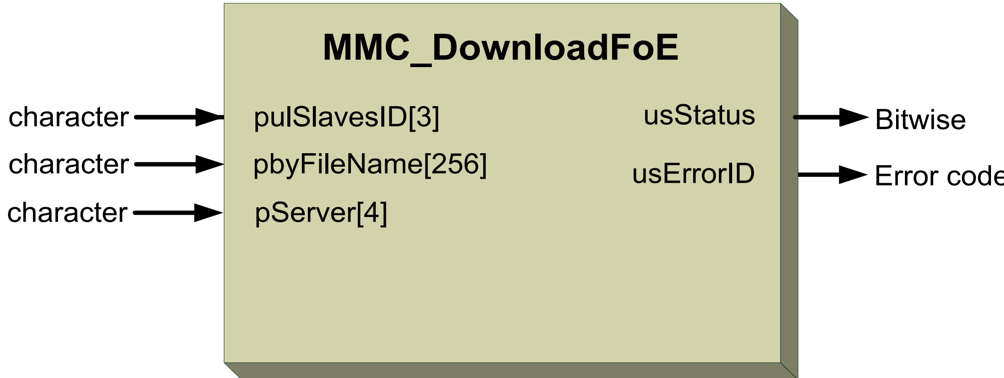


Figure 8-9: MMC_DownloadFoE function

8.1.9.2 Function Code and Implementation Example

```
void DownloadFoe()
{
     MMC_DOWNLOADFOE_IN dlfoe ;
     MMC_DOWNLOADFOE_OUT dlfoeout ;
     MMC_GETFOESTATUS_OUT foestat ;
     MMC_GET_GMASOP_MODE_OUT pOpmode ;

     MMC_GETCOMMSTATISTICSEX_IN gcstat_In ;
     MMC_GETCOMMSTATISTICSEX_OUT gcstat_Out ;
     int i ;
     //
     //
     // Before DownloadingFOE - It is good practice that drives will be reset because
     // if one of the drives is after DownloadFoE and was not reset, its state and statitstics
are unknown.
     //
     dlfoe.pwSlaveId[0]=0 ;      // Note: Slave ID is inserted here !!
     dlfoe.pwSlaveId[1]=1 ;      // Note: Slave ID is inserted here !!
     //
     dlfoe.ucSlavesNum = 2;      // Number of relevant slaves in the pwSlaveId array.
     //
     // Same for slave statistics:
     gcstat_In.pwSlaveId[0] = 0 ;
     gcstat_In.pwSlaveId[1] = 1 ;
     gcstat_In.ucSlavesNum = 2 ;
     //
     // Insert IP of tftp server. Usually the connection IP of the PC.
     dlfoe.pucServer[0] = 10 ;
     dlfoe.pucServer[1] = 10 ;
     dlfoe.pucServer[2] = 20 ;
     dlfoe.pucServer[3] = 55 ;
     //
     // Copy file path name to the structure. Should be relative to the tftp folder
     strcpy(dlfoe.pcFileName,"FoEFW  01.01.04.68 27Oct2011P01G.abs") ;

     // Start tftp server on host. Only then call the MMC_DownloadFoE.
     //
     int rc = MMC_DownloadFoE(conn_hndl,&dlfoe,&dlfoeout)   ;
     if(rc < 0)
     {
          // Error Calling MMC_DownloadFoE. Error in dlfoeout.sErrorID
          return ;
     }
     //
     // If we reached this line, the tftp was succesful. Poll the GMAS for results:
     while(TRUE)
     {
          Sleep(100) ;
```

```
        //
        // Check the FoE progress:
        MMC_GetFoEStatus(conn_hndl,&foestat);
        if(rc < 0)
        {
            // Error Calling MMC_GetFoEStatus. Error in foestat.sErrorID
            return ;
        }
        //
        // Check that the FoE started.
        if(foestat.ucFOEStarted)
        {
            rc = MMC_GetGMASOperationMode(conn_hndl,&pOpmode) ;
            if(rc < 0)
            {
                // Error Calling MMC_GetGMASOperationMode. Error in pOpmode.sErrorID
                return ;
            }
            // Print Remaining time - foestat.ucProgress
            //
            // Check Foe Download progress is over and GMAS back in operational mode.
            if ((foestat.ucProgress == 0) && (pOpmode.ucResult == 0))
            {
                //
                // Download over. Check to see if any drives failed.
                for(i = 0 ; i < dlfoe.ucSlavesNum ; i++)
                {
                    if(foestat.pstSlavesErrorID[i].sErrorID != 0)
                    {
                        // Error on one of the slaves. Print error:
                        // SlaveID: foestat.pstSlavesErrorID[i].usSlaveID  has error -
foestat.pstSlavesErrorID[i]
                    }
                }
                // Notify user to switch drives Off / On and then check the download status.
wait 5 sec's.
                //
                // please note - MAX 76 slaves can be read.
                rc = MMC_GetEthercatCommStatistics(conn_hndl,&gcstat_In,&gcstat_Out) ;
                if(rc < 0)
                {
                    // Error Calling MMC_GetEthercatCommStatistics. Error in
gcstat_Out.sErrorID
                    return ;
                }
                //
                // gcstat_Out.ucMasterState - Should be EcatStateO  . operational.
                // Good idea to read number of slaves on bus - gcstat_Out.usNumOfSlaves
                // gcstat_Out.ucMasterDiagnosticState - All bits should be 0, except for:
EcatMasterDiagnosticStateUpdated, EcatMasterDiagnosticStateDefaultDataWasSet bits.
                //
                for(i = 0 ; i < gcstat_In.ucSlavesNum ; i++)
                {
                    // gcstat_Out.pucAxesState[i] - Should be EcatStateO.
                    // gcstat_Out.pucAxesDiagnosticState[i] - Should be 0.
                    if((gcstat_Out.pstSII_Content[i].ulVendorId == 0x9A) &&
(gcstat_Out.pstSII_Content[i].ulRevisionNo <= 0xFF))
                    {
                        //
                        // Drive stuck in no firmware state. Notify User. In This case the
InitCmdFail in diagnostics is not relevant.
                    }
                }
                return ;
            }
        }
    }
}

int OnConnectGetDiagnostics()
```

```
{
     int rc ;
     MMC_GET_GMASOP_MODE_OUT pOpmode ;

     MMC_GETCOMMSTATISTICSEX_IN gcstat_In ;
     MMC_GETCOMMSTATISTICSEX_OUT gcstat_Out ;
     int i ;
     //
     rc = MMC_GetGMASOperationMode(conn_hndl,&pOpmode) ;
     if(rc < 0)
     {
          // Error Calling MMC_GetGMASOperationMode. Error in pOpmode.sErrorID
          return ;
     }
     //
     // Check GMAS Operational state. If == 2, then in Download FOE state.
     if (pOpmode.ucResult == 2)
     {
          // GMAS in Download FoE state. We decided that a mesage will be shown to user that 
the GMAS is in Download FoE.
     }

     rc = MMC_GetEthercatCommStatistics(conn_hndl,&gcstat_In,&gcstat_Out) ;
     if(rc < 0)
     {
          // Error Calling MMC_GetEthercatCommStatistics. Error in gcstat_Out.sErrorID
          return ;
     }
     //
     // gcstat_Out.ucMasterState - Should be EcatStateO. operational.
     // Good idea to read number of slaves on bus - gcstat_Out.usNumOfSlaves. Should be 
identical to number of drives configured.
     // gcstat_Out.ucMasterDiagnosticState - All bits should be 0, except for: 
EcatMasterDiagnosticStateUpdated, EcatMasterDiagnosticStateDefaultDataWasSet bits.
     //
     for(i = 0 ; i < gcstat_In.ucSlavesNum ; i++)
     {
          // gcstat_Out.pucAxesState[i] - Should be EcatStateO.
          //
          if((gcstat_Out.pstSII_Content[i].ulVendorId == 0x9A) && 
(gcstat_Out.pstSII_Content[i].ulRevisionNo <= 0xFF))
          {
               //
               // One of the Drives is stuck in no firmware state. Notify User to go to 
diagnostics tab - NOT TO CONFIGURATOR.
               // In this case - the gcstat_Out.pucAxesDiagnosticState[i] InitCmd bit may be 
set..
          }
          else
          {
               // pucAxesDiagnosticState[i] should be 0 !
          }

     }
}
```

8.1.10 MMC_Exit

Changes the Maestro from configuration mode back to regular mode.

```
MMC_LIB_API int MMC_ExitCmd
(IN MMC_CONNECT_HNDL hConn,
IN MMC_EXIT_IN* pInParam,
OUT MMC_EXIT_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_EXIT_IN** input data structure using the MMC_Exit function.

pOutParam

Points to the **MMC_EXIT_OUT** output structure receiving information, as a result of calling the MMC_Exit function.

Remarks

There are two Maestro operational modes:

- Normal
- Configuration

When any communication configuration parameters (network IP etc.) are changed using the Set command, this function is invoked to exit the configuration mode to the normal operational mode of the Maestro.

Scope

All

MMC_EXIT_IN Structure

typedef struct{
unsigned char dummy;
}MMC_EXIT_IN;

Parameters

dummy

Dummy character. Any +ve value accepted.

MMC_EXIT_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_EXIT_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-10 describes the function block for MMC_Exit

Figure 8-10: MMC_Exit function block

8.1.10.2 Function Block Code Example

```
int rc;
MMC_EXIT_IN        stExit_in;
MMC_EXIT_OUT       stExit_out;
//
// Inserting the structure parameters:
stExit_in.dummy    = 1;      //Function block handle
//
rc = MMC_ExitCmd (hConn, &stExit_in, &stExit_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.11 MMC_FreeFbStat

Returns debug information that contains the number of free function blocks in the system.

```
MMC_LIB_API int MMC_FreeFbStatCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_FREEFBSTAT_IN* pInParam,
OUT MMC_FREEFBSTAT_OUT* pOutParam
);
```

Motion Mode NC – Not relevant Distributed – Not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_FREEFBSTAT_IN** input data structure using the MMC_FreeFbStat function.

pOutParam

Points to the **MMC_FREEFBSTAT_OUT** output structure receiving information, as a result of calling the MMC_FreeFbStat function.

Remarks

None

Scope

All

MMC_FREEFBSTAT_IN Structure

```
typedef struct{
unsigned int uiHndl;
}MMC_FREEFBSTAT_IN;
```

Parameters

uiHndl

Returned function block handle. Integer with any +ve value

MMC_FREEFBSTAT_OUT Structure

```
typedef struct{
unsigned int uiFreeLargeFb;
unsigned int uiFreeMediumFb;
unsigned int uiFreeSmallFb;
unsigned short usStatus;
short sErrorID;
}MMC_FREEFBSTAT_OUT;
```

Parameters

uiFreeLargeFb

Number of free large size function blocks. Any +ve integer value.

uiFreeMediumFb

Number of free medium size function blocks. Any +ve integer value.

uiFreeSmallFb

Number of free small size function blocks. Any +ve integer value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-11 describes the function block for MMC_FreeFbStat

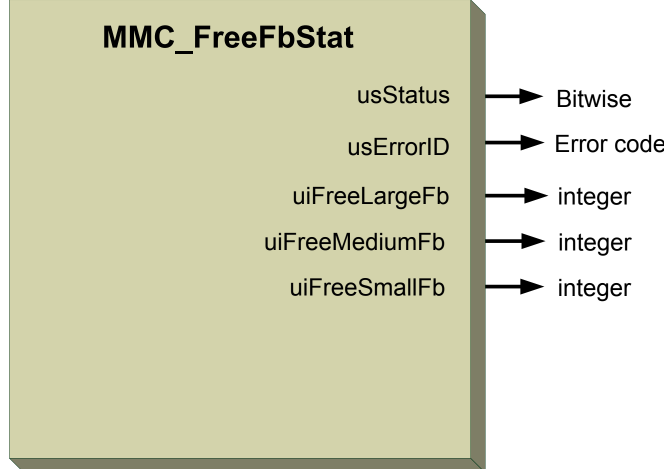


Figure 8-11: MMC_FreeFbStat function block

8.1.11.2 Function Block Code Example

```
int rc;
MMC_FREEFBSTAT_IN      stFreeFBStat_in;
MMC_FREEFBSTAT_OUT     stFreeFBStat_out;
//
// Inserting the structure parameters:
stFreeFBStat_in.uiHndl  = 10;   // Requested function block handle
//
rc = MMC_FreeFbStatCmd (hConn, &stFreeFBStat_in, &stFreeFBStat_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.12 MMC_GetActiveVectorsNum

Displays the number of active vectors (groups) attached and managed by the Maestro.

```
MMC_LIB_API int MMC_GetActiveVectorsNum(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETACTIVEVECTORSNUM_IN* pInParam,
OUT MMC_GETACTIVEVECTORSNUM_OUT* pOutParam
);
```

Motion Mode NC – Not Supported Distributed – Supported

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETACTIVEVECTORSNUM_IN** input data structure using the MMC_GetActiveVectorsNum function.

pOutParam

Points to the **MMC_GETACTIVEVECTORSNUM_OUT** output structure receiving information as a result of calling the MMC_GetActiveVectorsNum function.

Remarks

This function will provide this basic information without opening the Maestro Personality file.

Scope

All

MMC_GETACTIVEVECTORSNUM_IN Structure

```
typedef struct {
unsigned char dummy;
}MMC_GETACTIVEVECTORSNUM_IN;
```

Parameters

dummy

Any dummy values

MMC_GETACTIVEVECTORSNUM_OUT Structure

```
typedef struct {
int iActiveVectorsNum;
unsigned short usStatus;
short sErrorID;
}MMC_GETACTIVEVECTORSNUM_OUT;
```

Parameters

iActiveVectorsNum Provides the actives vectors in a group. +ve integer value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 8-12 describes the function block for MMC_GetActiveVectorsNum

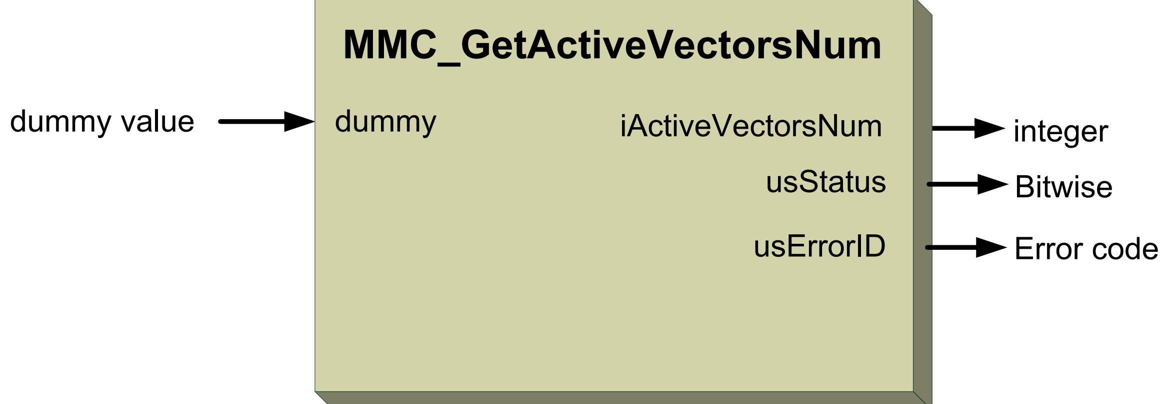


Figure 8-12: MMC_GetActiveVectorsNum function block

8.1.13 MMC_GetErrorCodeDescriptionByID

This function receives an error\warning code and returns the description and resolution from the Personality file.

```
MMC_LIB_API int MMC_GetErrorCodeDescriptionByID(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETERRORCODEDESCRIPTIONBYID_IN* pInParam,
OUT MMC_GETERRORCODEDESCRIPTIONBYID_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_general_API.h
GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETERRORCODEDESCRIPTIONBYID_IN** input data structure using the MMC_GetErrorCodeDescriptionByID function. The input [IN] receives the error\warning number.

pOutParam

Points to the **MMC_GETERRORCODEDESCRIPTIONBYID_OUT** output structure receiving information, as a result of calling the MMC_GetErrorCodeDescriptionByID function. The output [OUT] returns two strings, status and error id.

Remarks

None

Scope

Errors from the Maestro Personality file.

MMC_GETERRORCODEDESCRIPTIONBYID_IN Structure

```
typedef struct mmc_getcodedescriptionbyid_in{
int iCode;
Char cType;
} MMC_GETERRORCODEDESCRIPTIONBYID_IN;
```

Parameters

iCode

Error and or warning code value. Any integer value which may be +ve or –ve

cType

The code type, which may be one of the following:

1 – GMAS code

2 – Drive emergency code

3 – Drive Abortion Code

MMC_GETERRORCODEDESCRIPTIONBYID_OUT Structure

```
typedef struct mmc_getcodedescriptionbyid_out{
char pResolution[1100];
char pDescription[256];
unsigned short usStatus;
short sErrorID;
} MMC_GETERRORCODEDESCRIPTIONBYID_OUT;
```

Parameters

pResolution[1100]

Character value of the resolution with a maximum of 1100 characters

pDescription[256]

Character value of the description with a maximum value of 256 characters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-13 describes the function for MMC_GetErrorCodeDescriptionByID as applied within the IEC 61131 programming for MC_GetAxisRef.

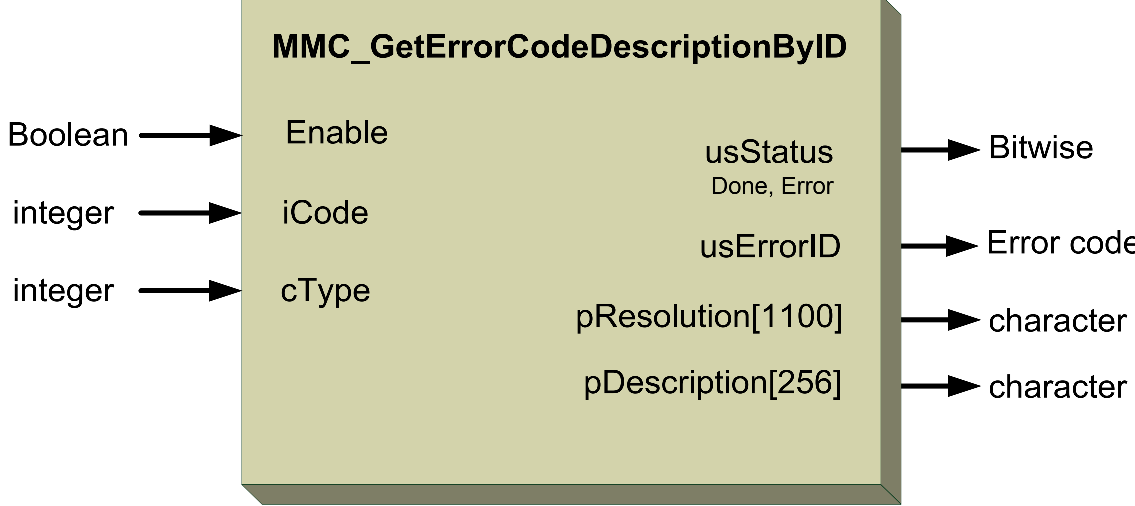


Figure 8-13: MMC_GetErrorCodeDescriptionByID function

8.1.14 MMC_GetFoEStatus

Obtains the File over EtherCAT status after a file download using MMC_DownloadFoE, from a host to the Maestro.

Important: To use this function refer to Elmo for support.

```
MMC_LIB_API int MMC_GetFoEStatus(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETFOESTATUS_IN* pInParam,
OUT MMC_GETFOESTATUS_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETFOESTATUS_IN** input data structure using the MMC_GetFoEStatus function.

pOutParam

Points to the **MMC_GETFOESTATUS_OUT** output structure receiving information, as a result of calling the MMC_GetFoEStatus function.

Remarks

During a FoE download procedure, the user can periodically poll for statistics related to the FoE download procedure via MMC_GetFoEStatus(). The statistics obtainable are:

- Percentage progress of the FoE download procedure.
- Status parameter for each slave which participates in the FoE download procedure. If this status parameter is less than zero, an error occurrs for the relevant slave and this slave is excluded from the procedure.
- General status flag which indicates the current status of the FoE download procedure. As long as the download FoE procedure continues successfully, this parameter will be zero. However if the download procedure fails, then this parameter contains the specific error-ID causing the procedure to fail.
- FOE download's start flag (rising edge flag) – this flag indicates that the FoE download procedure (a totally asynchrony procedure), actually started.

Scope

EtherCAT

MMC_GETFOESTATUS_IN Structure

```
typedef struct mmc_getfoestatus_in{
unsigned char ucDummy;
}MMC_GETFOESTATUS_IN;
```

Parameters

ucDummy

Dummy data. +ve characters of any value.

MMC_GETFOESTATUS_OUT Structure

```
typedef struct mmc_getfoestatus_out{
unsigned short usStatus;
unsigned short sErrorID;
short sFOEStatus;
FOE_SLAVE_INFO pstSlavesErrorID[NC_NODES_SING_AXIS_NUM];
unsigned char ucNumOfSlaves;
unsigned char ucProgress ;
unsigned char ucFOEStarted ;
}MMC_GETFOESTATUS_OUT;
```

Parameters

sFOEStatus

FoE status value, either an error code or No Error

pstSlavesErrorID

The Slaves error ID structure, dependant on the array [NC_NODES_SING_AXIS_NUM], the number of single axis nodes.

FOE_SLAVE_INFO

```
typedef struct foe_slave_info{
unsigned short usSlaveID;
short sErrorID;
}FOE_SLAVE_INFO;
```

usSlaveID

Slave ID for the FoE

sErrorID

The error ID of the slave if a fault occurs.

ucNumOfSlaves

The number of slaves for which the file is downloaded, with a maximum of 76 slaves.

ucProgress

The progress of the FoE as a +ve percentage value.

ucFOEStarted

The FoE actual start position flag when the download starts (rising edge) as a +ve character value. Used when a number of file downloads occur from the server. Prevents the progress bar starting before the file download occurs.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-14 describes the function for MMC_GetFoEStatus

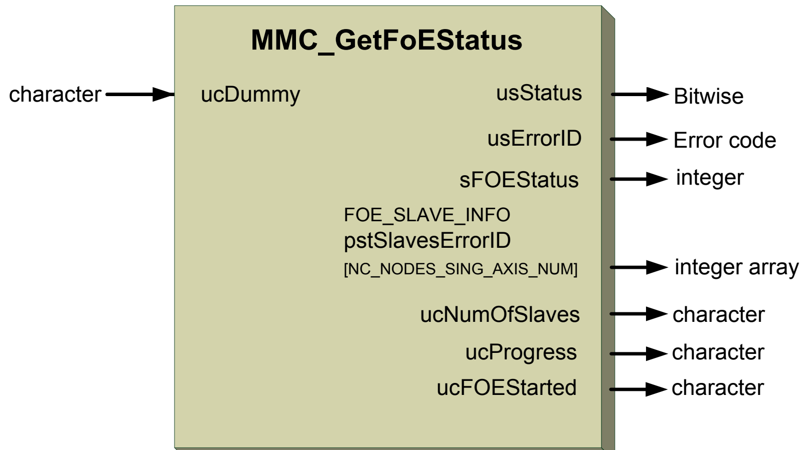


Figure 8-14: MMC_GetFoEStatus function

8.1.14.2 Function Code Example

```
void DownloadFoe()
{
     MMC_DOWNLOADFOE_IN dlfoe ;
     MMC_DOWNLOADFOE_OUT dlfoeout ;
     MMC_GETFOESTATUS_OUT foestat ;
     MMC_GET_GMASOP_MODE_OUT pOpmode ;

     MMC_GETCOMMSTATISTICSEX_IN gcstat_In ;
     MMC_GETCOMMSTATISTICSEX_OUT gcstat_Out ;
     int i ;
     //
     //
     // Before DownloadingFOE - It is good practice that drives will be reset because
     // if one of the drives is after DownloadFoE and was not reset, its state and statitstics
are unknown.
     //
     dlfoe.pwSlaveId[0]=0 ;      // Note: Slave ID is inserted here !!
     dlfoe.pwSlaveId[1]=1 ;      // Note: Slave ID is inserted here !!
     //
     dlfoe.ucSlavesNum = 2;      // Number of relevant slaves in the pwSlaveId array.
     //
     // Same for slave statistics:
     gcstat_In.pwSlaveId[0] = 0 ;
     gcstat_In.pwSlaveId[1] = 1 ;
     gcstat_In.ucSlavesNum = 2 ;
     //
     // Insert IP of tftp server. Usually the connection IP of the PC.
     dlfoe.pucServer[0] = 10 ;
     dlfoe.pucServer[1] = 10 ;
     dlfoe.pucServer[2] = 20 ;
     dlfoe.pucServer[3] = 55 ;
     //
     // Copy file path name to the structure. Should be relative to the tftp folder
     strcpy(dlfoe.pcFileName,"FoEFW  01.01.04.68 27Oct2011P01G.abs") ;

     // Start tftp server on host. Only then call the MMC_DownloadFoE.
     //
     int rc = MMC_DownloadFoE(conn_hndl,&dlfoe,&dlfoeout)   ;
     if(rc < 0)
     {
          // Error Calling MMC_DownloadFoE. Error in dlfoeout.sErrorID
          return ;
     }
     //
     // If we reached this line, the tftp was succesful. Poll the GMAS for results:
     while(TRUE)
     {
          Sleep(100) ;
          //
          // Check the FoE progress:
          MMC_GetFoEStatus(conn_hndl,&foestat);
          if(rc < 0)
          {
               // Error Calling MMC_GetFoEStatus. Error in foestat.sErrorID
               return ;
          }
          //
          // Check that the FoE started.
          if(foestat.ucFOEStarted)
          {
               rc = MMC_GetGMASOperationMode(conn_hndl,&pOpmode) ;
               if(rc < 0)
               {
                    // Error Calling MMC_GetGMASOperationMode. Error in pOpmode.sErrorID
                    return ;
               }
               // Print Remaining time - foestat.ucProgress
```

```
            //
            // Check Foe Download progress is over and GMAS back in operational mode.
            if ((foestat.ucProgress == 0) && (pOpmode.ucResult == 0))
            {
                //
                // Download over. Check to see if any drives failed.
                for(i = 0 ; i < dlfoe.ucSlavesNum ; i++)
                {
                    if(foestat.pstSlavesErrorID[i].sErrorID != 0)
                    {
                        // Error on one of the slaves. Print error:
                        // SlaveID: foestat.pstSlavesErrorID[i].usSlaveID  has error -
foestat.pstSlavesErrorID[i]
                    }
                }
                // Notify user to switch drives Off / On and then check the download status.
wait 5 sec's.
                //
                // please note - MAX 76 slaves can be read.
                rc = MMC_GetEthercatCommStatistics(conn_hndl,&gcstat_In,&gcstat_Out) ;
                if(rc < 0)
                {
                    // Error Calling MMC_GetEthercatCommStatistics. Error in
gcstat_Out.sErrorID
                    return ;
                }
                //
                // gcstat_Out.ucMasterState - Should be EcatStateO  . operational.
                // Good idea to read number of slaves on bus - gcstat_Out.usNumOfSlaves
                // gcstat_Out.ucMasterDiagnosticState - All bits should be 0, except for:
EcatMasterDiagnosticStateUpdated, EcatMasterDiagnosticStateDefaultDataWasSet bits.
                //
                for(i = 0 ; i < gcstat_In.ucSlavesNum ; i++)
                {
                    // gcstat_Out.pucAxesState[i] - Should be EcatStateO.
                    // gcstat_Out.pucAxesDiagnosticState[i] - Should be 0.
                    if((gcstat_Out.pstSII_Content[i].ulVendorId == 0x9A) &&
(gcstat_Out.pstSII_Content[i].ulRevisionNo <= 0xFF))
                    {
                        //
                        // Drive stuck in no firmware state. Notify User. In This case the
InitCmdFail in diagnostics is not relevant.
                    }
                }
                return ;
            }
        }
    }
}

int OnConnectGetDiagnostics()
{
    int rc ;
    MMC_GET_GMASOP_MODE_OUT pOpmode ;

    MMC_GETCOMMSTATISTICSEX_IN gcstat_In ;
    MMC_GETCOMMSTATISTICSEX_OUT gcstat_Out ;
    int i ;
    //
    rc = MMC_GetGMASOperationMode(conn_hndl,&pOpmode) ;
    if(rc < 0)
    {
        // Error Calling MMC_GetGMASOperationMode. Error in pOpmode.sErrorID
        return ;
    }
    //
    // Check GMAS Operational state. If == 2, then in Download FOE state.
    if (pOpmode.ucResult == 2)
    {
```

```
        // GMAS in Download FoE state. We decided that a mesage will be shown to user that
the GMAS is in Download FoE.
    }

    rc = MMC_GetEthercatCommStatistics(conn_hndl,&gcstat_In,&gcstat_Out) ;
    if(rc < 0)
    {
        // Error Calling MMC_GetEthercatCommStatistics. Error in gcstat_Out.sErrorID
        return ;
    }
    //
    // gcstat_Out.ucMasterState - Should be EcatStateO. operational.
    // Good idea to read number of slaves on bus - gcstat_Out.usNumOfSlaves. Should be
identical to number of drives configured.
    // gcstat_Out.ucMasterDiagnosticState - All bits should be 0, except for:
EcatMasterDiagnosticStateUpdated, EcatMasterDiagnosticStateDefaultDataWasSet bits.
    //
    for(i = 0 ; i < gcstat_In.ucSlavesNum ; i++)
    {
        // gcstat_Out.pucAxesState[i] - Should be EcatStateO.
        //
        if((gcstat_Out.pstSII_Content[i].ulVendorId == 0x9A) &&
(gcstat_Out.pstSII_Content[i].ulRevisionNo <= 0xFF))
        {
            //
            // One of the Drives is stuck in no firmware state. Notify User to go to
diagnostics tab - NOT TO CONFIGURATOR.
            // In this case - the gcstat_Out.pucAxesDiagnosticState[i] InitCmd bit may be
set..
        }
        else
        {
            // pucAxesDiagnosticState[i] should be 0 !
        }

    }
}
```

8.1.15 MMC_GetEnquireFbStatus

Obtains the current state global parameter Receive FB status in EAS.

```
MMC_LIB_API int MMC_GetEnquireFbStatusCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETENQUIREFBSTATUS_IN* pInParam,
OUT MMC_GETENQUIREFBSTATUS_OUT* pOutParam
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETENQUIREFBSTATUS_IN** input data structure using the MMC_GetEnquireFbStatus function.

pOutParam

Points to the **MMC_GETENQUIREFBSTATUS_OUT** output structure receiving information, as a result of calling the MMC_GetEnquireFbStatus function.

Remarks

This function is only for IPC programming in C, and C++, and allows the user to control the release mode of the function blocks. By default it is read from the Maestro resource file and should be set to False (0). If set to True (1) , the FBs will not be released from the FB queue until the user releases them implicitly. For EASII and any RPC programming, it will be set automatically to False (0) without user control.

In the IEC61131-3 program, the function parameter is also read from the Maestro resource file and is automatically set to True (1). This is because the IEC61131-3 function blocks require and provide real data from the Maestro and drives attached, during their motion, as they load and run each function block in turn. The function block will only be released from the queue when the Maestro and drives perform that function block and move to the next function block.

Scope

All

MMC_GETENQUIREFBSTATUS_IN Structure

```
typedef struct mmc_getenquirefbstatus_in{
unsigned char ucDummy;
} MMC_GETENQUIREFBSTATUS_IN;
```

Parameters

dummy

Dummy character. Any +ve value accepted.

MMC_GETENQUIREFBSTATUS_OUT Structure

```
typedef struct mmc_getenquirefbstatus_out{
unsigned char ucCurrentStatus;
unsigned short usStatus;
short sErrorID;
} MMC_GETENQUIREFBSTATUS_OUT;
```

Parameters

ucCurrentStatus

The current status of the function block, with +ve character values of 0 - 255.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-15describes the function for MMC_GetEnquireFbStatus

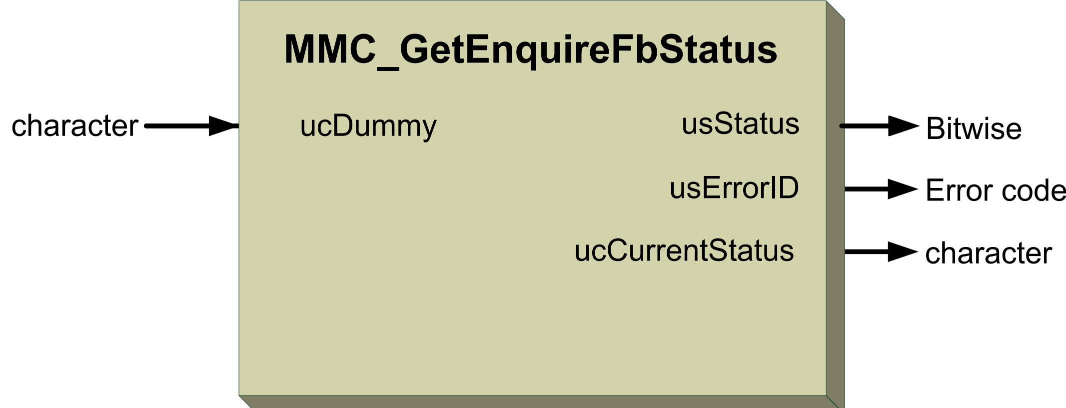


Figure 8-15: MMC_GetEnquireFbStatus function

8.1.16 MMC_GetAxisByName

Returns an axis index reference by its name.

MMC_LIB_API int MMC_GetAxisByNameCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXISBYNAME_IN* pInParam,
OUT MMC_AXISBYNAME_OUT* pOutParam
);

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h
GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_AXISBYNAME_IN** input data structure using the MMC_GetAxisByName function.

pOutParam

Points to the **MMC_AXISBYNAME_OUT** output structure receiving information, as a result of calling the MMC_GetAxisByName function.

Remarks

None

Scope

All

MMC_AXISBYNAME_IN Structure

```
typedef struct{
char cAxisName[NODE_NAME_MAX_LENGTH];
}MMC_AXISBYNAME_IN;
```

Parameters

cAxisName

Axis name. Any +ve character value.

[NODE_NAME_MAX_LENGTH] is an array of node maximum name lengths with values [1….n].

MMC_AXISBYNAME_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned short usAxisIdx;
}MMC_AXISBYNAME_OUT;
```

Parameters

usAxisIdx

Axis reference. Specific integer value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-16 describes the function block for MMC_GetAxisByName as applied within the IEC 61131 programming for MC_GetAxisRef.

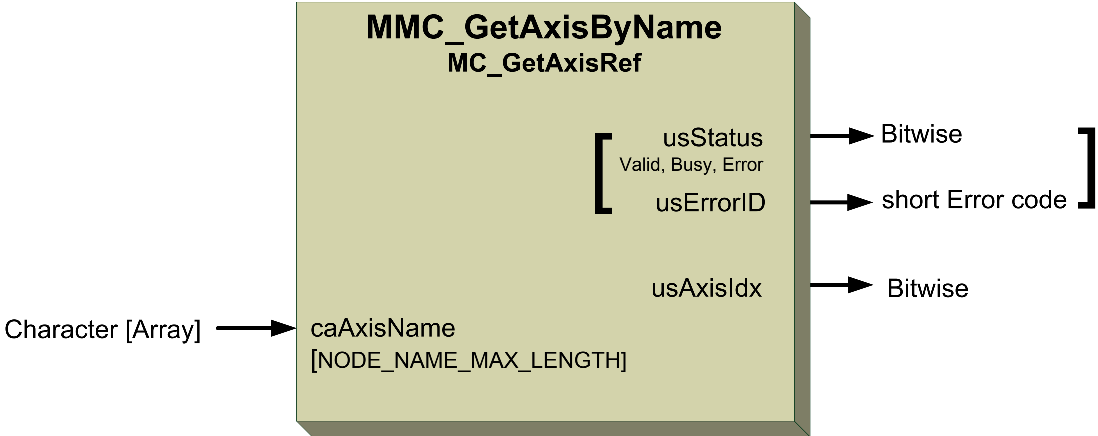


Figure 8-16: MMC_GetAxisByName function block

8.1.16.2 Function Block Code Example

```
int rc;
MMC_AXISBYNAME_IN      stAxisByName_in;
MMC_AXISBYNAME_OUT     stAxisByName_out;
//
// Inserting the structure parameters:
strcpy(stAxisByName_in.cAxisName, "1st_Axis_name");
strcpy(stAxisByName_in.cAxisName, "2nd_Axis_name");
strcpy(stAxisByName_in.cAxisName, "3rd_Axis_name");
//
rc = MMC_GetAxisByNameCmd (hConn, &stAxisByName_in, &stAxisByName_out);
printf("Axis State[%ld] ErrId[%d]\n", (long int)stAxisByName_out.usAxisIdx,
(short)stAxisByName_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

8.1.17 MMC_GetGroupByName

This function returns a group index reference by its name.

MMC_LIB_API int MMC_GetGroupByNameCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXISBYNAME_IN* pInParam,
OUT MMC_AXISBYNAME_OUT* pOutParam
);

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h
GMAS Programming(IEC 61331 Program)\ElmoGroupAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_AXISBYNAME_IN** input data structure using the MMC_GetGroupByName function.

pOutParam

Points to the **MMC_AXISBYNAME_OUT** output structure receiving information, as a result of calling the MMC_GetGroupByName function.

Remarks

From the IEC 61331-3 viewpoint, this is a function and not a function block. Therefore, it consists of name as an input and one single return code, which is the axis reference. No other outputs are available.

If the axis name is missing among GMAS resources, then this function returns -1 (or 65535)

Scope

All

MMC_AXISBYNAME_IN Structure

```
typedef struct{
char cAxisName[NODE_NAME_MAX_LENGTH];
}MMC_AXISBYNAME_IN;
```

Parameters

cAxisName

Axis name. Any +ve character value.

[NODE_NAME_MAX_LENGTH] is an array of node maximum name lengths with values [1….n].

MMC_AXISBYNAME_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned short usAxisIdx;
}MMC_AXISBYNAME_OUT;
```

Parameters

usAxisIdx

Axis index reference. Specific integer bitwise value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-17 describes the function block for MMC_GetGroupByName as applied within the IEC 61131 programming for MC_GetGroupAxisRef.

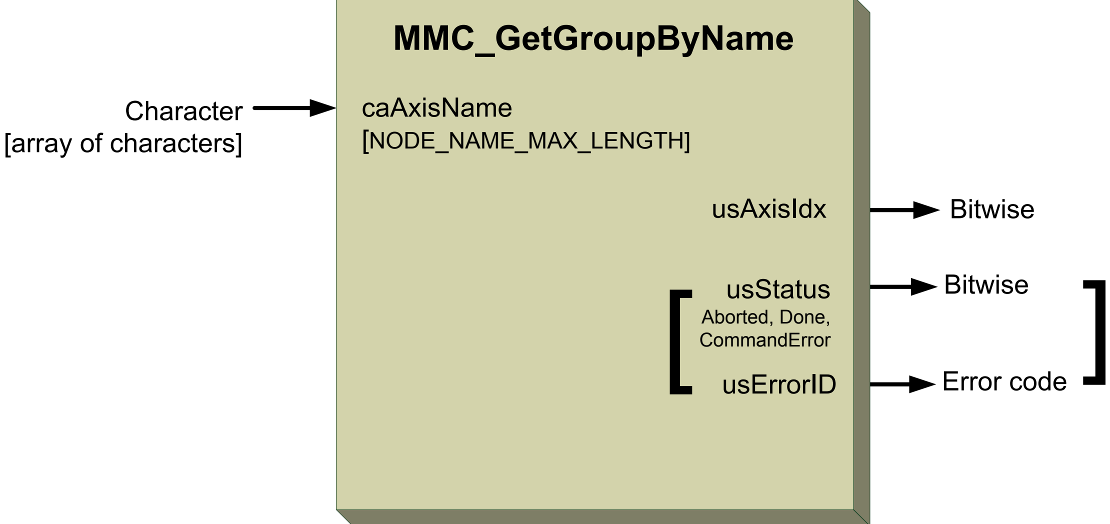


Figure 8-17: MMC_GetGroupByName function block

8.1.17.2 Function Block Code Example

```
int rc;
MMC_AXISBYNAME_IN      stAxisByName_in;
MMC_AXISBYNAME_OUT     stAxisByName_out;
//
// Inserting the structure parameters:
strcpy(stAxisByName_in.cAxisName, "1st_Group_name");
strcpy(stAxisByName_in.cAxisName, "2nd_Group_name");
strcpy(stAxisByName_in.cAxisName, "3rd_Group_name");
//
rc = MMC_GetGroupByNameCmd (hConn, &stAxisByName_in, &stAxisByName_out);
printf("Group State[%ld] ErrId[%d]\n", (long int)stAxisByName_out.usAxisIdx,
(short)stAxisByName_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

8.1.18 MMC_GetGMASOperationMode

Returns the current GMAS operation mode.

```
MMC_LIB_API int MMC_GetGMASOperationMode(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GET_GMASOP_MODE_IN* pInParam
OUT MMC_GET_GMASOP_MODE_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed - Not relevant
Source	GMAS\includes\MMC_general_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GET_GMASOP_IN** input data structure using the MMC_GetGMASOperationMode function.

pOutParam

Points to the **MMC_GET_GMASOP_OUT** output structure receiving information, as a result of calling the MMC_GetGMASOperationMode function.

Remarks

None

Scope

All

MMC_GET_GMASOP_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_GET_GMASOP_MODE_IN;
```

Parameters

dummy

Dummy input. Any +ve character value.

MMC_GET_GMASOP_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned char ucResult;
}MMC_GET_GMASOP_MODE_OUT;
```

Parameters

ucResult

Returned answer to question “Is the EtherCAT Config mode operational?” Yes or No. Boolean acceptable values of 0 or 1 accepted.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-18 describes the function for MMC_GetGMASOperationMode as applied within the IEC 61131 programming for MC_GetOpMode.

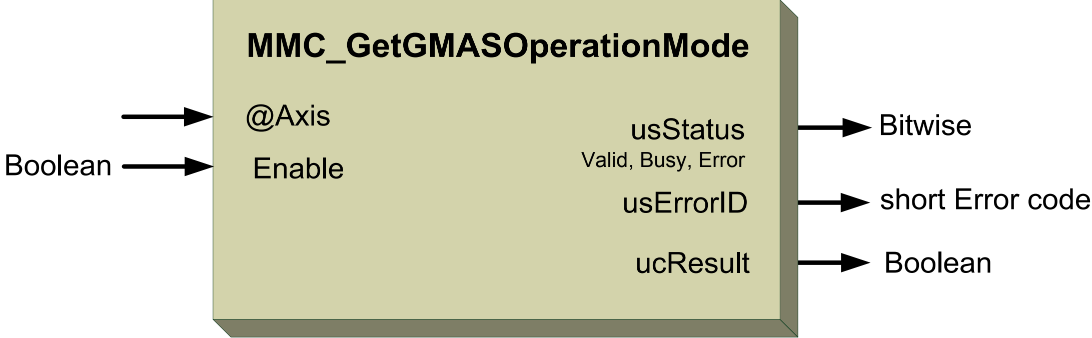


Figure 8-18: MMC_GetGMASOperationMode function

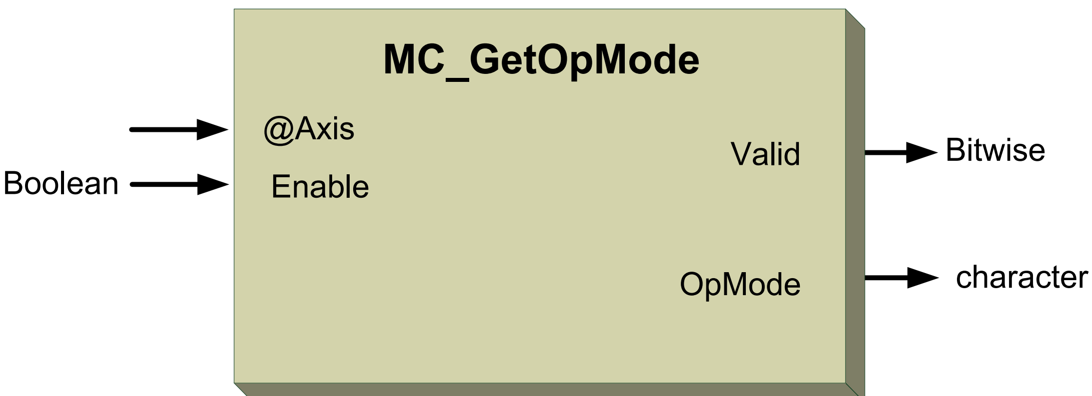


Figure 8-19: MMC_GetOpMode function

8.1.18.2 Function Block Code Example

```
int rc;
MMC_GET_GMASOP_MODE_IN      stGetGMASOpMode_in;
MMC_GET_GMASOP_MODE_OUT     stGetGMASOpMode_out;
//
// Inserting the structure parameters:
stGetGMASOpMode_in.ucDummy      = 1;      // Dummy input
//
rc = MMC_GetGMASOperationMode (hConn, &stGetGMASOpMode_out);
printf("GMAS Operation State[%ld] ErrId[%d]\n", (long int)stGetGMASOpMode_out.ucResult,
(short)stGetGMASOpMode_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

8.1.19 MMC_GetStatusRegister

The purpose of the function is to provide usable information regarding the Maestro and axes statuses.

```
int MMC_GetStatusRegisterCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETSTATUSREGISTER_IN* pInParam,
OUT MMC_GETSTATUSREGISTER_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Supported

Source GMAS\includes\MMC_general_API.h

GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETSTATUSREGISTER_IN** input data structure using the MMC_GetStatusRegister function.

pOutParam

Points to the **MMC_GETSTATUSREGISTER_OUT** output structure receiving information as a result of calling the MMC_GetStatusRegister function.

Remarks

None

Scope

All

MMC_GETSTATUSREGISTER_IN Structure

```
typedef struct mmc_getstatusregister_in{
unsigned char dummy;
} MMC_GETSTATUSREGISTER_IN;
```

Parameters

dummy

Any dummy value.

MMC_GETSTATUSREGISTER_OUT Structure

```
typedef struct mmc_getstatusregister_out{
unsigned int uiStatusRegister;
unsigned int uiMcsLimitRegister;
unsigned short usStatus;
short sErrorID;
unsigned char ucEndMotionReason
unsigned char cBuffer[32];
} MMC_GETSTATUSREGISTER_OUT;
```

Parameters

uiStatusRegister

This the Status Register, a 32 bit status register, with 10 lower bits related to the hardware/software limits feature. Refer to the section **14.2.2 below Status Register**. All other bits will be used in the future.

uiMcsLimitRegister

This is the MCS Limit Register, a 32 bit representation of the software limit status of all kinematic directions, 16 directions * 2 limits (High\Low) = 32

ucEndMotionReason

Inform reason for motion ending. +ve value char data.

cBuffer[32]

cBuffer is currently not in use, and this memory is allocated for future uses.

usStatus

Bitwise returned command status with the following values:

Aborted Done CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-20 describes the function block for MMC_GetStatusRegister as applied within the IEC 61131 programming.

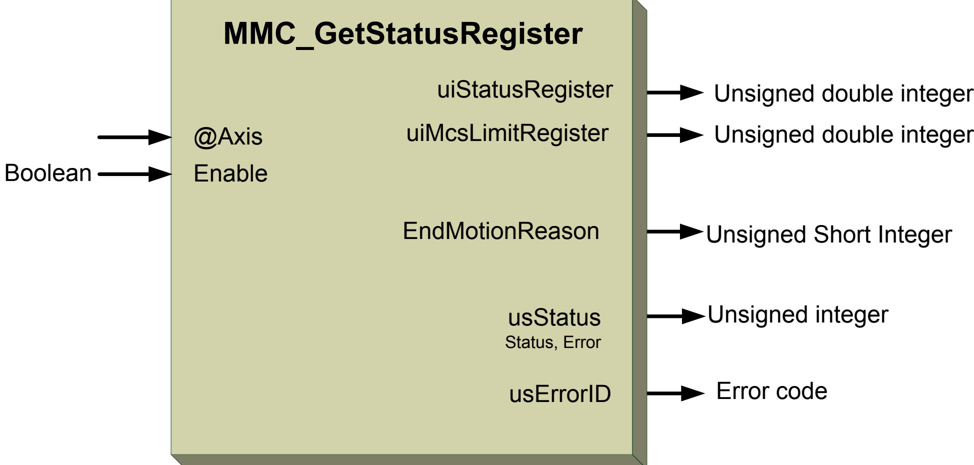


Figure 8-20: MMC_GetStatusRegister function block

8.1.19.2 Function Block Code Example

```
     CMMCConnection MyConnectionClass;
     MMC_CONNECT_HNDL ui_conn_hndl;
//
// Create Connection
     ui_conn_hndl = MyConnectionClass.ConnectIPC(0x7FFFFFFF,NULL);
//
// Initiate axis
     CMMCSingleAxis AxisA, AxisB;
     AxisA.InitAxisData("b01",ui_conn_hndl);
     AxisB.InitAxisData("b02",ui_conn_hndl);
//
     MMC_GETSTATUSREGISTER_IN StatusIN;
     MMC_GETSTATUSREGISTER_OUT StatusOUT;
     iRetval = MMC_GetStatusRegisterCmd(ui_conn_hndl,AxisA.GetRef(),&StatusIN,&StatusOUT);

          if(iRetval != NC_OK)
// Error handling
}
          else
{
// Status Register
          StatusOUT.uiStatusRegister;
// If the axis in SW Low limit the value will be 4 in decimal and 100 in binary

// MCS limit register relevant for group represents the status of MCS limits
          StatusOUT.uiMcsLimitRegister;

// Status
          StatusOUT.usStatus;

// Error ID
          StatusOUT.sErrorID;

// Future use
          StatusOUT.cBuffer[32];
}
```

8.1.20 MMC_GetResList

Returns the list of all resource files.

```
MMC_LIB_API int MMC_GetResListCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GET_RESLIST_IN* pInParam,
OUT MMC_GET_RESLIST_OUT* pOutParam);
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GET_RESLIST_IN** input data structure using the MMC_GetResList function.

pOutParam

Points to the **MMC_GET_RESLIST_OUT** output structure receiving information, as a result of calling the MMC_GetResList function.

Remarks

None

Scope

All

MMC_GET_RESLIST_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_GET_RESLIST_IN;
```

Parameters

dummy

Dummy input. Any +ve character value.

MMC_GET_RESLIST_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
char pResList[1024];
}MMC_GET_RESLIST_OUT;
```

Parameters

pResList[1024]

Resource file list. Any +ve characters limited to 1024 chars

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-21 describes the function block for MMC_GetResList

Figure 8-21: MMC_GetResList function block

8.1.20.2 Function Block Code Example

```
int rc;
MMC_GET_RESLIST_IN          stGetResList_in;
MMC_GET_RESLIST_OUT         stGetResList_out;
//
// Inserting the structure parameters:
stGetResList_in.dummy       = 1;      //Dummy input
//
rc = MMC_GetResListCmd (hConn, &stGetResList_in, &stGetResList_out);
printf("Resource List State[%ld] ErrId[%d]\n", (long int)stGetResList_out.pResList,
(short)stGetResList_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

8.1.21 MMC_GetResSnapshot

Save the resource configuration to temporary snapshot file.

```
MMC_LIB_API int MMC_GetResSnapshotCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_RESSNAPSHOT_IN* pInParam,
OUT MMC_RESSNAPSHOT_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_RESSNAPSHOT_IN** input data structure using the MMC_GetResSnapshot function.

pOutParam

Points to the **MMC_RESSNAPSHOT_OUT** output structure receiving information, as a result of calling the MMC_GetResSnapshot function.

Remarks

None

Scope

All

MMC_RESSNAPSHOT_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_RESSNAPSHOT_IN;
```

Parameters

dummy

Dummy input. Any +ve character value.

MMC_RESSNAPSHOT_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_RESSNAPSHOT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-22 describes the function block for MMC_GetResSnapshot

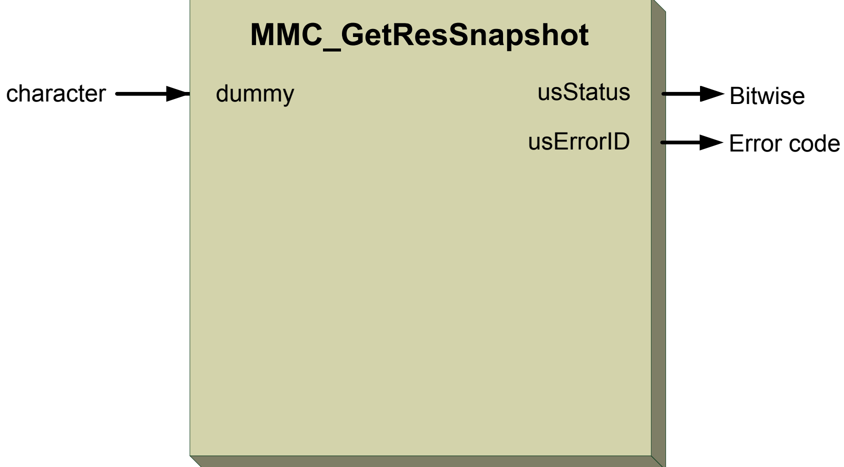


Figure 8-22: MMC_GetResSnapshot function block

8.1.21.2 Function Block Code Example

```
int rc;
MMC_RESSNAPSHOT_IN          stResSnapshot_in;
MMC_RESSNAPSHOT_OUT         stResSnapshot_out;
//
// Inserting the structure parameters:
stResSnapshot_in.dummy = 1;      //Dummy input
//
rc = MMC_GetResSnapshotCmd (hConn, &stResSnapshot_in, &stResSnapshot_out);
printf("ErrId[%d]\n", (short)stResSnapshot_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

8.1.22 MMC_GetVersion

Obtains the Maestro version in the output parameter.

```
MMC_LIB_API int MMC_GetVersionCmd(
IN MMC_CONNECT_HNDL hConn,
OUT MMC_GET_VER_OUT* sVersion
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

sVersion

Version data structure. Points to the **MMC_GET_VER_OUT** output structure receiving information, as a result of calling the MMC_GET_VER_OUT structure.

Remarks

This version reference consists of two parts:

- Firmware
- UBoot (bootloader which the Maestro CPU reads from the flash upon startup)

The maximal length of the MMC_GetVersion output string is 120 characters.

Scope

All

MMC_GET_VER_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_GET_VER_IN;
```

Parameters

dummy

Any dummy value.

MMC_GET_VER_OUT Structure

```
typedef struct{
unsigned int uiUbootVer;
unsigned short usStatus;
short sErrorID;
char cFirst;
char cSecond;
char cThird;
char cFourth;
}MMC_GET_VER_OUT;
```

Parameters

uiUbootVer

U-Boot version. Any +ve integer.

cFirst

First byte of the Maestro version. Any character.

cSecond

Second byte of the Maestro version. Any character.

cThird

Third byte of the Maestro version. Any character.

cFourth

Fourth byte of the Maestro version. Any character.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-23 describes the function block for MMC_GetVersion as applied within the IEC 61131 programming.

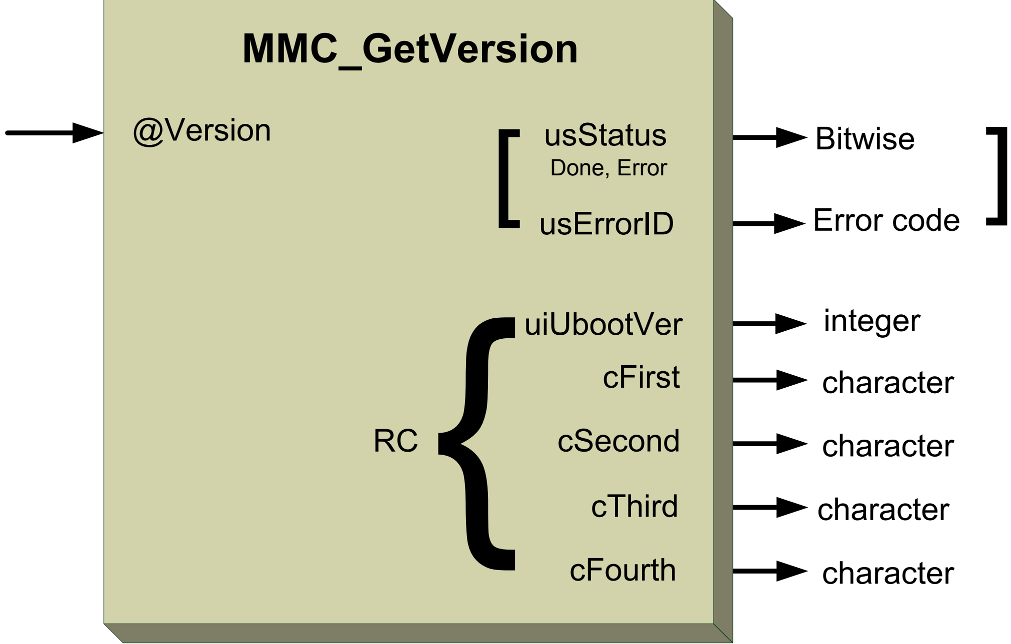


Figure 8-23: MMC_GetVersion function block

8.1.22.2 Function Block Code Example

```
int rc;
MMC_GET_VER_IN         stGetVer_in;
MMC_GET_VER_OUT        stGetVer_out;
//
// Inserting the structure parameters:
stGetVer_in.dummy = 1;     //Function block handle
//
rc = MMC_GetVersionCmd (hConn, &stGetVer_out);
printf("Version Status[%ld] ErrId[%d]\n", (long int)stGetVer_out.uiUbootVer,
(short)stGetVer_out.sErrorID);
printf("Version Status[%ld] [%ld] [%ld] [%ld]\n", (long int)stGetVer_out.cFirst, (long
int)stGetVer_out.cSecond, (long int)stGetVer_out.cThird, (long int)stGetVer_out.cFourth);
if (rc != 0)
{
    HandleError();
}
```

8.1.23 MMC_GetVersionEx

Obtains the Maestro extended version in the output parameter.

```
MMC_LIB_API int MMC_GetVersionExCmd(
IN MMC_CONNECT_HNDL hConn,
OUT MMC_GET_VEREX_OUT* sVersion
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

sVersion

[OUT] Version data structure. Points to the **MMC_GET_VEREX_OUT** output structure receiving information, as a result of calling the MMC_GET_VEREX_OUT structure.

Remarks

This version reference consists of two parts:

- Firmware
- UBoot (bootloader which the Maestro CPU reads from the flash upon startup)

The maximal length of the MMC_GetVersion output string is 120 characters.

Scope

All

MMC_GET_VEREX_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_GET_VEREX_IN;
```

Parameters

dummy

Any dummy value.

MMC_GET_VEREX_OUT Structure

```
typedef struct{
unsigned short usStatus;            //Returned command status
short sErrorID;                     //Returned command error ID
char pcData[MAX_GETVERSION_CHARS];
} MMC_GET_VEREX_OUT;
```

Parameters

pcData[MAX_GETVERSION_CHARS]

Output string containing details about the GMAS firmware version, creation firmware date and time, uboot version and processor revision ID. Only necessary to print the string for convenience.

For example, output from a Maestro with current firmware version: 1115 is:
creation date: Oct 18 2012, creation time: 16:53:29, uboot version: 9, revision ID: 0x10

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-24 describes the function block for MMC_GetVersionEx as applied within the IEC 61131 programming.

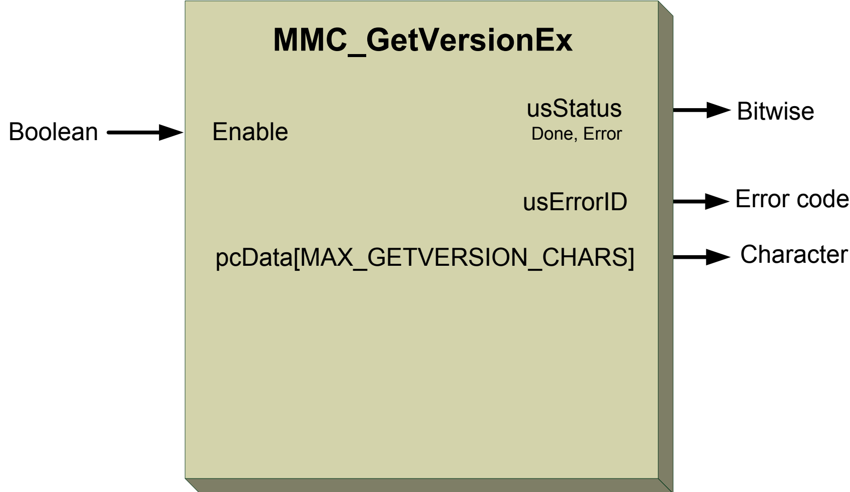


Figure 8-24: MMC_GetVersionEx function block

8.1.23.2 Function Block Code Example

```
int rc;
MMC_GET_VER_IN          stGetVer_in;
MMC_GET_VER_OUT         stGetVer_out;
//
// Inserting the structure parameters:
stGetVer_in.dummy = 1;      //Function block handle
//
MMC_GET_VEREX_OUT GetVerOut;
MMC_CONNECT_HNDL ConnHndl ;

rc = MMC_GetVersionExCmd(ConnHndl,&GetVerOut);
if(rc != 0)
     {
     printf("MMC_GetVersionExCmd failed, error %d", GetVerOut.sErrorID);
     goto lbl_exit;
     }
printf("%s\n",GetVerOut.pcData);
```

8.1.24 MMC_GetLastError

Returns the last error that occurred in the designated connection.

```
MMC_LIB_API void MMC_GetLastError (
IN MMC_CONNECT_HNDL hConn,
OUT char* chStr,
IN int iSize
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h GMAS Programming(IEC 61331 Program)\ElmoGlobal	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

chStr

[OUT] Pointer to the error string. Character value.

iSize

[IN] Size of the error string. Integer value

Remarks

None

Scope

All

Figure 8-25 describes the function for MMC_GetLastError as applied within the IEC 61131 programming.

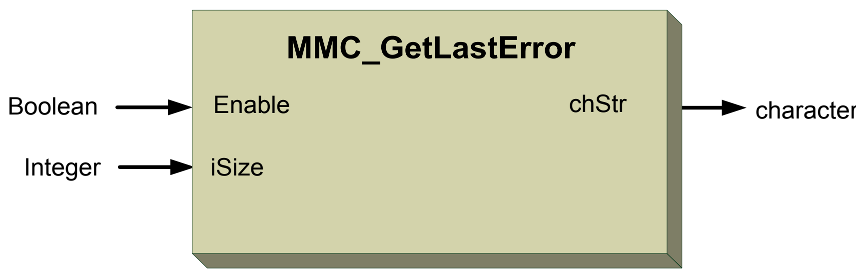


Figure 8-25: MMC_GetLastError function

8.1.25 MMC_InitConnection

Initiates connection to the Maestro server.

```
MMC_LIB_API int MMC_InitConnection(
IN MMC_CONNECTION_TYPE eType,
IN MMC_CONNECTION_PARAM_STRUCT sConnParam,
IN MMC_CB_FUNC pCbFunc,
OUT MMC_CONNECT_HNDL* pHndl
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

eType

[IN] Connection type (IPC(inter process communication) or RPC(remote procedure calls)) with the following possible MMC_CONNECTION_TYPE enumerator values:

MMC_RPC_CONN_TYPE	RPC connection to MMC server
MMC_IPC_CONN_TYPE	IPC connection to MMC server

sConnParam

[IN] Connection parameters. (e.g. IP port, UDP port for callback)

uiTcpPort

TCP Port. Any +ve integer value.

uiCbUdpPort

UDP Port. Any +ve integer value.

ucIp[16]

IP address. 16-bit character.

pCbFunc

[IN] MMC_CB_FUNC pointer to UDP callback function using pCbFunc. No character value or short integer. The source for the parameter is:

GMAS\includes\MMC_definitions.h

pHndl

[OUT] Connection handle output using pHndl, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

Remarks

None

Scope

This function initiates an IPC or RPC connection to the Maestro host.

Figure 8-26 describes the function block for MMC_InitConnection

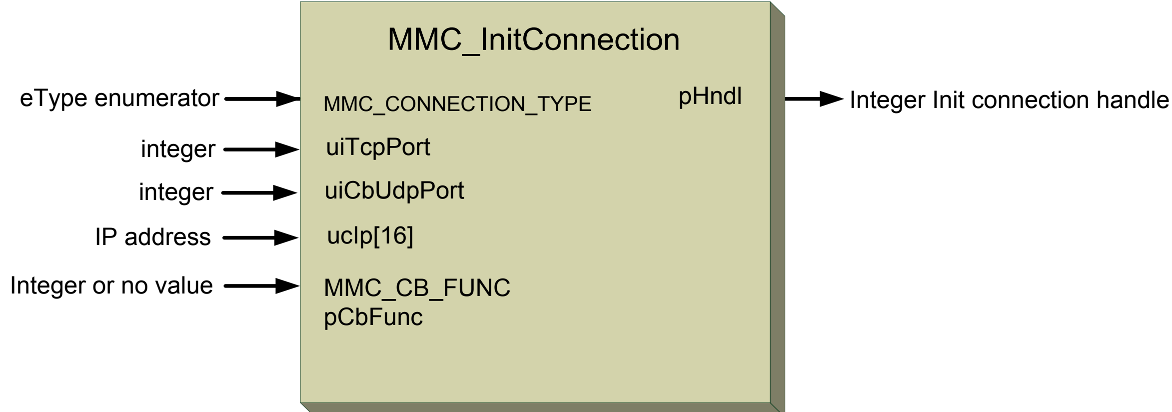


Figure 8-26: MMC_InitConnection function block

8.1.25.1 Function Block Code Example

```
int rc;
MMC_CONNECTION_PARAM_STRUCT stsConnParam;
//
// Inserting the structure parameters:
strcpy ((char*)stsConnParam.ucIp, "255.0.110.0");     //IP address
stsConnParam.uiCbUdpPort    = 1520;                 //UDP Port
stsConnParam.uiTcpPort      = 1910;                 //TCP Port
eType[1]                    = MMC_IPC_CONN_TYPE;    //Connection type
pCbFunc                     = NULL;                 //Pointer to UDP callback function
//
rc = MMC_InitConnection ((MMC_CONNECTION_TYPE) eType, stsConnParam, (MMC_CB_FUNC) pCbFunc,
&hConn);
printf("Connection State[%ld]\n", (long int)(MMC_CONNECT_HNDL) pHndl);
if (rc != 0)
{
    HandleError();
}
```

8.1.26 MMC_IPCInitConnection

Initiates IPC connection to Maestro server.

MMC_LIB_API int MMC_IPCInitConnection(
IN MMC_IPC_CONNECTION_PARAM_STRUCT sConnParam,

IN MMC_CB_FUNC pCbFunc,

OUT MMC_CONNECT_HNDL* pHndl
);

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

sConnParam

[IN] Connection parameters. (e.g. IP port, UDP port for callback) as a function of the parameter MMC_IPC_CONNECTION_PARAM_STRUCT.

typedef struct{
unsigned int uiParams;
} MMC_IPC_CONNECTION_PARAM_STRUCT

Sub-function Parameters

uiParams

Currently: bit0 an explicit connection. All other connections will be closed. +ve valued integer.

uiTcpPort

TCP Port. Any +ve integer value.

uiCbUdpPort

UDP Port. Any +ve integer value.

ucIp[16]

IP address. 16-bit character.

pCbFunc

[IN] MMC_CB_FUNC pointer to UDP callback function using pCbFunc. No character value or short integer. The source for the parameter is:

GMAS\includes\MMC_definitions.h

pHndl

[OUT] Connection handle output using pHndl, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

Remarks

None

Scope

This function is specifically designed for IPC connections.

Figure 8-29 describes the function block for MMC_IPCInitConnection

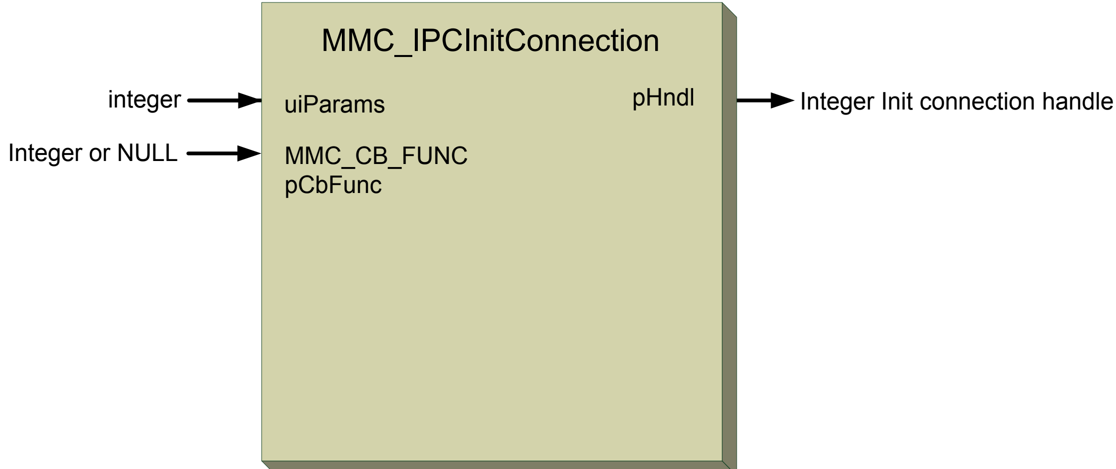


Figure 8-27: MMC_IPCInitConnection function block

8.1.26.1 Function Block Code Example

```
int rc;
MMC_IPC_CONNECTION_PARAM_STRUCT     stsConnParam;
//
// Inserting the structure parameters:
stsConnParam.uiParams  = 1;     //IP address
pCbFunc                = NULL;  //Pointer to UDP callback function
//
rc = MMC_IPCInitConnection (stsConnParam, (MMC_CB_FUNC) pCbFunc, &hConn);
printf("Connection State[%ld]\n", (long int)(MMC_CONNECT_HNDL) pHndl);
if (rc != 0)
{
    HandleError();
}
```

8.1.27 MMC_LoadParam

Loads the axis, group, and global parameters from the xml file at the location:

//mnt/jffs/MMC/config/parameters/MMCParameters.xml

to the Maestro

```
MMC_LIB_API int MMC_LoadParamCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_LOADPARAM_IN* pInParam,
OUT MMC_LOADPARAM_OUT* pOutParam
);
```

Motion Mode	NC - Not relevant	Distributed - Not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_LOADPARAM_IN** input data structure using the MMC_LoadParam function.

pOutParam

Points to the **MMC_LOADPARAM_OUT** output structure receiving information, as a result of calling the MMC_LoadParam function.

Remarks

None

Scope

All

MMC_LOADPARAM_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_LOADPARAMAXIS_IN;
```

Parameters

dummy

Any +ve dummy characters

MMC_LOADPARAM_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_LOADPARAMGLOBAL_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-28 describes the function block for MMC_LoadParam

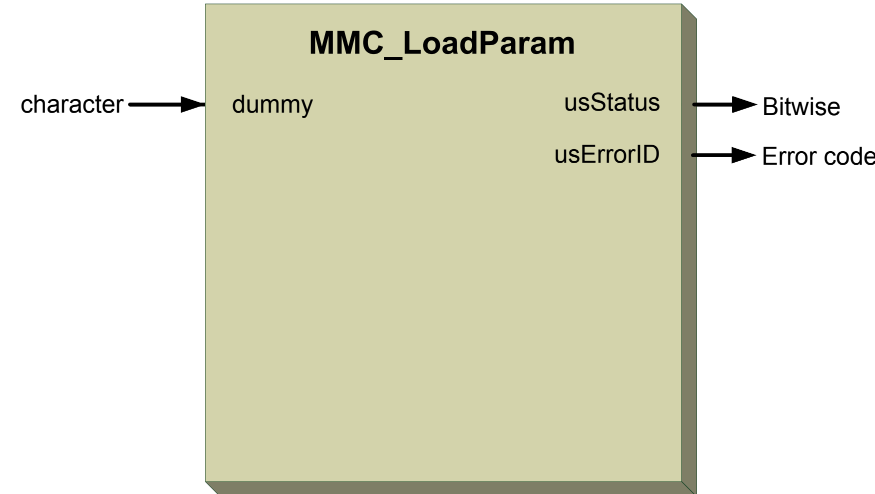


Figure 8-28: MMC_LoadParam function block

8.1.28 MMC_RpcInitConnection

Initiates RPC connection to Maestro server.

```
MMC_LIB_API int MMC_RpcInitConnection(
IN MMC_CONNECTION_TYPE eType,
IN MMC_CONNECTION_PARAM_STRUCT sConnParam,
IN MMC_CB_FUNC pCbFunc,
IN char* cpHostIPAddr,
OUT MMC_CONNECT_HNDL* pHndl
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

eType

[IN] Connection type (IPC(inter process communication) or RPC(remote procedure calls)) with the following possible MMC_CONNECTION_TYPE enumerator values:

MMC_RPC_CONN_TYPE	RPC connection to MMC server
MMC_IPC_CONN_TYPE	IPC connection to MMC server

sConnParam

[IN] Connection parameters. (e.g. IP port, UDP port for callback)

uiTcpPort

TCP Port. Any +ve integer value.

uiCbUdpPort

UDP Port. Any +ve integer value.

ucIp[16]

IP address. 16-bit character.

pCbFunc

[IN] MMC_CB_FUNC pointer to UDP callback function using pCbFunc. No character value or short integer. The source for the parameter is:

GMAS\includes\MMC_definitions.h

cpHostIPAddr

[IN] Host IP Address for multiple NIC support. Values accepted are the IP address characters. The source for the parameter is:

GMAS\includes\MMC_definitions.h

pHndl

[OUT] Connection handle output using pHndl, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions.

Returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

Remarks

None

Scope

This function is specifically designed for RPC connections, as it includes the parameter *cpHostIPAddr*, the host IP address for NIC support.

Figure 8-29 describes the function block for MMC_RpcInitConnection

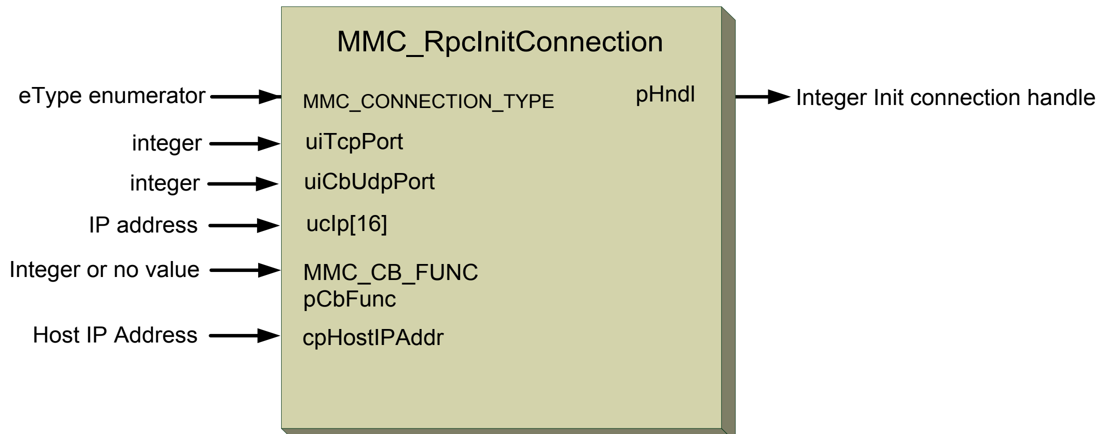


Figure 8-29: MMC_RpcInitConnection function block

8.1.28.1 Function Block Code Example

```
int rc;
MMC_CONNECTION_PARAM_STRUCT stsConnParam;
//
// Inserting the structure parameters:
strcpy ((char*)stsConnParam.ucIp, "255.0.110.0");      //IP address
stsConnParam.uiCbUdpPort    = 1520;                     //UDP Port
stsConnParam.uiTcpPort      = 1910;                     //TCP Port
eType[1]                    = MMC_IPC_CONN_TYPE;        //Connection type
pCbFunc                     = NULL;                     //Pointer to UDP callback function
strcpy ((char*)cpHostIPAddr, "97.110.110.1");           //Host IP Address
//
rc = MMC_RpcInitConnection ((MMC_CONNECTION_TYPE) eType, stsConnParam, (MMC_CB_FUNC) pCbFunc,
cpHostIPAddr, &hConn);
printf("Connection State[%ld]\n", (long int)(MMC_CONNECT_HNDL) pHndl);
if (rc != 0)
{
    HandleError();
}
```

8.1.29 MMC_RpcInitConnectionEx

Initiates RPC connection to Maestro server.

MMC_LIB_API int MMC_RpcInitConnectionEx(
IN MMC_CONNECTION_TYPE eType,
IN MMC_CONNECTION_PARAM_STRUCT sConnParam,
IN MMC_MB_CLBK pCbFunc,
IN char* cpHostIPAddr,
OUT MMC_CONNECT_HNDL* pHndl
);

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

eType

[IN] Connection type (IPC(inter process communication) or RPC(remote procedure calls)) with the following possible MMC_CONNECTION_TYPE enumerator values:

MMC_RPC_CONN_TYPE	RPC connection to MMC server
MMC_IPC_CONN_TYPE	IPC connection to MMC server

sConnParam

[IN] Connection parameters. (e.g. IP port, UDP port for callback)

uiTcpPort

TCP Port. Any +ve integer value.

uiCbUdpPort

UDP Port. Any +ve integer value.

ucIp[16]

IP address. 16-bit character.

pCbFunc

[IN] MMC_MB_CLBK pointer to UDP callback function using pCbFunc. No character value or short integer. The source for the parameter is:

GMAS\includes\MMC_definitions.h

cpHostIPAddr

[IN] Host IP Address for multiple NIC support. Values accepted are the IP address characters. The source for the parameter is:

GMAS\includes\MMC_definitions.h

pHndl

[OUT] Connection handle output using pHndl, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common

throughout all Maestro functions.

Returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

Remarks

None

Scope

This function is specifically designed for RPC connections, as it includes the parameter *cpHostIPAddr*, the host IP address for NIC support.

Figure 8-30 describes the function block for MMC_RpcInitConnection.

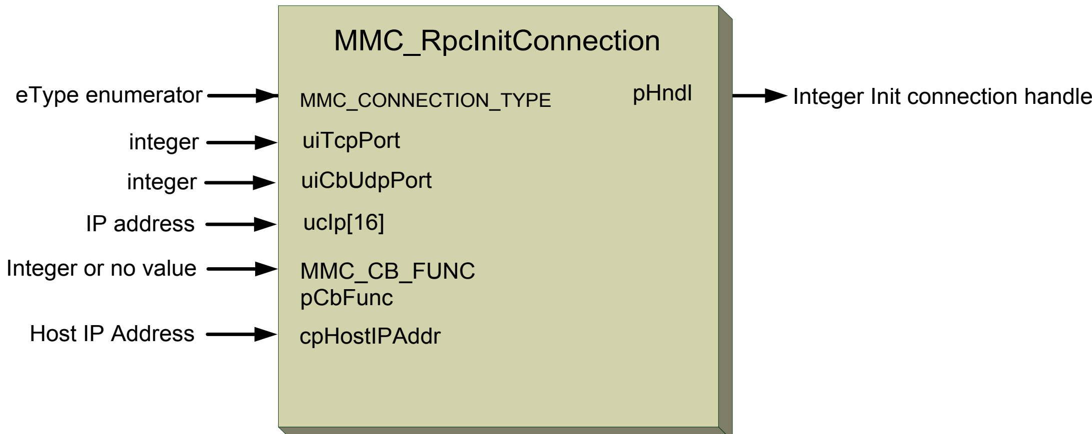


Figure 8-30: MMC_RpcInitConnection function block

8.1.29.1 Function Block Code Example

```
int rc;
MMC_CONNECTION_PARAM_STRUCT stsConnParam;
//
// Inserting the structure parameters:
strcpy ((char*)stsConnParam.ucIp, "255.0.110.0");      //IP address
stsConnParam.uiCbUdpPort    = 1520;                    //UDP Port
stsConnParam.uiTcpPort      = 1910;                    //TCP Port
eType[1]                    = MMC_IPC_CONN_TYPE;       //Connection type
pCbFunc                     = NULL;                    //Pointer to UDP callback function
strcpy ((char*)cpHostIPAddr, "97.110.110.1");          //Host IP Address
//
rc = MMC_RpcInitConnection ((MMC_CONNECTION_TYPE) eType, stsConnParam, (MMC_CB_FUNC) pCbFunc,
cpHostIPAddr, &hConn);
printf("Connection State[%ld]\n", (long int)(MMC_CONNECT_HNDL) pHndl);
if (rc != 0)
{
    HandleError();
}
```

8.1.30 MMC_ResetMultiAxisControl

Internal reset of the Maestro multi-axis control. Allows the Maestro's CPU to reset

```
MMC_LIB_API int MMC_ResetMultiAxisControl(
IN MMC_CONNECT_HNDL hConn,
IN MMC_EXIT_APP_IN *pInParam,
OUT MMC_EXIT_APP_OUT* pOutParam
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_EXIT_APP_IN** input data structure using the MMC_ResetMultiAxisControl function.

pOutParam

Points to the **MMC_EXIT_APP_OUT** output structure receiving information, as a result of calling the MMC_ResetMultiAxisControl function.

Remarks

This function closes MultiAxisControl by closing connection,shared memory, and nc_drv

Scope

All

MMC_EXIT_APP_IN Structure

```
typedef struct{
unsigned char ucEnable;
}MMC_EXIT_APP_IN;
```

Parameters

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_EXIT_APP_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_EXIT_APP_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-31 describes the function block for MMC_ResetMultiAxisControl

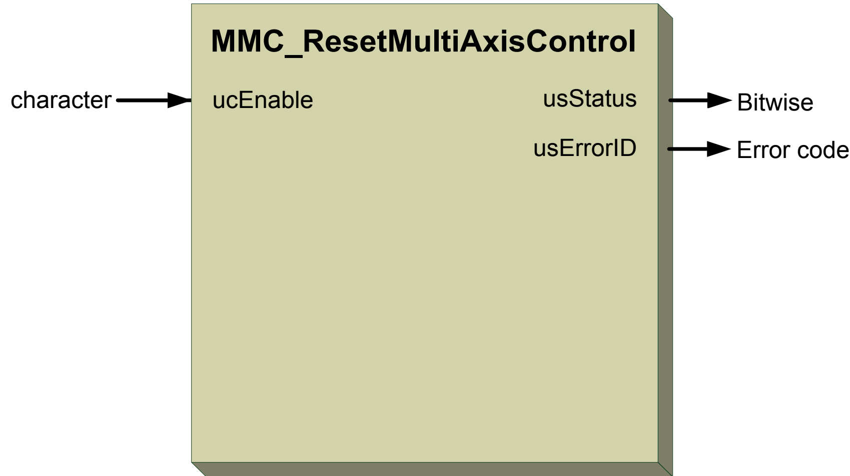


Figure 8-31: MMC_ResetMultiAxisControl function block

8.1.30.2 Function Block Code Example

```
int rc;
MMC_EXIT_APP_IN         stExitApp_in;
MMC_EXIT_APP_OUT        stExitApp_out;
//
// Inserting the structure parameters:
stExitApp_in.ucEnable  = 1;         // Enabled
//
rc = MMC_ResetMultiAxisControl (hConn, &stExitApp_in, &stExitApp_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.31 MMC_ResExportFile

Copies a requested file from the Maestro to the host via TFTP.

```
MMC_LIB_API int MMC_ResExportFileCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_RESEXPORTFILE_IN* pInParam,
OUT MMC_RESEXPORTFILE_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_RESEXPORTFILE_IN** input data structure using the MMC_ResExportFile function.

pOutParam

Points to the **MMC_RESEXPORTFILE_OUT** output structure receiving information, as a result of calling the MMC_ResExportFile function.

Remarks

The User file that can be downloaded is located in the following directory:

/mnt/jffs/user

Scope

All

MMC_RESEXPORTFILE_IN Structure

```
typedef struct{
char pFName[33];
char pServer[17];
char pFilePath[101];
char ucDownloadType;
}MMC_RESEXPORTFILE_IN;
```

Parameters

pFName[33]

Resource File Name. Any +ve character value with a limit of 33 characters

pServer[17]

Server name. Any +ve character value with a limit of 17 characters.

pFilePath[101]

Server file path. Any +ve character value with a limit of 101 characters.

ucDownloadType

File type to be copied. The following IDs are used to characterize the enumerator parameter:

The following resource enumerators define the download configuration resource file types for the variable MMC_DOWNLOAD_TYPE_ENUM.

ID	Parameters	Explanation
0	MMC_PARAMETER_FILE_DOWNLOAD	Download parameter file
1	MMC_RESOURCE_FILE_DOWNLOAD	Download resource file
2	MMC_SNAPSHOT_RESOURCE_FILE_DOWNLOAD	Download resource snapshot file
3	MMC_ETHERCAT_CFG_FILE_DOWNLOAD	Download Resource EtherCAT file
4	MMC_PERSONALITY_FILE_DOWNLOAD	Download Maestro Personality file
5	MMC_USER_FILE_DOWNLOAD	Download Maestro user file

MMC_RESEXPORTFILE_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_RESEXPORTFILE_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-32 describes the function block for MMC_ResExportFile

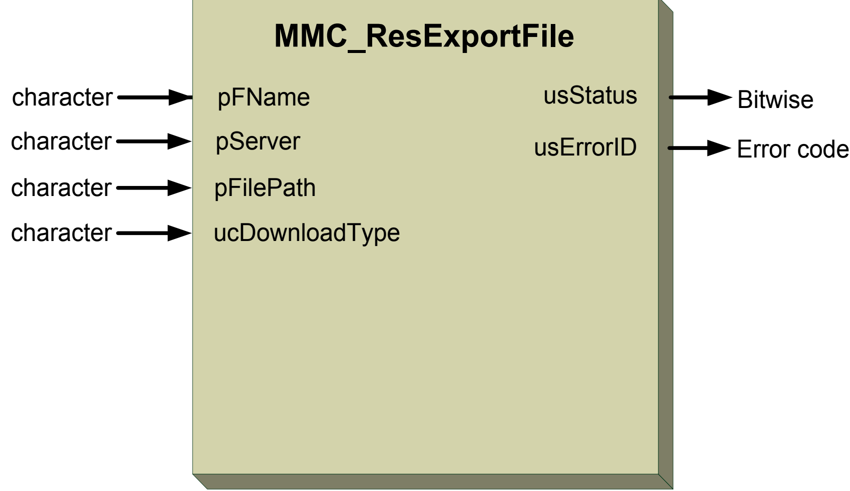


Figure 8-32: MMC_ResExportFile function block

8.1.31.2 Function Block Code Example

```
int rc;
MMC_RESEXPORTFILE_IN        stResExportFile_in;
MMC_RESEXPORTFILE_OUT       stResExportFile_out;
//
// Inserting the structure parameters:
strcpy(stResExportFile_in.pFName,"Resource_File_Name");         //Resource File Name
strcpy(stResExportFile_in.pServer,"ServerName");       //Server Name
strcpy(stResExportFile_in.pFilePath,"Server_File_Path_Name");  //Server file path
stResExportFile_in.ucDownloadType    = 1;                       //Download type
//
rc = MMC_ResExportFileCmd (hConn, &stResExportFile_in, &stResExportFile_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.32 MMC_ResImportFile

Copies a requested file from host to the Maestro via TFTP.

```
MC_LIB_API int MMC_ResImportFileCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_RESIMPORTFILE_IN* pInParam,
OUT MMC_RESIMPORTFILE_OUT* pOutParam
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_RESIMPORTFILE_IN** input data structure using the MMC_ResImportFile function.

pOutParam

Points to the **MMC_RESIMPORTFILE_OUT** output structure receiving information, as a result of calling the MMC_ResImportFile function.

Remarks

The User file that can be uploaded will be copied to the following directory:

/mnt/jffs/user

Scope

All

MMC_RESIMPORTFILE_IN Structure

```
typedef struct{
char pFName[33];
char pServer[17];
char ucDownloadType;
}MMC_RESIMPORTFILE_IN;
```

Parameters

pFName[33]

Resource File Name. Any +ve character value with a limit of 33 characters

pServer[17]

Server name. Any +ve character value with a limit of 17 characters.

ucDownloadType

File type to be copied. The following IDs are used to characterize the enumerator parameter:

The following resource enumerators define the download configuration resource file types for the variable MMC_DOWNLOAD_TYPE_ENUM.

ID	Parameters	Explanation
0	MMC_PARAMETER_FILE_DOWNLOAD	Download parameter file
1	MMC_RESOURCE_FILE_DOWNLOAD	Download resource file
2	MMC_SNAPSHOT_RESOURCE_FILE_DOWNLOAD	Download resource snapshot file
3	MMC_ETHERCAT_CFG_FILE_DOWNLOAD	Download Resource EtherCAT file
4	MMC_PERSONALITY_FILE_DOWNLOAD	Download Maestro Personality file
5	MMC_USER_FILE_DOWNLOAD	Download Maestro user file

MMC_RESIMPORTFILE_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_RESIMPORTFILE_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-33 describes the function block for MMC_ResImportFile

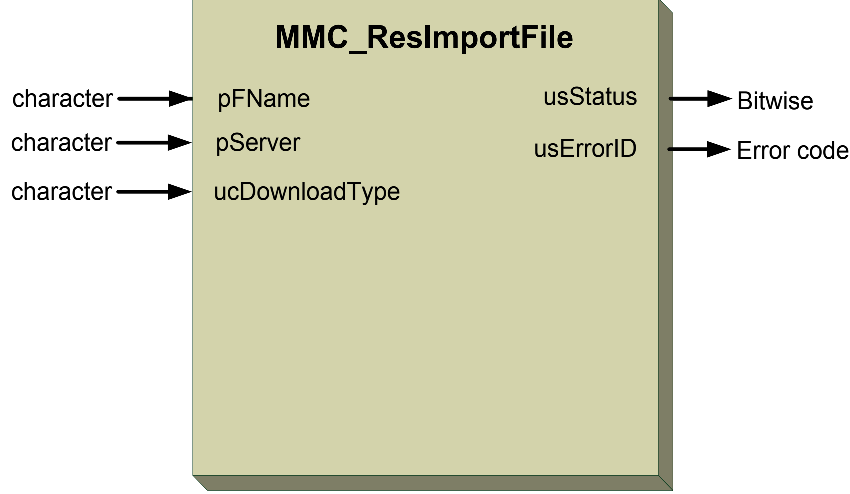


Figure 8-33: MMC_ResImportFile function block

8.1.32.2 Function Block Code Example

```
int rc;
MMC_RESIMPORTFILE_IN        stResImportFile_in;
MMC_RESIMPORTFILE_OUT       stResImportFile_out;
//
// Inserting the structure parameters:
strcpy(stResImportFile_in.pFName, "Resource_File_Name");           //Resource File Name
strcpy(stResImportFile_in.pServer, "ServerName");          //Server Name
stResImportFile_in.ucDownloadType   = 1;                   //Download type
//
rc = MMC_ResImportFileCmd (hConn, &stResImportFile_in, &stResImportFile_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.33 MMC_SaveParam

Save and/or update axes, group, and global parameters from the Maestro to a file at:

//mnt/jffs/MMC/config/parameters/MMCParameters.xml

```
MMC_LIB_API int MMC_SaveParamCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SAVEPARAM_IN* pInParam,
OUT MMC_SAVEPARAM_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed - Not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SAVEPARAM_IN** input data structure using the MMC_SaveParam function.

pOutParam

Points to the **MMC_SAVEPARAM_OUT** output structure receiving information, as a result of calling the MMC_SaveParam function.

Remarks

None

Scope

All

MMC_SAVEPARAM_IN Structure

```
typedef struct{
unsigned char dummy;
} MMC_SAVEPARAMAXIS_IN;
```

Parameters

dummy

Dummy input. Any +ve character value.

MMC_SAVEPARAM_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
} MMC_SAVEPARAMAXIS_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-34 describes the function block for MMC_SaveParam.

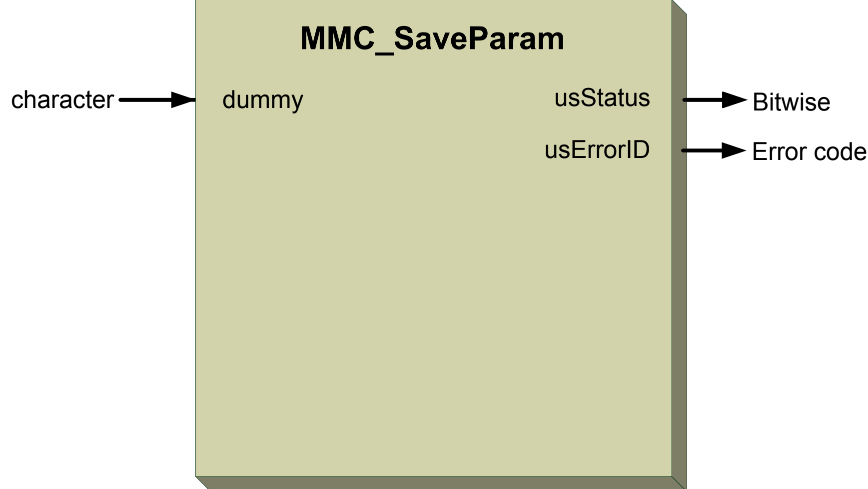


Figure 8-34: MMC_SaveParam function block

8.1.33.2 Function Block Code Example

```
int rc;
MMC_SAVEPARAM_IN      stSaveParam_in;
MMC_SAVEPARAM_OUT     stSaveParam_out;
//
// Inserting the structure parameters:
stSaveParam_in.dummy       = 1;     // Dummy input
//
rc = MMC_SaveParamCmd (hConn, &stSaveParam_in, &stSaveParam_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.34 MMC_SetEnquireFbStatus

Sets the state global parameter Receive FB status in EASII.

```
MMC_LIB_API int MMC_SetEnquireFbStatusCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SETENQUIREFBSTATUS_IN* pInParam,
OUT MMC_SETENQUIREFBSTATUS_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SETENQUIREFBSTATUS_IN** input data structure using the MMC_SetEnquireFbStatus function.

pOutParam

Points to the **MMC_SETENQUIREFBSTATUS_OUT** output structure receiving information, as a result of calling the MMC_SetEnquireFbStatus function.

Remarks

This function is only for IPC programming in C, and C++, and allows the user to control the release mode of the function blocks. By default it is read from the Maestro resource file and should be set to False (0). If set to True (1) , the FBs will not be released from the FB queue until the user releases them implicitly. For EAS and any RPC programming, it will be set automatically to False (0) without user control.

In the IEC61131-3 program, the function parameter is also read from the Maestro resource file and is automatically set to True (1). This is because the IEC61131-3 function blocks require and provide real data from the Maestro and drives attached, during their motion, as they load and run each function block in turn. The function block will only be released from the queue when the Maestro and drives perform that function block and move to the next function block.

Scope

All

MMC_SETENQUIREFBSTATUS_IN Structure

```
typedef struct mmc_setenquirefbstatus_in{
unsigned char ucStatus;
} MMC_SETENQUIREFBSTATUS_IN;
```

Parameters

eStatus

The status of the function block, with +ve character values of 0 - 255.

MMC_SETENQUIREFBSTATUS_OUT Structure

```
typedef struct mmc_setenquirefbstatus_out{
unsigned short usStatus;
short sErrorID;
} MMC_SETENQUIREFBSTATUS_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-35 describes the function for MMC_SetEnquireFbStatus

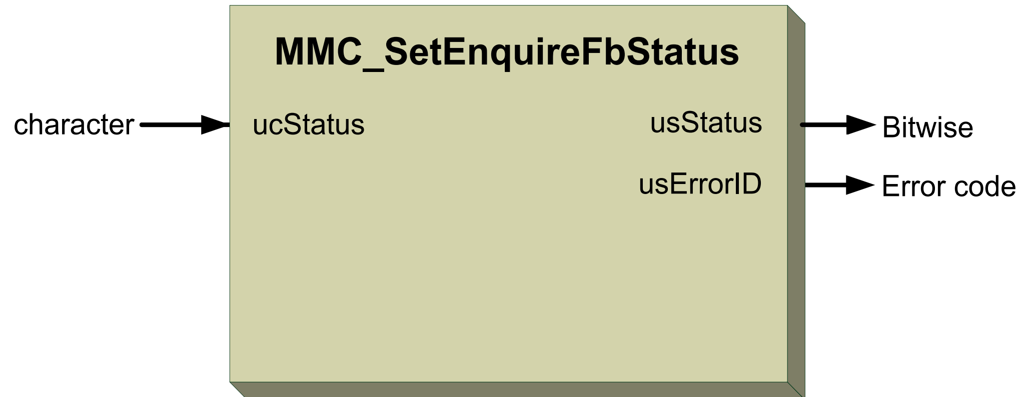


Figure 8-35: MMC_SetEnquireFbStatus function

8.1.35 MMC_SetDefaultParameters

Sets the Maestro default manufacturer parameters to a specific Axis, or Group in the Maestro.

```
MMC_LIB_API int MMC_SetDefaultParametersCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SETDEFAULTPARAMETERS_IN* pInParam,
OUT MMC_SETDEFAULTPARAMETERS_OUT* pOutParam
);
```

Motion Mode NC - Not relevant Distributed - Not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SETDEFAULTPARAMETERS_IN** input data structure using the MMC_SetDefaultParameters function.

pOutParam

Points to the **MMC_SETDEFAULTPARAMETERS_OUT** output structure receiving information, as a result of calling the MMC_SetDefaultParameters function.

Remarks

Resets the default manufacturer parameters to the specific axis/group in the Maestro.

Scope

All

MMC_SETDEFAULTPARAMETERS_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_SETDEFAULTPARAMETERS_IN;
```

Parameters

dummy

Any dummy value

MMC_SETDEFAULTPARAMETERS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETDEFAULTPARAMETERS_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-36 describes the function block for MMC_SetDefaultParameters

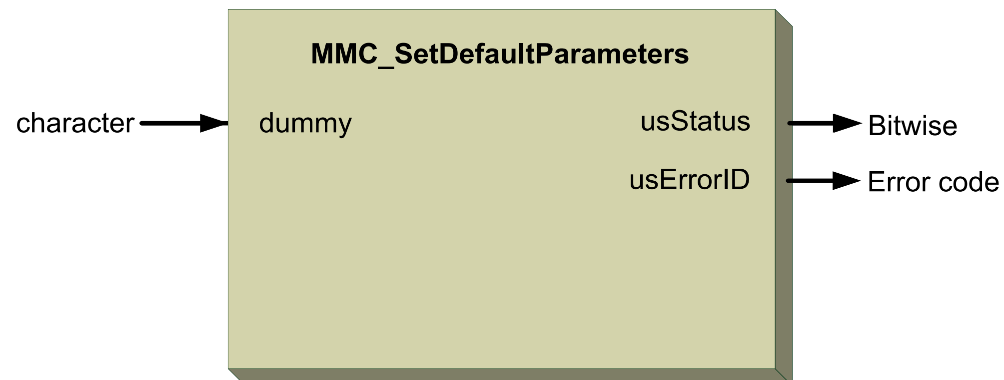


Figure 8-36: MMC_SetDefaultParameters function block

8.1.36 MMC_SetDefaultParametersGlobal

Sets the Maestro default manufacturer global parameters in the Maestro.

```
MMC_LIB_API int MMC_SetDefaultParametersGlobalCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SETDEFAULTPARAMETERSGLOBAL_IN* pInParam,
OUT MMC_SETDEFAULTPARAMETERSGLOBAL_OUT* pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed - Not relevant
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SETDEFAULTPARAMETERSGLOBAL_IN** input data structure using the MMC_SetDefaultParametersGlobal function.

pOutParam

Points to the **MMC_SETDEFAULTPARAMETERSGLOBAL_OUT** output structure receiving information, as a result of calling the MMC_SetDefaultParametersGlobal function.

Remarks

Resets the default manufacturer global parameters to the Maestro.

Scope

All

MMC_SETDEFAULTPARAMETERSGLOBAL_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_SETDEFAULTPARAMETERSGLOBAL_IN;
```

Parameters

dummy

Any dummy value.

MMC_SETDEFAULTPARAMETERSGLOBAL_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETDEFAULTPARAMETERSGLOBAL_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-37 describes the function block for MMC_SetDefaultParametersGlobal

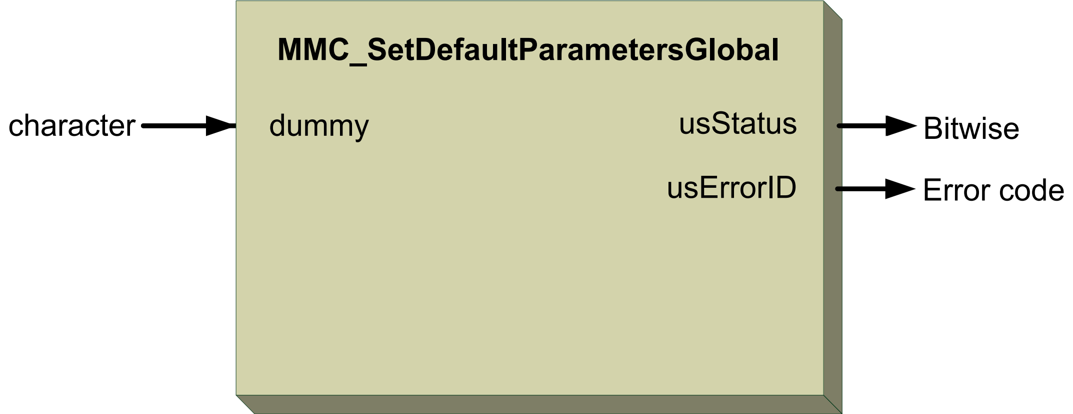


Figure 8-37: MMC_SetDefaultParametersGlobal function block

8.1.37 MMC_SetIsToLoadGlobalParams

Defines a flag whether to load or not, the global parameters, when updating the global parameters from a file to the Maestro.

```
MMC_LIB_API int MMC_SetIsToLoadGlobalParamsCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SETISTOLOADGLOBALPARAMS_IN* pInParam,
OUT MMC_SETISTOLOADGLOBALPARAMS_OUT* pOutParam
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SETISTOLOADGLOBALPARAMS_IN** input data structure using the MMC_SetIsToLoadGlobalParams function.

pOutParam

Points to the **MMC_SETISTOLOADGLOBALPARAMS_OUT** output structure receiving information, as a result of calling the MMC_SetIsToLoadGlobalParams function.

Remarks

In general, there are at two instances when updating the global parameters from a file to the Maestro:

- When using MMC_LoadParams
- After restarting (Power On or software restart) the Maestro

Scope

All

MMC_SETISTOLOADGLOBALPARAMS_IN Structure

```
typedef struct{
unsigned char ucValue;
}MMC_SETISTOLOADGLOBALPARAMS_IN;
```

Parameters

ucValue

The function recieves either 0 (not required to load the set global parameters) or 1 (required to load the set global parameters). +ve integer value

MMC_SETISTOLOADGLOBALPARAMS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETISTOLOADGLOBALPARAMS_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-38 describes the function block for MMC_SetIsToLoadGlobalParams

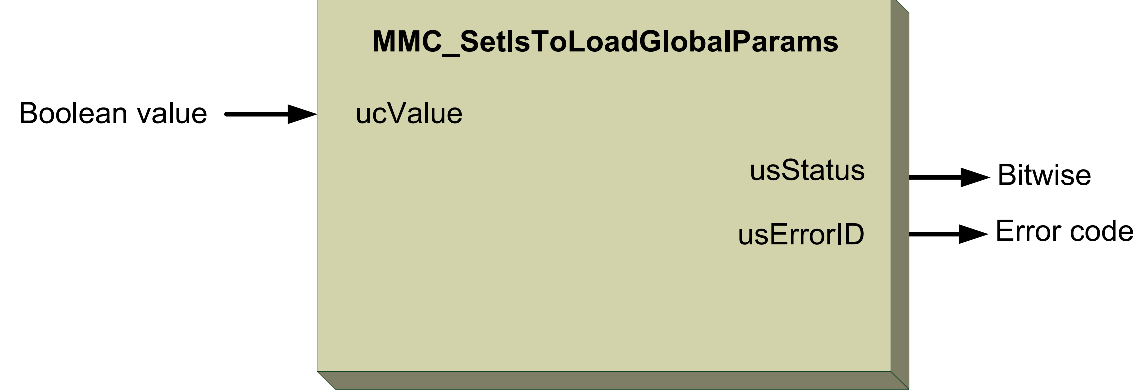


Figure 8-38: MMC_SetIsToLoadGlobalParams function block

8.1.38 MMC_ShowNodeStat

Displays the debug information for an Axis/Group.

```
MMC_LIB_API int MMC_ShowNodeStatCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SHOWNODESTAT_IN* pInParam,
OUT MMC_SHOWNODESTAT_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SHOWNODESTAT_IN** input data structure using the MMC_ShowNodeStat function.

pOutParam

Points to the **MMC_SHOWNODESTAT_OUT** output structure receiving information as a result of calling the MMC_ShowNodeStat function.

Remarks

None

Scope

All

MMC_SHOWNODESTAT_IN Structure

```
typedef struct{
unsigned int uiHndl;
}MMC_SHOWNODESTAT_IN;
```

Parameters

uiHndl

Requested function block handle. Integer with any +ve value

MMC_SHOWNODESTAT_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SHOWNODESTAT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 8-39 describes the function block for MMC_ShowNodeStat

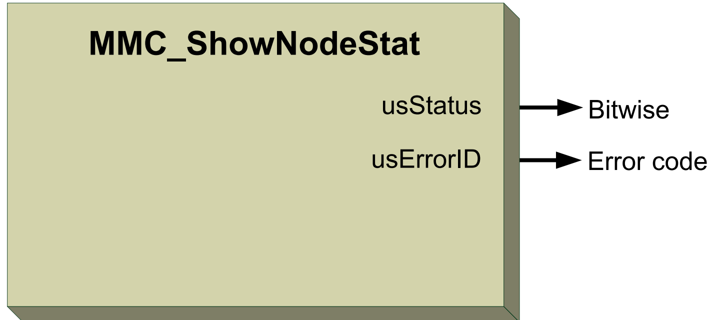


Figure 8-39: MMC_ShowNodeStat function block

8.1.38.2 Function Block Code Example

```
int rc;
MMC_SHOWNODESTAT_IN         stShowNodeStat_in;
MMC_SHOWNODESTAT_OUT        stShowNodeStat_out;
//
// Inserting the structure parameters:
stShowNodeStat_in.uiHndl    = 2;     // Requested function block handle
//
rc = MMC_ShowNodeStatCmd (hConn, iAxisRef, &stShowNodeStat_in, &stShowNodeStat_out);
if (rc != 0)
{
    HandleError();
}
```

8.1.39 MMC_GetActiveAxesNum

Displays the number of active axes attached and managed by the Maestro.

```
MMC_LIB_API int MMC_GetActiveAxesNum(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETACTIVEAXESNUM_IN* pInParam,
OUT MMC_GETACTIVEAXESNUM_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETACTIVEVECTORSNUM_IN** input data structure using the MMC_GetActiveAxesNum function.

pOutParam

Points to the **MMC_GETACTIVEVECTORSNUM_OUT** output structure receiving information as a result of calling the MMC_GetActiveAxesNum function.

Remarks

his function will provide this basic information without opening the Maestro Personality file.

Scope

All

MMC_GETACTIVEAXESNUM_IN Structure

```
typedef struct {
unsigned char dummy;
}MMC_GETACTIVEAXESNUM_IN;
```

Parameters

dummy

Any dummy value.

MMC_GETACTIVEAXESNUM_OUT Structure

```
typedef struct {
int iActiveAxesNum;
unsigned short usStatus;
short sErrorID;
}MMC_GETACTIVEAXESNUM_OUT;
```

Parameters

iActiveAxesNum

Provides the number of active axes. Any +ve integer value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 8-40 describes the function block for MMC_GetActiveAxesNum

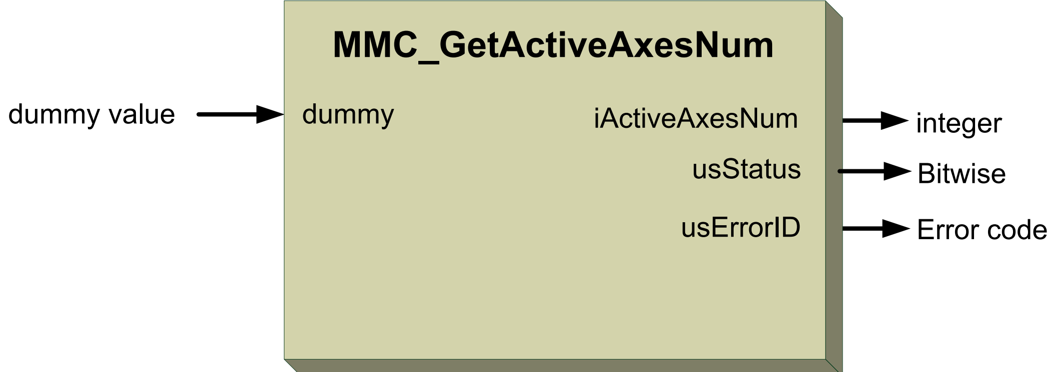


Figure 8-40: MMC_GetActiveAxesNum function block

8.1.40 MMC_ToggleConsoleOutput

Toggles the Console output. This function is not available at this moment.

```
MMC_LIB_API int MMC_ToggleConsoleOutputCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_TOGGLECONSOLEOUTPUT_IN* pInParam,
OUT MMC_TOGGLECONSOLEOUTPUT_OUT* pOutParam
);
```

Motion Mode NC – Supported? Distributed – Supported?

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_TOGGLECONSOLEOUTPUT_IN** input data structure using the MMC_ToggleConsoleOutput function.

pOutParam

Points to the **MMC_TOGGLECONSOLEOUTPUT_OUT** output structure receiving information as a result of calling the MMC_ToggleConsoleOutput function.

Remarks

None

Scope

All

MMC_TOGGLECONSOLEOUTPUT_IN Structure

```
typedef struct mmc_toggleconsoleoutput_in{
unsigned char ucEnable;
} MMC_TOGGLECONSOLEOUTPUT_IN;
```

Parameters

ucEnable

This parameter is not in use and is no longer relevant. Dummy parameter.

MMC_TOGGLECONSOLEOUTPUT_OUT Structure

```
typedef struct mmc_toggleconsoleoutput_out{
short sErrorID;
} MMC_TOGGLECONSOLEOUTPUT_OUT;
```

Parameters

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 8-41 describes the function block for MMC_ToggleConsoleOutput

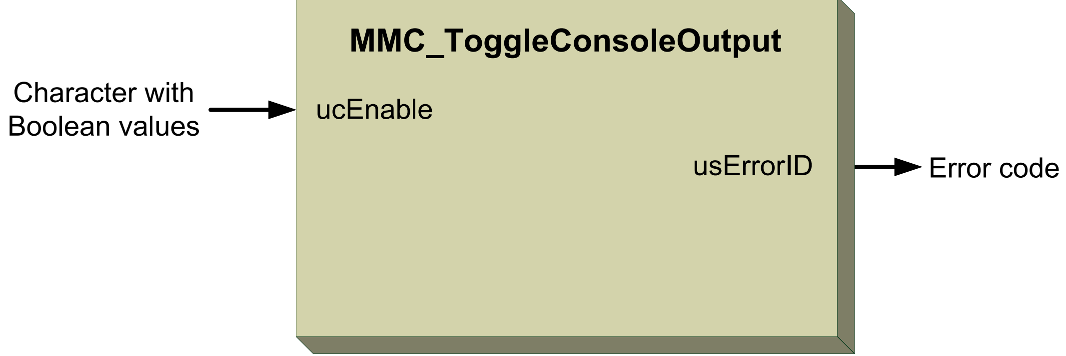


Figure 8-41: MMC_ToggleConsoleOutput function block

8.1.41 MMC_GetCyclesCounter

Obtains the Maestro cycles counter value

MMC_LIB_API int MMC_GetCyclesCounterCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETCYCLESCOUNTER_IN* pInParam,
OUT MMC_GETCYCLESCOUNTER_OUT* pOutParam
);

Motion Mode NC – Supported Distributed – Supported

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETCYCLESCOUNTER_IN** input data structure using the MMC_GetCyclesCounter function.

pOutParam

Points to the **MMC_GETCYCLESCOUNTER_OUT** output structure receiving information as a result of calling the MMC_GetCyclesCounter function.

Remarks

None

Scope

All

MMC_GETCYCLESCOUNTER_IN Structure

typedef struct MMC_GETCYCLESCOUNTER_IN{
unsigned char ucDummy;
} MMC_GETCYCLESCOUNTER_IN;

Parameters

ucdummy

Any dummy value.

MMC_GETCYCLESCOUNTER_OUT Structure

typedef struct MMC_GETCYCLESCOUNTER_OUT{
unsigned long ulCyclesCounter;
unsigned short usStatus;
short sErrorID;
} MMC_GETCYCLESCOUNTER_OUT;

Parameters

ulCyclesCounter

Value of the cycles counter. Any +ve value acceptable.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-42 describes the function block for MMC_GetCyclesCounter

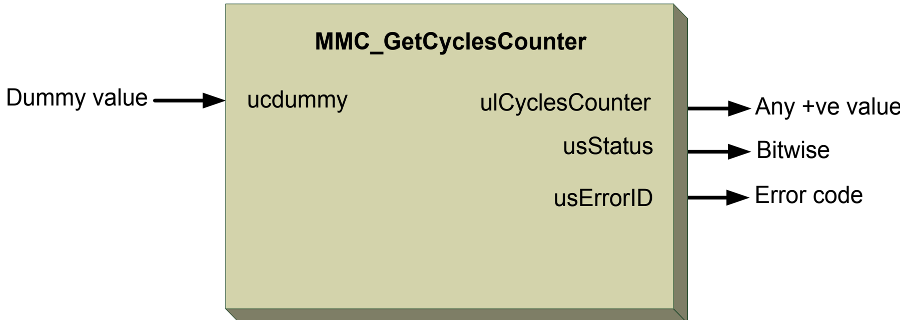


Figure 8-42: MMC_GetCyclesCounter function block

8.1.42 MMC_WriteGroupOfParameters

This function writes a group of parameters to the Maestro.

```
MMC_LIB_API int MMC_WriteGroupOfParameters(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEGROUPOFPARAMETERS_IN* pInParam,
OUT MMC_WRITEGROUPOFPARAMETERS_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Supported

Source GMAS\includes\MMC_general_API.h

GMAS Programming (IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEGROUPOFPARAMETERS_IN** input data structure using the MMC_WriteGroupOfParameters function.

pOutParam

Points to the **MMC_WRITEGROUPOFPARAMETERS_OUT** output structure receiving information as a result of calling the MMC_WriteGroupOfParameters function.

Remarks

None

Scope

All

MMC_WRITEGROUPOFPARAMETERS_IN Structure

```
typedef struct mmc_writegroupofparameters_in{
MMC_WRITEGROUPOFPARAMETERSMEMBER sParameters[GROUP_OF_PARAMETERS_MAXIMUM_SIZE];
MC_EXECUTION_MODE eExecutionMode;
unsigned char ucNumberOfParameters;
unsigned char ucMode;
unsigned char ucExecute;
} MMC_WRITEGROUPOFPARAMETERS_IN;
```

Parameters

MMC_WRITEGROUPOFPARAMETERSMEMBER sParameters[GROUP_OF_PARAMETERS_MAXIMUM_SIZE]

```
typedef struct mmc_writegroupofparametersmember{
double dbValue;
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterIndex;
unsigned short usAxisRef;
unsigned short usPadding1;
unsigned short usPadding2;
unsigned short usPadding3;
}MMC_WRITEGROUPOFPARAMETERSMEMBER;
```

MMC_PARAMETER_LIST_ENUM eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterIndex

The parameter index, a +ve integer value

usAxisRef

Axis reference. Any +ve integer.

UsPadding1

Alignment padding of data. This parameter is not in use at this time.

UsPadding2

Alignment padding of data. This parameter is not in use at this time.

usPadding3

Alignment padding of data. This parameter is not in use at this time.

sParameters[GROUP_OF_PARAMETERS_MAXIMUM_SIZE]

The parameters of the group to be read within the array.

Array with the maximum value of GROUP_OF_PARAMETERS_MAXIMUM_SIZE is 5

MC_EXECUTION_MODE eExecutionMode

Execution mode enumerator defining whether the execution is immediate or queued, with the following values:

eMMC_EXECUTION_MODE_IMMEDIATE = 0,

eMMC_EXECUTION_MODE_QUEUED

ucNumberOfParameters

The number of parameters in the array between 1 to 5, with 5 as a maximum number.

ucMode

Parameter not in use at this time

ucExecute

Parameter not in use at this time

MMC_WRITEGROUPOFPARAMETERS_OUT Structure

```
typedef struct mmc_writegroupofparameters_out{
unsigned int uiHndl;
unsigned short usStatus;
short usErrorID;
unsigned char ucProblematicEntry;
} MMC_WRITEGROUPOFPARAMETERS_OUT;
```

Parameters

uiHndl

Returned function block handle. Any +ve value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

usErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

ucProblematicEntry

The parameter entry which is faulty. +ve char value.

Figure 8-44 describes the function for MMC_WriteGroupOfParameters as applied within the IEC 61131 programming.

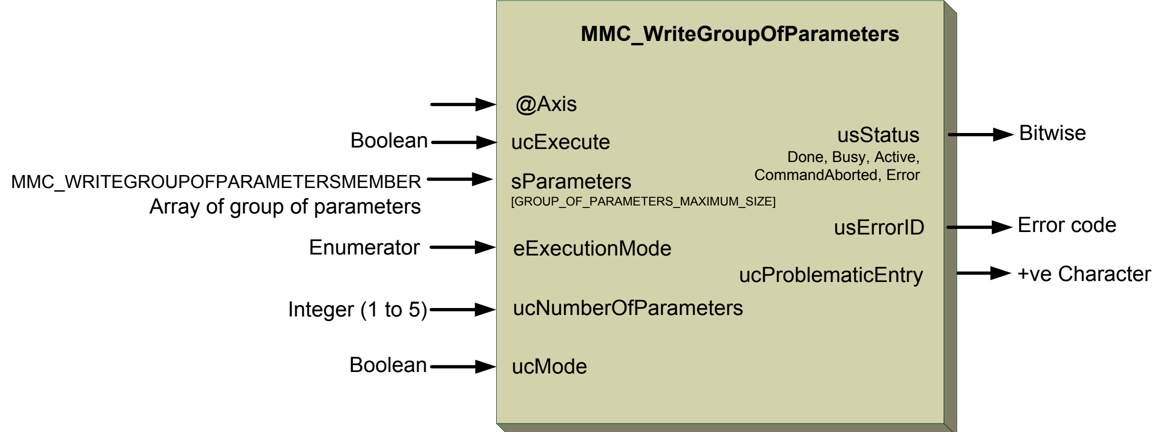


Figure 8-43: MMC_WriteGroupOfParameters function

8.1.43 MMC_WriteGroupOfParametersEx

This function writes a group of regular and PI function parameters to the Maestro.

```
MMC_LIB_API int MMC_WriteGroupOfParametersEx(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEGROUPOFPARAMETERSEx_IN* pInParam,
OUT MMC_WRITEGROUPOFPARAMETERSEx_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Supported

Source GMAS\includes\MMC_general_API.h

GMAS Programming (IEC 61331 Program)\ElmoGenAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEGROUPOFPARAMETERSEX_IN** input data structure using the MMC_WriteGroupOfParametersEx function.

pOutParam

Points to the **MMC_WRITEGROUPOFPARAMETERS_OUT** output structure receiving information as a result of calling the MMC_WriteGroupOfParametersEx function.

Remarks

When executing a PI operation on BOOL variables, it is necessary to extract the relevant bit. For example, on Wait Until Condition EX writing value 1 to varoffset 3

```
InWaitCondition.dbReferenceValue = 1 << 3;
InWaitCondition.eOperationType = MC_CONDITIONFB_OP_EQUAL;
InWaitCondition.iParameterID = 3;
InWaitCondition.ucDirection = ePI_OUTPUT;
```

For bulkread when reading BOOL PI variables according to varoffset : value & (1 << n)), set the value – return data from bulkread and n – varoffset according to the BOOL outputs.

Scope

Relevant for both PI function and regular parameters. However, as a PI write group of parameters interface, it is only relevant for single axis.

MMC_WRITEGROUPOFPARAMETERSEX_IN Structure

```
typedef struct mmc_writegroupofparametersex_in{
MMC_WRITEGROUPOFPARAMETERSMEMBEREX sParameters[GROUP_OF_PARAMETERS_MAXIMUM_SIZE];
MC_EXECUTION_MODE eExecutionMode;
unsigned char ucNumberOfParameters;
unsigned char ucMode;
unsigned char ucExecute;
} MMC_WRITEGROUPOFPARAMETERSEX_IN;
```

Parameters

MMC_WRITEGROUPOFPARAMETERSMEMBEREX sParameters[GROUP_OF_PARAMETERS_MAXIMUM_SIZE]

```
typedef struct mmc_writegroupofparametersmemberex{
double dbValue;
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterIndex;
unsigned short usAxisRef;
unsigned short usPIVarOffset;
unsigned char ucPiDirection;
unsigned char pPadding[3];
}MMC_WRITEGROUPOFPARAMETERSMEMBEREX;
```

dbValue

Value of the parameter requested within the group.

MMC_PARAMETER_LIST_ENUM eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterIndex

The parameter index, a +ve integer value

usAxisRef

Axis reference. Any +ve integer.

ucPiDirection

When the multiaxes reads the parameters list, the type of parameter is checked and handled accordingly. Appropriate for both PI function parameters and regular parameters. . Specify ePI_NONE for regular parameters, ePI_OUPUT for PI variables.

usPIVarOffset

Used to indicate the PI varaiable index when the eDIR is ePI_OUTPUT.

pPadding3

Alignment padding of data. This parameter is not in use at this time.

sParameters[GROUP_OF_PARAMETERS_MAXIMUM_SIZE]

The parameters of the group to be read within the array.

Array with the maximum value of GROUP_OF_PARAMETERS_MAXIMUM_SIZE is 5

eExecutionMode

Execution mode enumerator defining whether the execution is immediate or queued, with the following values:

eMMC_EXECUTION_MODE_IMMEDIATE = 0,

eMMC_EXECUTION_MODE_QUEUED

ucNumberOfParameters

The number of parameters in the array between 1 to 5, with 5 as a maximum number.

ucMode

Parameter not in use at this time

ucExecute

Parameter not in use at this time

MMC_WRITEGROUPOFPARAMETERSEX_OUT Structure

```
typedef struct mmc_writegroupofparametersex_out{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
unsigned char ucProblematicEntry;
} MMC_WRITEGROUPOFPARAMETERSEX_OUT;
```

Parameters

uiHndl

Returned function block handle. Any +ve value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

ucProblematicEntry

The parameter entry which is faulty. +ve char value.

Figure 8-44 describes the function for MMC_WriteGroupOfParametersEx as applied within the IEC 61131 programming.

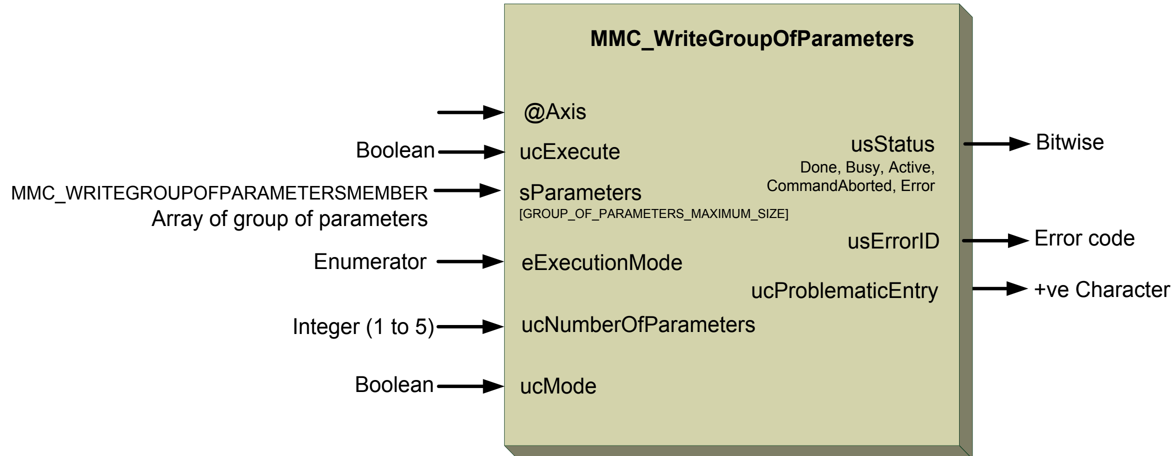


Figure 8-44: MMC_WriteGroupOfParametersEx function

8.1.43.2 Function Block Code Example

```
MMC_WRITEGROUPOFPARAMETERSEX_IN WriteIn;
MMC_WRITEGROUPOFPARAMETERSEX_OUT WriteOut;

WriteIn.eExecutionMode = eMMC_EXECUTION_MODE_QUEUED;
WriteIn.ucExecute = 1;
WriteIn.ucNumberOfParameters = 2;

                       // regular parameter
WriteIn.sParameters[0].eParameterNumber = MMC_DIGITAL_INPUT_LOGIC;
WriteIn.sParameters[0].dbValue = 3;
WriteIn.sParameters[0].ucPiDirection = ePI_NONE;
WriteIn.sParameters[0].usAxisRef = hAxisRef[5];

                       // PI variable
WriteIn.sParameters[1].dbValue = 333;
WriteIn.sParameters[1].ucPiDirection = ePI_OUTPUT;
WriteIn.sParameters[1].usAxisRef = hAxisRef[5];
WriteIn.sParameters[1].usPIVarOffset = 4;

rc = MMC_WriteGroupOfParametersEX(ConnHndl,hAxisRef[4],&WriteIn,&WriteOut);
if(rc)
{
      printf("MMC_WriteGroupOfParametersEX error %d\n",WriteOut.usErrorID);
      return -1;
}
```

8.1.44 MMC_ReadGroupOfParameters

This function retreive a group of parameters to the user.

```
MMC_LIB_API int MMC_ReadGroupOfParameters(
IN MMC_CONNECT_HNDL hConn,
IN MMC_READGROUPOFPARAMETERS_IN* pInParam,
OUT MMC_READGROUPOFPARAMETERS_OUT* pOutParam
);
```

Motion Mode	NC – Supported	Distributed – Supported
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READGROUPOFPARAMETERS_IN** input data structure using the MMC_ReadGroupOfParameters function.

pOutParam

Points to the **MMC_READGROUPOFPARAMETERS_OUT** output structure receiving information as a result of calling the MMC_ReadGroupOfParameters function.

Remarks

None

Scope

All

MMC_READGROUPOFPARAMETERS_IN Structure

```
typedef struct mmc_readgroupofparameters_in{
MMC_READGROUPOFPARAMETERSMEMBER sParameters[GROUP_OF_PARAMETERS_MAXIMUM_SIZE];
unsigned char ucNumberOfParameters;
} MMC_READGROUPOFPARAMETERS_IN;
```

Parameters

MMC_READGROUPOFPARAMETERSMEMBER

```
typedef struct mmc_readgroupofparametersmember{
MMC_PARAMETER_LIST_ENUM eParameterNumber;
int iParameterIndex;
unsigned short usAxisRef;
unsigned short usPadding;
}MMC_READGROUPOFPARAMETERSMEMBER;
```

MMC_PARAMETER_LIST_ENUM eParameterNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3.2 Parameters Tables** for the appropriate integer parameter to be used as enumerator.

iParameterIndex

Array parameter index for arrayed parammeters only. Integer values.

usAxisRef

Axis reference. Any +ve bitwise integer.

usPadding

Alignment padding of data. This parameter is not in use at this time.

sParameters[GROUP_OF_PARAMETERS_MAXIMUM_SIZE]

The parameters of the group to be read within the array.

Array with the maximum value of GROUP_OF_PARAMETERS_MAXIMUM_SIZE is 5

ucNumberOfParameters

The total number of parameters in the group to be read. +ve character value.

MMC_READGROUPOFPARAMETERS_OUT Structure

```
typedef struct mmc_readgroupofparameters_out{
double dbValue[GROUP_OF_PARAMETERS_MAXIMUM_SIZE];
unsigned short usStatus;
short sErrorID;
unsigned char ucProblematicEntry;
} MMC_READGROUPOFPARAMETERS_OUT;
```

Parameters

dbValue[GROUP_OF_PARAMETERS_MAXIMUM_SIZE]

Value of the parameter requested within the group.

Array with the maximum value of GROUP_OF_PARAMETERS_MAXIMUM_SIZE is 5

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block.
Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

ucProblematicEntry

The Maestro returns feedback on a problematic parameter, or which is incorrectly set in the Maestro. Unlimited character value allowed.

Figure 8-45 describes the function for MMC_ReadGroupOfParameters

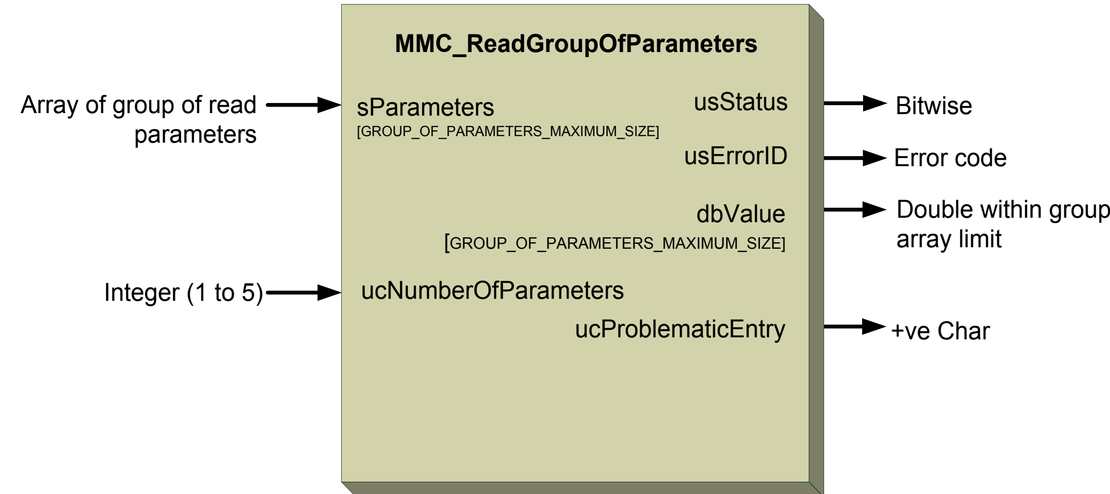


Figure 8-45: MMC_ReadGroupOfParameters function

8.1.45 MMC_WaitUntilConditionFB

The operation of this function block allows synchronization of numerous axes that are not part of a group, to start their motion together. In addition, it allows synchronization of numerous networked Maestro's by starting a motion when a specific bit on a shared IO is raised.

```
MMC_LIB_API int MMC_WaitUntilConditionFB(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WAITUNTILCONDITIONFB_IN* pInParam,
OUT MMC_WAITUNTILCONDITIONFB_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WAITUNTILCONDITIONFB_IN** input data structure using the MMC_WaitUntilConditionFB function.

pOutParam

Points to the **MMC_WAITUNTILCONDITIONFB_OUT** output structure receiving information as a result of calling the MMC_WaitUntilConditionFB function.

Remarks

This function block is inserted to the queue as an administrative function block, and when activated, the Maestro performs a user pre-defined condition test which if results in "TRUE" indicates that the function block has completed its operation, causing the Maestro move to the next function block.

Scope

All

MMC_WAITUNTILCONDITIONFB_IN Structure

```
typedef struct mmc_waituntilconditionfb_in{
double dbReferenceValue;
int iParameterID;
int iParameterIndex;
MC_CONDITIONFB_OPERATION_TYPE eOperationType;
unsigned long ulSpare;
unsigned short usSourceAxisReference;
unsigned char ucExecute;
unsigned char ucPadding;
unsigned char ucSpare[20];
}MMC_WAITUNTILCONDITIONFB_IN;
```

Parameters

dbReferenceValue

Paramater reference value. Constant "double" value, provided by the user

iParameterID

The ID of the parameter. Integer values.

iParameterIndex

Array parameter index for arrayed parammeters only. Integer values.

eOperationType

Operation enumerator parameter from the list of supported logic operation values of the MC_CONDITIONFB_OPERATION_TYPE enumerator according to:

MC_CONDITIONFB_OP_NONE = 0,

MC_CONDITIONFB_OP_EQUAL,

MC_CONDITIONFB_OP_LOWER,

MC_CONDITIONFB_OP_HIGHER,

MC_CONDITIONFB_OP_LOWER_EQUAL,

MC_CONDITIONFB_OP_HIGHER_EQUAL,

MC_CONDITIONFB_OP_MASK_AND,

MC_CONDITIONFB_OP_MASK_LAST

ulSpare

Spare. For internal use only. Any +ve integer value

usSourceAxisReference

The Source Value is a parameter from the list of Axis, Group, Global, Parameters detailed in section **5.3.**

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

ucPadding

Alignment padding of data. Unsigned +ve character values.

ucSpare[20]

Spare. For internal use only. Any +ve character value with a limit of 20 characters

MMC_WAITUNTILCONDITIONFB_OUT Structure

```
typedef struct mmc_waituntilconditionfb_out{
unsigned int uiHndl;
unsigned short usStatus;
unsigned short sErrorID;
}MMC_WAITUNTILCONDITIONFB_OUT;
```

Parameters

uiHndl

Returned function block handle. Any +ve value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-46 describes the function for MMC_WaitUntilConditionFB.

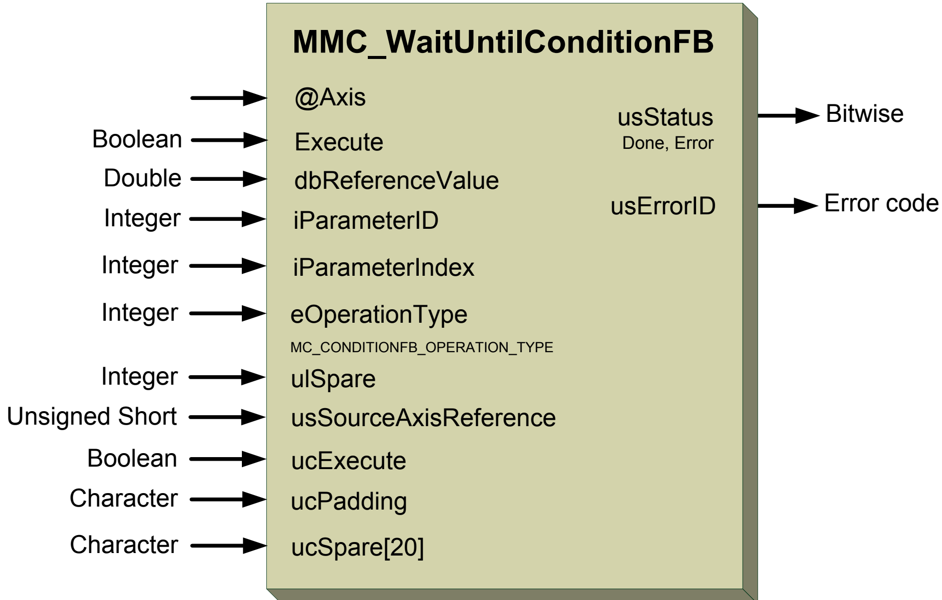


Figure 8-46: MMC_WaitUntilConditionFB function

8.1.46 MMC_WaitUntilConditionFBEx

The operation of this function block applies to both static and PI functions, and allows synchronization of numerous axes that are not part of a group, to start their motion together. In addition, it allows synchronization of numerous networked Maestro's by starting a motion when a specific bit on a shared IO is raised.

```
MMC_LIB_API int MMC_WaitUntilConditionFBEx(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WAITUNTILCONDITIONFBEx_IN* pInParam,
OUT MMC_WAITUNTILCONDITIONFBEx_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WAITUNTILCONDITIONFBEx_IN** input data structure using the MMC_WaitUntilConditionFBEx function.

pOutParam

Points to the **MMC_WAITUNTILCONDITIONFBEx_OUT** output structure receiving information as a result of calling the MMC_WaitUntilConditionFBEx function.

Remarks

This function block is inserted to the queue as an administrative function block, and when activated, the Maestro performs a user pre-defined condition test which if results in "TRUE" indicates that the function block has completed its operation, causing the Maestro move to the next function block.

When executing a PI operation on BOOL variables, it is necessary to extract the relevant bit. For example, on Wait Until Condition EX writing value 1 to varoffset 3

InWaitCondition.dbReferenceValue = 1 << 3;
InWaitCondition.eOperationType = *MC_CONDITIONFB_OP_EQUAL*;
InWaitCondition.iParameterID = 3;

InWaitCondition.ucDirection = *ePI_OUTPUT*;

For bulkread when reading BOOL PI variables according to varoffset : value & (1 << n)), set the value – return data from bulkread and n – varoffset according to the BOOL outputs.

Scope

All

MMC_WAITUNTILCONDITIONFBEx_IN Structure

```
typedef struct mmc_waituntilconditionfbex_in{
double dbReferenceValue;
int iParameterID;
int iParameterIndex;
MC_CONDITIONFB_OPERATION_TYPE eOperationType;
unsigned long ulSpare;
unsigned short usSourceAxisReference;
unsigned char ucExecute;
unsigned char ucDirection;
unsigned char ucSpare[20];
}MMC_WAITUNTILCONDITIONFBEX_IN;
```

Parameters

dbReferenceValue

Paramater reference value. Constant "double" value, provided by the user

iParameterID

The ID of the parameter. Integer values. PI var offset if ucDirection is not ePI_NONE

iParameterIndex

Array parameter index for arrayed parammeters only. Integer values.

eOperationType

Operation enumerator parameter from the list of supported logic operation values of the MC_CONDITIONFB_OPERATION_TYPE enumerator according to:

MC_CONDITIONFB_OP_NONE = 0,

MC_CONDITIONFB_OP_EQUAL,

MC_CONDITIONFB_OP_LOWER,

MC_CONDITIONFB_OP_HIGHER,

MC_CONDITIONFB_OP_LOWER_EQUAL,

MC_CONDITIONFB_OP_HIGHER_EQUAL,

MC_CONDITIONFB_OP_MASK_AND,

MC_CONDITIONFB_OP_MASK_LAST

ulSpare

Spare. For internal use only. Any +ve integer value

usSourceAxisReference

The Source Value is a parameter from the list of Axis, Group, Global, Parameters detailed in section **5.3.**

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0

ePI_OUTPUT = 1

ePI_NONE = 3 for regular parameter

ucSpare[20]

Spare. For internal use only. Any +ve character value with a limit of 20 characters

MMC_WAITUNTILCONDITIONFBEx_OUT Structure

```
typedef struct mmc_waituntilconditionfbex_out{
unsigned int uiHndl;
unsigned short usStatus;
unsigned short sErrorID;
}MMC_WAITUNTILCONDITIONFBEx_OUT;
```

Parameters

uiHndl

Returned function block handle. Any +ve value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-47 describes the function for MMC_WaitUntilConditionFBEx.

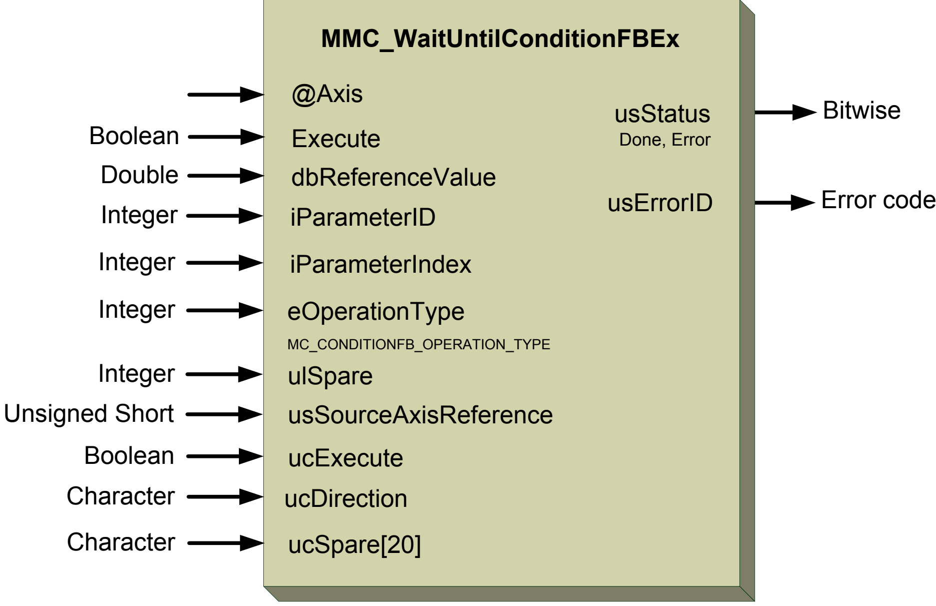


Figure 8-47: MMC_WaitUntilConditionFBEx function

8.1.46.2 Function Block Code Example

```
MMC_WAITUNTILCONDITIONFBEX_IN WaitIn;
MMC_WAITUNTILCONDITIONFBEX_OUT WaitOut;

// digital IO condition: 3rd digital input is changed into '1'
WaitIn.ucDirection = ePI_INPUT;
WaitIn.usSourceAxisReference = hAxisRef[2];
WaitIn.iParameterID = 2;
WaitIn.dbReferenceValue = 0x1 << 2;
WaitIn.eOperationType = MC_CONDITIONFB_OP_EQUAL;
WaitIn.ucExecute = 1;

                        // servo drive FB: wait until condition on the IO is true
rc = MMC_WaitUntilConditionFBEX(ConnHndl,hAxisRef[4],&WaitIn,&WaitOut);
if(rc)
{
      printf("MMC_WaitUntilConditionFBEX error %d\n",WaitOut.usErrorID);
      return -1;
}
```

8.1.47 MMC_WriteMemoryRange

This function writes a memory range for an EtherCAT slave.

MMC_LIB_API int MMC_WriteMemoryRange(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEMEMORYRANGE_IN* pInParam,
OUT MMC_WRITEMEMORYRANGE_OUT.* pOutParam
);

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEMEMORYRANGE_IN** input data structure using the MMC_WriteMemoryRange function.

pOutParam

Points to the **MMC_WRITEMEMORYRANGE_OUT** output structure receiving information as a result of calling the MMC_WriteMemoryRange function.

Remarks

Dependant on the maximum vale of the EtherCAT read and write memory defeined as:
ETHERCAT_MEMORY_READ_MAX_SIZE = 1400 bytes

ETHERCAT_MEMORY_WRITE_MAX_SIZE = 8 bytes

Scope

All

MMC_WRITEMEMORYRANGE_IN Structure

```
typedef struct mmc_writememoryrange_in{
unsigned short usRegAddr;
unsigned char ucLength;
unsigned char pData [ETHERCAT_MEMORY_WRITE_MAX_SIZE];
} MMC_WRITEMEMORYRANGE_IN;
```

Parameters

usRegAddr

Registry address of the EtherCAT memory for the slave.

ucLength

Length of the label string. Length with the precursor format of [Metronome command]##[label]. Any +ve character values.

pData [ETHERCAT_MEMORY_WRITE_MAX_SIZE]

String Data with the precursor format of [Metronome command]##[label]. Any +ve character values with a maximum length of 8 bytes

Dependant on the value of the ETHERCAT_MEMORY_WRITE_MAX_SIZE with a maximum length of 8 bytes

MMC_WRITEMEMORYRANGE_OUT Structure

```
typedef struct mmc_writememoryrange_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEMEMORYRANGE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-48 describes the function for MMC_WriteMemoryRange.

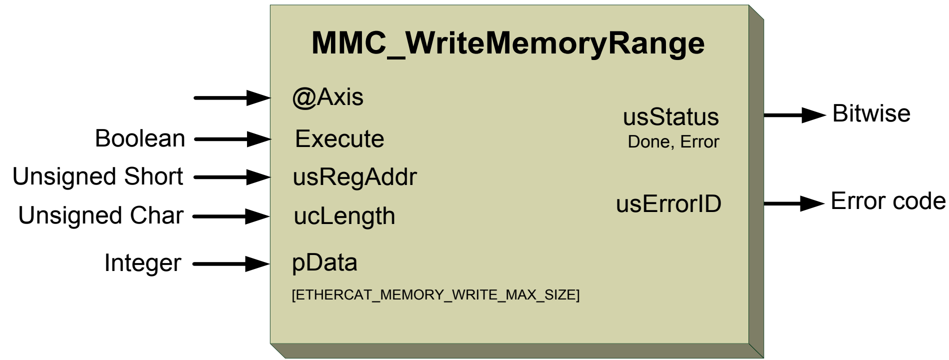


Figure 8-48: MMC_WriteMemoryRange function

8.1.48 MMC_ReadMemoryRange

This function reads a memory range from an EtherCAT slave.

MMC_LIB_API int MMC_ReadMemoryRange(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READMEMORYRANGE_IN* pInParam,
OUT MMC_READMEMORYRANGE_OUT* pOutParam
);

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READMEMORYRANGE_IN** input data structure using the MMC_ReadMemoryRange function.

pOutParam

Points to the **MMC_READMEMORYRANGE_OUT** output structure receiving information as a result of calling the MMC_ReadMemoryRange function.

Remarks

Dependant on the maximum vale of the EtherCAT read and write memory defeined as:
ETHERCAT_MEMORY_READ_MAX_SIZE = 1400 bytes

ETHERCAT_MEMORY_WRITE_MAX_SIZE = 8 bytes

Scope

All

MMC_READMEMORYRANGE_IN Structure

```
typedef struct mmc_readmemoryrange_in{
unsigned short usRegAddr;
unsigned char ucLength;
} MMC_READMEMORYRANGE_IN;
```

Parameters

usRegAddr

Registry address of the EtherCAT memory for the slave.

ucLength

Length of the label string. Length with the precursor format of [Metronome command]##[label]. Any +ve character values.

MMC_READMEMORYRANGE_OUT Structure

```
typedef struct mmc_readmemoryrange_out{
unsigned short usStatus;
short sErrorID;
unsigned char pData[ETHERCAT_MEMORY_READ_MAX_SIZE];
} MMC_READMEMORYRANGE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

pData [ETHERCAT_MEMORY_READ_MAX_SIZE]

String Data with the precursor format of [Metronome command]##[label]. Any +ve character values with a maximum length of 1400 bytes

Dependant on the value of the ETHERCAT_MEMORY_READ_MAX_SIZE with a maximum length of 1400 bytes

Figure 8-49 describes the function for MMC_ReadMemoryRange.

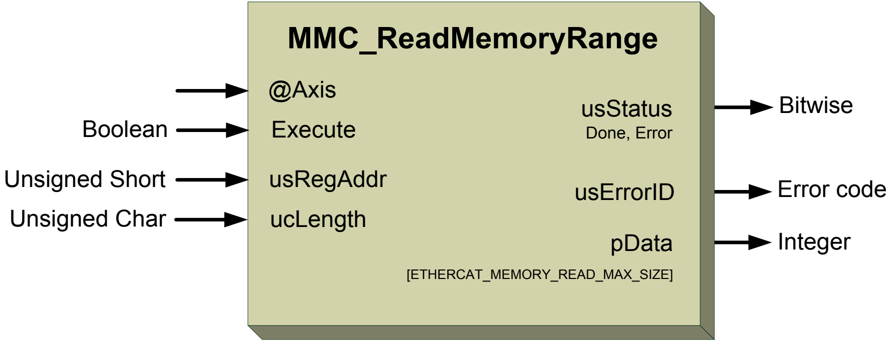


Figure 8-49: MMC_ReadMemoryRange function

8.1.49 MMC_SetDefaultResources

This function restores the Maestro resource file to its **factory default** according to the desired communication type; eCOMM_TYPE_ETHERCAT or eCOMM_TYPE_CAN

```
MMC_LIB_API int MMC_SetDefaultResources(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SETDEFAULTRESOURCES_IN* pInParam,
OUT MMC_SETDEFAULTRESOURCES_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SETDEFAULTRESOURCES_IN** input data structure using the MMC_SetDefaultResources function.

pOutParam

Points to the **MMC_SETDEFAULTRESOURCES_OUT** output structure receiving information as a result of calling the MMC_SetDefaultResources function.

Remarks

Scope

All

MMC_SETDEFAULTRESOURCES_IN Structure

```
typedef struct mmc_setdefaultresources_in{
unsigned char ucConnectionType;
} MMC_SETDEFAULTRESOURCES_IN;
```

Parameters

ucConnectionType

Connection type defined by the enumerator COMM_TYPES with the following values:

eCOMM_TYPE_ETHERCAT = 1

eCOMM_TYPE_CAN = 2

MMC_SETDEFAULTRESOURCES_OUT Structure

```
typedef struct mmc_setdefaultresources_out{
unsigned short usStatus;
short sErrorID;
} MMC_SETDEFAULTRESOURCES_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-50 describes the function for MMC_SetDefaultResources.

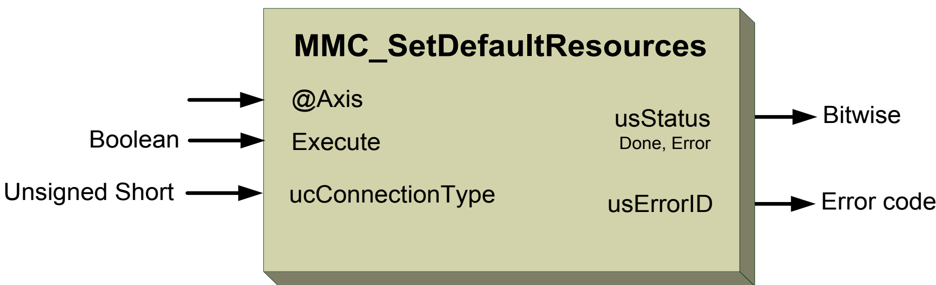


Figure 8-50: MMC_SetDefaultResources function

8.1.50 MMC_KillRepetitive

This function stops the repetitive motion after the current function block.

```
MMC_LIB_API int MMC_KillRepetitive
(IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_KILLREPETITIVE_IN* pInParam,
OUT MMC_KILLREPETITIVE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_KILLREPETITIVE_IN** input data structure using the MMC_KillRepetitive function.

pOutParam

Points to the **MMC_KILLREPETITIVE_OUT** output structure receiving information as a result of calling the MMC_KillRepetitive function.

Remarks

Scope

All

MMC_KILLREPETITIVE_IN Structure

typedef struct mmc_killrepetitive_in{
unsigned int uiDummy;
}MMC_KILLREPETITIVE_IN;

Parameters

uiDummy

Dummy values. Any +ve integer values acceptable.

MMC_KILLREPETITIVE_OUT Structure

typedef struct mmc_killrepetitive_out{
unsigned short usStatus;
unsigned short sErrorID;
}MMC_KILLREPETITIVE_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-51 describes the function for MMC_KillRepetitive.

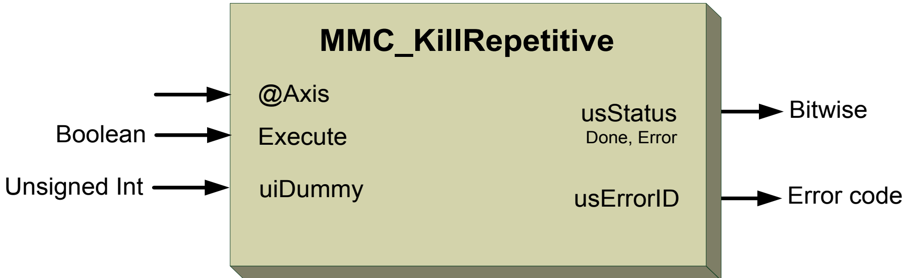


Figure 8-51: MMC_KillRepetitive function

8.1.51 MMC_UserCommandControl

This function executes a user command (user program or execute LINUX command).

```
MMC_LIB_API int MMC_UserCommandControl(
IN MMC_CONNECT_HNDL hConn,
IN MMC_USRCOMMAND_IN* pInParam,
OUT MMC_USRCOMMAND_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_USRCOMMAND_IN** input data structure using the MMC_UserCommandControl function.

pOutParam

Points to the **MMC_USRCOMMAND_OUT** output structure receiving information as a result of calling the MMC_UserCommandControl function.

Remarks

Scope

All

MMC_USRCOMMAND_IN Structure

```
typedef struct {
MC_COMMAND_OPERATION eUsrCommandOp;
char cUserCommand[256];
unsigned char ucSpare[20];
} MMC_USRCOMMAND_IN;
```

Parameters

MC_COMMAND_OPERATION eUsrCommandOp

The user command operation enumerator eUsrCommandOp describes the operations available to execute a user command, where MC_COMMAND_OPERATION has the following enumertor values:

eMMC_COMMAND_OPERATION_START =0,

eMMC_COMMAND_OPERATION_STOP =1,

eMMC_COMMAND_OPERATION_ISRUNNING =2,

eMMC_COMMAND_OPERATION_ISEXIST =3,

eMMC_COMMAND_OPERATION_REMOVE =4,

eMMC_COMMAND_OPERATION_ISEXECUTABLERUNNING =5,

cUserCommand[256]

Description of the user command. Char with a maximum of 256 chars.

ucSpare[20]

Spare description for future use. +ve char value with maximum of 20 chars.

MMC_USRCOMMAND_OUT Structure

```
typedef struct {
unsigned short usStatus;
short usErrorID;
unsigned char ucIsRunning;
unsigned char ucIsExist;
char cExecutableFileName[64];
char cSpear[448];
} MMC_USRCOMMAND_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

ucIsRunning

Is command running? Boolean but set as an unsigned character

ucIsExist

Does the program exist? Boolean but set as an unsigned character

cExecutableFileName[64]

The name of the executible file, with a maximum of 64 characters

cSpear[448]

Spare description for future use. Char value with maximum of 448 chars.

Figure 8-52 describes the function for MMC_UserCommandControl.

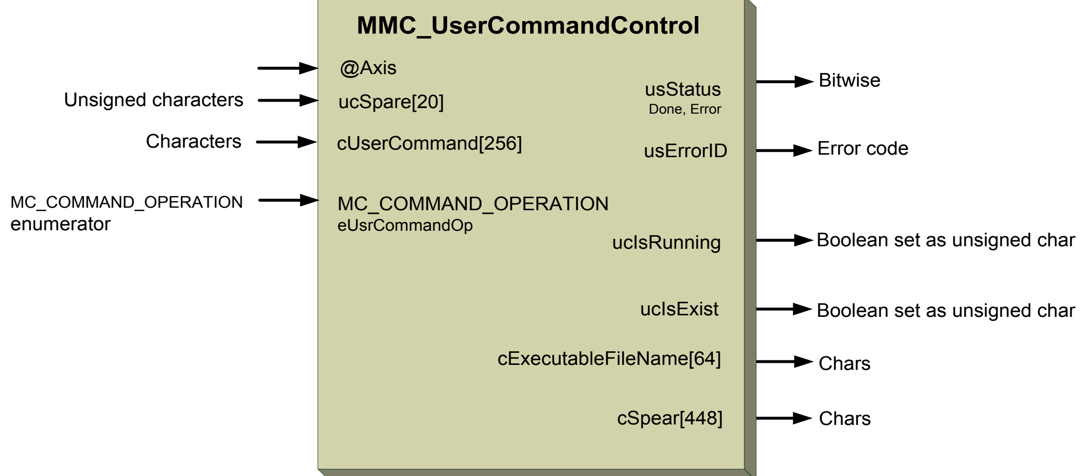


Figure 8-52: MMC_UserCommandControl function

8.1.52 MMC_SetAllFbExeModeImm

This function sets all function blocks to immediate execution mode

```
MMC_LIB_API int MMC_SetAllFbExeModeImm(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SETALLFBEXEMODETOIMM_IN* pInParam,
OUT MMC_SETALLFBEXEMODETOIMM_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SETALLFBEXEMODETOIMM_IN** input data structure using the MMC_SetAllFbExeModeImm function.

pOutParam

Points to the **MMC_SETALLFBEXEMODETOIMM_OUT** output structure receiving information as a result of calling the MMC_SetAllFbExeModeImm function.

Remarks

Scope

All

MMC_SETALLFBEXEMODETOIMM_IN Structure

```
typedef struct {
unsigned short usAxisRef;
} MMC_SETALLFBEXEMODETOIMM_IN;
```

Parameters

usAxisRef

The axis reference on which all the present function blocks will be excuted immediately.

+ve short value allowed.

MMC_SETALLFBEXEMODETOIMM_OUT Structure

```
typedef struct {
unsigned short usStatus;
short usErrorID;
} MMC_SETALLFBEXEMODETOIMM_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 8-53 describes the function for MMC_SetAllFbExeModeImm.

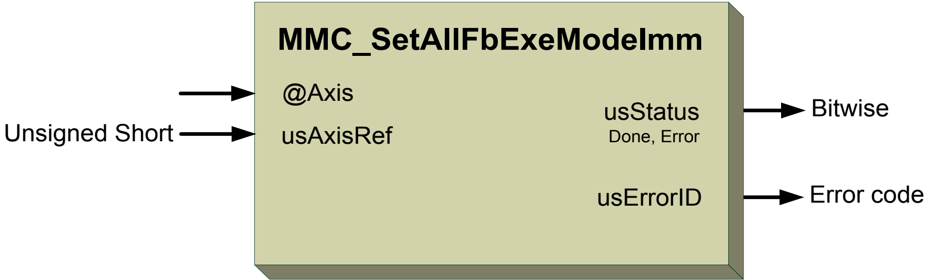


Figure 8-53: MMC_SetAllFbExeModeImm function

8.1.53 MMC_GetVerPath

Reads the Maestro firmware flash version path from the Maestro itself. This is set during a TFT firmware download from the host server.

When it becomes necessary to use this function block, please contact Elmo for further information.

8.1.54 MMC_DownloadVersion

Performs a download of new firmware version to the Maestro. When it becomes necessary to use this function block, please contact Elmo for further information.

8.1.55 MMC_ReadDownloadVersionStatus

Returns the status of the last downloaded version.

When it becomes necessary to use this function block, please contact Elmo for further information.

8.1.56 MMC_SetVerPath

As part of the download firmware functions, sets the Maestro firmware flash version path from the Maestro itself. This is set during a TFT firmware download from the host server.

When it becomes necessary to use this function block, please contact Elmo for further information.

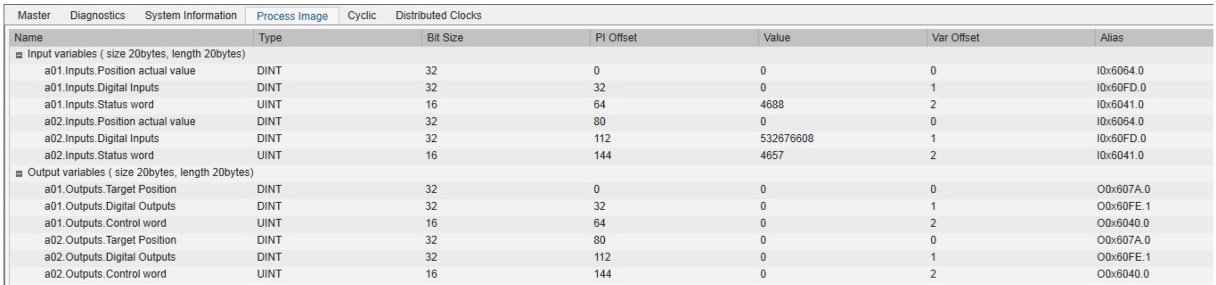
Chapter 9: Process Image(PI)

9.1 Introduction

Process Image is the cyclic data transferred between the EtherCAT Master to the Slave. The PI mechanism allows the user to read and write this cyclic data. The content of the cyclic data is constructed in the EtherCAT configuration tool – the EASII Configurator. The configuration is created based on the EtherCAT Slave Information (ESI) file, where the user selects the content of the process data in the FMMU tab within the Configurator. This provides the the Maestro with all-inclusive information to match the correct PDO group to the active inputs/outputs of the Master's process data image.

9.2 Variable Types

The EtherCAT configuration procedure in the EASII application provides considerable information on the Process Image.



Master | Diagnostics | System Information | Process Image | Cyclic | Distributed Clocks

Name	Type	Bit Size	PI Offset	Value	Var Offset	Alias
Input variables (size 20bytes, length 20bytes)						
a01.Inputs.Position actual value	DINT	32	0	0	0	I0x6064.0
a01.Inputs.Digital Inputs	DINT	32	32	0	1	I0x60FD.0
a01.Inputs.Status word	UINT	16	64	4688	2	I0x6041.0
a02.Inputs.Position actual value	DINT	32	80	0	0	I0x6064.0
a02.Inputs.Digital Inputs	DINT	32	112	532676608	1	I0x60FD.0
a02.Inputs.Status word	UINT	16	144	4657	2	I0x6041.0
Output variables (size 20bytes, length 20bytes)						
a01.Outputs.Target Position	DINT	32	0	0	0	O0x607A.0
a01.Outputs.Digital Outputs	DINT	32	32	0	1	O0x60FE.1
a01.Outputs.Control word	UINT	16	64	0	2	O0x6040.0
a02.Outputs.Target Position	DINT	32	80	0	0	O0x607A.0
a02.Outputs.Digital Outputs	DINT	32	112	0	1	O0x60FE.1
a02.Outputs.Control word	UINT	16	144	0	2	O0x6040.0

The following table describes the above Process Image tab in the EtherCAT configuration.

Input and Output Variables	
Name	Name of the input/output variable in the attached Slave devices, ranging from Position (actual value) Input, Target Position Output, Digital IOs, Status word input, and Control word ouput
Type	Data types of the input variable in attached Slave devices. Variables are either DINT (32 bit Double Integral), UINT (16 bit unsigned integral), or INT (8 bit Integral).
Bit Size	Bits occupied by the input variable in attached Slave devices
PI Offset	Location of the Slave's input variable in the memory
Value	Shows physical property value
Var Offset	Variable offset parameter for Maestro; read only value but recalculates itself when a different input is selected. For each drive input/ouput, an order 0, 1, 2 is created to maintain the orderly position of each data
Alias	The alias or a different name for the input/output variable

EtherCAT variable types are converted to the Maestro Variable type named VarType.Each variable section has the following attributes:

Variable	Attributes
CANopenIndex	The index of the PI object as appears in the EASII configurator FMMU tab in decimal format.
CANopenSubIndex	The sub index of the PI object in decimal format.
VarType	Field will indicate the type of the variable according to the following enumerator: ePI_BOOL = 0 ePI_SIGNED_CHAR = 1 ePI_UNSIGNED_CHAR = 2 ePI_SIGNED_SHORT = 3 ePI_UNSIGNED_SHORT = 4 ePI_SIGNED_INT = 5 ePI_UNSIGNED_INT = 6 ePI_SIGNED_LONG_LONG = 7 ePI_UNSIGNED_LONG_LONG = 8 ePI_FLOAT = 9 ePI_DOUBLE = 10 ePI_BITWISE = 11 ePI_8MULTIPLE = 12 ePI_INVALID The variable type will be selected according to the relevant EtherCAT type. For example: Sync Manager configurator MBoxOut (0x1800 , 140 , 0x26000100) MBoxIn (0x1900 , 140 , 0x22000100) Outputs (0x1100 , 12 , 0x64000100) Inputs (0x1180 , 14 , 0x60000100) 0x1A00 Inputs Position actual value (0x6064.0 , DINT , 32) Digital Inputs (0x60FD.0 , DINT , 32) Status word (0x6041.0 , UINT , 16) "position actual value" (0x6064) and "Target position" (0x607A) are defined in the slave object dictionary as DINT which is signed double (32 bits) integer. Therefore, both of the objects need to receive data type of ePI_SIGNED_INT. "controlword" (0x6040) and statusword (0x6041) are defined as UINT which is unsigned double integer. Therefore they need to receive the data type of ePI_UNSIGNED_INT. ePI_BITWISE is the matching type for every type which its bit size is not a whole multiple of 8, for example BIT4, BIT5 etc. ePI_8MULTIPLE is the matching type for every type which is not of the standard

Variable	Attributes
	types (int short, char…) and its bit size is a whole ,multiple of 8 for example INT24 or UINT48.
Alias	Field holds a string of user selected string for this variable. The aliasing field length is 34 bytes to store the variable aliasing name.
BitSize and BitOffs	Similar to the same field in the <ProcessImage> section.
ID	The ID attribute is the relevant slave ID.

9.3 PI User Functions

With respect to the end-user, there are 26 available library interfaces; Writing and Reading PI variables:

- READ/WRITE PI variable which supports in READ/WRITE operation.
- LARGEREAD/WRITE PI variable RAW.

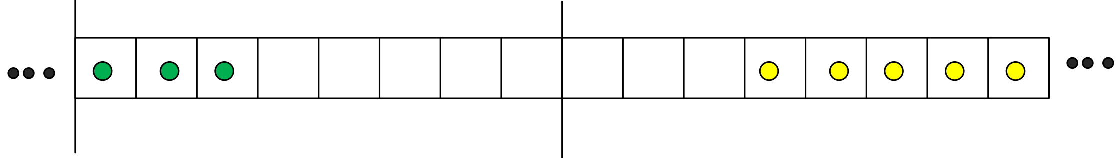
The unique key indicator for referring to specific PI variable depends on the position of this PI variable in the current PI configuration - i.e. first/second/third…variable. The PI variables are stored in the PI table according to their position order in the PI configuration (Var Offset), for fast access of the PI variable using the user READ/WRITE functions.

The READ\WRITE PI variable with bit size < 8 is divided into two methods:

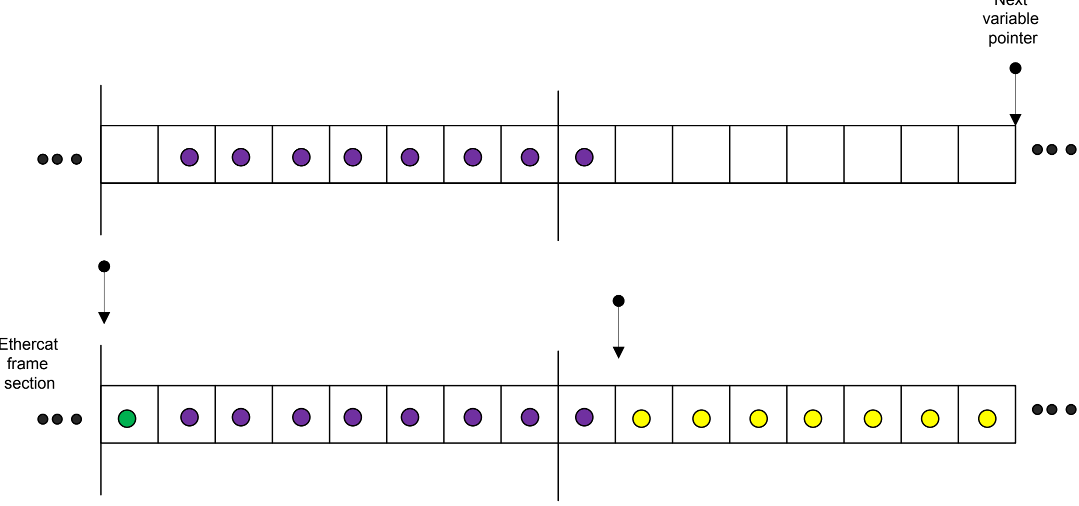
- READ\WRITE RAW a variable of bit size 2-N (not a whole multiple of 8) to represent the bit offset, e.g. when the variable ucBitOffset field varies from 0, then the user must consider that the first ucOffset bits are disregarded and masked.
- READ\WRITE BOOL a variable of bit size 1 (BOOL) is used by writing the value (0 or 1) to the input union, no offset considerations required.

9.4 Read/Write RAW Data

The RAW data variables is designed so that every variable has its own quantity of bits, not a multiple of eight. The data should be read/writtern to the Maestro with an offset as in the EtherCAT frame. The first and last offset bits are redundant and ignored.



In the above example the green variable is of bit size 3 and located in the frame in (bit offset % 8) = 0, the yellow variable is of bit size 5 and located in (bit offset % 8) = 3 in the frame.



However, in the above example the variable is spanned over two bytes and allows the next variable pointer to point only to the next byte. When copied to and from the EtherCAT frame, the data is transferred by the Maestro as-is. The first and last bits are masked, and only the relevant data (purple) is transferred.

READ/WRITE PI variables of the RAW type are divided into two type of variables, Large RAW and Regular RAW PI varaibles. The read/write operation with Regular RAW variables is performed using the functions MMC_ReadPIVarRaw and MMC_WritePIVarRaw, whereas the Large RAW variables are read/written using the functions MMC_ReadLargePIVarRaw and MMC_WriteLargePIVarRaw.

9.5 Recorder PI Mechanism Support

Recordable signals are predefined signals listed in the Personality file. These are all signals which are present in the Maestro static memory. In order to add a signal, new firmware is required.

PI variables are allocated dynamically upon Maestro startup according to the master xml. The Recorder PI mechanism enables the user to record every PI variable according to its direction and variable offset, and provides two interfaces for recording existing variables as a target position, i.e. according to their index in the Personality file and according to their PI variable offset.

9.5.1 Differentiation Between Regular Signal and PI Signal

Since the method of recording PI variables is different, the recorder differentiates between regular and PI signals. Refer to section **10.3 Using Data Recording in the Maestro** in the Maestro for details of the Recording regular variables. These variables are adapted with an added sub-function to define the PI variable type as follows:

```
typedef enum pi_directions
{
    ePI_INPUT      = 0,
    ePI_OUTPUT     = 1,
    ePI_NONE       = 3,
} PI_DIRECTIONS;
```

Where;

- Regular signals have *ePI_NONE* in the relevant cell.
- In order to record a PI input the relevant cell will be set to *ePI_INPUT*.
- For recording output the relevant cell will be set to *ePI_OUTPUT*.

When the two upper bits of the LSBs (least significant byte) of uiRv[i] (or uiRp[1]) is set as value different than ePI_NONE, the recorder considers the lower two bytes of uiRv or uiRp[1] (trigger variable number) as PI var offset (not including the direction bits) according to the given direction.

For example:

uiRv[i] = 0x [00][01][40][03] > PI Output number 3 of axis with axis ref 1

uiRv[i] = 0x [00][01][00][03] > PI Input number 3 of axis with axis ref 1

uiRv[i] = 0x [00][01][C0][03] > signal 3 of axis with axis ref 1

9.5.2 Determining the Recorded Signals Length

The remaining recorded signal input variables (e.g. parameters, signals bit mask, recorder gap and length) are determined as regular signals. The PI variables size can be acquired from the *MMC_GetPIVarInfo* function. This size is used for display purposes.

9.5.3 Recording Large PI Variables

Since the recorded data is uploaded in 4 byte sections, when recording a large PI variable in EASII, it is necessary to also receive the data in consecutive 4 byte sections.

Normally when recording a variable of 64 bits, EASII marks two consecutive signals for recording. Two bits are set in **uiRc** and two consecutive signal numbers are set in **uiRv**. In order to record a large PI variable, a bit per 4 byte section must be set in **uiRC** and the same number of entries in **uiRV** must hold the same value (Axis_Ref << 16 | PI var Offset and direction).

Currently recording of variables larger than 64 bits is not supported. Therefore two consecutive entries maximum should be entered into the variable **uiRV**.

9.5.4 Nonstandard Variable Types

BOOL variables	Recording is supported as used in IO devices. The BOOL variables are stored in the Maestro RAM according to their offset – up to 8 consecutive BOOL variables are stored in 1 byte. When the user reads a BOOL PI variable the Read function masks the redundant bits and performs the correct shifting so the user receives an Unsigned Char with the value of 0 or 1. In order to spare this processing in RT a whole byte is transferred to the EASII and the EASII performs the masking and shifting according to the bit offset field of the variables for display purposes.
Bitwise and 8_multiple variables < 64 bits	8_multiple variables are variables of the size 3,5,7 bytes. Bitwise variables are variables of bit size which is not a whole multiply of 8 – 2..7 bits, 9..15 bits, etc. Recording of these variables is not supported.

9.5.5 MMC_BeginRecordingEx

Starts the recording of internal controller variables and PI variables data from the Maestro server.

```
MMC_LIB_API int MMC_BeginRecordingCmdEx(
IN MMC_CONNECT_HNDL hConn,
IN MMC_BEGIN_RECORDING_IN* pInParam,
OUT MMC_BEGIN_RECORDING_OUT* pOutParam
);
```

Motion Mode	NC – N/A	Distributed – N/A
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_BEGIN_RECORDING_IN** input data structure using the MMC_BeginRecordingCmdEx function.

pOutParam

Points to the **MMC_BEGIN_RECORDING_OUT** output structure receiving information, as a result of calling the MMC_BeginRecordingCmdEx function.

Remarks

None

Scope

All

MMC_BEGIN_RECORDING_IN Structure

```
typedef struct{
unsigned long uiRg;
unsigned long uiRl;
unsigned long uiRc;
unsigned long uiRv[NC_MAX_REC_SIGNALS_NUM];
unsigned long uiRp[NC_MAX_REC_PARAMS_NUM];
}MMC_BEGIN_RECORDING_IN;
```

Parameters

uiRg

Recording Data Gap which specifies the sampling rate of the recorder. Any +ve integer value

uiRl

Recording Data Length with a buffer size (default size 4MB). Any +ve integer value.

uiRc

Recording Data Signals Bit mask according to the definitions described in section **10.4.1 Recording Data Signals Bitmask Definitions**. Defines which of mapped signals should be recorded with up to 32 different synchronized signals. Any +ve integer value.

uiRv

uiRv is the recording Signals ID mapping enumerator which maps the IDs of the recordable signals to logical IDs that the recorder can reference. It is a 32 bit mask assembled from the AxisNumber and the Signal parameter. The upper 16 bits are the axis reference, and the lower 16 bits the signal parameter, e.g. 0x00020015, where the 0002 refers to axis 02, and the 0015 refers to the signal ID 21 (in Hex) with PI direction of ePI_INPUT, i.e. the recorded signal will be PI input variable 21 of axis 2. Refer to the ID definitions described in section 10.4.2 Recording Parameters.

[NC_MAX_REC_SIGNALS_NUM] is an array value of between [1….22] and *uiRv* can have any +ve integer value.

uiRp

uiRp is the ID integer value of the Recording Parameters. Refer to the section **10.4.4 Trigger Modes** for further details. The recorder parameters defines which event will trigger the recorder, and the trigger position, according to the following definitions:

Recording Parameters - Rp[N]	ID	Definition
TG_RECORDING_SPARE	0	Spare
TG_RECORDING_TRIGGER_VALUE	1	Defines which of mapped signals should be recorded. The trigger variable does not need to be one of the recorded variables.
TG_RECORDING_PRE_TRIGGER_LENGTH	2	The percentage of the recorded signal taken before the trigger

		event. (recorder trigger delay*)
TG_RECORDING_TRIGGER_TYPE	3	
TG_RECORDING_TRIGGER_LEVEL_1	4	Level for positive slope trigger, or high side for window trigger.
TG_RECORDING_TRIGGER_LEVEL_2	5	Level for negative slope trigger, or low side for window trigger.
TG_RECORDING_TRIGGER_POLARITY	6	Logic for bit field trigger- 0 positive logic, 1 – negative
TG_RECORDING_TRIGGER_IN_MASK	7	Mask for bit field trigger

[NC_MAX_REC_PARAMS_NUM] is an array value of [1….8].

MMC_BEGIN_RECORDING_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_BEGIN_RECORDING_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-1 describes the function block for MMC_BeginRecordingCmdEx

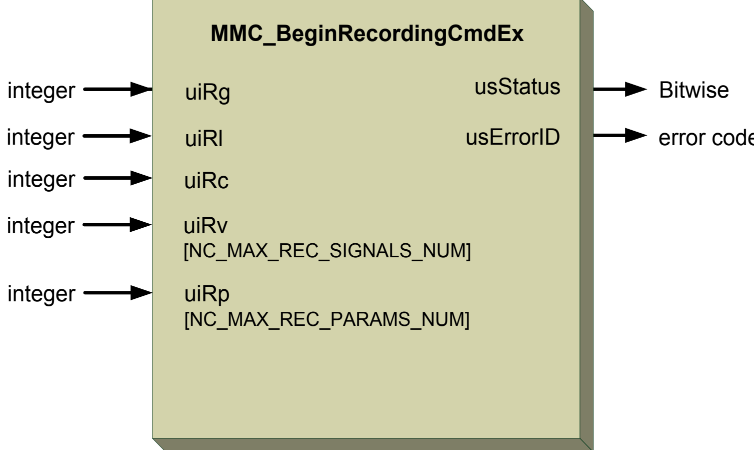


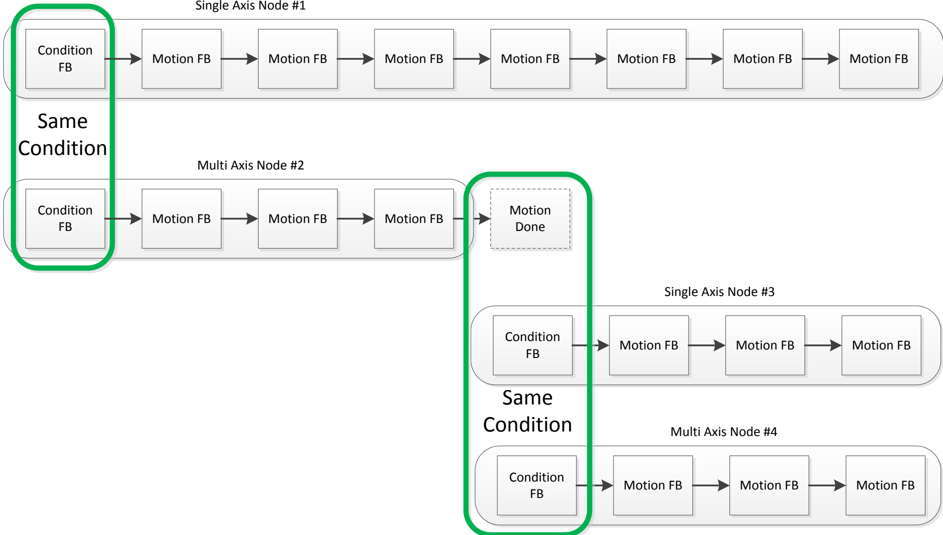
Figure 9-1: MMC_BeginRecordingCmdEx function block

9.5.5.2 Function Block Code Example

```
MMC_BEGIN_RECORDING_IN RecIn;
MMC_BEGIN_RECORDING_OUT RecOut;

memset(&RecIn,0,sizeof(MMC_BEGIN_RECORDING_IN));

// 3 signals
RecIn.uiRc = 0x7;
RecIn.uiRg = 1;
RecIn.uiRl = 5000;

// 3 signals of axis 4, 3rd PI output , 2nd input and 2nd output
RecIn.uiRv[0] = (hAxisRef[4] << 16) | (3 & 0x3FFF) | (ePI_OUTPUT << 14);
RecIn.uiRv[1] = (hAxisRef[4] << 16) | (2 & 0x3FFF) | (ePI_INPUT << 14);
RecIn.uiRv[2] = (hAxisRef[4] << 16) | (2 & 0x3FFF) | (ePI_OUTPUT << 14);

// trigger type rising edge and single
RecIn.uiRp[3] = 0x30001;

// assign trigger level 1
RecIn.uiRp[4] = 15;

                      // trigger on 4th output of axis 4
RecIn.uiRp[1] = (hAxisRef[4] << 16) | (4 & 0x3FFF) | (ePI_OUTPUT << 14);

rc = MMC_BeginRecordingCmdEX(ConnHndl,&RecIn,&RecOut);
if(rc)
{
      printf("begin recording error %d\n",RecOut.usErrorID);
}
```

9.6 Wait Condition Function Block

The current Wait Condition mechanism inserts a function block to the queue. This function block receives the state DONE only when a condition is true, and restrains the function block queue, not allowing other function blocks to execute until it recieves the state DONE.



The current implementation using the function MMC_WaitUntilConditionFB () only allows performing a condition on the available parameters which are static and are not dependent on the configuration. However, to include enabling a condition for a PI variable value, the user should use the more flexible version MMC_WaitUntilConditionFBEx. The user can set a condition not only on a parameter from the parameters list, but also on a PI input or output variable.

9.7 PI Functions

With the employment of PI functions, from Maestro version 1.1.2.0 Build 7 onwards, a number of previously used functions in EtherCAT become obsolete. However, they retain their importance as CANopen functions. The following lists the functions replaced by the PI Mechanism.

MMC_ECATIOReadAnalogInput	MMC_ECATIODisableDIChangedEvent
MMC_ECATIOWriteAnalogOutput	MMC_ReadDS401DInput
MMC_ECATIOWriteDigitalOutput	MMC_ReadDigitalOutputs
MMC_ECATIOReadDigitalInput	MMC_ReadDS401DIGroup
MMC_ECATIOEnableDIChangedEvent	MMC_WriteDS401DOGroup

The following table lists the new PI functions. Inaddition, all Read/Write **IEC** PI functions are generic to any type and the function MMC_WritePIByIndx can be used instead.

PI functions
MMC_ReadPIVarBOOL
MMC_ReadPIVarChar
MMC_ReadPIVarUChar
MMC_ReadPIVarShort
MMC_ReadPIVarUShort
MMC_ReadPIVarInt
MMC_ReadPIVarUInt
MMC_ReadPIVarFloat
MMC_ReadPIVarRaw
MMC_ReadPIVarLongLong
MMC_ReadPIVarULongLong
MMC_ReadPIVarDouble
MMC_ReadLargePIVarRaw
MMC_WritePIVarBool
MMC_WritePIVarChar

PI functions
MMC_WritePIVarUChar
MMC_WritePIVarUShort
MMC_WritePIVarShort
MMC_WritePIVarUInt
MMC_WritePIVarInt
MMC_WritePIVarFloat
MMC_WritePIVarRaw
MMC_WritePIVarULongLong
MMC_WritePIVarLongLong
MMC_WritePIVarDouble
MMC_WriteLargePIVarRaw
MMC_GetPIVarInfo
MMC_GetPIVarInfoByAlias
MMC_GetPIVarsRangeInfo

9.7.1 MMC_ReadPIVarBOOL

This function reads a Processing Image input\output Boolean variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarBOOL(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARBOOL_IN* pInParam,
OUT MMC_READPIVARBOOL_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARBOOL_IN** input data structure using the MMC_ReadPIVarBOOL function.

pOutParam

Points to the **MMC_READPIVARBOOL_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarBOOL function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH	35
PI_REG_VAR_SIZE	4

Scope

All

MMC_READPIVARBOOL_IN Structure

```
typedef struct mmc_readpivarbool_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARBOOL_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARBOOL_OUT Structure

```
typedef struct mmc_readpivarbool_out{
unsigned short usStatus;
short sErrorID;
unsigned char ucBOOL;
} MMC_READPIVARBOOL_OUT;
```

Parameters

ucBOOL

The Boolean parameter requested from the processed image.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-2 describes the function for MMC_ReadPIVarBOOL.

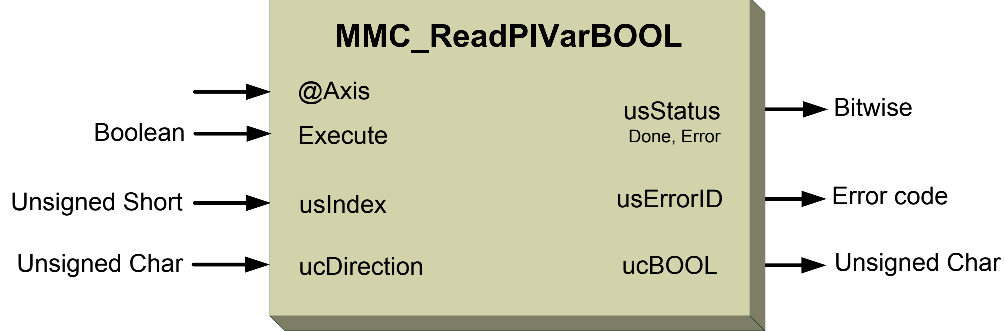


Figure 9-2: MMC_ReadPIVarBOOL function

9.7.1.2 Function Block Code Example

```
    case BOOL_BITSIZE:
    {
        MMC_READPIVARBOOL_IN *pReadIn;
        MMC_READPIVARBOOL_OUT ReadOut;

        pReadIn = (MMC_READPIVARBOOL_IN *)&ReadIn;

        rc = MMC_ReadPIVarBOOL(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
        if(rc)
        {
printf("MMC_ReadPIVarBOOL failed error %d",ReadOut.sErrorID);
break;
        }

        printf("read the BOOL value: %d\n",ReadOut.ucBOOL);
    }
    break;
    case CHAR_BITSIZE:
    {
```

9.7.2 MMC_ReadPIVarChar

This function reads a Processing Image input\output Character variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarChar(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARCHAR_IN* pInParam,
OUT MMC_READPIVARCHAR_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARCHAR_IN** input data structure using the MMC_ReadPIVarChar function.

pOutParam

Points to the **MMC_READPIVARCHAR_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarChar function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

PI_REG_VAR_SIZE 4

Scope

All

MMC_READPIVARCHAR_IN Structure

```
typedef struct mmc_readpivarsignedchar_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARCHAR_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARCHAR_OUT Structure

```
typedef struct mmc_readpivarsignedchar_out{
unsigned short usStatus;
short sErrorID;
signed char cData;
} MMC_READPIVARCHAR_OUT;
```

Parameters

cData

The signed character parameter requested from the processed image.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-3 describes the function for MMC_ReadPIVarChar.

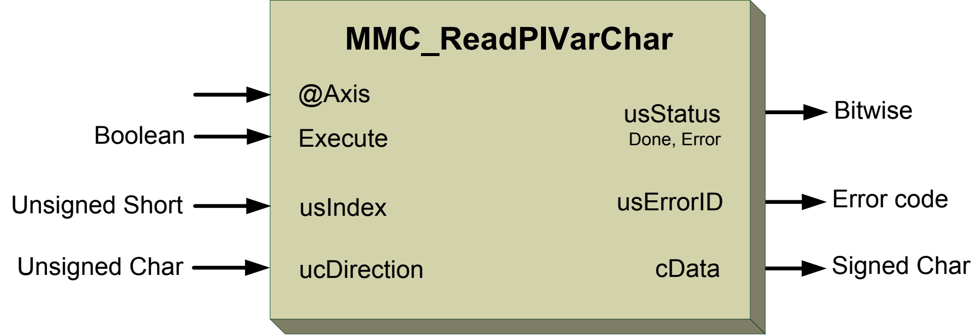


Figure 9-3: MMC_ReadPIVarChar function

9.7.2.2 Function Block Code Example

```
    case CHAR_BITSIZE:
{
    // check signess of the variable (signed\unsigned)
    switch(ucVarType)
    {
    case ePI_SIGNED_CHAR:
        {
            MMC_READPIVARCHAR_IN *pReadIn = (MMC_READPIVARCHAR_IN *)&ReadIn;
            MMC_READPIVARCHAR_OUT ReadOut;

            rc = MMC_ReadPIVarChar(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
            if(rc)
            {
                printf("MMC_ReadPIVarSignedChar failed error %d",ReadOut.sErrorID);
                break;
            }

            printf("read the signed CHAR value: %d\n",ReadOut.cData);
        }
        break;
```

9.7.3 MMC_ReadPIVarUChar

This function reads a Processing Image input\output Unsigned Character variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarUChar(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARUCHAR_IN* pInParam,
OUT MMC_READPIVARUCHAR_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARUCHAR_IN** input data structure using the MMC_ReadPIVarUChar function.

pOutParam

Points to the **MMC_READPIVARUCHAR_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarUChar function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

PI_REG_VAR_SIZE 4

Scope

All

MMC_READPIVARUCHAR_IN Structure

```
typedef struct mmc_readpivarunsignedchar_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARUCHAR_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARUCHAR_OUT Structure

```
typedef struct mmc_readpivarunsignedchar_out{
unsigned short usStatus;
short sErrorID;
unsigned char ucData;
} MMC_READPIVARUCHAR_OUT;
```

Parameters

ucData

The unsigned character parameter requested from the processed image.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-4 describes the function for MMC_ReadPIVarUChar.

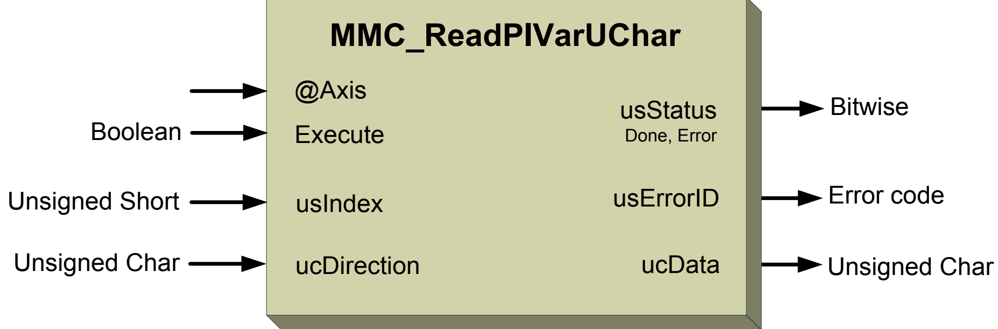


Figure 9-4: MMC_ReadPIVarUChar function

9.7.3.2 Function Block Code Example

```
     case CHAR_BITSIZE:
     {
          // check signess of the variable (signed\unsigned)
          switch(ucVarType)
          {
          case ePI_SIGNED_CHAR:
{
     MMC_READPIVARCHAR_IN *pReadIn = (MMC_READPIVARCHAR_IN *)&ReadIn;
     MMC_READPIVARCHAR_OUT ReadOut;

     rc = MMC_ReadPIVarChar(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
     if(rc)
     {
          printf("MMC_ReadPIVarSignedChar failed error %d",ReadOut.sErrorID);
          break;
     }

     printf("read the signed CHAR value: %d\n",ReadOut.cData);
}
break;
          case ePI_UNSIGNED_CHAR:
{
     MMC_READPIVARUCHAR_IN *pReadIn = (MMC_READPIVARUCHAR_IN *)&ReadIn;
     MMC_READPIVARUCHAR_OUT ReadOut;

     rc = MMC_ReadPIVarUChar(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
     if(rc)
     {
          printf("MMC_ReadPIVarUnsignedChar failed error %d",ReadOut.sErrorID);
          break;
     }
     printf("read the unsigned CHAR value: %d\n",ReadOut.ucData);
}
break;
          default:
{
     printf("CHAR: somethng is wrong\n");
}
break;
          }
     }
     break;
```

9.7.4 MMC_ReadPIVarShort

This function reads a Processing Image input\output Short Variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarShort(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARSHORT_IN* pInParam,
OUT MMC_READPIVARSHORT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARSHORT_IN** input data structure using the MMC_ReadPIVarShort function.

pOutParam

Points to the **MMC_READPIVARSHORT_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarShort function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

PI_SHORT_BITSIZE 16

Scope

All

MMC_READPIVARSHORT_IN Structure

```
typedef struct mmc_readpivarsignedshort_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARSHORT_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARSHORT_OUT Structure

```
typedef struct mmc_readpivarsignedshort_out{
short sData;
unsigned short usStatus;
short sErrorID;
} MMC_READPIVARSHORT_OUT;
```

Parameters

sData

The signed short parameter requested from the processed image.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-5 describes the function for MMC_ReadPIVarShort.

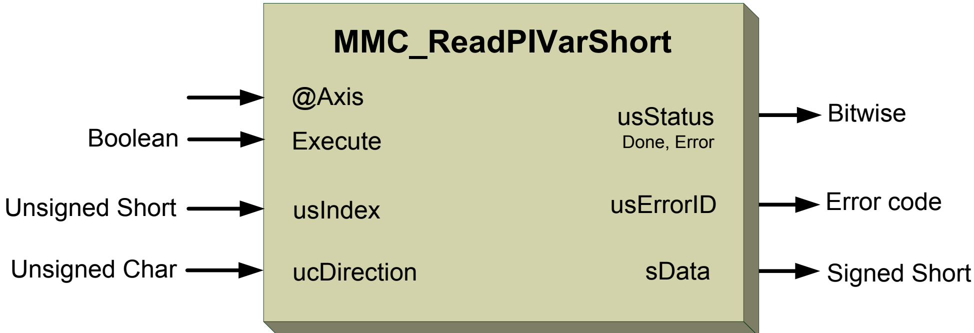


Figure 9-5: MMC_ReadPIVarShort function

9.7.4.2 Function Block Code Example

```
    case SHORT_BITSIZE:
{
    // check signess of the variable (signed\unsigned)
    switch(ucVarType)
    {
    case ePI_SIGNED_SHORT:
        {
            MMC_READPIVARSHORT_IN *pReadIn = (MMC_READPIVARSHORT_IN *)&ReadIn;
            MMC_READPIVARSHORT_OUT ReadOut;

            rc = MMC_ReadPIVarShort(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
            if(rc)
            {
                printf("MMC_ReadPIVarSignedShort failed error %d",ReadOut.sErrorID);
                break;
            }

            printf("read the signed SHORT value: %d\n",ReadOut.sData);
        }
        break;
```

9.7.5 MMC_ReadPIVarUShort

This function reads a Processing Image input\output Unsigned Short Variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarUShort(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARUSHORT_IN* pInParam,
OUT MMC_READPIVARUSHORT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARUSHORT_IN** input data structure using the MMC_ReadPIVarUShort function.

pOutParam

Points to the **MMC_READPIVARUSHORT_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarUShort function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

PI_SHORT_BITSIZE 16

Scope

All

MMC_READPIVARUSHORT_IN Structure

```
typedef struct mmc_readpivarunsignedshort_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARUSHORT_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARUSHORT_OUT Structure

```
typedef struct mmc_readpivarunsignedshort_out{
unsigned short usData;
unsigned short usStatus;
short sErrorID;
} MMC_READPIVARUSHORT_OUT;
```

Parameters

usData

The unsigned short parameter requested from the processed image.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-6 describes the function for MMC_ReadPIVarUShort.

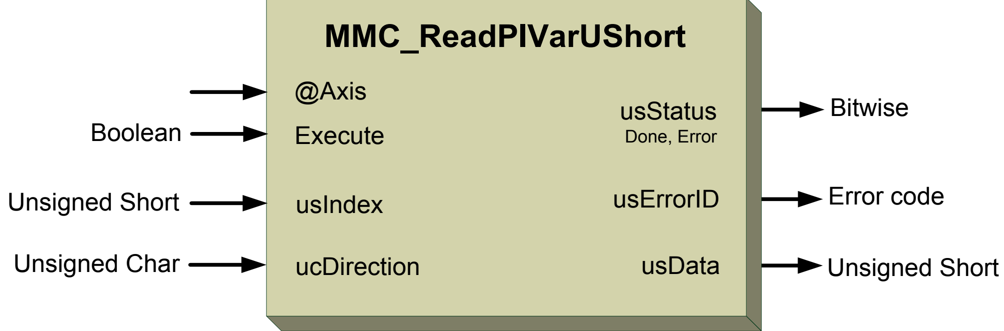


Figure 9-6: MMC_ReadPIVarUShort function

9.7.5.2 Function Block Code Example

```
case ePI_UNSIGNED_SHORT:
         {
              MMC_READPIVARUSHORT_IN *pReadIn = (MMC_READPIVARUSHORT_IN *)&ReadIn;
              MMC_READPIVARUSHORT_OUT ReadOut;

              rc = MMC_ReadPIVarUShort(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
              if(rc)
              {
                   printf("MMC_ReadPIVarUnsignedShort failed error %d",ReadOut.sErrorID);
                   break;
              }

              printf("read the unsigned SHORT value: %d\n",ReadOut.usData);
         }
         break;
    default:
         {
              printf("SHORT: somethng is wrong\n");
         }
         break;
    }
}
break;
```

9.7.6 MMC_ReadPIVarInt

This function reads a Processing Image input\output Integer Variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarInt(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARINT_IN* pInParam,
OUT MMC_READPIVARINT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARINT_IN** input data structure using the MMC_ReadPIVarInt function.

pOutParam

Points to the **MMC_READPIVARINT_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarInt function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

PI_INT_BITSIZE 32

Scope

All

MMC_READPIVARINT_IN Structure

```
typedef struct mmc_readpivarsignedint_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARINT_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARINT_OUT Structure

```
typedef struct mmc_readpivarsignedint_out{
int iData;
unsigned short usStatus;
short sErrorID;
} MMC_READPIVARINT_OUT;
```

Parameters

iData

The integer parameter requested from the processed image.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-7 describes the function for MMC_ReadPIVarInt.

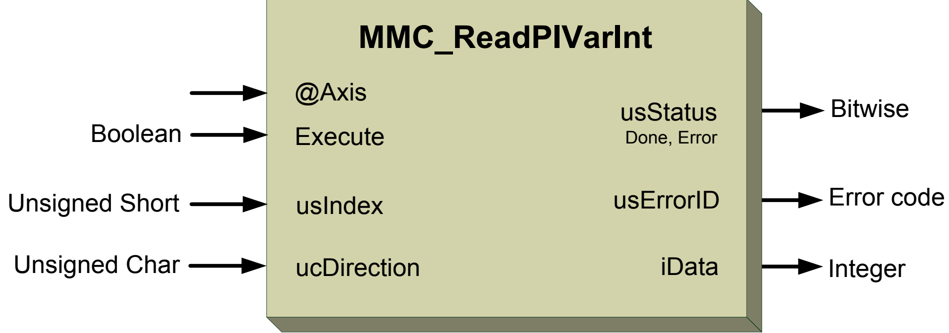


Figure 9-7: MMC_ReadPIVarInt function

9.7.6.2 Function Block Code Example

```
case INT_BITSIZE:
{
     // check signess of the variable (signed\unsigned)
     switch(ucVarType)
     {
     case ePI_SIGNED_INT:
          {
               MMC_READPIVARINT_IN *pReadIn = (MMC_READPIVARINT_IN *)&ReadIn;
               MMC_READPIVARINT_OUT ReadOut;

               rc = MMC_ReadPIVarInt(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
               if(rc)
               {
                    printf("MMC_ReadPIVarSignedInt failed error %d",ReadOut.sErrorID);
                    system("pause");
                    break;
               }

               printf("read the signed INT value from VarOffset %d:
%d\n",ReadIn.usIndex,ReadOut.iData);
          }
          break;
```

9.7.7 MMC_ReadPIVarUInt

This function reads a Processing Image input\output Unsigned Integer Variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarUInt(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARUINT_IN* pInParam,
OUT MMC_READPIVARUINT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARUINT_IN** input data structure using the MMC_ReadPIVarUInt function.

pOutParam

Points to the **MMC_READPIVARUINT_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarUInt function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

PI_INT_BITSIZE 32

Scope

All

MMC_READPIVARUINT_IN Structure

```
typedef struct mmc_readpivarunsignedint_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARUINT_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARUINT_OUT Structure

```
typedef struct mmc_readpivarunsignedint_out{
unsigned int uiData;
unsigned short usStatus;
short sErrorID;
} MMC_READPIVARUINT_OUT;
```

Parameters

uiData

The unsigned integer parameter requested from the processed image.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-8 describes the function for MMC_ReadPIVarUInt.

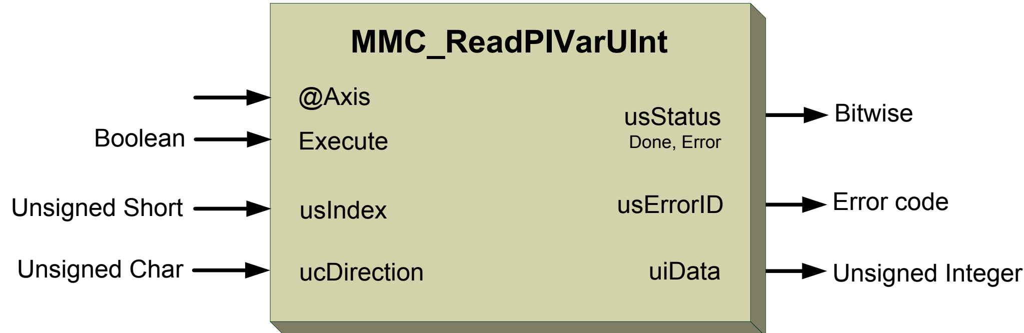


Figure 9-8: MMC_ReadPIVarUInt function

9.7.7.2 Function Block Code Example

```
case ePI_UNSIGNED_INT:
         {
              MMC_READPIVARUINT_IN *pReadIn = (MMC_READPIVARUINT_IN *)&ReadIn;
              MMC_READPIVARUINT_OUT ReadOut;

              rc = MMC_ReadPIVarUInt(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
              if(rc)
              {
                   printf("MMC_ReadPIVarUnsignedInt failed error %d",ReadOut.sErrorID);
                   break;
              }
              printf("read the unsigned INT value from VarOffset %d:
%u\n",ReadOut.uiData,ReadIn.usIndex);
         }
         break;
```

9.7.8 MMC_ReadPIVarFloat

This function reads a Processing Image input\output Float Variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarFloat(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARFLOAT_IN* pInParam,
OUT MMC_READPIVARFLOAT_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARFLOAT_IN** input data structure using the MMC_ReadPIVarFloat function.

pOutParam

Points to the **MMC_READPIVARFLOAT_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarFloat function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH	35
PI_REG_VAR_SIZE	4

Scope

All

MMC_READPIVARFLOAT_IN Structure

```
typedef struct mmc_readpivarfloat_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARFLOAT_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARFLOAT_OUT Structure

```
typedef struct mmc_readpivarfloat_out{
float fData;
unsigned short usStatus;
short sErrorID;
} MMC_READPIVARFLOAT_OUT;
```

Parameters

fData

The float parameter requested from the processed image.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-9 describes the function for MMC_ReadPIVarFloat.

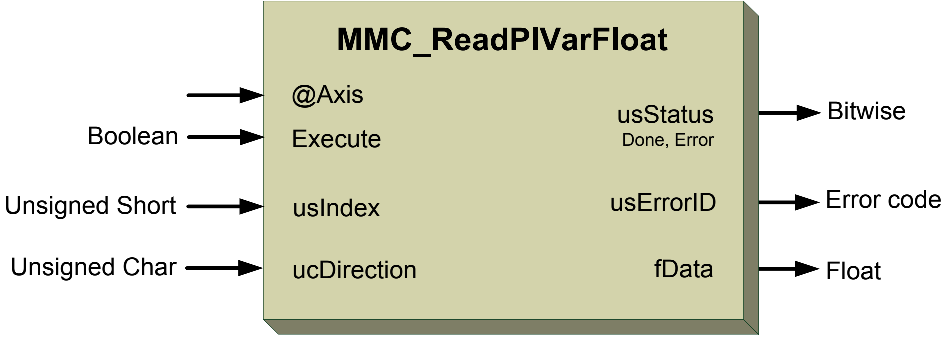


Figure 9-9: MMC_ReadPIVarFloat function

9.7.8.2 Function Block Code Example

```
case ePI_FLOAT:
         {
             MMC_READPIVARFLOAT_IN *pReadIn = (MMC_READPIVARFLOAT_IN *)&ReadIn;
             MMC_READPIVARFLOAT_OUT ReadOut;

             rc = MMC_ReadPIVarFloat(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
             if(rc)
             {
                  printf("MMC_ReadPIVarFloat failed error %d",ReadOut.sErrorID);
                  break;
             }

             printf("read the FLOAT value: %f\n",ReadOut.fData);
         }
         break;
    default:
         {
             printf("INT: somethng is wrong\n");
         }
         break;
    }
```

9.7.9 MMC_ReadPIVarRaw

This function reads a Processing Image input\output RAW Variable according to its index, where the variable ≤ 32 bit

```
MMC_LIB_API int MMC_ReadPIVarRaw(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARRAW_IN* pInParam,
OUT MMC_READPIVARRAW_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARRAW_IN** input data structure using the MMC_ReadPIVarRaw function.

pOutParam

Points to the **MMC_READPIVARRAW_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarRaw function.

Remarks

The processing image defines the following:

PI_REG_VAR_SIZE 4

Scope

All

MMC_READPIVARRAW_IN Structure

```
typedef struct mmc_readpivarraw_in{
unsigned short usIndex;
unsigned char ucDirection;
unsigned char ucByteLength;
} MMC_READPIVARRAW_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

ucByteLength

Length of the RAW data including the BitStart offset.

The BitStart offset is the PI variable bit offset Modulo 8, e.g. Bit Offset = 25 with the BitStart = 1. The Byte length is (BitStart + Bit size) Modulo 8.

MMC_READPIVARRAW_OUT Structure

```
typedef struct mmc_readpivarraw_out{
unsigned short usStatus;
short sErrorID;
unsigned char pRawData[PI_REG_VAR_SIZE];
} MMC_READPIVARRAW_OUT;
```

Parameters

pRawData[PI_REG_VAR_SIZE]

The RAW parameter requested from the processed image. The byte size of the RAW data is (BitStart + Bit size) Modulo 8, which is inserted as the ucByteLength.

Dependant on the variable type array PI_REG_VAR_SIZE which is a maximum of 4.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-10 describes the function for MMC_ReadPIVarRaw.

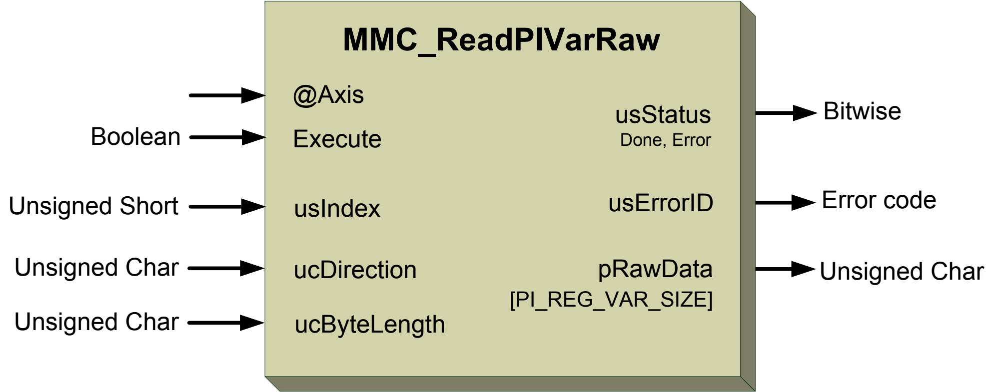


Figure 9-10: MMC_ReadPIVarRaw function

9.7.9.2 Function Block Code Example

```
// bit size is not from the standard types, use read PI raw API
default:
{
    MMC_READPIVARRAW_IN ReadRawIn;
    MMC_READPIVARRAW_OUT ReadOut;

    // bit start is the bits offset from the begining of the byte
    unsigned char ucBitStart =  GetOut.VarInfo.uiBitOffset & 0x07;

    // calculate number of bytes according to the number of bits
    unsigned int uiCount  = ((GetOut.VarInfo.uiBitSize  + ucBitStart - 1) >> 3) + 1;

    unsigned int ui = 0;

    ReadRawIn.ucDirection = ReadIn.ucDirection;
    ReadRawIn.usIndex = ReadIn.usIndex;
    ReadRawIn.ucByteLength = uiCount;
    rc = MMC_ReadPIVarRaw(ConnHndl,hAxisRef[ucID],&ReadRawIn,&ReadOut);
    if(rc)
    {
        printf("MMC_ReadPIVarRaw failed error %d",ReadOut.sErrorID);
        break;
    }
    printf("the %d data bytes are 0x\n",uiCount);
    for(ui = 0; ui < uiCount; ui++)
    {
        printf("[%X]",ReadOut.pRawData[ui]);
    }
    printf("\n");

}
    }
}
    printf("what variable to read? 0 - exit, 1 - input, 2 - output\n");
    scanf("%hu",&usAns);
    }
}
```

9.7.10 MMC_ReadPIVarLongLong

This function reads a Processing Image input\output Long Long Variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarLongLong(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARLONGLONG_IN* pInParam,
OUT MMC_READPIVARLONGLONG_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARLONGLONG_IN** input data structure using the MMC_ReadPIVarRaw function.

pOutParam

Points to the **MMC_READPIVARLONGLONG_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarRaw function.

Remarks

The processing image defines the following:

PI_LONG_LONG_BITSIZE 64

Scope

All

MMC_READPIVARLONGLONG_IN Structure

```
typedef struct mmc_readpivarsignedlonglong_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARLONGLONG_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARLONGLONG_OUT Structure

```
typedef struct mmc_readpivarsignedlonglong_out{
#ifdef WIN32
	__int64 llData;
#else
	long long llData;
#endif
unsigned short usStatus;
short sErrorID;
} MMC_READPIVARLONGLONG_OUT;
```

Parameters

__int64 llData or long long llData

The signed long long parameter requested from the processed image.

If the function is defined for WIN32 then use *__int64 llData*, else use *llData*.

Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-11 describes the function for MMC_ReadPIVarLongLong.

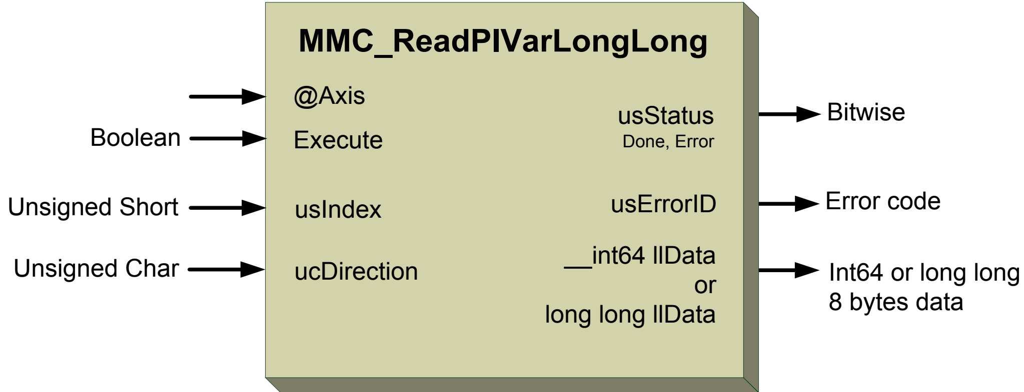


Figure 9-11: MMC_ReadPIVarLongLong function

9.7.11 MMC_ReadPIVarULongLong

This function reads a Processing Image input\output Unsigned Long Long Variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarULongLong(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARULONGLONG_IN* pInParam,
OUT MMC_READPIVARULONGLONG_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARULONGLONG_IN** input data structure using the MMC_ReadPIVarULongLong function.

pOutParam

Points to the **MMC_READPIVARULONGLONG_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarULongLong function.

Remarks

The processing image defines the following:

PI_LONG_LONG_BITSIZE 64

Scope

All

MMC_READPIVARULONGLONG_IN Structure

```
typedef struct mmc_readpivarunsignedlonglong_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARULONGLONG_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARULONGLONG_OUT Structure

```
typedef struct mmc_readpivarunsignedlonglong_out{
#ifdef WIN32
	unsigned__int64 ullData;
#else
	unsigned long long ullData;
#endif
unsigned short usStatus;
short sErrorID;
} MMC_READPIVARULONGLONG_OUT;
```

Parameters

unsigned__int64 ullData or unsigned long long ullData

The unsigned long long parameter requested from the processed image.

If the function is defined for WIN32 then use *unsigned __int64 ullData*, else use *unsigned long long ullData*.

Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-12 describes the function for MMC_ReadPIVarULongLong.

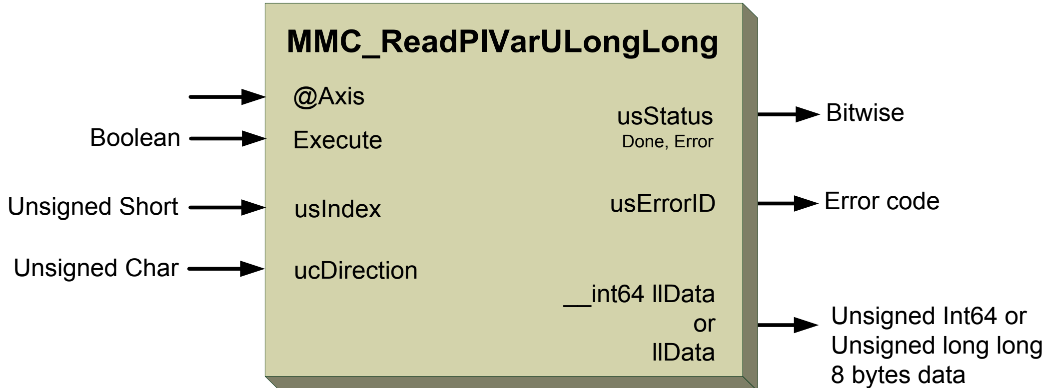


Figure 9-12: MMC_ReadPIVarULongLong function

9.7.12 MMC_ReadPIVarDouble

This function reads a Processing Image input\output Double Variable according to its index

```
MMC_LIB_API int MMC_ReadPIVarDouble(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READPIVARDOUBLE_IN* pInParam,
OUT MMC_READPIVARDOUBLE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READPIVARDOUBLE_IN** input data structure using the MMC_ReadPIVarDouble function.

pOutParam

Points to the **MMC_READPIVARDOUBLE_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarDouble function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

Scope

All

MMC_READPIVARDOUBLE_IN Structure

```
typedef struct mmc_readpivardouble_in{
unsigned short usIndex;
unsigned char ucDirection;
} MMC_READPIVARDOUBLE_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_READPIVARDOUBLE_OUT Structure

```
typedef struct mmc_readpivardouble_out{
double dbData;
unsigned short usStatus;
short sErrorID;
} MMC_READPIVARDOUBLE_OUT;
```

Parameters

dbData

The double parameter requested from the processed image.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-13 describes the function for MMC_ReadPIVarDouble.

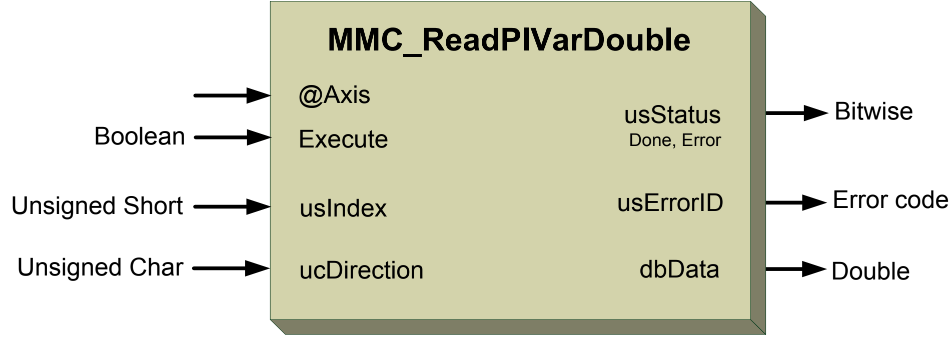


Figure 9-13: MMC_ReadPIVarDouble function

9.7.13 MMC_ReadLargePIVarRaw

This function reads a large Processing Image input\output RAW Variable according to its index, where the variable > 32 bit.

```
MMC_LIB_API int MMC_ReadLargePIVarRaw(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READLARGEPIVARRAW_IN* pInParam,
OUT MMC_READLARGEPIVARRAW_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READLARGEPIVARRAW_IN** input data structure using the MMC_ReadLargePIVarRaw function.

pOutParam

Points to the **MMC_READLARGEPIVARRAW_OUT** output structure receiving information as a result of calling the MMC_ReadLargePIVarRaw function.

Remarks

The processing image defines the following:

PI_LARGE_VAR_SIZE 1400

Scope

All

MMC_READLARGEPIVARRAW_IN Structure

```
typedef struct mmc_readlargepivarraw_in{
unsigned short usIndex;
unsigned short usByteLength;
unsigned char ucDirection;
} MMC_READLARGEPIVARRAW_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

usByteLength

Length of the RAW data including the BitStart offset.

The BitStart offset is the PI variable bit offset Modulo 8, e.g. Bit Offset = 25 with the BitStart = 1. The Byte length is (BitStart + Bit size) Modulo 8.

MMC_READLARGEPIVARRAW_OUT Structure

```
typedef struct mmc_readlargepivarraw_out{
unsigned short usStatus;
short sErrorID;
unsigned char pRawData[PI_LARGE_VAR_SIZE];
} MMC_READLARGEPIVARRAW_OUT;
```

Parameters

pRawData[PI_LARGE_VAR_SIZE]

The RAW parameter requested from the processed image. The byte size of the RAW data is (BitStart + Bit size) Modulo 8, which is inserted as the ucByteLength.

Dependant on the variable type array PI_REG_VAR_SIZE which is a maximum of 4.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-14 describes the function for MMC_ReadLargePIVarRaw.

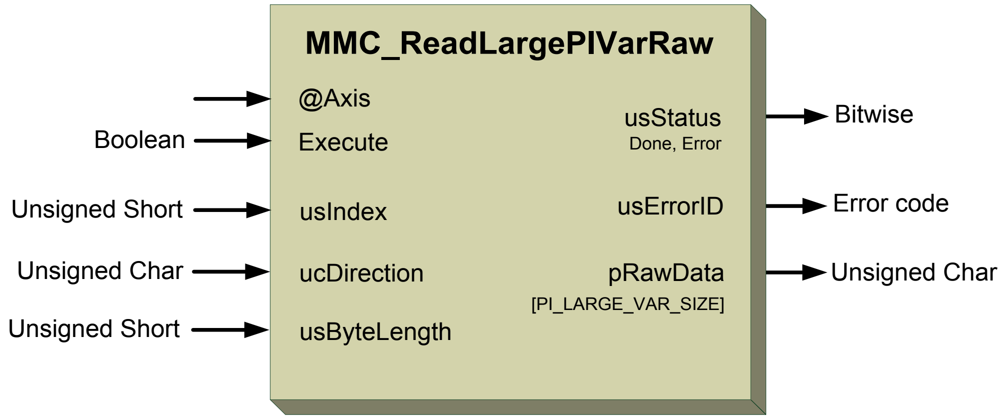


Figure 9-14: MMC_ReadLargePIVarRaw function

9.7.14 MMC_WritePIVarBool

This function writes a Processing Image input\output Boolean variable according to its index

```
MMC_LIB_API int MMC_WritePIVarBool(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARBOOL_IN* pInParam,
OUT MMC_WRITEPIVARBOOL_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARBOOL_IN** input data structure using the MMC_WritePIVarBool function.

pOutParam

Points to the **MMC_WRITEPIVARBOOL_OUT** output structure receiving information as a result of calling the MMC_WritePIVarBool function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

Scope

All

MMC_WRITEPIVARBOOL_IN Structure

```
typedef struct mmc_writepivarbool_in{
unsigned short usIndex;
unsigned char ucBOOL;
} MMC_WRITEPIVARBOOL_IN;
```

Parameters

ucBOOL

The Boolean parameter to be written to the processed image.

usIndex

The required index to be written to the processed image. Any +ve integer values.

MMC_WRITEPIVARBOOL_OUT Structure

```
typedef struct mmc_writepivarbool_out{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEPIVARBOOL_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-15 describes the function for MMC_WritePIVarBool.

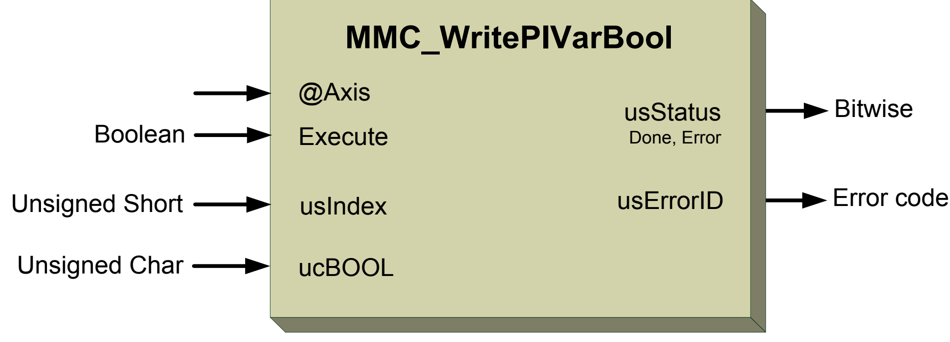


Figure 9-15: MMC_WritePIVarBool function

9.7.14.2 Function Block Code Example

```
case BOOL_BITSIZE:
{
     MMC_WRITEPIVARBOOL_IN WriteIn;
     MMC_WRITEPIVARBOOL_OUT WriteOut;

     WriteIn.usIndex = usIndex;

     printf("enter a BOOL value:\n");
     scanf("%hhu",&WriteIn.ucBOOL);

     rc = MMC_WritePIVarBool(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
     if(rc)
     {
          short sErr = WriteOut.sErrorID;

          printf("MMC_WritePIVarBool failed error %d",sErr);
          PrintError(sErr);
          break;
     }

}
break;
```

9.7.15 MMC_WritePIVarChar

This function writes a Processing Image input\output Character variable according to its index

MMC_LIB_API int MMC_WritePIVarChar(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARCHAR_IN* pInParam,
OUT MMC_WRITEPIVARCHAR_OUT* pOutParam
);

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARCHAR_IN** input data structure using the MMC_WritePIVarChar function.

pOutParam

Points to the **MMC_WRITEPIVARCHAR_OUT** output structure receiving information as a result of calling the MMC_WritePIVarChar function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

PI_INT_BITSIZE 32

Scope

All

MMC_WRITEPIVARCHAR_IN Structure

```
typedef struct mmc_writepivarsignedchar_in{
unsigned short usIndex;
signed char cData;
} MMC_WRITEPIVARCHAR_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

cData

The signed character parameter to be written to the processed image.

MMC_WRITEPIVARCHAR_OUT Structure

```
typedef struct mmc_writepivarsignedchar_out{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEPIVARCHAR_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-16 describes the function for MMC_WritePIVarChar.

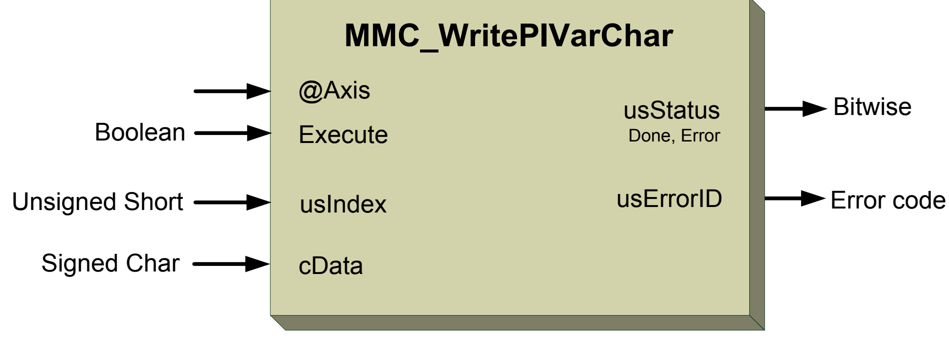


Figure 9-16: MMC_WritePIVarChar function

9.7.15.2 Function Block Code Example

```
case CHAR_BITSIZE:
{
     // check signess of the variable (signed\unsigned)
     switch(ucVarType)
     {
     case ePI_SIGNED_CHAR:
          {
               MMC_WRITEPIVARCHAR_IN WriteIn;
               MMC_WRITEPIVARCHAR_OUT WriteOut;
               char cData = 0;

               WriteIn.usIndex = usIndex;

               printf("enter a signed CHAR value to be written to index %d:\n",usIndex);
               scanf("%hhd",&cData);
               WriteIn.cData = cData;

               rc = MMC_WritePIVarChar(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
               if(rc)
               {
                    short sErr = WriteOut.sErrorID;

                    printf("MMC_WritePIVarSignedChar failed error %d",sErr);
                    PrintError(sErr);
                    break;
               }
          }
          break;
```

9.7.16 MMC_WritePIVarUChar

This function writes a Processing Image input\output Unsigned Character variable according to its index

```
MMC_LIB_API int MMC_WritePIVarUChar(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARUCHAR_IN* pInParam,
OUT MMC_WRITEPIVARUCHAR_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARUCHAR_IN** input data structure using the MMC_WritePIVarUChar function.

pOutParam

Points to the **MMC_WRITEPIVARUCHAR_OUT** output structure receiving information as a result of calling the MMC_WritePIVarUChar function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

PI_REG_VAR_SIZE 4

Scope

All

MMC_WRITEPIVARUCHAR_IN Structure

```
typedef struct mmc_writepivarunsignedchar_in{
unsigned short usIndex;
unsigned char ucData;
} MMC_WRITEPIVARUCHAR_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

ucData

The unsigned character parameter to be written to the processed image.

MMC_WRITEPIVARUCHAR_OUT Structure

```
typedef struct mmc_writepivarunsignedchar_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARUCHAR_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-17 describes the function for MMC_WritePIVarUChar.

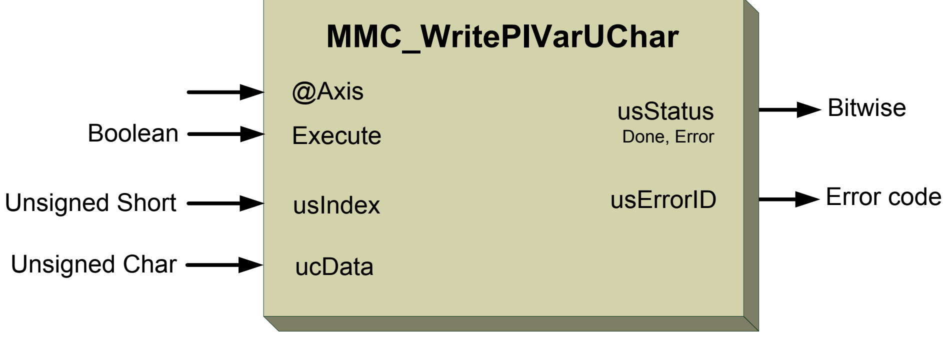


Figure 9-17: MMC_WritePIVarUChar function

9.7.16.2 Function Block Code Example

```
case ePI_UNSIGNED_CHAR:
         {
              MMC_WRITEPIVARUCHAR_IN WriteIn;
              MMC_WRITEPIVARUCHAR_OUT WriteOut;
              unsigned char ucData = 0;

              WriteIn.usIndex = usIndex;

              printf("enter an usigned CHAR value:\n");
              scanf("%hhu",&ucData);
              WriteIn.ucData = ucData;
              printf("scanned the value %d, %hhu\n",ucData,ucData);

              rc = MMC_WritePIVarUChar(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
              if(rc)
              {
                   short sErr = WriteOut.sErrorID;

                   printf("MMC_WritePIVarUnsignedChar failed error %d",sErr);
                   PrintError(sErr);
                   break;
              }
         }
         break;
    default:
         {
              printf("someting is wrong\n");
         }
         break;
    }
}
```

9.7.17 MMC_WritePIVarUShort

This function writes a Processing Image input\output Unsigned Short variable according to its index

```
MMC_LIB_API int MMC_WritePIVarUShort(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARUSHORT_IN* pInParam,
OUT MMC_WRITEPIVARUSHORT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARUSHORT_IN** input data structure using the MMC_WritePIVarUShort function.

pOutParam

Points to the **MMC_WRITEPIVARUSHORT_OUT** output structure receiving information as a result of calling the MMC_WritePIVarUShort function.

Remarks

The processing image defines the following:

PI_SHORT_BITSIZE 16

Scope

All

MMC_WRITEPIVARUSHORT_IN Structure

```
typedef struct mmc_writepivarunsignedshort_in{
unsigned short usData;
unsigned short usIndex;
} MMC_WRITEPIVARUSHORT_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

usData

The unsigned short parameter to be written to the processed image.

MMC_WRITEPIVARUSHORT_OUT Structure

```
typedef struct mmc_writepivarunsignedshort_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARUSHORT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-18 describes the function for MMC_WritePIVarUShort.

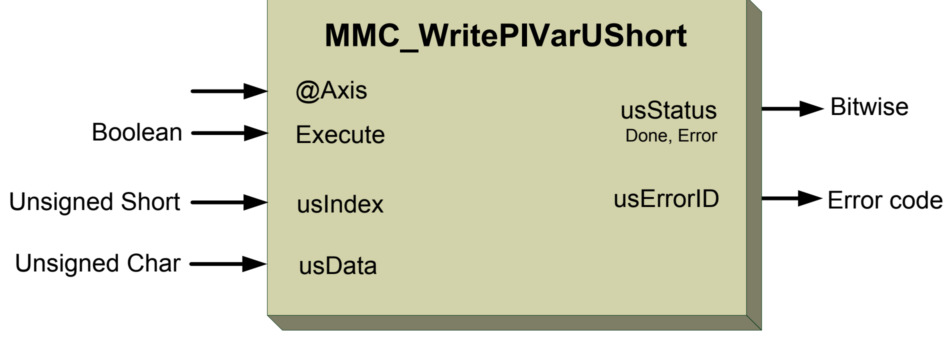


Figure 9-18: MMC_WritePIVarUShort function

9.7.17.2 Function Block Code Example

```
case ePI_UNSIGNED_SHORT:
         {
              MMC_WRITEPIVARUSHORT_IN WriteIn;
              MMC_WRITEPIVARUSHORT_OUT WriteOut;
              unsigned short usData = 0;

              WriteIn.usIndex = usIndex;

              printf("enter an unsigned SHORT value to be written to index %d:\n",usIndex);
              scanf("%hu",&usData);

              WriteIn.usData = usData;

              printf("scanned the data %d, now writing to index 
%d\n",WriteIn.usData,WriteIn.usIndex);

              rc = MMC_WritePIVarUShort(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
              if(rc)
              {
                   short sErr = WriteOut.sErrorID;

                   printf("MMC_WritePIVarUnsignedShort failed error %d",sErr);
                   PrintError(sErr);
                   break;
              }
         }
         break;
     default:
         {
              printf("someting is wrong\n");
         }
         break;
     }
}
break;
```

9.7.18 MMC_WritePIVarShort

This function writes a Processing Image input\output Short variable according to its index

```
MMC_LIB_API int MMC_WritePIVarShort(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARSHORT_IN* pInParam,
OUT MMC_WRITEPIVARSHORT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARSHORT_IN** input data structure using the MMC_WritePIVarShort function.

pOutParam

Points to the **MMC_WRITEPIVARSHORT_OUT** output structure receiving information as a result of calling the MMC_WritePIVarShort function.

Remarks

The processing image defines the following:

PI_SHORT_BITSIZE 16

Scope

All

MMC_WRITEPIVARSHORT_IN Structure

typedef struct mmc_writepivarsignedshort_in{
short sData;
unsigned short usIndex;
} MMC_WRITEPIVARSHORT_IN;

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

sData

The signed short parameter to be written to the processed image.

MMC_WRITEPIVARSHORT_OUT Structure

typedef struct mmc_writepivarsignedshort_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARSHORT_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-19 describes the function for MMC_WritePIVarShort.

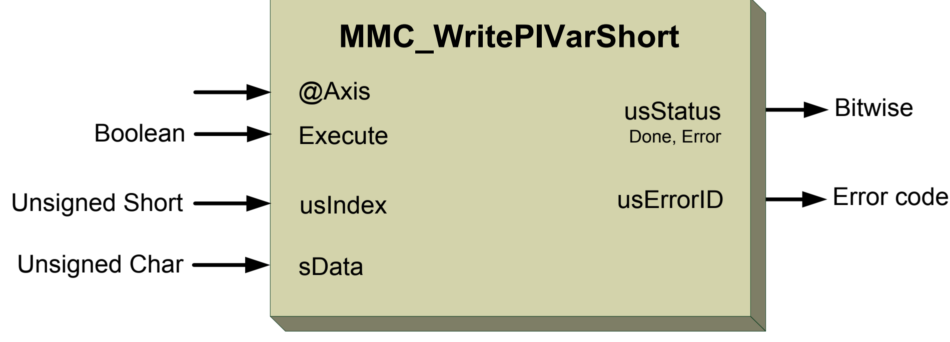


Figure 9-19: MMC_WritePIVarShort function

9.7.18.2 Function Block Code Example

```
case ePI_SIGNED_SHORT:
         {
              MMC_WRITEPIVARSHORT_IN WriteIn;
              MMC_WRITEPIVARSHORT_OUT WriteOut;
              short sData = 0;

              WriteIn.usIndex = usIndex;

              printf("enter a signed SHORT value to be written to index %d:\n",usIndex);
              scanf("%hd",&sData);

              WriteIn.sData = sData;

              printf("scanned the data %d, now writing to index 
%d\n",WriteIn.sData,WriteIn.usIndex);

              rc = MMC_WritePIVarShort(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
              if(rc)
              {
                   short sErr = WriteOut.sErrorID;

                   printf("MMC_WritePIVarSignedShort failed error %d",sErr);
                   PrintError(sErr);
                   break;
              }
         }
```

9.7.19 MMC_WritePIVarUInt

This function writes a Processing Image input\output Unsigned Integer Variable according to its index

```
MMC_LIB_API int MMC_WritePIVarUInt(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARUINT_IN* pInParam,
OUT MMC_WRITEPIVARUINT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARUINT_IN** input data structure using the MMC_WritePIVarUInt function.

pOutParam

Points to the **MMC_WRITEPIVARUINT_OUT** output structure receiving information as a result of calling the MMC_WritePIVarUInt function.

Remarks

The processing image defines the following:

PI_INT_BITSIZE 32

Scope

All

MMC_WRITEPIVARUINT_IN Structure

```
typedef struct mmc_writepivarunsignedint_in{
unsigned int uiData;
unsigned short usIndex;
} MMC_WRITEPIVARUINT_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

uiData

The unsigned integer parameter to be written to the processed image.

MMC_WRITEPIVARUINT_OUT Structure

```
typedef struct mmc_writepivarunsignedint_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARUINT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-20 describes the function for MMC_WritePIVarUInt.

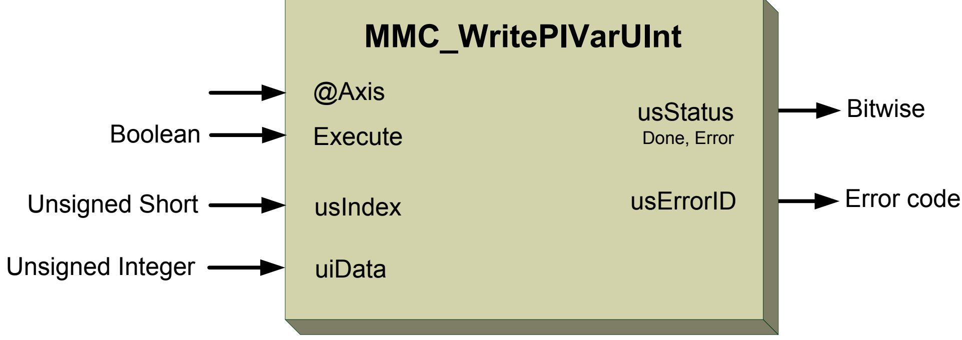


Figure 9-20: MMC_WritePIVarUInt function

9.7.19.2 Function Block Code Example

```
case ePI_UNSIGNED_INT:
         {
             MMC_WRITEPIVARUINT_IN WriteIn;
             MMC_WRITEPIVARUINT_OUT WriteOut;
             unsigned int uiData = 0;

             WriteIn.usIndex = usIndex;

             printf("enter an unsigned INT value:\n");
             scanf("%d",&uiData);

             WriteIn.uiData = uiData;

             printf("writing the value %d, to index %d\n",WriteIn.uiData,WriteIn.usIndex);

             rc = MMC_WritePIVarUInt(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
             if(rc)
             {
                  short sErr = WriteOut.sErrorID;

                  printf("MMC_WritePIVarUnsignedInt failed error %d",sErr);
                  PrintError(sErr);
                  break;
             }
         }
         break;
```

9.7.20 MMC_WritePIVarInt

This function writes a Processing Image input\output Integer Variable according to its index

```
MMC_LIB_API int MMC_WritePIVarInt(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARINT_IN* pInParam,
OUT MMC_WRITEPIVARINT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARINT_IN** input data structure using the MMC_WritePIVarInt function.

pOutParam

Points to the **MMC_WRITEPIVARINT_OUT** output structure receiving information as a result of calling the MMC_WritePIVarInt function.

Remarks

The processing image defines the following:

PI_INT_BITSIZE 32

Scope

All

MMC_WRITEPIVARINT_IN Structure

```
typedef struct mmc_writepivarsignedint_in{
int iData;
unsigned short usIndex;
} MMC_WRITEPIVARINT_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

iData

The signed integer parameter to be written to the processed image.

MMC_WRITEPIVARINT_OUT Structure

```
typedef struct mmc_writepivarsignedint_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARINT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-21 describes the function for MMC_WritePIVarInt.

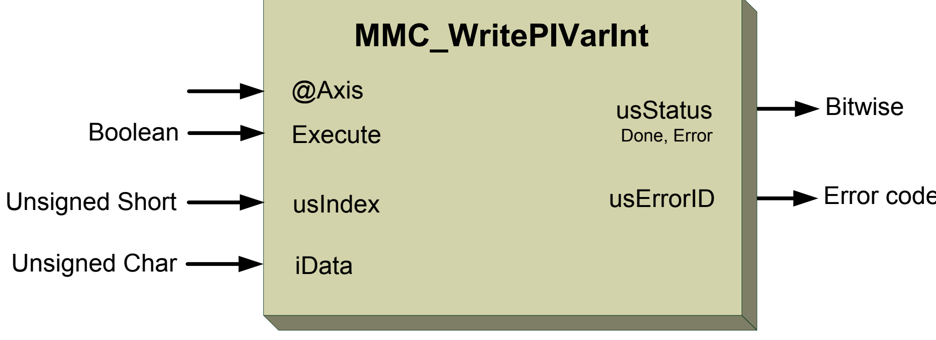


Figure 9-21: MMC_WritePIVarInt function

9.7.20.2 Function Block Code Example

```
case INT_BITSIZE:
{
     // check signess of the variable (signed\unsigned)
     switch(ucVarType)
     {
     case ePI_SIGNED_INT:
          {
               MMC_WRITEPIVARINT_IN WriteIn;
               MMC_WRITEPIVARINT_OUT WriteOut;
               int iData = 0;

               WriteIn.usIndex = usIndex;

               printf("enter a signed INT value:\n");
               scanf("%d",&iData);

               WriteIn.iData = iData;

               rc = MMC_WritePIVarInt(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
               if(rc)
               {
                    short sErr = WriteOut.sErrorID;

                    printf("MMC_WritePIVarSignedInt failed error %d",sErr);
                    PrintError(sErr);
                    break;
               }
          }
```

9.7.21 MMC_WritePIVarFloat

This function writes a Processing Image input\output Float Variable according to its index

```
MMC_LIB_API int MMC_WritePIVarFloat(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARFLOAT_IN* pInParam,
OUT MMC_WRITEPIVARFLOAT_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARFLOAT_IN** input data structure using the MMC_WritePIVarFloat function.

pOutParam

Points to the **MMC_WRITEPIVARFLOAT_OUT** output structure receiving information as a result of calling the MMC_WritePIVarFloat function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH	35
PI_REG_VAR_SIZE	4

Scope

All

MMC_WRITEPIVARFLOAT_IN Structure

```
typedef struct mmc_writepivarfloat_in{
float fData;
unsigned short usIndex;
} MMC_WRITEPIVARFLOAT_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

fData

The float parameter to be written to the processed image.

MMC_WRITEPIVARFLOAT_OUT Structure

```
typedef struct mmc_writepivarfloat_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARFLOAT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted

Done

CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-22 describes the function for MMC_WritePIVarFloat.

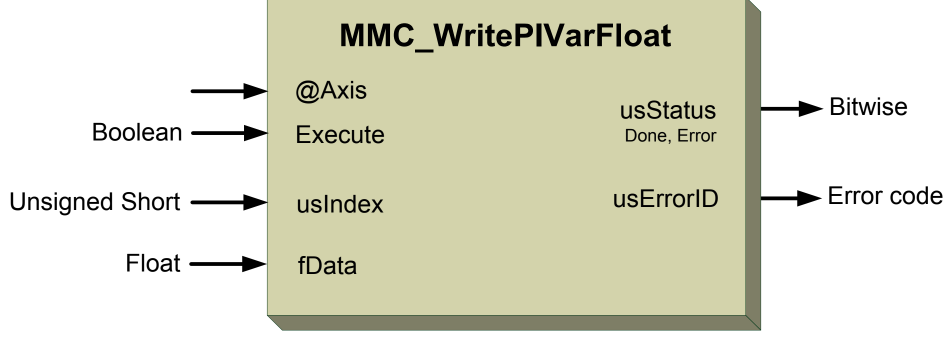


Figure 9-22: MMC_WritePIVarFloat function

9.7.21.2 Function Block Code Example

```
case ePI_FLOAT:
         {
              MMC_WRITEPIVARFLOAT_IN WriteIn;
              MMC_WRITEPIVARFLOAT_OUT WriteOut;
              float fData = 0;

              WriteIn.usIndex = usIndex;

              printf("enter FLOAT value:\n");
              scanf("%f",&fData);

              WriteIn.fData = fData;

              rc = MMC_WritePIVarFloat(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
              if(rc)
              {
                   short sErr = WriteOut.sErrorID;

                   printf("MMC_WritePIVarFloat failed error %d",sErr);
                   PrintError(sErr);
                   break;
              }
         }break;
    default:
         {
              printf("someting is wrong\n");
         }
```

9.7.22 MMC_WritePIVarRaw

This function writes a Processing Image input\output RAW Variable according to its index

```
MMC_LIB_API int MMC_WritePIVarRaw(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARRAW_IN* pInParam,
OUT MMC_WRITEPIVARRAW_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARRAW_IN** input data structure using the MMC_WritePIVarRaw function.

pOutParam

Points to the **MMC_WRITEPIVARRAW_OUT** output structure receiving information as a result of calling the MMC_WritePIVarRaw function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH 35

PI_REG_VAR_SIZE 4

Scope

All

MMC_WRITEPIVARRAW_IN Structure

```
typedef struct mmc_writepivarraw_in{
unsigned short usIndex;
unsigned char ucByteLength;
unsigned char pRawData[PI_REG_VAR_SIZE];
} MMC_WRITEPIVARRAW_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

pRawData[PI_REG_VAR_SIZE]

The RAW parameter to be written to the processed image. The byte size of the RAW data is (BitStart + Bit size) Modulo 8, which is inserted as the ucByteLength.

Dependant on the variable type array PI_REG_VAR_SIZE which is a maximum of 4.

ucByteLength

Length of the RAW data including the BitStart offset.

The BitStart offset is the PI variable bit offset Modulo 8, e.g. Bit Offset = 25 with the BitStart = 1. The Byte length is (BitStart + Bit size) Modulo 8.

MMC_WRITEPIVARRAW_OUT Structure

```
typedef struct mmc_writepivarraw_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARRAW_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-23 describes the function for MMC_WritePIVarRaw.

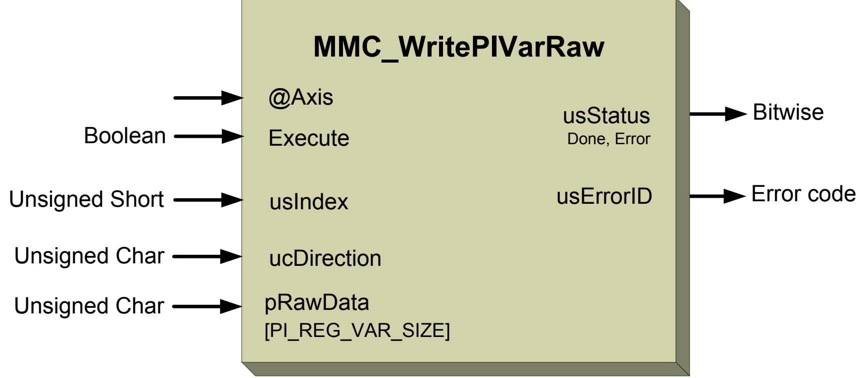


Figure 9-23: MMC_WritePIVarRaw function

9.7.22.2 Function Block Code Example

```
// 24 bits variables need to be written as raw data
case 24:
{
    MMC_WRITEPIVARRAW_IN WriteIn;
    MMC_WRITEPIVARRAW_OUT WriteOut;
    unsigned char *pRawData = WriteIn.pRawData;
    unsigned char ucBitStart =  GetOut.VarInfo.uiBitOffset & 0x07;
    unsigned int uiCount   = ((GetOut.VarInfo.uiBitSize  + ucBitStart - 1) >> 3) + 1;

    WriteIn.usIndex = usIndex;
    WriteIn.ucByteLength = uiCount;

    printf("writing the value 0x[1][2][3] to the variable\n");

    pRawData[0] = 1;
    pRawData[1] = 2;
    pRawData[2] = 3;

    rc = MMC_WritePIVarRaw(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
    if(rc)
    {
            short sErr = WriteOut.sErrorID;

            printf("MMC_WritePIVarRaw failed error %d",sErr);
            PrintError(sErr);
            break;
    }
}
                            }
                        }

    printf("write? 0 - exit, 1 - write\n");
    scanf("%hu",&usAns);
    }
}
```

9.7.22.3 Function Block Code IEC XML Example 1

From within the EASII application, import the XML program ..\NetHelp\Software Manual Pdfs\piTestRAW.xml to the IEC projects on the IEC Motion Programming page, and run the program.

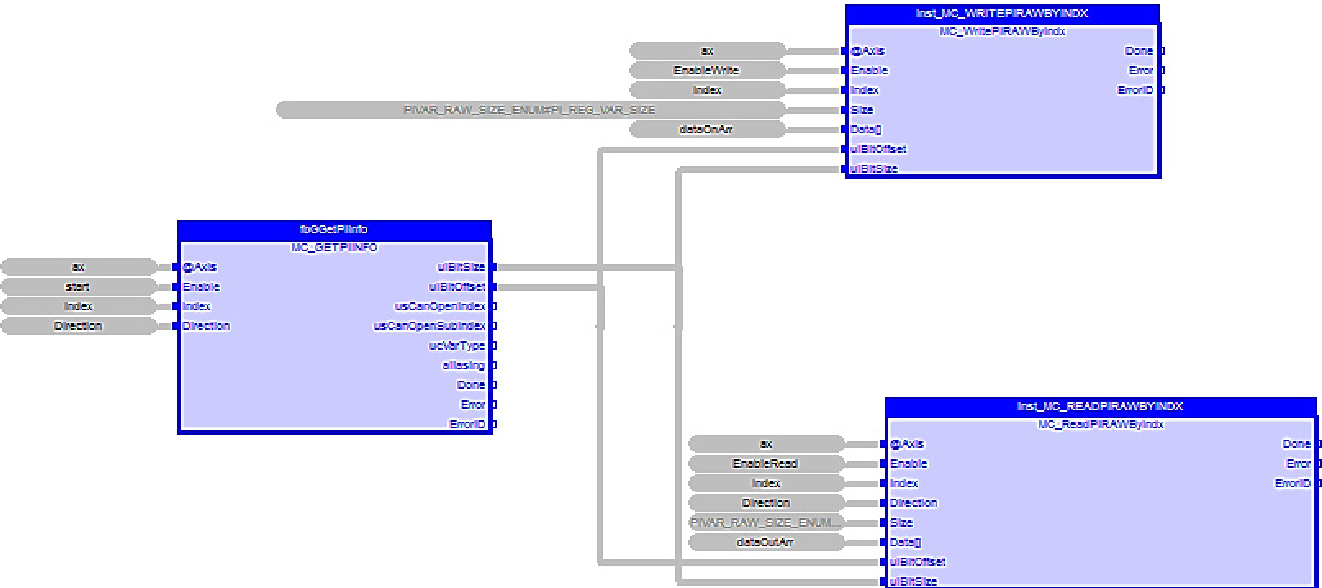


Figure 9-24: IEC Function MMC_ReadPIVarRaw and MMC_WritePIVarRaw

9.7.23 MMC_WritePIVarULongLong

This function writes a Processing Image input\output Unsigned Long Long Variable according to its index

MMC_LIB_API int MMC_WritePIVarULongLong(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARULONGLONG_IN* pInParam,
OUT MMC_WRITEPIVARULONGLONG_OUT* pOutParam
);

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARULONGLONG_IN** input data structure using the MMC_WritePIVarULongLong function.

pOutParam

Points to the **MMC_WRITEPIVARULONGLONG_OUT** output structure receiving information as a result of calling the MMC_WritePIVarULongLong function.

Remarks

The processing image defines the following:

PI_LONG_LONG_BITSIZE 64

Scope

All

MMC_WRITEPIVARULONGLONG_IN Structure

```
typedef struct mmc_writelargepivarunsignedlonglong_in{
#ifdef WIN32
        unsigned __int64 ullData;
#else
        unsigned long long ullData;
#endif
unsigned short usIndex;
} MMC_WRITEPIVARULONGLONG_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

unsigned __int64 ullData or unsigned long long ullData

The unsigned long long parameter to be written to the processed image.

If the function is defined for WIN32 then use *unsigned __int64 ullData*, else use *unsigned long long ullData*.

Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

MMC_WRITEPIVARULONGLONG_OUT Structure

```
typedef struct mmc_writepivarunlonglong_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARULONGLONG_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-25 describes the function for MMC_WritePIVarULongLong.

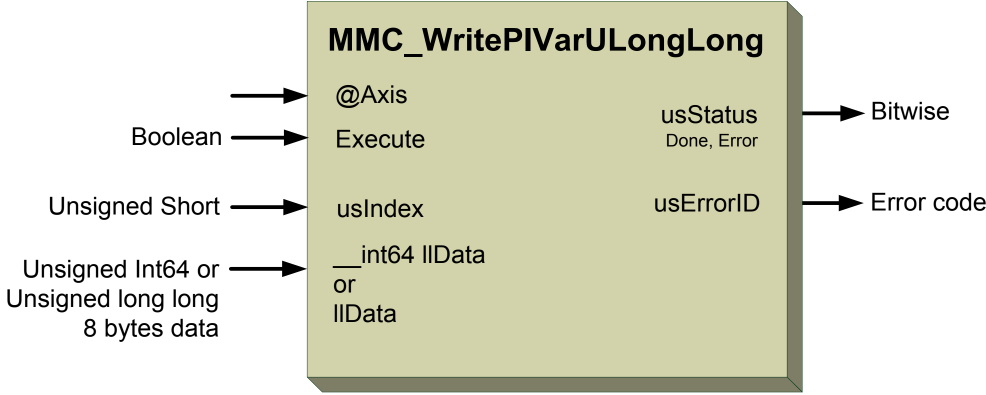


Figure 9-25: MMC_WritePIVarULongLong function

9.7.24 MMC_WritePIVarLongLong

This function writes a Processing Image input\output Long Long Variable according to its index

```
MMC_LIB_API int MMC_WritePIVarLongLong(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARLONGLONG_IN* pInParam,
OUT MMC_WRITEPIVARLONGLONG_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARLONGLONG_IN** input data structure using the MMC_WritePIVarLongLong function.

pOutParam

Points to the **MMC_WRITEPIVARLONGLONG_OUT** output structure receiving information as a result of calling the MMC_WritePIVarLongLong function.

Remarks

The processing image defines the following:

PI_LONG_LONG_BITSIZE 64

Scope

All

MMC_WRITEPIVARLONGLONG_IN Structure

```
typedef struct mmc_writepivarlonglong_in{
#ifdef WIN32
        __int64 llData;
#else
        long long llData;
#endif
        unsigned short usIndex;
} MMC_WRITEPIVARLONGLONG_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

__int64 ullData or long long ullData

The long long parameter to be written to the processed image.

If the function is defined for WIN32 then use *__int64 ullData*, else use *long long ullData*.

Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

MMC_WRITEPIVARLONGLONG_OUT Structure

```
typedef struct mmc_writepivarlonglong_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARLONGLONG_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-26 describes the function for MMC_WritePIVarLongLong.

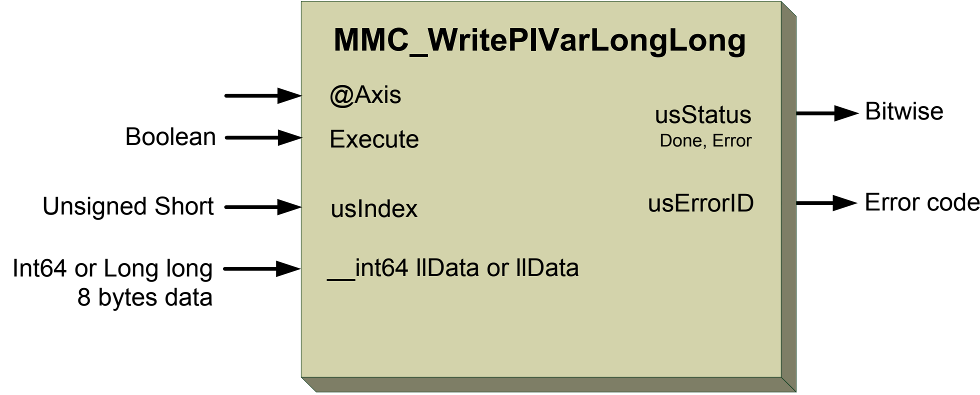


Figure 9-26: MMC_WritePIVarLongLong function

9.7.25 MMC_WritePIVarDouble

This function writes a Processing Image input\output Double Variable according to its index

```
MMC_LIB_API int MMC_WritePIVarDouble(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEPIVARDOUBLE_IN* pInParam,
OUT MMC_WRITEPIVARDOUBLE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEPIVARDOUBLE_IN** input data structure using the MMC_WritePIVarDouble function.

pOutParam

Points to the **MMC_WRITEPIVARDOUBLE_OUT** output structure receiving information as a result of calling the MMC_WritePIVarDouble function.

Remarks

The processing image defines the following:

PI_ALIASING_LENGTH	35
PI_REG_VAR_SIZE	4

Scope

All

MMC_WRITEPIVARDOUBLE_IN Structure

```
typedef struct mmc_writepivardouble_in{
double dbVal;
unsigned short usIndex;
} MMC_WRITEPIVARDOUBLE_IN;
```

Parameters

usIndex

The required index to be written to the processed image. Any +ve integer values.

dbVal

The double parameterto be written to the processed image.

MMC_WRITEPIVARDOUBLE_OUT Structure

```
typedef struct mmc_writepivardouble_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITEPIVARDOUBLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-27 describes the function for MMC_WritePIVarDouble.

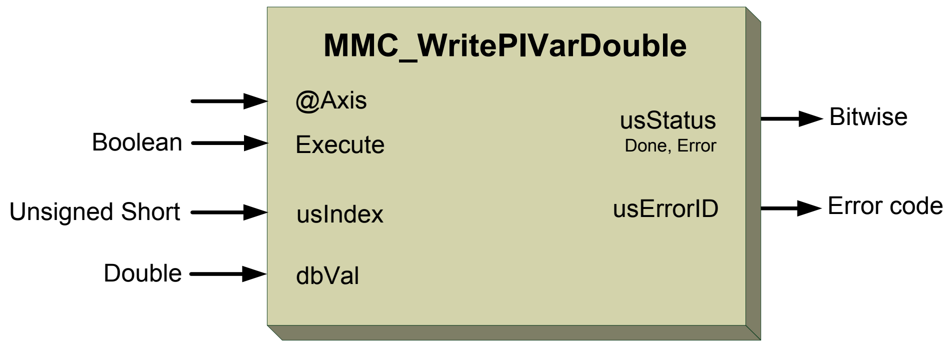


Figure 9-27: MMC_WritePIVarDouble function

9.7.26 MMC_WriteLargePIVarRaw

This function writes a large Processing Image input\output RAW Variable according to its index

MMC_LIB_API int MMC_WriteLargePIVarRaw(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITELARGEPIVARRAW_IN* pInParam,
OUT MMC_WRITELARGEPIVARRAW_OUT* pOutParam
);

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITELARGEPIVARRAW_IN** input data structure using the MMC_WriteLargePIVarRaw function.

pOutParam

Points to the **MMC_WRITELARGEPIVARRAW_OUT** output structure receiving information as a result of calling the MMC_WriteLargePIVarRaw function.

Remarks

The processing image defines the following:

PI_LARGE_VAR_SIZE	1400
PI_ALIASING_LENGTH	35
PI_REG_VAR_SIZE	4

Scope

All

MMC_WRITELARGEPIVARRAW_IN Structure

```
typedef struct mmc_writelargepivarraw_in{
unsigned short usIndex;
unsigned short usByteLength;
unsigned char pRawData[PI_LARGE_VAR_SIZE];
} MMC_WRITELARGEPIVARRAW_IN;
```

Parameters

usIndex

The required index to be read from the processed image. Any +ve integer values.

usByteLength

Length of the RAW data including the BitStart offset.

The BitStart offset is the PI variable bit offset Modulo 8, e.g. Bit Offset = 25 with the BitStart = 1. The Byte length is (BitStart + Bit size) Modulo 8.

pRawData[PI_LARGE_VAR_SIZE]

The RAW parameter requested from the processed image. The byte size of the RAW data is (BitStart + Bit size) Modulo 8, which is inserted as the ucByteLength.

Dependant on the variable type array PI_LARGE_VAR_SIZE which is a maximum of 1400.

MMC_WRITELARGEPIVARRAW_OUT Structure

```
typedef struct mmc_writelargepivarraw_out{
unsigned short usStatus;
short sErrorID;
} MMC_WRITELARGEPIVARRAW_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-28 describes the function for MMC_WriteLargePIVarRaw.

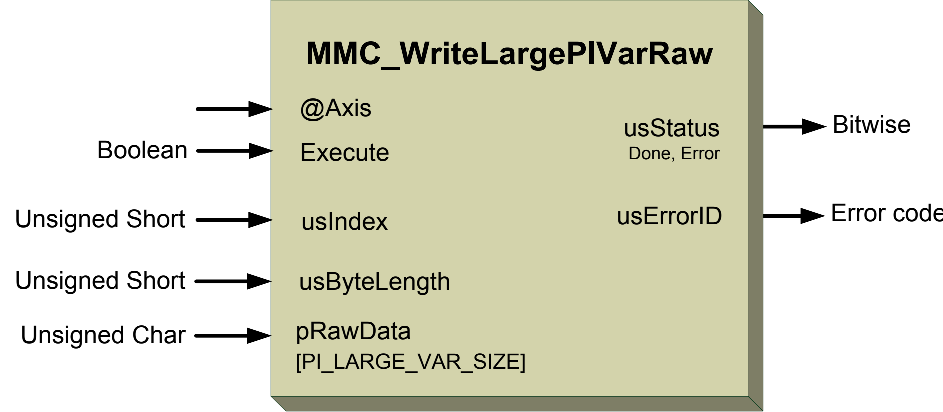


Figure 9-28: MMC_WriteLargePIVarRaw function

9.7.27 MMC_GetPIVarInfo

This function returns the detailed information about a required Processing Image variable, reading the variable according to its index

```
MMC_LIB_API int MMC_GetPIVarInfo(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETPIVARINFO_IN* pInParam,
OUT MMC_GETPIVARINFO_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETPIVARINFO_IN** input data structure using the MMC_ReadPIVarInfo function.

pOutParam

Points to the **MMC_GETPIVARINFO_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarInfo function.

Remarks

The processing image defines the following:

PI_LARGE_VAR_SIZE	1400
PI_ALIASING_LENGTH	35
PI_SHORT_BITSIZE	16
PI_INT_BITSIZE	32
PI_LONG_LONG_BITSIZE	64
PI_REG_VAR_SIZE	4

Scope

All

MMC_GETPIVARINFO_IN Structure

```
typedef struct mmc_getvarinfo_in{
unsigned short usPIVarIndex;
unsigned char ucDirection;
} MMC_GETPIVARINFO_IN;
```

Parameters

usPIVarIndex

The required PI variable index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_GETPIVARINFO_OUT Structure

```
typedef struct mmc_getpivarinfo_out{
NC_PI_ENTRY VarInfo;
unsigned short usStatus;
short sErrorID;
} MMC_GETPIVARINFO_OUT;
```

Parameters

NC_PI_ENTRY VarInfo

```
typedef struct pientry{
unsigned int uiBitSize;
unsigned int uiBitOffset;
unsigned short usCanOpenIndex;
unsigned char ucCanOpenSubIndex;
unsigned char ucVarType;
char pAliasing[PI_ALIASING_LENGTH];
unsigned char ucPadding;
} NC_PI_ENTRY;
```

uiBitSize

The size of the Bit. Any +ve integer values

uiBitOffset

The Offset of the Bit. Any +ve integer values.

usCanOpenIndex

CANopen index. 2 Byte +ve values

ucCanOpenSubIndex

CANopen subindex. +ve character values

ucVarType

Variable type according to the following enumerator values:

ePI_BOOL = 0,
ePI_SIGNED_CHAR = 1,
ePI_UNSIGNED_CHAR = 2,
ePI_SIGNED_SHORT = 3,
ePI_UNSIGNED_SHORT = 4,
ePI_SIGNED_INT = 5,
ePI_UNSIGNED_INT = 6,
ePI_SIGNED_LONG_LONG = 7,
ePI_UNSIGNED_LONG_LONG = 8,
ePI_FLOAT = 9,
ePI_DOUBLE = 10,
ePI_BITWISE = 11,
ePI_8MULTIPLE = 12,

pAliasing [PI_ALIASING_LENGTH]

The alias name of the variable with a maximum name length of PI_ALIASING_LENGTH characters complemented with the IO name.

ucPadding

Alignment padding of data. Unsigned +ve character values.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-29 describes the function for MMC_GetPIVarInfo.

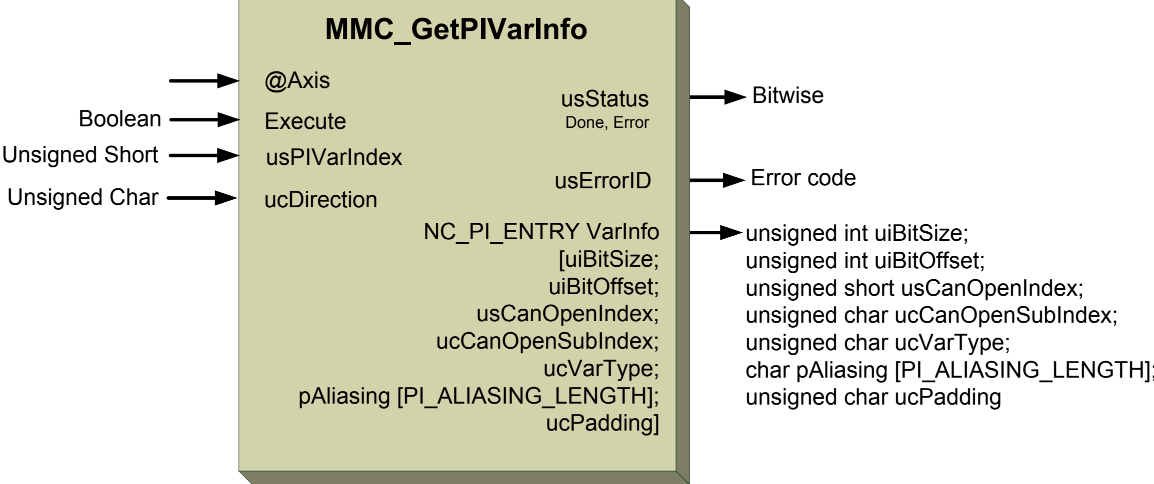


Figure 9-29: MMC_GetPIVarInfo function

9.7.27.2 Function Block Code Example

```
{
    MMC_GETPIVARINFO_IN GetIn;
    MMC_GETPIVARINFO_OUT GetOut;
    unsigned short usID = 0;

    printf("which axis?\n");
    scanf("%hu",&usID);

    printf("0 - input or 1 - output?\n");
    scanf("%hu",&usAns);
    if(usAns == 0)
    {
        GetIn.ucDirection = ePI_INPUT;
        printf("reading INPUTs\n");
    }
    else
    {
        GetIn.ucDirection = ePI_OUTPUT;
        printf("reading OUTPUTs\n");
    }

    printf("how many variables are there?\n");
    scanf("%hu",&usAns);

    for(i = 0; i < usAns; i++)
    {
        NC_PI_ENTRY *p = NULL;

        GetIn.usPIVarIndex = i;
        rc = MMC_GetPIVarInfo(ConnHndl,hAxisRef[usID],&GetIn,&GetOut);
        if(rc)
        {
printf("MMC_GetPIVarInfo error %d\n",GetOut.sErrorID);
goto lbl_motor_off;
        }

        p = &GetOut.VarInfo;

printf("var %d: bit size %d, bitOffset %d, canOpen index 0x%x, sub index %x, aliasing %s, Var Type %d\n",i,p->uiBitSize,p->uiBitOffset,p->usCanOpenIndex,p->ucCanOpenSubIndex,p->pAliasing,p->ucVarType);
    }
}
```

9.7.28 MMC_GetPIVarInfoByAlias

This function returns the detailed number of mapped Processing Image variables, reading the variable alias as a key.

```
MMC_LIB_API int MMC_GetPIVarInfoBy Alias(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETPIVARINFO_INBYALIAS* pInParam,
OUT MMC_GETPIVARINFO_OUTBYALIAS* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETPIVARINFOBYALIAS_IN** input data structure using the MMC_ReadPIVarInfoByAlias function.

pOutParam

Points to the **MMC_GETPIVARINFOBYALIAS_OUT** output structure receiving information as a result of calling the MMC_ReadPIVarInfoByAlias function.

Remarks

The Alias string consists of (I for inputs or O for outputs).objectNum.Subindex format, whereas the Display Name is the parameter added to the XML file and relevant EASII windows. This parameter is not limited in size, and is downloaded as part of the cfg.xml file, but is not parsed by the Maestro.

Scope

All

MMC_GETPIVARINFOBYALIAS_IN Structure

```
typedef struct mmc_getpivarinfobyalias_in{
char pAliasing[PI_ALIASING_LENGTH];
} MMC_GETPIVARINFOBYALIAS_IN;
```

Parameters

pAliasing[PI_ALIASING_LENGTH]

The alias name of the variable with a maximum name length of PI_ALIASING_LENGTH characters.

MMC_GETPIVARINFOBYALIAS_OUT Structure

```
typedef struct mmc_getpivarinfobyalias_out{
NC_PI_INFO_BY_ALIAS VarInfo;
unsigned short usStatus;
short usErrorID;
} MMC_GETPIVARINFOBYALIAS_OUT;
```

Parameters

NC_PI_INFO_BY_ALIAS VarInfo

This is the detailed information on the PI variable. The CanOpen index and sub-index define the PI variable object in the slave, whereas Bit size and offset define its location in the cyclic EtherCAT frame.

```
typedef struct piinfobyalias{
unsigned int uiBitSize;
unsigned int uiBitOffset;
unsigned short usCanOpenIndex;
unsigned short usPIVarOffset;
unsigned char ucCanOpenSubIndex;
unsigned char ucVarType;
} NC_PI_INFO_BY_ALIAS;
```

uiBitSize

Bit size is the number of bits forming the variable. Any +ve integer values

uiBitOffset

Bit offset is the offset of the first bit in the EtherCAT frame. Any +ve integer values.

usCanOpenIndex

CANopen index. 2 Byte +ve values

usPIVarOffset

Offset of the PI variable. 2 Byte +ve values

ucCanOpenSubIndex

CANopen subindex. +ve character values

ucVarType

Variable type according to the following enumerator values:

ePI_BOOL	= 0,
ePI_SIGNED_CHAR	= 1,
ePI_UNSIGNED_CHAR	= 2,
ePI_SIGNED_SHORT	= 3,
ePI_UNSIGNED_SHORT	= 4,
ePI_SIGNED_INT	= 5,
ePI_UNSIGNED_INT	= 6,
ePI_SIGNED_LONG_LONG	= 7,
ePI_UNSIGNED_LONG_LONG	= 8,
ePI_FLOAT	= 9,
ePI_DOUBLE	= 10,
ePI_BITWISE	= 11,
ePI_8MULTIPLE	= 12,

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-30 describes the C++ function for MMC_GetPIVarInfoByAlias.

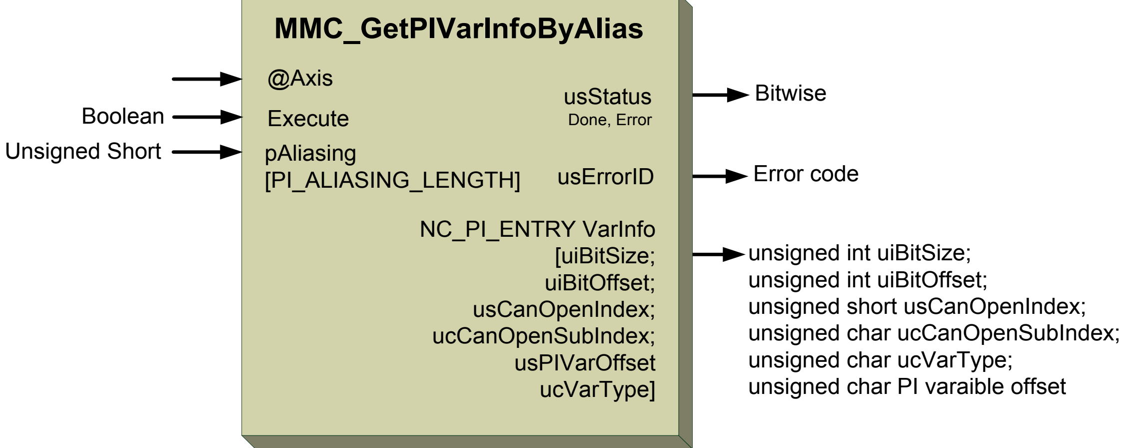


Figure 9-30: MMC_GetPIVarInfoByAlias function

9.7.28.2 Function Block Code C++ Example 1

```
MMC_GETPIVARINFOBYALIAS_IN GetIn;
MMC_GETPIVARINFOBYALIAS_OUT GetOut;

      printf("what is the string\n");
      scanf("%s",GetIn.pAliasing);

  rc = MMC_GetPIVarInfoByAlias(ConnHndl,hAxisRef[usSelect],&GetIn,&GetOut);
      if(rc)
      {
  printf("MMC_GetPIVarInfoByAlias error %d\n",GetOut.usErrorID);
                                    return -1;
      }

      printf("the requested variable data: direction = %hhu, var offset = %d, index =
0x%X(%d), sub index = %hhu, bit size = %d, bit offset = %d, type =
%hhu\n",GetOut.VarInfo.ucDirection,GetOut.VarInfo.usPIVarOffset,GetOut.VarInfo.usCanOpenIndex,
GetOut.VarInfo.usCanOpenIndex,GetOut.VarInfo.ucCanOpenSubIndex,GetOut.VarInfo.uiBitSize,GetOut
.VarInfo.uiBitOffset,GetOut.VarInfo.ucVarType);
```

9.7.28.3 Function Block Code Example 2

```
MMC_GETPIVARINFOBYALIAS_IN GetIn;
MMC_GETPIVARINFOBYALIAS_OUT GetOut;
MMC_WRITEPIVARSHORT_IN WriteIn;
MMC_WRITEPIVARSHORT_OUT WriteOut;

strcpy(GetIn.pAliasing,"CW");
GetIn.ucDirection = ePI_OUTPUT;

// acquire the PI var offset by the alias field
MMC_GetPIVarInfoByAlias(uiConnHndl, usAxisRef, &GetIn, &GetOut);

WriteIn.usIndex = GetOut.VarInfo.usPIVarOffset;
WriteIn.sData = 6;

// write the data to the desired variable

MMC_WritePIVarShort(uiConnHndl,usAxisRef,&WriteIn,&WriteOut);
```

9.7.28.4 Function Block Code IEC XML Example 2

From within the EASII application, import the XML program ..\NetHelp\Software Manual Pdfs\PIInfoByAlias.xml to the IEC projects on the IEC Motion Programming page, and run the program.

Figure 9-31: IEC Function MMC_GetPIInfoByAlias

9.7.29 MMC_GetPIVarsRangeInfo

This function allows the user to upload multiple PI variables information.

```
MMC_LIB_API int MMC_GetPIVarsRangeInfo(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETPIVARSRANGEINFO_IN* pInParam,
OUT MMC_GETPIVARSRANGEINFO_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_General_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETPIVARSRANGEINFO_IN** input data structure using the MMC_GetPIVarsRangeInfo function.

pOutParam

Points to the **MMC_GETPIVARSRANGEINFO_OUT** output structure receiving information as a result of calling the MMC_GetPIVarsRangeInfo function.

Remarks

The PI variables chapter contains a single function interface to send one PI variable info. However, this function allows uploading of a large number of PI variables info (or the entire PI inputs\outputs of a slave).

Scope

All

MMC_GETPIVARSRANGEINFO_IN Structure

```
typedef struct mmc_getvarsrangeinfo_in{
unsigned short usFirstIndex;
unsigned short usLastIndex;
unsigned char ucDirection;
} MMC_GETPIVARSRANGEINFO_IN;
```

Parameters

usFirstIndex

First index of the PI variable. Any +ve value.

usLastIndex

Last index of the PI variable. Any +ve value.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

MMC_GETPIVARSRANGEINFO_OUT Structure

```
typedef struct mmc_getvarsrangeinfo_out{
NC_PI_ENTRY VarInfo[MAX_PI_VARIABLES_NUM];
unsigned short usStatus;
short usErrorID;
} MMC_GETPIVARSRANGEINFO_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

NC_PI_ENTRY VarInfo[MAX_PI_VARIABLES_NUM]

The user can receive a maximum total of `MAX_PI_VARIABLES_NUM = 50` PI variables structures.

The PIentry sub-structure contains the details of the requested PI variables.

```
typedef struct pientry{
unsigned int uiBitSize;
unsigned int uiBitOffset;
unsigned short usCanOpenIndex;
unsigned char ucCanOpenSubIndex;
unsigned char ucVarType;
char pAliasing[PI_ALIASING_LENGTH];
} NC_PI_ENTRY;
```

uiBitSize

The number of bits this variable occupies. Any +ve integer values

uiBitOffset

The Offset of the variable in the EtherCAT frame. Any +ve integer values.

usCanOpenIndex

CANopen index. 2 Byte +ve values

ucCanOpenSubIndex

CANopen subindex. +ve character values

ucVarType

Variable type according to the following enumerator values:

ePI_BOOL	= 0,
ePI_SIGNED_CHAR	= 1,
ePI_UNSIGNED_CHAR	= 2,
ePI_SIGNED_SHORT	= 3,
ePI_UNSIGNED_SHORT	= 4,
ePI_SIGNED_INT	= 5,
ePI_UNSIGNED_INT	= 6,
ePI_SIGNED_LONG_LONG	= 7,
ePI_UNSIGNED_LONG_LONG	= 8,
ePI_FLOAT	= 9,
ePI_DOUBLE	= 10,
ePI_BITWISE	= 11,
ePI_8MULTIPLE	= 12,

pAliasing [PI_ALIASING_LENGTH]

The alias name of the variable with a maximum name length of PI_ALIASING_LENGTH characters as defined in the EASII EtherCAT configurator..

Figure 9-32 describes the function for MMC_GetPIVarsRangeInfo.

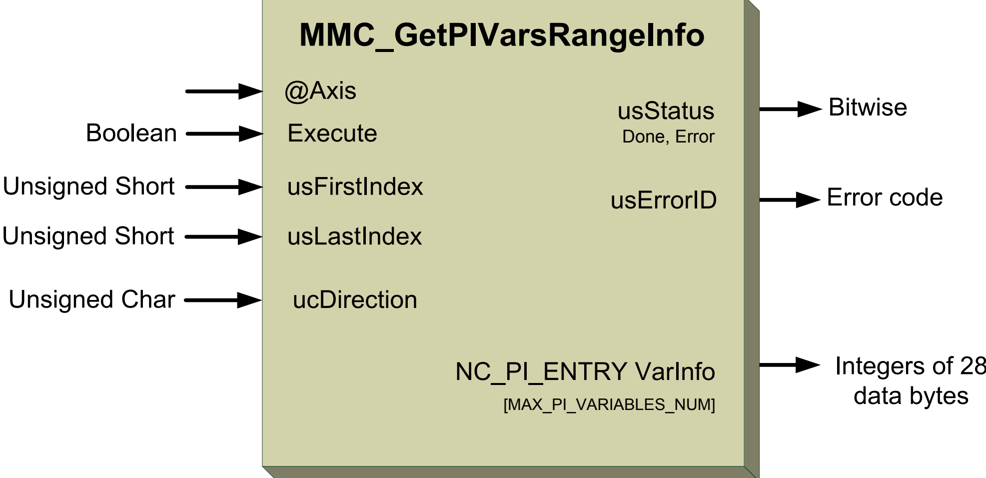


Figure 9-32: MMC_GetPIVarsRangeInfo function

9.7.29.2 Function Block Code Example

```
MMC_GETPIVARSRANGEINFO_IN GetIn;
MMC_GETPIVARSRANGEINFO_OUT GetOut;
int iMaxNum = 0;

printf("which Axis?\n");
scanf("%hu",&usSelect);

                        GetIn.ucDirection = ePI_INPUT;

printf("enter first index\n");
scanf("%hu",&GetIn.usFirstIndex);

printf("enter last index\n");
scanf("%hu",&GetIn.usLastIndex);

iMaxNum = GetIn.usLastIndex - GetIn.usFirstIndex;

rc = MMC_GetPIVarsRangeInfo(ConnHndl,hAxisRef[usSelect],&GetIn,&GetOut);
if(rc)
{
      printf("MMC_GetPIVarsRangeInfo error %d\n",GetOut.usErrorID);
      return -1;
}

for(i = 0; i <= iMaxNum; i++)
{
      printf("var %d info: index = 0x%X, subindex = %d, aliasing = %s, varType = %hhu, bit
offset = %u, bit size =
%hhu\n",i,GetOut.pVarInfo[i].usCanOpenIndex,GetOut.pVarInfo[i].ucCanOpenSubIndex,
GetOut.pVarInfo[i].pAliasing, GetOut.pVarInfo[i].ucVarType,
GetOut.pVarInfo[i].uiBitOffset,GetOut.pVarInfo[i].uiBitSize);
```

9.8 PI Bulk Read User Functions

In the general Bulk Read User Functions, the user function forces the user to read all the signals for all the nodes. If some signals were not mapped to some of the nodes, the user must read these variables, producing a non mapped variable value of zero and a waste of space that can be configured for another variable.

When trying to access a non-existent PI variable the user receives an error, forcing the user to map all the necessary variables for each node. In some cases this is impossible, e.g. when using a network with an IO device, the user cannot Bulk read signals from the IO and from the drives. Two different reads of data are therefore necessary. In order to enable Bulk Reading General PI Variables, a new function is defined i.e. MMC_ConfigureBulkReadPI.

The following table lists the new PI Bulk Read User functions.

PI functions
MMC_ConfigureBulkReadPI
MMC_PerformBulkReadCmdPI

9.8.1 MMC_ConfigureBulkReadPI

This function allows the user to configure PI bulk read of parameters.

```
int MMC_ConfigureBulkReadPI(
MMC_CONNECT_HNDL hConn
MMC_PICONFIGBULKREAD_IN *pInParam
MMC_PICONFIGBULKREAD_OUT *pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_PICONFIGBULKREAD_IN** input data structure using the MMC_ConfigureBulkReadPI function.

pOutParam

Points to the **MMC_PICONFIGBULKREAD_OUT** output structure receiving information as a result of calling the MMC_ConfigureBulkReadPI function.

Remarks

It is important note that PI variables in the same index but in different axes might be different in size (and even do not exist in some axes), depending on the network configuration and the user selection at the configuration stage. Therefore the PI bulk read API will perform validations against every axis to verify that all PI variables do exist and that the space required for reading all the data is not larger than the possible space. If any error occurs, the user will be notified.

When executing a PI operation on BOOL variables it is necessary to extract the relevant bit. Therefore, when reading BOOL PI variables according to varoffset : value & (1 << n)), set the value – return data from bulkread and n – varoffset according to the BOOL outputs.

Scope

All the variables up to 64 bits are supported with the exception of ePI_8Multiple and ePI_BITWISE.

MMC_PICONFIGBULKREAD_IN Structure

```
typedef struct{
PI_BULKREAD_ENTRY pVarsArray [NC_MAX_PI_BULK_READ_VARIABLES];
NC_BULKREAD_CONFIG_PI_ENUM eConfiguration;
} MMC_PICONFIGBULKREAD_IN;
```

Parameters

PI_BULKREAD_ENTRY

Defines what PI variables will be read. An entry with the varaible ucPiDirection equal to ePI_NONE will signal that no more variables are to be read. If no such value is located (and not all entries are valid) a junk value might be considered as a signal requested.

```
typedef struct{
unsigned short usIndex;
unsigned short usAxisRef;
unsigned char ucPiDirection;
unsigned char pPadding[3];
}PI_BULKREAD_ENTRY;
```

usIndex

The PI variable offset. This is the required index to be read from the process image. Any +ve integer values.

usAxisRef

The axis reference the signal is related to.

ucPiDirection

Indicates whether this is an input or an output. This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0

ePI_OUTPUT = 1

ePI_NONE = 3 signals that no more valid entries are given

pPadding[3]

pVarsArray [NC_MAX_PI_BULK_READ_VARIABLES]

The array of *pVarsArray [NC_MAX_PI_BULK_READ_VARIABLES]* defined by *[NC_MAX_PI_BULK_READ_VARIABLES]* represents an array of PI parameters stored PI table entries to be retrieved from the Maestro, with a maximal data size of (175 PI variables) since it is in compliance to the maximal TCP packet size. *ePI_NONE* indicates the last array entry.

Refer to example in section **9.8.2.2 Function Block Code Example**.

NC_BULKREAD_CONFIG_PI_ENUM eConfiguration

New enumerators are used for PI variables.

Defines the reading source. eBULKREAD_CONFIG_PI_3, eBULKREAD_CONFIG_PI_4 are reserved to the EASII application. NC_BULKREAD_CONFIG_PI_ENUM defines the following values:

eBULKREAD_CONFIG_PI_1 = 0,

eBULKREAD_CONFIG_PI_2 = 1,

eBULKREAD_CONFIG_PI_3 = 2,

eBULKREAD_CONFIG_PI_4 = 3,

eBULKREAD_CONFIG_PI_MAX,

MMC_PICONFIGBULKREAD_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
} MMC_PICONFIGBULKREAD_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-33 describes the function for MMC_ConfigureBulkReadPI.

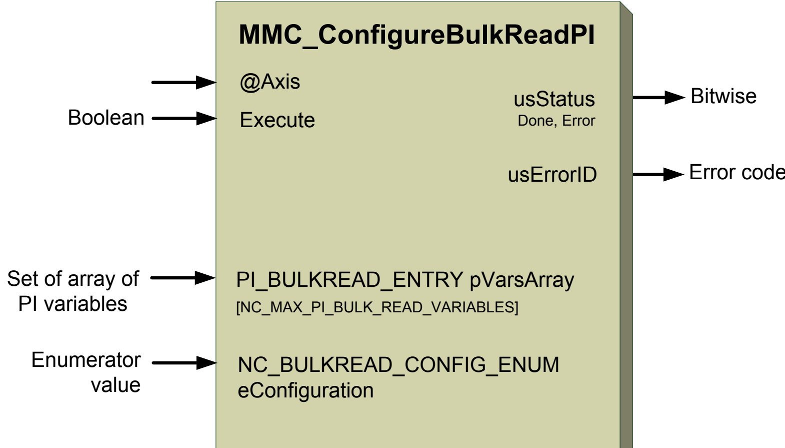


Figure 9-33: MMC_ConfigureBulkReadPI function

9.8.1.2 Function Block Code Example

See the example when both MMC_ConfigureBulkReadPI and MMC_PerformBulkReadCmdPI are in operation in section **9.8.2.2**.

9.8.2 MMC_PerformBulkReadCmdPI

This function allows the user to perform PI bulk read of parameters.

```
MMC_LIB_API int MMC_PerformBulkReadCmdPI(
IN MMC_CONNECT_HNDL hConn,
IN MMC_PERFORMBULKREADPI_IN* pInParam,
OUT MMC_PERFORMBULKREADPI_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_General_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_PERFORMBULKREADPI_IN** input data structure using the MMC_ConfigureBulkReadCmdPI function.

pOutParam

Points to the **MMC_PERFORMBULKREADPI_OUT** output structure receiving information as a result of calling the MMC_ConfigureBulkReadCmdPI function.

Remarks

It is important note that PI variables in the same index but in different axes might be different in size (and even do not exist in some axes), depending on the network configuration and the user selection at the configuration stage. Therefore the PI bulk read function will perform validations against every axis to verify that all PI variables do exist and that the space required for reading all the data is not larger than the possible space. If any error occurs, the user will be notified.

Scope

MMC_PerformBulkReadCmdPI will not operate unless the function MMC_ConfigBulkReadPI is initiated. In addition, this function will only operate on PI varaiables that were configured by the function MMC_ConfigureBulkReadPI.

All the variables up to 64 bits are supported with the exception of ePI_8Multiple and ePI_BITWISE.

64 bit variables occupy two consecutive entries in the MMC_PerformBulkReadCmdPI function.

MMC_PERFORMBULKREADPI_IN Structure

```
typedef struct mmc_performbulkreadpi_in{
NC_BULKREAD_CONFIG_PI_ENUM eConfiguration;
} MMC_PERFORMBULKREADPI_IN;
```

Parameters

NC_BULKREAD_CONFIG_PI_ENUM eConfiguration

New enumerators are used for PI variables.

Defines the reading source. eBULKREAD_CONFIG_PI_3, eBULKREAD_CONFIG_PI_4 is reserved to the EASII application. NC_BULKREAD_CONFIG_PI_ENUM defines the following values:

eBULKREAD_CONFIG_PI_1 = 0,

eBULKREAD_CONFIG_PI_2 = 1,

eBULKREAD_CONFIG_PI_3 = 2,

eBULKREAD_CONFIG_PI_4 = 3,

eBULKREAD_CONFIG_PI_MAX,

MMC_PERFORMBULKREADPI_OUT Structure

```
typedef struct mmc_performbulkreadpi_out{
unsigned long ulOutBuf[NC_MAX_BULK_READ_READABLE_PACKET_SIZE];
unsigned short usStatus;
short usErrorID;
} MMC_PERFORMBULKREADPI_OUT;
```

Parameters

ulOutBuf[NC_MAX_BULK_READ_READABLE_PACKET_SIZE]

Defines the output buffer read data with a maximum [NC_MAX_BULK_READ_READABLE_PACKET_SIZE] array size of 350.

Only applies to

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 9-34 describes the function for MMC_PerformBulkReadCmdPI.

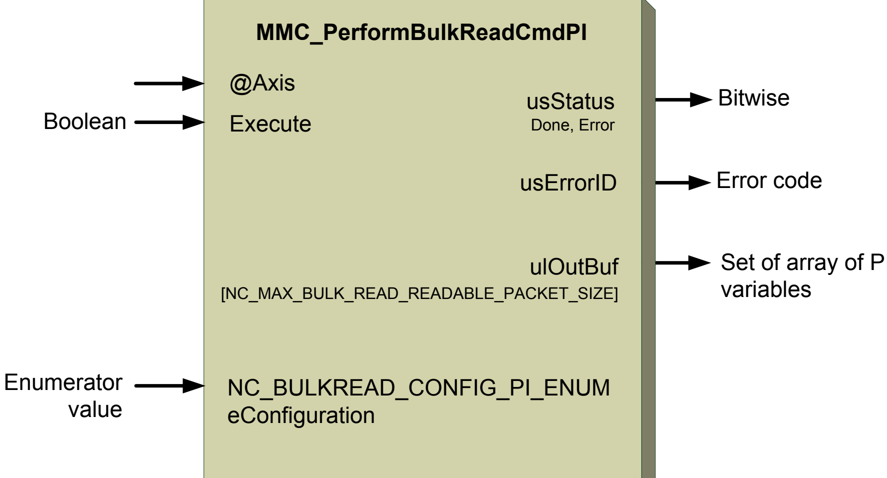


Figure 9-34: MMC_PerformBulkReadCmdPI function

9.8.2.2 Function Block Code Example

```
MMC_CONFIGBULKREADPI_IN ConfigIn;
MMC_CONFIGBULKREADPI_OUT ConfigOut;
MMC_PERFORMBULKREADPI_IN PerformIn;
MMC_PERFORMBULKREADPI_OUT PerformOut;

// char from 5th axis
ConfigIn.eConfiguration = eBULKREAD_CONFIG_PI_1;
ConfigIn.pVarsArray[0].ucPiDirection = ePI_OUTPUT;
ConfigIn.pVarsArray[0].usAxisRef = hAxisRef[5];
ConfigIn.pVarsArray[0].usIndex = 3;

// float from 4th axis
ConfigIn.pVarsArray[1].ucPiDirection = ePI_OUTPUT;
ConfigIn.pVarsArray[1].usAxisRef = hAxisRef[4];
ConfigIn.pVarsArray[1].usIndex = 3;

                       // marker for the end
ConfigIn.pVarsArray[2].ucPiDirection = ePI_NONE;

rc = MMC_ConfigBulkReadCmdPI(ConnHndl,&ConfigIn,&ConfigOut);
if(rc)
{
      printf("MMC_ConfigBulkReadCmdPI error %d\n",ConfigOut.usErrorID);
      return -1;
}

PerformIn.eConfiguration = eBULKREAD_CONFIG_PI_1;

rc = MMC_PerformBulkReadCmdPI(ConnHndl,&PerformIn,&PerformOut);
if(rc)
{
      printf("MMC_PerformBulkReadCmdPI error %d\n",PerformOut.usErrorID);
      return -1;
}

printf("the char value is %hhu\n",(unsigned char)PerformOut.ulOutBuf[0]);
printf("the float value is %f\n",(float)*(float *)&PerformOut.ulOutBuf[1]);
```

9.9 PI Functions and Implementation Examples

9.9.1 C++ Example

```
// for variables larger than 32 bits
case 4:
{
     // this structure shouldn't be used by users, they should only use 
MMC_READPIVARDOUBLE_IN, MMC_READPIVARLONGLONG_IN etc.
     MMC_READLARGEPIVAR_IN ReadIn;
     MMC_GETPIVARINFO_IN GetIn;
     MMC_GETPIVARINFO_OUT GetOut;
     unsigned short usAns = 5;
     unsigned char ucID = 0, ucVarType = 0;
     unsigned char ucSize = 0;

     printf("which axis?\n");
     scanf("%hhu",&ucID);

     printf("what variable to read? 0 - exit, 1 - input, 2 - output\n");
     scanf("%hu",&usAns);
     while(usAns)
     {
          if(usAns == 1)
          {
ReadIn.ucDirection = ePI_INPUT;
GetIn.ucDirection = ePI_INPUT;
printf("reading INPUT\n");
          }
          else
          {
ReadIn.ucDirection = ePI_OUTPUT;
GetIn.ucDirection = ePI_OUTPUT;
printf("reading OUTPUT\n");
          }

          printf("what is the variable offset\n");
          scanf("%hu",&usAns);
          ReadIn.usIndex = usAns;
          GetIn.usPIVarIndex = usAns;

          rc = MMC_GetPIVarInfo(ConnHndl,hAxisRef[ucID],&GetIn,&GetOut);
          if(rc)
          {
printf("GetPIVarInfo error %d, var offset %d\n",GetOut.sErrorID,usAns);
          }
          else
          {
ucSize = GetOut.VarInfo.uiBitSize;
ucVarType = GetOut.VarInfo.ucVarType;

if(ucSize == LONG_LONG_BITSIZE)
{
if(ucVarType == ePI_DOUBLE)
{
     MMC_READPIVARDOUBLE_IN *pReadIn = (MMC_READPIVARDOUBLE_IN *)&ReadIn;
     MMC_READPIVARDOUBLE_OUT ReadOut;

     rc = MMC_ReadPIVarDouble(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
     if(rc)
     {
          short sErr = ReadOut.sErrorID;

          printf("MMC_ReadLargePIVarDouble error %d\n",sErr);
          PrintError(sErr);
          break;
     }

     printf("read the DOUBLE value %lf\n",ReadOut.dbData);
}

// check signess of the variable (signed\unsigned)
else if(ucVarType == ePI_UNSIGNED_LONG_LONG)
{
     MMC_READPIVARULONGLONG_IN *pReadIn = (MMC_READPIVARULONGLONG_IN *)&ReadIn;
     MMC_READPIVARULONGLONG_OUT ReadOut;
```

```
    rc = MMC_ReadPIVarULongLong(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
    if(rc)
    {
        short sErr = ReadOut.sErrorID;

        printf("MMC_ReadLargePIVarUnsignedLongLong error %d\n",sErr);
        PrintError(sErr);
        break;
    }

    printf("read the unsigned LONG LONG value %lld\n",ReadOut.ullData);
}
else
{
    MMC_READPIVARLONGLONG_IN *pReadIn = (MMC_READPIVARLONGLONG_IN *)&ReadIn;
    MMC_READPIVARLONGLONG_OUT ReadOut;

    rc = MMC_ReadPIVarLongLong(ConnHndl,hAxisRef[ucID],pReadIn,&ReadOut);
    if(rc)
    {
        short sErr = ReadOut.sErrorID;

        printf("MMC_ReadLargePIVarSignedLongLong error %d\n",sErr);
        PrintError(sErr);
        break;
    }

    printf("read the signed LONG LONG value %lld\n",ReadOut.llData);
}
}

// non standard type should be read as raw data
else
{
MMC_READLARGEPIVARRAW_IN ReadRawIn;
MMC_READLARGEPIVARRAW_OUT ReadOut;
unsigned char ucBitStart =  GetOut.VarInfo.uiBitOffset & 0x07;
unsigned int uiCount  = ((GetOut.VarInfo.uiBitSize  + ucBitStart - 1) >> 3) + 1;
unsigned int i = 0;

ReadRawIn.ucDirection = ReadIn.ucDirection;
ReadRawIn.usIndex = ReadIn.usIndex;
ReadRawIn.usByteLength = uiCount;

rc = MMC_ReadLargePIVarRaw(ConnHndl,hAxisRef[ucID],&ReadRawIn,&ReadOut);
if(rc)
{
    short sErr = ReadOut.sErrorID;

    printf("MMC_ReadLargePIVarRaw error %d\n",sErr);
    PrintError(sErr);
    break;
}

printf("THE data is 0x");
for(i = 0; i < uiCount; i++)
{
    printf("[%02X]",ReadOut.pRawData[i]);
}
printf("\n");
}
        }

        printf("what variable to read? 0 - exit, 1 - input, 2 - output\n");
        scanf("%hu",&usAns);
    }
}
break;

case 5:
{
    MMC_GETPIVARINFO_IN GetIn;
    MMC_GETPIVARINFO_OUT GetOut;
    unsigned short usAns = 5;
    unsigned char ucID = 0;

    printf("which axis?\n");
    scanf("%hhu",&ucID);
```

```
    printf("what is the variable offset (29 - exit)\n");
    scanf("%hu",&usAns);

    while(usAns != 29)
    {
        GetIn.ucDirection = ePI_OUTPUT;
        GetIn.usPIVarIndex = usAns;
        rc = MMC_GetPIVarInfo(ConnHndl,hAxisRef[ucID],&GetIn,&GetOut);
        if(rc)
        {
printf("GetPIVarInfo error %d\n",GetOut.sErrorID);
        }

        // check signess of the variable (signed\unsigned)
        if(GetOut.VarInfo.ucVarType == ePI_UNSIGNED_LONG_LONG)
        {
unsigned long long ullVal = 0;
MMC_WRITEPIVARULONGLONG_IN WriteIn;
MMC_WRITEPIVARULONGLONG_OUT WriteOut;

printf("what is unsigned long long the value?\n");
scanf("%lld",&ullVal);

WriteIn.usIndex = usAns;
WriteIn.ullData = ullVal;

rc = MMC_WritePIVarULongLong(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
if(rc)
{
short sErr = WriteOut.sErrorID;

printf("MMC_WriteLargePIVarUnsignedLongLong error %d\n",sErr);
PrintError(sErr);
break;
}
        }

        // check signess of the variable (signed\unsigned)
        else if(GetOut.VarInfo.ucVarType == ePI_SIGNED_LONG_LONG)
        {
long long llVal = 0;
MMC_WRITEPIVARLONGLONG_IN WriteIn;
MMC_WRITEPIVARLONGLONG_OUT WriteOut;

printf("what is the signed long long value?\n");
scanf("%lld",&llVal);

WriteIn.usIndex = usAns;
WriteIn.llData = llVal;

rc = MMC_WritePIVarLongLong(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
if(rc)
{
short sErr = WriteOut.sErrorID;

printf("MMC_WriteLargePIVarSignedLongLong error %d\n",sErr);
PrintError(sErr);
break;
}
        }
        else if(GetOut.VarInfo.ucVarType == ePI_DOUBLE)
        {
double dbVal = 0;
MMC_WRITEPIVARDOUBLE_IN WriteIn;
MMC_WRITEPIVARDOUBLE_OUT WriteOut;

printf("what is the double value?\n");
scanf("%lf",&dbVal);

WriteIn.usIndex = usAns;
WriteIn.dbVal = dbVal;

rc = MMC_WritePIVarDouble(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
if(rc)
{
short sErr = WriteOut.sErrorID;

printf("MMC_WriteLargePIVarDouble error %d\n",sErr);
```

```
PrintError(sErr);
break;
}
          }

          // non standard type with bit size which is a multiple of 8
          else if(GetOut.VarInfo.ucVarType == ePI_8MULTIPLE)
          {

MMC_WRITELARGEPIVARRAW_IN WriteIn;
MMC_WRITELARGEPIVARRAW_OUT WriteOut;
unsigned char *pRawData = NULL;
unsigned char ucBitStart =  GetOut.VarInfo.uiBitOffset & 0x07;
unsigned int uiCount  = ((GetOut.VarInfo.uiBitSize  + ucBitStart - 1) >> 3) + 1;

printf("writing to the first 5 bytes of the 8MULTIPLE variable -0x[1][2][3][4][5]\n");

WriteIn.usIndex = usAns;
pRawData = WriteIn.pRawData;
WriteIn.usByteLength = uiCount;

pRawData[0] = 0x1;
pRawData[1] = 0x2;
pRawData[2] = 0x3;
pRawData[3] = 0x4;
pRawData[4] = 0x5;

rc = MMC_WriteLargePIVarRaw(ConnHndl,hAxisRef[ucID],&WriteIn,&WriteOut);
          }

          printf("what is the variable offset (29 - exit)\n");
          scanf("%hu",&usAns);
     }

}
```

9.9.2 General PI Test IEC Example

From within the EASII application, import the XML program ..\NetHelp\Software Manual Pdfs\pitest.xml to the IEC projects on the IEC Motion Programming page, and run the program.

9.9.3 PI Full Example in C, and IEC with EtherCAT Configuration Settings

For the C project example

Within either the GDS, Visual Studio, or equivalent application open the PI Project .

For the IEC project example

From within the EASII application, import the XML program ..\NetHelp\Software Manual Pdfs\PI\IEC\PI_FBD.xml to the IEC projects on the IEC Motion Programming page, and run the program.

For the EtherCAT Configuration Settings

From within the EASII application, import the EtherCAT settings.

Chapter 10: Data Recording

Data recording is a powerful feature of the Maestro that allows the user to record internal controller variables, store them in local a temporary array, and upload them to a host computer using either one of the controller's communication channels.

This is an advanced API option. Care should be taken when using data recording. It should only be used when the Maestro parameters have been set and the servo drivers functioning correctly. Only use TCP/IP communication to perform the data recording.

The Maestro has the following data recording capabilities:

- Simultaneous recording of up to a maximum of 32 bit internal controller signals/variables
- Up to one million data recorded points. The user can select to record up to a maximum of 32 bit vectors with 31,250 sample points, or for single vectors, one million sample points, or any other combination
- Various recording vectors
- Advanced triggering options

In most situations where the variable is described as a char, short, int, float, etc., each variable can be recorded as a single vector, and therefore a maximum 32 bit variable is recorded. However, where the variable is a double, the vector memory space allocated requires two vectors, therefore only allowing a 16 bit variable to be recorded.

10.1 Triggering a Recording

The Maestro supports advanced trigger options that further enhance the data logging capabilities of the system, and provide a powerful tool for monitoring and debugging servo applications. In general, Trigger support refers to the ability of the controller to start a data recording process, and to condition the actual execution of the data logging based on a specific event. The user can select the event source, type, and condition, as well as to perform pre-trigger data logging (logging data prior to the trigger event).

The Maestro support numerous trigger event types, generally divided into the following groups:

Group	Explanation
Edge Type Triggers	Where a change in signal level around some threshold is detected.
Level Type Triggers	Where the signal level is detected (not necessarily a change).
Single Level or Window conditions	The trigger condition can be defined as a single level or a window (Min/Max).
Bitwise Masking of signals	The trigger condition can be defined by a user defined Bitwise mask of the requested signal.

10.2 Active Range Support

The fast upload data rate supported via the TCP/IP link, and double buffering capability, allows the host computer to display data recording in an active range, with all of the triggering options. There are four data recording types:

Type	Explanation
AUTO	The data recording is performed with no triggering, but is cyclic (the double buffering recording mechanism is used).
SINGLE	The data recording is performed with triggers, but is not cyclic. The trigger is searched for, once only, and then the recording ends.
NOTRIGGER	The data recording is performed with no triggering, and is not cyclic (The double buffering recording mechanism is not used).
NORMAL	The data recording is performed with triggering, and is cyclic (the double buffering recording mechanism is used).

10.3 Using Data Recording in the Maestro

The GMAS supports Data Recording using the following keywords:

Keywords	Variable	Command
Begin / Stop Data Recording command		MMC_BeginRecordingCmd
Data Recording Configuration Parameters:		MMC_BEGIN_RECORDING_IN
• Select Recorded variables parameter	uiRp	
• Select Recording Length parameter	uiRl	
• Select Recording GAP parameter	uiRg	
Report Recording Status command Rest Recording Index	 uiRr	MMC_RecStatusCmd
Uploading the array From/To/Buffered index	uiFrom uiTo uiBufIdx	MMC_UploadDataCmd
Triggering Options Recorder Trigger Status	 uiSr	
Data Recording Array		

10.3.1 Excluding Triggers

When normal data recording without triggers is used, the Recording Status variable uiRr is automatically initialized by the Begin Recording command, with the user defined Recording Length uiRr=uiRl (number of recorded points). During the recording process it is decremented, every gap (defined by uiRg) servo samples by 1 until it reaches 0 (Rr=0). At this point, the data recording is terminated and the data itself can be uploaded.

10.3.2 Including Triggers

The ability to perform data recording with Triggers Support general depends on the capability of the controller to initiate a data recording process, and to condition its actual execution and progress bases on user-defined events and conditions. For the Pre-Triggering phase, the controller supports data logging before the trigger event occurred. The GMAS supports pre-trigger buffer length from 0% to 100%.

The Arming (using variable uiSr) state indicates that the controller is in the Pre-Trigger phase, collecting data, and filling the recording buffer, to the size of pre-trigger size defined. When using triggers, the phase starts immediately with the Start Recording command. The variable uiRr is initially set to uiRl, and is decremented until the size of the Pre-Trigger is reached. For example, if uiRl =1000, and the Pre-Trigger is 25% - i.e. 250 points, uiRr will be decremented during the Pre-Trigger state to 750.

The condition whereby the system is waiting for an Inverse Trigger state is present only in Edge Trigger modes, and indicates that the controller is waiting for the opposite trigger condition, to operate the trigger condition itself. This is required for Edge or Change Detection. At this phase, data logging is continuously performed in the internal data recording buffers, but the actual recording is paused. The Recording Status variable uiRr is unchanged.

The delay for a Trigger Condition state is present in all Trigger modes, to indicate that the controller is waiting for the trigger event condition. At this phase, data logging is continuously performed in the internal data recording buffers, but the actual recording is paused. Similarly, the Recording Status variable uiRr is unchanged.

When the Triggered and Recording state is invoked, the trigger event is detected, and normal data recoding continues. The End of Recording process always completes when the Recording Status Variable reach Zero value uiRr =0.

When using normal data recording without triggers, the Start of Recording (Begin Recording Command) is referred to the 0 time-base in the output data buffer shown on the graphic plots. When operating with triggers, the 0 time-base will depend on the trigger Event condition. The actual Start of Recording time is no longer relevant, as the trigger event condition can occur long after the Start Recording command was issued. In this case, the 0 time-base of the Output Buffers refers to the point that is the Pre-Trigger time before the Trigger Event. For example, if a 1 second recording process is initiated, with a pre-trigger of 50%, the Pre-Trigger point is always (by definition) at 0.5 second after start.

10.4 Recording Definitions and Parameters

The following is a list of recording enumerator definitions and parameters.

10.4.1 Recording Data Signals Bitmask Definitions

The Recording Data Signal Bitmask variable uiRc, and ulRc create a mask using the following memory buffers to store the input parameters.

Bit Mask Parameter	Memory used
MC_REC_TRIGGER_TYPE_MASK	0x0000ffff
MC_REC_TRIGGER_PARAM_MASK	0xffff0000
MC_REC_TRIGGERG_STATE_MASK	0x00ff
MC_REC_BUF_STATE_MASK	0xff00
MC_SCOPE_BITS_NONE_BUF_READY	0x000
MC_SCOPE_BITS_BUFFER1_READY	0x100
MC_SCOPE_BITS_BUFFER2_READY	0x200
MC_NORMAL_TRIGGER	0x10000
MC_AUTO_TRIGGER	0x20000
MC_SINGLE_TRIGGER	0x30000

10.4.2 Recording Parameters

The following table lists the Recording Parameters Rp(i).

Recording Parameters	ID
TG_RECORDING_SPARE	0
TG_RECORDING_TRIGGER_VALUE	1
TG_RECORDING_PRE_TRIGGER_LENGTH	2
TG_RECORDING_TRIGGER_TYPE	3
TG_RECORDING_TRIGGER_LEVEL_1	4
TG_RECORDING_TRIGGER_LEVEL_2	5
TG_RECORDING_TRIGGER_POLARITY	6
TG_RECORDING_TRIGGER_IN_MASK	7

10.4.3 Recording Signal Parameters

The following parameters are recording signal variables and their IDs.

Recording Signal Variable	ID
NC_REC_DESIRED_POS_LOW_PARAM	0
NC_REC_DESIRED_POS_HIGH_PARAM	1
NC_REC_DESIRED_VEL_LOW_PARAM	2
NC_REC_DESIRED_VEL_HIGH_PARAM	3
NC_REC_GROUP_VEL_LOW_PARAM	4
NC_REC_GROUP_VEL_HIGH_PARAM	5
NC_REC_GROUP_AC_LOW_PARAM	6
NC_REC_GROUP_AC_HIGH_PARAM	7
NC_REC_GROUP_DC_LOW_PARAM	8
NC_REC_GROUP_DC_HIGH_PARAM	9
NC_REC_GROUP_AC_DC_LOW_PARAM	10
NC_REC_GROUP_AC_DC_HIGH_PARAM	11
NC_REC_GROUP_JERK_LOW_PARAM	12
NC_REC_GROUP_JERK_HIGH_PARAM	13
NC_REC_SMOOTH_FACTOR_AC_LOW_PARAM	14
NC_REC_SMOOTH_FACTOR_AC_HIGH_PARAM	15
NC_REC_SMOOTH_FACTOR_DC_LOW_PARAM	16
NC_REC_SMOOTH_FACTOR_DC_HIGH_PARAM	17
NC_REC_POS_INCR_LOW_PARAM	18
NC_REC_POS_INCR_HIGH_PARAM	19
NC_REC_CYCLE_CNT_PARAM	20
NC_REC_TARGET_POS_PARAM	21
NC_REC_TARGET_VEL_PARAM	22
NC_REC_F_POS_PARAM	23
NC_REC_F_VEL_PARAM	24
NC_REC_ACTUAL_POS_PARAM	25
NC_REC_ACTUAL_VEL_PARAM	26
NC_REC_AXIS_STATUS_PARAM	27
NC_REC_MAX_NUM_PARAM	28
NC_REC_ACTUAL_POS_PARAM	30

Recording Signal Variable	ID
NC_REC_ACTUAL_VEL_PARAM	31
NC_REC_AXIS_STATUS_PARAM	32
NC_REC_ACTUAL_TORQUE_PARAM	33
NC_REC_I_USER_1_PARAM	34
NC_REC_I_USER_AUX_1_PARAM	35
NC_REC_F_USER_1_PARAM	36
NC_REC_F_USER_AUX_1_PARAM	37
NC_REC_I_USER_2_PARAM	38
NC_REC_I_USER_AUX_2_PARAM	39
NC_REC_F_USER_2_PARAM	40
NC_REC_F_USER_AUX_2_PARAM	41
NC_REC_I_USER_3_PARAM	42
NC_REC_I_USER_AUX_3_PARAM	43
NC_REC_F_USER_3_PARAM	44
NC_REC_F_USER_AUX_3_PARAM	45
NC_REC_I_USER_4_PARAM	46
NC_REC_I_USER_AUX_4_PARAM	47
NC_REC_F_USER_4_PARAM	48
NC_REC_F_USER_AUX_4_PARAM	49
NC_REC_POS_FOLLOWING_ERR_PARAM	50
NC_REC_DIGITAL_INPUTS_PARAM	51
NC_REC_DIGITAL_OUTPUTS_PARAM	52
NC_REC_TRACKING_ERROR_LOW_PARAM	53
NC_REC_TRACKING_ERROR_HIGH_PARAM	54
NC_REC_ERROR_CORRECTION_POS_PARAM	55
NC_REC_ACTUAL_HW_POSITION_PARAM	56
NC_REC_CONTROL_WORD_PARAM	57
NC_REC_STATUS_WORD_PARAM	58
NC_REC_MOTION_MODE_PARAM	59
NC_REC_DI_LOW_PARAM	60
NC_REC_DI_HIGH_PARAM	61
NC_REC_DO_LOW_PARAM	62
NC_REC_DO_HIGH_PARAM	63

Recording Signal Variable	ID
NC_REC_AXIS_COMM_ERROR_PARAM	64
NC_REC_AXIS_LAST_EMCY_CODE_PARAM	65
NC_STATUS_REGISTER	66
NC_MCS_LIMIT_REGISTER	67
NC_REC_DESIRED_PCS_X_POS_LOW_PARAM	68
NC_REC_DESIRED_PCS_X_POS_HIGH_PARAM	69
NC_REC_DESIRED_PCS_Y_POS_LOW_PARAM	70
NC_REC_DESIRED_PCS_Y_POS_HIGH_PARAM	71
NC_REC_DESIRED_PCS_Z_POS_LOW_PARAM	72
NC_REC_DESIRED_PCS_Z_POS_HIGH_PARAM	73
NC_REC_DESIRED_PCS_U_POS_LOW_PARAM	74
NC_REC_DESIRED_PCS_U_POS_HIGH_PARAM	75
NC_REC_DESIRED_PCS_V_POS_LOW_PARAM	76
NC_REC_DESIRED_PCS_V_POS_HIGH_PARAM	77
NC_REC_DESIRED_PCS_W_POS_LOW_PARAM	78
NC_REC_DESIRED_PCS_W_POS_HIGH_PARAM	79
NC_REC_DESIRED_MCS_N1_POS_LOW_PARAM	80
NC_REC_DESIRED_MCS_N1_POS_HIGH_PARAM	81
NC_REC_DESIRED_MCS_N2_POS_LOW_PARAM	82
NC_REC_DESIRED_MCS_N2_POS_HIGH_PARAM	83
NC_REC_DESIRED_MCS_N3_POS_LOW_PARAM	84
NC_REC_DESIRED_MCS_N3_POS_HIGH_PARAM	85
NC_REC_DESIRED_MCS_N4_POS_LOW_PARAM	86
NC_REC_DESIRED_MCS_N4_POS_HIGH_PARAM	87
NC_REC_DESIRED_MCS_N5_POS_LOW_PARAM	88
NC_REC_DESIRED_MCS_N5_POS_HIGH_PARAM	89
NC_REC_DESIRED_MCS_N6_POS_LOW_PARAM	90
NC_REC_DESIRED_MCS_N6_POS_HIGH_PARAM	91
NC_REC_DESIRED_MCS_N7_POS_LOW_PARAM	92
NC_REC_DESIRED_MCS_N7_POS_HIGH_PARAM	93
NC_REC_DESIRED_MCS_N8_POS_LOW_PARAM	94
NC_REC_DESIRED_MCS_N8_POS_HIGH_PARAM	95
NC_REC_DESIRED_MCS_N9_POS_LOW_PARAM	96

Recording Signal Variable	ID
NC_REC_DESIRED_MCS_N9_POS_HIGH_PARAM	97
NC_REC_DESIRED_MCS_S_POS_LOW_PARAM	98
NC_REC_DESIRED_MCS_S_POS_HIGH_PARAM	99
NC_REC_DESIRED_PCS_X_VEL_LOW_PARAM	100
NC_REC_DESIRED_PCS_X_VEL_HIGH_PARAM	101
NC_REC_DESIRED_PCS_Y_VEL_LOW_PARAM	102
NC_REC_DESIRED_PCS_Y_VEL_HIGH_PARAM	103
NC_REC_DESIRED_PCS_Z_VEL_LOW_PARAM	104
NC_REC_DESIRED_PCS_Z_VEL_HIGH_PARAM	105
NC_REC_DESIRED_PCS_U_VEL_LOW_PARAM	106
NC_REC_DESIRED_PCS_U_VEL_HIGH_PARAM	107
NC_REC_DESIRED_PCS_V_VEL_LOW_PARAM	108
NC_REC_DESIRED_PCS_V_VEL_HIGH_PARAM	109
NC_REC_DESIRED_PCS_W_VEL_LOW_PARAM	110
NC_REC_DESIRED_PCS_W_VEL_HIGH_PARAM	111
NC_REC_DESIRED_MCS_N1_VEL_LOW_PARAM	112
NC_REC_DESIRED_MCS_N1_VEL_HIGH_PARAM	113
NC_REC_DESIRED_MCS_N2_VEL_LOW_PARAM	114
NC_REC_DESIRED_MCS_N2_VEL_HIGH_PARAM	115
NC_REC_DESIRED_MCS_N3_VEL_LOW_PARAM	116
NC_REC_DESIRED_MCS_N3_VEL_HIGH_PARAM	117
NC_REC_DESIRED_MCS_N4_VEL_LOW_PARAM	118
NC_REC_DESIRED_MCS_N4_VEL_HIGH_PARAM	119
NC_REC_DESIRED_MCS_N5_VEL_LOW_PARAM	120
NC_REC_DESIRED_MCS_N5_VEL_HIGH_PARAM	121
NC_REC_DESIRED_MCS_N6_VEL_LOW_PARAM	122
NC_REC_DESIRED_MCS_N6_VEL_HIGH_PARAM	123
NC_REC_DESIRED_MCS_N7_VEL_LOW_PARAM	124
NC_REC_DESIRED_MCS_N7_VEL_HIGH_PARAM	125
NC_REC_DESIRED_MCS_N8_VEL_LOW_PARAM	126
NC_REC_DESIRED_MCS_N8_VEL_HIGH_PARAM	127
NC_REC_DESIRED_MCS_N9_VEL_LOW_PARAM	128
NC_REC_DESIRED_MCS_N9_VEL_HIGH_PARAM	129

Recording Signal Variable	ID
NC_REC_DESIRED_MCS_S_VEL_LOW_PARAM	130
NC_REC_DESIRED_MCS_S_VEL_HIGH_PARAM	131
NC_REC_DESIRED_PCS_X_AC_DC_LOW_PARAM	132
NC_REC_DESIRED_PCS_X_AC_DC_HIGH_PARAM	133
NC_REC_DESIRED_PCS_Y_AC_DC_LOW_PARAM	134
NC_REC_DESIRED_PCS_Y_AC_DC_HIGH_PARAM	135
NC_REC_DESIRED_PCS_Z_AC_DC_LOW_PARAM	136
NC_REC_DESIRED_PCS_Z_AC_DC_HIGH_PARAM	137
NC_REC_DESIRED_PCS_U_AC_DC_LOW_PARAM	138
NC_REC_DESIRED_PCS_U_AC_DC_HIGH_PARAM	139
NC_REC_DESIRED_PCS_V_AC_DC_LOW_PARAM	140
NC_REC_DESIRED_PCS_V_AC_DC_HIGH_PARAM	141
NC_REC_DESIRED_PCS_W_AC_DC_LOW_PARAM	142
NC_REC_DESIRED_PCS_W_AC_DC_HIGH_PARAM	143
NC_REC_DESIRED_MCS_N1_AC_DC_LOW_PARAM	144
NC_REC_DESIRED_MCS_N1_AC_DC_HIGH_PARAM	145
NC_REC_DESIRED_MCS_N2_AC_DC_LOW_PARAM	146
NC_REC_DESIRED_MCS_N2_AC_DC_HIGH_PARAM	147
NC_REC_DESIRED_MCS_N3_AC_DC_LOW_PARAM	148
NC_REC_DESIRED_MCS_N3_AC_DC_HIGH_PARAM	149
NC_REC_DESIRED_MCS_N4_AC_DC_LOW_PARAM	150
NC_REC_DESIRED_MCS_N4_AC_DC_HIGH_PARAM	151
NC_REC_DESIRED_MCS_N5_AC_DC_LOW_PARAM	152
NC_REC_DESIRED_MCS_N5_AC_DC_HIGH_PARAM	153
NC_REC_DESIRED_MCS_N6_AC_DC_LOW_PARAM	154
NC_REC_DESIRED_MCS_N6_AC_DC_HIGH_PARAM	155
NC_REC_DESIRED_MCS_N7_AC_DC_LOW_PARAM	156
NC_REC_DESIRED_MCS_N7_AC_DC_HIGH_PARAM	157
NC_REC_DESIRED_MCS_N8_AC_DC_LOW_PARAM	158
NC_REC_DESIRED_MCS_N8_AC_DC_HIGH_PARAM	159
NC_REC_DESIRED_MCS_N9_AC_DC_LOW_PARAM	160
NC_REC_DESIRED_MCS_N9_AC_DC_HIGH_PARAM	161
NC_REC_DESIRED_MCS_S_AC_DC_LOW_PARAM	162

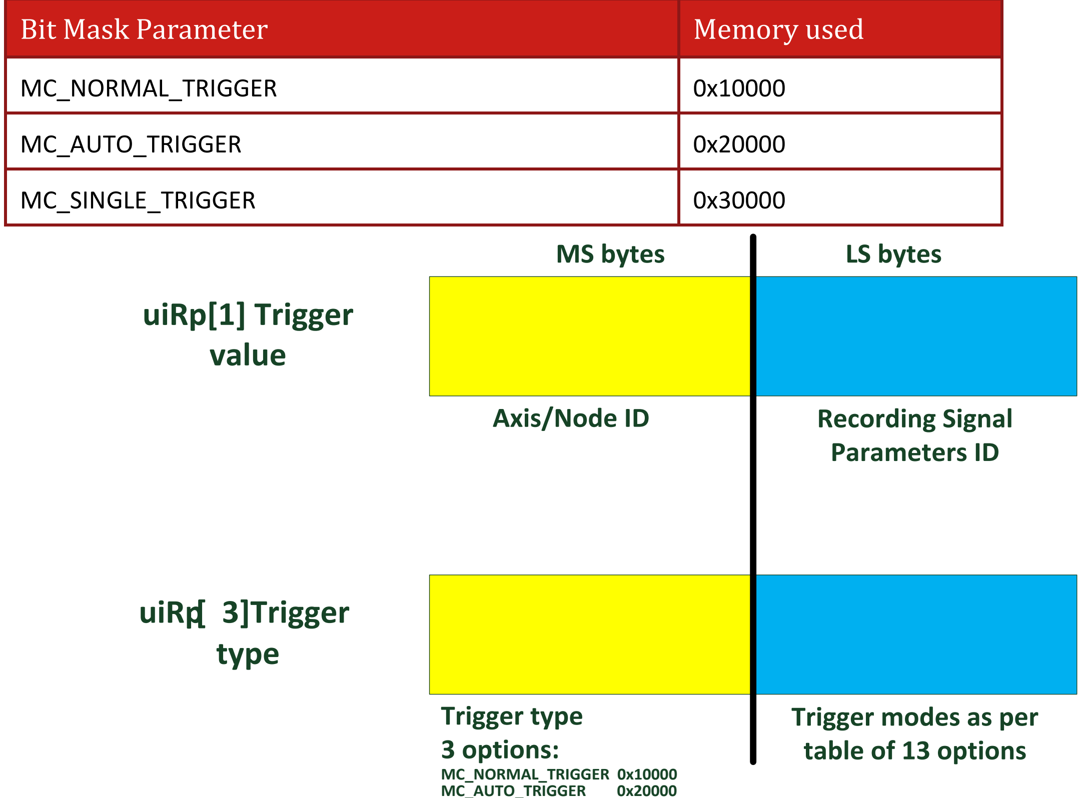
Recording Signal Variable	ID
NC_REC_DESIRED_MCS_S_AC_DC_HIGH_PARAM	163
NC_REC_END_MOTION_REASON_PARAM	164
NC_REC_ANALOG_INPUT_PARAM	165
NC_REC_DESIRED_MCS_X_POS_LOW_PARAM	166
NC_REC_DESIRED_MCS_X_POS_HIGH_PARAM	167
NC_REC_DESIRED_MCS_Y_POS_LOW_PARAM	168
NC_REC_DESIRED_MCS_Y_POS_HIGH_PARAM	169
NC_REC_DESIRED_MCS_Z_POS_LOW_PARAM	170
NC_REC_DESIRED_MCS_Z_POS_HIGH_PARAM	171
NC_REC_DESIRED_MCS_U_POS_LOW_PARAM	172
NC_REC_DESIRED_MCS_U_POS_HIGH_PARAM	173
NC_REC_DESIRED_MCS_V_POS_LOW_PARAM	174
NC_REC_DESIRED_MCS_V_POS_HIGH_PARAM	175
NC_REC_DESIRED_MCS_W_POS_LOW_PARAM	176
NC_REC_DESIRED_MCS_W_POS_HIGH_PARAM	177
NC_REC_DESIRED_ACS_A1_POS_LOW_PARAM	178
NC_REC_DESIRED_ACS_A1_POS_HIGH_PARAM	179
NC_REC_DESIRED_ACS_A2_POS_LOW_PARAM	180
NC_REC_DESIRED_ACS_A2_POS_HIGH_PARAM	181
NC_REC_DESIRED_ACS_A3_POS_LOW_PARAM	182
NC_REC_DESIRED_ACS_A3_POS_HIGH_PARAM	183
NC_REC_DESIRED_ACS_A4_POS_LOW_PARAM	184
NC_REC_DESIRED_ACS_A4_POS_HIGH_PARAM	185
NC_REC_DESIRED_ACS_A5_POS_LOW_PARAM	186
NC_REC_DESIRED_ACS_A5_POS_HIGH_PARAM	187
NC_REC_DESIRED_ACS_A6_POS_LOW_PARAM	188
NC_REC_DESIRED_ACS_A6_POS_HIGH_PARAM	189
NC_REC_DESIRED_MCS_X_VEL_LOW_PARAM	190
NC_REC_DESIRED_MCS_X_VEL_HIGH_PARAM	191
NC_REC_DESIRED_MCS_Y_VEL_LOW_PARAM	192
NC_REC_DESIRED_MCS_Y_VEL_HIGH_PARAM	193
NC_REC_DESIRED_MCS_Z_VEL_LOW_PARAM	194
NC_REC_DESIRED_MCS_Z_VEL_HIGH_PARAM	195

Recording Signal Variable	ID
NC_REC_DESIRED_MCS_U_VEL_LOW_PARAM	196
NC_REC_DESIRED_MCS_U_VEL_HIGH_PARAM	197
NC_REC_DESIRED_MCS_V_VEL_LOW_PARAM	198
NC_REC_DESIRED_MCS_V_VEL_HIGH_PARAM	199
NC_REC_DESIRED_MCS_W_VEL_LOW_PARAM	200
NC_REC_DESIRED_MCS_W_VEL_HIGH_PARAM	201
NC_REC_DESIRED_ACS_A1_VEL_LOW_PARAM	202
NC_REC_DESIRED_ACS_A1_VEL_HIGH_PARAM	203
NC_REC_DESIRED_ACS_A2_VEL_LOW_PARAM	204
NC_REC_DESIRED_ACS_A2_VEL_HIGH_PARAM	205
NC_REC_DESIRED_ACS_A3_VEL_LOW_PARAM	206
NC_REC_DESIRED_ACS_A3_VEL_HIGH_PARAM	207
NC_REC_DESIRED_ACS_A4_VEL_LOW_PARAM	208
NC_REC_DESIRED_ACS_A4_VEL_HIGH_PARAM	209
NC_REC_DESIRED_ACS_A5_VEL_LOW_PARAM	210
NC_REC_DESIRED_ACS_A5_VEL_HIGH_PARAM	211
NC_REC_DESIRED_ACS_A6_VEL_LOW_PARAM	212
NC_REC_DESIRED_ACS_A6_VEL_HIGH_PARAM	213
NC_REC_DESIRED_MCS_X_AC_DC_LOW_PARAM	214
NC_REC_DESIRED_MCS_X_AC_DC_HIGH_PARAM	215
NC_REC_DESIRED_MCS_Y_AC_DC_LOW_PARAM	216
NC_REC_DESIRED_MCS_Y_AC_DC_HIGH_PARAM	217
NC_REC_DESIRED_MCS_Z_AC_DC_LOW_PARAM	218
NC_REC_DESIRED_MCS_Z_AC_DC_HIGH_PARAM	219
NC_REC_DESIRED_MCS_U_AC_DC_LOW_PARAM	220
NC_REC_DESIRED_MCS_U_AC_DC_HIGH_PARAM	221
NC_REC_DESIRED_MCS_V_AC_DC_LOW_PARAM	222
NC_REC_DESIRED_MCS_V_AC_DC_HIGH_PARAM	223
NC_REC_DESIRED_MCS_W_AC_DC_LOW_PARAM	224
NC_REC_DESIRED_MCS_W_AC_DC_HIGH_PARAM	225
NC_REC_DESIRED_ACS_A1_AC_DC_LOW_PARAM	226
NC_REC_DESIRED_ACS_A1_AC_DC_HIGH_PARAM	227
NC_REC_DESIRED_ACS_A2_AC_DC_LOW_PARAM	228

Recording Signal Variable	ID
NC_REC_DESIRED_ACS_A2_AC_DC_HIGH_PARAM	229
NC_REC_DESIRED_ACS_A3_AC_DC_LOW_PARAM	230
NC_REC_DESIRED_ACS_A3_AC_DC_HIGH_PARAM	231
NC_REC_DESIRED_ACS_A4_AC_DC_LOW_PARAM	232
NC_REC_DESIRED_ACS_A4_AC_DC_HIGH_PARAM	233
NC_REC_DESIRED_ACS_A5_AC_DC_LOW_PARAM	234
NC_REC_DESIRED_ACS_A5_AC_DC_HIGH_PARAM	235
NC_REC_DESIRED_ACS_A6_AC_DC_LOW_PARAM	236
NC_REC_DESIRED_ACS_A6_AC_DC_HIGH_PARAM	237
NC_REC_SPEED_OVERRIDE_PARAM	238

10.4.4 Trigger Modes

The parameter uiRp controls the trigger value and type. These trigger values consist of high and low bytes. The uiRp[1] higher 2 bytes control the axis or node ID, wheras the uiRp[3] low 2 bytes may control the motion. Under these conditions uiRp[1] low 2 bytes should be 0. The uiRp[3] high 2 bytes control the trigger type which may be of three values as detailed in the section 10.4.1 above:



Bit Mask Parameter	Memory used
MC_NORMAL_TRIGGER	0x10000
MC_AUTO_TRIGGER	0x20000
MC_SINGLE_TRIGGER	0x30000

The following Trigger modes and their IDs are used in the Recording Data functions.

Parameter	ID	Explanation
TG_RECORDING_TRIGGER_TYPE_NO_TRIGGER	0	Edge : Rising (Positive Slope Over: TRIGVAL >= Level#1)
TG_RECORDING_TRIGGER_TYPE_EDGE_Rise	1	Edge : Rising (Positive Slope Over: TRIGVAL >= Level#1)
TG_RECORDING_TRIGGER_TYPE_EDGE_Fall	2	Edge : Falling (Negative Slope over: TRIGVAL <= Level#1)
TG_RECORDING_TRIGGER_TYPE_EDGE_WindowIn	3	Edge : Window In (Into the Window defined by: Level#2 <= TRIGVAL <= LEVEL#1)
TG_RECORDING_TRIGGER_TYPE_EDGE_WindowOut	4	Edge : Window Out (Out Of the Window defined by: Level#2 <= TRIGVAL <= LEVEL#1)
TG_RECORDING_TRIGGER_TYPE_LEVEL_GE	5	Level : >= (GreaterEqual The: TRIGVAL >= Level#1)
TG_RECORDING_TRIGGER_TYPE_LEVEL_SE	6	Level : <= (SmallerEqual Then: TRIGVAL <= Level#1)
TG_RECORDING_TRIGGER_TYPE_LEVEL_WindowInside	7	Level : Inside Window (Inside of Window defined by: Level#2 <= TRIGVAL <= LEVEL#1)

TG_RECORDING_TRIGGER_TYPE_LEVEL_WindowOutside	8	Level : outside Window (Outside of Window defined by: Level#2 <= TRIGVAL <= LEVEL#1)
TG_RECORDING_TRIGGER_TYPE_EDGE_Rise_Mask	9	Rising-Edge + MASK (Positive Slope Over: (TRIGVAL & MASK) == MASK)
TG_RECORDING_TRIGGER_TYPE_EDGE_Fall_Mask	10	Falling-Edge + MASK (Negative Slope Over: (TRIGVAL & MASK) != MASK)
TG_RECORDING_TRIGGER_TYPE_LEVEL_GE_Mask	11	Grater-Equal + Mask (Equal TO: (TRIGVAL & MASK) == MASK)
TG_RECORDING_TRIGGER_TYPE_LEVEL_SE_Mask	12	Smaller-Equal + Mask (Not Equal TO: (TRIGVAL & MASK) != MASK)
TG_RECORDING_TRIGGER_TYPE_BEGIN_MOTION_Mask	13	Begin Motion

10.5 Data Recording Functions

10.5.1 MMC_BeginRecording

Starts the recording of internal controller variables data from the Maestro server.

```
MMC_LIB_API int MMC_BeginRecordingCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_BEGIN_RECORDING_IN* pInParam,
OUT MMC_BEGIN_RECORDING_OUT* pOutParam
);
```

Motion Mode NC – N/A Distributed – N/A

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_BEGIN_RECORDING_IN** input data structure using the MMC_BeginRecording function.

pOutParam

Points to the **MMC_BEGIN_RECORDING_OUT** output structure receiving information, as a result of calling the MMC_BeginRecording function.

Remarks

None

Scope

All

MMC_BEGIN_RECORDING_IN Structure

```
typedef struct{
unsigned long uiRg;
unsigned long uiRl;
unsigned long uiRc;
unsigned long uiRv[NC_MAX_REC_SIGNALS_NUM];
unsigned long uiRp[NC_MAX_REC_PARAMS_NUM];
}MMC_BEGIN_RECORDING_IN;
```

Parameters

uiRg

Recording Data Gap which specifies the sampling rate of the recorder. Any +ve integer value

uiRl

Recording Data Length with a buffer size (default size 4MB). Any +ve integer value.

uiRc

Recording Data Signals Bit mask according to the definitions described in section 10.4.1 **Recording Data Signals Bitmask Definitions**. Defines which of mapped signals should be recorded with up to 32 different synchronized signals. Any +ve integer value.

uiRv

uiRv is the recording Signals ID mapping enumerator which maps the IDs of the recordable signals to logical IDs that the recorder can reference. It is a 32 bit mask assembled from the AxisNumber and the Signal parameter. The upper 16 bits are the axis reference, and the lower 16 bits the signal parameter, e.g. 0x00020015, where the 0002 refers to axis 02, and the 0015 refers to the signal ID 21 (in Hex). Refer to the ID definitions described in section **10.4.2 Recording Parameters**

The following table lists the Recording Parameters Rp(i).

Recording Parameters	ID
TG_RECORDING_SPARE	0
TG_RECORDING_TRIGGER_VALUE	1
TG_RECORDING_PRE_TRIGGER_LENGTH	2
TG_RECORDING_TRIGGER_TYPE	3
TG_RECORDING_TRIGGER_LEVEL_1	4
TG_RECORDING_TRIGGER_LEVEL_2	5
TG_RECORDING_TRIGGER_POLARITY	6
TG_RECORDING_TRIGGER_IN_MASK	7

Recording Signal Parameters.

[NC_MAX_REC_SIGNALS_NUM] is an array value of between [1….22] and *uiRv* can have any +ve integer value.

uiRp

uiRp is the ID integer value of the Recording Parameters. Refer to the section **10.4.4 Trigger Modes** for further details. The recorder parameters defines which event will trigger the recorder, and the trigger position, according to the following definitions:

Recording Parameters - RP[N]	ID	Definition
TG_RECORDING_SPARE	0	Spare
TG_RECORDING_TRIGGER_VALUE	1	Defines which of mapped signals should be recorded, but only 1 bit may be non-zero. The trigger variable does not need to be one of the recorded variables.
TG_RECORDING_PRE_TRIGGER_LENGTH	2	The percentage of the recorded signal taken before the trigger event. (recorder trigger delay*)
TG_RECORDING_TRIGGER_TYPE	3	
TG_RECORDING_TRIGGER_LEVEL_1	4	Level for positive slope trigger, or high side for window trigger.
TG_RECORDING_TRIGGER_LEVEL_2	5	Level for negative slope trigger, or low side for window trigger.
TG_RECORDING_TRIGGER_POLARITY	6	Logic for bit field trigger- 0 positive logic, 1 – negative
TG_RECORDING_TRIGGER_IN_MASK	7	Mask for bit field trigger

[NC_MAX_REC_PARAMS_NUM] is an array value of [1….8].

MMC_BEGIN_RECORDING_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_BEGIN_RECORDING_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 10-1 describes the function block for MMC_BeginRecording

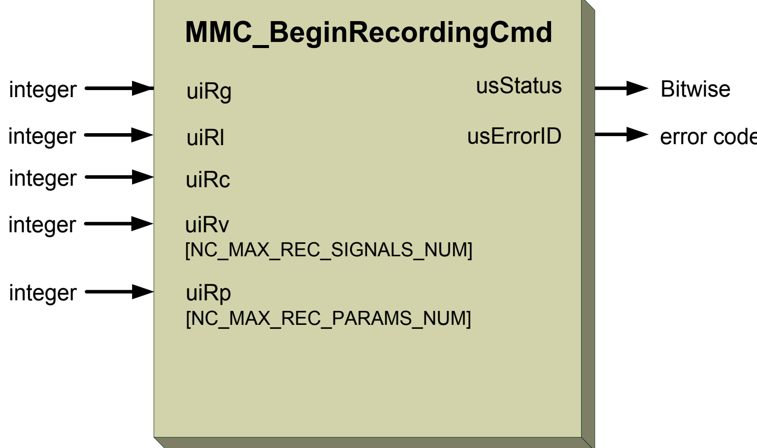


Figure 10-1: MMC_BeginRecording function block

10.5.1.2 Function Block Code Example

```
int rc;
MMC_BEGIN_RECORDING_IN     stBeginRecording_in;
MMC_BEGIN_RECORDING_OUT    stBeginRecording_out;
//
// Inserting the structure parameters:
stBeginRecording_in.uiRg        = 1000;       // Recording Data Gap
stBeginRecording_in.uiRl        = 10;         // Recording Data Length
stBeginRecording_in.uiRc        = 0x10000;    // Parameter array index
stBeginRecording_in.uiRv[1]     = 0x0020015;           // Recording axis number and the Signals
ID mapping
stBeginRecording_in.uiRv[2]     = 0x0030015;
stBeginRecording_in.uiRv[3]     = 0x0040015;
stBeginRecording_in.uiRp[1]     = 1; //Recorder parameters defines which event will trigger the
recorder
stBeginRecording_in.uiRp[2]     = 2;
stBeginRecording_in.uiRp[3]     = 3;
//
rc = MMC_BeginRecordingCmd (hConn, &stBeginRecording_in, &stBeginRecording_out);
if (rc != 0)
{
    HandleError() ;
}
```

10.5.2 MMC_StopRecording

Halts recording of the Maestro server data.

```
MMC_LIB_API int MMC_StopRecordingCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_STOP_RECORDING_IN* pInParam,
OUT MMC_STOP_RECORDING_OUT* pOutParam
);
```

Motion Mode	NC – N/A	Distributed – N/A
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_STOP_RECORDING_IN** input data structure using the MMC_StopRecording function.

pOutParam

Points to the **MMC_STOP_RECORDING_OUT** output structure receiving information, as a result of calling the MMC_StopRecording function.

Remarks

None

Scope

All

MMC_STOP_RECORDING_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_STOP_RECORDING_IN;
```

Parameters

dummy

Dummy values. Any +ve character value.

MMC_STOP_RECORDING_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_STOP_RECORDING_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 10-2 describes the function block for MMC_StopRecording

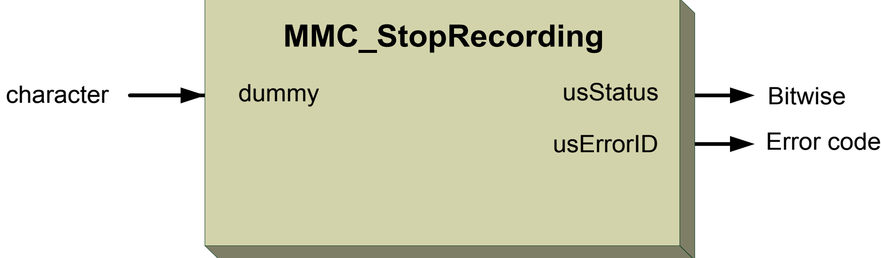


Figure 10-2: MMC_StopRecording function block

10.5.2.2 Function Block Code Example

```
int rc;
MMC_STOP_RECORDING_IN      stStopRecording_in;
MMC_STOP_RECORDING_OUT     stStopRecording_out;
//
// Inserting the structure parameters:
stStopRecording_in.dummy   = 1;     // dummy data
//
rc = MMC_StopRecordingCmd (hConn, &stStopRecording_in, &stStopRecording_out);
if (rc != 0)
{
    HandleError() ;
}
```

10.5.3 MMC_UploadData

Uploads recording data to the Maestro.

```
MMC_LIB_API int MMC_UploadDataCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_UPLOAD_DATA_IN* pInParam,
OUT MMC_UPLOAD_DATA_OUT* pOutParam
);
```

Motion Mode	NC – N/A	Distributed – N/A
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_UPLOAD_DATA_IN** input data structure using the MMC_UploadData function.

pOutParam

Points to the **MMC_UPLOAD_DATA_OUT** output structure receiving information, as a result of calling the MMC_UploadData function.

Remarks

None

Scope

All

MMC_UPLOAD_DATA_IN Structure

```
typedef struct{
unsigned int uiFrom;
unsigned int uiTo;
unsigned int uiBufIdx;
}MMC_UPLOAD_DATA_IN;
```

Parameters

uiFrom

Upload from index. Any +ve integer value.

uiTo

Upload to index. Any +ve integer value.

uiBufIdx

Buffer Index. Any +ve integer value.

MMC_UPLOAD_DATA_OUT Structure

```
typedef struct{
long ulUpdatData[NC_MAX_LONG];
unsigned short usStatus;
short sErrorID;
}MMC_UPLOAD_DATA_OUT;
```

Parameters

ulUpdatData

Update data status. Any +ve or –ve integer value, with [NC_MAX_LONG] having array values [1 - ??]

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 10-3 describes the function block for MMC_UploadData

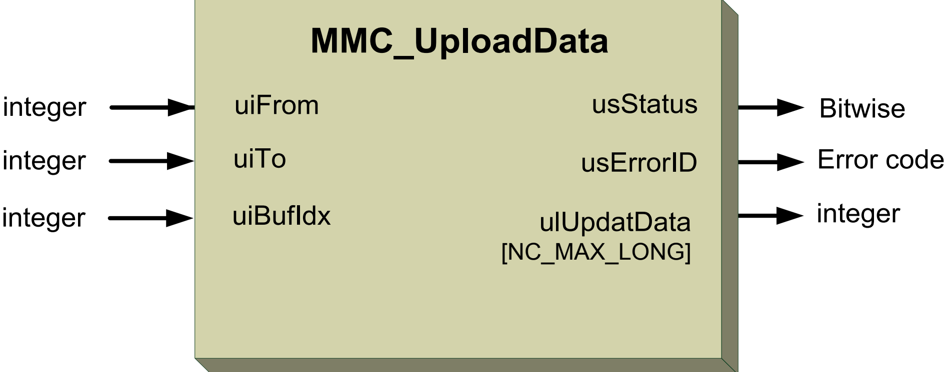


Figure 10-3: MMC_UploadData function block

10.5.3.2 Function Block Code Example

```
int rc;
MMC_UPLOAD_DATA_IN          stUploadData_in;
MMC_UPLOAD_DATA_OUT         stUploadData_out;
//
// Inserting the structure parameters:
stUploadData_in.uiFrom      = 1;     // Upload from index
stUploadData_in.uiTo        = 100;   // Upload to index
stUploadData_in.uiBufIdx    = 1000;  // Buffer Index
//
rc = MMC_UploadDataCmd (hConn, &stUploadData_in, &stUploadData_out);
if (rc != 0)
{
    HandleError() ;
}
```

10.5.4 MMC_RecStatus

Requests the status of the recording.

```
MMC_LIB_API int MMC_RecStatusCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_REC_STATUS_IN* pInParam,
OUT MMC_REC_STATUS_OUT* pOutParam
);
```

Motion Mode	NC – N/A	Distributed – N/A
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_REC_STATUS_IN** input data structure using the MMC_RecStatus function.

pOutParam

Points to the **MMC_REC_STATUS_OUT** output structure receiving information, as a result of calling the MMC_RecStatus function.

Remarks

None

Scope

All

MMC_REC_STATUS_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_REC_STATUS_IN;
```

Parameters

dummy

Dummy values. Any +ve character value.

MMC_REC_STATUS_OUT Structure

```
typedef struct{
unsigned long uiRr;
unsigned long uiSr;
unsigned short usStatus;
short sErrorID;
}MMC_REC_STATUS_OUT;
```

Parameters

uiRr

Rest Recording Index. Reads back recorder status. Any +ve integer value

uiSr

Recorder Trigger Status. Status register, which indicates the status of the recorder; idle, armed, triggered and recording, or ready with data, where the lower 8 bits display the following options to be entered:

0 – Arming

1 – Waiting Opposite trigger

2 – Waiting Trigger

3 – Trigger Detected

4 – No Trigger

The following 8 bits (bitwise) display the following options to be entered:

0 – No Buffer Ready

1 – Buffer 1 ready

2 – Buffer 2 ready

3 – Both Buffers ready

Values accepted are any +ve integer value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 10-4 describes the function block for MMC_RecStatus

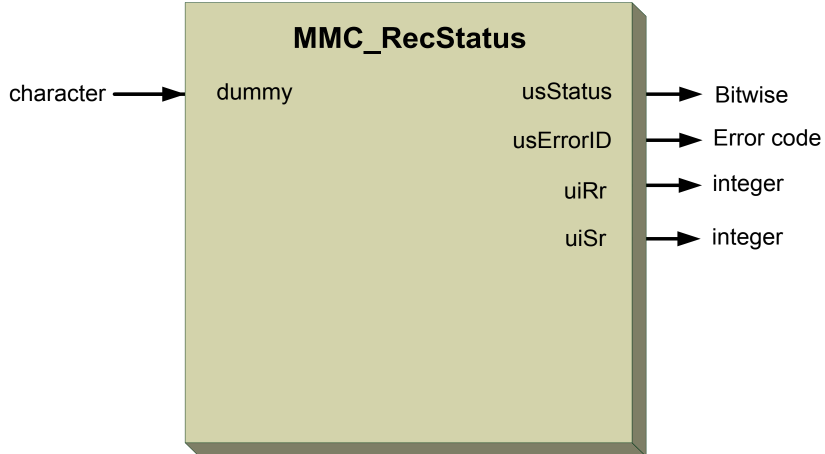


Figure 10-4: MMC_RecStatus function block

10.5.4.2 Function Block Code Example

```
int rc;
MMC_REC_STATUS_IN      stRecStatus_in;
MMC_REC_STATUS_OUT     stRecStatus_out;
//
// Inserting the structure parameters:
stRecStatus_in.dummy   = 1;      // dummy data
//
rc = MMC_RecStatusCmd (hConn, &stRecStatus_in, &stRecStatus_out);
if (rc != 0)
{
    HandleError() ;
}
```

10.5.5 MMC_UploadDataHeader

Recorder upload data header.

```
MMC_LIB_API int MMC_UploadDataHeaderCmd(
IN MMC_CONNECT_HNDL hConn,
OUT NC_UPLOAD_REC_HEADER_STRUCT* pOutParam
);
```

Motion Mode	NC – N/A	Distributed – N/A
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pOutParam

[OUT] Points to the **MMC_POSITIONPROFILE_OUT** output structure receiving information, as a result of calling the MMC_UploadDataHeader function.

Remarks

None

Scope

All

NC_UPLOAD_REC_HEADER_STRUCT Output Structure

```
typedef struct {
unsigned long ulDummy;
unsigned long ulRc;
unsigned long ulRg;
unsigned long ulRl;
NC_REC_RV_STRUCT usRv[NC_MAX_REC_SIGNALS_NUM];
unsigned long ulRp[NC_MAX_REC_PARAMS_NUM];
unsigned long ulTi;
unsigned long ulTs;
unsigned long ulSpare[3];
unsigned short usStatus;
short sErrorID;
unsigned char dummy [952];
}NC_UPLOAD_REC_HEADER_STRUCT;
```

Parameters

ulDummy

Dummy data. To align and upload data field in a common message. Any +ve character value.

ulRc

Recording data signals bit mask according to the definitions described in section 10.4.1 **Recording Data Signals Bitmask Definitions**. Defines which of mapped signals should be recorded with up to 32 different synchronized signals. Any +ve integer value.

ulRg

Recording data gap, which specifies the sampling rate of the recorder. Any +ve integer value.

ulRl

Recording data length with a buffer size (default size 4MB). Any +ve integer value.

usRv

usRv is the recording Signals ID mapping enumerator which maps the IDs of the recordable signals to logical IDs that the recorder can reference. Refer to the ID definitions described in section **10.4.2 Recording Parameters** and **10.4.4 Trigger Modes.**

[NC_MAX_REC_SIGNALS_NUM] is an array value of between [1….32] and *usRv* can have any +ve integer value.

NC_REC_RV_STRUCT is the recorder signal value structure with the following parameters:

ulValue

Signal value reference handle of the axis. Any +ve integer value.

ulType

Signal value type. The Enumerator ID has the following variable values, which describe the enumerator NC_RV_TYPE_ENUM, and are the recorder supported data types.

Recorder Supported Data Types	ID
NC_UCHAR_TYPE	0
NC_CHAR_TYPE	1
NC_USHORT_TYPE	2
NC_SHORT_TYPE	3
NC_UINT_TYPE	4
NC_INT_TYPE	5
NC_ULONG_TYPE	6
NC_LONG_TYPE	7
NC_FLOAT_TYPE	8
NC_DOUBLE_L_TYPE	9
NC_DOUBLE_H_TYPE	10

ulFactor

Signal value multiple factor of the cycle time. Any positive float value

ulRp

ulRp is the ID integer value of the Recording Parameters. Refer to the section **10.4.4 Trigger Modes** for further details. The recorder parameters defines which event will trigger the recorder, and the trigger position, according to the following definitions:

Recording Parameters - RP[N]	ID	Definition
TG_RECORDING_SPARE	0	Spare
TG_RECORDING_TRIGGER_VALUE	1	Defines which of mapped signals should be recorded, but only 1 bit may be non-zero. The trigger variable does not need to be one of the recorded variables.
TG_RECORDING_PRE_TRIGGER_LENGTH	2	The percentage of the recorded signal taken before the trigger event. (recorder trigger delay*)
TG_RECORDING_TRIGGER_TYPE	3	
TG_RECORDING_TRIGGER_LEVEL_1	4	Level for positive slope trigger, or high side for window trigger.
TG_RECORDING_TRIGGER_LEVEL_2	5	Level for negative slope trigger, or low side for window trigger.
TG_RECORDING_TRIGGER_POLARITY	6	Logic for bit field trigger- 0 positive logic, 1 –

		negative
TG_RECORDING_TRIGGER_IN_MASK	7	Mask for bit field trigger

[NC_MAX_REC_PARAMS_NUM] is an array value of [1....8].

ulTi

Trigger Index. Any +ve integer value.

ulTs

Recorder Update Time. Sampling time, the basic resolution of recorder, which is the basic time of the NC process. Any +ve integer value.

ulSpare[3]

Spare. For internal use only. Any +ve integer value with a maximum of 3 integers.

dummy [952]

Dummy data. Any +ve character value to a maximum of 952 characters.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 10-5 describes the function block for MMC_UploadDataHeader

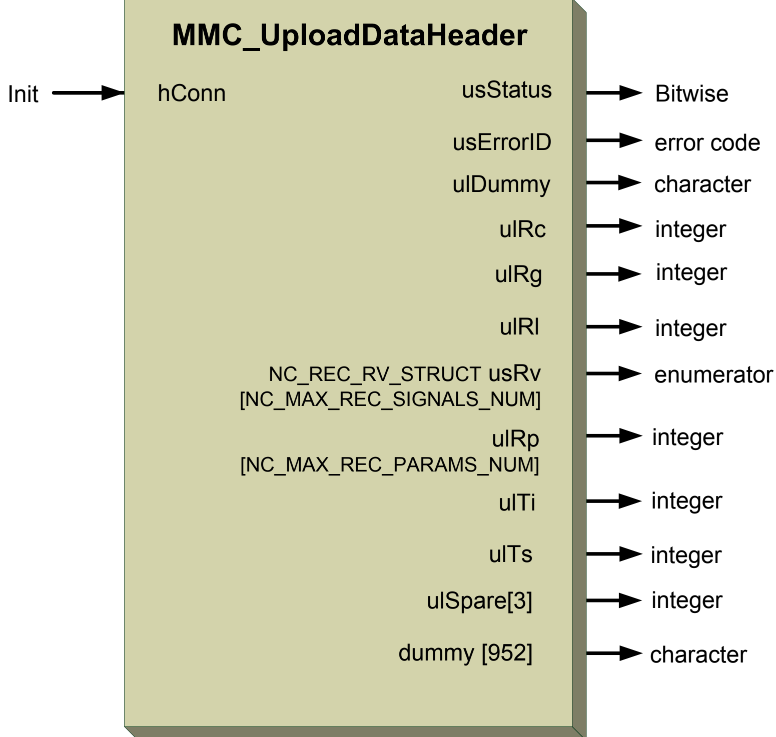


Figure 10-5: MMC_UploadDataHeader function block

10.5.5.2 Function Block Code Example

```
int rc;
NC_UPLOAD_REC_HEADER_STRUCT     stUploadRecHeadStr;
//NC_REC_RV_STRUCT (ulValue, ulType, ulFactor);
//
//
rc = MMC_UploadDataHeaderCmd (hConn, &stUploadRecHeadStr);
if (rc != 0)
{
    HandleError() ;
}
```

Chapter 11: Bulk Parameters Reading

This set of functions allows the user to retrieve all the parameters for a limited number of single axes operating simultaneously using a single function call. The Maestro allows a user program or host via Ethernet interface to retrieve all required axis parameters. As the number of axes and parameters increases, the time to retrieve the parameters increases proportionately. This operation is relatively slow, and this chapter explains the purpose of bulk read, which increases the performance of the parameters retrieval procedure dramatically.

This feature can be used in two difference scenarios:

- To import multiple parameters when using Maestro Multiple Axes in the EASII software
- To retrieve multiple parameters for multiple single axis when programming in C or C++ (or any other host software)

11.1 Bulk Reading Functions

The following Bulk Reading functions are described:

Bulk Reading functions
MMC_ConfigBulkRead
MMC_PerformBulkRead

11.1.1 MMC_ConfigBulkRead

Configures the function to read all parameters from multiple axes.

```
MMC_LIB_API int MMC_ConfigBulkReadCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_CONFIGBULKREAD_IN* pInParam,
OUT MMC_CONFIGBULKREAD_OUT* pOutParam
);
```

Motion Mode	NC – Immaterial	Distributed – Immaterial
Source	GMAS\includes\MMC_general_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CONFIGBULKREAD_IN** input data structure using the MMC_ConfigBulkRead function.

pOutParam

Points to the **MMC_CONFIGBULKREAD_OUT** output structure receiving information, as a result of calling the MMC_ConfigBulkRead function.

Remarks

The Maestro has three function block queues, large, medium, and small, function blocks. These are common to all nodes. Every time a function block is inserted, a function block is taken from one of these queues. The parameters *iFreeLargeFbsNumber, iFreeMediumFbsNumber and iFreeSmallFbsNumber* indicate the number of free function blocks available in each queue.

Scope

All

MMC_CONFIGBULKREAD_IN Structure

```
typedef struct mmc_configbulkread_in{
NC_BULKREAD_PARAMETERS_UNION uBulkReadParams;
NC_BULKREAD_CONFIG_ENUM eConfiguration;
unsigned short usAxisRefArray[NC_MAX_AXES_PER_BULK_READ];
unsigned short usNumberOfAxes;
unsigned char ucIsPreset;
}MMC_CONFIGBULKREAD_IN;
```

Parameters

uBulkReadParams

Defines what parameters will be read. It can be either one of the predefined presets, or a custom array with user filled values. ulBulkReadParameters is an array of size 32 Bit, and to set the number of signals to be read from the bulk read user, it is necessary to set the value of this parameter to 0 after the signals requested. If two signals are required, then the ulBulkReadParameters[2] value should be -1 otherwise a junk value that was in the array could be considered as a signal requested

The Union parameter NC_BULKREAD_PARAMETERS_UNION is defined as follows:

```
typedef union{
NC_BULKREAD_PRESET_ENUM eBulkReadPreset;
unsigned long ulBulkReadParameters[NC_MAX_REC_SIGNALS_NUM];
} NC_BULKREAD_PARAMETERS_UNION;
```

Where the user can select to use either the parameters:

NC_BULKREAD_PRESET_ENUM eBulkReadPreset

Or

unsigned long ulBulkReadParameters[NC_MAX_REC_SIGNALS_NUM]

eBulkReadPreset

The bulk read preset enumerator defined by NC_BULKREAD_PRESET_ENUM with values:

```
eNC_BULKREAD_PRESET_NONE,
eNC_BULKREAD_PRESET_1,
eNC_BULKREAD_PRESET_2,
eNC_BULKREAD_PRESET_3,
eNC_BULKREAD_PRESET_4,
eNC_BULKREAD_PRESET_5,
eNC_BULKREAD_PRESET_MAX,
```

Each of the bulk read presets defines a fixed set of parameters to be read:

```
typedef struct eNC_BULKREAD_PRESET_1{
int aPos;
int aVel;
int aTorque;
unsigned long ulAxisStatus;
unsigned int uiInputs;
OPM402 eOpMode;
```

```
}NC_BULKREAD_PRESET_1;

typedef struct eNC_BULKREAD_PRESET_2{
NC_BULKREAD_PRESET_1 stAxisParams;
int iFreeLargeFbsNumber;
int iFreeMediumFbsNumber
int iFreeSmallFbsNumber;
}NC_BULKREAD_PRESET_2;

typedef struct eNC_BULKREAD_PRESET_3{
int aPos;
int aVel;
int aTorque;
unsigned long ulAxisStatus;
unsigned int uiInputs;
OPM402 eOpMode;
NC_BULKREAD_VARIABLE_UNION ucCommError;
NC_BULKREAD_VARIABLE_UNION usLastEmcyErrorCode;
NC_BULKREAD_VARIABLE_UNION usControlWord;
NC_BULKREAD_VARIABLE_UNION usStatusWord;
}NC_BULKREAD_PRESET_3;

typedef struct eNC_BULKREAD_PRESET_4{
int aPos;
int aHWPos;
int iPosFollowingErr;
int aVel;
int aTorque;
unsigned long ulAxisStatus;
unsigned int uiInputs;
OPM402 eOpMode;
unsigned int uiStatusRegister;
unsigned int uiMcsLimitRegister;
}NC_BULKREAD_PRESET_4;

typedef struct eNC_BULKREAD_PRESET_5{
int aPos;
int aHWPos;
int iPosFollowingErr;
int aVel;
int aTorque;
unsigned long ulAxisStatus;
unsigned int uiInputs;
OPM402 eOpMode;
unsigned int uiStatusRegister;
unsigned int uiMcsLimitRegister;
NC_BULKREAD_VARIABLE_UNION usLastEmcyErrorCode;
NC_BULKREAD_VARIABLE_UNION usControlWord;
NC_BULKREAD_VARIABLE_UNION usStatusWord;
NC_BULKREAD_VARIABLE_UNION ucCommError;
}NC_BULKREAD_PRESET_5;
```

```
typedef union{
char cVar;
unsigned char ucVar;
short sVar;
unsigned short usVar;
int iVar;
unsigned int uiVar;
long lVar;
unsigned long ulVar;
}NC_BULKREAD_VARIABLE_UNION;
```

Where the user can select to use either of the parameters:

Character cVar, Unsigned character ucVar, short sVar, unsigned short usVar, integer iVar, unsigned integer uiVar, long lVar, or un signed long ulVar.

Var is the value of the variable.

Each of these preset parameters have the following definitions:

aPos

Actual position integer. Integer values

aHWPos

Actual hardware position. Integer values

iPosFollowingErr

Position following error. Integer values/

aVel

Actual velocity integer

aTorque

Actual Torque integer

ulAxisStatus

Status of the axis with +ve bitwise values

uiInputs

Digital inputs with any +ve integer value

eOpMode

Motion mode

uiStatusRegister

Variable provides information on the special status of an axis. Refer to the chapter **14.2 Interfaces** . +ve integer value.

uiMcsLimitRegister

Parameter represents the status of the MCS limits in a specific group. +ve integer value.

iFreeLargeFbsNumber

Number of large free function blocks available in the queue for this size.

iFreeMediumFbsNumber

Number of medium sized free function blocks available in the queue for this size. Integer value accepted

iFreeSmallFbsNumber

Number of small free function blocks available in each queue. Integer value accepted

usControlWord

CANopen DS402 control word. Any +ve short value.

usStatusWord

CANopen DS402 status word. Any +ve short value.

ucCommError

Input axis communication error. +ve character values.

usLastEmcyErrorCode

Last recorded emergency code. usLastEmcyErrorCode is the emergency error code received from the drive.

ulBulkReadParameters

The array of ulBulkReadParameters defined by [NC_MAX_REC_SIGNALS_NUM] represents a set of parameters to be retrieved from the Maestro, with a maximum of 32 parameters.

NC_BULKREAD_CONFIG_ENUM eConfiguration

Defines the reading source. eBULKREAD_CONFIG_1 is reserved to the EAS application. NC_BULKREAD_CONFIG_ENUM defines the following values:

eBULKREAD_CONFIG_NONE = -1,

eBULKREAD_CONFIG_1 = 0,

eBULKREAD_CONFIG_2 = 1,

eBULKREAD_CONFIG_3 = 2,

eBULKREAD_CONFIG_4 = 3,

eBULKREAD_CONFIG_MAX,

usAxisRefArray

Defines the array that will contain the axisrefs to be read **(**not masked), where [NC_MAX_AXES_PER_BULK_READ] has a range between 1 and 100.

If an error is created, it should return the NC_BULK_READ_NUM_OF_AXES_OUT_OF_RANGE error.

usNumberOfAxes

Defines the number of axes, and is the total number of axes to be bulk read.

ucIsPreset

Whether the preset parameters are used or not. Values accepted are 0, or 1.

MMC_CONFIGBULKREAD_OUT Structure

```
typedef struct mmc_configbulkread_out{
float fFactorsArray[NC_MAX_BULK_READ_READABLE_PACKET_SIZE];
unsigned short usStatus;
short sErrorID;
} MMC_CONFIGBULKREAD_OUT;
```

Parameters

fFactorsArray

Defines what multiplication factor is needed to apply to each read parameter. Dependent on the array [NC_MAX_BULK_READ_READABLE_PACKET_SIZE] with values of 0 – 350.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within the function. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 11-1 describes the function for MMC_ConfigBulkRead.

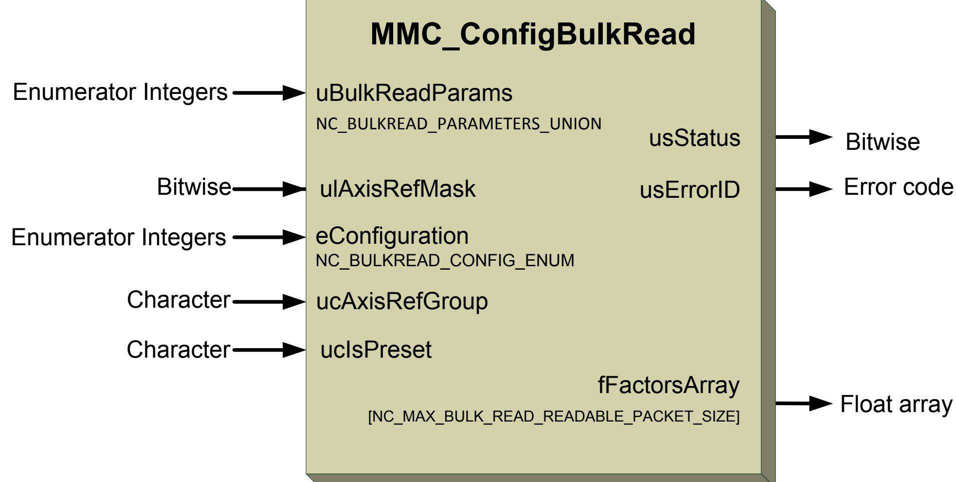


Figure 11-1: MMC_ConfigBulkRead function

11.1.1.2 Function Code Example

```
MMC_CONFIGBULKREAD_IN stCfgIn;
MMC_CONFIGBULKREAD_OUT stCfgOut;

int rc = NC_OK;

stCfgIn.eConfiguration                    = eBULKREAD_CONFIG_2;
stCfgIn.uBulkReadParams.eBulkReadPreset   = eNC_BULKREAD_PRESET_2;
stCfgIn.ucIsPreset                        = 1;
stCfgIn.usAxisRefArray[0]                 = 0;
stCfgIn.usAxisRefArray[1]                 = 1;
stCfgIn.usNumberOfAxes                    = 2;

rc = MMC_ConfigBulkReadCmd(g_hConnectHndl, &stCfgIn, &stCfgOut);

etc.
```

11.1.2 MMC_PerformBulkRead

Reads those parameters which were configured by a call to ConfigBulkRead, from multiple axes.

```
MMC_LIB_API int MMC_PerformBulkReadCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_PERFORMBULKREAD_IN* pInParam,
OUT MMC_PERFORMBULKREAD_OUT* pOutParam
);
```

Motion Mode NC – Immaterial Distributed – Immaterial

Source GMAS\includes\MMC_general_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_PERFORMBULKREAD_IN** input data structure using the MMC_PerformBulkRead function.

pOutParam

Points to the **MMC_PERFORMBULKREAD_OUT** output structure receiving information, as a result of calling the MMC_PerformBulkRead function.

Remarks

None

Scope

All

MMC_PERFORMBULKREAD_IN Structure

```
typedef struct mmc_performbulkread_in{
NC_BULKREAD_CONFIG_ENUM eConfiguration;
} MMC_PERFORMBULKREAD_IN;
```

Parameters

eConfiguration

Defines the reading source. eBULKREAD_CONFIG_1 is reserved to the EAS application. Acceptable values are eBULKREAD_CONFIG_1 and eBULKREAD_CONFIG_2.

NC_BULKREAD_CONFIG_ENUM defines the following values:

eBULKREAD_CONFIG_NONE,

eBULKREAD_CONFIG_1,

eBULKREAD_CONFIG_2,

eBULKREAD_CONFIG_MAX,

MMC_PERFORMBULKREAD_OUT Structure

```
typedef struct mmc_performbulkread_out{
unsigned long ulOutBuf[NC_MAX_BULK_READ_READABLE_PACKET_SIZE];
NC_BULKREAD_PRESET_ENUM eChosenPreset;
unsigned short usStatus;
short sErrorID;
} MMC_PERFORMBULKREAD_OUT;
```

Parameters

ulOutBuf[NC_MAX_BULK_READ_READABLE_PACKET_SIZE];

Defines the output buffer read data with a maximum [NC_MAX_BULK_READ_READABLE_PACKET_SIZE] array size of 350.

eChosenPreset

The bulk read preset enumerator defined by NC_BULKREAD_PRESET_ENUM with values:

eNC_BULKREAD_PRESET_NONE,
eNC_BULKREAD_PRESET_1,
eNC_BULKREAD_PRESET_2,
eNC_BULKREAD_PRESET_3,
eNC_BULKREAD_PRESET_4,
eNC_BULKREAD_PRESET_5,
eNC_BULKREAD_PRESET_MAX,

If a user defined parameters array is used for ulBulkReadParameters[NC_MAX_REC_SIGNALS_NUM], then the value of *eChosenPreset* will be eNC_BULKREAD_PRESET_MAX.

Each of the bulk read presets defines a fixed set of parameters to be read:

typedef struct eNC_BULKREAD_PRESET_1{

```
int aPos;
int aVel;
int aTorque;
unsigned long ulAxisStatus;
unsigned int uiInputs;
OPM402 eOpMode;
}NC_BULKREAD_PRESET_1;

typedef struct eNC_BULKREAD_PRESET_2{
NC_BULKREAD_PRESET_1 stAxisParams;
int iFreeLargeFbsNumber;
int iFreeMediumFbsNumber
int iFreeSmallFbsNumber;
}NC_BULKREAD_PRESET_2;

typedef struct eNC_BULKREAD_PRESET_3{
int aPos;
int aVel;
int aTorque;
unsigned long ulAxisStatus;
unsigned int uiInputs;
OPM402 eOpMode;
NC_BULKREAD_VARIABLE_UNION ucCommError;
NC_BULKREAD_VARIABLE_UNION usLastEmcyErrorCode;
NC_BULKREAD_VARIABLE_UNION usControlWord;
NC_BULKREAD_VARIABLE_UNION usStatusWord;
}NC_BULKREAD_PRESET_3;

typedef struct eNC_BULKREAD_PRESET_4{
int aPos;
int aHWPos;
int iPosFollowingErr;
int aVel;
int aTorque;
unsigned long ulAxisStatus;
unsigned int uiInputs;
OPM402 eOpMode;
unsigned int uiStatusRegister;
unsigned int uiMcsLimitRegister;
}NC_BULKREAD_PRESET_4;

typedef struct eNC_BULKREAD_PRESET_5{
int aPos;
int aHWPos;
int iPosFollowingErr;
int aVel;
int aTorque;
unsigned long ulAxisStatus;
unsigned int uiInputs;
OPM402 eOpMode;
unsigned int uiStatusRegister;
unsigned int uiMcsLimitRegister;
NC_BULKREAD_VARIABLE_UNION usLastEmcyErrorCode;
```

```
NC_BULKREAD_VARIABLE_UNION usControlWord;
NC_BULKREAD_VARIABLE_UNION usStatusWord;
NC_BULKREAD_VARIABLE_UNION ucCommError;
}NC_BULKREAD_PRESET_5;

typedef union{
char cVar;
unsigned char ucVar;
short sVar;
unsigned short usVar;
int iVar;
unsigned int uiVar;
long lVar;
unsigned long ulVar;
}NC_BULKREAD_VARIABLE_UNION;
```

Where the user can select to use either of the parameters:

Character cVar, Unsigned character ucVar, short sVar, unsigned short usVar, integer iVar, insigned integer uiVar, long lVar, or un signed long ulVar.

Var is the value of the variable.

Each of these preset parameters have the following definitions:

aPos

Actual position integer. Integer values

aHWPos

Actual hardware position. Integer values

iPosFollowingErr

Position following error. Integer values/

aVel

Actual velocity integer

aTorque

Actual Torque integer

ulAxisStatus

Status of the axis with +ve bitwise values

uiInputs

Digital inputs with any +ve integer value

eOpMode

Motion mode

uiStatusRegister

Variable provides information on the special status of an axis. Refer to the chapter 14.2

Interfaces. +ve integer value.

uiMcsLimitRegister

Parameter represents the status of the MCS limits in a specific group. +ve integer value.

iFreeLargeFbsNumber

Number of large free function blocks avaliable in the queue for this size.

iFreeMediumFbsNumber

Number of medium sized free function blocks avaliable in the queue for this size. Integer value accepted

iFreeSmallFbsNumber

Number of small free function blocks avaliable in each queue. Integer value accepted

usControlWord

CANopen DS402 control word. Any +ve short value.

usStatusWord

CANopen DS402 status word. Any +ve short value.

ucCommError

Input axis communication error. +ve character values.

usLastEmcyErrorCode

Last recorded emergency code. usLastEmcyErrorCode is the emergency error code received from the drive.

ulBulkReadParameters

The array of ulBulkReadParameters defined by [NC_MAX_REC_SIGNALS_NUM] represents a set of parameters to be retrieved from the Maestro, with a maximum of 32 parameters.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within the function. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and ***4.9* NC Profiler Error IDs**.

Figure 11-2 describes the function for MMC_PerformBulkRead.

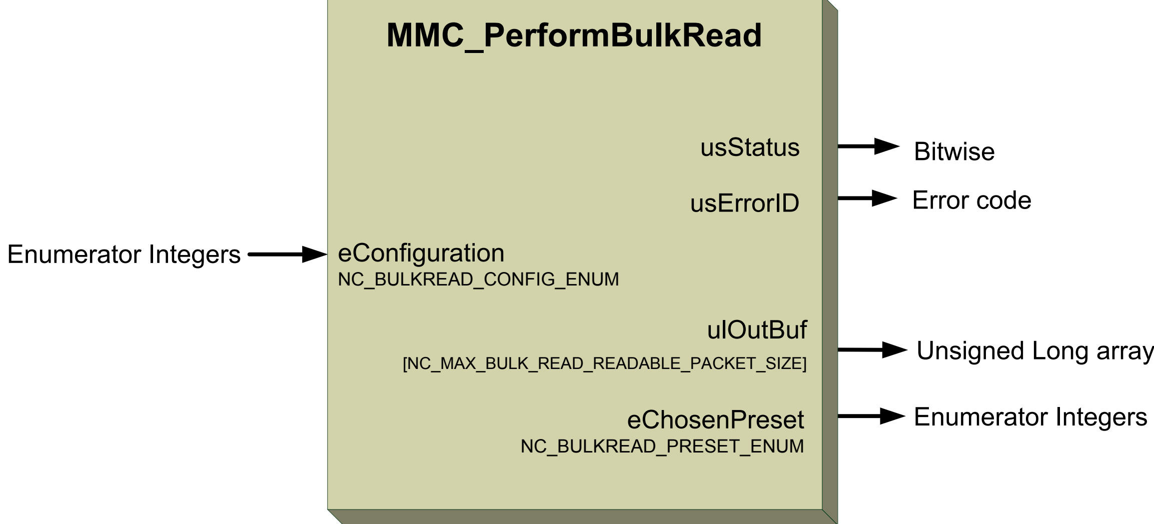


Figure 11-2: MMC_PerformBulkRead function

11.1.2.2 Function Code Example

```
MMC_PERFORMBULKREAD_IN stPerformBulkReadIn;
MMC_PERFORMBULKREAD_OUT stPerformBulkReadOut;
int rc = NC_OK;

stPerformBulkReadIn.eConfiguration = eBULKREAD_CONFIG_2;
rc = MMC_PerformBulkReadCmd(g_hConnectHndl, &stPerformBulkReadIn, &stPerformBulkReadOut);
if (NC_OK != rc)
{
HandleError();
}
```

Chapter 12: API Events

Event handling in the Maestro is the ability to capture specific events occurring within the Maestro, and send Asynchronous Events Callback messages to a host, thus mirroring the occurrences of the event.

Note: **Wherever the Status is produced as a result of the event, it may be set to zero (0) or non-zero.**
If zero, then the event indicates a successful operation
If non-zero, then the event indicates an error whose specific error code can be acquired from the ErrorID.
When this specific error code is sent with the callback message, refer to the Chapter 15: Saving Maestro User Program Parameters for details of the error codes.

The mechanism to handle events in the Maestro involves the following:

- Communication, async replies from the drive, e.g. the function block MMC_SendSDO
- Process progress, notifies the host regarding the progress of a long ongoing process, such as Download Firmware
- Errors in the drive notifications, per node
- PDO3 and PDO4 receive, per node
- System Errors – General system failures (Not yet implemented)
- On Motion End, per node
- On Heartbeat Error, per node
- Emergencies, per node
- Modbus writes from hosts
- Touch Probe event received
- Node Connection Event
- Node Errors
- Axis stopped due to limit
- PVT Underflow data warning
- CAN Node returning to network
- CAN ASYNC reply from drive is ready to be read

This chapter describes the situations when such an event is triggered, the data structure and format of each event.

The treatment of an event is per connection, in the open UDP port. The UDP port is automatically opened by the Maestro function when the MMC_InitConnection function is called and MMC_OpenUdpChannel is invoked. If the MMC_InitConnection and MMC_OpenUdpChannel functions were called with a callback function that is a valid pointer to a callback function, a UDP listening port is automatically created. This UDP port listens for incoming messages, actually on a thread. Once a message is received, the registered callback function is called.

After the MMC_InitConnection function is called, the GMAS_OpenUDPResponceChannel function is called. This function sends the previously opened appropriate port, the command MMC_InitConnection, to the Maestro. In addition to the port, a 32-bit variable is sent, stating the event types that are to be registered in the Maestro.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
														TABLE_UNDERFLOW_EVT	STOP_ON_LIMIT_EVT	NODE_ERROR_EVT	TOUCH_PROBE_ENDED_EVT	MODBUS_WRITE_EVT	SYSTEMERROR_EVT	HOME_ENDED_EVT	EMIT_EVT	DRVERROR_EVT	PDORCV_EVT	HBEAT_EVT	MOTIONENDED_EVT	EMCY_EVT	DOWNLOAD_FW_EVT				ASYNC_REPLY_EVT

Figure 12-1: 32-bit variable

The user sends the appropriate events he wishes to receive as a callback event. There are situations when stating the event is insufficient, and additional function calls must be made. This will be discussed per event type.

12.1 Communication Byte Order

The user must recognize the value of the endianness (little/big endian) of the system on which his program is executed. The message structure of asynchronous call-back buffer, as described hereby refers to data location (offset) of different types. The user may have to convert the endianness, if his program runs on a Windows system.

12.2 Communication ASYNC Replies (Events) From Drives

This event is received if an error (or timeout) occurred when calling an SDO download or Drive Command via the binary interpreter mechanism. Please note that in this situation, unlike other events, the Axis reference, is taken from offset 14 (instead of 12).

The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
0	ASYNC_REPLY_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	14	2	
		Status	Unsigned short	8	2	
		Error ID	Unsigned short	10	2	
		COB ID	Unsigned short	12	2	
		Length	Unsigned char	16	1	Length in bytes
		Data	Unsigned char	17	According to Length	
		Async Event Type	Unsigned char	25	1	

When an asynchronous operation is successful, and this is a response for an inquiry operation (similar to the binary interpreter command "get" which sends a raw data operation or SDO upload request) the status and error ID fields will hold the value of zero, and also data is returned in the data field.

When the operation fails, the status field holds value other than zero and the error code field states a specific error code which defines the occurrence during the operation. If the error occurred during an SDO upload or download, the data field will hold the full CAN message returned from the node.

The Async Event Type field holds values from the Async events numerators group found in the ***MMC_events_API.h*** file. The result of these asynchronous operational events are used to indicate the following:

Response	Response Type
Always Send Response	SendSDO operations (upload \ download)
	Binary interpreter operation Command **Get**
	Read DI Group
	Reset node operation
	Motion mode change
	PDO3\4 operations (mapping, canceling or choosing communication parameter): MMC_CfgRegParamEvPDO3Cmd, MMC_CfgRegParamEvPDO4Cmd, MMC_CancelParamEvPDO3Cmd, MMC_CfgUserParamEvPDO3Cmd, MMC_CancelParamEvPDO4Cmd, MMC_CfgUserParamEvPDO4Cmd, MMC_ChangeDefaultPDOConfiguration
	Virtual encoder configuration
Response is sent only on failure	OS Command
	Binary interpreter operation Command **Set**
	Write DO Group
	Bulk upload
Response is only sent on success (simulates timeout on failure)	Send raw data

12.3 Download Firmware Notifications

This event is received to update the progress of an ongoing download firmware procedure. Please contact to Elmo's representative for further support.

12.4 Emergency Event

Emergency event is triggered by the occurrence of a CANopen device internal error situation.

The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
5	EMCY_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Emergency Error Code	Unsigned short	14	2	As per CiA DS301 (chapter 9.2.5.3)
		MSEF	Unsigned char	16	5	Manufacturer-specific error code
		ER	Unsigned char	15	1	Error register

12.5 Motion Ended Event

Motion ended event is triggered by GMAS when an axis (single as well as group axis) motion is ended. The user must register for this event in order to receive event notification and in addition he must call "MMC_EnableMotionEndedEventCmd" API and specify the particular axis from which he wishes to receive Motion Ended Events. The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
6	MOTIONENDED_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Error ID	short	10	2	

12.6 HeartBeat Event

Heart Beat error event is triggered by Maestro when no heartbeat event was received by Maestro from one of the devices.

The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
7	HBEAT_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	

12.7 PDO Receive Event

This event is triggered by the Maestro when a PDO (3 or 4) for a specific axis is received, and was configured to be sent as notification to the user. Note that the term of the variable located at offset 14 (see below) refers to the Event Group /PDO Number when handling DS 402/401 respectively. The definition of User Data 1 and User Data 2 is Data which is a 32bit float for event groups 15,16. When using MMC_ConfigEventModePDO3\4 with cyclic event mode, the notification is sent on the next GMAS cycle and after it was endian swapped. When using immediate event mode, the data is sent immediately but an endian swap won't be made by GMAS.

The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
8	PDORCV_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Event Group (PDO No. if referring to DS-401) with possible values of 1-17 (Refer to next subsections)	Unsigned char	14	2	Event Group or PDO No. is assigned by the values 1 – 17 and the remaining UDP data is arranged according to the value

12.7.1 Event Group equals to 5 or 6

The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
8	PDORCV_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Event Group	Unsigned char	14	2	= 5 or 6
		User Data 1	short	15	2	Result must be multiplied by rated current.
		User Data 2	long	17	4	

12.7.2 Event Group equals to 11

The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
8	PDORCV_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Event Group	Unsigned char	14	2	= 11
		User Data	Long	15	4	

12.7.3 Event Group Equals to 16 or 17

The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
8	PDORCV_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Event Group	Unsigned char	14	2	= 16 or 17
		User Data	Long long	15	8	

12.7.4 Event Group equals to 1 – 15 besides 5, 6, 11, 16 and 17

The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
8	PDORCV_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Event Group	Unsigned char	14	2	= 1-15 excluding 5,6,11
		User Data 1	Long	15	4	Data is 32bit float for group 15,16.
		User Data 2	long	19	4	

12.8 Home Ended Event

This event is triggered by the Maestro when a Single axis finished the homing procedure. The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
11	HOME_ENDED_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	

12.9 Modbus Write Event

Modbus write event is triggered by Maestro when user writes to Modbus Holding registers. The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
13	MODBUS_WRITE_EVT	Event No.	Unsigned short	0	2	

12.10 Touch Probe Ended Event

Touch Probe event is triggered by Maestro when a touch probe position is received from one of the drives on the network. This is relevant to Ethercat drives only, and drives must be configured appropriately. The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
14	TOUCH_PROBE_ENDED_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Touch probe Position	long	14	2	

12.11 Node Connected Event

Node connected event is triggered by Maestro when an axis is re-connected to the network. This event will occur in two possible situations:

- A node sent a NMT boot-up message after power on
- A node sent a heartbeat message after it was in "heartbeat error state" – a heartbeat error event occurred prior to this event.

In both cases node will be initialized by Maestro. When a node reconnects (second scenario) it will enter error state.The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
18	NODE_CONNECTED_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	

12.12 Node Initialization Completed

The Node initialization completed event indicates whether a successful or unsuccessful node initialization has occurred. This event supplies the user the ending state of the initialization and if an error occurs, an error number is indicated.

The Node initialization is performed only after a node sends boot – up message (Refer to the DS301 documentation for further details), but not after every node connection on the bus. A node connected event will always take place before this event, but a node initialization completed event will not always occur after node connected event. The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
20	NODE_INIT_FINISHED_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Error id	short	10		0 - indicates successful initialization

12.13 Node Error Event

Node error event is triggered when error occurs on one of axes (Single / Group). The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
15	NODE_ERROR_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Error ID	Unsigned short	10	2	The error id field is a bitwise field that indicates what kind of errors have occurred. The error bits are: 0x1 – Status word fault bit rose 0x2 – heartbeat error 0x4 – Emergency message was received 0x8 – Communication error 0x10 - Configuration file error
		Emergency Code	Unsigned short	14	2	

12.14 Stop ON Limit Event

This event is received when axes (single or group) stopped on a software /hardware limit. The UDP Data received is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
16	STOP_ON_LIMIT_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Error ID	Short	10	2	
		Status	Unsigned integer	14	4	
		MSC Limit	Unsigned integer	18	4	

12.15 Table Underflow Event

This event is received when the number of remaining PVT points in the table to be executed by the profiler has reached the minimal limit that was set by the user with MMC_InitTableCmd.

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
17	TABLE_UNDERFLOW_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	

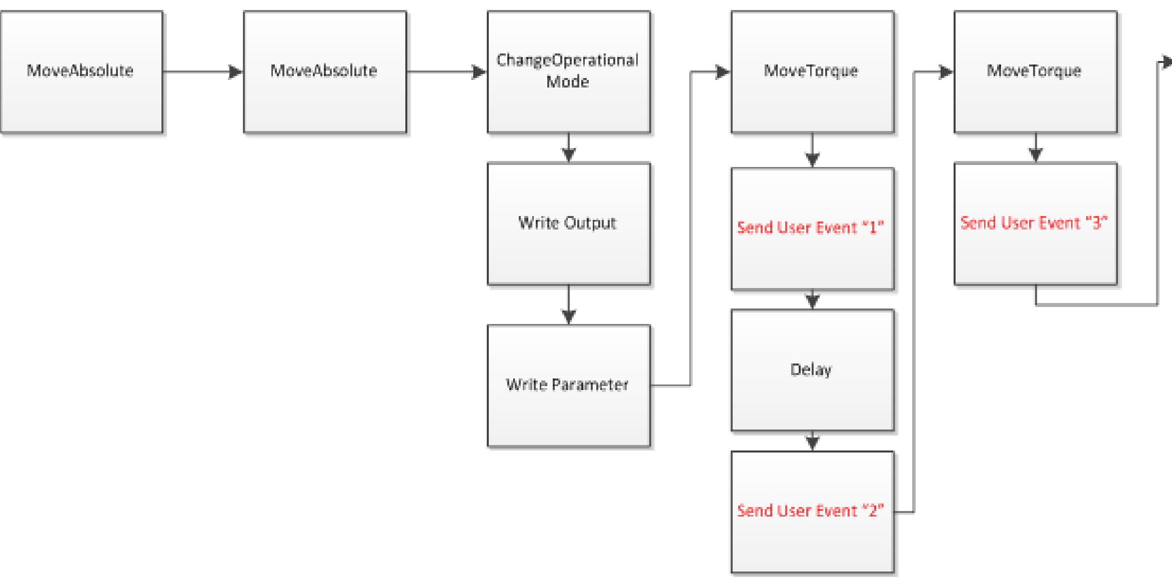
12.16 Global Async Reply Event

This event is received when an asynchronous operation that is not related to a specific node is ended. The event indicates the result of this operation. When in error, the Maestro error code is returned and the status field is different than zero. When successful, the status and error fields are zero.

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
19	GLOBAL_ASYNC_REPLY_EVT	Event No.	Unsigned short	0	2	The global operations that are indicated by this event are: • Set sync time • Set heartbeat consumer command • Reset System end result The enumerators of these events can be found in the MMC_events_API.h header file.
		Status	Unsigned short	8	2	
		Error ID	Short	10	2	
		Function ID	Unsigned char	12	1	

12.17 Notification Function Block Event

When function block MMC_InsertNotificationFb is inserted to a function block queue, a UDP event is generated when called from the function block queue. The UDP event sends a EventID defined by the user,which becomes part of the Motion and Administrative queue.



For details of the function block MMC_InsertNotificationFb, refer to section **12.22.1**.

The event is received when MMC_InsertNotificationFb inserted to the queue is called. When the user enters the function block, an eventID is automatically entered. This may be any user defined Long value (for example id=1 meaning after power on). This value will return in the event data.

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
21	FB_NOTIFICATION_EVT	Event No.	Unsigned short	0	2	
		Axis Ref	Unsigned short	12	2	
		Event code	Long	14	4	

12.18 Policy Ended Event

This category of events' registration is performed separately from regular events mechanism. See the Error handling documentation for further details.

There are two types of sub events:

- System policy ended event
- Node policy ended event

The UDP Data received for these events is as follows:

Event No.	Event Constant Definition	Data				Comment
		Name	**Type**	**Offset**	**Length (bytes)**	
22	POLICY_ENDED_EVT	Event No.	Unsigned short	0	2	The policy type, end state and error type values match the correlating error handling numerators. These event enumerators can be found at MMC_events_API.h and MMC_PLCopen_single_A.h header files
		Error id	short	10	2	0 – indicates that policy ended successfully
		Axis Ref	Unsigned short	12	2	
		Policy type	Unsigned char	14	1	0 – node policy. 1 – system policy.
		Policy end state	Unsigned char	15	1	Indicates in which state the policy failed, or that it has reached the final state

12.19 Communication Event Mechanism

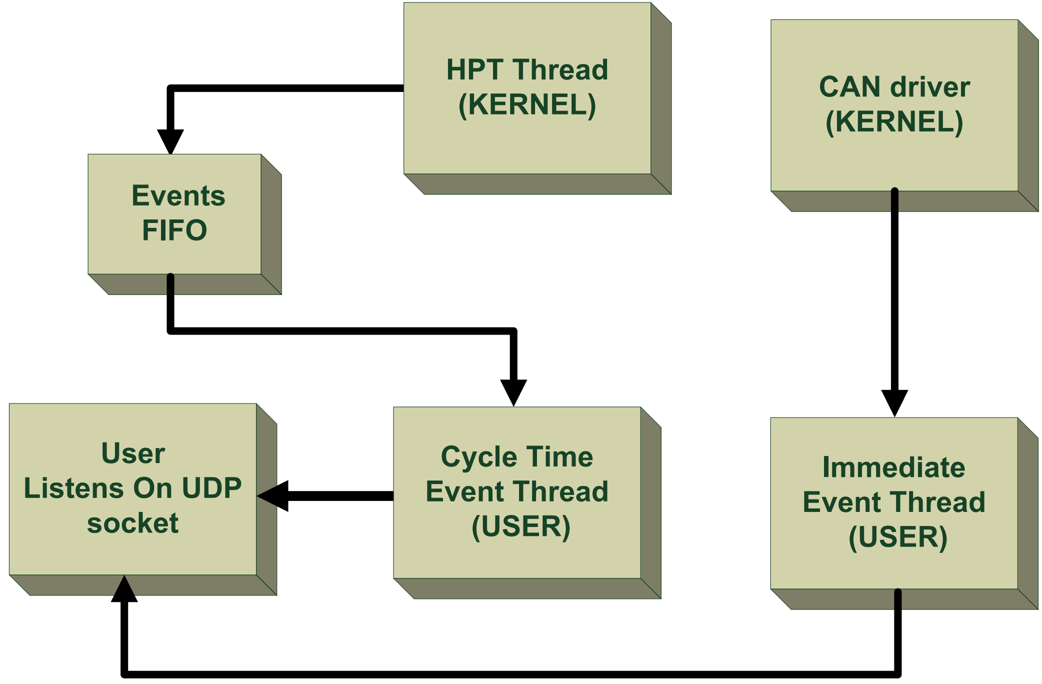


Figure 12-2: Communication events mechanism

Figure 12-2 describes the communication event mechanism in the Maestro. The HPT thread in the kernel is scheduled every basic cycle time and is responsible for populating the Events FIFO in the kernel, when a specific event condition is fulfilled. In the user space (other side), there is a Cycle Time Event thread which awakens every time a new event is entered to the FIFO. This thread processes the new event and sends a UDP message to the registered user on the event. The Cycle Time Event thread is only involved with cycle time based events, i.e. events that are generated in the HPT thread scheduling period.

Another thread in the user space, involving event processing, is the Immediate Event thread. This is responsible for processing events, which require immediate delivery to the registered user. When a condition for an event is fulfilled, at the CAN driver level, the event's data is sent to a special FIFO in the CAN driver. The Immediate Event thread is awakened, when a new event is added to the FIFO. The event is processed and sent via UDP message to the registered user.

It should be noted that the default event mode is 0, No notification to user (default). Therefore, if the event mode is not properly set to cyclic or immediate notification, no event notification will be received.

There are two advantages of the Immediate Event mechanism relative to the Cycle Time Event mechanism:

- Immediate and fast event notification to the user. No need to wait until the next cycle time in order to deliver the event to the user.
- Events can be missed in the Cycle Time Event mechanism, e.g. If two input changes occurred between one cycle and the next, only the last input change event will be delivered to the user, the first input change event is lost. In the Immediate Event mechanism, the above phenomenon does not occur.

12.20 Events Mask and Enumeration

The following lists the Events enumerator and their specific ID, relevant to the parameter *iEventsMask* in C, and *iEventMask* in C++. These events are therefore additionally applicable to C++ events and error states.

Events Mask	Value	Events Enumeration	ID
eEVENT_TYPE_COMM_ASYNC_REPLY	(1 << 0)	ASYNC_REPLY_EVT	0
eEVENT_TYPE_DOWNLOAD_FW	(1 << 4)	DOWNLOAD_FW_EVT	4
eEVENT_TYPE_COMM_EMGCY	(1 << 5)	EMCY_EVT	5
eEVENT_TYPE_COMM_MOTION_ENDED	(1 << 6)	MOTIONENDED_EVT	6
eEVENT_TYPE_COMM_HEARTBEAT_ERROR	(1 << 7)	HBEAT_EVT	7
eEVENT_TYPE_COMM_PDO_RECEIVED	(1 << 8)	PDORCV_EVT	8
eEVENT_TYPE_DRIVE_ERROR	(1 << 9)	DRVERROR_EVT	9
eEVENT_TYPE_EMIT	(1 << 10)	EMIT_EVT	10
eEVENT_TYPE_HOME_ENDED	(1 << 11)	HOME_ENDED_EVT	11
eEVENT_TYPE_SYSTEM_ERROR	(1 << 12)	SYSTEMERROR_EVT	12
eEVENT_TYPE_MODBUS	(1 << 13)	MODBUS_WRITE_EVT	13
eEVENT_TYPE_TOUCH_PROBE_ENDED	(1 << 14)	TOUCH_PROBE_ENDED_EVT	14
eEVENT_TYPE_NODE_ERROR	(1 << 15)	NODE_ERROR_EVT	15
eEVENT_TYPE_STOP_ON_LIMIT	(1 << 16)	STOP_ON_LIMIT_EVT	16
eEVENT_TYPE_TABLE_UNDERFLOW	(1 << 17)	TABLE_UNDERFLOW_EVT	17
eEVENT_TYPE_SEND_ASYNC_EVENT	(1 << 18)		
eEVENT_TYPE_NODE_CONNECTED	(1 << 19)	NODE_CONNECTED_EVT	18
eEVENT_TYPE_GLOBAL_ASYNC_REPLY	(1 << 20)	GLOBAL_ASYNC_REPLY_EVT	19
eEVENT_TYPE_NODE_INIT_FINISHED	(1 << 21)	NODE_INIT_FINISHED_EVT	20
eEVENT_TYPE_FB_NOTIFICATION	(1 << 22)	FB_NOTIFICATION_EVT	21
eEVENT_TYPE_NODE_POLICY	(1 << 23)	POLICY_ENDED_EVT	22
Note that a policy ended event will only be sent according to the registered policy without consideration of the "value" field			

12.21 Asynchronous Events Callback

This section describes the buffer data structure (pBuff character), which is provided by the Maestro together with the Event enumerator described in section 12.20 above, as the first input parameter to the callback function. To use asynchronous events callback, it is necessary to implement the mechanism for Callback registration (via MMC_IPCInitConnection, or MMC_RPCInitConnection) and PDO configuration. The user must pre-configure the system to receive Callbacks, e.g. to perform Homing or motion functions, configure PDO's etc.

In addition to an async reply event being received on an error regardless of the event mask, the SEND_ASYNC_EVENT event mask is set. This event triggers the callback function even when the async function has completed successfully (error = 0).

12.21.1 Callback Prototype

The Type definition (MMC_CB_FUNC) is part of the MMC_Definitions.h header file, which also contains the integer CallbackFunc(unsigned char* pBuff, short sSize, void* pIPSock) function.

pBuff

Buffer consisting of all data related to a specific event

sSize

Buffer size (in bytes). +ve numeric format.

pIPSock

The IP socket used. This should not be changed or edited.

12.21.2 Data Structure

Each event sends a buffer with a different data structure. To extract the relevant data from the buffer according to the Event Type, use the following definitions. These definitions are only present in the CPP header file MMCConnection.h.

12.21.3 Event Extraction Example

This user implementation is only for Illustration purposes, but describes specifically how the *pBuff* data in the *xCallbackFunc* function is extracted according to the Event Type, which describes where the data is situated and exactly what the data consists of. For the various local Windows, or .NET select the NetToLocal and endian_swaps below.

The following offset alias table will help in the understanding of the code described below.

Offset Alias	Index
EVENT_ID_INDX	0
AXIS_REF_INDX	12
ASYNC_EVENT_LEN_INDX	14
PDO_GROUP_INDX	14

Offset Alias	Index
PDO_DATA_INDX	15
MSG_DATA_INDX	8
EMGCY_LEN_INDX	4
EMGCY_DATA_INDX	14
TOUCHP_POS_INDX	14

User implementation function Events ID and other type definitions will compile correctly since they employ the same user MMC_DEFINITIONS.H header file.

```
int UserImplementation(unsigned short usAxisRef,...) {/*do yours*/}
int xCallbackFunc(unsigned char* pBuff, short sSize, void* pIPSock) {
        unsigned short usAxisRef;
        unsigned short usStatus;
        unsigned short usErrorId;
        unsigned short usEventID;
        unsigned short usCobID;
        unsigned short usDataLen;
        unsigned short usEventGrp = 0;
        unsigned short usEmergencyCode;
        unsigned long ulData1;
        unsigned long ulData2;
        unsigned char ucEventType;
        MMC_CAN_REPLY_DATA_OUT* pCanReply;
        NetToLocal((void *)&pBuff[EVENT_ID_INDX], &usEventID);
        usAxisRef = (unsigned short)(*(unsigned short*)(&pBuff[AXIS_REF_INDX]));
#ifdef WIN32
endian_swap16(&(usAxisRef));
#endif
switch(usEventID)
{
case ASYNC_REPLY_EVT:
            pCanReply = (MMC_CAN_REPLY_DATA_OUT*)pBuff;
            usStatus = pCanReply->usStatus;
            usErrorId = pCanReply->usErrorid;
            usCobID = pCanReply->usCOB_ID;
            usAxisRef = pCanReply->usAxisRef;
#ifdef WIN32
        endian_swap16(&(usStatus));
```

```
            endian_swap16(&(usErrorId));
            endian_swap16(&(usCobID));
#endif
            usDataLen = pBuff[ASYNC_EVENT_LEN_INDX];
            UserImplementation(usAxisRef, usStatus, usErrorId, usCobID, usDataLen, pBuff);
break ;
case EMCY_EVT:
                UserImplementation(
usAxisRef, (*(unsigned short *)&pBuff[EMGCY_DATA_INDX])); //send axis ref end
    emergency code
break ;
case MOTIONENDED_EVT:
                NetToLocal((void *)&pBuff[MSG_DATA_INDX+2], &usErrorId);
                UserImplementation(usAxisRef, usErrorId == 0); //send axis ref end OK or not
break;
case HBEAT_EVT:
                UserImplementation(usAxisRef); //send axis ref
break ;
case PDORCV_EVT:
                usEventGrp = pBuff[PDO_GROUP_INDX];
    switch (usEventGrp)
{
            case 1:
            case 2:
            case 3:
            case 4:
            case 7:
            case 8:
            case 9:
            case 10:
            case 12:
            case 13:
            case 14:
            case 15:
    NetToLocal((void *)&pBuff[PDO_DATA_INDX], (void *)&ulData1); //type casting as needed
    NetToLocal((void *)&pBuff[PDO_DATA_INDX+4], (void *)&ulData2); //type casting as needed
    break;
            case 5:
            case 6:
            NetToLocal((void *)&pBuff[PDO_DATA_INDX], &ulData1); //type casting as needed
```

```
            NetToLocal((void *)&pBuff[PDO_DATA_INDX+2], (void *)&ulData2); //type casting as needed
        break;
            case 11:
                NetToLocal((void *)&pBuff[PDO_DATA_INDX], (void *)&ulData1);
            ulData2 = 0; //irrelevance
        break;
            default:
        break;
 }
        UserImplementation(usAxisRef, usEventGrp, ulData1, ulData2);
        break ;
            case HOME_ENDED_EVT:
                NetToLocal((void *)&pBuff[MSG_DATA_INDX+2], &usErrorId);
                UserImplementation(usAxisRef, usErrorId); //send axis ref and error
        break ;
            case MODBUS_WRITE_EVT:
            //<UserImplementation();>
        break ;
            case TOUCH_PROBE_ENDED_EVT:
                UserImplementation( usAxisRef, *((long*)(&pBuff[TOUCHP_POS_INDX])));
        break ;
case NODE_ERROR_EVT:
                NetToLocal((void *)&pBuff[MSG_DATA_INDX+2], &usErrorId);
                NetToLocal((void *)&pBuff[MSG_DATA_INDX+6], &usEmergencyCode);
                UserImplementation(usAxisRef,usErrorId,usEmergencyCode); //send axis ref end OK or
            not
        break ;
default:
        break ;
            case STOP_ON_LIMIT_EVT:
            TBD
        break ;
case TABLE_UNDERFLOW_EVT:
TBD
        break ;
case NODE_CONNECTED_EVT:
TBD
        break ;
            case GLOBAL_ASYNC_REPLY_EVT:
            unsigned char ucFuncID = *((unsigned char*)&buffer[12]);
```

```
            UserImplementation(ucFuncID,usErrorId,usStatus);
        break ;
            int fnCallback(unsigned char* ucBuffer, short sReqID, void* pSock)
{
        unsigned char ucEventID = ucBuffer[2];
        if (ucBuffer[0] == 20) //Event No. is EIP_EVENT
        switch (ucEventID)
        {
case NM_REQUEST_RESPONSE_RECEIVED:
//    printf("NM_REQUEST_RESPONSE_RECEIVED: sReqID = %d\n", sReqID);
                break;
            case NM_ASSEMBLY_NEW_INSTANCE_DATA:
//    printf("NM_ASSEMBLY_NEW_INSTANCE_DATA: assembly instance = %d\n", sReqID);
                break;
            case NM_ASSEMBLY_NEW_MEMBER_DATA:
                //New data received for the specified assembly member. sReqID contains assembly instance.
                printf("NM_ASSEMBLY_NEW_MEMBER_DATA: assembly instance = %dn", sReqID;
                break;
            case NM_REQUEST_FAILED_INVALID_NETWORK_PATH:
                break;
            case NM_REQUEST_TIMED_OUT:
        printf("NM_REQUEST_TIMED_OUT: sReqID = %d\n", sReqID);
                break;
            case NM_CONNECTION_ESTABLISHED:
//              printf("New connection opened with instance %d\n", sReqID);
                break;
            case NM_CONNECTION_VERIFICATION:
                printf("NM_CONNECTION_VERIFICATION\n");
                break;
            case NM_CONNECTION_RECONFIGURED:
        printf("NM_CONNECTION_RECONFIGURED\n");
        break;
            case NM_CONNECTION_TIMED_OUT:
//              printf("Connection with instance %d timed out\n", sReqID);
                break;
            case NM_CONNECTION_CLOSED:
//printf("Connection with instance %d closed\n", sReqID);
                break;
            case NM_CLIENT_OBJECT_REQUEST_RECEIVED:
                printf("NM_CLIENT_OBJECT_REQUEST_RECEIVED\n");
                break;
```

```
        case NM_PENDING_REQUESTS_LIMIT_REACHED:
            printf("NM_PENDING_REQUESTS_LIMIT_REACHED\n");
            break;
        default:
            printf("%s Unhandled(unknown) response event. %d\n", __func__, ucEventID);
            break;
    }
    return 0;
}
```

12.21.4 Net To local Conversion

The following code was extricated from the C++ library, and is displayed here to describe the principle. It is however, necessary to create a customized implementation for these functions/macros.

```
inline void NetToLocal(void* NetBuff, unsigned short *usVal)
{
	memcpy((unsigned char*)usVal,(unsigned char*)NetBuff, 2);
	#ifdef WIN32
		endian_swap16((unsigned short *)usVal);
	#endif
}
inline void NetToLocal(void* NetBuff, void *iVal)
{
	memcpy((unsigned char*)iVal,(unsigned char*)NetBuff,4);
	#ifdef WIN32
		endian_swap32((unsigned int *)iVal);
	#endif
}
inline void endian_swap16(unsigned short* x)
{*x = (*x>>8) | (*x<<8);}
inline void endian_swap32(unsigned int* x)
{*x = (*x>>24) | ((*x<<8) & 0x00FF0000) |((*x>>8) & 0x0000FF00) | (*x<<24);
}
```

12.22 Notification and Events Function Blocks

The following Notification and event function blocks are described:

Notification and Events
MMC_InsertNotificationFb
MMC_ClearEventsMask
MMC_DisableMotionEndedEvent
MMC_EnableMotionEndedEvent
MMC_GetEventsMask
MMC_SetEventsMask

12.22.1 MMC_InsertNotificationFb

Inserts a notification function block within a queue to trigger an event. For details refer to section 12.17

```
MMC_LIB_API int MMC_InsertNotificationFb(
IN  MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_INSNOTIFICATIONFB_IN* pInParam,
OUT MMC_INSNOTIFICATIONFB_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed – Not Supported

Source GMAS\includes\MMC_events_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_INSNOTIFICATIONFB_IN** input data structure using the MMC_InsertNotificationFb function.

pOutParam

Points to the **MMC_INSNOTIFICATIONFB_OUT** output structure receiving information as a result of calling the MMC_InsertNotificationFb function.

Remarks

None

Scope

All

MMC_INSNOTIFICATIONFB_IN Structure

```
typedef struct mmc_insnotificationfb_in{
int iEventCode;
long lSpare[8];
}MMC_INSNOTIFICATIONFB_IN;
```

Parameters

iEventCode

This value will be received in the notification event data. Any int value.

lSpare[8]

For internal use only. 8 bits code data reserved.

MMC_INSNOTIFICATIONFB_OUT Structure

```
typedef struct mmc_insnotificationfb_out{
unsigned int uiHndl;
unsigned short usStatus;
short sErrorID;
}MMC_INSNOTIFICATIONFB_OUT;
```

Parameters

uiHndl

Returned function block handle. Any +ve value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 12-4 describes the function block for MMC_InsertNotificationFb

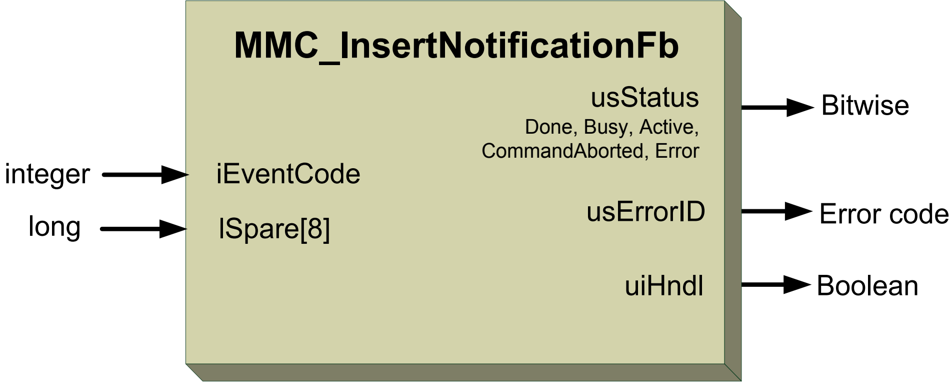


Figure 12-3: MMC_InsertNotificationFb function block

12.22.1.2 Function Block Code Example

```
int rc;
MMC_INSNOTIFICATIONFB_IN pInParam;
MMC_INSNOTIFICATIONFB_OUT pOutParam;

pInParam.iEventCode = iEventCode;

if(MMC_InsertNotificationFb(hConn,aRef,&pInParam,&pOutParam)!=0)
{
printf("MMC_InsertNotificationFb error id %d\n",(short)pOutParam.sErrorID);
     }
Receive event in callback:
unsigned short usAxisRef;
         long Data=0;
usAxisRef = (unsigned short)(*(unsigned short*)(&recvBuffer[12]));
Data=*((long*)(&recvBuffer[14]));

..

case FB_NOTIFICATION_EVT:
       printf("usAxisRef = %d\n",usAxisRef);
    printf("Data = %ld\n",Data);
                 break;
```

12.22.2 MMC_ClearEventsMask

Clears the events mask for a specific connection depending to the input mask.

```
MMC_LIB_API int MMC_ClearEventsMaskCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_CLEAREVENTSMASK_IN* pInParam,
OUT MMC_CLEAREVENTSMASK_OUT* pOutParam
);
```

Motion Mode	NC – Supported	Distributed - Supported
Source	GMAS\includes\MMC_events_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CLEAREVENTSMASK_IN** input data structure using the MMC_ClearEventsMask function.

pOutParam

Points to the **MMC_CLEAREVENTSMASK_OUT** output structure receiving information as a result of calling the MMC_ClearEventsMask function.

Remarks

This involves zeroing the final event data integer of a 32-bit event data with the result of removing the event from the Maestro. The event to be cleared will depend on the ID event input of the parameter *iEventsMask*.

Scope

All

MMC_CLEAREVENTSMASK_IN Structure

```
typedef struct{
int iEventsMask;
}MMC_CLEAREVENTSMASK_IN;
```

Parameters

iEventsMask

Defined according to the event IDs described in the section 12.20 **Events Mask** and Enumeration **on page 1093**. Bitwise +ve integer ID.

MMC_CLEAREVENTSMASK_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CLEAREVENTSMASK_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 12-4 describes the function block for MMC_ClearEventsMask

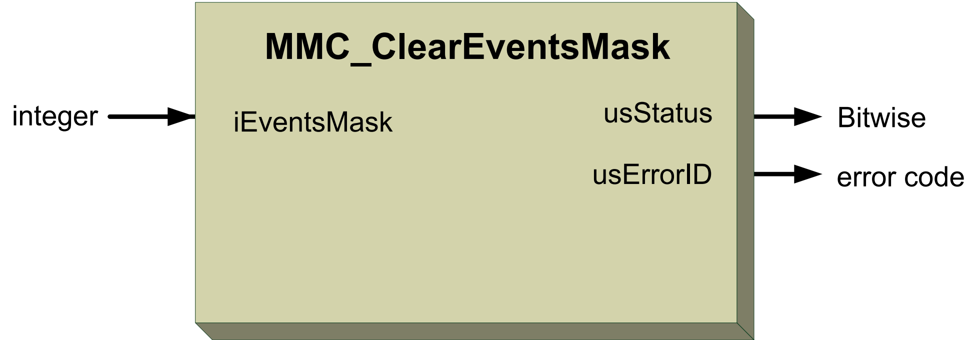


Figure 12-4: MMC_ClearEventsMask function block

12.22.2.2 Function Block Code Example

```
int rc;
MMC_CLEAREVENTSMASK_IN      stClearEventsMask_in;
MMC_CLEAREVENTSMASK_OUT     stClearEventsMask_out;
//
// Inserting the structure parameters:
stClearEventsMask_in.iEventsMask         = 64; //Events mask ID 7 bit 7 is on 1000000 = 64(Dec)
//
rc = MMC_ClearEventsMaskCmd (hConn, &stClearEventsMask_in, &stClearEventsMask_out);
if (rc != 0)
{
    HandleError();
}
```

12.22.3 MMC_DisableMotionEndedEvent

Disables the motion ended event mechanism for a specific node, and no feedback is sent from the Maestro regarding the progress of the motion.

```
MMC_LIB_API int MMC_DisableMotionEndedEventCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_DISABLEMOTIONENDEDEVENT_IN* pInParam,
OUT MMC_DISABLEMOTIONENDEDEVENT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_events_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_DISABLEMOTIONENDEDEVENT_IN** input data structure using the MMC_DisableMotionEndedEvent function.

pOutParam

Points to the **MMC_DISABLEMOTIONENDEDEVENT_OUT** output structure receiving information, as a result of calling the MMC_DisableMotionEndedEvent function.

Remarks

None

Scope

All

MMC_DISABLEMOTIONENDEDEVENT_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_DISABLEMOTIONENDEDEVENT_IN;
```

Parameters

dummy

Dummy input. Any +ve character value.

MMC_DISABLEMOTIONENDEDEVENT_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_DISABLEMOTIONENDEDEVENT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 12-5 describes the function block for MMC_DisableMotionEndedEvent

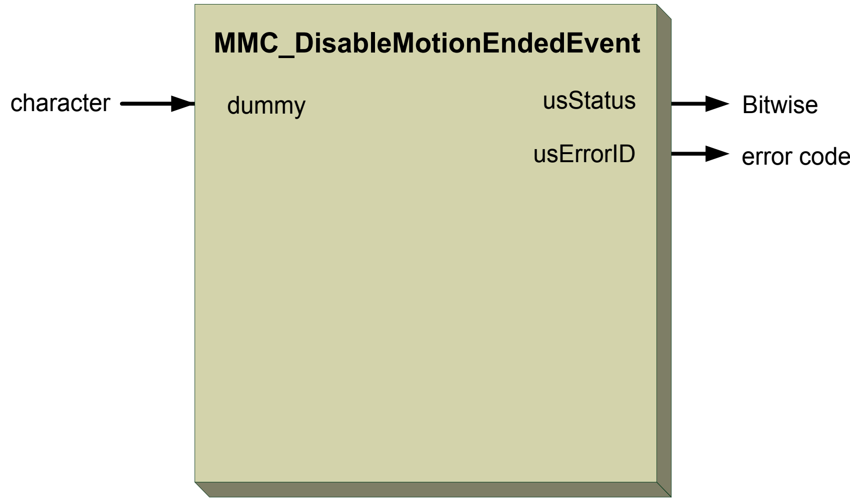


Figure 12-5: MMC_DisableMotionEndedEvent function block

12.22.3.2 Function Block Code Example

```
int rc;
MMC_DISABLEMOTIONENDEDEVENT_IN      stDisableMotionEndEvent_in;
MMC_DISABLEMOTIONENDEDEVENT_OUT     stDisableMotionEndEvent_out;
//
// Inserting the structure parameters:
stDisableMotionEndEvent_in.dummy        = 1;     //Dummy input
//
rc = MMC_DisableMotionEndedEventCmd (hConn, iAxisRef, &stDisableMotionEndEvent_in,
&stDisableMotionEndEvent_out);
if (rc != 0)
{
    HandleError();
}
```

12.22.4 MMC_EnableMotionEndedEvent

Enables the motion ended event mechanism for a specific node.

MMC_LIB_API int MMC_EnableMotionEndedEventCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_ENABLEMOTIONENDEDEVENT_IN* pInParam,
OUT MMC_ENABLEMOTIONENDEDEVENT_OUT* pOutParam
);

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_events_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_ENABLEMOTIONENDEDEVENT_IN** input data structure using the MMC_EnableMotionEndedEvent function.

pOutParam

Points to the **MMC_ENABLEMOTIONENDEDEVENT_OUT** output structure receiving information, as a result of calling the MMC_EnableMotionEndedEvent function.

Remarks

Events are sent regularly from the Maestro regarding the status of the motion.

Scope

All

MMC_ENABLEMOTIONENDEDEVENT_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_ENABLEMOTIONENDEDEVENT_IN;
```

Parameters

dummy

Dummy input. Any +ve character value.

MMC_ENABLEMOTIONENDEDEVENT_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_ENABLEMOTIONENDEDEVENT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 12-6 describes the function block for MMC_EnableMotionEndedEvent

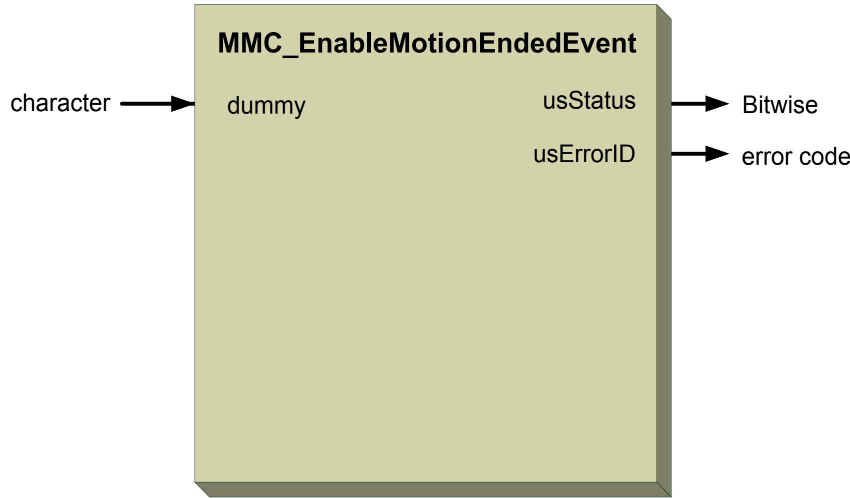


Figure 12-6: MMC_EnableMotionEndedEvent function block

12.22.4.2 Function Block Code Example

```
int rc;
MMC_ENABLEMOTIONENDEDEVENT_IN       stEnableMotionEndEvent_in;
MMC_ENABLEMOTIONENDEDEVENT_OUT      stEnableMotionEndEvent_out;
//
// Inserting the structure parameters:
stEnableMotionEndEvent_in.dummy      = 1; //Dummy input
//
rc = MMC_EnableMotionEndedEventCmd (hConn, iAxisRef, &stEnableMotionEndEvent_in,
&stEnableMotionEndEvent_out);
if (rc != 0)
{
    HandleError();
}
```

12.22.5 MMC_GetEventsMask

Returns the 32 bit events mask for a specific connection.

```
MMC_LIB_API int MMC_GetEventsMaskCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETEVENTSMASK_IN* pInParam,
OUT MMC_GETEVENTSMASK_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_events_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETEVENTSMASK_IN** input data structure using the MMC_GetEventsMask function.

pOutParam

Points to the **MMC_GETEVENTSMASK_OUT** output structure receiving information, as a result of calling the MMC_GetEventsMask function.

Remarks

None

Scope

All

MMC_GETEVENTSMASK_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_GETEVENTSMASK_IN;
```

Parameters

dummy

Dummy input. Any +ve character value.

MMC_GETEVENTSMASK_OUT Structure

```
typedef struct{
int iEventsMask;
unsigned short usStatus;
short sErrorID;
}MMC_GETEVENTSMASK_OUT;
```

Parameters

iEventsMask

Defined according to the event IDs described in the section **12.20 Events Mask and Enumeration on page 1093.** +ve integer Bitwise ID values.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 12-7 describes the function block for MMC_GetEventsMask

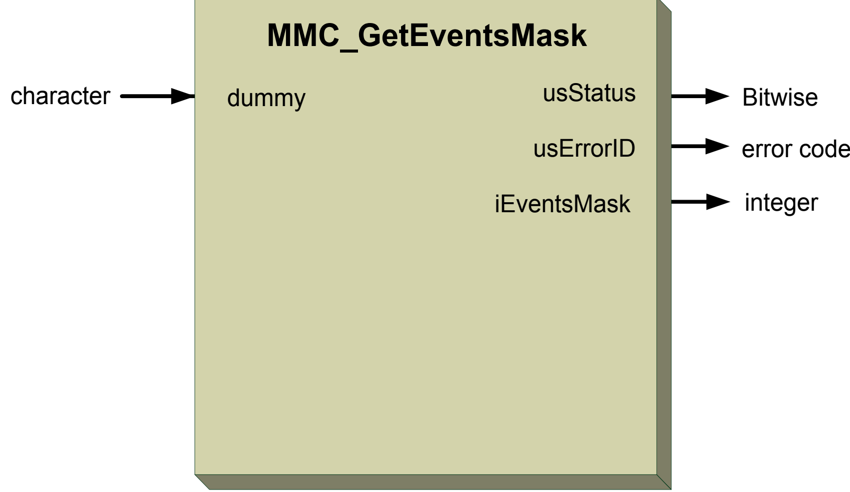


Figure 12-7: MMC_GetEventsMask function block

12.22.5.2 Function Block Code Example

```
int rc;
MMC_GETEVENTSMASK_IN        stGetEventsMask_in;
MMC_GETEVENTSMASK_OUT       stGetEventsMask_out;
//
// Inserting the structure parameters:
stGetEventsMask_in.dummy        = 1;      //Dummy input
//
rc = MMC_GetEventsMaskCmd (hConn, &stGetEventsMask_in, &stGetEventsMask_out);
printf("Events Mask Status[%ld] ErrId[%d]\n", (long int)stGetEventsMask_out.iEventsMask,
(short)stGetEventsMask_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

12.22.6 MMC_SetEventsMask

Sets the 32-bit events mask for a specific connection defined by the input mask parameter *iEventsMask*.

```
MMC_LIB_API int MMC_SetEventsMaskCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SETEVENTSMASK_IN* pInParam,
OUT MMC_SETEVENTSMASK_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_events_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SETEVENTSMASK_IN** input data structure using the MMC_SetEventsMask function.

pOutParam

Points to the **MMC_SETEVENTSMASK_OUT** output structure receiving information, as a result of calling the MMC_SetEventsMask function.

Remarks

None

Scope

All

MMC_SETEVENTSMASK_IN Structure

```
typedef struct{
int iEventsMask;
}MMC_SETEVENTSMASK_IN;
```

Parameters

iEventsMask

Defined according to the event IDs described in the section 12.20 **Events Mask** and Enumeration **on page 1093.** +ve integer Bitwise ID values.

MMC_SETEVENTSMASK_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETEVENTSMASK_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 12-8 describes the function block for MMC_SetEventsMask

Figure 12-8: MMC_SetEventsMask function block

12.22.6.2 Function Block Code Example

```
int rc;
MMC_SETEVENTSMASK_IN        stSetEventsMask_in;
MMC_SETEVENTSMASK_OUT       stSetEventsMask_out;
//
// Inserting the structure parameters:
stSetEventsMask_in.iEventsMask  = 7;      //Dummy input
//
rc = MMC_SetEventsMaskCmd (hConn, &stSetEventsMask_in, &stSetEventsMask_out);
if (rc != 0)
{
    HandleError();
}
```

Chapter 13: Error Correction Mechanism

This chapter describes the correction mechanism to correct non-linear mechanical position errors. The correction is performed by calculating pre-defined correction values at discrete position points (as measurement grid points) prior to the motion, creating an error correction table in the Maestro. In order to compute the error correction value at a given point, a linear interpolation (1D or higher) between two adjacent points in the error correction table is applied, and the value is added to the position calculated by the profiler. This sum is downloaded to the drive and the position correction reported to the user. This is the actual encoder position, which has not changed.

The Position Correction is performed in the low-level code immediately after retrieving the hardware position values, but just before sending the next target position command. This correction is therefore transparent to the end user and considered normal software operation.

In distributed control architecture, the 1D, 2D and 3D error compensations are implemented at the master controller level in the Maestro, which has overall control of the multiple axes. Therefore, Maestro reads the axes X/Y/Z etc. position data to check the actual position, calculates the Error Compensation Tables, and sends the corrected intermittent target commands via the field bus network.

It is important to note, that when running in NC mode (cyclic/interpolated position), the correction is performed continuously in real time every Sync cycle time, throughout path execution. In distributed motions, (e.g. Profile Position), only the final target position is corrected.

13.1 2-D Error Correction

2-D Error correction refers to every correction point (*x,y)* on a two dimensional grid is defined as a function of any two axes positions. The point actually defines an error for a specific axis. For instance, axis Z correction may be a function of X and Y. X correction, can be a function of X itself and Y position. Therefore:

CorrectedPosition=HardwareReading+ErrorCorrection

where, the *ErrorCorrection* is a function of two position inputs. In general, this may be summed as:

$\gamma = f(\alpha, \beta)$, where, γ is the corrected position of any one of the axes. α, β are inputs to γ, and may be any of the Maestro axes.

In reality, there can be numerous axes in the system (over the field bus). Maestro support four separate (independent) compensation tables, either in 1D, 2D or 3D with a maximum total of six points allowed for all four tables.

The position inputs for the table are the axes position as reported over the field bus. Usually it is the main servo loop position feedback returning from the drive. The Corrected Position per axis, can then be defined as a mapping function on any two different hardware position inputs. This is demonstrated in **Figure 13-1** overpage.

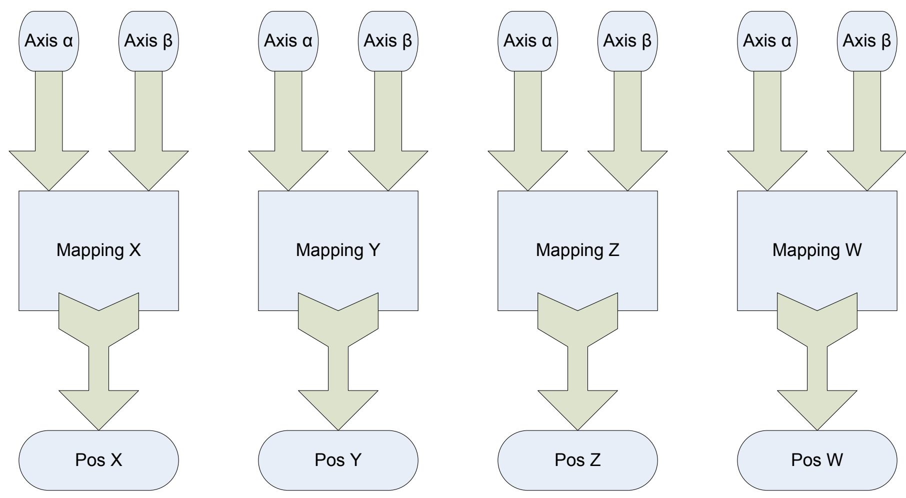


Figure 13-1: Schematic example of the Corrected Position axis mapping function

Looking at a general α, β grid, and we want to calculate the γ (height) from the grid, as a function of the α, β input axes, it would look similar to the Height defined by the Error Correction Function shown in **Figure** 13-2 below.

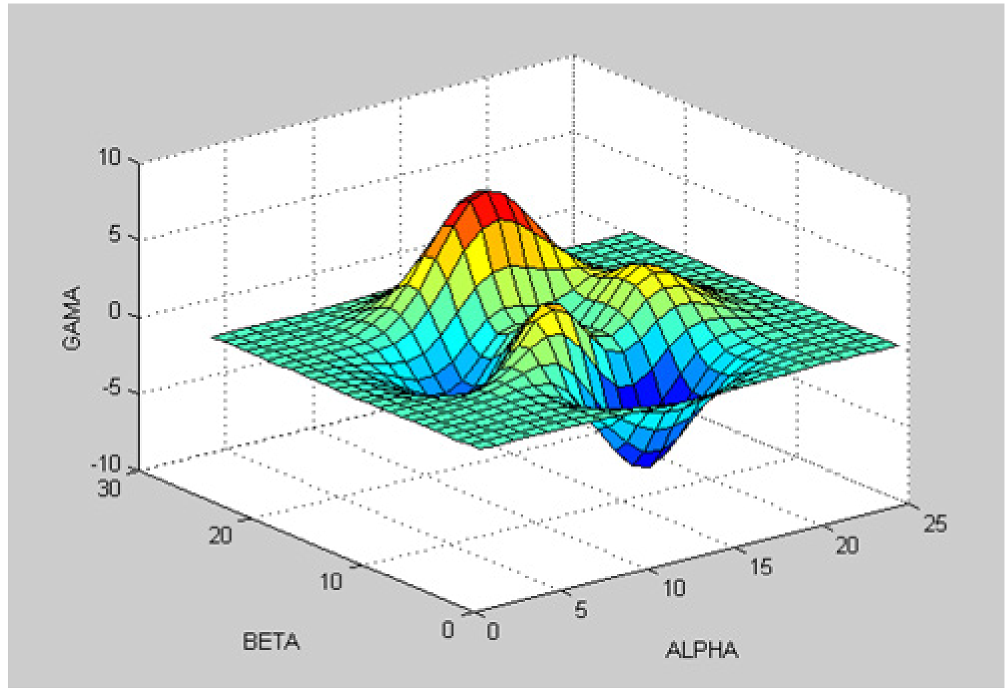


Figure 13-2: Error Correction Function 3-D Graph

13.2 3-D Error Correction

3-D error correction is somewhat similar to 2-D error correction. In 2-D error correction, a 2-D table and grid is defined for any specific two axes. The idea of 3-D error correction is having **multiple layered** grids. As defined for the 2-D mode, the third axis may also be defined as any of the axes, and must be user predefined (as part of the general setup).

Generally, the correction at any point for any specific axis χ, can be defined as a function of any other three axes (α, β, δ):

γ = f (α, β, δ)

For the purpose of a 3-D correction, a set of 2-D corrections, with identical grid points (on the α, β plan) are used. A graphic presentation is shown in **Figure 13-3** below.

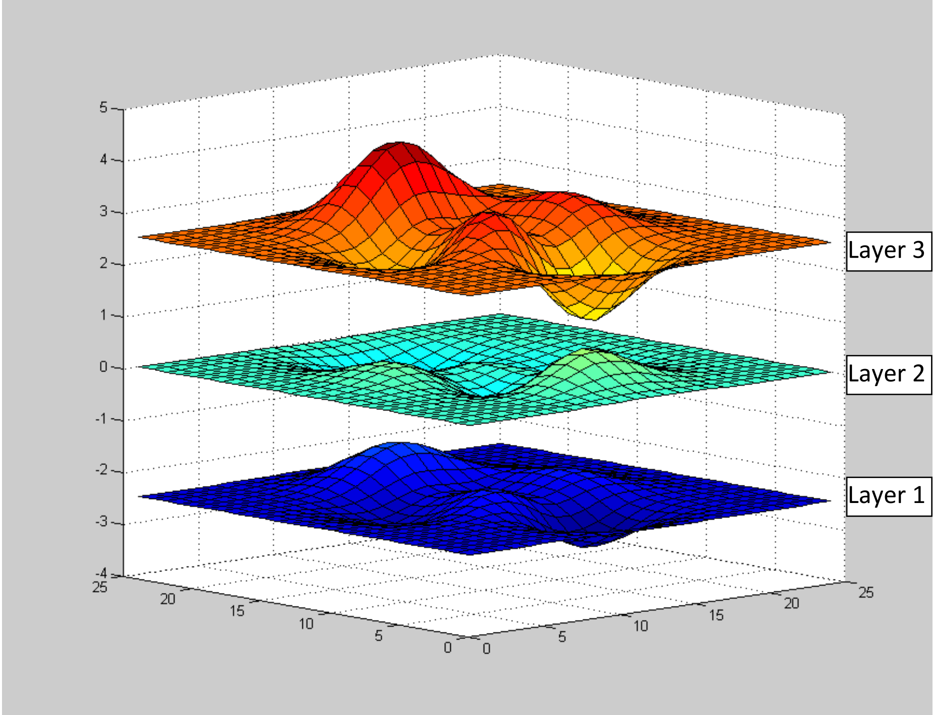


Figure 13-3: Error Correction Function layered 3-D Graph

Figure 13-3 displays an m x n x k dimensional grid, where

k = 3 x 2-D grids of 25 x 25 points each. All 2-D grids have the same definitions.

The method involves locating a third point (correction) called δ, defining the third dimension (Z axis in **Figure** 13-3).

Initially, the index of two relevant 2-D m x n grids we are using is to be located. Actually, this requires searching in the δ axis, between the two 2-D grids we are located. We are therefore searching between Layers 1 and 2 (**Figure** 13-3), performing the calculation using the identical 2-D equations on the two grids where we are positioned in the middle. We wish to calculate 2 x γ's (one per 2-D grid) as described in the previous section for the 2-D correction.

13.3 Data Representation

Data representation in the controller has considerable consequences on the code efficiency.

13.3.1 1-D Representation

The 1D data representation remains intact, and is supported in the Maestro level. The matrix is represented as a linear vector.

9	10	11	12

The Error Correction file is actually a tab-separated text file, which can be created in Excel and takes the following format (example), for say each α and β value above.

Note: **The Header will always starts with [header] and end with [header/].**

[header]

1D Example

Table size	5
Error table dimension	1
Start position	20000
Target axis	a01
Reference axes	a02
Axis grid size	16
Table dimensions	5

[header/]

The final data table appears as shown below:

Note: **The Data area will always starts with [table] and end with [table/].**

The table data can be either integer or float numbers.

[table]

Table #1 Start Data

0 100 0 100 0

Table #1 End Data

[table/]

13.3.2 2-D Representation

The 2-D data representation involves the error correction data for 2-D grid points, stored in the same ET. The matrix is represented as a two dimensional matrix.

9	10	11	12
5	6	7	8
1	2	3	4

Each number in the above matrix represents the value of the error correction in the given (α, β) point.

The Error Correction file is actually a tab-separated text file, which can be created in Excel, and takes the following format (example), for say each α and β value above.

Note: **The Header will always starts with [header] and end with [header/].**

[header]

2D Example

Table size	25		This is the total table size (actual number of points)
Error table dimension	2		Either 1D/2D/3D
Target axis	a01		The axis to be corrected
Reference axes	a02	a03	The input axes to the error correction table
Start position	20000	20000	The start position of the error correction. It may either be an integer of floating point number.
Axis grid size	16	16	The actual resolution between sample points. It may either be an integer of floating point number.
Table dimensions	5	5	Number of rows and columns. In a 3D table, the third parameter will be the number of tables.

[header/]

The final data table appears as shown below:

Note: **The Data area will always starts with [table] and end with [table/].**

The table data can be either integer or float numbers.

[table]

Table #1 Start Data

0	100	0	100	0
0	100	0	100	0
0	100	0	100	0
0	100	0	100	0
0	100	0	100	0

Table #1 End Data

[table/]

13.3.3 3-D Representation

The 3-D data is saved as consecutive 2-D matrices.

9	10	11	12
5	6	7	8
1	2	3	4

Grid#1

21	22	23	24
17	18	19	20
13	14	15	16

Grid#2

etc… for additional grid's.

1. For each axis, the user must define the input axes α, β, δ (of which axis are α, β, δ, from the possible Maestro Axis nodes).
2. For each α, β, and δ value, the following must be inserted (refer to the 2-D Example **on page 1122 (2D Example)**:
    a. Table size
    b. Error table dimension
    c. Target axis a01
    d. Reference axis a02
    e. Start position
    f. Axis grid size
    g. Table dimensions

13.4 Error Correction Functions

The following Error Correction functions are described:

Error Correction functions
MMC_LoadErrorCorrTable
MMC_UnloadErrorCorrTable
MMC_EnableErrorCorrTable
MMC_DisableErrorCorrTable
MMC_GetErrorTableStatus

13.4.1 MMC_LoadErrorCorrTable

Loads an error correction table to memory. Error correction is then performed according to this table.

```
MMC_LIB_API int MMC_LoadErrorCorrTableCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_LOADERRORTABLE_IN* pInParam,
OUT MMC_LOADERRORTABLE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS/includes/MMC_ErrorCorr_API.h GMAS Programming(IEC 61331 Program)\ElmoGlobal	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_LOADERRORTABLE_IN** input data structure using the MMC_LoadErrorCorrTable function.

pOutParam

Points to the **MMC_LOADERRORTABLE_OUT** output structure receiving information, as a result of calling the MMC_LoadErrorCorrTable function.

Remarks

The Error Correction Table is loaded to the memory and requires enabling using the function **MMC_EnableErrorCorrTable**, in order to be functional. If loaded, and enabled, the table can be retained in memory but disabled using the function **MMC_DisableErrorCorrTable**. To change its functionality to enable, perform MMC_EnableErrorCorrTable again. To unload the table from the memory perform the function **MMC_UnloadErrorCorrTable**.

Scope

All

MMC_LOADERRORTABLE_IN Structure

```
typedef struct{
double dMaxCorrectionDelta;
NC_ERROR_TABLE_NUMBER eETNumber;
unsigned char pPathToETFile[NC_MAX_ET_FILE_PATH_LENGTH];
}MMC_LOADERRORTABLE_IN;
```

Parameters

dMaxCorrectionDelta

This parameter define the maximum allowed correction input value. If you try to insert a table where one of the correction values is above the MaxCorrectionDelta, an error is received; **-342**.

If you set this value to "0", the max correction delta is unlimited.

NC_ERROR_TABLE_NUMBER eETNumber

Defines the error table letter assigned.

NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

pPathToETFile

Defines the path to the error table file.

[NC_MAX_ET_FILE_PATH_LENGTH] is the maximum size of the error table file path. It is limited to 100.

If you set "NULL" in this input, the default file path will be used.

The default path is set to the:

/mnt/jffs/usr/ directory and the filename will be depended on the table index:

ErTBL_#.txt where the # is the table index (A,B,C,D,E,F).

MMC_LOADERRORTABLE_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_LOADERRORTABLE_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within the function. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 13-4 describes the function for MMC_LoadErrorCorrTable as applied within the IEC 61131 programming.

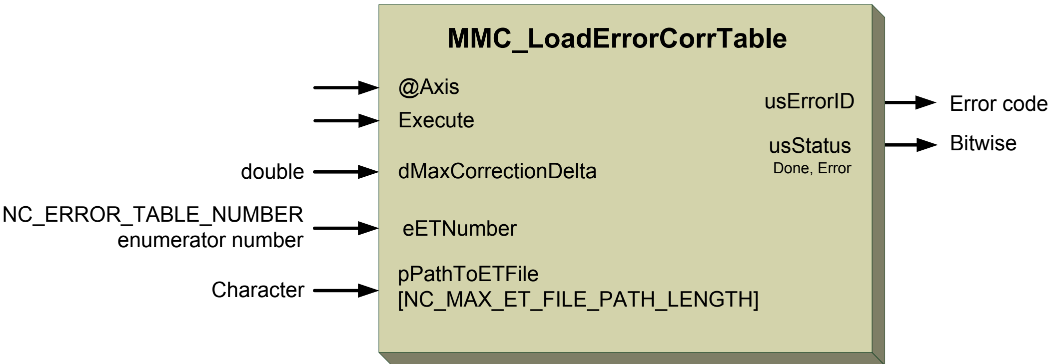


Figure 13-4: MMC_LoadErrorCorrTable function

13.4.1.2 Function Code Example

```
MMC_LOADERRORTABLE_IN      stLoadErrorTableIn;
MMC_LOADERRORTABLE_OUT     stLoadErrorTableOut;

stLoadErrorTableIn.eETNumber            = NC_ERROR_TABLE_A;
strcpy(stLoadErrorTableIn.pPathToETFile, szFileName);

rc = MMC_LoadErrorCorrTableCmd(hConnHndl, &stLoadErrorTableIn, &stLoadErrorTableOut);
if (NC_OK != rc)
{
HandleError();
}
```

13.4.2 MMC_EnableErrorCorrTable

EnableErrorCorrTable

MMC_LIB_API int MMC_EnableErrorCorrTableCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_ENABLEERRORTABLE_IN* pInParam,
OUT MMC_ENABLEERRORTABLE_OUT* pOutParam
);

Motion Mode NC - Supported Distributed - Supported

Source GMAS/includes/MMC_ErrorCorr_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_ENABLEERRORTABLE_IN** input data structure using the MMC_EnableErrorCorrTable function.

pOutParam

Points to the **MMC_ENABLEERRORTABLE_OUT** output structure receiving information, as a result of calling the MMC_EnableErrorCorrTable function.

Remarks

The Error Correction Table is loaded to the memory and requires enabling using the function **MMC_EnableErrorCorrTable**, in order to be functional. If loaded, and enabled, the table can be retained in memory but disabled using the function **MMC_DisableErrorCorrTable**. To change its functionality to enable, perform MMC_EnableErrorCorrTable again. To unload the table from the memory perform the function **MMC_UnloadErrorCorrTable**.

Scope

All

MMC_ENABLEERRORTABLE_IN Structure

```
typedef struct{
NC_ERROR_TABLE_NUMBER eTableNumber;
}MMC_ENABLEERRORTABLE_IN;
```

Parameters

NC_ERROR_TABLE_NUMBER eTableNumber

Defines the error table letter assigned to be enabled.

NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

MMC_ENABLEERRORTABLE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_ENABLEERRORTABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within the function. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 13-5 describes the function for MMC_EnableErrorCorrTable as applied within the IEC 61131 programming.

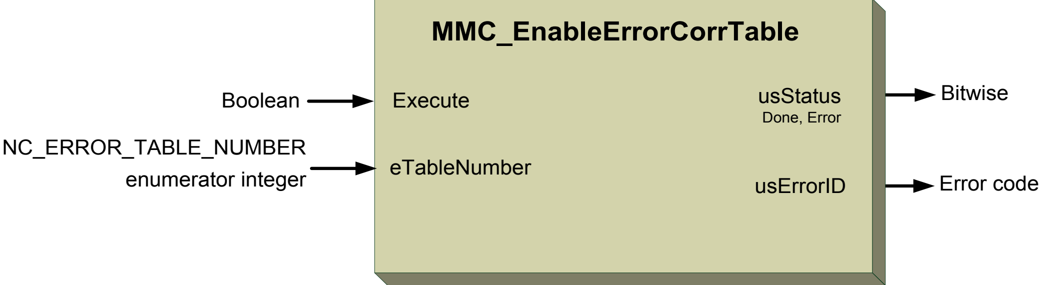


Figure 13-5: MMC_EnableErrorCorrTable function

13.4.2.2 Function Code Example

```
MMC_ENABLEERRORTABLE_IN stEnableErrorTableIn;
MMC_ENABLEERRORTABLE_OUT stEnableErrorTableOut;

stEnableErrorTableIn.eTableNumber = NC_ERROR_TABLE_A;

rc = MMC_EnableErrorCorrTableCmd(hConnHndl, &stEnableErrorTableIn, &stEnableErrorTableOut);
if (NC_OK != rc)
{
HandleError();
}
```

13.4.3 MMC_GetErrorTableStatus

Function recieves the table number as input and returns an answer whether the table is loaded and/or enabled.

```
MMC_LIB_API int MMC_GetErrorTableStatusCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETERRORTABLESTATUS_IN* pInParam,
OUT MMC_GETERRORTABLESTATUS_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS/includes/MMC_ErrorCorr_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETERRORTABLESTATUS_IN** input data structure using the MMC_GetErrorTableStatus function.

pOutParam

Points to the **MMC_GETERRORTABLESTATUS_OUT** output structure receiving information, as a result of calling the MMC_GetErrorTableStatus function.

Remarks

The Error Correction Table is loaded to the memory and requires enabling using the function **MMC_EnableErrorCorrTable**, in order to be functional. If loaded, and enabled, the table can be retained in memory but disabled using the function **MMC_DisableErrorCorrTable**. To change its functionality to enable, perform MMC_EnableErrorCorrTable again. To unload the table from the memory perform the function **MMC_UnloadErrorCorrTable**.

Scope

All

MMC_GETERRORTABLESTATUS_IN Structure

```
typedef struct{
NC_ERROR_TABLE_NUMBER eTableNumber;
}MMC_GETERRORTABLESTATUS_IN;
```

Parameters

NC_ERROR_TABLE_NUMBER eTableNumber

Defines the error table letter assigned.

NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

MMC_GETERRORTABLESTATUS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned char ucIsTableEnabled;
unsigned char ucIsTableLoaded;
NC_NODE_HNDL_T hReferenceAxesRef[NC_ERROR_TABLE_DIMENSION_3D];
NC_NODE_HNDL_T hTargetAxisRef;
char cFileName[NC_MAX_ET_FILE_PATH_LENGTH];
char sSpare[20];
}MMC_GETERRORTABLESTATUS_OUT;
```

Parameters

ucIsTableEnabled

Returns the Boolean solution to the question, whether the table is **enabled** or not.

ucIsTableLoaded

Returns the Boolean solution to the question, whether the table is **loaded** or not.

NC_NODE_HNDL_T hReferenceAxesRef

This array represent the axes references of the error correction table input.

The array [NC_ERROR_TABLE_DIMENSION_3D] is the dimension of the 3D error table.

NC_NODE_HNDL_T hTargetAxisRef

The parameter hTargetAxisRef represents the reference of the target axis.

cFileName[NC_MAX_ET_FILE_PATH_LENGTH]

Defines the file name.

[NC_MAX_ET_FILE_PATH_LENGTH] is the maximum size of the error table file path. It is limited to 100.

sSpare[20]

Spare. For internal use only. Any +ve integer value to a maximum of 20 characters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within the function. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 13-6 describes the function for MMC_GetErrorTableStatus as applied within the IEC 61131 programming.

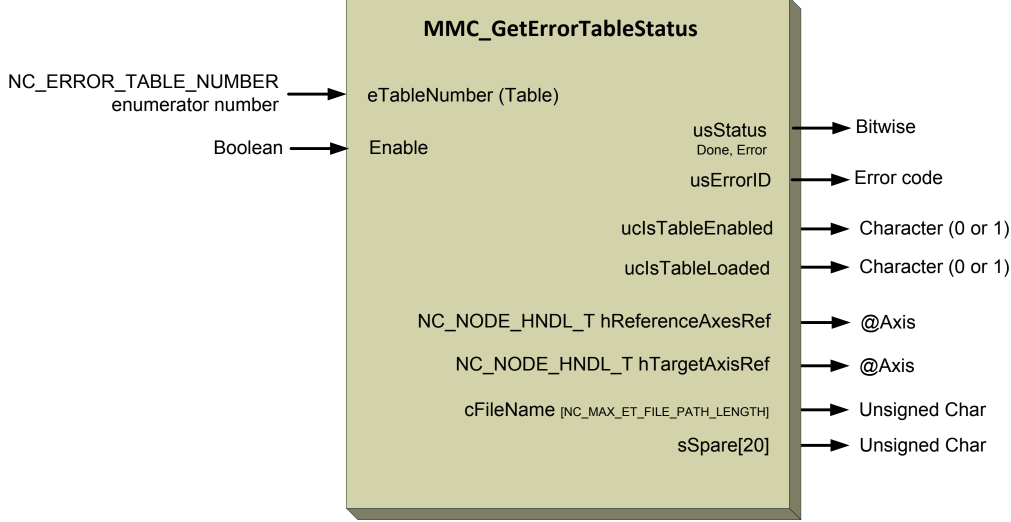


Figure 13-6: MMC_GetErrorTableStatus function

13.4.3.2 Function Code Example

```
MMC_GETERRORTABLESTATUS_IN stGetErrorTableStatusIn;
MMC_GETERRORTABLESTATUS_OUT stGetErrorTableStatusOut;

stGetErrorTableStatusIn.eTableNumber = NC_ERROR_TABLE_A;

rc = MMC_GetErrorTableStatusCmd(hConnHndl, &stGetErrorTableStatusIn,
&stGetErrorTableStatusOut);

if (NC_OK != rc)
{
HandleError();
}
```

13.4.4 MMC_DisableErrorCorrTable

Disables the usage of an error table.

MMC_LIB_API int MMC_DisableErrorCorrTableCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_DISABLEERRORTABLE_IN* pInParam,
OUT MMC_DISABLEERRORTABLE_OUT* pOutParam
);

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS/includes/MMC_ErrorCorr_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoGlobal	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_DISABLEERRORTABLE_IN** input data structure using the MMC_DisableErrorCorrTable function.

pOutParam

Points to the **MMC_DISABLEERRORTABLE_OUT** output structure receiving information, as a result of calling the MMC_DisableErrorCorrTable function.

Remarks

The Error Correction Table is loaded to the memory and requires enabling using the function **MMC_EnableErrorCorrTable**, in order to be functional. If loaded, and enabled, the table can be retained in memory but disabled using the function **MMC_DisableErrorCorrTable**. To change its functionality to enable, perform MMC_EnableErrorCorrTable again. To unload the table from the memory perform the function **MMC_UnloadErrorCorrTable**.

Scope

All

MMC_DISABLEERRORTABLE_IN Structure

```
typedef struct{
NC_ERROR_TABLE_NUMBER eTableNumber;
}MMC_DISABLEERRORTABLE_IN;
```

Parameters

NC_ERROR_TABLE_NUMBER eTableNumber

Defines the error table letter assigned to be disabled.

NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

MMC_DISABLEERRORTABLE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_DISABLEERRORTABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within the function. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 13-7 describes the function for MMC_DisableErrorCorrTable as applied within the IEC 61131 programming.

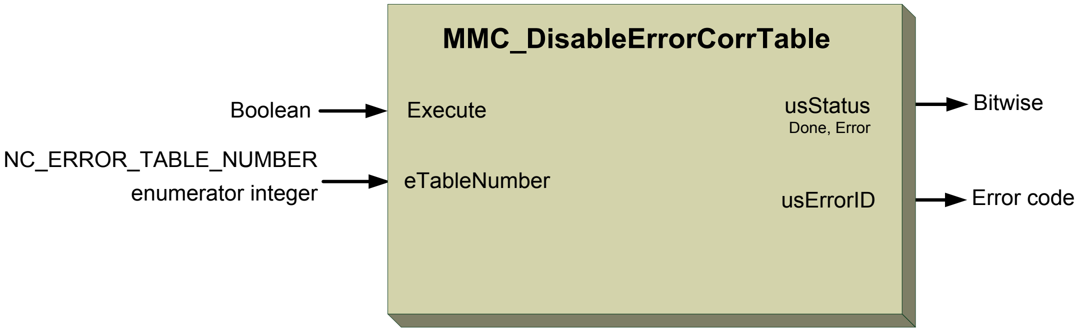


Figure 13-7: MMC_DisableErrorCorrTable function

13.4.4.2 Function Code Example

```
MMC_DISABLEERRORTABLE_IN stDisableErrorTableIn;
MMC_DISABLEERRORTABLE_OUT stDisableErrorTableOut;

stDisableErrorTableIn.eTableNumber = NC_ERROR_TABLE_A;

rc = MMC_DisableErrorCorrTableCmd(hConnHndl, &stDisableErrorTableIn, &stDisableErrorTableOut);
if (NC_OK != rc)
{
HandleError();
}
```

13.4.5 MMC_UnloadErrorCorrTable

Unloads an error correction table from memory.

```
MMC_LIB_API int MMC_UnloadErrorCorrTableCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_UNLOADERRORTABLE_IN* pInParam,
OUT MMC_UNLOADERRORTABLE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS/includes/MMC_ErrorCorr_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_UNLOADERRORTABLE_IN** input data structure using the MMC_UnloadErrorCorrTable function.

pOutParam

Points to the **MMC_UNLOADERRORTABLE_OUT** output structure receiving information, as a result of calling the MMC_UnloadErrorCorrTable function.

Remarks

The Error Correction Table is loaded to the memory and requires enabling using the function **MMC_EnableErrorCorrTable**, in order to be functional. If loaded, and enabled, the table can be retained in memory but disabled using the function **MMC_DisableErrorCorrTable**. To change its functionality to enable, perform MMC_EnableErrorCorrTable again. To unload the table from the memory perform the function **MMC_UnloadErrorCorrTable**.

Scope

All

MMC_UNLOADERRORTABLE_IN Structure

```
typedef struct{
NC_ERROR_TABLE_NUMBER eTableNumber;
}MMC_UNLOADERRORTABLE_IN;
```

Parameters

NC_ERROR_TABLE_NUMBER eTableNumber

Defines the error table letter assigned to be unloaded.

NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

MMC_UNLOADERRORTABLE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_UNLOADERRORTABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within the function. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 13-8 describes the function for MMC_UnloadErrorCorrTable as applied within the IEC 61131 programming.

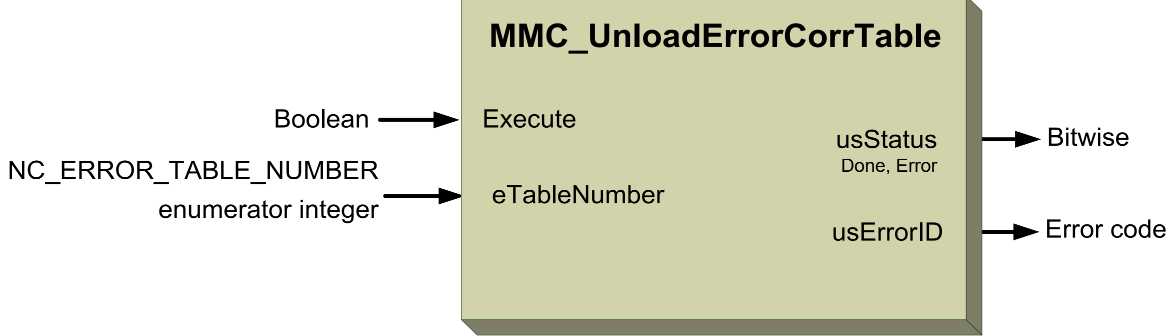


Figure 13-8: MMC_UnloadErrorCorrTable function

13.4.5.2 Function Code Example

```
MMC_UNLOADERRORTABLE_IN     stUnloadErrorTableIn;
MMC_UNLOADERRORTABLE_OUT    stUnloadErrorTableOut;

stUnloadErrorTableIn.eTableNumber = NC_ERROR_TABLE_A;

rc = MMC_UnloadErrorCorrTableCmd(hConnHndl, &stUnloadErrorTableIn, &stUnloadErrorTableOut);

if (NC_OK != rc)
{
HandleError();
}
```

Chapter 14: Maestro Hardware and Software Limits Handling

14.1 Introduction

The Maestro behaves as a Multi Axis Motion controller and regularly updates the system statuses of each drive. However, the user may not wish to see and manage individual drives, only a single interface of the complete system. The disadvantage of this model single interface management, occurs when the behavior of the actual motion of a drive is unexpected, due to the limitations of the drive itself. In this situation the user has no knowledge when an axis or group limit is reached. The profiler continues to maintain its status and handle the axis motion as if no limit exists.

This chapter therefore describes the procedure to manage hardware and software limits to achieve the following:

- Protect the hardware system application from damage or otherwise due to the axis, or group overshooting
- Control the axis position according to the axis /group limits from the Maestro
- Obtain each axis limit status, either when operating as single axes or group axes

14.2 Interfaces

The limit management mechanism involves interfacing with the following:

- Software axis/group position limits
- Stop parameters
- Status Register
- MCS Limit Register
- Stop on Limit event

14.2.1 Software Position Limits

This uses the presently defined parameters mechanism for single axis/group to determine the limit:

Axis Definition	Limits	
ACS – Axis Level	High Limit SW_LIM_HIGH Low Limit SW_LIM_LOW	
MCS – Group Level	High Limit array MCS_SW_LIMIT_HIGH_POS_ARRAY Low Limit array	

Axis Definition	Limits	
	MCS_SW_LIMIT_LOW_POS_ARRAY	

To describe the difference between High and Low limits the following diagrams display the differences between motion Outside a High or Low Limit and Inside the limits.

Inside High Limit

Actual Position

Low Limit

High Limit

Inside Low Limit

Actual Position

Low Limit

High Limit

Out of Limits

Actual Position

Low Limit

High Limit

14.2.2 Status Register

This variable provides information on the special status of an axis. The Status Register represents the status of the axis according to the following definitions:

Bit Value	Feature	Description
Bit 0	Limit handling	HW limit - Reverse Limit Switch (RLS)
Bit 1	Limit handling	HW limit - Forward Limit Switch (FLS)
Bit 2	Limit handling	SW limit - low limit on axis
Bit 3	Limit handling	SW limit - high limit on axis
Bit 4	Limit handling	SW limit - low limit on kinematic direction
Bit 5	Limit handling	SW limit - high limit on kinematic direction
Bit 6	Limit handling	Internal use
Bit 7	Limit handling	Internal use
Bit 8	Limit handling	Internal use
Bit 9	Limit handling	Internal use
Bit 10	Target Radius And Time (TRaT)	Target Reached
Bit 11	Target Radius And Time (TRaT)	Node Settled
Bit 12	Target Radius And Time (TRaT)	Node in StandStill\StandBy
Bit 13	Target Radius And Time (TRaT)	Internal use
Bit 14	Target Radius And Time (TRaT)	Internal use
Bit 15	Tracking Error (TReR)	The axis is in tracking error
Bit 16	AxisLink	Axis linked as Master
Bit 17	AxisLink	Axis linked as Slave
Bit 18	Home Attain	Attaining homing
Bit 19	Not in use	-
Bit 20	Not in use	-
Bit 21	Not in use	-
Bit 22	Not in use	-
Bit 23	Not in use	-
Bit 24	Not in use	-
Bit 25	Not in use	-
Bit 26	Not in use	-
Bit 27	Not in use	-
Bit 28	Not in use	-

Bit 29	Not in use	-
Bit 30	Not in use	-
Bit 31	Not in use	-

To verify the current limit, use one of the following methods:

- Check the function **MMC_GetStatusRegister** (refer to section 8.1.19 for the function details) and look at the uiStatusRegister variable value
- For the Bulk Read mechanism, use presets 4 or 5
- For the Recording mechanism, use the vector NC_STATUS_REGISTER = 66

14.2.3 ACS\SingleAxis

The following bits are relevant for ACS limits:

Register Bit Value	Register Bit Value
Bit0 – RLS	Bit2 - Low
Bit1 – FLS	Bit3 – High

14.2.4 MCS

The following bits are relevant for MCS limits:

Bit Value	Represents
Bit 4	MCS Low limit
Bit 5	MCS High limit

14.2.5 MCS Limit Register (32 bits)

This parameter represents the status of the MCS limits in a specific group. To obtain this parameter, use one of the following methods:

- Check the function **MMC_GetStatusRegister** (refer to section **8.1.19** for the function details) and look at the uiStatusRegister variable value
- For the Bulk Read mechanism (refer to the **Chapter 11: Bulk Parameters Reading**), use the Status Register with presets 4 or 5
- For the Recording mechanism (refer to the section **9.9 PI Functions and Implementation Examples**), use the vector NC_STATUS_REGISTER = 67

Register Bit Value	Register Bit Value	Register Bit Value	Register Bit Value
Bit0 – X Low	Bit7 - U High	Bit14 – N2 Low	Bi21 – N5 High
Bit1 - X High	Bit8 – V Low	Bit15 – N2 High	Bit22 – N6 Low
Bit2 - Y Low	Bit9 - V High	Bit16 – N3 Low	Bit23 – N6 High

Register Bit Value	Register Bit Value	Register Bit Value	Register Bit Value
Bit3 - Y High	Bit10 - W Low	Bit17 – N3 High	Bit24 – N7 Low
Bit4 – Z Low	Bit11 - W High	Bit18 – N4 Low	Bit25 – N7 High
Bit5 – Z High	Bit12 – N1 Low	Bit19 – N4 High	Bit26 – N8 Low
Bit6 - U Low	Bit13 – N1 High	Bit20 – N5 Low	Bit27 - N8 High
Bit28 – N9 Low	Bit29 – N9 High	Bit30 - S Low	Bit31 - S High

14.2.6 Stop Parameters

These parameters used to define the kinematic of the stop procedure when the limit is detected. To obtain and set these parameters, use the Maestro parameters mechanism and the following specific parameters:

- MMC_LIMIT_STOP_DECELERATION (57)
- MMC_LIMIT_STOP_JERK (58)

These parameters can be applied for both, single axis or group.

14.2.7 Stop-on-Limit Event

The special event Stop-on-Limit is created at application level (when the event is registered) to supply the following data:

- Stop\Group_stop error code
- Axis\Group reference
- Status Register
- MCS Limits Register

As a result, an applicable error code is created when that Stop-on-Limit was unsuccessful. Otherwise the Stop/Group error code should return 0.

14.3 Function Block Pre-Insertion Behavior

The behavior of a function block prior to its insertion in the operational queue varies between single and multiple grouped axes.

14.3.1 Single Axis

For a single axis, the function block will automatically check whether the software limits have been reached, and if reached, will then check the allowed direction for the function block operation on the axis.

NOTE: **For optimal performance in Distributed mode (CAN), configure the PDO with the actual position. When the hardware limit is reached in Distributed mode, the state of the axis changes to StandStill.**

The following table describes the alternative behaviour for any single axis function block type described in this document.

Inserted FB type for Single Axis			
Current state	**Admin FB**	**NC - Motion FB**	**NonNC - Motion FB**
Out of limit	Always - √	If the FB is one of this: MoveAbs. MoveAbsRep. MoveRel. MoveRelRep. MoveAdd. MoveAddRep. MoveVel. Target position outside limit - √ Target position inside limit - X For all other FB's – always √	If the FB is one of this: MoveAbs. MoveAbsRep. MoveRel. MoveRelRep. Target position outside limit - √ Target position inside limit – X For all other FB's – always √
Inside limit	Always - √	If the FB is one of this: MoveAbs. MoveRel. MoveAdd. MoveVel. HW limit If RLS – positive direction - √ If RLS – negative direction - X If FLS – positive direction - X If FLS – negative direction - √ SW limit If Low limit – positive direction - √ If Low limit – negative direction - X If High limit – positive direction - X If High limit – negative direction - √ For all other FB's – always X	If the FB is one of this: MoveAbs. MoveRel. HW limit If RLS – positive direction - √ If RLS – negative direction - X If FLS – positive direction - X If FLS – negative direction - √ SW limit If Low limit – positive direction - √ If Low limit – negative direction - X If High limit – positive direction - X If High limit – negative direction - √ For all other FB's – always X

14.3.2 Multi Axes Group

For a multi-axes group, the function block will automatically check whether the software limits have been reached for each axis in the group. If reached, will then check the allowed direction for the function block operation on each problematic axis in the group.

If the function block is inserted in ACS mode, the actual position is checked against the ACS software limits in the axis level (for further details on the definitions of kinematic transformations, refer to the section **5.10.1 Coordinate System and kinematic transformation**).

If in MCS mode, the desired position of each kinematic direction is checked against the MCS software limits in the group level.

The following table describes the alternative behaviour for any multiple-axes group axis function block type described in this document.

Inserted FB type for Multi Axes		
Current state	Admin FB	Motion FB
All axes Out of limit	Always - √	If the FB is one of this: MoveLinearAbsolute MoveLinearAbsoluteRepetitive MoveLinearRelative MoveLinearRelativeRepetitive MoveLinearAdditive MoveCirculatAbsolute Target position outside limit - √ Target position inside limit - X For all other FB’s – always √
One or more axes Inside limit	Always - √	If the FB is one of this: MoveLinearAbsolute MoveLinearRelative MoveLinearAdditive HW limit (per groups member) If RLS – positive direction - √ If RLS – negative direction - X If FLS – positive direction - X If FLS – negative direction - √ SW limit (per groups member) If Low limit – positive direction - √ If Low limit – negative direction - X If High limit – positive direction - X If High limit – negative direction - √

Inserted FB type for Multi Axes		
		For the following FB's – always X MoveLinearAbsoluteRepetitive MoveLinearRelativeRepetitive MoveCirculatAbsolute For all other FB's – always √

14.4 Real Time Behavior

How regularly is the hardware and software limit 32bits Status Register updated? The answer is generally every cycle. For single axis or ACS group motion, the hardware and software limits (ACS bits, 0-3 bits) are checked every cycle. However, for the MCS group, the MCS software limits (4-5 bits) are only tested when the group is in motion. The following table summarizes this behavior:

MCS Software Limits	When checked?
Single Axis	never
Multi Axis (ACS)	never
Multi Axis (MCS)	when in motion

The limit bits in the Group Status Register are only updated when the group is not in Disabled state.

When the limit is detected, and the axis or group is in motion, the motion will be stopped immediately. Then a Stop event is sent to the application level provided that the Stop event is registered. After the Stop is completed (if in motion), the position of all relevant axes are synchronized.

However, for Distributed (Non NC) axes, when the limit is detected, the motion is not stopped from the Maestro, but reverts to the servo drive, which will manage the stopping of the axis. Then a Stop event is sent to the application level provided that the Stop event is registered.

14.5 System Constraints And Limitations

The hardware limit recognition only operates under the two following conditions:

- RLS and FLS data representatives are mapped on the drive level
- Digital Input object is mapped to the Maestro

If both these conditions are not accomplished, the Maestro cannot manage the hardware limits. However, if the conditions are fulfilled, but the user wishes to disable the servo drive management of the limits, then set the command **XA[4]** on the Gold servo drive (only Gold servo drives support this command).

This command disables the following:

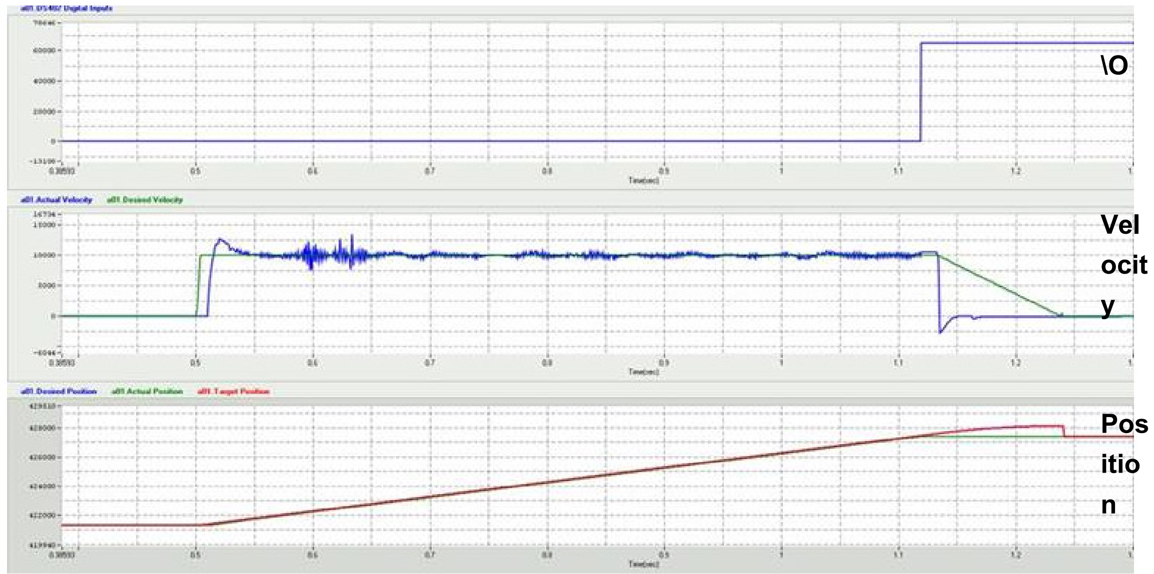
Bit	Disabled
Bit 0	bypass position software limit
Bit 1	bypass acceleration limit (Stop Deceleration(SD))
Bit 2	bypass hardware limit

The profiler is normally synchronized with the actual position of the drive, when the limit is detected. However, a situation may occur when the limit is detected, the profiler and the drive position are mismatched due to a different stop deceleration on the Maestro and the drive. This may cause the drive to unexpectedly jump position on the next motion.

This situation is managed, as described above, by synchronizing the position of all relevant axes. Refer to the two graphs below for an example.

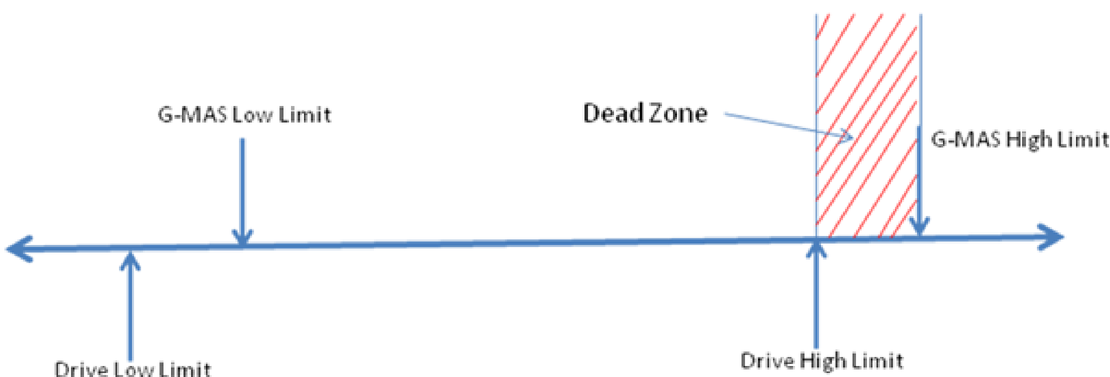
SD < GMAS_DC

SD > GMAS_DC



The Maestro is not responsible for unexpected behavior when the Drive Software limits are lower than Software Limit of the Maestro.

Drive Limit is wrong



The following should be noted:

- The Software Limits can be changed during the axis or group motion.
- There is no protection against non-motion deadlocks:
  - RLS & HighLimit
  - FLS & LowLimit

When the limit is reached, buffered or blended mode is forbidden. In addition, for Distributed mode, the stop event will be sent when the command **XA[4]** = 7.

Chapter 15: Saving Maestro User Program Parameters

This chapter describes the method to extract UPXML (**U**ser **P**arameters maintained in **XML** form) user application Parameters from XML files in the Maestro. The procedure uses the function within the MMCPP libraries class for IPC to accomplish the extraction.

15.1 Introduction

Previous to adding the new class methods, modifying a specific parameter within a Maestro User Application program required the user to modify hard coded constants (modify the code), compile, and create a new executable file. Now while using an XML file containing the Maestro parameters it is possible to edit only the XML file. Examples of Maestro User Application program data are:

- Axis motion parameters
- Communication parameters
- Program behaviors. Flags, etc. …

Presently, the user can now read all types of parameters using the dedicated UPXML functions. The Maestro – UPXML functions allow the user to retrieve parameters from a textual based file and set the values to program variables. These functions are to be supported in the CPP Library only, currently only when working with IPC.

An example of a well formed Maestro UPXML is:

```
<?xml version="1.0" encoding="utf-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="proposed.xsd">
     <FILE_DESCRIPTION NAME="Parameters" VERSION="NovaScan 1236" />
     <CATEGORY NAME="Profiler">
          <RESOURCES NAME="a01">
               <AC>10000000</AC>
               <DC>10000000</DC>
               <JERK>1073742336</JERK>
               <DRIVE_ID>'81'</DRIVE_ID>
          </RESOURCES>
          <RESOURCES NAME="a02">
               <AXIS_MODE>0</AXIS_MODE>
               <OP_MODE>8</OP_MODE>
               <KUKU>'1073742336'</KUKU>
               <DRIVE_ID>0</DRIVE_ID>
          </RESOURCES>
     </CATEGORY >
     <CATEGORY NAME="Communication">
          <RESOURCES NAME="a01">
               <TIMEOUT>0</TIMEOUT>
               <NUM_VARS>8</NUM_VARS>
               <KUKU>'1073742336'</KUKU>
          </RESOURCES>
          <RESOURCES NAME="a02">
               <TIMEOUT>0</TIMEOUT>
               <NUM_VARS>8</NUM_VARS>
               <KUKU>'1073742336'</KUKU>
          </RESOURCES>
     </CATEGORY >
     <CATEGORY NAME="Misc">
          <RESOURCES NAME="Global">
               <DOHOMEALWAYS>0</DOHOMEALWAYS>
               <SETPOSATTARGET>8</SETPOSATTARGET>
               <DONOTHINGATALL>'1073742336'</DONOTHINGATALL>
          </RESOURCES>
     </CATEGORY >
</root>
```

15.2 The MMCUserParams C++ Class

Based on the well-formed Maestro UPXML, the XML elements in the data file are defined:

Element	Value
The highest element **name** (under the "root") is **CATEGORY**	User defined Attribute (value) E.g. "Profiler" in XML element: <CATEGORY NAME="Profiler"> E.g. "Communication" in XML element: <CATEGORY NAME="Communication">
The element **name** under CATEGORY is **RESOURCES**	User define Attribute (value) E.g. "a02" in XML element: <RESOURCES NAME="a02"> E.g. "Global" in XML element: <RESOURCES NAME="Global">
The element **name** under RESOURCES is user defined and its value is returned from the saved parameters	XML element. The user defines the Element name, the saved value appears as XML data of the element E.g. value of 10000000 for parameter name AC: <AC>10000000</AC> E.g. value True for parameter name PRM032: <PRM032>TRUE</PRM032> **Note:** The elements require to be positioned in a suitable hierarchy location on the XML file.

The MMCUserParams class therefore includes the following methods described in detail in the following subsections:

Open

Opens the XML file, with specific parameters, such as:

- File name
- File location (path)
- How to behave related to subsequence read operation from this file, case of requested element name not found in file

Close

Closes the file pointed to the XML file and release resource used for parsing the file

GetXmlFileRoot

Function that retrieves the root-data of the XML file, and its specific description from the XML file header**.**

GetXmlFileDescrp

Function that retrieves data from the XML file, specifically the description of the XML file header.

Read

List of overloaded function that retrieves data for a given variable. In some cases also ensures that the data is within specific limitations.

I n addition, for double and long variable types, array of values are returned.

15.2.1 Open

Opens the XML file, with specific parameters

```
int Open(
char* cFileName=DEFAULT_XML_FILE_NAME,
unsigned int uiFlags=UPXML_SET_DEF_REQ_FLG,
char* cFilePath=DEFAULT_XML_FILE_PATH
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCUserParams.h

.NET Definition

Function Parameters

cFileName

Where DEFAULT_XML_FILE_NAME is the file UserParams.xml, the default XML file name.

uiFlags

If the flag is set (= UPXML_SET_DEF_REQ_FLG or 1), then a default setting is requested. When reading from this file, if no suitable value is found in the XML data, it returns the value of the Default parameter.

cFilePath

File path for the data to be read.

Where DEFAULT_XML_FILE_PATH is the "/mnt/jffs/usr/" default XML path when the cFilePath parameters do not exist.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

Return Value

0 if OK. Otherwise error code as detailed in section 4.12 Internal Library Error IDs e.g.:

- sequence error (E.g. reopen before close previous file)
- Cannot open file (no such file or no permit ion for access).
- File format error
- File too long…

15.2.2 Close

Closes the file pointed to the XML file and release resource used for parsing the file

```
int Close(
);
```

Source GMAS\includes\CPP\MMCUserParams.h

.NET Definition

Return Value

0 if OK (always).

15.2.3 Read

List of overloaded function that retrieves data to given variable. The Values may be of a double (single or array), long (single or array), Boolean, or String according to the number, type, and order of the parameters:

Read single value parameters

- Double, retrieve one parameter of type Double
- Long, retrieve one parameter of type Long
- Boolean, retrieve one parameter of type Boolean, ignores white space, but expects True / False
- String, retrieve one parameter of type string

Read Array of parameters values

- Retrieve array of double
- Retrieve array of long

Single Value Parameters

```
int Read (
char* pCtgryVal,
char*.pRsrcVal,
char* pTagName,

double &dVal,
[long &lVal,]
[Bool &bVal,]
[char* pStr,]

double dDefault,
[long lDefault,]
[Bool bDefault=0]

double dMin=DBL_MIN,
[long lMin=LONG_MIN,]

double dMax=DBL_MAX,
[long lMax=LONG_MAX,]
[long lLen,]

) throw (CMMCException);
```

Array of parameters values

```
int ReadArr (
char* pCtgryVal,
char*.pRsrcVal,
char* pTagName,

double dVal[],
[long lVal[],]

double dDefault,
[long lDefault,]

unsigned int& iActRdElm,
unsigned int iReqRdElm=1,

double dMin=DBL_MIN,
[long lMin=LONG_MIN,]
double dMax=DBL_MAX,
[long lMax=LONG_MAX,]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCUserParams.h

.NET Definition

Function Parameters

pCtgryVal

Pointer to the NULL terminated string. The string is the value of the tag name <CATEGORY

pRsrcVal

Pointer to the NULL terminated string. The string is the value of the tag name <RESOURCES>.

pTagName

Pointer to the NULL terminated string. The string is the Name of the users defined tag.

&dVal, &lVal, &bVal, pStr, dVal[], lVal[]

Reference variable to copy the value to.

dDefault, LDefault, bDefault

Default value to be set if the tag was not found.

dMin, lMin

Value read should not be less than this value. DBL_MIN has a minimum value of 1E-37

LONG_MIN is compiler depended constant.

dMax, lMax

Value read should not be greater than this value. DBL_MAX has a maximum value of 1E+37.

LONG_MAX is compiler depended constant.

lLen

Size of the read buffer.

& iActRdElm

Number of actual read elements.

iReqRdElm=1(default)

Number of read elements requested. This is the maximum value, the function will not read more elements.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

Return Value

0 if OK. Otherwise warning whether:

- Tag not found and default value was set.
- Default value set due to exceed Min/Max.

Note: **Make sure that there is sufficient space to store the read element.**

The Return Value is 0 if OK. Otherwise warning whether:

- Tag not found and default value was set
- Default value set due to exceed Min/Max

Remarks

For arrays, whole array elements should be of same type, with a comma separating elements values, and the Element values not interspaced with not broken by chars not belong to element,

Single value example:

Example:

/* For refer to tag name <CATEGORY> has value "Profiler ", put: */

pCtgrVal = "Profiler";

/* For refer to tag name <RESOURCES> has value "a01", put: */

pRsrcVal = "a01";

/* For refer to tag name <DC> */

/* (in context of other parameter, currently set to: */

/* CATEGORY="Profiler", RESOURCES="a01"), put: */

pTagName = "DC";

15.2.4 GetXmlFileRoot

Returns the XML file root (XSI ID values) pAtt1 and XSI Location

```
int GetXmlFileRoot (
char* pAtt1,
char* pAtt2,
long lLen
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCUserParams.h

.NET Definition

Function Parameters

pAtt1

Pointer to the buffer of size lLen. The attribute name: 'FILE_DESCRIPTION' is copied to it.

pAtt2

Pointer to buffer of size lLen, the attribute name: 'VERSION' of level1 element has name 'FILE_DESCRIPTION' coping to it.

lLen

Size of pAtt1 & pAtt2 buffer in bytes. The buffer size for returned values are at least lLen. Look for and parse XML file root into pAtt1, and xsi pAtt2. The buffer size for return values are at least lLen.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

Remarks

For example, for an XML file line where:

<root xmlns:XSI=http://www.w3.org/2001/XMLSchema-instance
XSI:noNamespaceSchemaLocation="proposed.xsd">

This will return the two parameters values:

pAtt1 = http://www.w3.org/2001/XMLSchema-instance

pAtt2 = "proposed.xsd"

If both attribute founds, then rt_val= MMC_OK, Otherwise, rt_val= MMC_LIB_UPXML_NOT_FOUND

if only one attribute is found copy it and return value MMC_LIB_UPXML_NOT_FOUND .

In this case, if you wish determined about coping data, put 0 at firs location, check it after the call, if it still there => no data copied.

15.2.5 GetXmlFileDescrp

Returns the XML "file description name" represented by pAtt1, and XML file version as pAtt2, the buffer size for return values which are at least lLen in size.

```
int GetXmlFileDescrp(
char* pAtt1,
char* pAtt2,
long lLen
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCUserParams.h

.NET Definition

Function Parameters

pAtt1

Pointer to the buffer of size lLen. The attribute name: 'FILE_DESCRIPTION' is copied to it.

pAtt2

Pointer to buffer of size lLen,

the root attribute name: 'xsi:noNamespaceSchemaLocation' coping to it.

lLen

Size of pAtt1 & pAtt2 buffer in bytes. The buffer size for returned values are at least lLen. Look for and parse XML file root into pAtt1, and xsi pAtt2. The buffer size for return values are at least lLen.

If both attribute founds, then rt_val= MMC_OK; Otherwise,

rt_val= MMC_LIB_UPXML_NOT_FOUND

if only one attribute is found, copy it and return value MMC_LIB_UPXML_NOT_FOUND.

In this case, if you wish determined about coping data, put 0 at firs location, check it after the call, if it still there => no data copied.

E.g.:

For XML file has the line (on level 1):

```
<FILE_DESCRIPTION NAME="Parameters" VERSION="NovaScan 1236" />
pAtt1 ="Parameters";
pAtt2 ="NovaScan 1236";
rt_val = MMC_OK;
```

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

15.2.6 SetSpeakDbgLvl

Sets the

```
void setSpeakDbgLvl (
unsigned int uiSpeak_lvl
);
```

Source GMAS\includes\CPP\MMCUserParams.h

.NET Definition

Function Parameters

uiSpeak_lvl

For internal use only

15.2.7 UPXML Functions Code Examples

```
/*
===============================================================================
 Name : UserParamTest.cpp
 Author : Haim Hillel
 Version :
 Description : GMAS C++ project source file for:
                 test program for Class "UPXML" User Param XML.
                 XML file as source for read parameters (for GMAS).
===============================================================================
*/

#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

#include <iostream>
#include <ctime>
#include <unistd.h>
#include "MMC_Definitions.h"
#include "UserParamDocTest.h"
#include "MMCUserParams.h"

using namespace std;

#define BUFPRT_VALSIZE 1024
#define  MAX_ARY_ELM       10

int main()
//  ======
{
long         min = -1000;      /* def. Min number val. */
long         max = 1400;       /* def. Max number Val. */
long         def = 1224;       /* def. number Val.    */

       /* XML Level 1 Attribute Value;*/
       /* Attribute of CATEGORY  E.g: */
       /*  Profiler001       */
char*              lvl1AttVal = "Profiler000";
            /* XML Level 2 Attribute value; Attribute     */
       /* of RESOURCES E.g: a01                */
char*              lvl2AttVal = "a00";
       /* XML Level 3 Tag Name; E.g.: PRM003           */
char*              lvl3Name    = "PRM025";
int                ind;
char               bufPrt_val  [BUFPRT_VALSIZE];
char               bufPrt_val1 [BUFPRT_VALSIZE];
long               bufPrt_valSize;
int                ErrId;
double             dVal [MAX_ARY_ELM];
long          lVal [MAX_ARY_ELM];
bool          bVal;
unsigned int  iActRdElm;
unsigned int  iReqRdElm;

MMCUserParams up;

printf("\n ==================================================================== ");
printf("\n Testing UPXML Id: %s %s %s", __FILE__, __DATE__, __TIME__);
printf("\n ==================================================================== \n");

             /* program seting def. file name; open def is: "UserParams.xml" */
ErrId = up.Open("UpxmlEg.xml", UPXML_SET_DEF_REQ_FLG);

bufPrt_valSize =  BUFPRT_VALSIZE;
```

```
ErrId = up.GetXmlFileRoot(bufPrt_val, bufPrt_val1, bufPrt_valSize);
printf("\n Root:    =11   ErrId=%d ======\n Val: <%s>  <%s> \n",  ErrId, bufPrt_val,
bufPrt_val1);

ErrId = up.GetXmlFileDescrp (bufPrt_val, bufPrt_val1, bufPrt_valSize);
printf("\n FileDescrp:   =12   ErrId=%d ======\n Val: <%s>  <%s> \n",  ErrId, bufPrt_val,
bufPrt_val1);

ErrId = up.Read  (lvl1AttVal,  lvl2AttVal, lvl3Name, bufPrt_val, bufPrt_valSize);
printf("\n buf:     =14   ErrId=%d ======%s=<%s> \n",  ErrId, lvl3Name, bufPrt_val);

dVal[0] = -1;
ErrId = up.Read  (lvl1AttVal,  lvl2AttVal, lvl3Name, dVal[0], (double)def,
                  (double)min, (double)max);
printf("\n double:  =15   ErrId=%d ======%s=<%f> \n",  ErrId, lvl3Name, dVal[0]);

lVal[0] = -1;
ErrId = up.Read  (lvl1AttVal,  lvl2AttVal, lvl3Name, lVal[0], (long)def,
                  (long)min, (long)max);
printf("\n long:    =16   ErrId=%d ======%s=<%ld> \n",  ErrId, lvl3Name, lVal[0]);

ErrId = up.Read  (lvl1AttVal,  lvl2AttVal,  lvl3Name, bVal, 0);
printf("\n Boolean: =17 def=0  ErrId=%d =====%s=<%d> \n",  ErrId, lvl3Name, bVal);

ErrId = up.Read  (lvl1AttVal,  lvl2AttVal,  lvl3Name, bVal, 1);
printf("\n Boolean: =18 def=1  ErrId=%d =====%s=<%d> \n",  ErrId, lvl3Name, bVal);

iReqRdElm = 4;
for (ind=0; ind < (int)iReqRdElm; ind++)
     dVal[ind] = -1.;

ErrId = up.ReadArr(lvl1AttVal, lvl2AttVal, lvl3Name, dVal, (double)def, iActRdElm,
iReqRdElm,   (double)min, (double)max);
printf("\n Array double: =19 #Act=%d, #Req=%d  ErrId=%d =====%s=\n Val: ", iActRdElm,
         iReqRdElm, ErrId, lvl3Name);
for (ind=0; ind < (int)iReqRdElm; ind++)
     printf("<%f> ", dVal[ind]);
printf("\n");

for (ind=0; ind < (int)iReqRdElm; ind++)
     lVal[ind] = -1;
ErrId = up.ReadArr(lvl1AttVal, lvl2AttVal, lvl3Name, lVal, (long)def,   iActRdElm,
                          iReqRdElm, (long)min, (long)max);
printf("\n Array long:   =20 #Act=%d, #Req=%d  ErrId=%d =====%s=\n Val: ", iActRdElm,
         iReqRdElm, ErrId, lvl3Name);
for (ind=0; ind < (int)iReqRdElm; ind++)
     printf("<%ld> ", lVal[ind]);
printf("\n");

lvl2AttVal = "a02";
lvl3Name   = "BoolPrm01";
ErrId = up.Read  (lvl1AttVal,  lvl2AttVal,  lvl3Name, bVal, 0);
printf("\n Boolean: =21 def=0  ErrId=%d =====%s=<%d> \n",  ErrId, lvl3Name, bVal);

lvl3Name   = "BoolPrm02";
ErrId = up.Read  (lvl1AttVal,  lvl2AttVal,  lvl3Name, bVal, 0);
printf("\n Boolean: =22 def=0  ErrId=%d =====%s=<%d> \n",  ErrId, lvl3Name, bVal);

up.Close();
return 0;
}
```

15.2.8 UpxmlEg.xml - Input File Example

```
<?xml version="1.0" encoding="utf-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="proposed.xsd">
    <FILE_DESCRIPTION NAME="Parameters" VERSION="Elmo Eg 1.1" />
    <CATEGORY NAME='Profiler'>
        <RESOURCES NAME="a01">
            <AC>1111,1111</AC>
            <DC>22222222</DC>
            <JERK>1073742336</JERK>
            <DRIVE_ID>81</DRIVE_ID>
        </RESOURCES>
        <RESOURCES NAME="a02">
            <AXIS_MODE>0</AXIS_MODE>
            <OP_MODE>5</OP_MODE>
            <KUKU>'444444444'</KUKU>
            <DRIVE_ID>0</DRIVE_ID>
        </RESOURCES>
    </CATEGORY>

    <CATEGORY NAME='Profiler000'>
        <RESOURCES NAME="a00">
            <AC>1111,1111</AC>
            <DC>22222222</DC>
            <JERK>1073742336</JERK>
            <PRM025>81,     82,83</DRIVE_ID>
        </RESOURCES>
        <RESOURCES NAME="a02">
            <AXIS_MODE>0</AXIS_MODE>
            <BoolPrm01>TRUE</AXIS_MODE>
            <OP_MODE>5</OP_MODE>
            <KUKU>'444444444'</KUKU>
            <BoolPrm02>TRUE</AXIS_MODE>
            <DRIVE_ID>0</DRIVE_ID>
        </RESOURCES>
    </CATEGORY>

    <CATEGORY NAME="Communication">
        <RESOURCES NAME="a02">
            <TIMEOUT>7</TIMEOUT>
            <NUM_VARS>8</NUM_VARS>
            <KUKU>'666666666'</KUKU>
        </RESOURCES>
    </CATEGORY>

    </root>
```

15.2.9 Program output example

```
===================================================================
Testing UPXML Id: ..\UserParamDocTest.cpp May 28 2013 13:36:47
===================================================================


Root:    =11   ErrId=0 ======
Val: <http://www.w3.org/2001/XMLSchema-instance>  <proposed.xsd>


FileDescrp:   =12   ErrId=0 ======
Val: <Parameters>  <Elmo Eg 1.1>


buf:     =14   ErrId=0 ======PRM025=<81,    82,83>


double:  =15   ErrId=0 ======PRM025=<81.000000>


long:    =16   ErrId=0 ======PRM025=<81>


**** MMCPPThrow: UPXML Read boolean iRetCode=300, iErrId=3021
Boolean: =17 def=0  ErrId=3021 =====PRM025=<0>


**** MMCPPThrow: UPXML Read boolean iRetCode=300, iErrId=3021
Boolean: =18 def=1  ErrId=3021 =====PRM025=<1>


**** MMCPPThrow: UPXML Read array iRetCode=300, iErrId=3020
Array double: =19 #Act=3, #Req=4  ErrId=3020 =====PRM025=
Val: <81.000000> <82.000000> <83.000000> <1224.000000>


**** MMCPPThrow: UPXML Read array iRetCode=300, iErrId=3020
Array long:   =20 #Act=3, #Req=4  ErrId=3020 =====PRM025=
Val: <81> <82> <83> <1224>


Boolean: =21 def=0  ErrId=0 =====BoolPrm01=<1>


Boolean: =22 def=0  ErrId=0 =====BoolPrm02=<1>
```

Chapter 16: Connectivity and Configuration

The following sections describe the connectivity and configuration function blocks utilized in the Maestro to communicate with a host and other devices described in the section **Chapter 2: Maestro Overview**.

16.1 Network Function Blocks

The following function blocks are utilized in the network communications to, and as part of the Maestro. The network node host system is defined by its host IP addresses, which are communicated to the Maestro FLASH. The following network function blocks are described, with the exclusion of MMC_NodesInfo, which is a structure:

Network
MMC_CloseUdpChannel
MMC_GetDefGateway
MMC_GetIpAddr
MMC_GetIpMask
MMC_GetServerIp
MMC_NetworkInfo
MMC_NetworkScan
MMC_OpenUdpChannel
MMC_SetDefGateway
MMC_SetDhcp
MMC_SetIpAddr
MMC_SetIpMask
MMC_SetServerIp

16.1.1 MMC_CloseUdpChannel

Closes a UDP channel per RPC/IPC connection.

```
MMC_LIB_API int MMC_CloseUdpChannelCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_CLOSEUDPCHANNEL_IN* pInParam,
OUT MMC_CLOSEUDPCHANNEL_OUT* pOutParam
);
```

Motion Mode NC – Not Relevant Distributed - Not Relevant

Source GMAS\includes\MMC_network_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CLOSEUDPCHANNEL_IN** input data structure using the MMC_CloseUdpChannel function.

pOutParam

Points to the **MMC_CLOSEUDPCHANNEL_OUT** output structure receiving information, as a result of calling the MMC_CloseUdpChannel function.

Remarks

None

Scope

Not limited

MMC_CLOSEUDPCHANNEL_IN Structure

```
typedef struct {
unsigned char dummy;
} MMC_CLOSEUDPCHANNEL_IN;
```

Parameters

dummy

Dummy IP address input. Any +ve character value.

MMC_CLOSEUDPCHANNEL_OUT Structure

typedef struct {
unsigned short usStatus;
short sErrorID;
}MMC_CLOSEUDPCHANNEL_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-1 describes the function block for MMC_CloseUdpChannel

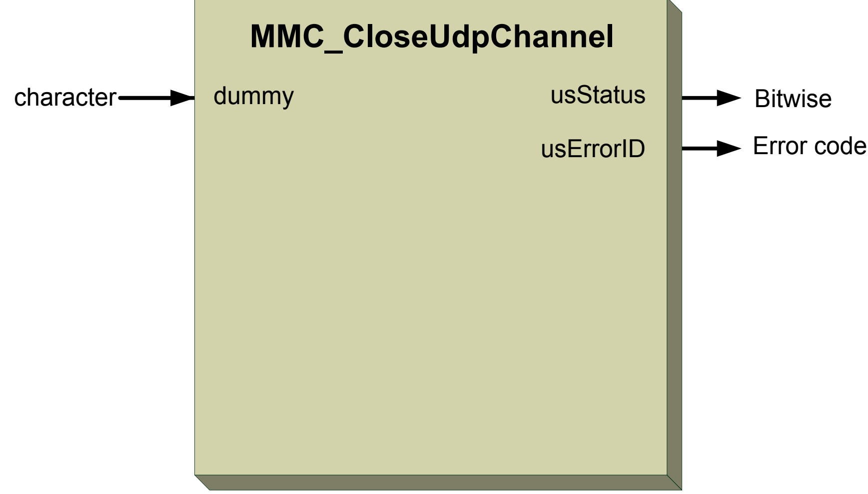


Figure 16-1: MMC_CloseUdpChannel function block

16.1.1.2 Function Block Code Example

```
int rc;
MMC_CLOSEUDPCHANNEL_IN      stCloseUDPChannel_in;
MMC_CLOSEUDPCHANNEL_OUT     stCloseUDPChannel_out;
//
// Inserting the structure parameters:
stCloseUDPChannel_in.dummy   = 1;         // dummy IP address
//
rc = MMC_CloseUdpChannelCmd (hConn, &stCloseUDPChannel_in, &stCloseUDPChannel_out);
if (rc != 0)
{
    HandleError();
}
```

16.1.2 MMC_GetDefGateway

Reads the default gateway IP address.

```
MMC_LIB_API int MMC_GetDefGatewayCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GET_DEFGATEWAY_IN* pInParam,
OUT MMC_GET_DEFGATEWAY_OUT* pOutParam
);
```

Motion Mode	NC - Not Relevant	Distributed - Not Relevant
Source	GMAS\includes\MMC_network_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GET_DEFGATEWAY_IN** input data structure using the MMC_GetDefGateway function.

pOutParam

Points to the **MMC_GET_DEFGATEWAY_OUT** output structure receiving information, as a result of calling the MMC_GetDefGateway function.

Remarks

None

Scope

Not limited

MMC_GET_DEFGATEWAY_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_GET_DEFGATEWAY_IN;
```

Parameters

dummy

Dummy IP address input. Any +ve character value 0 - 255.

MMC_GET_DEFGATEWAY_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned char cFirst;
unsigned char cSecond;
unsigned char cThird;
unsigned char cFourth;
}MMC_GET_DEFGATEWAY_OUT;
```

Parameters

cFirst

First octet gateway IP Address. Any three digit character number up to 255

cSecond

Second octet gateway IP Address. Any three digit character number up to 255

cThird

Third octet gateway IP Address. Any three digit character number up to 255

cFourth

Fourth octet gateway IP Address. Any three digit character number up to 255

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-2 describes the function block for MMC_GetDefGateway

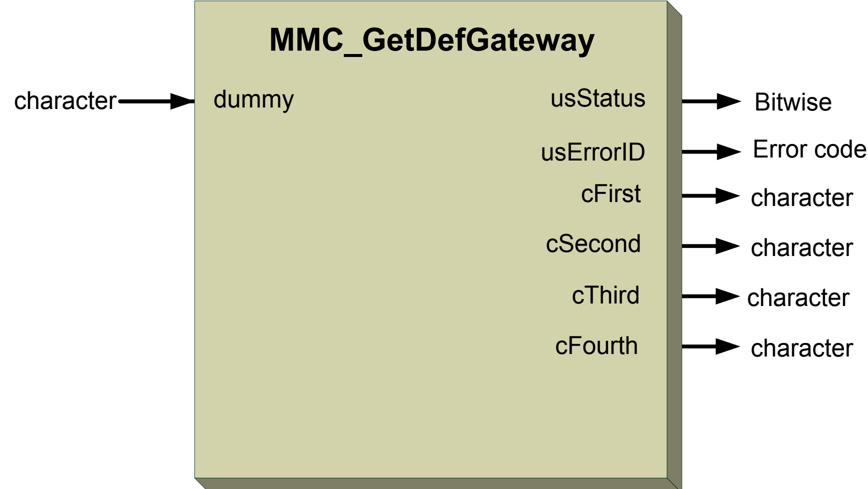


Figure 16-2: MMC_GetDefGateway function block

16.1.2.2 Function Block Code Example

```
int rc;
MMC_GET_DEFGATEWAY_IN       stGetDefGateway_in;
MMC_GET_DEFGATEWAY_OUT      stGetDefGateway_out;
//
// Inserting the structure parameters:
stGetDefGateway_in.dummy   = 1;           // dummy IP address
//
rc = MMC_GetDefGatewayCmd (hConn, &stGetDefGateway_in, &stGetDefGateway_out);
printf("Default Gateway Status[%ld][%ld][%ld][%ld] ErrId[%d]\n", (long
int)stGetDefGateway_out.cFirst, (long int)stGetDefGateway_out.cSecond, (long
int)stGetDefGateway_out.cThird, (long int)stGetDefGateway_out.cFourth,
(short)stGetDefGateway_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.1.3 MMC_GetDhcp

Reads the DHCP mode.

```
MMC_LIB_API int MMC_GetDhcpCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GET_DHCP_IN* pInParam,
OUT MMC_GET_DHCP_OUT* pOutParam
);
```

Motion Mode	NC - Not Relevant	Distributed - Not Relevant
Source	GMAS\includes\MMC_network_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GET_DHCP_IN** input data structure using the MMC_GetDhcp function.

pOutParam

Points to the **MMC_GET_DHCP_OUT** output structure receiving information, as a result of calling the MMC_GetDhcp function.

Remarks

None

Scope

Not limited

MMC_GET_DHCP_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_GET_DHCP_IN;
```

Parameters

dummy

Default Gateway IP address input. Any +ve character value.

MMC_GET_DHCP_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned char ucMode;
}MMC_GET_DHCP_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

ucMode

Mode is either static or dynamic. It is recommended to select Static and manually select the DHCP. If dynamic, the DHCP is automatically provided from the pool. Boolean values of TRUE/FALSE accepted

Figure 16-3 describes the function block for MMC_GetDhcp

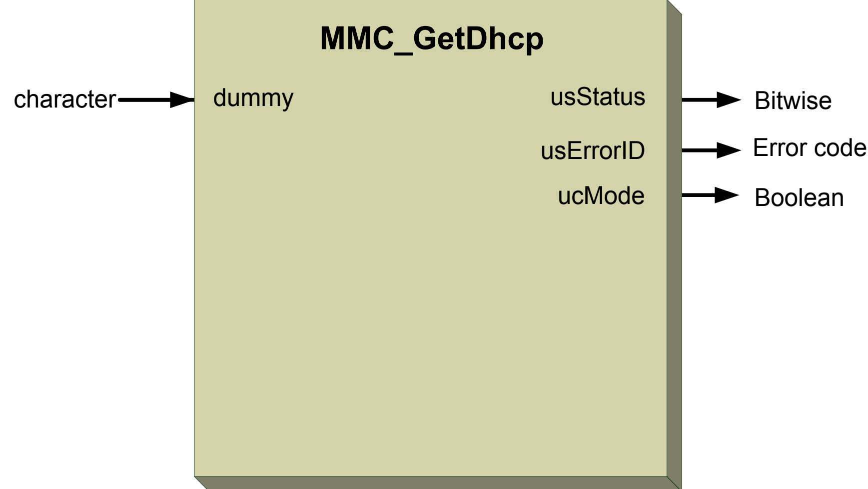


Figure 16-3: MMC_GetDhcp function block

16.1.3.2 Function Block Code Example

```
int rc;
MMC_GET_DHCP_IN         stGetDHCP_in;
MMC_GET_DHCP_OUT        stGetDHCP_out;
//
// Inserting the structure parameters:
stGetDHCP_in.dummy  = 1;       // dummy IP address
printf("DHCP Status[%ld] ErrId[%d]\n", (long int)stGetDHCP_out.ucMode,
(short)stGetDHCP_out.sErrorID);
//
rc = MMC_GetDhcpCmd (hConn, &stGetDHCP_in, &stGetDHCP_out);
if (rc != 0)
{
    HandleError();
}
```

16.1.4 IMMC_GetIpAddr

Reads the DHCP mode.

```
MMC_LIB_API int MMC_GetIpAddrCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GET_IP_ADDRESS_IN* pInParam,
OUT MMC_GET_IP_ADDRESS_OUT* pOutParam
);
```

Motion Mode NC - Not Relevant Distributed - Not Relevant

Source GMAS\includes\MMC_network_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GET_IP_ADDRESS_IN** input data structure using the MMC_GetIpAddr function.

pOutParam

Points to the **MMC_GET_IP_ADDRESS_OUT** output structure receiving information, as a result of calling the MMC_GetIpAddr function.

Remarks

None

Scope

Not limited

MMC_GET_IP_ADDRESS_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_GET_IP_ADDRESS_IN;
```

Parameters

dummy

Dummy IP address input. Any +ve character value 0 - 255.

MMC_GET_IP_ADDRESS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
char cFirst;
char cSecond;
char cThird;
char cFourth;
}MMC_GET_IP_ADDRESS_OUT;
```

Parameters

cFirst

First octet IP Address. Any three digit character number up to 255

cSecond

Second octet IP Address. Any three digit character number up to 255

cThird

Third octet IP Address. Any three digit character number up to 255

cFourth

Fourth octet IP Address. Any three digit character number up to 255

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-4 describes the function block for MMC_GetIpAddr

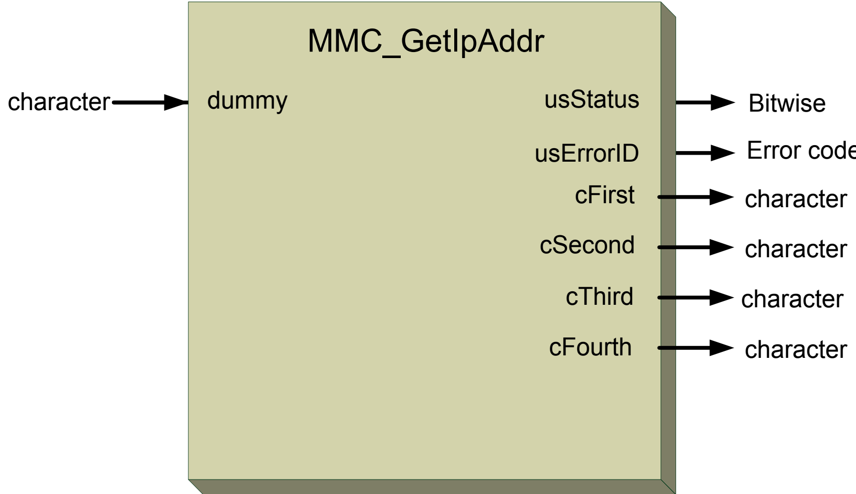


Figure 16-4: MMC_GetIpAddr function block

16.1.4.2 Function Block Code Example

```
int rc;
MMC_GET_IP_ADDRESS_IN      stGetIPAddress_in;
MMC_GET_IP_ADDRESS_OUT     stGetIPAddress_out;
//
// Inserting the structure parameters:
stGetIPAddress_in.dummy  = 1;       // dummy IP address
//
rc = MMC_GetIpAddrCmd (hConn, &stGetIPAddress_in, &stGetIPAddress_out);
printf("IP Address Status[%ld][%ld][%ld][%ld] ErrId[%d]\n", (long
int)stGetIPAddress_out.cFirst, (long int)stGetIPAddress_out.cSecond, (long
int)stGetIPAddress_out.cThird, (long int)stGetIPAddress_out.cFourth,
(short)stGetIPAddress_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.1.5 MMC_GetIpMask

Reads the IP mask.

```
MMC_LIB_API int MMC_GetIpMaskCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GET_IP_MASK_IN* pInParam,
OUT MMC_GET_IP_MASK_OUT* pOutParam
);
```

Motion Mode	NC - Not Relevant	Distributed - Not Relevant
Source	GMAS\includes\MMC_network_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GET_IP_MASK_IN** input data structure using the MMC_GetIpMask function.

pOutParam

Points to the **MMC_GET_IP_MASK_OUT** output structure receiving information, as a result of calling the MMC_GetIpMask function.

Remarks

None

Scope

Not limited

MMC_GET_IP_MASK_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_GET_IP_MASK_IN;
```

Parameters

dummy

Dummy IP mask input. Any +ve character value 0 - 255.

MMC_GET_IP_MASK_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
char cFirst;
char cSecond;
char cThird;
char cFourth;
}MMC_GET_IP_MASK_OUT;
```

Parameters

cFirst

First octet IP mask. Any three digit character number up to 255

cSecond

Second octet IP mask. Any three digit character number up to 255

cThird

Third octet IP mask. Any three digit character number up to 255

cFourth

Fourth octet IP mask. Any three digit character number up to 255

usStatus

Bitwise returned command status with the following values:

Aborted, Done, or CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-5 describes the function block for MMC_GetIpMask

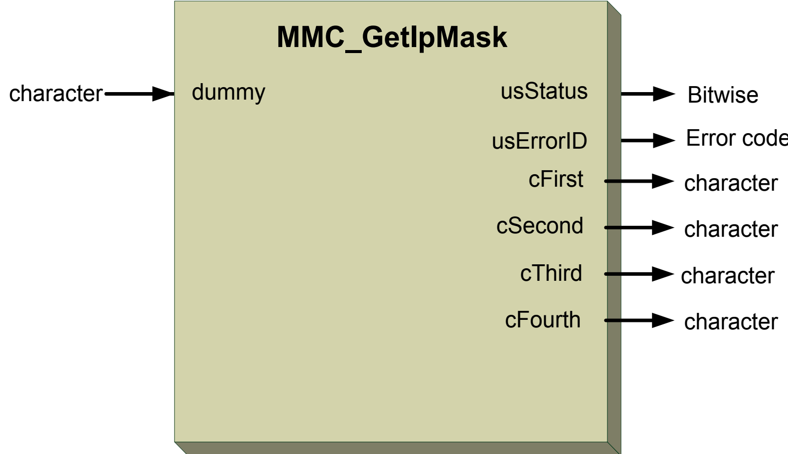


Figure 16-5: MMC_GetIpMask function block

16.1.5.2 Function Block Code Example

```
int rc;
MMC_GET_IP_MASK_IN          stGetIPMask_in;
MMC_GET_IP_MASK_OUT         stGetIPMask_out;
//
// Inserting the structure parameters:
stGetIPMask_in.dummy   = 1;      // dummy IP address
//
rc = MMC_GetIpMaskCmd (hConn, &stGetIPMask_in, &stGetIPMask_out);
printf("IP Mask Status[%ld][%ld][%ld][%ld] ErrId[%d]\n", (long int)stGetIPMask_out.cFirst,
(long int)stGetIPMask_out.cSecond, (long int)stGetIPMask_out.cThird, (long
int)stGetIPMask_out.cFourth, (short)stGetIPMask_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.1.6 MMC_GetServerIp

Obtain the Server IP address.

```
MMC_LIB_API int MMC_GetServerIpCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GET_SERVERIP_IN* pInParam,
OUT MMC_GET_SERVERIP_OUT* pOutParam
);
```

Motion Mode NC - Not Relevant Distributed - Not Relevant

Source GMAS\includes\MMC_network_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GET_SERVERIP_IN** input data structure using the MMC_GetServerIp function.

pOutParam

Points to the **MMC_GET_SERVERIP_OUT** output structure receiving information, as a result of calling the MMC_GetServerIp function.

Remarks

None

Scope

Not limited

MMC_GET_SERVERIP_IN Structure

```
typedef struct {
unsigned char dummy;
}MMC_GET_SERVERIP_IN;
```

Parameters

dummy

Server IP address exists. Boolean TRUE/FALSE values accepted.

MMC_GET_SERVERIP_OUT Structure

```
typedef struct{
int iPort;
unsigned short usStatus;
short sErrorID;
char cFirst;
char cSecond;
char cThird;
char cFourth;
}MMC_GET_SERVERIP_OUT;
```

Parameters

iPort

Port Address for the Server IP. Any +v integer.

cFirst

First octet IP address. Any three digit character number up to 255

cSecond

Second octet IP address. Any three digit character number up to 255

cThird

Third octet IP address. Any three digit character number up to 255

cFourth

Fourth octet IP address. Any three digit character number up to 255

usStatus

Bitwise returned command status with the following values:

Aborted Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-6 describes the function block for MMC_GetServerIp

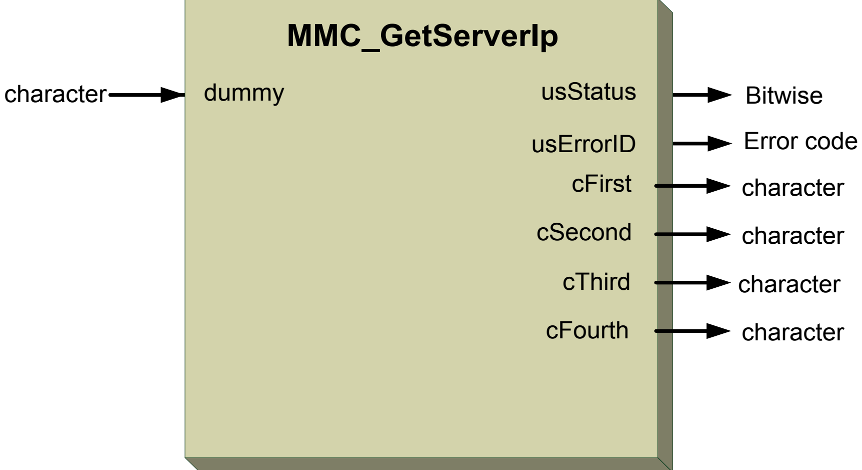


Figure 16-6: MMC_GetServerIp function block

16.1.6.2 Function Block Code Example

```
int rc;
MMC_GET_SERVERIP_IN         stGetServerIP_in;
MMC_GET_SERVERIP_OUT        stGetServerIP_out;
//
// Inserting the structure parameters:
stGetServerIP_in.dummy      = 1;     // dummy IP address
//
rc = MMC_GetServerIpCmd (hConn, &stGetServerIP_in, &stGetServerIP_out);
printf("Server Status[%ld][%ld][%ld][%ld] ErrId[%d]\n", (long int)stGetServerIP_out.cFirst,
(long int)stGetServerIP_out.cSecond, (long int)stGetServerIP_out.cThird, (long
int)stGetServerIP_out.cFourth, (short)stGetServerIP_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.1.7 MMC_NetworkInfo

Returns the network information, detailing the systems connected and/or defined in the resources file located in the Maestro FLASH.

```
MMC_LIB_API int MMC_NetworkInfoCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_NETWORKINFO_IN* pInParam,
OUT MMC_NETWORKINFO_OUT* pOutParam
);
```

Motion Mode	NC - Not Relevant	Distributed - Not Relevant
Source	GMAS\includes\MMC_network_API.h GMAS Programming(IEC 61331 Program)\ElmoGlobal	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_NETWORKINFO_IN** input data structure using the MMC_NetworkInfo function.

pOutParam

Points to the **MMC_NETWORKINFO_OUT** output structure receiving information, as a result of calling the MMC_NetworkInfo function.

Remarks

The network information may range from servo drives, Buses, etc. connected to the Maestro via CANbus connections.

Scope

Not limited

MMC_NETWORKINFO_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_NETWORKINFO_IN;
```

Parameters

dummy

Dummy values

MMC_NETWORKINFO_OUT Structure

```
typedef struct{
int iBusType;
int iBusBaud;
int iNumOfActiveNodes;
int iNumOfResFileNodes;
unsigned short usStatus;
short sErrorID;
MMC_NODESINFO stNodesInfo[CAN_ID_MAX - 1];
}MMC_NETWORKINFO_OUT;
```

Parameters

iBusType

BUS type based on the following enumerator types:

eCOMM_TYPE_NONE =0

eCOMM_TYPE_ETHERCAT =1

eCOMM_TYPE_CAN =2

iBusType can have values of 0, 1, or 2

iBusBaud

BUS baud value according to the following list:

10, 20, 50, 100, 125, 250, 500, 800, 1000

Values of 1 – 1000 based on the specific values detailed in the description.

iNumOfActiveNodes

Number of active nodes in the resource file in the Maestro and physically connected to the BUS.

iNumOfResFileNodes

Number of nodes in the resource files. Values of 1 – 127 accepted.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

stNodesInfo

Maximum no of all CANbus connected systems, excluding the Maestro. The enumerator *stNodesInfo* is the Integer ID.

The array [CAN_ID_MAX - 1] can have values of [0 – 127].

The MMC_NODESINFO parameter defines the network nodes and their states. It has the following sub-parameter structure:

uiNodeType

Note: **This field is only valid for Elmo DS402 devices.**

32bit bitwise network node type with specific +ve integer values according to the following definition:

Bits	Description	
	Value	Product
0…7	0	Saxophone
	1	Clarinet
	2	Reserved
	3	Harmonica
	4	Cello
	5	Bassoon
	6	Trumpet
	7	Tuba
	8	Banjo
	9	Cornet
	10	Whistle
	11	Didge
	12	Tweeter
	13	Drum
	14	Reserved
	15	Bee
	16	Hornet
	17	Hawk
	18	Falcon
	19	Eagle
	20	Harp
	21	Guitar
	22	Reserved (for Bell)
	23	Trombone

	24	Panther
	25	Duo
	26 - 255	Reserved
8…11	Reserved for product name	
12…13	Always 0	
14	0 = SimplIQ 1 = Gold	
15	Always 0	
16	0: No CAN 1: CAN included	
17	0: EtherCAT active 1: TCP\IP	
18 - 20	Feedback type (Port A + Port B) more details see Gold Guitar brochure : 0: E type (Encoder + Encoder, Analog sensor) 1: A type (Absolute + Encoder, Analog sensor) 2: R type (Encoder, Analog sensor + Resolver) 3: M type (Absolute + Resolver)	
21 - 23	Reserved for HW type ("option in Gold Guitar" brochure)	
23 - 31	Reserved for general needs	

ucNodeID

Network Node ID. Numeric character values of 0 – 255 accepted.

ucCommType

The *ucCommType* enumerator communication node type. Can have the following values:

NODE_ELMO = 1

NODE_DS301 = 2

NODE_DS401 = 3

NODE_DS402= 4

NODE_DS406= 5

ucNodeState

The *ucNodeState* enumerator NodeState can have the following values:

eMMC_NODE_STATE_ENABLED = 0
Node exists on the CAN bus AND Node exists in the resource file.

eMMC_NODE_STATE_DISABLED
Node does not exist on the CAN bus AND Node exists in the resource file.

eMMC_NODE_STATE_UNKNOWN
Node exists on the CAN bus AND Node does not exist in the resource file.

dummy

The enumerator value of the node state. Character values of 0 – 255 accepted.

Figure 16-7 describes the function block for MMC_NetworkInfo as applied within the IEC 61131 programming for MC_GetNetworkInfo.

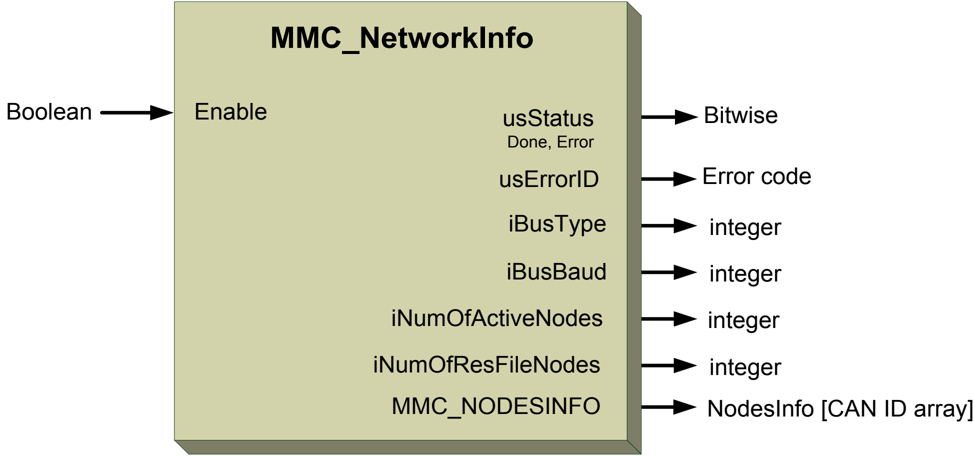


Figure 16-7: MMC_NetworkInfo function block

16.1.7.2 Function Block Code Example

```
int rc;
MMC_NETWORKINFO_IN          stNetworkInfo_in;
MMC_NETWORKINFO_OUT         stNetworkInfo_out;
//
// Inserting the structure parameters:
stNetworkInfo_in.dummy  = 1;     // dummy IP address
//
rc = MMC_NetworkInfoCmd (hConn, &stNetworkInfo_in, &stNetworkInfo_out);
if (rc != 0)
{
    HandleError();
}
```

16.1.8 MMC_NetworkScan

Scans the network to locate nodes on the network.

```
MMC_LIB_API int MMC_NetworkScanCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_NETWORKSCAN_IN* pInParam,
OUT MMC_NETWORKSCAN_OUT* pOutParam
);
```

Motion Mode	NC - Not Relevant	Distributed - Not Relevant
Source	GMAS\includes\MMC_network_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_NETWORKSCAN_IN** input data structure using the MMC_NetworkScan function.

pOutParam

Points to the **MMC_NETWORKSCAN_OUT** output structure receiving information, as a result of calling the MMC_NetworkScan function.

Remarks

None

Scope

Operation allowed in StandStill or Disabled state, but forbidden in any motion state.

Under certain circumstances, using this function in StandStill may cause the axis to enter Error Stop state.

MMC_NETWORKSCAN_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_NETWORKSCAN_IN;
```

Parameters

dummy

Server IP address exists. Boolean TRUE/FALSE values accepted.

MMC_NETWORKSCAN_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_NETWORKSCAN_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-8 describes the function block for MMC_NetworkScan

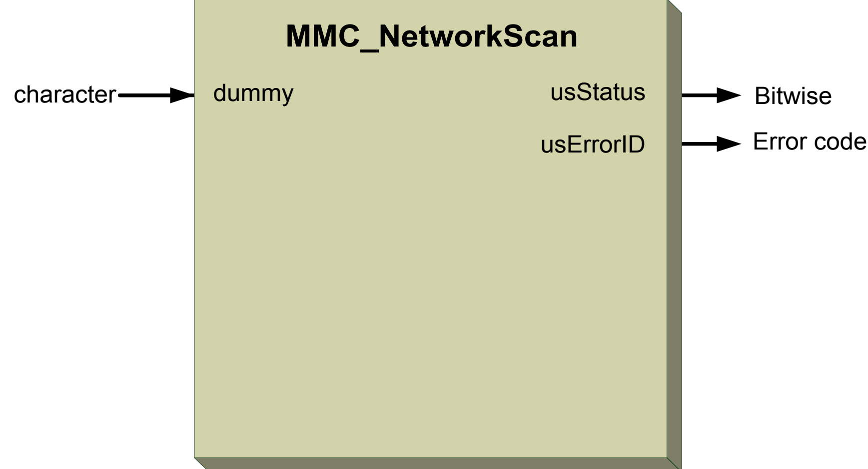


Figure 16-8: MMC_NetworkScan function block

16.1.8.2 Function Block Code Example

```
int rc;
MMC_NETWORKSCAN_IN          stNetworkScan_in;
MMC_NETWORKSCAN_OUT         stNetworkScan_out;
//
// Inserting the structure parameters:
//
rc = MMC_NetworkScanCmd (hConn, &stNetworkScan_in, &stNetworkScan_out);
if (rc != 0)
{
    HandleError();
}
```

16.1.9 MMC_OpenUdpChannel

Opens a UDP channel per RPC/IPC connection.

```
MMC_LIB_API int MMC_OpenUdpChannelCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_OPENUDPCHANNEL_IN* pInParam,
OUT MMC_OPENUDPCHANNEL_OUT* pOutParam
);
```

Motion Mode	NC - Not Relevant	Distributed - Not Relevant
Source	GMAS\includes\MMC_network_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_OPENUDPCHANNEL_IN** input data structure using the MMC_OpenUdpChannel function.

pOutParam

Points to the **MMC_OPENUDPCHANNEL_OUT** output structure receiving information, as a result of calling the MMC_OpenUdpChannel function.

Remarks

The UDP port has already been assigned in the MMC_InitConnection Function block.

Scope

Not limited

MMC_OPENUDPCHANNEL_IN Structure

```
typedef struct {
int iEventsMask;
int iPort;
char cFirst;
char cSecond;
char cThird;
char cFourth;
}MMC_OPENUDPCHANNEL_IN;
```

Parameters

iEventsMask

Describes the events mask for the callback function from the Maestro to the host program. Defined according to the event IDs described in the section 12.20 **Events Mask** and Enumeration **on page 1093**. +ve bitwise integer ID.

iPort

Port Address for the UDP Port Server IP. Any +ve integer.

cFirst

First octet IP address. Any three digit character number up to 255

cSecond

Second octet IP address. Any three digit character number up to 255

cThird

Third octet IP address. Any three digit character number up to 255

cFourth

Fourth octet IP address. Any three digit character number up to 255

MMC_OPENUDPCHANNEL_OUT Structure

typedef struct {
unsigned short usStatus;
short sErrorID;
}MMC_OPENUDPCHANNEL_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-9 describes the function block for MMC_OpenUdpChannel

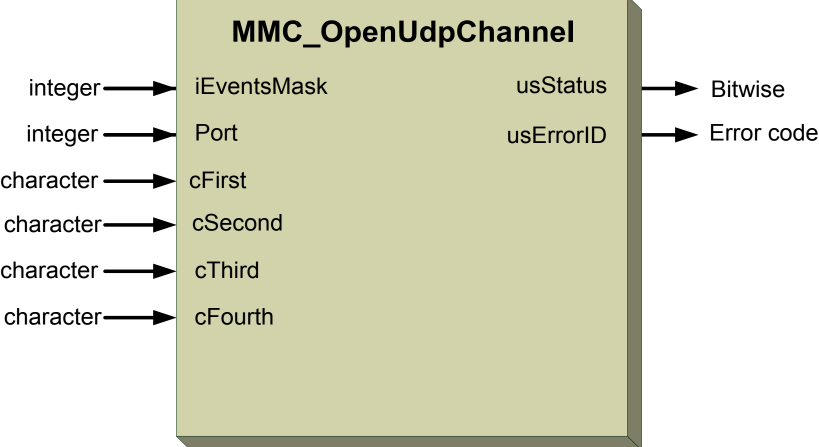


Figure 16-9: MMC_OpenUdpChannel function block

16.1.9.2 Function Block Code Example

```
int rc;
MMC_OPENUDPCHANNEL_IN      stOpenUDPChannel_in;
MMC_OPENUDPCHANNEL_OUT     stOpenUDPChannel_out;
//
// Inserting the structure parameters:
stOpenUDPChannel_in.iEventsMask      = 101; // Describes the events mask
stOpenUDPChannel_in.iPort            = 54;  // Selects the UDP Port
stOpenUDPChannel_in.cFirst           = 152; // First octet IP address
stOpenUDPChannel_in.cSecond          = 93;  // Second octet IP address
stOpenUDPChannel_in.cThird           = 43;  // Third octet IP address
stOpenUDPChannel_in.cFourth          = 15;  // Fourth octet IP address
//
rc = MMC_OpenUdpChannelCmd (hConn, &stOpenUDPChannel_in, &stOpenUDPChannel_out);
if (rc != 0)
{
    HandleError();
}
```

16.1.10 MMC_SetDefGateway

Set default gateway IP address.

```
MMC_LIB_API int MMC_SetDefGatewayCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SET_DEFGATEWAY_IN* pInParam,
OUT MMC_SET_DEFGATEWAY_OUT* pOutParam
);
```

Motion Mode NC - Not Relevant Distributed - Not Relevant

Source GMAS\includes\MMC_network_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SET_DEFGATEWAY_IN** input data structure using the MMC_SetDefGateway function.

pOutParam

Points to the **MMC_SET_DEFGATEWAY_OUT** output structure receiving information, as a result of calling the MMC_SetDefGateway function.

Remarks

When new, by default, the gateway IP address of the Maestro is set to 192.168.1.1.

Scope

Not limited

MMC_SET_DEFGATEWAY_IN Structure

```
typedef struct{
char cFirst;
char cSecond;
char cThird;
char cFourth;
}MMC_SET_DEFGATEWAY_IN;
```

Parameters

cFirst

First octet gateway IP Address. Any three digit character number up to 255

cSecond

Second octet gateway IP Address. Any three digit character number up to 255

cThird

Third octet gateway IP Address. Any three digit character number up to 255

cFourth

Fourth octet gateway IP Address. Any three digit character number up to 255

MMC_SET_DEFGATEWAY_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SET_DEFGATEWAY_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-10 describes the function block for MMC_SetDefGateway

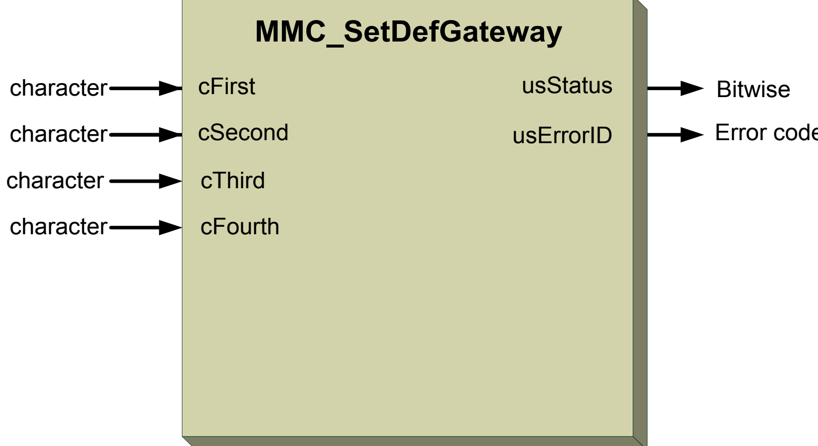


Figure 16-10: MMC_SetDefGateway function block

16.1.10.2 Function Block Code Example

```
int rc;
MMC_SET_DEFGATEWAY_IN      stSetDefGateway_in;
MMC_SET_DEFGATEWAY_OUT     stSetDefGateway_out;
//
// Inserting the structure parameters:
stSetDefGateway_in.cFirst       = 152; // First octet IP address
stSetDefGateway_in.cSecond      = 93;  // Second octet IP address
stSetDefGateway_in.cThird       = 43;  // Third octet IP address
stSetDefGateway_in.cFourth      = 15;  // Fourth octet IP address
//
rc = MMC_SetDefGatewayCmd (hConn, &stSetDefGateway_in, &stSetDefGateway_out);
if (rc != 0)
{
    HandleError();
}
```

16.1.11 MMC_SetDhcp

Sets the DHCP Mode for the Maestro.

```
MMC_LIB_API int MMC_SetDhcpCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SET_DHCP_IN* pInParam,
OUT MMC_SET_DHCP_OUT* pOutParam
);
```

Motion Mode	NC - Not Relevant	Distributed - Not Relevant
Source	GMAS\includes\MMC_network_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SET_DHCP_IN** input data structure using the MMC_SetDhcp function.

pOutParam

Points to the **MMC_SET_DHCP_OUT** output structure receiving information, as a result of calling the MMC_SetDhcp function.

Remarks

None

Scope

Not limited

MMC_SET_DHCP_IN Structure

```
typedef struct{
unsigned char ucMode;
}MMC_SET_DHCP_IN;
```

Parameters

ucMode

Mode is either static or dynamic. It is recommended to select Static and manually select the DHCP. If dynamic, the DHCP is automatically provided from the pool.

Values accepted are Boolean, TRUE/FALSE.

MMC_SET_DHCP_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_NETWORKSCAN_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-11 describes the function block for MMC_SetDhcp

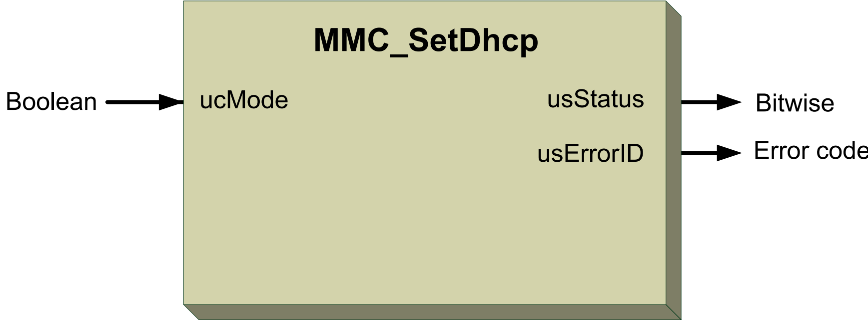


Figure 16-11: MMC_SetDhcp function block

16.1.11.2 Function Block Code Example

```
int rc;
MMC_SET_DHCP_IN        stSetDHCP_in;
MMC_SET_DHCP_OUT       stSetDHCP_out;
//
// Inserting the structure parameters:
stSetDHCP_in.ucMode  = 1;    // Mode is either static or dynamic
//
rc = MMC_SetDhcpCmd (hConn, &stSetDHCP_in, &stSetDHCP_out);
if (rc != 0)
{
    HandleError();
}
```

16.1.12 MMC_SetIpAddr

Sets the Maestro IP address.

```
MMC_LIB_API int MMC_SetIpAddrCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SET_IP_ADDRESS_IN* pInParam,
OUT MMC_SET_IP_ADDRESS_OUT* pOutParam
);
```

Motion Mode	NC - Not Relevant	Distributed - Not Relevant
Source	GMAS\includes\MMC_network_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SET_IP_ADDRESS_IN** input data structure using the MMC_SetIpAddr function.

pOutParam

Points to the **MMC_SET_IP_ADDRESS_OUT** output structure receiving information, as a result of calling the MMC_SetIpAddr function.

Remarks

When new, by default, the IP address of the Maestro is set to 192.168.1.3.

Scope

Not limited

MMC_SET_IP_ADDRESS_IN Structure

```
typedef struct{
char cFirst;
char cSecond;
char cThird;
char cFourth;
}MMC_SET_IP_ADDRESS_IN;
```

Parameters

cFirst

First octet IP Address. Any three digit character number up to 255

cSecond

Second octet IP Address. Any three digit character number up to 255

cThird

Third octet IP Address. Any three digit character number up to 255

cFourth

Fourth octet IP Address. Any three digit character number up to 255

MMC_SET_IP_ADDRESS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SET_IP_ADDRESS_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-12 describes the function block for MMC_SetIpAddr.

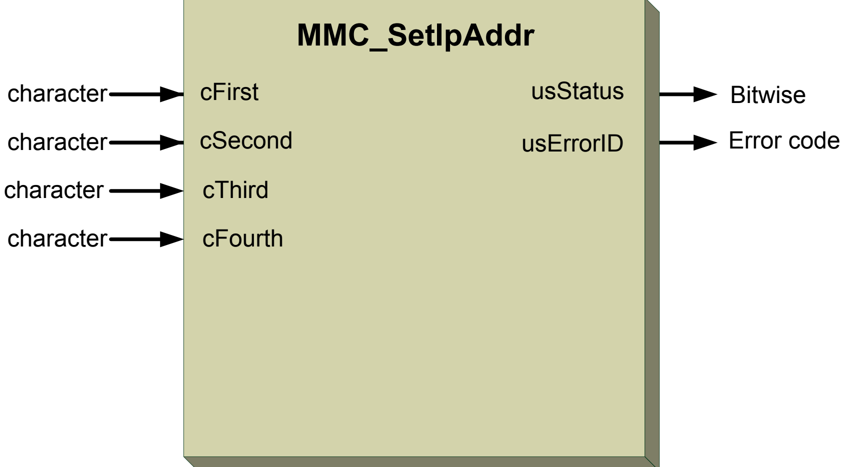


Figure 16-12: MMC_SetIpAddr function block

16.1.12.2 Function Block Code Example

```
int rc;
MMC_SET_IP_ADDRESS_IN      stSetIPAddress_in;
MMC_SET_IP_ADDRESS_OUT     stSetIPAddress_out;
//
// Inserting the structure parameters:
stSetIPAddress_in.cFirst   = 152; // First octet IP address
stSetIPAddress_in.cSecond  = 93;  // Second octet IP address
stSetIPAddress_in.cThird   = 43;  // Third octet IP address
stSetIPAddress_in.cFourth  = 15;  // Fourth octet IP address
//
rc = MMC_SetIpAddrCmd (hConn, &stSetIPAddress_in, &stSetIPAddress_out);
if (rc != 0)
{
    HandleError();
}
```

16.1.13 MMC_SetIpMask

Set the IP netmask of the Maestro.

```
MMC_LIB_API int MMC_SetIpMaskCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SET_IP_MASK_IN* pInParam,
OUT MMC_SET_IP_MASK_OUT* pOutParam
);
```

Motion Mode	NC - Not Relevant	Distributed - Not Relevant
Source	GMAS\includes\MMC_network_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SET_IP_MASK_IN** input data structure using the MMC_SetIpMask function.

pOutParam

Points to the **MMC_SET_IP_MASK_OUT** output structure receiving information, as a result of calling the MMC_SetIpMask function.

Remarks

When new, by default, the IP netmask address of the Maestro is set to 255.255.255.0.

Scope

Not limited

MMC_SET_IP_MASK_IN Structure

```
typedef struct{
char cFirst;
char cSecond;
char cThird;
char cFourth;
}MMC_SET_IP_MASK_IN;
```

Parameters

cFirst

First octet IP address mask. Any three digit character number up to 255

cSecond

Second octet IP address mask. Any three digit character number up to 255

cThird

Third octet IP address mask. Any three digit character number up to 255

cFourth

Fourth octet IP address mask. Any three digit character number up to 255

MMC_SET_IP_MASK_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SET_IP_MASK_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-13 describes the function block for MMC_SetIpMask

Figure 16-13: MMC_SetIpMask function block

16.1.13.2 Function Block Code Example

```
int rc;
MMC_SET_IP_MASK_IN          stSetIPMask_in;
MMC_SET_IP_MASK_OUT         stSetIPMask_out;
//
// Inserting the structure parameters:
stSetIPMask_in.cFirst  = 252;  // First octet IP mask
stSetIPMask_in.cSecond = 183;  // Second octet IP mask
stSetIPMask_in.cThird  = 143;  // Third octet IP mask
stSetIPMask_in.cFourth = 115;  // Fourth octet IP mask
//
rc = MMC_SetIpMaskCmd (hConn, &stSetIPMask_in, &stSetIPMask_out);
if (rc != 0)
{
    HandleError();
}
```

16.1.14 MMC_SetServerIp

Set the Server IP address of the host.

```
MMC_LIB_API int MMC_SetServerIpCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SET_SERVERIP_IN* pInParam,
OUT MMC_SET_SERVERIP_OUT* pOutParam
);
```

Motion Mode NC - Not Relevant Distributed - Not Relevant

Source GMAS\includes\MMC_network_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SET_SERVERIP_IN** input data structure using the MMC_SetServerIp function.

pOutParam

Points to the **MMC_SET_SERVERIP_OUT** output structure receiving information, as a result of calling the MMC_SetServerIp function.

Remarks

When new, by default, the host IP address is set to 192.168.1.2.

Scope

Not limited

MMC_SET_SERVERIP_IN Structure

```
typedef struct {
unsigned char cFirst;
unsigned char cSecond;
unsigned char cThird;
unsigned char cFourth;
}MMC_SET_SERVERIP_IN;
```

Parameters

cFirst

First octet IP address. Any three digit character number up to 255

cSecond

Second octet IP address. Any three digit character number up to 255

cThird

Third octet IP address. Any three digit character number up to 255

cFourth

Fourth octet IP address. Any three digit character number up to 255

MMC_SET_SERVERIP_OUT Structure

```
typedef struct {
unsigned short usStatus
short sErrorID;
}MMC_SET_SERVERIP_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-14 describes the function block for MMC_SetServerIp

Figure 16-14: MMC_SetServerIp function block

16.1.14.2 Function Block Code Example

```
int rc;
MMC_SET_SERVERIP_IN         stSetServerIP_in;
MMC_SET_SERVERIP_OUT        stSetServerIP_out;
//
// Inserting the structure parameters:
stSetServerIP_in.cFirst     = 148;  // First octet IP mask
stSetServerIP_in.cSecond    = 148;  // Second octet IP mask
stSetServerIP_in.cThird     = 0;    // Third octet IP mask
stSetServerIP_in.cFourth    = 1;    // Fourth octet IP mask
//
rc = MMC_SetServerIpCmd (hConn, &stSetServerIP_in, &stSetServerIP_out);
if (rc != 0)
{
    HandleError();
}
```

16.2 Host Communication

Host communications consists of Modbus communications and will in the future consist of further communication devices and protocols.

16.3 Modbus Communication Function Blocks

The Modbus interface allows the client to communicate using TCP/IP protocol with a Maestro compiled program, and manipulate the functions of axis motions via the windows client Modbus program. Modbus always uses default port 502, to poll data from Maestro, and update shared data.

Registry values adjust the movement of the drive, whereas Coil values act as switches, and are therefore 0 or 1. The server API operates the Modbus using specific Registry or Coil tables, which are:

- Read/write to register values
- Read/write to coils Boolean values
- Read input coils
- Close Modbus

This is performed using a specific thread in *MultiAxisControl* that listens to a specific port, for changes in values in registers/coils, through client's Modbus application. However, the Modbus Read values for both the coils and registry tables cannot be input from external sources.

The server can read/write to specific registers/coils and use the values of the Modbus registers/coils, to start/stop axes, or move engines to specific destinations using input parameters that come from Modbus. These parameters can be altered through the application connected to the Maestro server.

The Modbus application can connect to a specific Maestro IP using a specific table ID opened using the Maestro's client application, start address, number of available values, read/write registers, read/write coils, and via the Modbus application, alter running axes, and movement of axes through values shared between the Maestro client's C application, and the Windows Modbus application.

The Windows Modbus application can alter the table ID it uses, read/write registers, read/write coils, read inputs, and change the refresh rates of Modbus shared data displayed in the application.

The following Modbus communication function blocks are described:

Modbus Communication
MMC_MbusIsRunning
MMC_MbusReadCoilsTable
MMC_MbusReadHoldingRegisterTable
MMC_MbusReadInputsTable
MMC_MbusStartServer
MMC_MbusStopServer
MMC_MbusWriteCoilsTable
MMC_MbusWriteHoldingRegisterTable

16.3.1 MMC_MbusIsRunning

Signals that the Modbus connection is operational.

```
MMC_LIB_API int MMC_MbusIsRunning(
IN MMC_CONNECT_HNDL hConn,
IN MMC_MODBUSISRUNNING_IN* pInParam,
OUT MMC_MODBUSISRUNNING_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_host_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_MODBUSISRUNNING_IN** input data structure using the MMC_MbusIsRunning function.

pOutParam

Points to the **MMC_MODBUSISRUNNING_OUT** output structure receiving information, as a result of calling the MMC_MbusIsRunning function.

Remarks

This function block checks whether the Modbus thread is running, and returns the *isrunning* parameter with value 1, if the Modbus is running.

Scope

Not limited

MMC_MODBUSISRUNNING_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_MODBUSISRUNNING_IN;
```

Parameters

dummy

Modbus is connected to the server ID, with the following values:

MODBUS_NOT_STARTED = 0

MODBUS_RUNNING=1

MMC_MODBUSISRUNNING_OUT Structure

```
typedef struct{
unsigned short isrunning;
unsigned short usStatus;
short sErrorID;
}MMC_MODBUSISRUNNING_OUT;
```

Parameters

isrunning

Returns 1 if Modbus is running (MODBUS_RUNNING), otherwise 0 (MODBUS_NOT_STARTED).

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-15 describes the function block for MMC_MbusIsRunning as applied within the IEC 61131 programming.

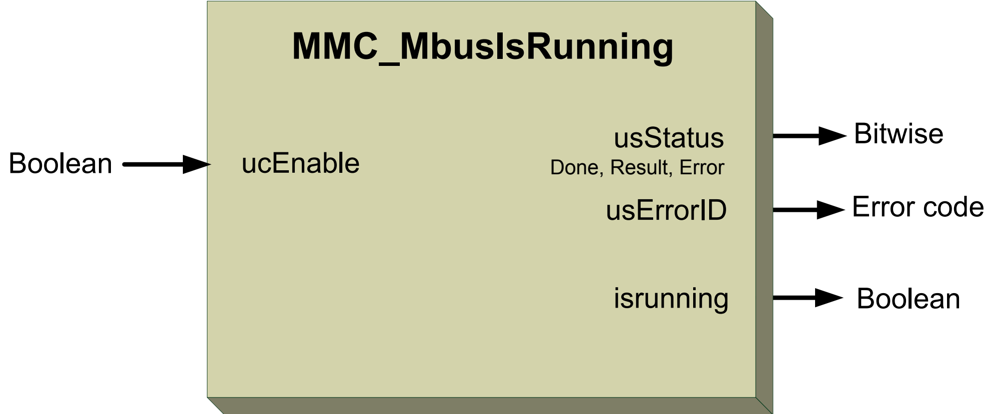


Figure 16-15: MMC_MbusIsRunning function block

16.3.1.2 Function Block Code Example

```
int rc;
MMC_MODBUSISRUNNING_IN      stMbusIsRunning_in;
MMC_MODBUSISRUNNING_OUT     stMbusIsRunning_out;
//
// Inserting the structure parameters:
stMbusIsRunning_in.dummy  = 1;      // Modbus is running (Boolean)
//
rc = MMC_MbusIsRunning (hConn, &stMbusIsRunning_in, &stMbusIsRunning_out);
if (rc != 0)
{
    HandleError();
}
```

16.3.2 MMC_MbusReadCoilsTable

Reads part of Modbus coils table.

```
MMC_LIB_API int MMC_MbusReadCoilsTable(
IN MMC_CONNECT_HNDL hConn,
IN MMC_MODBUSREADCOILS_IN* pInParam,
OUT MMC_MODBUSREADCOILS_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_host_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_MODBUSREADCOILS_IN** input data structure using the MMC_MbusReadCoilsTable function.

pOutParam

Points to the **MMC_MODBUSREADCOILS_OUT** output structure receiving information, as a result of calling the MMC_MbusReadCoilsTable function.

Remarks

Reads the coils table inside the Modbus where every value >0, is similar to Boolean value 1 in the coil. The function block variables include start reference, and reference count number of parameters to read. The internal output parameter is *coilsArr* with Modbus values.

Scope

Not limited

MMC_MODBUSREADCOILS_IN Structure

```
typedef struct{
int startRef;
int refCnt;
}MMC_MODBUSREADCOILS_IN;
```

Parameters

startRef

Start Reference from the base coil table of linear parameters. Any +ve integer values accepted

refCnt

Reference count. Any +ve integer values

MMC_MODBUSREADCOILS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
char coilsArr[MODBUS_IPC_READ_VALUES];
}MMC_MODBUSREADCOILS_OUT;
```

Parameters

coilsArr

Value of the coils array, with 250 as the maximum number of items to read from Modbus coils table. Array of +ve string values.

[MODBUS_IPC_READ_VALUES] has a an array value of [0….250]

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs.** -ve or +ve integer values

Figure 16-16 describes the function block for MMC_MbusReadCoilsTable as applied within the IEC 61131 programming.

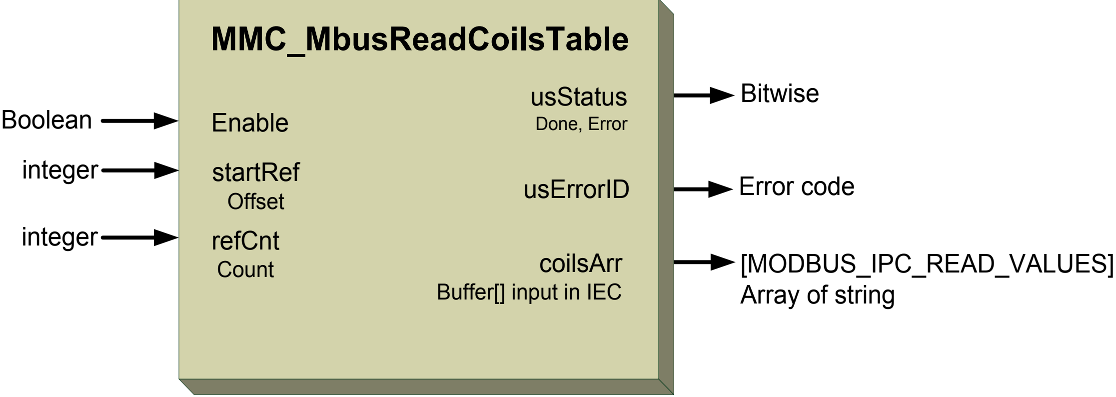


Figure 16-16: MMC_MbusReadCoilsTable function block

16.3.2.2 Function Block Code Example

```
int rc;
MMC_MODBUSREADCOILS_IN     stMbusReadCoils_in;
MMC_MODBUSREADCOILS_OUT    stMbusReadCoils_out;
//
// Inserting the structure parameters:
stMbusReadCoils_in.startRef     = 0;  // Start Reference from the base coil table of linear
parameters
stMbusReadCoils_in.refCnt       = 249;// Reference count

//
rc = MMC_MbusReadCoilsTable (hConn, &stMbusReadCoils_in, &stMbusReadCoils_out);
printf("Mbus Coils Table Status[%ld][%ld] ErrId[%d]\n", (long
int)stMbusReadCoils_out.coilsArr[0], (long int)stMbusReadCoils_out.coilsArr[1],
(short)stMbusReadCoils_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.3.3 MMC_MbusReadHoldingRegisterTable

Reads part of Modbus holding register table.

```
MMC_LIB_API int MMC_MbusReadHoldingRegisterTable(
IN MMC_CONNECT_HNDL hConn,
IN MMC_MODBUSREADHOLDINGREGISTERSTABLE_IN *pInParam,
OUT MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT *pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_host_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_MODBUSREADHOLDINGREGISTERSTABLE_IN** input data structure using the MMC_MbusReadHoldingRegisterTable function.

pOutParam

Points to the **MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT** output structure receiving information, as a result of calling the MMC_MbusReadHoldingRegisterTable function.

Remarks

Reads the registers table inside the Modbus, with parameters including, start reference, and reference count number of parameters to read. The internal output parameter is *regArr* with Modbus values.

Scope

Not limited

MMC_MODBUSREADHOLDINGREGISTERSTABLE_IN Structure

```
typedef struct{
int startRef;
int refCnt;
}MMC_MODBUSREADHOLDINGREGISTERSTABLE_IN;
```

Parameters

startRef

Start Reference from the base holding register table of linear parameters. Any +ve integer values accepted

refCnt

Reference count. Any +ve integer values

MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT Structure

```
typedef struct{
short regArr[MODBUS_IPC_READ_VALUES];
unsigned short usStatus;
short sErrorID;
}MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT;
```

Parameters

regArr

Displays the array values of the registry tables, with 250 as the maximum number of items to read from Modbus registry table. Array of +ve string values.

[MODBUS_IPC_READ_VALUES] has a an array value of [0….250]

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs.** -ve or +ve integer values

Figure 16-17 describes the function block for MMC_MbusReadHoldingRegisterTable as applied within the IEC 61131 programming.

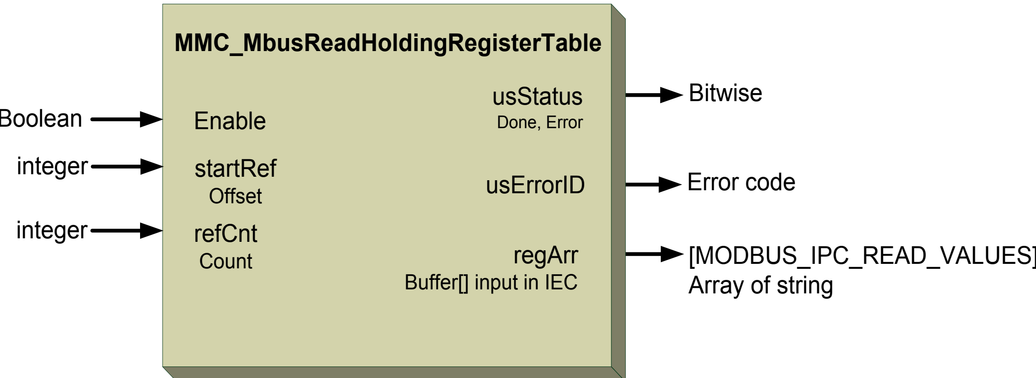


Figure 16-17: MMC_MbusReadHoldingRegisterTable function block

16.3.3.2 Function Block Code Example

```
int rc;
MMC_MODBUSREADHOLDINGREGISTERSTABLE_IN       stMbusReadHoldingTable_in;
MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT      stMbusReadHoldingTable_out;
//
// Inserting the structure parameters:
stMbusReadHoldingTable_in.startRef  = 0;//Start Reference from the base coil table of linear
parameters
stMbusReadHoldingTable_in.refCnt    = 249;// Reference count
//
rc = MMC_MbusReadHoldingRegisterTable (hConn, &stMbusReadHoldingTable_in,
&stMbusReadHoldingTable_out);
printf("Mbus Read Holding Register Table Status[%ld][%ld] ErrId[%d]\n", (long
int)stMbusReadHoldingTable_out.regArr[0], (long int)stMbusReadHoldingTable_out.regArr[1],
(short)stMbusReadHoldingTable_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.3.4 MMC_MbusReadInputsTable

Reads inputs to the Modbus Inputs Table.

```
MMC_LIB_API int MMC_MbusReadInputsTable(
IN MMC_CONNECT_HNDL hConn,
IN MMC_MODBUSREADINPUTS_IN *pInParam,
OUT MMC_MODBUSREADINPUTS_OUT *pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_host_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_MODBUSREADINPUTS_IN** input data structure using the MMC_MbusReadInputsTable function.

pOutParam

Points to the **MMC_MODBUSREADINPUTS_OUT** output structure receiving information, as a result of calling the MMC_MbusReadInputsTable function.

Remarks

Reads the Inputs table inside the Modbus, with parameters including, start reference, and reference count number of parameters to read. The internal output parameter is *InputsArr* with Modbus values. The inputs table cannot be changed by the Windows Modbus application, or using this API.

Scope

Not limited

MMC_MODBUSREADINPUTS_IN Structure

```
typedef struct{
int startRef;
int refCnt;
}MMC_MODBUSREADINPUTS_IN;
```

Parameters

startRef

Start reference from the base inputs table of linear parameters. Any +ve integer values accepted

refCnt

Reference count. Any +ve integer values

MMC_MODBUSREADINPUTS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
char inputsArr[MODBUS_IPC_READ_VALUES];
}MMC_MODBUSREADINPUTS_OUT;
```

Parameters

inputsArr

Value of the inputs array, with 250 as the maximum number of items to read from the Modbus inputs table. Array of +ve string values.

[MODBUS_IPC_READ_VALUES] has a an array value of [0….250]

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs.** -ve or +ve integer values

Figure 16-18 describes the function block for MMC_MbusReadInputsTable

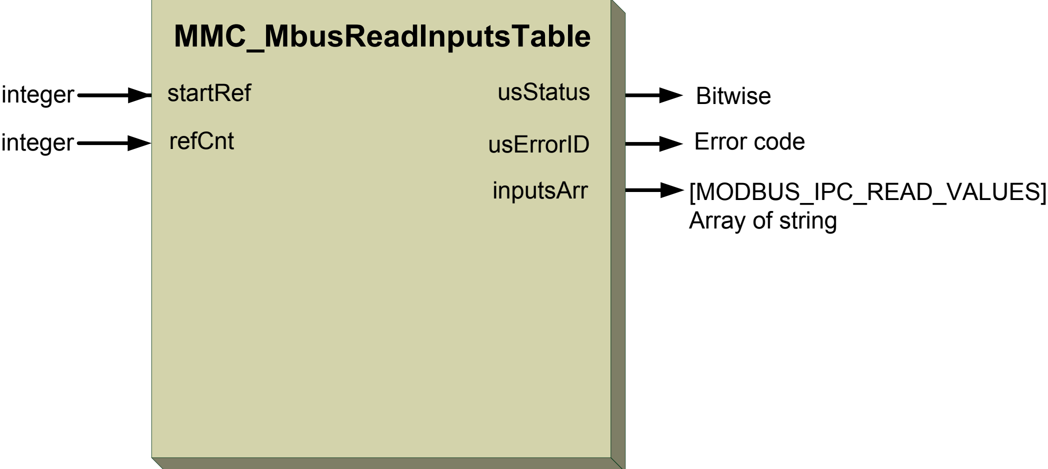


Figure 16-18: MMC_MbusReadInputsTable function block

16.3.4.2 Function Block Code Example

```
int rc;
MMC_MODBUSREADINPUTS_IN         stMbusReadInputs_in;
MMC_MODBUSREADINPUTS_OUT        stMbusReadInputs_out;
//
// Inserting the structure parameters:
stMbusReadInputs_in.startRef    = 0;  // Start Reference from the base coil table of linear
parameters
stMbusReadInputs_in.refCnt      = 250;// Reference count
//
rc = MMC_MbusReadInputsTable (hConn, &stMbusReadInputs_in, &stMbusReadInputs_out);
printf("Mbus Read Inputs Table Status[%ld][%ld] ErrId[%d]\n", (long
int)stMbusReadInputs_out.inputsArr[0], (long int)stMbusReadInputs_out.inputsArr[1],
(short)stMbusReadInputs_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.3.5 MMC_MbusStartServer

Starts the Modbus server listening thread with an ID value as a parameter.

```
MMC_LIB_API int MMC_MbusStartServer(
IN MMC_CONNECT_HNDL hConn,
IN MMC_MODBUSSTARTSERVER_IN *pInParam,
OUT MMC_MODBUSSTARTSERVER_OUT *pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_host_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_MODBUSSTARTSERVER_IN** input data structure using the MMC_MbusStartServer function.

pOutParam

Points to the **MMC_MODBUSSTARTSERVER_OUT** output structure receiving information, as a result of calling the MMC_MbusStartServer function.

Remarks

None

Scope

Not limited

MMC_MODBUSSTARTSERVER_IN Structure

```
typedef struct{
unsigned short id;
}MMC_MODBUSSTARTSERVER_IN;
```

Parameters

id

Modbus start server enumerator ID has the following values:

MODBUS_NOT_STARTED = 0

MODBUS_RUNNING = 1

MODBUS_STOPPED = 2

After the Maestro is powered-up, Modbus server is in the MODBUS_NOT_STARTED state – Initial state, the Modbus server does not exist.

After the Modbus server state is changed to MODBUS _RUNNING – the Modbus server is created, transmissions from Modbus clients will be handled by the server.

When the Modbus server state is changed to MODBUS_STOPPED – the Modbus server connection is removed, no transmissions will be handled from different Modbus clients.

MMC_MODBUSSTARTSERVER_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_MODBUSSTARTSERVER_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs.** -ve or +ve integer values

Figure 16-19 describes the function block for MMC_MbusStartServer

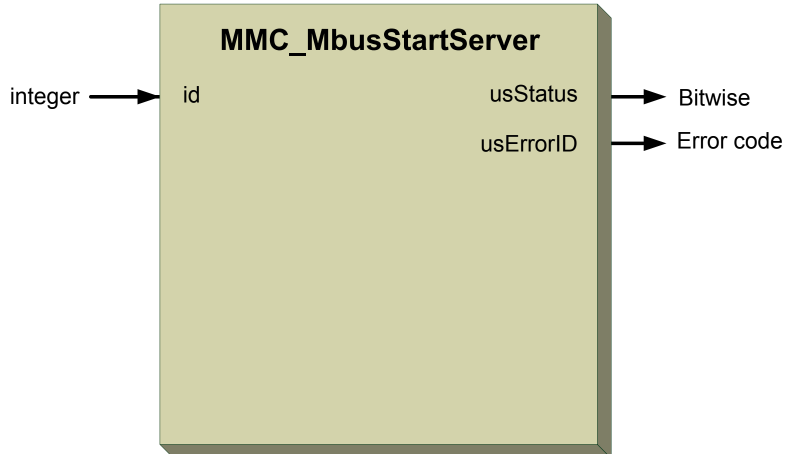


Figure 16-19: MMC_MbusStartServer function block

16.3.5.2 Function Block Code Example

```
int rc;
MMC_MODBUSSTARTSERVER_IN        stMbusStartServer_in;
MMC_MODBUSSTARTSERVER_OUT       stMbusStartServer_out;
//
// Inserting the structure parameters:
stMbusStartServer_in.id    = 1;     // Modbus start server enumerator ID
//
rc = MMC_MbusStartServer (hConn, &stMbusStartServer_in, &stMbusStartServer_out);
if (rc != 0)
{
    HandleError();
}
```

16.3.6 MMC_MbusStopServer

Stops the Modbus server listening thread.

```
MMC_LIB_API int MMC_MbusStopServer(
IN MMC_CONNECT_HNDL hConn,
IN MMC_MODBUSSTOPSERVER_IN* pInParam,
OUT MMC_MODBUSSTOPSERVER_OUT *pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_host_comm_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_MODBUSSTOPSERVER_IN** input data structure using the MMC_MbusStopServer function.

pOutParam

Points to the **MMC_MODBUSSTOPSERVER_OUT** output structure receiving information, as a result of calling the MMC_MbusStopServer function.

Remarks

None

Scope

Not limited

MMC_MODBUSSTOPSERVER_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_MODBUSSTOPSERVER_IN;
```

Parameters

dummy

Dummy Modbus stops server input. 0, 1, 2 integer values accepted.

MMC_MODBUSSTOPSERVER_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_MODBUSSTOPSERVER_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**.

Figure 16-20 describes the function block for MMC_MbusStopServer

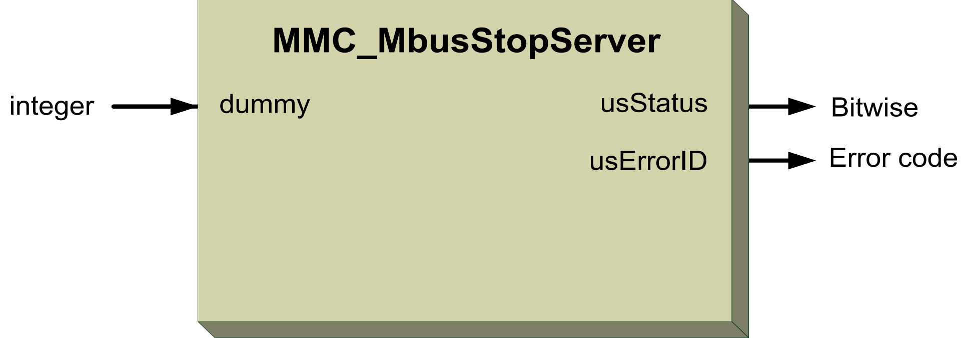


Figure 16-20: MMC_MbusStopServer function block

16.3.6.2 Function Block Code Example

```
int rc;
MMC_MODBUSSTOPSERVER_IN         stMbusStopServer_in;
MMC_MODBUSSTOPSERVER_OUT        stMbusStopServer_out;
//
// Inserting the structure parameters:
stMbusStopServer_in.dummy  = 0;  // Modbus stops server input
//
rc = MMC_MbusStopServer (hConn, &stMbusStopServer_in, &stMbusStopServer_out);
if (rc != 0)
{
    HandleError();
}
```

16.3.7 MMC_MbusWriteCoilsTable

Writes to part of Modbus coils table inside the Modbus where every parameter >0, is similar to Boolean value 1.

```
MMC_LIB_API int MMC_MbusWriteCoilsTable(
IN MMC_CONNECT_HNDL hConn,
IN MMC_MODBUSWRITECOILS_IN *pInParam,
OUT MMC_MODBUSWRITECOILS_OUT *pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_host_comm_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_MODBUSWRITECOILS_IN** input data structure using the MMC_MbusWriteCoilsTable function.

pOutParam

Points to the **MMC_MODBUSWRITECOILS_OUT** output structure receiving information, as a result of calling the MMC_MbusWriteCoilsTable function.

Remarks

The function parameters include start reference, reference count number of parameters, and table of parameters.

Scope

Not limited

MMC_MODBUSWRITECOILS_IN Structure

```
typedef struct{
int startRef;
int refCnt;
char coilsArr[MODBUS_IPC_WRITE_VALUES];
}MMC_MODBUSWRITECOILS_IN;
```

Parameters

startRef

Start Reference from the base coil table of linear parameters. Any +ve integer values accepted

refCnt

Reference count. Any +ve integer values

coilsArr

Value of the coils array, with 250 as the maximum number of items to read from Modbus coils table. Array of +ve string values.

[MODBUS_IPC_READ_VALUES] has a an array value of [0….250]

MMC_MODBUSWRITECOILS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_MODBUSWRITECOILS_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs.** -ve or +ve integer values

Figure 16-21 describes the function block for MMC_MbusWriteCoilsTable

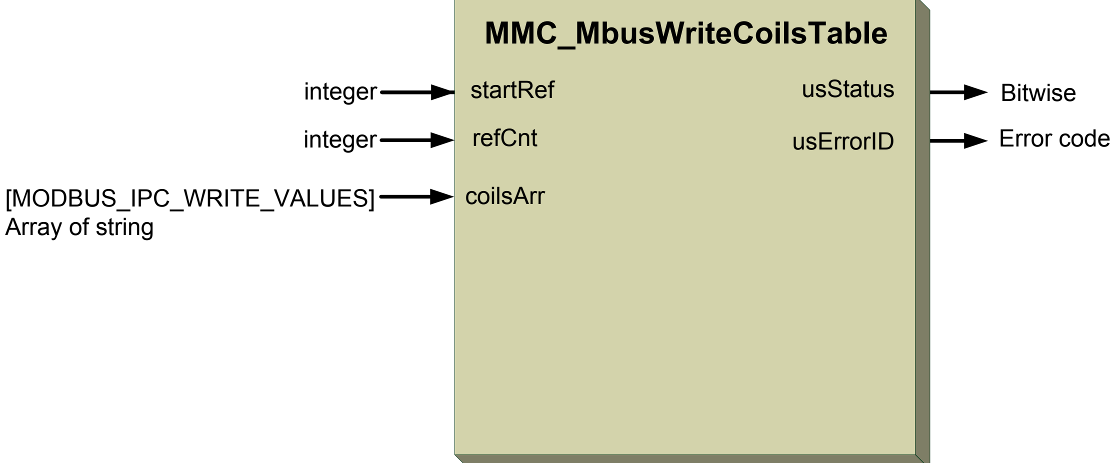


Figure 16-21: MMC_MbusWriteCoilsTable function block

16.3.7.2 Function Block Code Example

```
int rc;
MMC_MODBUSWRITECOILS_IN         stMbusWriteCoils_in;
MMC_MODBUSWRITECOILS_OUT        stMbusWriteCoils_out;
//
// Inserting the structure parameters:
stMbusWriteCoils_in.startRef        = 0;   // Start Reference from the base coil table of
linear parameters
stMbusWriteCoils_in.refCnt          = 249; // Reference count
stMbusWriteCoils_in.coilsArr[10]    = 2;   // Reference count
//
rc = MMC_MbusWriteCoilsTable (hConn, &stMbusWriteCoils_in, &stMbusWriteCoils_out);
if (rc != 0)
{
    HandleError();
}
```

16.3.8 MMC_MbusWriteHoldingRegisterTable

Writes to part of the Modbus register table inside the Modbus.

```
MMC_LIB_API int MMC_MbusWriteHoldingRegisterTable(
IN MMC_CONNECT_HNDL hConn,
IN MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN* pInParam,
OUT MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_host_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN** input data structure using the MMC_MbusWriteHoldingRegisterTable function.

pOutParam

Points to the **MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_OUT** output structure receiving information, as a result of calling the MMC_MbusWriteHoldingRegisterTable function.

Remarks

The function parameters include start reference, reference count number of parameters to write, and the register table with values to write into the Modbus table.

Scope

Not limited

MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN Structure

```
typedef struct{
int startRef;
int refCnt;
short regArr[MODBUS_IPC_WRITE_VALUES];
}MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN;
```

Parameters

startRef

Start Reference from the base coil table of linear parameters. Any +ve integer values accepted

refCnt

Reference count. Any +ve integer values

regArr

Array value of the register table, with 250 as the maximum number of items to write from the Modbus registry table.

[MODBUS_IPC_READ_VALUES] has a an array value of [0….250]

MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs.** -ve or +ve integer values

Figure 16-22 describes the function block for MMC_MbusWriteHoldingRegisterTable

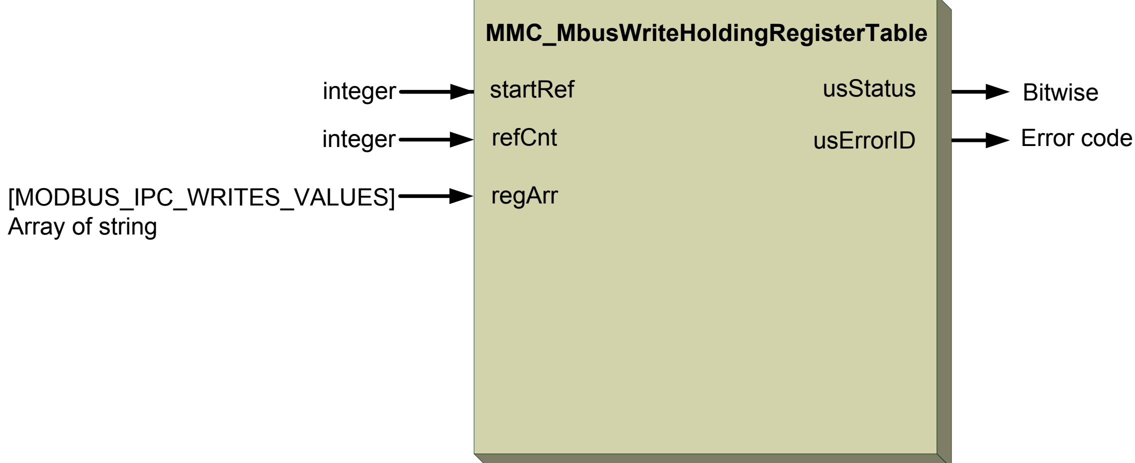


Figure 16-22: MMC_MbusWriteHoldingRegisterTable function block

16.3.8.2 Function Block Code Example

```
int rc;
MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN      stMbusWriteHoldingRegTable_in;
MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_OUT     stMbusWriteHoldingRegTable_out;
//
// Inserting the structure parameters:
stMbusWriteHoldingRegTable_in.startRef   = 0;      // Start Reference from the base coil table of
                                                      linear parameters
stMbusWriteHoldingRegTable_in.refCnt     = 249;    // Reference count
stMbusWriteHoldingRegTable_in.regArr[10] = 65534;  // Array value of the register table
//
rc = MMC_MbusWriteHoldingRegisterTable (hConn, &stMbusWriteHoldingRegTable_in,
                                                 &stMbusWriteHoldingRegTable_out);
if (rc != 0)
{
    HandleError();
}
```

16.4 CANbus Drive Communication

This section refers to the CANbus motion (DS-402) communication to the Maestro. This form of communication uses the CANopen protocol and device profile specification for embedded systems used in automation. CANopen implements the layers above and includes the network layer. The CANopen standard consists of an addressing scheme, several small communication protocols, and an application layer defined by the device profile. The communication protocols support network management, device monitoring, and communication between nodes, including a simple transport layer for message segmentation/desegmentation. Elmo implements CANbus communications via the Process Data Object (PDO) and/or Service Data Object (SDO) protocols.

The physical layer of CANopen, CANbus, can only transmit short packages consisting of an 11-bit ID, remote transmission request (RTR) bit and 0 to 8 bytes of data. The CANopen standard divides the 11-bit CAN frame ID into a 4-bit function code and 7-bit CANopen node ID. This limits the number of devices in a CANopen network to 127. An extension to the CANbus standard (CAN 2.0 B) allows extended frame IDs of 29 bits. The 11-bit ID of a CAN-frame is called the Communication Object Identifier, or COB-ID. In case of a transmission collision, the bus arbitration used in the CANbus allows the frame with the smallest ID to be transmitted first and without a delay. Since in CANopen frames the first 4 bits of the frame ID are reserved for the function code, giving a low code number for time critical functions ensures the lowest possible delay.

The standard reserves certain COB-IDs to network management and SDO transfers. Some function codes and COB-IDs have to be mapped as standard functions after device initialization, but can be configured for other uses later.

The Process Data Object (PDO) protocol processes real time data among various nodes. Up to 8 bytes (64bits) of data per single PDO may be processed either from or to the Maestro. Elmo uses PDOs of which PDO1 and PDO2 are reserved for internal use, allowing PDO3 and PDO4 for transmission and receiving of operational data.

There are two kinds of PDOs, transmit and receive PDOs (TPDO and RPDO, or as used in Elmo TX PDO and RX PDO). The former is for data coming from the servo drive and Maestro and the latter is for data going to the Maestro, and then servo drive. PDOs can be sent synchronously or asynchronously. Synchronous PDOs are sent after the SYNC message whereas asynchronous messages are sent after an internal or external trigger.

The SDO protocol is used to access set and read values from the object directory of the Host system (SDO server), PLC, or HMI Panel. The Maestro (SDO client) always initiates communication with the SDO server. In CANopen terminology, communication is viewed from the SDO server, so that a read from an object directory results in an SDO upload and a write to directory is an SDO download.

As the object directory values can be larger than the 8 byte limit of a CAN frame, the SDO protocol implements segmentation and desegmentation of longer messages. There are two types of SDO protocols, SDO download/upload, and SDO Block download/upload. The SDO block transfer is a newer addition to the standard, which allows large amounts of data to be transferred with slightly less protocol overhead.

16.4.1 Master – Slave Relations

A CAN master (or client) is a controller that sends requests to nodes to respond to its commands. A CAN slave (or server) responds to the commands issued by the CAN master. The CAN protocol permits both single-master and multiple-master networks. Every servo drive has a unique ID in the range [1…127], but the network master does not require an ID. The slave servo drive never sends an unrequested message, other than emergencies, and only responds to messages addressed to its ID or to broadcast messages, which have an ID of 0. All messages sent by a servo drive are marked with its own ID.

16.4.2 CANopen DS-402 Modes of Operation

The Maestro uses the CANopen DS-402 standard to operate the servo drives in the various defined user motion modes. The operation of the Drive (PDS - Power Drive System) depends on the activated mode of operation. The PDS may implement several modes of operation. Since it is not possible to operate the modes in parallel, the Maestro system activates the required function by selecting a mode of operation, using the correct sequence of commands (compliant to the DS-402 standard). The Maestro control device writes to the modes of operation object to select the operation mode. The drive device provides the modes of operation display object, to indicate the actual activated operation mode. These modes use a controlword, statusword, and/or setpoint, target position etc., which are mode-specific. They are specific background mapping PDOs, which are transparent to the user and operate in the background. The following ID numbered event modes of operation are used by the parameter ***ucMotionMode***, supported by the Drive as per the DS-402 specification:

Modes available	Enumerator	ID
No operational DS-402 mode	OPM402_NO	-1
Profile position mode	OPM402_PROFILE_POSITION_MODE	1
Homing mode	OPM402_HOMING_MODE	6
Interpolated position mode	OPM402_INTERPOLATED_POSITION_MODE	7
Profile velocity mode (partial e.g. servo drives)	OPM402_PROFILE_VELOCITY_MODE	3
Torque profile mode	OPM402_TORQUE_PROFILE_MODE	4
Velocity mode (e.g. frequency converter)		
Cyclic sync position mode (EtherCAT)	OPM402_CYCLIC_SYNC_POSITION_MODE	8
Cyclic sync velocity mode	OPM402_CYCLIC_SYNC_VELOCITY_MODE	9
Cyclic sync torque mode	OPM402_CYCLIC_SYNC_TORQUE_MODE	10

With the exception of the Homing mode, the listed modes of operation use setpoints. In addition, manufacturer-specific modes of operation may also be implemented. These are not limited to setpoints. This may be the case in the new Elmo servo drives, to support the extended Drive Profiler and motion modes available.

The Maestro Axis (Node) State Management mechanism is responsible to operate theDS-402 axis in the correct operation mode, in order to implement the *requestedPLCMotion* Motion mode. This is true for both NC as well as Distributed motions.

16.4.3 PDO Mapping

The PDO mapping determines which CANopen objects are to be mapped as RX or TX PDOs and how each PDO is triggered. Four RX and TX PDOs are designated in the system, and the Maestro defaults PDO3 and PDO4, are not mapped at power up. The PDO mapping can only be performed in the pre-operational, operation states, but the RX or TX PDO itself can be a synchronized PDO, or event driven PDO. The type of PDO is determined by its communication type object:

1 Synchronized PDO – triggered by SYNC command

255 Event driven PDO – triggered by value change

PDO	1		2		3		4	
	RX	TX	RX	TX	RX	TX	RX	TX
PP mode	CW(255)+Target Position	SW(255)+Actual Position	Binary Interpreter		User configurable		User configurable	
PV mode	CW(255)+Target Vel	SW(255) + Actual Vel						
PT mode	CW(255)+Target Torque	SW(1) + Actual Torque						
IP mode	CW(1)+Target Position	SW(1) + Actual Position						
HM mode	CW(255)	SW(255)						

Figure 16-23: DS-402 PDO mapping table

Figure 16-23 displays the DS-402 PDO mapping table where the values in parentheses are the default values, per motion mode. The PDO mapping can be changed dynamically, as a function of the specific DS-402 motion mode. However, some of the available PDO mappings are hard-coded per DS-402 motion mode. The correct mapping is sent to the nodes when the DS-402 motion mode is changed.

The user configurable TX PDO's used with the parameter MC_PDO_TYPE_TXPDO can be configured to the event groups defined by the enumerator NC_COMM_EVENT_GROUP and the variable ucEventGroup(or ucEventType) (enumerator value). These event groups are divided into two sections:

Event Groups	Application values for the variable ucEventGroup
Regular Event Groups	MMC_CONFIGREGULARPARAMEVENTPDO3/4_IN inputs
User Event Groups	MMC_CONFIGUSERPARAMEVENTPDO3/4_IN inputs

The User groups are defined by the parameters received from the Maestro via the EAS application and are included in the table below:

Enumerator value	Explanation
Regular Event Groups	
NC_COMM_EVENT_NO_GROUP	
NC_COMM_EVENT_GROUP1	Position + Velocity (32 + 32 bits)
NC_COMM_EVENT_GROUP2	Position + Digital Inputs (32 + 32 bits)
NC_COMM_EVENT_GROUP3	Digital Inputs + Velocity (32 + 32 bits)
NC_COMM_EVENT_GROUP4	Digital Inputs + Current (32 + 16 bits)
NC_COMM_EVENT_GROUP5	Current + Position (16 + 32 bits)
NC_COMM_EVENT_GROUP6	Current + Velocity (16 + 32 bits)
NC_COMM_EVENT_GROUP11	Digital Inputs (32 bits)
User Event Groups	**(All 32 + 32 bits)**
NC_COMM_EVENT_GROUP7	iUser_1 + Position
NC_COMM_EVENT_GROUP8	iUser_1 + Digital Inputs
NC_COMM_EVENT_GROUP9	fUser_1 + Position
NC_COMM_EVENT_GROUP10	fUser_1 + Digital Inputs
NC_COMM_EVENT_GROUP12	iUser_1 + iUserAux_1
NC_COMM_EVENT_GROUP13	iUser_2 + iUserAux_2
NC_COMM_EVENT_GROUP14	fUser_1 + fUserAux_1
NC_COMM_EVENT_GROUP15	fUser_2 + fUserAux_2
NC_COMM_EVENT_GROUP16	General Purpose group parameter1 (value 0)
NC_COMM_EVENT_GROUP17	General Purpose group parameter2 (value 1)

Once the TX PDO's are received by the Maestro, they are automatically copied to the relevant parameter within the Maestro node. If a User variable is used, they will be copied to the 2 x 2 Network Parameters (Float + Integer) available in the node. These can be used from the user program.

In addition, the user may choose to implement a callback event, by registering the callback, for a specific PDO per node. In this situation, the user sets one of the above parameters (Inputs, Position, and Velocity), then use one of the following functions to return to the last updated object:

- ReadActualPosition
- ReadActualVelocity

- ReadDigitalInputs
- ReadActualTorque

Just as the TX PDO's are sent from the servo drive to the Maestro and then host server, similarly, the host server can send RX PDO's to the Maestro and servo drive. The user configurable RX PDO's uses the parameter MC_PDO_TYPE_RPDO, and can be configured to the following event groups defined by the enumerator NC_COMM_RXEVENT_GROUP and variable ***ucEventGroup (or ucEventType)*** (enumerator value):

Enumerator value	Explanation
User Event Groups	
NC_COMM_RXEVENT_NO_GROUP	
NC_COMM_RXEVENT_GROUP1	iUser_3 + iUserAux_3
NC_COMM_RXEVENT_GROUP2	iUser_4 + iUserAux_4
NC_COMM_RXEVENT_GROUP3	fUser_3 + fUserAux_3
NC_COMM_RXEVENT_GROUP4	fUser_4 + fUserAux_4
NC_COMM_RXEVENT_GROUP5	iUser_3 + fUserAux_3
NC_COMM_RXEVENT_GROUP6	General Purpose group parameter1
NC_COMM_RXEVENT_GROUP7	General Purpose group parameter2

16.4.4 Using Event Groups 16 and 17

This section explains how to use the Event Groups 16 and 17 with the functions MMC_CfgUserParamEvPDO4Cmd or MMC_CfgUserParamEvPDO3Cmd. It is important to note that MMC_CfgUserParamEvPDO4Cmd or MMC_CfgUserParamEvPDO3Cmd do not support DS-406 or DS-401 nodes. Mapping a general PDO is not possible for DS-406 devices, and with a DS-401 node, it is necessary to use the same method as in DS-402, but only with the functions MMC_ConfigGeneralTPDO3 and MMC_ConfigGeneralTPDO3. The ***ucEventType*** parameter field may hold the value of event group 16 or 17.

1. Map the desired TPDO objects by manually sending SDOs to the node. This procedure should be performed in compliance with the device user manual and according to the CANopen standards.

When using event groups 16 and 17, the Maestro will not map anything by itself, but will only store the data arriving after the user has manually mapped the PDO, in conditions that allow extraction by the user.

2. Before mapping the final TPDO objects by sending the last SDO (that triggers the node to send the TPDOs), use the functions MMC_CfgUserParamEvPDO4Cmd or MMC_CfgUserParamEvPDO3Cmd with event groups 16 or 17 and the proper communication parameter defined in the mapping process for SYNC, ASYNC or "on event". All other inputs have no importance.
3. Send the last SDO to Trigger the node to send the TPDOs
4. From now on, the Maestro saves the data which the node sends.
5. The data can be extracted using the function MMC_PDOGeneralReadCmd with the following ucPDONum parameter values:

ucPDONum Value	Action
0	Extracts the data of event group 16,
1	Extracts the data of event group 17.

6. In addition, when the TPDO arrives it is possible to receive an event with the data, refer to the use of the functions MMC_CfgEventModePDO4Cmd and MMC_CfgEventModePDO3Cmd.

16.4.5 Servo Drive Sub-Index

The variable ***ucSubIndex*** uses a single integer to represent the User Integer (UI) and User Float (UF) of the servo drive, dependent on the Regular and User Event Group selected for the variable ***ucEventGroup (or ucEventType)***. Since PDO3 and PDO4 functions are available, ucSubIndex may be set for the PDO3 function to any UI[1], UI[2]… etc., and another UI value for the PDO4 function. Alternatively, the PDO3 function may be set to a UI value, and the PDO4 function set to a UF value (UF[1], UF[2], ..etc.). The limitation is the AxisRef of the servo drive.

16.4.6 SYNC and Time Stamp

The SYNC message has two purposes:

- Synchronize the operation of synchronous PDOs. Only synchronous TPDOs can be used to transmit data from SimplIQ digital servo drives upon receiving a SYNC message.
- Synchronize the motion clock of the servo drive with a clock in the network master (Maestro server).

Synchronization is performed in conjunction with the Time Stamp message. The servo drive motion clock counts microseconds (regardless of the sampling time of the drive). It is cyclic and has 32 bits, and therefore completes a full cycle in 4,295 seconds (approximately 72 minutes). When the motion clocks of all connected servo drives are synchronized to the motion clock of the master (Maestro server), multiple servo drives can perform complex synchronized motion with exact timing set by the network master.

The drives are synchronized by the transmission of a SYNC message, whose arrival time is captured by the drive. Upon receipt of the SYNC, the drive keys in its internal timer. A Time Stamp is a 32-bit message that contains the master internal clock generated, when the client's own SYNC is received. The Time Stamp causes a clock synchronization cycle to be executed. The drive uses the Time Stamp as the absolute timer and adjusts its internal time in relation to the time keyed in at the last SYNC1. To synchronize the master and drive clocks to full precision, the synchronization process is filtered in order to ensure that the timing jitter of the time stamp process does not adversely affect the smoothness of the servo drive motion. It takes about 200 SYNC-Time Stamp pairs to ensure that all clocks are fully synchronized. COB-ID 256 (0x100) is a constant dedicated ID used for this purpose. The master can send Time Stamps at any time. A Time Stamp always refers to the previous SYNC message and must come no later than 5 seconds after the relevant SYNC message.

16.4.7 CAN Bulk Upload

The CAN bulk upload feature is intended to optimize the process of upload of any data from drive to host (recorder data, personality, parameters, etc.). Currently, the whole upload process of data is managed by the host. For example, to upload the recording data buffer, it is necessary to send an SDO message, receive an SDO response, and retrieve the data via UDP. This is a tedious and awkward process that increases the network and bus load.

Elmo proposes that the Maestro will manage the upload process upon request from the host, i.e. the host will send the "Begin Upload" command, and the Maestro will upload the recording buffer. Afterwards, the host will be able to send the "Get Uploaded Data" command, and the data buffer will be returned immediately. During the whole process, a status can be retrieved using the "Get Upload Status" command, which will return to the user, the following data:

- Amount of data received
- Upload state (init/in progress/error/etc.)
- Communication error (if any)
- Process error (if any)

16.4.8 CAN – PDO, SDO Configurator

In addition to PDO settings, the Maestro resource file also supports the ability to send SDO's to devices (DS301, DS402, DS406). These SDO's can be sent in Preoperational or Operational modes. The user can configure in advance, PDOs that are mapped, and SDOs to be sent when the node is in preoperational state or in operational state. The configuration is performed via the EAS application.

16.5 CANbus Function Blocks

The following CANbus drive communication function blocks are described:

Drive Communication
MMC_CancelVirtualEncoder
MMC_CancelParamEvPDO3
MMC_CancelParamEvPDO4
MMC_CfgRegParamEvPDO3
MMC_CfgRegParamEvPDO4
MMC_CfgUserParamEvPDO3
MMC_CfgUserParamEvPDO4
MMC_ChangeDefaultPDOConfiguration
MMC_ChngOpMode
MMC_ConfigEventModePDO3
MMC_ConfigEventModePDO4
MMC_ConfigVirtualEncoder

MMC_GetAxisByCanId
MMC_GetPDOInfo
MMC_GetSyncTime
MMC_PDOGeneralRead
MMC_PDOGeneralWrite
MMC_ReceiveCANRawData
MMC_SendCANRawData
MMC_SendCmd
MMC_SetHeartBeatConsumer
MMC_SetSyncTime
MMC_StartBulkUpload
MMC_GetBulkUploadStatus
MMC_GetBulkUploadData

16.5.1 MMC_CancelVirtualEncoder

This function cancels a defined servo-drive as the virtual CAN encoder.

```
MMC_LIB_API int MMC_CancelVirtualEncoderCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CANCELVIRTUALENCODER_IN* pInParam,
OUT MMC_CANCELVIRTUALENCODER_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_CANCELVIRTUALENCODER_IN** input data structure using the MMC_CancelVirtualEncoder function.

pOutParam

Points to the **MMC_CANCELVIRTUALENCODER_OUT** output structure receiving information, as a result of calling the MMC_CancelVirtualEncoder function.

Remarks

The virtual encoder is based on the DS406 CAN encoder protocol on the network. It allows the user to broadcast the position of a servo drive to the other servo drives connected to the Maestro. Any of the servo-drives may be defined as the encoder, as each of the following functions is axis related:

- MMC_SetPosition
- MMC_ConfigVirtualEncoder
- MMC_CancelVirtualEncoder

Scope

The servo drives must set as a group address. SDO's must be set to manually 'listen' on the group address for the position. The SYNC function should be used to set the Maestro.

MMC_CANCELVIRTUALENCODER_IN Structure

```
typedef struct mmc_cancelvirtualencoder_in{
unsigned char ucDummy;
}MMC_CANCELVIRTUALENCODER_IN;
```

Parameters

ucDummy

Dummy value. Any -ve or +ve character.

MMC_CANCELVIRTUALENCODER_OUT Structure

```
typedef struct mmc_cancelvirtualencoder_out{
unsigned short usStatus;
short sErrorID;
}MMC_CANCELVIRTUALENCODER_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-24 describes the function block for MMC_CancelVirtualEncoder as applied within the IEC 61131 programming.

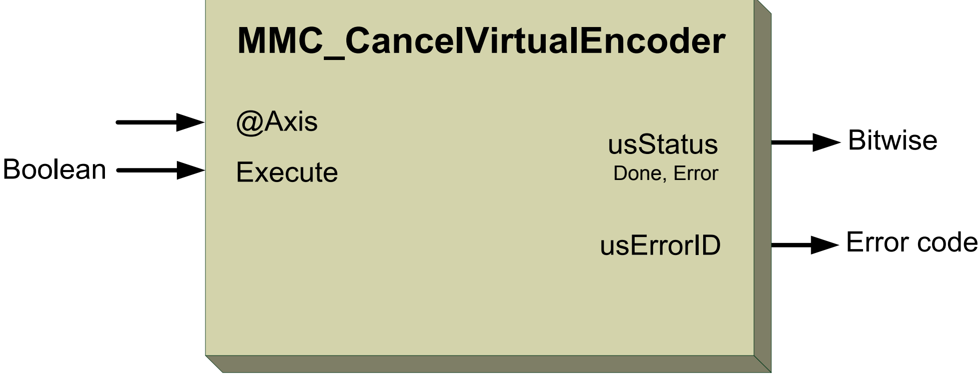


Figure 16-24: MMC_CancelVirtualEncoder function block

16.5.2 MMC_CancelParamEvPDO3

Cancels the TPDO3 and RXPDO3 event processing.

```
MMC_LIB_API int MMC_CancelParamEvPDO3Cmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CANCELPARAMEVENTPDO3_IN* pInParam,
OUT MMC_CANCELPARAMEVENTPDO3_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CANCELPARAMEVENTPDO3_IN** input data structure using the MMC_CancelParamEvPDO3 function.

pOutParam

Points to the **MMC_CANCELPARAMEVENTPDO3_OUT** output structure receiving information as a result of calling the MMC_CancelParamEvPDO3 function.

Remarks

None

Scope

All

MMC_CANCELPARAMEVENTPDO3_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_CANCELPARAMEVENTPDO3_IN;
```

Parameters

dummy

Dummy input. Any +ve integer values accepted.

MMC_CANCELPARAMEVENTPDO3_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CANCELPARAMEVENTPDO3_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-25 describes the function block for MMC_CancelParamEvPDO3 as applied within the IEC 61131 programming.

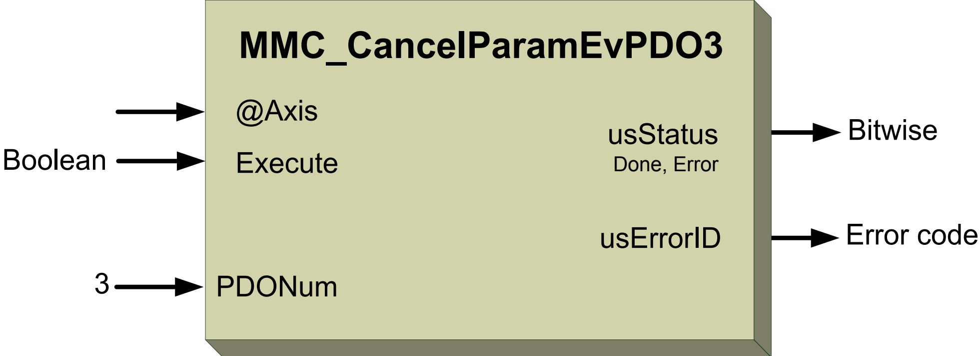


Figure 16-25: MMC_CancelParamEvPDO3 function block

16.5.2.2 Function Block Code Example

```
int rc;
MMC_CANCELPARAMEVENTPDO3_IN     stCancelParamEventPDO3_in;
MMC_CANCELPARAMEVENTPDO3_OUT    stCancelParamEventPDO3_out;
//
// Inserting the structure parameters:
stCancelParamEventPDO3_in.dummy     = 0; //Any dummy inputs
//
rc = MMC_CancelParamEvPDO3Cmd (hConn, iAxisRef, &stCancelParamEventPDO3_in,
&stCancelParamEventPDO3_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.3 MMC_CancelParamEvPDO4

Cancels the TPDO4 and RXPDO4 event processing.

```
MMC_LIB_API int MMC_CancelParamEvPDO3Cmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CANCELPARAMEVENTPDO4_IN* pInParam,
OUT MMC_CANCELPARAMEVENTPDO4_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CANCELPARAMEVENTPDO4_IN** input data structure using the MMC_CancelParamEvPDO4 function.

pOutParam

Points to the **MMC_CANCELPARAMEVENTPDO4_OUT** output structure receiving information as a result of calling the MMC_CancelParamEvPDO4 function.

Remarks

None

Scope

All

MMC_CANCELPARAMEVENTPDO4_IN Structure

typedef struct{
unsigned char dummy;
}MMC_CANCELPARAMEVENTPDO4_IN;

Parameters

dummy

Dummy input. Any +ve integer values accepted.

MMC_CANCELPARAMEVENTPDO4_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CANCELPARAMEVENTPDO4_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-26 describes the function block for MMC_CancelParamEvPDO4 as applied within the IEC 61131 programming.

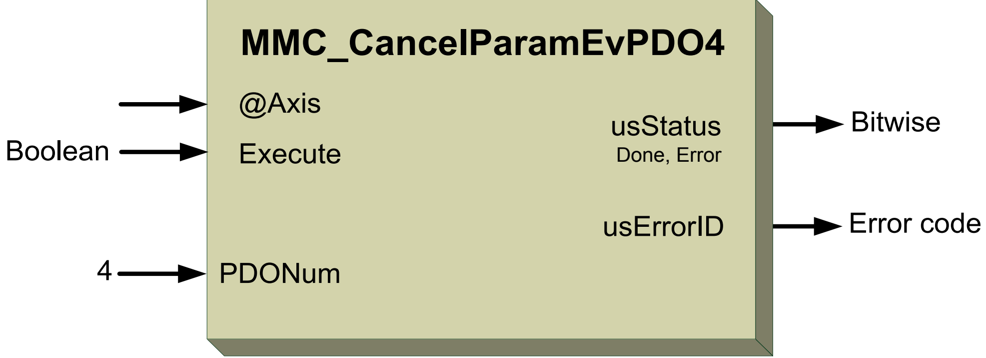


Figure 16-26: MMC_CancelParamEvPDO4 function block

16.5.3.2 Function Block Code Example

```
int rc;
MMC_CANCELPARAMEVENTPDO4_IN     stCancelParamEventPDO4_in;
MMC_CANCELPARAMEVENTPDO4_OUT    stCancelParamEventPDO4_out;
//
// Inserting the structure parameters:
stCancelParamEventPDO4_in.dummy     = 0; //Any dummy inputs
//
rc = MMC_CancelParamEvPDO4Cmd (hConn, iAxisRef, &stCancelParamEventPDO4_in,
&stCancelParamEventPDO4_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.4 MMC_CfgRegParamEvPDO3

Configures regular parameter event PDO3 according group type.

```
MMC_LIB_API int MMC_CfgRegParamEvPDO3Cmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGREGULARPARAMEVENTPDO3_IN* pInParam,
OUT MMC_CONFIGREGULARPARAMEVENTPDO3_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h
GMAS\includes\MMC_DS401_API.h
GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGREGULARPARAMEVENTPDO3_IN** input data structure using the MMC_CfgRegParamEvPDO3 function.

pOutParam

Points to the **MMC_CONFIGREGULARPARAMEVENTPDO3_OUT** output structure receiving information as a result of calling the MMC_CfgRegParamEvPDO3 function.

Remarks

None

Scope

The PDO3 cannot be mapped when the axis is in *ErrorStop* and MMC_Reset must be performed.

Make sure when using this function block in Distributed mode, the Sync Time is set, using the function MMC_SetSyncTime.

MMC_CONFIGREGULARPARAMEVENTPDO3_IN Structure

```
typedef struct{
unsigned int uiPDOCommParamEvent;
unsigned short usEventTimer;
unsigned char ucEventGroup;
unsigned char ucPDOCommParam;
}MMC_CONFIGREGULARPARAMEVENTPDO3_IN;
```

Parameters

uiPDOCommParamEvent

This parameter inserts a PDO Events mechanism in the Maestro for servo drive operations like emit, motion complete, motion started etc. When the operation is performed, an event is sent back to the Maestro, and thereon to any connected host.

Any +ve integer in bitwise form is accepted.

usEventTimer

The timer for the PDO3 event. Has the following conditions depending on the drive specifications. Acceptable values are any integer in millisecs:

0 for Synchronous data

>0 for Asynchronous data

For the following PDO3 event times:

MC_PDO_TIMER_NON = 0

MC_PDO_TIMER_1_MILISEC = 1

MC_PDO_TIMER_2_MILISEC = 2

MC_PDO_TIMER_3_MILISEC = 3

MC_PDO_TIMER_4_MILISEC = 4

MC_PDO_TIMER_5_MILISEC = 5

MC_PDO_TIMER_10_MILISEC = 10

MC_PDO_TIMER_20_MILISEC = 20

MC_PDO_TIMER_25_MILISEC = 25

MC_PDO_TIMER_50_MILISEC = 50

MC_PDO_TIMER_100_MILISEC = 100

MC_PDO_TIMER_150_MILISEC = 150

MC_PDO_TIMER_200_MILISEC = 200

MC_PDO_TIMER_250_MILISEC = 250

MC_PDO_TIMER_255_MILISEC = 255

ucEventGroup

Defines which group of events are to be transferred from the Maestro. Refer to the section 16.4.3 **PDO Mapping on page 1239** the correct definition to be used. Any +ve character values are acceptable.

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC	0x01
PDO_COM_PARAM_ASYNC	0xFF
PDO_COM_PARAM_EVENT	0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

MMC_CONFIGREGULARPARAMEVENTPDO3_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGREGULARPARAMEVENTPDO3_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-27 describes the function block for MMC_CfgRegParamEvPDO3 as applied within the IEC 61131 programming. The IEC function block serves CfgUserParamEvPDO3, CfgUserParamEvPDO4, CfgRegParamEvPDO3, and CfgRegParamEvPDO4 together.

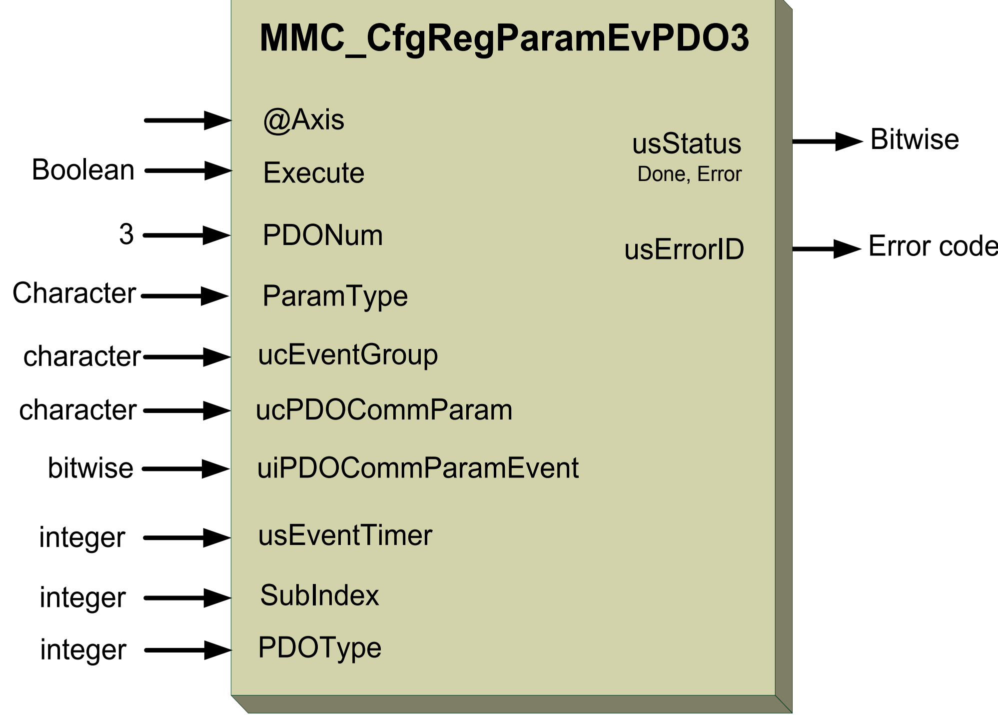


Figure 16-27: MMC_CfgRegParamEvPDO3 function block

16.5.4.2 Function Block Code Example

```
int rc;
MMC_CONFIGREGULARPARAMEVENTPDO3_IN       stConfigRegParamEventPDO3_in;
MMC_CONFIGREGULARPARAMEVENTPDO3_OUT      stConfigRegParamEventPDO3_out;
//
// Inserting the structure parameters:
stConfigRegParamEventPDO3_in.uiPDOCommParamEvent  = 0xDD;                // inserts a PDO
Events mechanism in the GMAS for servo drive operations
stConfigRegParamEventPDO3_in.usEventTimer         = 1;                   //Event timer
stConfigRegParamEventPDO3_in.ucEventGroup         = NC_COMM_EVENT_GROUP1;    //Defines which
group of events are to be transferred from the G MAS
stConfigRegParamEventPDO3_in.ucPDOCommParam       = 0xFE;                //PDO communications
parameter
//
rc = MMC_CfgRegParamEvPDO3Cmd (hConn, iAxisRef, &stConfigRegParamEventPDO3_in,
&stConfigRegParamEventPDO3_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.5 MMC_CfgRegParamEvPDO4

Configures regular parameter event PDO4 according group type.

```
MMC_LIB_API int MMC_CfgRegParamEvPDO4Cmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGREGULARPARAMEVENTPDO4_IN* pInParam,
OUT MMC_CONFIGREGULARPARAMEVENTPDO4_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGREGULARPARAMEVENTPDO4_IN** input data structure using the MMC_CfgRegParamEvPDO4 function.

pOutParam

Points to the **MMC_CONFIGREGULARPARAMEVENTPDO4_OUT** output structure receiving information as a result of calling the MMC_CfgRegParamEvPDO4 function.

Remarks

None

Scope

The PDO4 cannot be mapped when the axis is in *ErrorStop* and MMC_Reset must be performed.

Make sure, when using this function block in Distributed mode, the Sync Time is set, using the function MMC_SetSyncTime.

MMC_CONFIGREGULARPARAMEVENTPDO4_IN Structure

```
typedef struct{
unsigned int uiPDOCommParamEvent;
unsigned short usEventTimer;
unsigned char ucEventGroup;
unsigned char ucPDOCommParam;
}MMC_CONFIGREGULARPARAMEVENTPDO4_IN;
```

Parameters

uiPDOCommParamEvent

This parameter inserts a PDO Events mechanism in the Maestro for servo drive operations like emit, motion complete, motion started etc. When the operation is performed, an event is sent back to the Maestro, and thereon to any connected host.

Any +ve integer in bitwise form is accepted.

usEventTimer

The timer for the PDO4 event. Has the following conditions depending on the drive specifications. Acceptable values are any integer in millisecs:

0 for Synchronous data

>0 for Asynchronous data

For the following PDO4 event times:

MC_PDO_TIMER_NON = 0

MC_PDO_TIMER_1_MILISEC = 1

MC_PDO_TIMER_2_MILISEC = 2

MC_PDO_TIMER_3_MILISEC = 3

MC_PDO_TIMER_4_MILISEC = 4

MC_PDO_TIMER_5_MILISEC = 5

MC_PDO_TIMER_10_MILISEC = 10

MC_PDO_TIMER_20_MILISEC = 20

MC_PDO_TIMER_25_MILISEC = 25

MC_PDO_TIMER_50_MILISEC = 50

MC_PDO_TIMER_100_MILISEC = 100

MC_PDO_TIMER_150_MILISEC = 150

MC_PDO_TIMER_200_MILISEC = 200

MC_PDO_TIMER_250_MILISEC = 250

MC_PDO_TIMER_255_MILISEC = 255

ucEventGroup

Defines which group of events are to be transferred from the Maestro. Refer to the section 16.4.3 **PDO Mapping on page 1239** the correct definition to be used. Any +ve character values are acceptable.

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC	0x01
PDO_COM_PARAM_ASYNC	0xFF
PDO_COM_PARAM_EVENT	0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

MMC_CONFIGREGULARPARAMEVENTPDO4_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGREGULARPARAMEVENTPDO4_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-28 describes the function block for MMC_CfgRegParamEvPDO4 as applied within the IEC 61131 programming. The IEC function block serves CfgUserParamEvPDO3, CfgUserParamEvPDO4, CfgRegParamEvPDO3, and CfgRegParamEvPDO4 together.

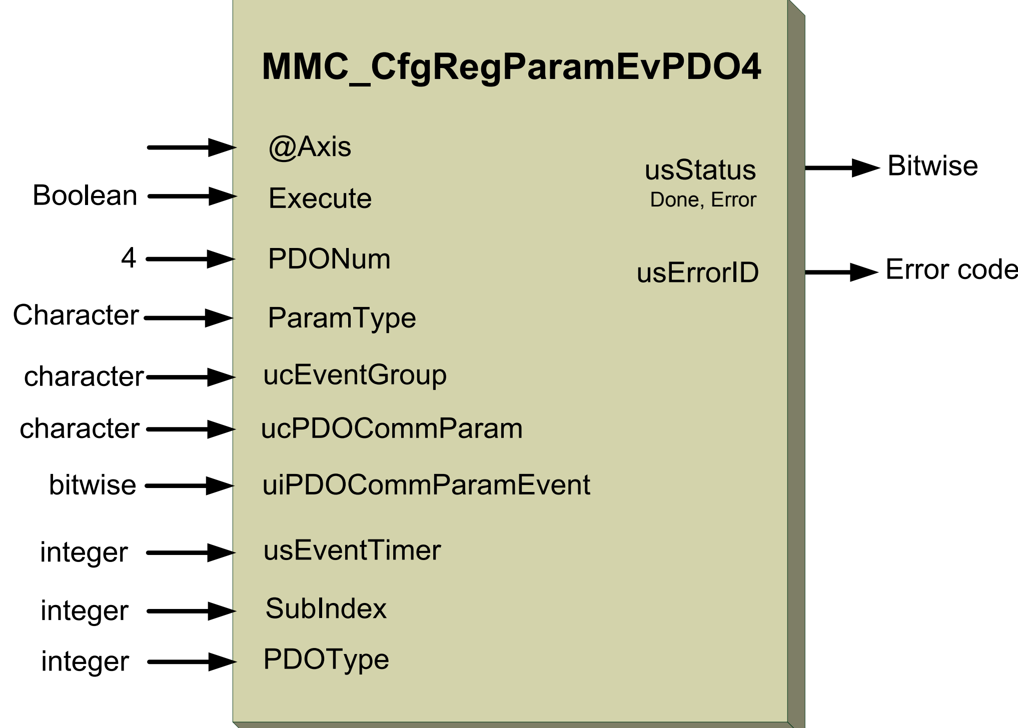


Figure 16-28: MMC_CfgRegParamEvPDO4 function block

16.5.5.2 Function Block Code Example

```
int rc;
MMC_CONFIGREGULARPARAMEVENTPDO4_IN      stConfigRegParamEventPDO4_in;
MMC_CONFIGREGULARPARAMEVENTPDO4_OUT     stConfigRegParamEventPDO4_out;
//
// Inserting the structure parameters:
stConfigRegParamEventPDO4_in.uiPDOCommParamEvent  = 0xDE;               //inserts a PDO Events
mechanism in the GMAS for servo drive operations
stConfigRegParamEventPDO4_in.usEventTimer         = 1;                  //Event timer
stConfigRegParamEventPDO4_in.ucEventGroup         = NC_COMM_EVENT_GROUP1;//Defines which group
of events are to be transferred from the G MAS
stConfigRegParamEventPDO4_in.ucPDOCommParam       = 0xFE;                    //PDO
communications parameter
//
rc = MMC_CfgRegParamEvPDO4Cmd (hConn, iAxisRef, &stConfigRegParamEventPDO4_in,
&stConfigRegParamEventPDO4_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.6 MMC_CfgUserParamEvPDO3

Configures user parameter event PDO3 according to group type.

```
MMC_LIB_API int MMC_CfgUserParamEvPDO3Cmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGUSERPARAMEVENTPDO3_IN* pInParam,
OUT MMC_CONFIGUSERPARAMEVENTPDO3_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGUSERPARAMEVENTPDO3_IN** input data structure using the MMC_CfgUserParamEvPDO3 function.

pOutParam

Points to the **MMC_CONFIGUSERPARAMEVENTPDO3_OUT** output structure receiving information as a result of calling the MMC_CfgUserParamEvPDO3 function.

Remarks

None

Scope

For both NC and Distributed modes. To use this function in distributed mode with MC_PDO_COMM_ON_SYNC , the function block MMC_SetSyncTime should be called first.

MMC_CONFIGUSERPARAMEVENTPDO3_IN Structure

```
typedef struct{
unsigned int uiPDOCommParamEvent;
unsigned short usEventTimer;
unsigned char ucEventGroup;
unsigned char ucSubIndex;
unsigned char ucPDOCommParam;
unsigned char ucPDOType;
}MMC_CONFIGUSERPARAMEVENTPDO3_IN;
```

Parameters

uiPDOCommParamEvent

This parameter inserts a PDO Events mechanism in the Maestro for servo drive operations like emit, motion complete, motion started etc. When the operation is performed, an event is sent back to the Maestro, and thereon to any connected host.

Any +ve integer in bitwise form is accepted.

usEventTimer

The timer for the PDO3 event. Has the following conditions depending on the drive specifications. Acceptable values are any integer in millisecs:

0 for Synchronous data

>0 for Asynchronous data

For the following PDO3 event times:

MC_PDO_TIMER_NON = 0

MC_PDO_TIMER_1_MILISEC = 1

MC_PDO_TIMER_2_MILISEC = 2

MC_PDO_TIMER_3_MILISEC = 3

MC_PDO_TIMER_4_MILISEC = 4

MC_PDO_TIMER_5_MILISEC = 5

MC_PDO_TIMER_10_MILISEC = 10

MC_PDO_TIMER_20_MILISEC = 20

MC_PDO_TIMER_25_MILISEC = 25

MC_PDO_TIMER_50_MILISEC = 50

MC_PDO_TIMER_100_MILISEC = 100

MC_PDO_TIMER_150_MILISEC = 150

MC_PDO_TIMER_200_MILISEC = 200

MC_PDO_TIMER_250_MILISEC = 250

MC_PDO_TIMER_255_MILISEC = 255

ucEventGroup

Defines which group of events are to be transferred from the Maestro. Refer to the section 16.4.3 **PDO Mapping on page 1239** the correct definition to be used. Any +ve character values are acceptable.

ucSubIndex

Defines which index value signifies User Integer and User Float values of the servo drive. Refer to the section **16.4.4 Using Event Groups 16 and 17**. Any +ve character integers are accepted as values

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC 0x01

PDO_COM_PARAM_ASYNC 0xFF

PDO_COM_PARAM_EVENT 0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

ucPDOType

The direction of the PDO, according to the MC_PDO_TYPE_ENUM enumerator:

MC_PDO_TYPE_RPDO

MC_PDO_TYPE_TXPDO

This will be dependent on the value of the parameters ucEventGroup, and ucSubIndex.

MMC_CONFIGUSERPARAMEVENTPDO3_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGUSERPARAMEVENTPDO3_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-29 describes the function block for MMC_CfgUserParamEvPDO3 as applied within the IEC 61131 programming. The IEC function block serves CfgUserParamEvPDO3, CfgUserParamEvPDO4, CfgRegParamEvPDO3, and CfgRegParamEvPDO4 together.

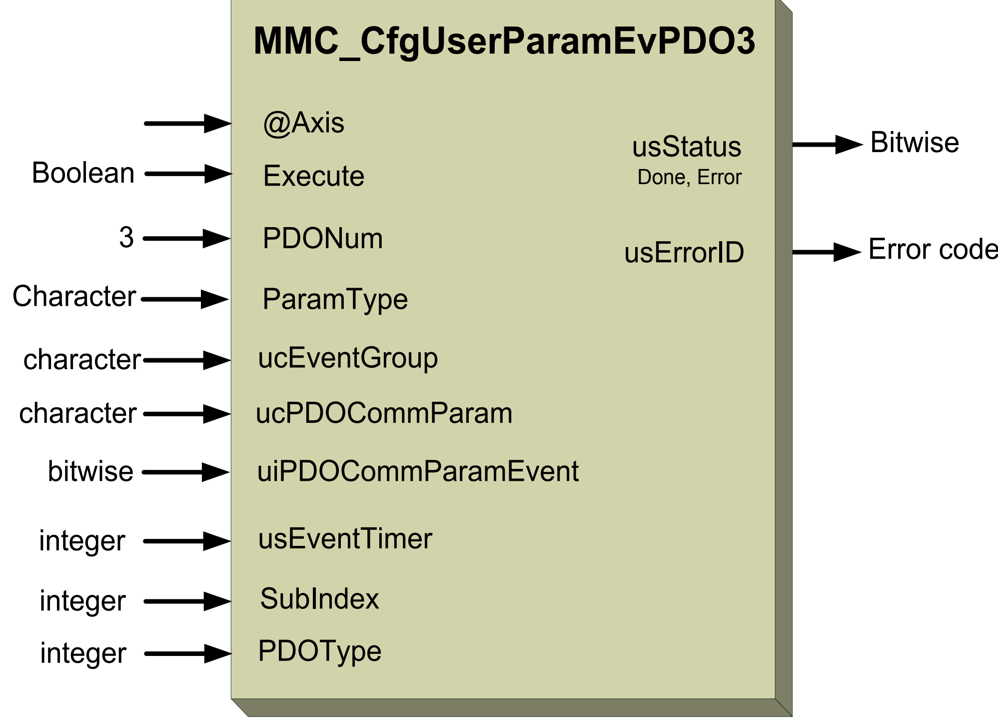


Figure 16-29: MMC_CfgUserParamEvPDO3 function block

16.5.6.2 Function Block Code Example

```
int rc;
MMC_CONFIGUSERPARAMEVENTPDO3_IN          stConfigUserParamEventPDO3_in;
MMC_CONFIGUSERPARAMEVENTPDO3_OUT         stConfigUserParamEventPDO3_out;
//
```

```
// Inserting the structure parameters:
stConfigUserParamEventPDO3_in.uiPDOCommParamEvent = 0xDD;                //inserts a PDO Events
mechanism in the GMAS for servo drive operations
stConfigUserParamEventPDO3_in.usEventTimer        = 2;                   //Event timer
stConfigUserParamEventPDO3_in.ucEventGroup        = NC_COMM_EVENT_GROUP7;//Defines which group
of events are to be transferred from the G-MAS
stConfigUserParamEventPDO3_in.ucSubIndex          = 2,1;                 //Defines which index
value signifies the group of events to be transferred from the G-MAS
stConfigUserParamEventPDO3_in.ucPDOCommParam      = 0xFE;                //PDO communications
parameter
stConfigUserParamEventPDO3_in.ucPDOType           = MC_PDO_TYPE_TXPDO;   //The direction of the
PDO
//
rc = MMC_CfgUserParamEvPDO3Cmd (hConn, iAxisRef, &stConfigUserParamEventPDO3_in,
&stConfigUserParamEventPDO3_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.7 MMC_CfgUserParamEvPDO4

Configures user parameter event PDO4 according to group type.

```
MMC_LIB_API int MMC_CfgUserParamEvPDO4Cmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGUSERPARAMEVENTPDO4_IN* pInParam,
OUT MMC_CONFIGUSERPARAMEVENTPDO4_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGUSERPARAMEVENTPDO4_IN** input data structure using the MMC_CfgUserParamEvPDO4 function.

pOutParam

Points to the **MMC_CONFIGUSERPARAMEVENTPDO4_OUT** output structure receiving information as a result of calling the MMC_CfgUserParamEvPDO4 function.

Remarks

None

Scope

For both NC and Distributed modes. To use this function in distributed mode with MC_PDO_COMM_ON_SYNC , the function block MMC_SetSyncTime should be called first.

MMC_CONFIGUSERPARAMEVENTPDO4_IN Structure

```
typedef struct{
unsigned int uiPDOCommParamEvent;
unsigned short usEventTimer;
unsigned char ucEventGroup;
unsigned char ucSubIndex;
unsigned char ucPDOCommParam;
unsigned char ucPDOType;
}MMC_CONFIGUSERPARAMEVENTPDO4_IN;
```

Parameters

uiPDOCommParamEvent

This parameter inserts a PDO Events mechanism in the Maestro for servo drive operations like emit, motion complete, motion started etc. When the operation is performed, an event is sent back to the Maestro, and thereon to any connected host.

Any +ve integer in bitwise form is accepted.

usEventTimer

The timer for the PDO4 event. Has the following conditions depending on the drive specifications. Acceptable values are any integer in millisecs:

0 for Synchronous data

>0 for Asynchronous data

For the following PDO4 event times:

MC_PDO_TIMER_NON = 0

MC_PDO_TIMER_1_MILISEC = 1

MC_PDO_TIMER_2_MILISEC = 2

MC_PDO_TIMER_3_MILISEC = 3

MC_PDO_TIMER_4_MILISEC = 4

MC_PDO_TIMER_5_MILISEC = 5

MC_PDO_TIMER_10_MILISEC = 10

MC_PDO_TIMER_20_MILISEC = 20

MC_PDO_TIMER_25_MILISEC = 25

MC_PDO_TIMER_50_MILISEC = 50

MC_PDO_TIMER_100_MILISEC = 100

MC_PDO_TIMER_150_MILISEC = 150

MC_PDO_TIMER_200_MILISEC = 200

MC_PDO_TIMER_250_MILISEC = 250

MC_PDO_TIMER_255_MILISEC = 255

ucEventGroup

Defines which group of events are to be transferred from the Maestro. Refer to the section 16.4.3 **PDO Mapping on page 1239** the correct definition to be used. Any +ve character values are acceptable.

ucSubIndex

Defines which index value signifies User Integer and User Float values of the servo drive. Refer to the section **16.4.4 Using Event Groups 16 and 17**. Any +ve character integers are accepted as values.

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC 0x01

PDO_COM_PARAM_ASYNC 0xFF

PDO_COM_PARAM_EVENT 0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

ucPDOType

The direction of the PDO, according to the MC_PDO_TYPE_ENUM enumerator:

MC_PDO_TYPE_RPDO

MC_PDO_TYPE_TXPDO

This will be dependent on the value of the parameters ucEventGroup, and ucSubIndex.

MMC_CONFIGUSERPARAMEVENTPDO4_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGUSERPARAMEVENTPDO4_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-30 describes the function block for MMC_CfgUserParamEvPDO4 as applied within the IEC 61131 programming. The IEC function block serves CfgUserParamEvPDO3, CfgUserParamEvPDO4, CfgRegParamEvPDO3, and CfgRegParamEvPDO4 together.

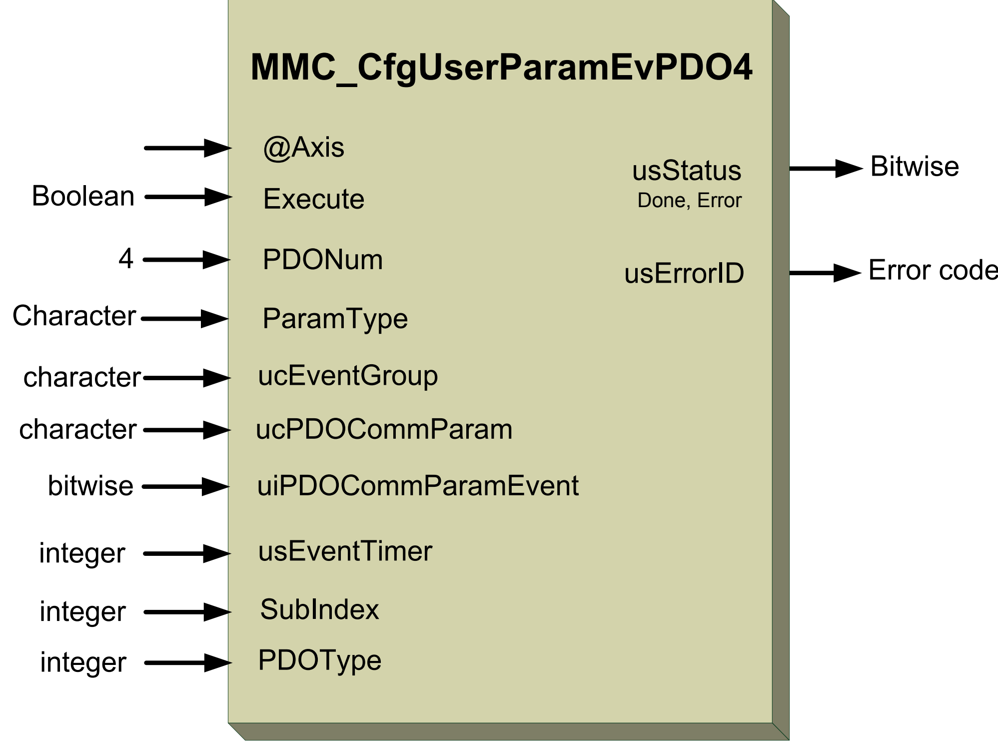


Figure 16-30: MMC_CfgUserParamEvPDO4 function block

16.5.7.2 Function Block Code Example

```
int rc;
MMC_CONFIGUSERPARAMEVENTPDO4_IN          stConfigUserParamEventPDO4_in;
MMC_CONFIGUSERPARAMEVENTPDO4_OUT         stConfigUserParamEventPDO4_out;
```

```
//
// Inserting the structure parameters:
stConfigUserParamEventPDO4_in.uiPDOCommParamEvent = 0xCC;               //inserts a PDO Events
mechanism in the GMAS for servo drive operations
stConfigUserParamEventPDO4_in.usEventTimer       = 1;                   //Event timer
stConfigUserParamEventPDO4_in.ucEventGroup       = NC_COMM_EVENT_GROUP7;    //Defines which
group of events are to be transferred from the G-MAS
stConfigUserParamEventPDO4_in.ucSubIndex         = 2,1;                 //Defines which index
value signifies the group of events to be transferred from the G-MAS
stConfigUserParamEventPDO4_in.ucPDOCommParam     = 0xFE;                //PDO communications
parameter
stConfigUserParamEventPDO4_in.ucPDOType          = MC_PDO_TYPE_TXPDO;   //The direction of the
PDO
//
rc = MMC_CfgUserParamEvPDO4Cmd (hConn, iAxisRef, &stConfigUserParamEventPDO4_in,
&stConfigUserParamEventPDO4_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.8 MMC_ChangeDefaultPDOConfiguration

Changes the default PDO communication parameter.

```
MMC_LIB_API int MMC_ChangeDefaultPDOConfiguration(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGPDOCOMMPARAM_IN* pInParam,
OUT MMC_CONFIGPDOCOMMPARAM_OUT *pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGPDOCOMMPARAM_IN** input data structure using the MMC_ChangeDefaultPDOConfiguration function.

pOutParam

Points to the **MMC_CONFIGPDOCOMMPARAM_OUT** output structure receiving information as a result of calling the MMC_ChangeDefaultPDOConfiguration function.

Remarks

Allows the communication parameter to be changed from Sync to Async and visa versa.

Scope

All

MMC_CONFIGPDOCOMMPARAM_IN Structure

typedef struct mmc_configpdocommparam_in{
unsigned char ucPDONum;
unsigned char ucPDODir;
unsigned char ucPDOCommParam;
}MMC_CONFIGPDOCOMMPARAM_IN;

Parameters

ucPDONum

Changes a specific PDO's communication from sync to async and visa versa. Allowed values are 3 and 4, representing the PDO3 and PDO4.

ucPDODir

Changes the RX or TX specific PDO communication. Allowed values are:

RXPDO=0

TXPDO=1

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC 0x01 (1)

PDO_COM_PARAM_ASYNC 0xFF (255)

PDO_COM_PARAM_EVENT 0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

MMC_CONFIGPDOCOMMPARAM_OUT Structure

typedef struct mmc_configpdocommparam_out{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGPDOCOMMPARAM_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-31 describes the function block for MMC_ChangeDefaultPDOConfiguration.

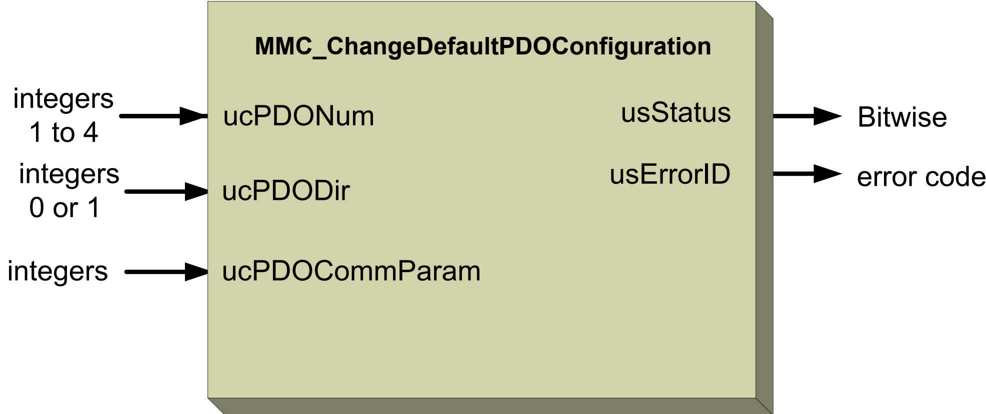


Figure 16-31: MMC_ChangeDefaultPDOConfiguration function block

16.5.9 MMC_ChngOpMode

Changes the motion mode between NC and Distributed. This is previous determined in the DS-402 mode.

```
MMC_LIB_API int MMC_ChngOpMode(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CHANGEMOTIONMODE_IN* pInParam,
OUT MMC_CHANGEMOTIONMODE_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed – Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CHANGEMOTIONMODE_IN** input data structure using the MMC_ChngOpMode function.

pOutParam

Points to the **MMC_CHANGEMOTIONMODE_OUT** output structure receiving information as a result of calling the MMC_ChngOpMode function.

Remarks

None

Scope

The motion mode must be changed in distributed state e.g. from position mode to velocity mode.

MMC_CHANGEMOTIONMODE_IN Structure

```
typedef struct{
unsigned char ucMotionMode;
}MMC_CHANGEMOTIONMODE_IN;
```

Parameters

ucMotionMode

Enumerator for the motion mode. The DS-402 motion modes and IDs available are:

Profile position mode = 1

Profiled velocity mode (partial) = 3

Homing Mode = 6

Interpolated position mode = 7

Any of the above +ve values accepted

MMC_CHANGEMOTIONMODE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
} MMC_CHANGEMOTIONMODE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-32 describes the function block for MMC_ChngOpMode as applied within the IEC 61131 programming.

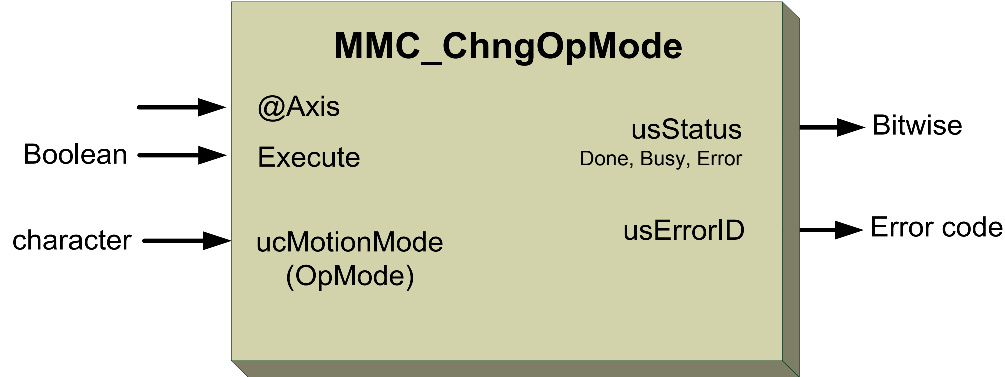


Figure 16-32: MMC_ChngOpMode function block

16.5.9.2 Function Block Code Example

```
int rc;
MMC_CHANGEMOTIONMODE_IN     stChangeOpMode_in;
MMC_CHANGEMOTIONMODE_OUT    stChangeOpMode_out;
//
// Inserting the structure parameters:
stChangeOpMode_in.ucMotionMode  = 1; //Enumerator for the motion mode
//
rc = MMC_ChngOpMode (hConn, iAxisRef, &stChangeOpMode_in, &stChangeOpMode_out) ;
if (rc != 0)
{
    HandleError() ;
}
```

16.5.10 MMC_ConfigEventModePDO3

Configures event mode for the PDO3 according group type.

```
MMC_LIB_API int MMC_CfgEventModePDO3Cmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGEVENTMODEPDO3_IN* pInParam,
OUT MMC_CONFIGEVENTMODEPDO3_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed – Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGEVENTMODEPDO3_IN** input data structure using the MMC_ConfigEventModePDO3 function.

pOutParam

Points to the **MMC_CONFIGEVENTMODEPDO3_OUT** output structure receiving information as a result of calling the MMC_ConfigEventModePDO3 function.

Remarks

None

Scope

All

MMC_CONFIGEVENTMODEPDO3_IN Structure

```
typedef struct{
unsigned char ucPDOEventMode;
}MMC_CONFIGEVENTMODEPDO3_IN;
```

Parameters

ucPDOEventMode

PDO3 event mode. This enumerator has the following values:

MC_PDO_EVENT_NO_NOTIF = 0,

MC_PDO_EVENT_CYCLIC_NOTIF,

MC_PDO_EVENT_IMMIDIATE_NOTIF

of which the default event type is MC_PDO_EVENT_NO_NOTIF. When using MC_PDO_EVENT_IMMIDIATE_NOTIF mode, no endian swap is created on the data.

MMC_CONFIGEVENTMODEPDO3_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGEVENTMODEPDO3_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs**, and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-33 describes the function block for MMC_ConfigEventModePDO3 as applied within the IEC 61131 programming.

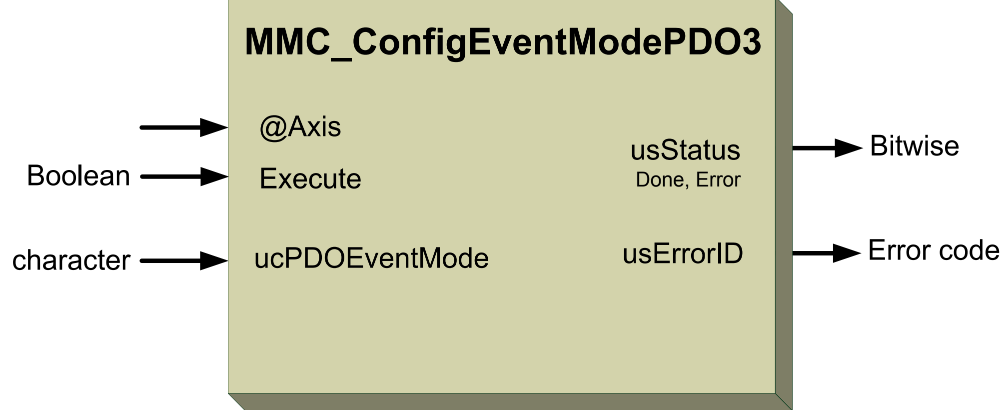


Figure 16-33: MMC_ConfigEventModePDO3 function block

16.5.10.2 Function Block Code Example

```
int rc;
MMC_CONFIGEVENTMODEPDO3_IN      stCfgEventModePDO3_in;
MMC_CONFIGEVENTMODEPDO3_OUT     stCfgEventModePDO3_out;
//
// Inserting the structure parameters:
stCfgEventModePDO3_in.ucPDOEventMode     = MC_PDO_EVENT_NO_NOTIF;        //PDO3 event mode
//
rc = MMC_CfgEventModePDO3Cmd (hConn, iAxisRef, &stCfgEventModePDO3_in,
&stCfgEventModePDO3_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.11 MMC_ConfigEventModePDO4

Configures event mode for the PDO4 according group type.

```
MMC_LIB_API int MMC_CfgEventModePDO4Cmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGEVENTMODEPDO4_IN* pInParam,
OUT MMC_CONFIGEVENTMODEPDO4_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed – Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGEVENTMODEPDO4_IN** input data structure using the MMC_ConfigEventModePDO4 function.

pOutParam

Points to the **MMC_CONFIGEVENTMODEPDO4_OUT** output structure receiving information as a result of calling the MMC_ConfigEventModePDO4 function.

Remarks

None

Scope

All

MMC_CONFIGEVENTMODEPDO4_IN Structure

```
typedef struct{
unsigned char ucPDOEventMode;
}MMC_CONFIGEVENTMODEPDO4_IN;
```

Parameters

ucPDOEventMode

PDO4 event mode. This enumerator has the following values:

MC_PDO_EVENT_NO_NOTIF = 0,

MC_PDO_EVENT_CYCLIC_NOTIF = 1

MC_PDO_EVENT_IMMIDIATE_NOTIF = 2

of which the default event type is MC_PDO_EVENT_NO_NOTIF. When using MC_PDO_EVENT_IMMIDIATE_NOTIF mode, no endian swap is created on the data.

MMC_CONFIGEVENTMODEPDO4_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGEVENTMODEPDO4_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections 4.3 **Maestro Error IDs**, and 4.9 **NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-34 describes the function block for MMC_ConfigEventModePDO4 as applied within the IEC 61131 programming.

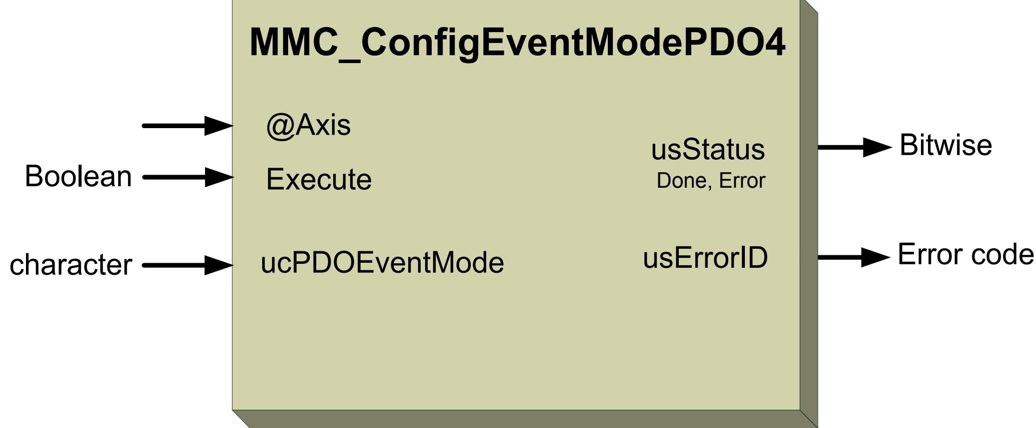


Figure 16-34: MMC_ConfigEventModePDO4 function block

16.5.11.2 Function Block Code Example

```
int rc;
MMC_CONFIGEVENTMODEPDO4_IN      stCfgEventModePDO4_in;
MMC_CONFIGEVENTMODEPDO4_OUT     stCfgEventModePDO4_out;
//
// Inserting the structure parameters:
stCfgEventModePDO4_in.ucPDOEventMode     = MC_PDO_EVENT_NO_NOTIF;        //PDO4 event mode
//
rc = MMC_CfgEventModePDO4Cmd (hConn, iAxisRef, &stCfgEventModePDO4_in,
&stCfgEventModePDO4_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.12 MMC_ConfigVirtualEncoder

This function defines a servo-drive as the virtual CAN encoder.

MMC_LIB_API int MMC_ConfigVirtualEncoderCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGVIRTUALENCODER_IN* pInParam,
OUT MMC_CONFIGVIRTUALENCODER_OUT* pOutParam
);

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_PLCopen_single_API.h
GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGVIRTUALENCODER_IN** input data structure using the MMC_ConfigVirtualEncoder function.

pOutParam

Points to the **MMC_CONFIGVIRTUALENCODER_OUT** output structure receiving information, as a result of calling the MMC_ConfigVirtualEncoder function.

Remarks

The virtual encoder is based on the DS406 CAN encoder protocol on the network. It allows the user to broadcast the position of a servo drive to the other servo drives connected to the Maestro. Any of the servo-drives may be defined as the encoder, as each of the following functions is axis related:

- MMC_SetPosition
- MMC_ConfigVirtualEncoder
- MMC_CancelVirtualEncoder

Scope

The servo drives must set as a group address. SDO's must be set to manually 'listen' on the group address for the position. The SYNC function should be used to set the Maestro. It should be noted that the group ID is limited to between 1 to 127.

MMC_CONFIGVIRTUALENCODER_IN Structure

```
typedef struct MMC_CONFIGVIRTUALENCODER_IN{
double dbLowPos;
double dbHighPos;
float fFactor;
unsigned char ucMode;
unsigned char ucGroupID;
}MMC_CONFIGVIRTUALENCODER_IN;
```

Parameters

dbLowPos

Low range of the virtual encoder. Any -ve or +ve double values in technical unit [u]

dbHighPos

High range of the virtual encoder. Any -ve or +ve double values in technical unit [u]

fFactor

Encoder factor. Any +ve integer value accepted.

ucMode

Defines the virtual encoder mode of the encoder with the following options:

NC_NODE_VIRTUAL_ENC_MODE_DISABLED = 0

NC_NODE_VIRTUAL_ENC_MODE_TARGET_POS

NC_NODE_VIRTUAL_ENC_MODE_ACTUAL_POS

ucGroupID

Group CAN ID. +ve integer accepted. The group ID is limited to between 1 to 127.

MMC_CONFIGVIRTUALENCODER_OUT Structure

```
typedef struct MMC_CONFIGVIRTUALENCODER_OUT{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGVIRTUALENCODER_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-35 describes the function block for MMC_ConfigVirtualEncoder as applied within the IEC 61131 programming.

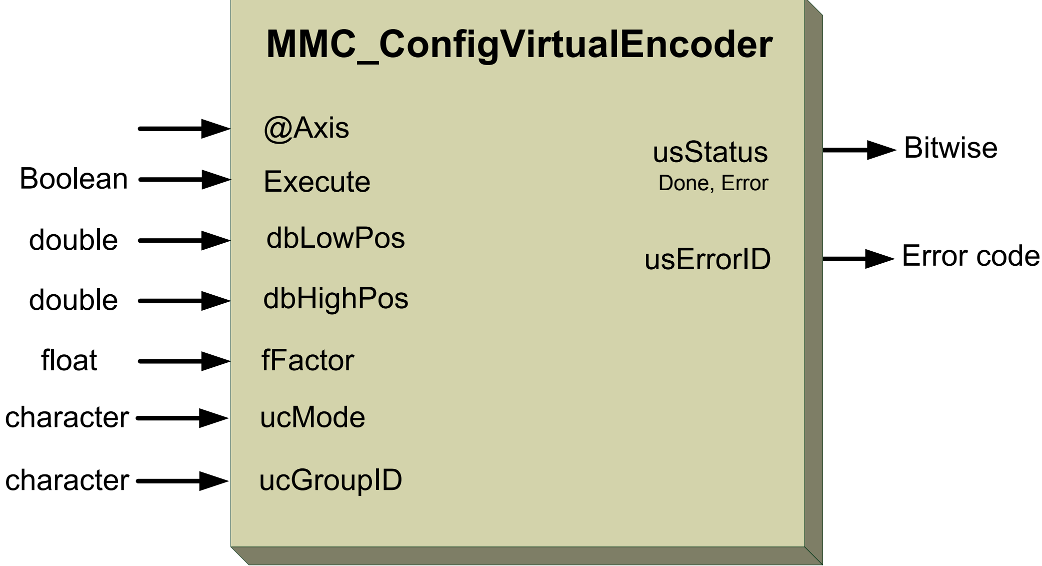


Figure 16-35: MMC_ConfigVirtualEncoder function block

16.5.12.2 Function Block Code Example

```
short CongVirtualEncoder(MMC_AXIS_REF_HNDL  aRef, unsigned char theGroupID)
{
MMC_CONFIGVIRTUALENCODER_IN pInParam;
MMC_CONFIGVIRTUALENCODER_OUT pOutParam;
pInParam.dbHighPos=0;
pInParam.dbLowPos=0;
pInParam.ucGroupID=theGroupID;
pInParam.ucMode=2;
int rc;
rc=MMC_ConfigVirtualEncoder(cHndl,aRef,&pInParam,&pOutParam)
if (rc != 0)
{
    HandleError();
}
 }
 return pOutParam.sErrorID;
}
```

16.5.13 MMC_GetAxisByCanId

Obtains axis handle according to the CANbus identity.

```
MMC_LIB_API int MMC_GetAxisByCanIdCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETAXISREFFROMCANID_IN* pInParam,
OUT MMC_GETAXISREFFROMCANID_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed – Supported
Source	GMAS\includes\MMC_drive_comm_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETAXISREFFROMCANID_IN** input data structure using the MMC_GetAxisByCanId function.

pOutParam

Points to the **MMC_GETAXISREFFROMCANID_OUT** output structure receiving information as a result of calling the MMC_GetAxisByCanId function.

Remarks

None

Scope

All

MMC_GETAXISREFFROMCANID_IN Structure

```
typedef struct{
unsigned char ucNodeID;
}MMC_GETAXISREFFROMCANID_IN;
```

Parameters

ucNodeID

Node ID from the CANbus. Any +ve integer accepted.

MMC_GETAXISREFFROMCANID_OUT Structure

```
typedef struct{
unsigned short usAxisRef;
unsigned short usStatus;
short sErrorID;
}MMC_GETAXISREFFROMCANID_OUT;
```

Parameters

usAxisRef

Axis reference. Any +ve bitwise integer.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-36 describes the function block for MMC_GetAxisByCanId

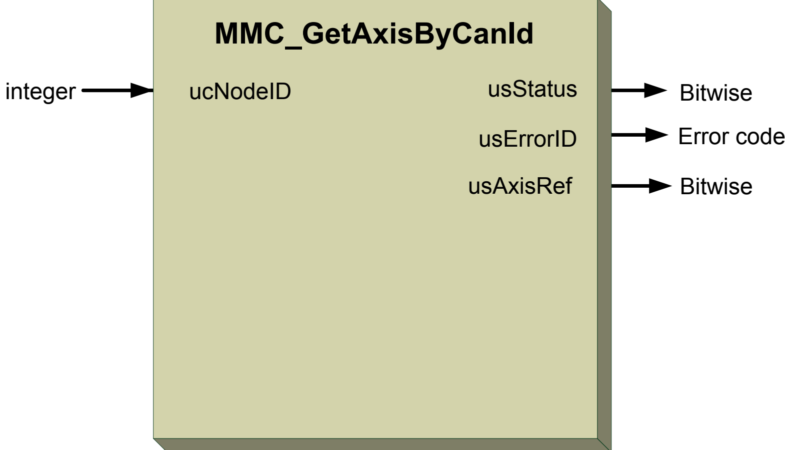


Figure 16-36: MMC_GetAxisByCanId function block

16.5.13.2 Function Block Code Example

```
int rc;
MMC_GETAXISREFFROMCANID_IN      stGetAxisRefFromCANID_in;
MMC_GETAXISREFFROMCANID_OUT     stGetAxisRefFromCANID_out;
//
// Inserting the structure parameters:
stGetAxisRefFromCANID_in.ucNodeID   = 4; //Node ID from the CANbus
//
rc = MMC_GetAxisByCanIdCmd (hConn, &stGetAxisRefFromCANID_in, &stGetAxisRefFromCANID_out);
printf("Axis By CAN ID Status[%ld] ErrId[%d]\n", (long
int)stGetAxisRefFromCANID_out.usAxisRef, (short)stGetAxisRefFromCANID_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.5.14 MMC_GetPDOInfo

Obtains the PDO information of PDO 3 and 4.

```
MMC_LIB_API int MMC_GetPDOInfoCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETPDOINFO_IN* pInParam,
OUT MMC_GETPDOINFO_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed – Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETPDOINFO_IN** input data structure using the MMC_GetPDOInfo function.

pOutParam

Points to the**MMC_GETPDOINFO_OUT** output structure receiving information as a result of calling the MMC_GetPDOInfo function.

Remarks

None

Scope

All

MMC_GETPDOINFO_IN Structure

```
typedef struct{
unsigned char ucPDONumber;
}MMC_GETPDOINFO_IN;
```

Parameters

ucPDONumber

The PDO index number. Changes a specific PDO's communication from sync to async and visa versa. Allowed values are 3 and 4, representing the PDO3 and PDO4.

MMC_GETPDOINFO_OUT Structure

```
typedef struct mmc_getpdoinfo_out {
int iPDOEventMode;
unsigned int uiCommParamEventPDO;
unsigned short usStatus;
short sErrorID;
unsigned short usEventTimerPDO;
unsigned char ucRPDOCommType;
unsigned char ucTPDOCommType;
unsigned char ucTPDOCommEventGroup;
unsigned char ucRPDOCommEventGroup;
unsigned char ucSubIndexRPDO;
unsigned char ucSubIndexTPDO;
}MMC_GETPDOINFO_OUT;
```

Parameters

iPDOEventMode

The PDO event mode integer. This is the notification of an event sent or not sent. This enumerator has the following values:

MC_PDO_EVENT_NO_NOTIF = 0,

MC_PDO_EVENT_CYCLIC_NOTIF = 1 (sent after change in event)

MC_PDO_EVENT_IMMIDIATE_NOTIF = 2 (sent immediately after event)

of which the default event type is MC_PDO_EVENT_NO_NOTIF. When using MC_PDO_EVENT_IMMIDIATE_NOTIF mode, no endian swap is created on the data.

uiCommParamEventPDO

Indicates a type of event dependant on the drive

usEventTimerPDO

Defined by the enumerator MC_PDO_TIMER_EVENT_ENUM

MC_PDO_TIMER_NON = 0,

MC_PDO_TIMER_1_MILISEC,

MC_PDO_TIMER_2_MILISEC,

MC_PDO_TIMER_3_MILISEC,

MC_PDO_TIMER_4_MILISEC,

MC_PDO_TIMER_5_MILISEC,

MC_PDO_TIMER_10_MILISEC = 10,

MC_PDO_TIMER_20_MILISEC = 20,

MC_PDO_TIMER_25_MILISEC = 25,

MC_PDO_TIMER_50_MILISEC = 50,

MC_PDO_TIMER_100_MILISEC = 100,

MC_PDO_TIMER_150_MILISEC = 150,

MC_PDO_TIMER_200_MILISEC = 200,

MC_PDO_TIMER_250_MILISEC = 250,

MC_PDO_TIMER_255_MILISEC = 255

………

Continuing to MC_PDO_TIMER_65535_MILISEC = 65535

ucRPDOCommType

RPDO communication type as a +ve character value. Defined by:

PDO_COM_PARAM_SYNC	1(0x01)
PDO_COM_PARAM_ASYNC	0(0xFF)
PDO_COM_PARAM_EVENT	0xFE

ucTPDOCommType

TPDO communication type as a +ve character value. Defined by:

PDO_COM_PARAM_SYNC	0x01
PDO_COM_PARAM_ASYNC	0xFF
PDO_COM_PARAM_EVENT	0xFE

ucTPDOCommEventGroup

TPDO communication event group as a +ve character value. Refer to the User Event Groups in section **16.4.3 PDO Mapping**.

ucRPDOCommEventGroup

RPDO communication event group as a +ve character value. Refer to the User Event Groups in section **16.4.3 PDO Mapping**.

ucSubIndexRPDO

Defines which RPDO index value signifies User Integer and User Float values of the servo drive. Refer to the section **16.4.4 Using Event Groups 16 and 17**. Any +ve character integers are accepted as values.

ucSubIndexTPDO

Defines which TPDO index value signifies User Integer and User Float values of the servo drive. Refer to the section **16.4.4 Using Event Groups 16 and 17**. Any +ve character integers are accepted as values.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-37 describes the function block for MMC_GetPDOInfo as applied within the IEC 61131 programming.

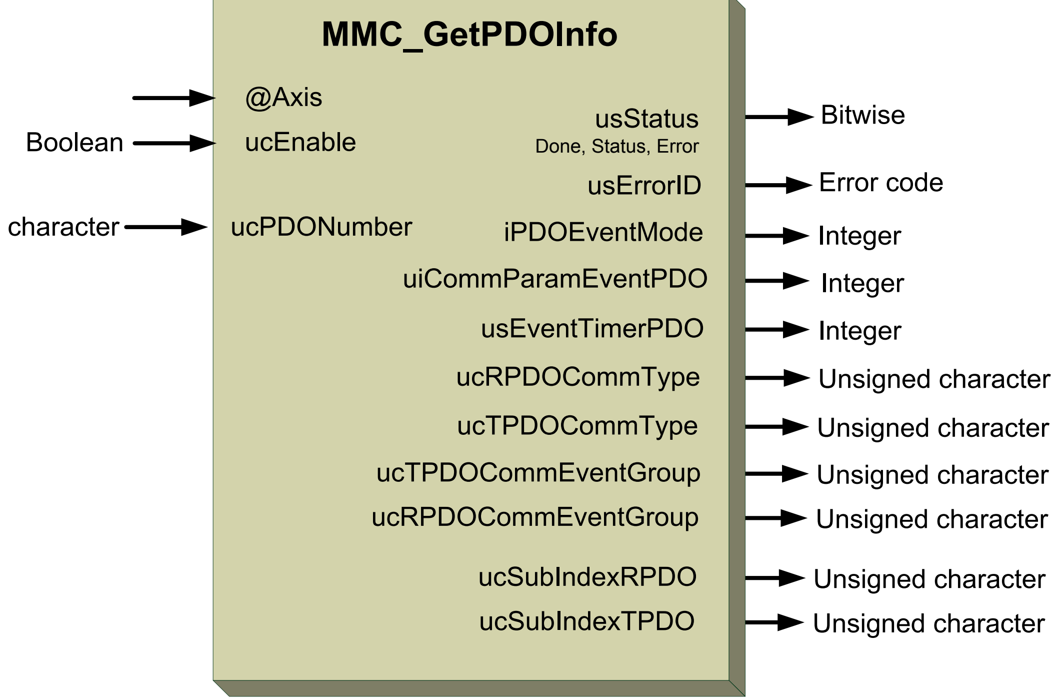


Figure 16-37: MMC_GetPDOInfo function block

16.5.15 MMC_GetSyncTime

Where CANbus communication is relevant, returns the SYNC time.

```
MMC_LIB_API int MMC_GetSyncTimeCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETSYNCTIME_IN* pInParam,
OUT MMC_GETSYNCTIME_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETSYNCTIME_IN** input data structure using the MMC_GetSyncTime function.

pOutParam

Points to the **MMC_GETSYNCTIME_OUT** output structure receiving information as a result of calling the MMC_GetSyncTime function.

Remarks

None

Scope

All

MMC_GETSYNCTIME_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_GETSYNCTIME_IN;
```

Parameters

dummy

Dummy Sync time input. Any +ve values accepted.

MMC_GETSYNCTIME_OUT Structure

```
typedef struct{
unsigned short usSYNCTime;
unsigned short usStatus;
short sErrorID;
}MMC_GETSYNCTIME_OUT;
```

Parameters

usSYNCTime

Sync time output. Refer to the explanation in section 16.4.6 **SYNC and Time Stamp on page 1243**. Any +ve integer value between 0 to 1000.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-38 describes the function block for MMC_GetSyncTime as applied within the IEC 61131 programming.

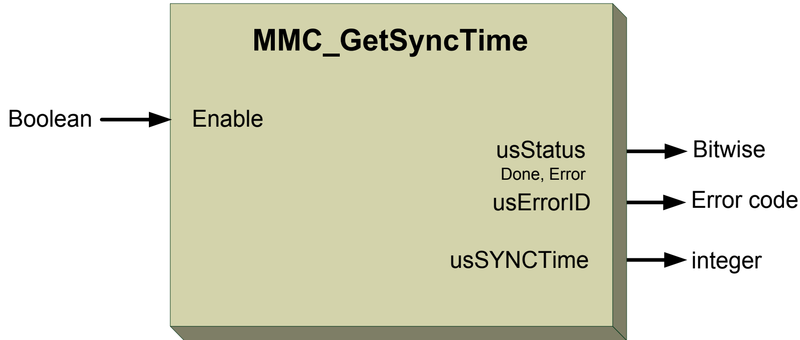


Figure 16-38: MMC_GetSyncTime function block

16.5.15.2 Function Block Code Example

```
int rc;
MMC_GETSYNCTIME_IN          stGetSYNCTime_in;
MMC_GETSYNCTIME_OUT         stGetSYNCTime_out;
//
// Inserting the structure parameters:
stGetSYNCTime_in.dummy      = 1; //Any dummy inputs
//
rc = MMC_GetSyncTimeCmd (hConn, &stGetSYNCTime_in, &stGetSYNCTime_out);
printf("Sync Time Status[%ld] ErrId[%d]\n", (long int)stGetSYNCTime_out.usSYNCTime,
(short)stGetSYNCTime_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.5.16 MMC_PDOGeneralRead

Reads a specific PDO message command.

```
MMC_LIB_API int MMC_PDOGeneralReadCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GENERALPARAMPDOREAD_IN* pInParam,
OUT MMC_GENERALPARAMPDOREAD_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GENERALPARAMPDOREAD_IN** input data structure using the MMC_PDOGeneralRead function.

pOutParam

Points to the **MMC_GENERALPARAMPDOREAD_OUT** output structure receiving information as a result of calling the MMC_PDOGeneralRead function.

Remarks

Reads a specific package of data sent

Scope

To gain the correct feedback data, make sure to set the DS-401 PD03 and/or PDO4 at the drives and Maestro as per requirements. Refer to the sections **16.12.1** to **16.12.8**

MMC_GENERALPARAMPDOREAD_IN Structure

```
typedef struct{
unsigned char ucParam;
}MMC_GENERALPARAMPDOREAD_IN;
```

Parameters

ucParam

Defines which of the general group of 32bit assigned events 16 or 17 is to be transferred to the Maestro. Refer to the section **16.4.3 PDO Mapping on page 1239**. Use the values 0, 1 denoted below to represent the assignment:

NC_COMM_EVENT_GROUP16 0

NC_COMM_EVENT_GROUP17 1

MMC_GENERALPARAMPDOREAD_OUT Structure

```
typedef struct{
#ifdef WIN32
unsigned __int64 ulliVal;
#else
unsigned long long int ulliVal;
#endif
unsigned short usStatus;
short sErrorID;
}MMC_GENERALPARAMPDOREAD_OUT;
```

Parameters

__int64 ulliVal or ulliVal

If function is defined for WIN32 then use *__int64 ulliVal*, else use *ulliVal*. Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-39 describes the function block for MMC_PDOGeneralRead as applied within the IEC 61131 programming.

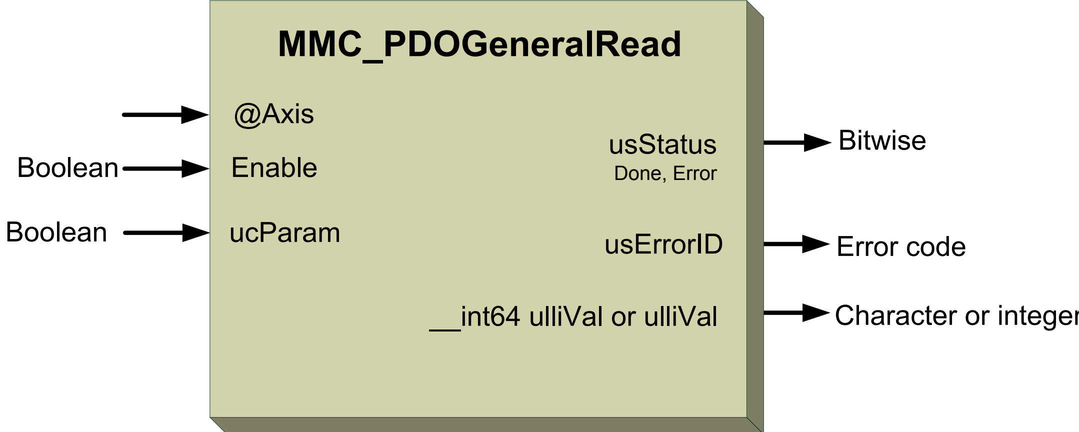


Figure 16-39: MMC_PDOGeneralRead function block

16.5.17 MMC_PDOGeneralWrite

Writes a specific PDO message command.

```
MMC_LIB_API int MMC_PDOGeneralWriteCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GENERALPARAMPDOWRITE_IN* pInParam,
OUT MMC_GENERALPARAMPDOWRITE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GENERALPARAMPDOWRITE_IN** input data structure using the MMC_PDOGeneralWrite function.

pOutParam

Points to the **MMC_GENERALPARAMPDOWRITE_OUT** output structure receiving information as a result of calling the MMC_PDOGeneralWrite function.

Remarks

Writes a specific data package to be sent.

Scope

To gain the correct feedback data, make sure to set the DS-401 PD03 and/or PDO4 at the drives and Maestro as per requirements. Refer to the sections **16.12.1** to **16.12.8.**

MMC_GENERALPARAMPDOWRITE_IN Structure

```
typedef struct{
#ifdef WIN32
unsigned __int64 ulliVal;
#else
unsigned long long int ulliVal;
#endif
unsigned char ucParam;
}MMC_GENERALPARAMPDOWRITE_IN;
```

Parameters

__int64 ulliVal or ulliVal

If function is defined for WIN32 then use *__int64 ulliVal*, else use *ulliVal*. Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

ucParam

Defines which of the general group of 32bit assigned events 16 or 17 is to be transferred to the Maestro. Refer to the section **16.4.3 PDO Mapping on page 1239**. Use the values 0, 1 denoted below to represent the assignment:

NC_COMM_EVENT_GROUP16 0

NC_COMM_EVENT_GROUP17 1

MMC_GENERALPARAMPDOWRITE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_GENERALPARAMPDOWRITE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-40 describes the function block for MMC_PDOGeneralWrite as applied within the IEC 61131 programming.

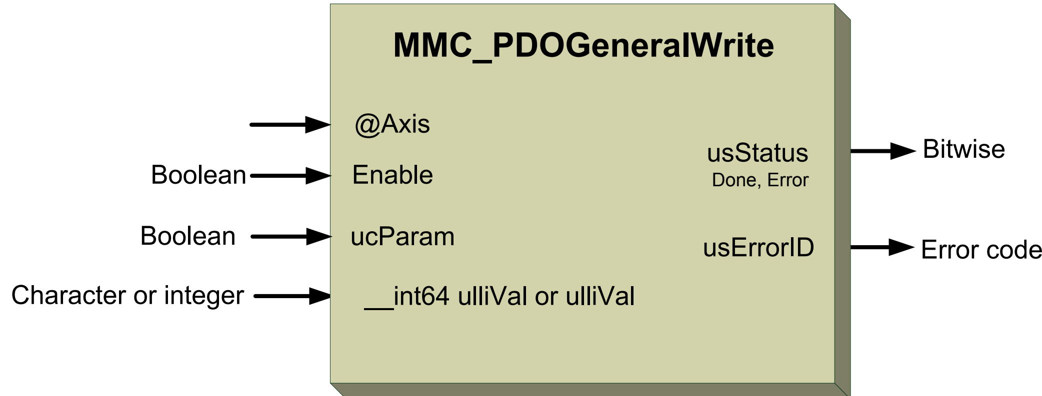


Figure 16-40: MMC_PDOGeneralWrite function block

16.5.18 MMC_ReceiveCANRawData

Receives prepared CANopen RAW data (DS-301 or DS-402).

```
MMC_LIB_API int MMC_ReceiveCANRawData(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN unsigned short iTimeOutms,
OUT MMC_CAN_REPLY_DATA_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **iTimeOutms** input data structure using the MMC_ReceiveCANRawData function.

pOutParam

Points to the **MMC_CAN_REPLY_DATA_OUT** output structure receiving information as a result of calling the MMC_ReceiveCANRawData function.

Remarks

None

Scope

All

iTimeOutms Input

[IN] Time out delay to receive data. Integer time value in ms.

MMC_CAN_REPLY_DATA_OUT Structure

```
typedef struct{
unsigned short usFunctionID;
unsigned short usNumerator;
unsigned short usDatasize;
unsigned short usPadding;
unsigned short usStatus;
short sErrorID;
unsigned short usCOB_ID;
unsigned short usAxisRef;
unsigned char can_data_length;
unsigned char data[8];
}MMC_CAN_REPLY_DATA_OUT;
```

Parameters

usFunctionID

Request function ID of the raw data. Any +ve bitwise integer

usNumerator

Response to the send data. Any +ve bitwise integer

usDatasize

Size of the data sent. Any +ve bitwise integer

usPadding

Alignment padding of data. This parameter is not in use.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

usCOB_ID

11-bit ID of the CAN-frame. Any +ve bitwise number

usAxisRef

Axis reference. Any +ve bitwise integer.

can_data_length

Length of the CAN data. Any character integer value.

data[8]

Data. Character value with a maximum length of 8 bits.

Figure 16-41 describes the function block for MMC_ReceiveCANRawData

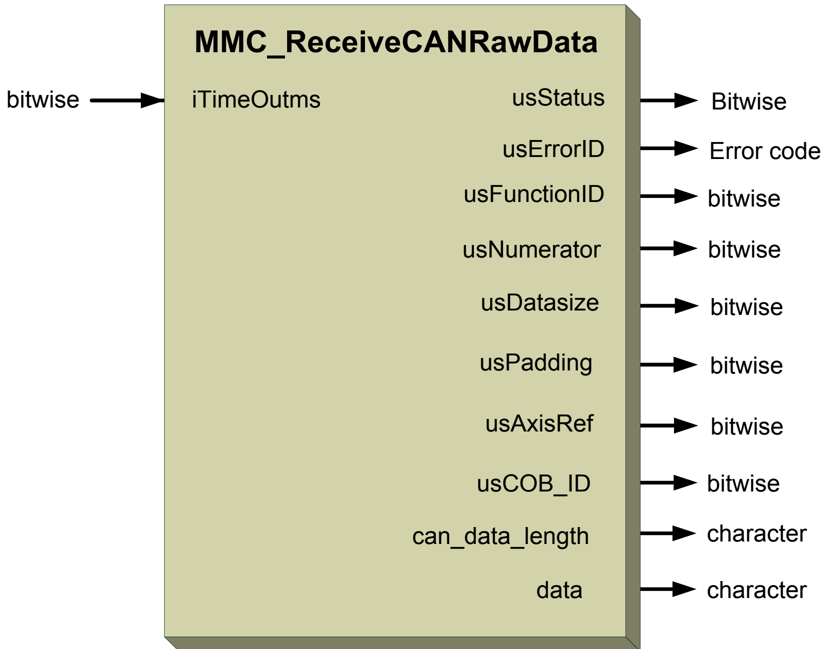


Figure 16-41: MMC_ReceiveCANRawData function block

16.5.18.2 Function Block Code Example

```
int rc;
MMC_CAN_REPLY_DATA_OUT     stCANReplyData_out;
//
// Inserting the structure parameters:
iTimeOutms = 1001;     //Time delay
//
rc = MMC_ReceiveCANRawData (hConn, iAxisRef, iTimeOutms, &stCANReplyData_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.19 MMC_SendCANRawData

Sends prepared CANopen RAW data (DS-301 or DS-402).

```
MMC_LIB_API int MMC_SendCANRawData(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SENDRAWDATA_IN* pInParam,
OUT MMC_SENDRAWDATA_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SENDRAWDATA_IN** input data structure using the MMC_SendCANRawData function.

pOutParam

Points to the **MMC_SENDRAWDATA_OUT** output structure receiving information as a result of calling the MMC_SendCANRawData function.

Remarks

None

Scope

All

MMC_SENDRAWDATA_IN Structure

```
typedef struct{
unsigned short usCOB_ID;
unsigned char ucLength;
unsigned char pData[8];
}MMC_SENDRAWDATA_IN;
```

Parameters

usCOB_ID

11-bit ID of the CAN-frame. Any +ve bitwise number

ucLength

Length of the raw data. Any +ve character value

pData[8]

Array raw data input dependant on the array size with a maximum array length of 8 bytes. Character value with a maximum length of 8 bits.

MMC_SENDRAWDATA_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SENDRAWDATA_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-42 describes the function block for MMC_SendCANRawData as applied within the IEC 61131 programming.

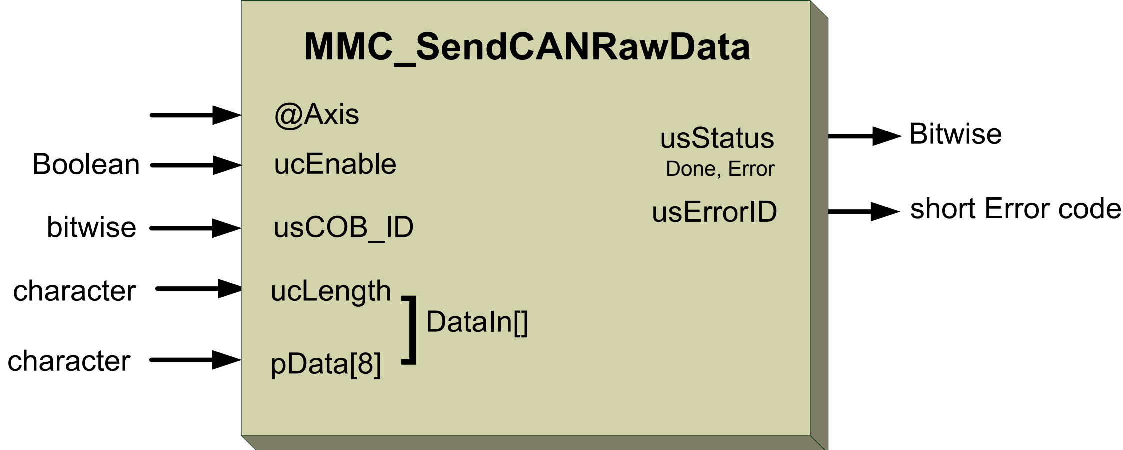


Figure 16-42: MMC_SendCANRawData function block

16.5.19.2 Function Block Code Example

```
int rc;
MMC_SENDRAWDATA_IN      stSendRawData_in;
MMC_SENDRAWDATA_OUT     stSendRawData_out;
//
// Inserting the structure parameters:
stSendRawData_in.usCOB_ID  = 111;        //11-bit ID of the CAN-frame
stSendRawData_in.ucLength  = 8;          //Length of the raw data
stSendRawData_in.pData[0]  = 101;        //8 byte raw data
stSendRawData_in.pData[1]  = 100;        //8 byte raw data
stSendRawData_in.pData[2]  = 110;        //8 byte raw data
//
rc = MMC_SendCANRawData (hConn, iAxisRef, &stSendRawData_in, &stSendRawData_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.20 MMC_SendandReceiveCANRawData

Sends and receives prepared CANopen RAW data (DS-301 or DS-402).

```
MMC_LIB_API int MMC_SendandReceiveCANRawData(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SENDRAWDATA_IN *pInParam,
OUT MMC_CAN_REPLY_DATA_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SENDRAWDATA_IN** input data structure using the MMC_SendandReceiveCANRawData function.

pOutParam

Points to the **MMC_CAN_REPLY_DATA_OUT** output structure receiving information as a result of calling the MMC_SendandReceiveCANRawData function.

Remarks

None

Scope

All

MMC_SENDRAWDATA_IN Structure

```
typedef struct{
unsigned short usCOB_ID;
unsigned char ucLength;
unsigned char pData[8];
}MMC_SENDRAWDATA_IN;
```

Parameters

usCOB_ID

11-bit ID of the CAN-frame. Any +ve bitwise number

ucLength

Length of the raw data. Any +ve character value

pData[8]

Array raw data input dependant on the array size with a maximum array length of 8 bytes. Character value with a maximum length of 8 bits.

MMC_CAN_REPLY_DATA_OUT Structure

```
typedef struct{
unsigned short usFunctionID;
unsigned short usNumerator;
unsigned short usDatasize;
unsigned short usPadding;
unsigned short usStatus;
short sErrorID;
unsigned short usCOB_ID;
unsigned short usAxisRef;
unsigned char can_data_length;
unsigned char data[8];
unsigned char ucAsyncEventType;
}MMC_CAN_REPLY_DATA_OUT;
```

Parameters

usFunctionID

Request function ID of the raw data. Any +ve bitwise integer

usNumerator

Response to the send data. Any +ve bitwise integer

usDatasize

Size of the data sent. Any +ve bitwise integer

usPadding

Alignment padding of data. This parameter is not in use.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

usCOB_ID

11-bit ID of the CAN-frame. Any +ve bitwise number

usAxisRef

Axis reference. Any +ve bitwise integer.

can_data_length

Length of the CAN data. Any character integer value.

data[8]

Data. Character value with a maximum length of 8 bits.

ucAsyncEventType

This event is received if an error (or timeout) occurred when calling an SDO download or Drive Command via the binary interpreter mechanism. Refer to section **12.2 Communication ASYNC Replies (Events) From Drives** for details.

Figure 16-42 describes the function block for MMC_Sendand ReceiveCANRawData as applied within the IEC 61131 programming.

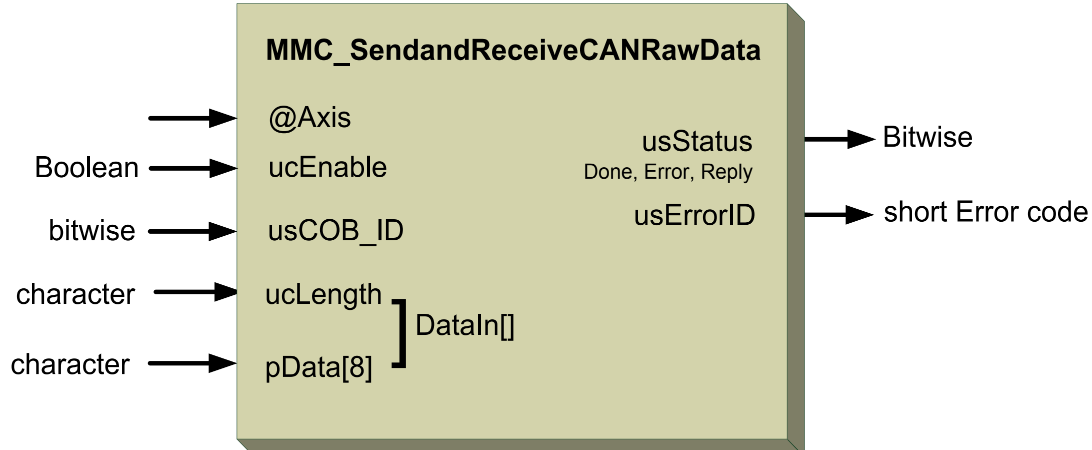


Figure 16-43: MMC_SendCANRawData function block

16.5.21 MMC_SendCmd

Not in operation

Sends a command string to the drive.

```
MMC_LIB_API int MMC_SendCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SENDCMD_IN* pInParam,
OUT MMC_SENDCMD_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SENDCMD_IN** input data structure using the MMC_SendCmd function.

pOutParam

Points to the **MMC_SENDCMD_OUT** output structure receiving information as a result of calling the MMC_SendCmd function.

Remarks

None

Scope

All

MMC_SENDCMD_IN Structure

```
typedef struct{
char pCmd[80];
}MMC_SENDCMD_IN;
```

Parameters

pCmd[80]

Command value. Any +ve or –ve array values with a maximum of 80 characters

MMC_SENDCMD_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SENDCMD_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-44 describes the function block for MMC_SendCmd

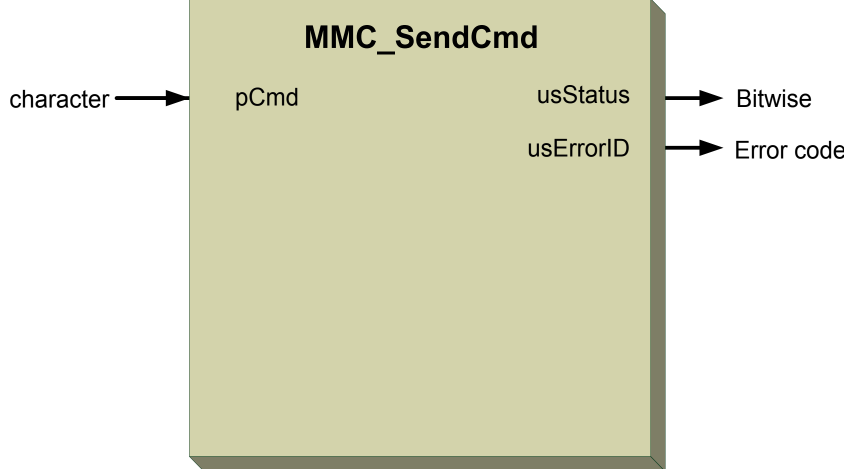


Figure 16-44: MMC_SendCmd function block

16.5.21.2 Function Block Code Example

```
int rc;
MMC_SENDCMD_IN     stSendCmd_in;
MMC_SENDCMD_OUT    stSendCmd_out;
//
// Inserting the structure parameters:
strcpy(stSendCmd_in.pCmd, "11111");          //Command value
//
rc = MMC_SendCmd (hConn, iAxisRef, &stSendCmd_in, &stSendCmd_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.22 MMC_SetHeartBeatConsumer

Sets the consumer heartbeat as an event to the user.

```
MMC_LIB_API int MMC_SetHeartBeatConsumerCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SETHEARTBEATCONSUMER_IN* pInParam,
OUT MMC_SETHEARTBEATCONSUMER_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoGlobal	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SETHEARTBEATCONSUMER_IN** input data structure using the MMC_SetHeartBeatConsumer function.

pOutParam

Points to the **MMC_SETHEARTBEATCONSUMER_OUT** output structure receiving information as a result of calling the MMC_SetHeartBeatConsumer function.

Remarks

The heartbeat process allows the Maestro (CAN slave) to monitor the servo drives and to determine if they are active, according to their periodic heartbeat messages. The Maestro monitors the heartbeat meassge from the servo drives, which may wish to monitor if this slave is active. The consumer heartbeat time defines the expected heartbeat cycle time. Monitoring of the heartbeat starts after the reception of the first heartbeat. If the consumer heartbeat time is 0, the corresponding entry is not used.

If the servo drive does not send a heartbeat message to the Maestro within the defined period of time (which this API is capable of changing), then a Heartbeat Error Event is sent to the Host Server after a period greater than the Heartbeat message period between the Maestro and the servo drive.

Scope

Heartbeat mechanism for the CANbus is automatically started when the Maestro is powered ON, and is set according to the Resource file basic cycle time. A Heartbeat Event is automatically sent to the host system when the heartbeat is delayed by a set time preconfigured in the Maestro.

The Heartbeat mechanism is set per axis in the Resource file.

MMC_SETHEARTBEATCONSUMER_IN Structure

```
typedef struct{
unsigned int uiHeartbeatTimeFactor;
}MMC_SETHEARTBEATCONSUMER_IN;
```

Parameters

uiHeartbeatTimeFactor

Heart beat time factor is a multiple of 1 ms. The calculation of the basic cycle time (predetermined in the Resource file), multiplied by this heartbeat time factor, and 1 ms, will set the Heartbeat time.

Values accepted are:

Min heartbeat factor = 0

>0, dependant on the cycle time:

$$\text{Max Heartbeat} = \frac{65535}{\text{G--MAS cycle time [ms]}}$$

MMC_SETHEARTBEATCONSUMER_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETHEARTBEATCONSUMER_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-45 describes the function block for MMC_SetHeartBeatConsumer as applied within the IEC 61131 programming.

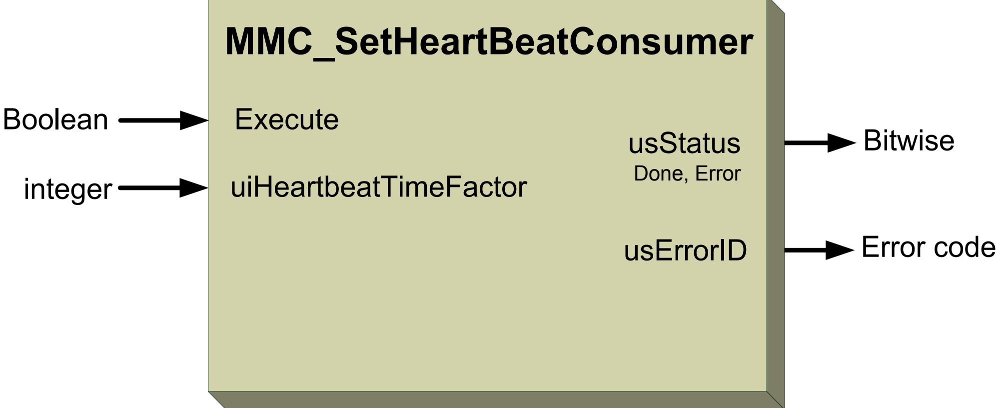


Figure 16-45: MMC_SetHeartBeatConsumer function block

16.5.22.2 Function Block Code Example

```
int rc;
MMC_SETHEARTBEATCONSUMER_IN     stSetHrtBtConsumer_in;
MMC_SETHEARTBEATCONSUMER_OUT    stSetHrtBtConsumer_out;
//
// Inserting the structure parameters:
stSetHrtBtConsumer_in.uiHeartbeatTimeFactor  = 216;   //Heart beat factor per cycle time
//
rc = MMC_SetHeartBeatConsumer (hConn, iAxisRef, &stSetHrtBtConsumer_in,
&stSetHrtBtConsumer_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.23 MMC_SetSyncTime

Where CANbus communication is relevant, sets the Sync time in the communication module, and updates the relevant nodes whose motion mode has an IP address. It also updates the kernel with the Sync time.

```
MMC_LIB_API int MMC_SetSyncTimeCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_SETSYNCTIME_IN* pInParam,
OUT MMC_SETSYNCTIME_OUT* pOutParam
);
```

Motion Mode NC – Not Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_SETSYNCTIME_IN** input data structure using the MMC_SetSyncTime function.

pOutParam

Points to the **MMC_SETSYNCTIME_OUT** output structure receiving information as a result of calling the MMC_SetSyncTime function.

Remarks

None

Scope

For use only with distributed systems

MMC_SETSYNCTIME_IN Structure

```
typedef struct{
unsigned short usSYNCTime;
}MMC_SETSYNCTIME_IN;
```

Parameters

usSYNCTime

Sync time input. Refer to the explanation in section 16.4.6 **SYNC and Time Stamp on page 1243**. Values are the following:

Min Sync Factor = 1

If the cycle time **>** 25.5[ms], then the Max Sync Factor = $\frac{255}{\text{cycle time}[\text{ms}]}$

Otherwise the cycle time **=<** 25.5, then

Max Sync Factor = $\frac{255}{10 * \text{cycle time}[\text{ms}]}$

MMC_SETSYNCTIME_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_SETSYNCTIME_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-46 describes the function block for MMC_SetSyncTime as applied within the IEC 61131 programming.

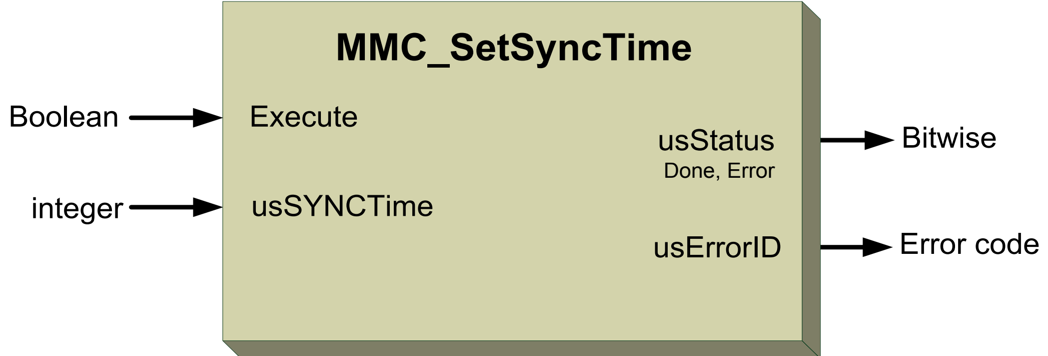


Figure 16-46: MMC_SetSyncTime function block

16.5.23.2 Function Block Code Example

```
int rc;
MMC_SETSYNCTIME_IN          stSetSyncTime_in;
MMC_SETSYNCTIME_OUT         stSetSyncTime_out;
//
// Inserting the structure parameters:
stSetSyncTime_in.usSYNCTime     = 65534; //Sync time input
//
rc = MMC_SetSyncTimeCmd (hConn, &stSetSyncTime_in, &stSetSyncTime_out);
if (rc != 0)
{
    HandleError();
}
```

16.5.24 MMC_StartBulkUpload

The Maestro manages the bulk upload process upon request from a host, i.e. the host sends this function command to the Maestro, and the Maestro uploads the recording buffer.

int MMC_StartBulkUploadCmd(IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_STARTBULKUPLOAD_IN* pInParam,
OUT MMC_STARTBULKUPLOAD_OUT* pOutParam
);

Motion Mode NC –Not Supported Distributed – Supported

Source GMAS\includes\MMC_drive_comm_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_STARTBULKUPLOAD_IN** input data structure using the MMC_StartBulkUpload function.

pOutParam

Points to the **MMC_STARTBULKUPLOAD_OUT** output structure receiving information as a result of calling the MMC_StartBulkUpload function.

Remarks

None

Scope

For use only with distributed systems.

The parameters usIndex, ucSubIndex, describe a CAN object that can be uploaded. However, not all CAN objects can be uploaded with bulk upload.

MMC_STARTBULKUPLOAD_IN Structure

```
typedef struct mmc_startbulkupload_in{
unsigned short usIndex;
unsigned char ucSubIndex;
}MMC_STARTBULKUPLOAD_IN;
```

Parameters

usIndex

COB index. Any +ve integer values (2 bytes).

ucSubIndex

Defines which index value signifies the group of events to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping on page 1239**. From events group 7 and above, the values represent the RPDO output. This parameter should mirror the enumerator value of the ucEventGroup variable, where applicable.

Any +ve character values.

MMC_STARTBULKUPLOAD_OUT Structure

```
typedef struct mmc_startbulkupload_out{
unsigned short usStatus;
short sErrorID;
} MMC_STARTBULKUPLOAD_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-47 describes the function block for MMC_StartBulkUpload.

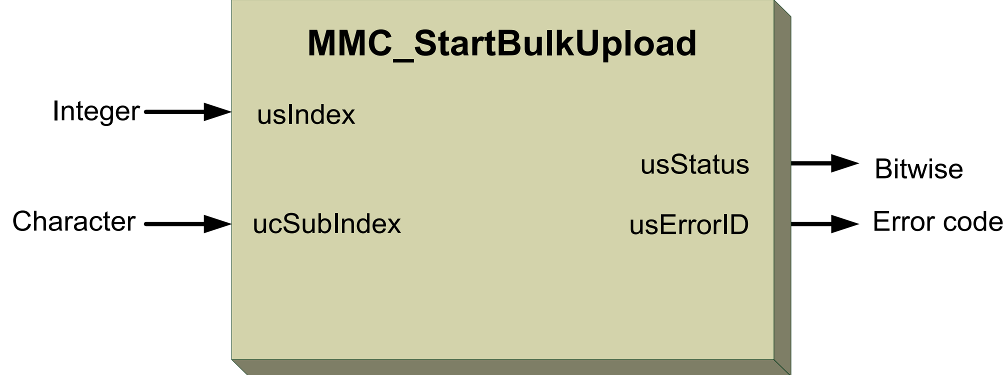


Figure 16-47: MMC_StartBulkUpload function block

16.5.25 MMC_GetBulkUploadStatus

During the whole process of a bulk upload, the status is retrieved using this function command.

```
int MMC_GetBulkUploadStatusCmd(IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETBULKUPLOADSTATUS_IN* pInParam,
OUT MMC_GETBULKUPLOADSTATUS_OUT* pOutParam
);
```

Motion Mode NC –Supported Distributed – Supported

Source GMAS\includes\MMC_drive_comm_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETBULKUPLOADSTATUS_IN** input data structure using the MMC_GetBulkUploadStatus function.

pOutParam

Points to the **MMC_GETBULKUPLOADSTATUS_OUT** output structure receiving information as a result of calling the MMC_GetBulkUploadStatus function.

Remarks

This function returns the user with the following data:

- Amount of data received
- Upload state (init/in progress/error/etc.)
- Communication error (if any)
- Process error (if any)

Scope

For use only with distributed systems

MMC_GETBULKUPLOADSTATUS_IN Structure

```
typedef struct mmc_getbulkuploadstatus_in{
unsigned char ucDummy;
}MMC_GETBULKUPLOADSTATUS_IN;
```

Parameters

ucDummy

Dummy value. Any -ve or +ve character.

MMC_GETBULKUPLOADSTATUS_OUT Structure

```
typedef struct mmc_getbulkuploadstatus_out{
unsigned int uiSizeCompleted;
unsigned long ulSizedReported
unsigned short usStatus;
short sErrorID;
short usCommError;
short usUploadError;
unsigned char ucUploadState;
} MMC_GETBULKUPLOADSTATUS_OUT;
```

Parameters

uiSizeCompleted

Number of bytes already received.

ulSizedReported

When the upload starts, the drive reports how many bytes it is transmitting.

usCommError

Communication error. If 0 then upload status is OK. If <0, then the parameter ucUploadState will also produce an error. Refer to the errors listed in sections 4.3 **Maestro Error IDs** and **4.5 Continued Maestro Error IDs.**

usUploadError

DS301 error. If 0 then upload status is OK. If <0, then refer to the errors listed in sections 4.3 **Maestro Error IDs** and **4.5 Continued Maestro Error IDs.**

ucUploadState

Upload phase. This may be one of the following:

- Not started
- Pre-initiated
- Initiated
- In progress
- Completed OK
- Completed with failure

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections 4.3 **Maestro Error IDs**, **4.5 Continued Maestro Error IDs** and 4.9 **NC Profiler Error IDs on page 57 - 108**. Displays an error

code as -ve or +ve integers.

Figure 16-48 describes the function block for MMC_GetBulkUploadStatus

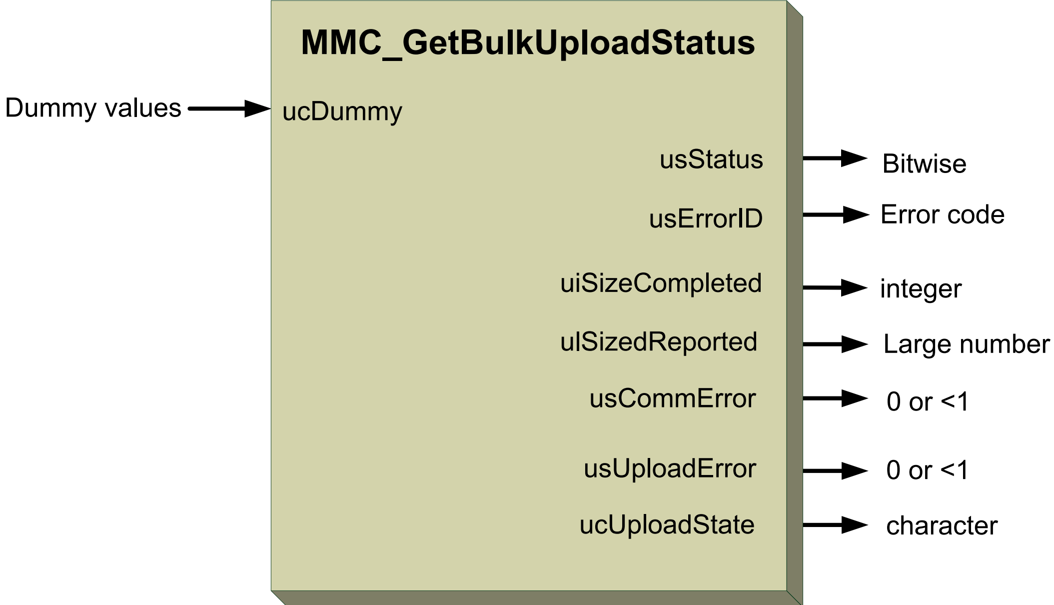


Figure 16-48: MMC_GetBulkUploadStatus function block

16.5.26 MMC_GetBulkUploadData

The Maestro manages the upload process upon request from a host, i.e. the host sends a "begin upload" command, and the Maestro uploads the recording buffer. Afterwards, the host sends this function command to retrieve the data buffer.

```
int MMC_GetBulkUploadDataCmd(IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETBULKUPLOADDATA_IN* pInParam,
OUT MMC_GETBULKUPLOADDATA_OUT* pOutParam)
);
```

Motion Mode	NC –Supported	Distributed – Supported
Source	GMAS\includes\MMC_drive_comm_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETBULKUPLOADDATA_IN** input data structure using the MMC_GetBulkUploadData function.

pOutParam

Points to the **MMC_GETBULKUPLOADDATA_OUT** output structure receiving information as a result of calling the MMC_GetBulkUploadData function.

Remarks

None

Scope

For use only with distributed systems?

MMC_GETBULKUPLOADDATA_IN Structure

```
typedef struct mmc_getbulkuploaddata_in{
unsigned short usStartIndex;
unsigned short usEndIndex;
} MMC_GETBULKUPLOADDATA_IN;
```

Parameters

usStartIndex

Start index value of the uploaded data. +ve values accepted

usEndIndex

End index value of the uploaded data. +ve values accepted

MMC_GETBULKUPLOADDATA_OUT Structure

```
typedef struct mmc_getbulkuploaddata_out{
char cDataBuffer[NC_MAX_REC_PACKET_SIZE];
unsigned short usStatus;
unsigned short sErrorID;
} MMC_GETBULKUPLOADDATA_OUT;
```

Parameters

cDataBuffer[NC_MAX_REC_PACKET_SIZE]

cDataBuffer is the received data. It should be noted that it is the host responsibility to format the data according to its requirements (the data is received as a buffer of bytes)

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-49 describes the function block for MMC_GetBulkUploadData

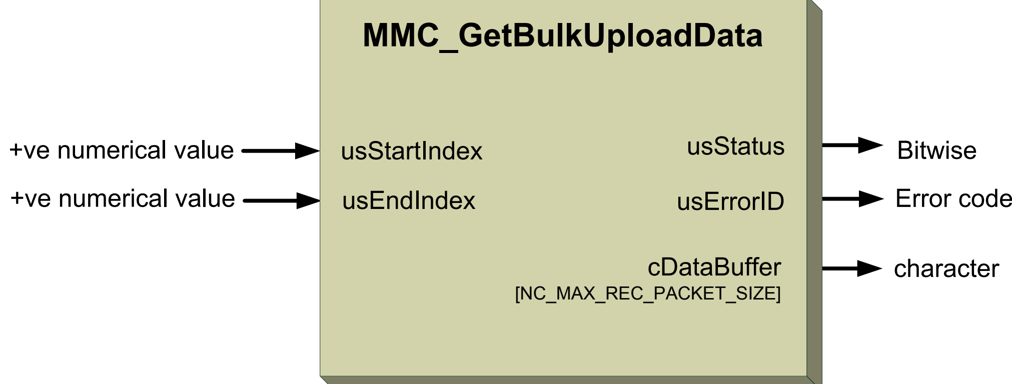


Figure 16-49: MMC_GetBulkUploadData function block

16.6 EtherCAT Drive Communication

Ethernet for Control Automation Technology (EtherCAT) is an open high performance Ethernet-based fieldbus system, which uses the family of industrial computer network protocols used for real-time distributed control, now standardized as IEC 61158. It is a highly flexible Ethernet network protocol that runs over a fast real time Master–Slave network.

The EtherCAT communication speed is up to 100 Mbps full duplex and can include a maximum of 65,535 stations in a single network configuration such as Ethernet star, line or tree without using switches.

Figure 16-50 describes a network of EtherCAT slaves in a ring topology. The Master controls the traffic in the network by initiating the transactions.

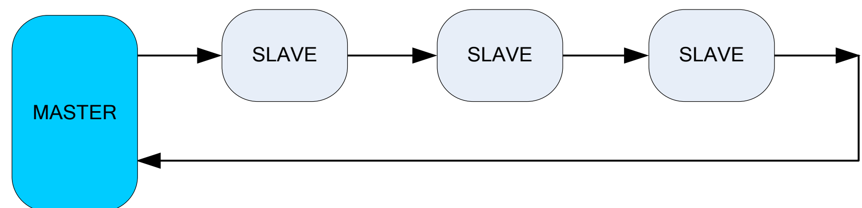


Figure 16-50: EtherCAT Network Configuration

Usually, a control system requires the following in periodic time intervals:

- **Inputs**
  Latched Sensors Data such as Positions, Velocities, Currents, System Status, IO's etc.,
- **Outputs**
  Control Law commands, or Trajectory Information, or Higher Drive Level Commands.

The specific nature of the data transferred via the network depends on the operation mode of the slave drive. The Device Profile describes the application parameters and the functional behavior of the devices including the device class-specific state machines. A common standard for Drive Device Profiles is the DS-402, CANopen Device Profile, and CoE (Can Over EtherCAT).

The EtherCAT protocol is optimized for process data and is transported directly within the standard IEEE 802.3 Ethernet frame. Each Ethernet frame can include several EtherCAT frames, each serving another slave.

EtherCAT network use a processing on the fly, whereby the Ethernet frame is received and processed while the telegram passes through the device. The frames only delay by a fraction of a microsecond in each node. Using EtherCAT, the entire network can be addressed with just one frame.

The data sequence is independent of the physical order of the nodes in the network; addressing can be in any order. Broadcast, multicast and communication between slaves are possible and must be performed by the master device.

The EtherCAT protocol can be inserted into UDP/IP datagrams. This also enables any control with an Ethernet protocol stack to address EtherCAT systems.

Using the Master configuration tool, the Master scans the EtherCAT network and uses the EtherCAT Slave library to compare the slave memory area that includes information about the slave such as Vendor ID, Product Code, and Slave Configuration.

16.6.1 Elmo EtherCAT

The ELMO environment comprises of three levels (**Figure 16-51**):

- EAS EtherCAT configuration tools
- EtherCAT Maestro master
- Elmo EtherCAT slave drives

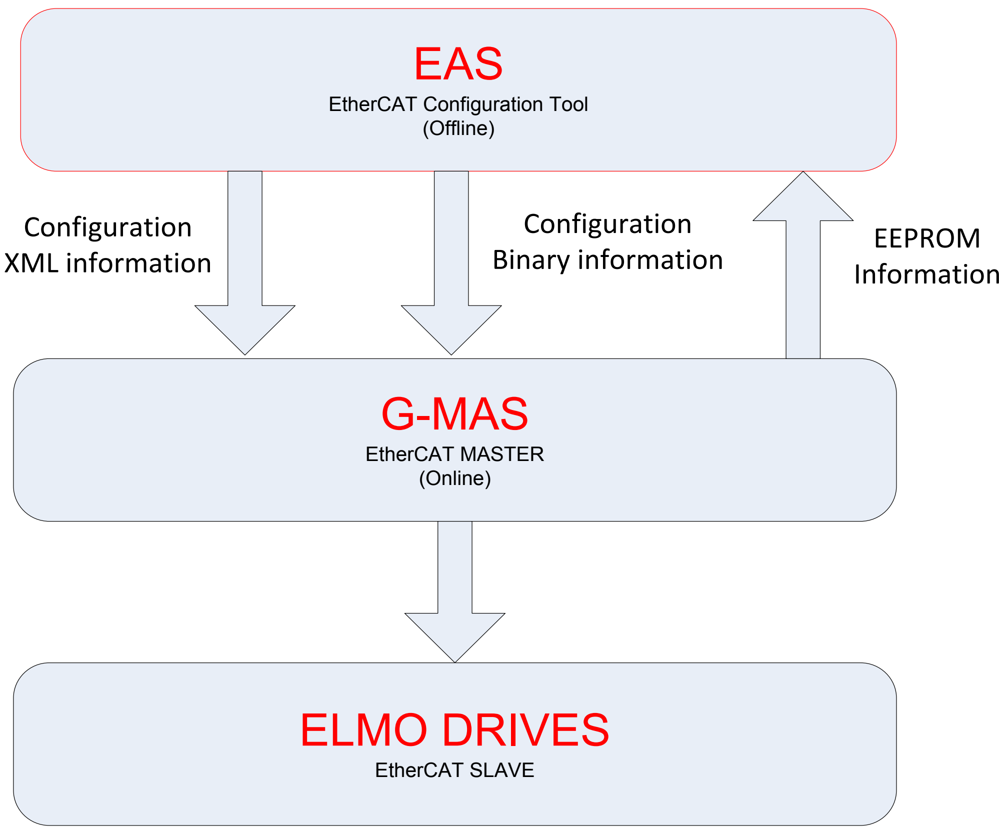


Figure 16-51: EtherCAT Environment

16.6.2 Elmo Slave Drives

The following diagram describes the EtherCAT communication of the drive.

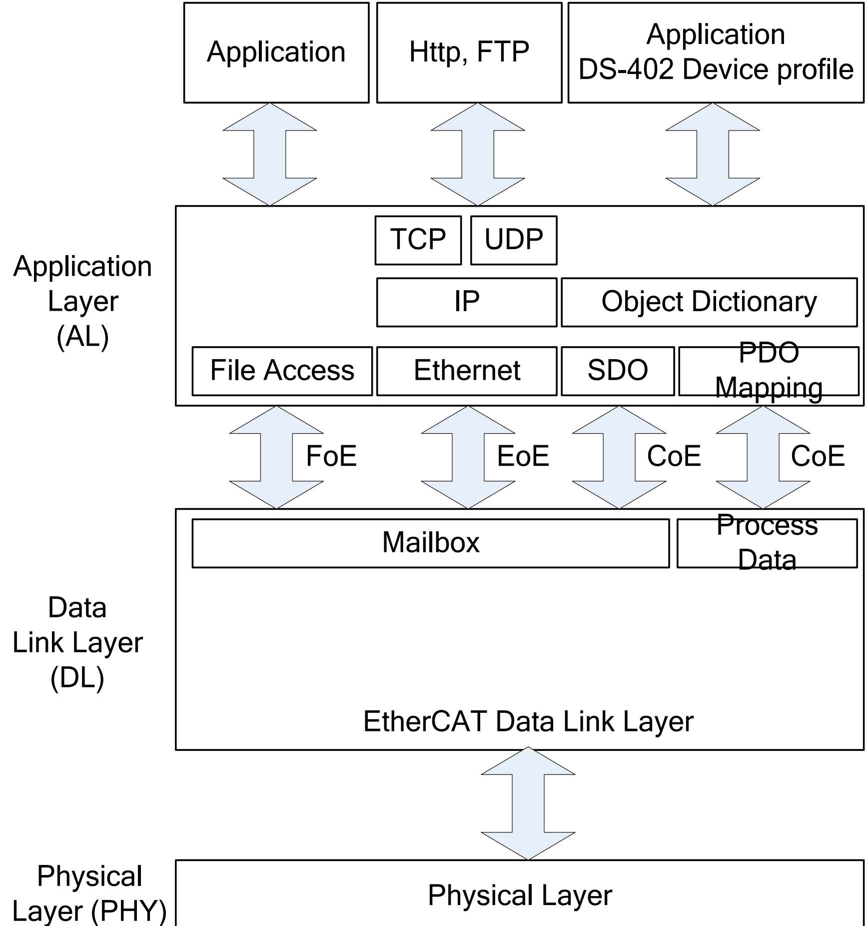


Figure 16-52: Layered Communication protocol in EtherCAT

Physical Layer

The Physical layer of the EtherCAT is a 100Mbits/sec Ethernet port over twisted per cable.

Data Link Layer

This supports two mechanisms of data transfer:

- **Process data**
  Allows writing and reading data simultaneously. This mode is used to transfer the Process data objects (PDO). The PDO transfers via SYNC Manager 2 (PDO_Out) and SYNC Manager 3 (PDO_IN)
- **Mailbox**
  The mailbox mechanism assure that the data will reach to the target. The mailbox is used to transfer the SDOs. The SDO transfers via SYNC Manager 0 (MailboxOut) and SYNC Manager 1 (MailboxIn). SDO objects are used for user triggered access. With SDO services, all of the OD's entries can be read or written. The SDO transport works in asynchronous mode only.

The Elmo drive supports the following communication protocols:

CoE (CANopen over EtherCAT)	Defines a standard way to access the CANopen protocol and includes an object dictionary, SDO, PDO and emergency messages.
EoE (Ethernet over EtherCAT)	Fully Ethernet compatible, defines a standard way to exchange or tunnel standard Ethernet frames.
FoE (File over EtherCAT)	Similar to TFTP, enables access to any data structure in the device, and defines a standard way to download and upload firmware and other files.

The **Object Dictionary (OD)** contains parameters, application data and the mapping information between the process data interface and application data (PDO mapping). Its entries can be accessed via the Service Data Object (SDO).

An Object Dictionary is a naming system that provides a unique identifier to each data item or “object” communicated over the CoE protocol. An object is identified by an index, and if a complex object, by a sub-index as well. CoE and EoE protocols require a set of mandatory objects.

Elmo drive supports distributed clock in order to **synchronize** between the Master and Slaves on the EtherCAT network.

16.6.3 EtherCAT with Maestro

While a single servo drive can run as a stand-alone drive, without the Maestro, using its inner profiler and filter. In order to perform synchronized multi axis motions in the system (such as circle, line etc..), a real time communication protocol must be used, and all drives must be synchronized to a specific SYNC signal in the system. The EtherCAT communication protocol enables synchronization of all the controllers to the same SYNC signal by updating the drives in the system with the Maestro's master time. Thus, all drives in the system are synchronized to the master clock, and all generate an interrupt at exactly the same time.

A profiler can run in the Maestro, on the condition that the axis (axes) is defined as a vector axis (axes). A vector axis may consist of 1 - 16 axes. The Multi Axis Indexer (MAI) is the profiler that runs within the Maestro, which sends (via a high priority interrupt routine) a calculated set point to the axes in the system and can perform vector calculations for up to 16 axes. The profiler EtherCAT and CAN outputs are points that are to be sent to the specific drives belonging to the vector. Therefore, a number of combination options are available:

- 1 x 16 axes (One vector profiler performing profiles for 16 axes),
- 16 x 1 axes (16 profilers for 16 vector axes), or,
- Any combination of M x N axes – as long as M x N < 16

The SYNC interrupt signal to the drives is based on the ET1100 component in the servo drive. The master Maestro does not receive this signal, but can calculate when the SYNC signal is generated. This is because the master EtherCAT in the Maestro is responsible for updating the SYNC cycle time in the servo drives, and therefore knows when the SYNC is generated. The MAI can operate at varying cycle times, dependent on a number of parameters, such as the:

- Desired response from the system
- Number of axes participating in the MAI. The more axes, the higher the cycle rate

16.6.4 EtherCAT Gateway

Under normal circumstances, the Maestro task manages the EtherCAT communications and all devices connected via the EtherCAT. In this situation, adding or removing communications to an Input/Output module or servo drive involves programming the specific EtherCAT resource file. This can involve a quite complex procedure. A more sophisticated alternative exits to stop the operation of the Maestro normal task manager for EtherCAT via the *MMC_EnableEthercatConfigMode* function block, and directly add or remove the Input/Output module or servo drive, automatically updating the EtherCAT resource file during the process. To perform this Gateway process, the EAS application is used. When the process is completed, the function block *MMC_DisableEthercatConfigMode* is run in the Maestro to return the task management to the Maestro using the updated EtherCAT resource file.

16.7 EtherCAT and CANbus Function Blocks

The following EtherCAT drive communication function blocks are described, with the exception of MMC_ETHERCAT_DIAGNOSTICS_INFO, which is a structure:

Drive Communication
MMC_DisableEthercatConfigMode
MMC_EnableEthercatConfigMode
MMC_ECATIODisableDIChangedEvent
MMC_ECATIOEnableDIChangedEvent
MMC_GetCommStatistics
MMC_GetCommDiagnostics
MMC_Get ReactorStatistics

MMC_IsEthercatConfigMode
MMC_ECATIOReadDigitalInput
MMC_ECATIOReadAnalogInput
MMC_ResetCommDiagnostics
MMC_ResetCommStatistics
MMC_SendSDO
MMC_ECATIOWriteAnalogOutput
MMC_ECATIOWriteDigitalOutput

16.7.1 MMC_DisableEthercatConfigMode

Enables the Maestro task manager and disables direct programming of the Maestro via the Gateway.

MMC_LIB_API int MMC_DisableEthercatConfigMode(
IN MMC_CONNECT_HNDL hConn,
(IN MMC_DISABLE_ECATCONFIGMODE_IN* pInParam)
OUT MMC_DISABLE_ECATCONFIGMODE_OUT* pOutParam
);

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_DISABLE_ECATCONFIGMODE_IN** input data structure using the MMC_DisableEthercatConfigMode function.

pOutParam

Points to the **MMC_DISABLE_ECATCONFIGMODE_OUT** output structure receiving information, as a result of calling the MMC_DisableEthercatConfigMode function.

Remarks

Extension of the CANopen technology disabling the Gateway communication between a host system and the Maestro.

Scope

All

MMC_DISABLE_ECATCONFIGMODE_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_DISABLE_ECATCONFIGMODE_IN;
```

Parameters

ucDummy

Dummy value. Any -ve or +ve character.

MMC_DISABLE_ECATCONFIGMODE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_DISABLE_ECATCONFIGMODE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-53 describes the function block for DisableEthercatConfigMode.

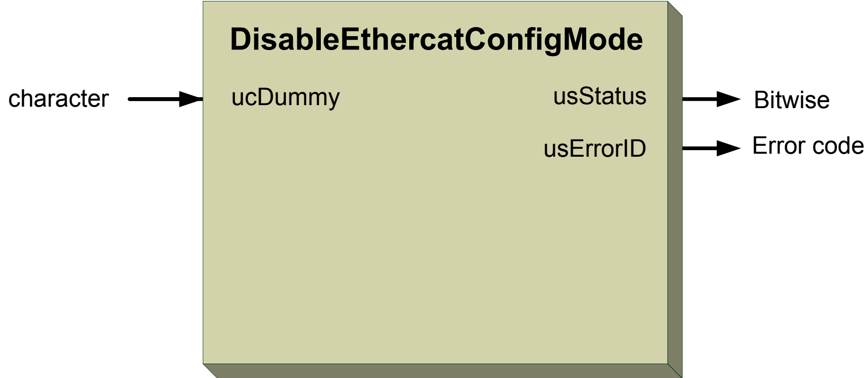


Figure 16-53: DisableEthercatConfigMode function block

16.7.1.2 Function Block Code Example

```
int rc;
MMC_DISABLE_ECATCONFIGMODE_IN       stDisableEcatConfigMode_in;
MMC_DISABLE_ECATCONFIGMODE_OUT      stDisableEcatConfigMode_out;
//
// Inserting the structure parameters:
stDisableEcatConfigMode_in.ucDummy      = 1;     // Dummy input
//
rc = MMC_DisableEthercatConfigMode (hConn, &stDisableEcatConfigMode_out);
if (rc != 0)
{
    HandleError();
}
```

16.7.2 MMC_EnableEthercatConfigMode

Disables the Maestro task manager to enable direct programming of the Maestro via the Gateway.

```
MMC_LIB_API int MMC_EnableEthercatConfigMode(
IN MMC_CONNECT_HNDL hConn,
OUT MMC_ENABLE_ECATCONFIGMODE_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pOutParam

Points to the **MMC_ENABLE_ECATCONFIGMODE_OUT** output structure receiving information, as a result of calling the MMC_EnableEthercatConfigMode function.

Remarks

Extension of the CANopen technology enabling the Gateway communication between a host system and the Maestro.

Scope

All

MMC_ENABLE_ECATCONFIGMODE_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_ENABLE_ECATCONFIGMODE_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-54 describes the function block for MMC_EnableEthercatConfigMode

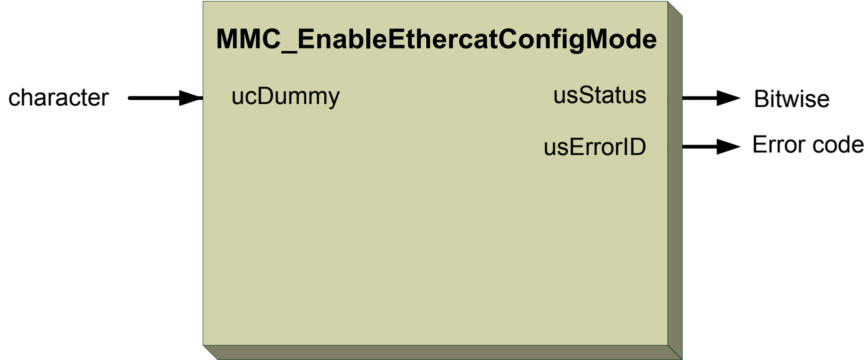


Figure 16-54: MMC_EnableEthercatConfigMode function block

16.7.2.2 Function Block Code Example

```
int rc;
MMC_ENABLE_ECATCONFIGMODE_IN         stEnableEcatConfigMode_in;
MMC_ENABLE_ECATCONFIGMODE_OUT        stEnableEcatConfigMode_out;
//
// Inserting the structure parameters:
stEnableEcatConfigMode_in.ucDummy        = 1;      // Dummy input
//
rc = MMC_EnableEthercatConfigMode (hConn, &stEnableEcatConfigMode_out);
if (rc != 0)
{
    HandleError();
}
```

16.7.3 MMC_ECATIODisableDIChangedEvent

Disables an EtherCAT I/O input event change against an I/O module.

```
MMC_LIB_API int MMC_ECATIODisableDIChangedEvent (
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_DISABLEDICHANGEDEVENT_IN* pInParam,
OUT MMC_DISABLEDICHANGEDEVENT_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_ECATIO_API.h GMAS Programming(IEC 61331 Program)\ElmoECATIO	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_DISABLEDICHANGEDEVENT_IN** input data structure using the MMC_ECATIODisableDIChangedEvent function.

pOutParam

Points to the **MMC_DISABLEDICHANGEDEVENT_OUT** output structure receiving information, as a result of calling the MMC_ECATIODisableDIChangedEvent function.

Remarks

When enabled, any EtherCAT I/O input event change is sent from the I/O module to the Maestro and then host server (if connected).

Scope

All

MMC_DISABLEDICHANGEDEVENT_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_DISABLEDICHANGEDEVENT_IN;
```

Parameters

ucDummy

Dummy data input. Any +ve character value.

MMC_DISABLEDICHANGEDEVENT_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_DISABLEDICHANGEDEVENT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-55 describes the function block for MMC_ECATIODisableDIChangedEvent as applied within the IEC 61131 programming.

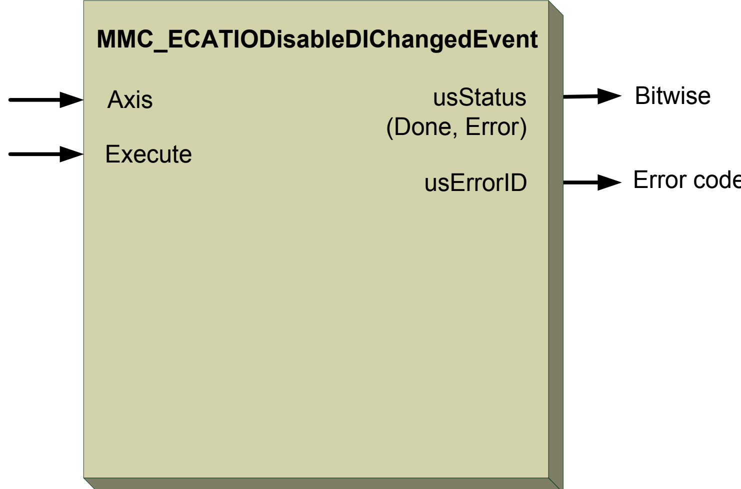


Figure 16-55: MMC_ECATIODisableDIChangedEvent function block

16.7.3.2 Function Block Code Example

```
int rc;
MMC_DISABLEDICHANGEDEVENT_IN        stDisableDIChangeEv_in;
MMC_DISABLEDICHANGEDEVENT_OUT       stDisableDIChangeEv_out;
//
// Inserting the structure parameters:
stDisableDIChangeEv_in.ucDummy  = 1;     //Dummy data input
//
rc = MMC_ECATIODisableDIChangedEvent (hConn, iAxisRef, &stDisableDIChangeEv_in,
&stDisableDIChangeEv_out);
if (rc != 0)
{
    HandleError();
}
```

16.7.4 MMC_ECATIOEnableDIChangedEvent

Enables an EtherCAT I/O input event change.

MMC_LIB_API int MMC_EnableDS401DIChangedEvent(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_ENABLEDICHANGEDEVENT_IN* pInParam,
OUT MMC_ENABLEDICHANGEDEVENT_OUT* pOutParam
);

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_ECATIO_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoECATIO	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_ENABLEDICHANGEDEVENT_IN** input data structure using the MMC_ECATIOEnableDIChangedEvent function.

pOutParam

Points to the **MMC_ENABLEDICHANGEDEVENT_OUT** output structure receiving information, as a result of calling the MMC_ECATIOEnableDIChangedEvent function.

Remarks

When enabled, any EtherCAT I/O input event change is sent from the I/O module to the Maestro and then host server (if connected).

Scope

All

MMC_ENABLEDICHANGEDEVENT_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_ENABLEDICHANGEDEVENT_IN;
```

Parameters

ucDummy

Dummy data input. Any +ve character value.

MMC_ENABLEDICHANGEDEVENT_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_ENABLEDICHANGEDEVENT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-56 describes the function block for MMC_ECATIOEnableDIChangedEvent as applied within the IEC 61131 programming.

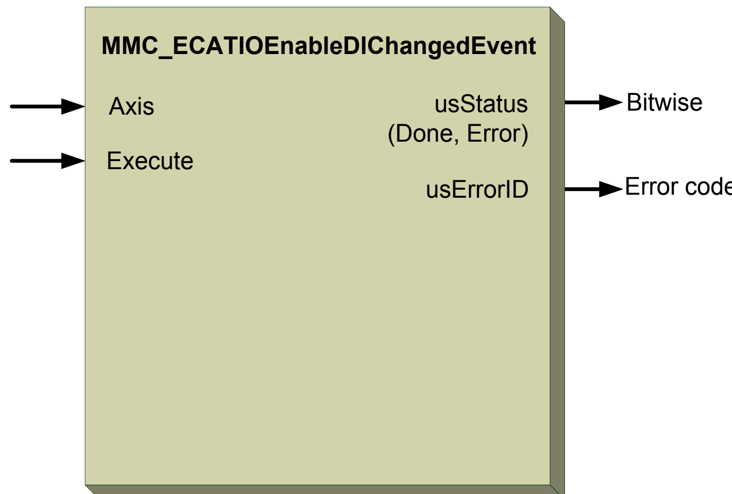


Figure 16-56: MMC_ECATIOEnableDIChangedEvent function block

16.7.4.2 Function Block Code Example

```
int rc;
MMC_ENABLEDICHANGEDEVENT_IN         stEnableDIChangeEv_in;
MMC_ENABLEDICHANGEDEVENT_OUT        stEnableDIChangeEv_out;
//
// Inserting the structure parameters:
stEnableDIChangeEv_in.ucDummy   = 1;     //Dummy data input
//
rc = MMC_ECATIOEnableDIChangedEvent (hConn, iAxisRef, &stEnableDIChangeEv_in,
&stEnableDIChangeEv_out);
if (rc != 0)
{
    HandleError();
}
```

16.7.5 MMC_ECATIOReadDigitalInput

Reads the EtherCAT I/O input of all 64 bit I/O's in one action, increasing the communication speed proportionately versus reading 8 x groups of 8 I/O's.

```
MMC_LIB_API int MMC_ReadDS401DInput(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READDI_IN* pInParam,
OUT MMC_READDI_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_ECATIO_API.h
GMAS Programming(IEC 61331 Program)\ElmoECATIO

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READDI_IN** input data structure using the MMC_ECATIOReadDigitalInput function.

pOutParam

Points to the **MMC_READDI_OUT** output structure receiving information, as a result of calling the MMC_ECATIOReadDigitalInput function.

Remarks

None

Scope

All

MMC_READDI_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_READDI_IN;
```

Parameters

dummy

Dummy data input. Any +ve character value.

MMC_READDI_OUT Structure

```
typedef struct{
#ifdef WIN32
unsigned __int64 ulliDI;
#else
unsigned long long int ulliDI;
#endif
unsigned short usStatus;
short sErrorID;
}MMC_READDI_OUT;
```

Parameters

__int64 ulliDI or ulliDI

If function is defined for WIN32 then use *__int64 ulliDI*, else use *ulliDI*. Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-57 describes the function block for MMC_ECATIOReadDigitalInput as applied within the IEC 61131 programming.

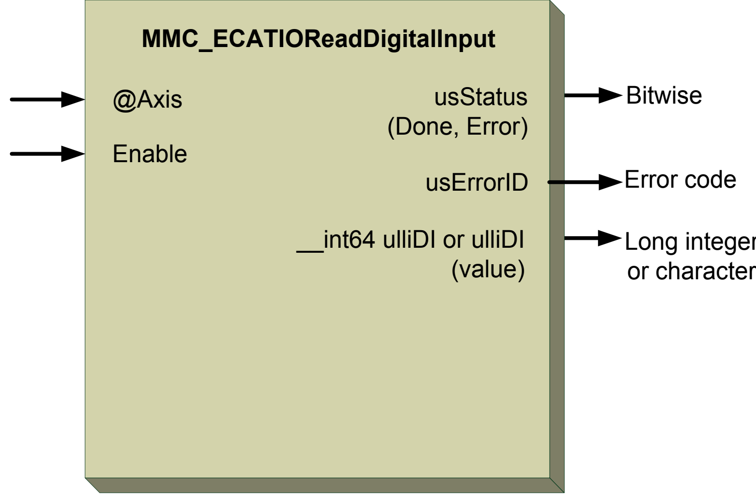


Figure 16-57: MMC_ECATIOReadDigitalInput function block

16.7.5.2 Function Block Code Example

```
int rc;
MMC_READDI_IN      stReadDI_in;
MMC_READDI_OUT     stReadDI_out;
//
// Inserting the structure parameters:
stReadDI_in.dummy      = 1;     //dummy input
//
rc = MMC_ECATIOReadDigitalInput (hConn, iAxisRef, &stReadDI_in, &stReadDI_out);
printf("EtherCAT Input Status[%ld] ErrId[%d]\n", (long int)stReadDI_out.ulliDI,
(short)stReadDI_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.7.6 MMC_ECATIOReadAnalogInput

Reads the EtherCAT I/O analog input.

```
MMC_LIB_API int MMC_ECATIOReadAnalogInput(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READAI_IN* pInParam,
OUT MMC_READAI_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_ECATIO_API.h

GMAS Programming(IEC 61331 Program)\ElmoECATIO

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READAI_IN** input data structure using the MMC_ECATIOReadAnalogInput function.

pOutParam

Points to the **MMC_READAI_OUT** output structure receiving information, as a result of calling the MMC_ECATIOReadAnalogInput function.

Remarks

None

Scope

All

MMC_READAI_IN Structure

```
typedef struct mmc_readai_in{
unsigned char ucIndex;
}MMC_READAI_IN;
```

Parameters

ucIndex

Analog input index. Any +ve character value.

MMC_READAI_OUT Structure

```
typedef struct mmc_readai_out{
short sAI;
unsigned short usStatus;
short sErrorID;
}MMC_READAI_OUT;
```

Parameters

sAI

Analog Input value. Any +ve value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-58 describes the function block for MMC_ECATIOReadAnalogInput as applied within the IEC 61131 programming.

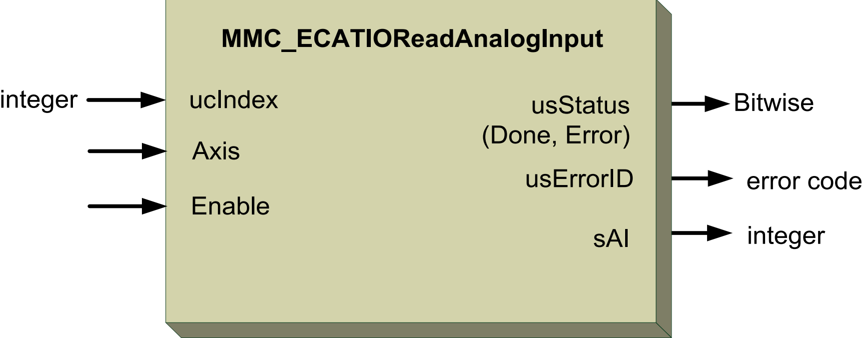


Figure 16-58: MMC_ECATIOReadAnalogInput function block

16.7.6.2 Function Block Code Example

```
int rc;
MMC_READDI_IN      stReadDI_in;
MMC_READDI_OUT     stReadDI_out;
//
// Inserting the structure parameters:
stReadDI_in.dummy      = 1;      //dummy input
//
rc = MMC_ECATIOReadAnalogInput (hConn, iAxisRef, &stReadDI_in, &stReadDI_out);
printf("EtherCAT Analog Input Status[%ld] ErrId[%d]\n", (long int)stReadDI_out.ulliDI,
(short)stReadDI_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.7.7 MMC_ECATIOWriteAnalogOutput

Writes to the EtherCAT I/O analog outputs.

```
MMC_LIB_API int MMC_ECATIOWriteAnalogOutput(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEAO_IN* pInParam,
OUT MMC_WRITEAO_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_ECATIO_API.h

GMAS Programming(IEC 61331 Program)\ElmoECATIO

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEAO_IN** input data structure using the MMC_ECATIOWriteAnalogOutput function.

pOutParam

Points to the **MMC_WRITEAO_OUT** output structure receiving information, as a result of calling the MMC_ECATIOWriteAnalogOutput function.

Remarks

None

Scope

All

MMC_WRITEAO_IN Structure

```
typedef struct mmc_writeao_in{
short sAO;
unsigned char ucIndex;
}MMC_WRITEAO_IN;
```

Parameters

sAO

Analog Output value. Any +ve value.

ucIndex

Analog input index. Any +ve character value.

MMC_WRITEAO_OUT Structure

```
typedef struct mmc_writeao_out{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEAO_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-59 describes the function block for MMC_ECATIOWriteAnalogOutput as applied within the IEC 61131 programming.

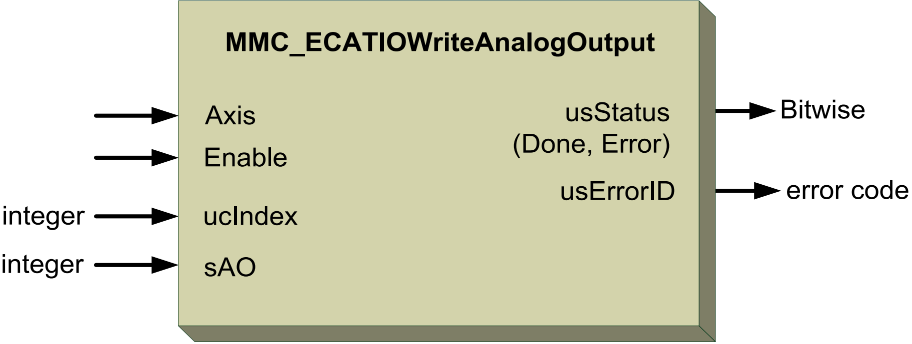


Figure 16-59: MMC_ECATIOWriteAnalogOutput function block

16.7.7.2 Function Block Code Example

```
int rc;
MMC_WRITEDO_IN         stWriteDO_in;
MMC_WRITEDO_OUT        stWriteDO_out;
//
// Inserting the structure parameters:
stWriteDO_in.ulliDO    = 1;     //Index of the group axes
//
rc = MMC_ECATIOWriteDigitalOutput (hConn, iAxisRef, &stWriteDO_in, &stWriteDO_out);
if (rc != 0)
{
    HandleError();
}
```

16.7.8 MMC_ECATIOWriteDigitalOutput

Writes to the EtherCAT I/O outputs of all 64 bit I/O's in one action, increasing the communication speed proportionately versus writing to 8 x groups of 8 I/O's.

```
MMC_LIB_API int MMC_ECATIOWriteDigitalOutput(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEDO_IN* pInParam,
OUT MMC_WRITEDO_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_ECATIO_API.h GMAS Programming(IEC 61331 Program)\ElmoECATIO	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEDO_IN** input data structure using the MMC_ECATIOWriteDigitalOutput function.

pOutParam

Points to the **MMC_WRITEDO_OUT** output structure receiving information, as a result of calling the MMC_ECATIOWriteDigitalOutput function.

Remarks

None

Scope

All

MMC_WRITEDO_IN Structure

```
typedef struct{
#ifdef WIN32
unsigned __int64 ulliDO;
#else
unsigned long long int ulliDO;
#endif
}MMC_WRITEDO_IN;
```

Parameters

__int64 ulliDI or ulliDI

If function is defined for WIN32 then use *__int64 ulliDI*, else use *ulliDI*. Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

MMC_WRITEDO_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEDO_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-60 describes the function block for MMC_ECATIOWriteDigitalOutput as applied within the IEC 61131 programming.

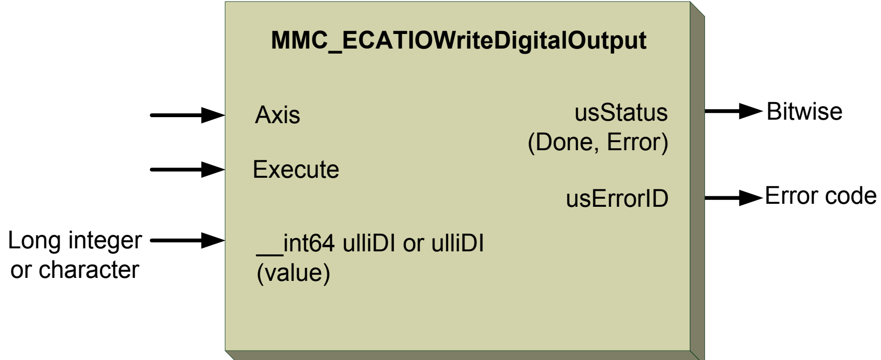


Figure 16-60: MMC_ECATIOWriteDigitalOutput function block

16.7.8.2 Function Block Code Example

```
int rc;
MMC_WRITEDO_IN          stWriteDO_in;
MMC_WRITEDO_OUT         stWriteDO_out;
//
// Inserting the structure parameters:
stWriteDO_in.ulliDO    = 1;     //Index of the group axes
//
rc = MMC_ECATIOWriteDigitalOutput (hConn, iAxisRef, &stWriteDO_in, &stWriteDO_out);
if (rc != 0)
{
    HandleError();
}
```

16.7.9 MMC_GetCommStatistics

Receives communication statistics for a specific axis.

It is recommended to use the function **MMC_GetEthercatCommStatistics on page 1367** rather than this function. The GetCommStatistics function is for legacy applications only.

```
MMC_LIB_API int MMC_GetCommStatistics(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETCOMMSTATISTICS_IN* pInParam,
OUT MMC_GETCOMMSTATISTICS_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoGlobal	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETCOMMSTATISTICS_IN** input data structure using the MMC_GetCommStatistics function.

pOutParam

Points to the **MMC_GETCOMMSTATISTICS_OUT** output structure receiving information as a result of calling the MMC_GetCommStatistics function.

Remarks

None

Scope

All

MMC_GETCOMMSTATISTICS_IN Structure

```
typedef struct{
unsigned short usAxesRef        /*usSlaveID*/;
}MMC_GETCOMMSTATISTICS_IN;
```

Parameters

usAxesRef

Axis/group reference handle type returned by GetAxisRef command

MMC_GETCOMMSTATISTICS_OUT Structure

```
typedef struct{
unsigned long dwSendErrors;
unsigned long dwReceiveErrors;
unsigned long dwWrongWC;
unsigned long dwParseErrors;
unsigned short usNumOfSlaves;
unsigned short usStatus;
short sErrorID;
unsigned char ucMasterState;
unsigned char ucSlaveState;
}MMC_GETCOMMSTATISTICS_OUT;
```

Parameters

dwSendErrors

Send errors counter. Any +ve 32 bit value.

dwReceiveErrors

Receive errors counter. Any +ve 32 bit value.

dwWrongWC

Wrong writer counter number. Any +ve 32 bit value

dwParseErrors

Parse error counter. Any +ve 32 bit value.

usNumOfSlaves

Checks the realistic number of drives connected to the Maestro. Limited +ve integer value.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

ucMasterState

Maestro specification. +ve character value

ucSlaveState

Specification state of the slaves.

Figure 16-61 describes the function block for MMC_GetCommStatistics as applied within the IEC 61131 programming.

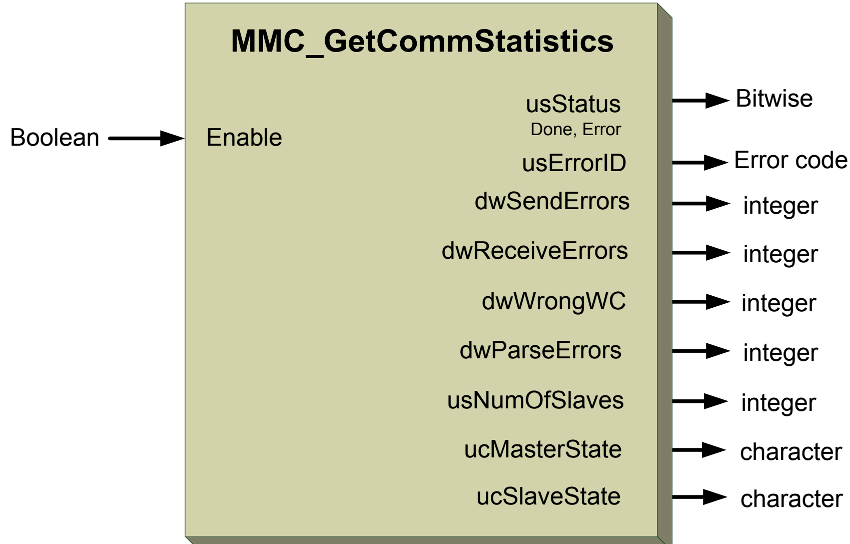


Figure 16-61: MMC_GetCommStatistics function block

16.7.9.2 Function Block Code Example

```
int rc;
MMC_GETCOMMSTATISTICS_IN        stGetCOMMStats_in;
MMC_GETCOMMSTATISTICS_OUT       stGetCOMMStats_out;
//
// Inserting the structure parameters:
stGetCOMMStats_in.usAxesRef     = 19;     //Axis reference
//
rc = MMC_GetCommStatistics (hConn, iAxisRef, &stGetCOMMStats_out);
printf("Comm Statistics Status[%ld][%ld][%ld][%ld] ErrId[%d]\n", (long
int)stGetCOMMStats_out.dwSendErrors, (long int)stGetCOMMStats_out.dwReceiveErrors, (long
int)stGetCOMMStats_out.dwWrongWC, (long int)stGetCOMMStats_out.dwParseErrors,
(short)stGetCOMMStats_out.sErrorID);
printf("Comm Statistics Status[%ld][%ld][%ld]\n", (long int)stGetCOMMStats_out.usNumOfSlaves,
(long int)stGetCOMMStats_out.ucMasterState, (long int)stGetCOMMStats_out.ucSlaveState);
if (rc != 0)
{
    HandleError();
}
```

16.7.10 MMC_GetEthercatCommStatistics

Obtains the EtherCAT communication statistics used as part of the FoE download mechanism in the EAS application.

```
MMC_LIB_API int MMC_GetEthercatCommStatistics(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETCOMMSTATISTICSEX_IN* pInParam,
OUT MMC_GETCOMMSTATISTICSEX_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_general_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETCOMMSTATISTICSEX_IN** input data structure using the MMC_GetEthercatCommStatistics function.

pOutParam

Points to the **MMC_GETCOMMSTATISTICSEX_OUT** output structure receiving information, as a result of calling the MMC_GetEthercatCommStatistics function.

Remarks

A maximum number of 76 slaves are allowed to be attached for downloading using this single function.

Scope

EtherCAT

MMC_GETCOMMSTATISTICSEX_IN Structure

```
typedef struct mmc_getcommstatisticsex_in{
unsigned short pwSlaveId[ETHERCAT_STATISTICSEX_MAX_SLAVES];
unsigned char ucSlavesNum;
}MMC_GETCOMMSTATISTICSEX_IN;
```

Parameters

pwSlaveId[ETHERCAT_STATISTICSEX_MAX_SLAVES]

The array of slave ID that the user wishes to view with a maximum number of 76 slaves. The user enters each slave ID from 0 to to the (maximum number of slave supported-1).

ucSlavesNum

The number of slaves that the user wishes to view attached to the master Maestro. Values are +ve characters

MMC_GETCOMMSTATISTICSEX_OUT Structure

```
typedef struct mmc_getcommstatisticsex_out{
unsigned long dwSendErrors;
unsigned long dwReceiveErrors;
unsigned long dwWrongWC;
unsigned long dwParseErrors;
MMC_ECAT_SII_CONTENT pstSII_Content[ETHERCAT_STATISTICSEX_MAX_SLAVES];
unsigned short usNumOfSlaves;
unsigned short usStatus;
unsigned short sErrorID;
unsigned char ucMasterState;
unsigned char pucAxesState[ETHERCAT_STATISTICSEX_MAX_SLAVES];
unsigned char pucAxesDiagnosticState[ETHERCAT_STATISTICSEX_MAX_SLAVES];
unsigned char ucMasterDiagnosticState;
}MMC_GETCOMMSTATISTICSEX_OUT;
```

Parameters

dwSendErrors

The number of EtherCAT transmission errors. Any +ve value is recieved.

dwReceiveErrors

The number of EtherCAT recieve errors. Any +ve value is recieved.

dwWrongWC

The value of the incorrect working counter. Any +ve value is recieved.

dwParseErrors

The number of Parse errors encountered. Any +ve value is recieved.

MMC_ECAT_SII_CONTENT pstSII_Content[ETHERCAT_STATISTICSEX_MAX_SLAVES]

The details of the drive connected to the Master, the ECAT System Information in an array of system information for a maximum number of 76 slaves.

```
typedef struct _MMC_ECAT_SII_CONTENT{
unsigned long ulVendorId;
unsigned long ulProductCode;
unsigned long ulRevisionNo;
```

unsigned long ulSerialNo;
} MMC_ECAT_SII_CONTENT;

ulVendorId

The vendor ID of the slave drive. Any +ve value is recieved.

ulProductCode

The product code for the slave drive. Any +ve value is recieved.

ulRevisionNo

The revision number of the save drive. Any +ve value is recieved.

ulSerialNo

The serial number of the slave drive. Any +ve value is recieved.

usNumOfSlaves

The number of slaves attached to the master Maestro. Values are +ve numbers

ucMasterState

The state of the master according to the following definition:

EcatStateNotSet	= 0x00	State is not set
EcatStateI	= 0x01	EtherCAT State "Init"
EcatStateP	= 0x02	EtherCAT State "Pre-Operational”
EcatStateB	= 0x03	EtherCAT State "Bootstrap”
EcatStateS	= 0x04	EtherCAT State "Safe-Operational”
EcatStateO	= 0x08	EtherCAT State "Operational"

pucAxesState[ETHERCAT_STATISTICSEX_MAX_SLAVES]

The array of slave states with a maximum of 76, each displayed with the state defined by the following:

EcatStateNotSet	= 0x00	State is not set
EcatStateI	= 0x01	EtherCAT State "Init"
EcatStateP	= 0x02	EtherCAT State "Pre-Operational”
EcatStateB	= 0x03	EtherCAT State "Bootstrap”
EcatStateS	= 0x04	EtherCAT State "Safe-Operational”
EcatStateO	= 0x08	EtherCAT State "Operational"

pucAxesDiagnosticState[ETHERCAT_STATISTICSEX_MAX_SLAVES]

The diagnostics of an array of slaves with a maximum number of 76 slaves. This diagnostics has the following values, and explanations:

Slave doesn't respond to commands	EcatSlaveDiagnosticStateOffLine	0x00000001
Slave's state is different as the set one	EcatSlaveDiagnosticStateErrorEcatState	0x00000002
Slave is not configured	EcatSlaveDiagnosticStateNotConfigured	0x00000004
Slave's configuration doesn't match the configuration for this slave found in master	EcatSlaveDiagnosticStateWrongConfiguration	0x00000008
	EcatSlaveDiagnosticStateInitCmdError	0x00000010
	EcatSlaveDiagnosticStateMailboxInitCmdError	0x00000020

ucMasterDiagnosticState

The diagnostics of the Master with the following values and explanations:

Diagnostics completed successfully	EcatMasterDiagnosticStateUpdated	0x00000001
Error while sending/receiving a frame	EcatMasterDiagnosticStateSendReceiveError	0x00000002
Error while processing the received frame	EcatMasterDiagnosticStateParseError	0x00000004
No connection between the NIC adapter and slaves	EcatMasterDiagnosticStateLinkDown	0x00000008
Wrong configuration	EcatMasterDiagnosticStateWrongConfiguration	0x00000010
Slave-to-slave timeout	EcatMasterDiagnosticStateS2STimeout	0x00000020
Default data was set	EcatMasterDiagnosticStateDefaultDataWasSet	0x00000040
WatchDogTimeOut	EcatMasterDiagnosticStateWatchDogTimeOut	0x00000080

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-62 describes the function for MMC_GetEthercatCommStatistics as applied within the IEC 61131 programming.

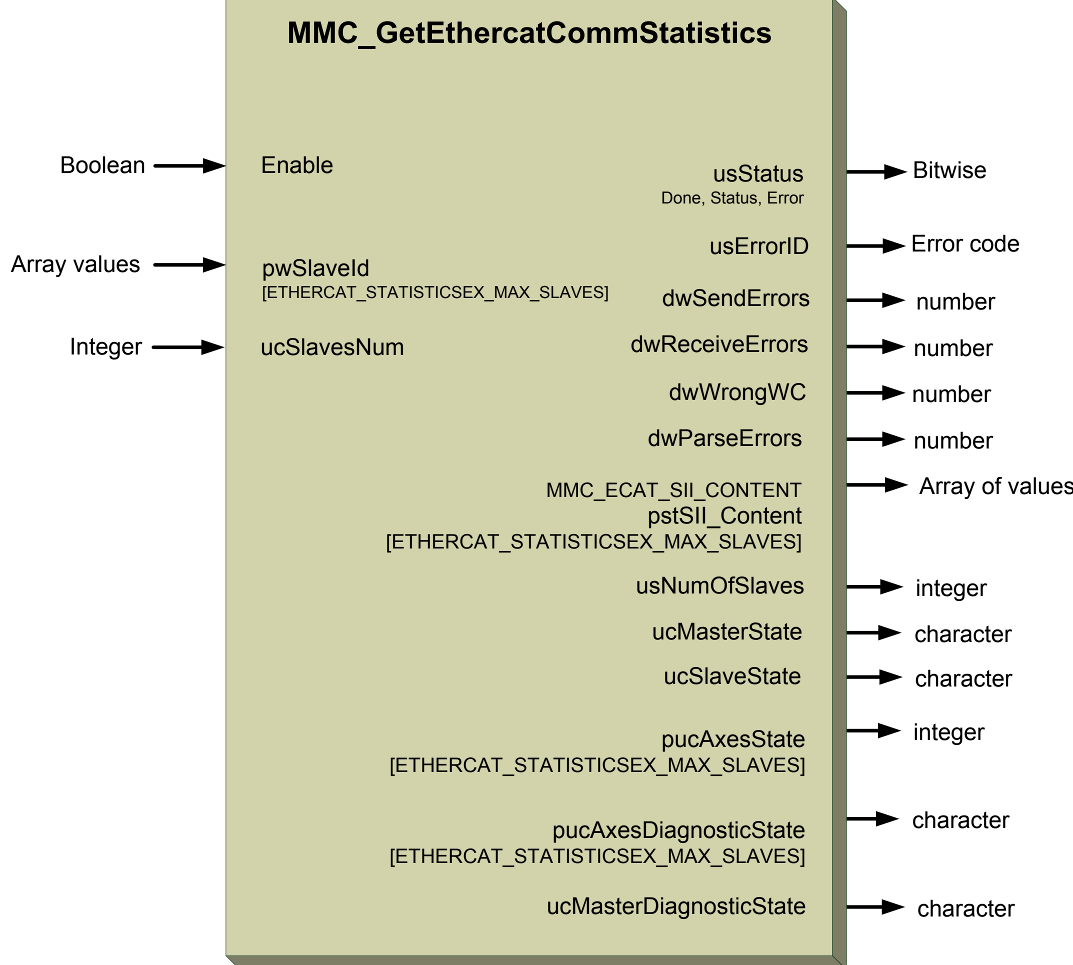


Figure 16-62: MMC_GetEthercatCommStatistics function

16.7.10.2 Function Code Example

```
void DownloadFoe()
{
     MMC_DOWNLOADFOE_IN dlfoe ;
     MMC_DOWNLOADFOE_OUT dlfoeout ;
     MMC_GETFOESTATUS_OUT foestat ;
     MMC_GET_GMASOP_MODE_OUT pOpmode ;

     MMC_GETCOMMSTATISTICSEX_IN gcstat_In ;
     MMC_GETCOMMSTATISTICSEX_OUT gcstat_Out ;
     int i ;
     //
     //
     // Before DownloadingFOE - It is good practice that drives will be reset because
     // if one of the drives is after DownloadFoE and was not reset, its state and statitstics
are unknown.
     //
     dlfoe.pwSlaveId[0]=0 ;     // Note: Slave ID is inserted here !!
     dlfoe.pwSlaveId[1]=1 ;     // Note: Slave ID is inserted here !!
     //
```

```
    dlfoe.ucSlavesNum = 2;      // Number of relevant slaves in the pwSlaveId array.
    //
    // Same for slave statistics:
    gcstat_In.pwSlaveId[0] = 0 ;
    gcstat_In.pwSlaveId[1] = 1 ;
    gcstat_In.ucSlavesNum = 2 ;
    //
    // Insert IP of tftp server. Usually the connection IP of the PC.
    dlfoe.pucServer[0] = 10 ;
    dlfoe.pucServer[1] = 10 ;
    dlfoe.pucServer[2] = 20 ;
    dlfoe.pucServer[3] = 55 ;
    //
    // Copy file path name to the structure. Should be relative to the tftp folder
    strcpy(dlfoe.pcFileName,"FoEFW  01.01.04.68 27Oct2011P01G.abs") ;

    // Start tftp server on host. Only then call the MMC_DownloadFoE.
    //
    int rc = MMC_DownloadFoE(conn_hndl,&dlfoe,&dlfoeout)  ;
    if(rc < 0)
    {
        // Error Calling MMC_DownloadFoE. Error in dlfoeout.sErrorID
        return ;
    }
    //
    // If we reached this line, the tftp was succesful. Poll the GMAS for results:
    while(TRUE)
    {
        Sleep(100) ;
        //
        // Check the FoE progress:
        MMC_GetFoEStatus(conn_hndl,&foestat);
        if(rc < 0)
        {
            // Error Calling MMC_GetFoEStatus. Error in foestat.sErrorID
            return ;
        }
        //
        // Check that the FoE started.
        if(foestat.ucFOEStarted)
        {
            rc = MMC_GetGMASOperationMode(conn_hndl,&pOpmode) ;
            if(rc < 0)
            {
                // Error Calling MMC_GetGMASOperationMode. Error in pOpmode.sErrorID
                return ;
            }
            // Print Remaining time - foestat.ucProgress
            //
            // Check Foe Download progress is over and GMAS back in operational mode.
            if  ((foestat.ucProgress == 0) && (pOpmode.ucResult == 0))
            {
                //
                // Download over. Check to see if any drives failed.
                for(i = 0 ; i < dlfoe.ucSlavesNum ; i++)
                {
                    if(foestat.pstSlavesErrorID[i].sErrorID != 0)
                    {
                        // Error on one of the slaves. Print error:
                        // SlaveID: foestat.pstSlavesErrorID[i].usSlaveID  has error -
foestat.pstSlavesErrorID[i]
                    }
                }
                // Notify user to switch drives Off / On and then check the download status.
wait 5 sec's.
                //
                // please note - MAX 76 slaves can be read.
                rc = MMC_GetEthercatCommStatistics(conn_hndl,&gcstat_In,&gcstat_Out) ;
                if(rc < 0)
```

```
                  {
                      // Error Calling MMC_GetEthercatCommStatistics. Error in
gcstat_Out.sErrorID
                      return ;
                  }
                  //
                  // gcstat_Out.ucMasterState - Should be EcatStateO  . operational.
                  // Good idea to read number of slaves on bus - gcstat_Out.usNumOfSlaves
                  // gcstat_Out.ucMasterDiagnosticState - All bits should be 0, except for:
EcatMasterDiagnosticStateUpdated, EcatMasterDiagnosticStateDefaultDataWasSet bits.
                  //
                  for(i = 0 ; i < gcstat_In.ucSlavesNum ; i++)
                  {
                      // gcstat_Out.pucAxesState[i] - Should be EcatStateO.
                      // gcstat_Out.pucAxesDiagnosticState[i] - Should be 0.
                      if((gcstat_Out.pstSII_Content[i].ulVendorId == 0x9A) &&
(gcstat_Out.pstSII_Content[i].ulRevisionNo <= 0xFF))
                      {
                          //
                          // Drive stuck in no firmware state. Notify User. In This case the
InitCmdFail in diagnostics is not relevant.
                      }
                  }
                  return ;
              }
          }
     }
}

int OnConnectGetDiagnostics()
{
     int rc ;
     MMC_GET_GMASOP_MODE_OUT pOpmode ;

     MMC_GETCOMMSTATISTICSEX_IN gcstat_In ;
     MMC_GETCOMMSTATISTICSEX_OUT gcstat_Out ;
     int i ;
     //
     rc = MMC_GetGMASOperationMode(conn_hndl,&pOpmode) ;
     if(rc < 0)
     {
          // Error Calling MMC_GetGMASOperationMode. Error in pOpmode.sErrorID
          return ;
     }
     //
     // Check GMAS Operational state. If == 2, then in Download FOE state.
     if (pOpmode.ucResult == 2)
     {
          // GMAS in Download FoE state. We decided that a mesage will be shown to user that
the GMAS is in Download FoE.
     }

     rc = MMC_GetEthercatCommStatistics(conn_hndl,&gcstat_In,&gcstat_Out) ;
     if(rc < 0)
     {
          // Error Calling MMC_GetEthercatCommStatistics. Error in gcstat_Out.sErrorID
          return ;
     }
     //
     // gcstat_Out.ucMasterState - Should be EcatStateO. operational.
     // Good idea to read number of slaves on bus - gcstat_Out.usNumOfSlaves. Should be
identical to number of drives configured.
     // gcstat_Out.ucMasterDiagnosticState - All bits should be 0, except for:
EcatMasterDiagnosticStateUpdated, EcatMasterDiagnosticStateDefaultDataWasSet bits.
     //
     for(i = 0 ; i < gcstat_In.ucSlavesNum ; i++)
     {
          // gcstat_Out.pucAxesState[i] - Should be EcatStateO.
          //
```

```
        if((gcstat_Out.pstSII_Content[i].ulVendorId == 0x9A) &&
(gcstat_Out.pstSII_Content[i].ulRevisionNo <= 0xFF))
        {
            //
            // One of the Drives is stuck in no firmware state. Notify User to go to
diagnostics tab - NOT TO CONFIGURATOR.
            // In this case - the gcstat_Out.pucAxesDiagnosticState[i] InitCmd bit may be
set..
        }
        else
        {
            // pucAxesDiagnosticState[i] should be 0 !
        }

    }
}
```

16.7.11 MMC_GetCommDiagnostics

Receives communication diagnostics for specific axis.

```
MMC_LIB_API int MMC_GetCommDiagnostics(
IN MMC_CONNECT_HNDL hConn,
IN MMC_GETCOMMDIAGNOSTICS_IN* pInParam,
OUT MMC_GETCOMMDIAGNOSTICS_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoGlobal	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_GETCOMMDIAGNOSTICS_IN** input data structure using the MMC_GetCommDiagnostics function.

pOutParam

Points to the **MMC_GETCOMMDIAGNOSTICS_OUT** output structure receiving information, as a result of calling the MMC_GetCommDiagnostics function.

Remarks

None

Scope

Not limited

MMC_GETCOMMDIAGNOSTICS_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_GETCOMMDIAGNOSTICS_IN;
```

Parameters

ucDummy

Dummy data input. Any +ve character value.

MMC_GETCOMMDIAGNOSTICS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
MMC_ETHERCAT_DIAGNOSTICS_INFO pDiagnosticsSlavesArr[ETHERCAT_ID_MAX];
}MMC_GETCOMMDIAGNOSTICS_OUT;
```

Parameters

pDiagnosticsSlavesArr

pDiagnosticsSlavesArr is an enumerator and ID integer ranging between [1 – 100] describing the maximum number of all EtherCAT connected systems, including the Maestro.

Where [ETHERCAT_ID_MAX] is an ID with values [1 – 100] array of all slaves attached to master bus.

MMC_ETHERCAT_DIAGNOSTICS_INFO defines the EtherCAT diagnostics info. This structure physically checks the connection of the four connection ports of the Maestro, and checks for three possible types of errors.

ucRXErrorsPort0

Receiving errors at the Port 0. Any +ve character

ucInvalidFramesPort0

Invalid frames at the Port 0. Any +ve character

ucLostLinkErrorsPort0

Lost link errors at the Port 0. Any +ve character

ucRXErrorsPort1

Receiving errors at the Port 1. Any +ve character

ucInvalidFramesPort1

Invalid frames at the Port 1. Any +ve character

ucLostLinkErrorsPort1

Lost link errors at the Port 1. Any +ve character

ucRXErrorsPort2

Receiving errors at the Port 2. Any +ve character

ucInvalidFramesPort2

Invalid frames at the Port 2. Any +ve character

ucLostLinkErrorsPort2

Lost link errors at the Port 2. Any +ve character

ucRXErrorsPort3

Receiving errors at the Port 3. Any +ve character

ucInvalidFramesPort3

Invalid frames at the Port 3. Any +ve character

ucLostLinkErrorsPort3

Lost link errors at the Port 3. Any +ve character

Figure 16-63 describes the function block for MMC_GetCommDiagnostics as applied within the IEC 61131 programming.

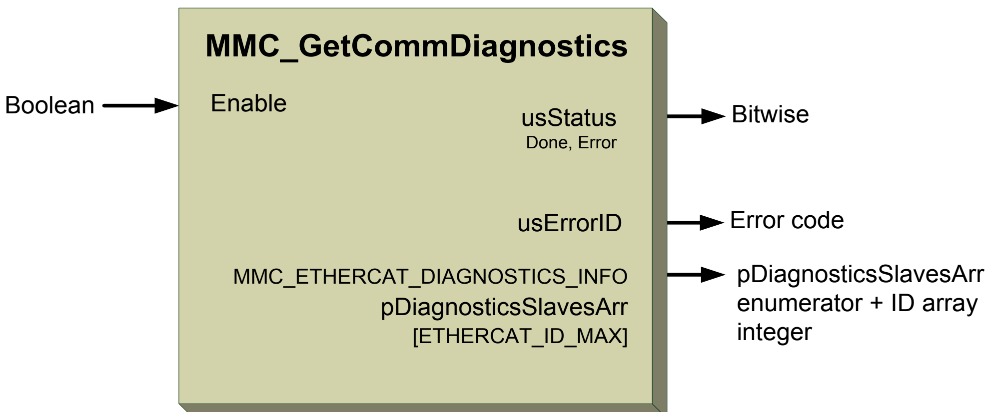


Figure 16-63: MMC_GetCommDiagnostics function block

16.7.11.2 Function Block Code Example

```
int rc;
MMC_GETCOMMDIAGNOSTICS_IN       stGetCOMMDiags_in;
MMC_GETCOMMDIAGNOSTICS_OUT      stGetCOMMDiags_out;
//
// Inserting the structure parameters:
stGetCOMMDiags_in.ucDummy  = 1; //Dummy data input
//
rc = MMC_GetCommDiagnostics (hConn, &stGetCOMMDiags_in, &stGetCOMMDiags_out);
printf("Comm Diagnostics Status[%ld] ErrId[%d]\n", (long
int)stGetCOMMDiags_out.pDiagnosticsSlavesArr, (short)stGetCOMMDiags_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.7.12 MMC_GetReactorStatistics

Obtains the statistics from the Maestro server base processor.

```
MMC_LIB_API int MMC_GetReactorStatistics(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_GETREACTORSTATISTICS_IN* pInParam,
OUT MMC_GETREACTORSTATISTICS_OUT* pOutParam
);
```

Motion Mode	NC – Supported	Distributed –Supported
Source	C:\GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_GETREACTORSTATISTICS_IN** input data structure using the MMC_GetReactorStatistics function.

pOutParam

Points to the **MMC_GETREACTORSTATISTICS_OUT** output structure receiving information as a result of calling the MMC_GetReactorStatistics function.

Remarks

This base processor is responsible for organizing the PDO's and performing their respective operations in their relevant order. This function block counts the number of PDO's sent to the base processor and issues a report of the total count.

Scope

All

MMC_GETREACTORSTATISTICS_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_GETREACTORSTATISTICS_IN;
```

Parameters

ucDummy

Dummy input data. Any +ve character value.

MMC_GETREACTORSTATISTICS_OUT Structure

```
typedef struct{
int iReactorQueueSize;
unsigned short usStatus;
unsigned short sErrorID;
}MMC_GETREACTORSTATISTICS_OUT;
```

Parameters

iReactorQueueSize

Returned reactor queue size inquiry integer. Integer with any +ve value

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-64 describes the function block for MMC_GetReactorStatistics as applied within the IEC 61131 programming.

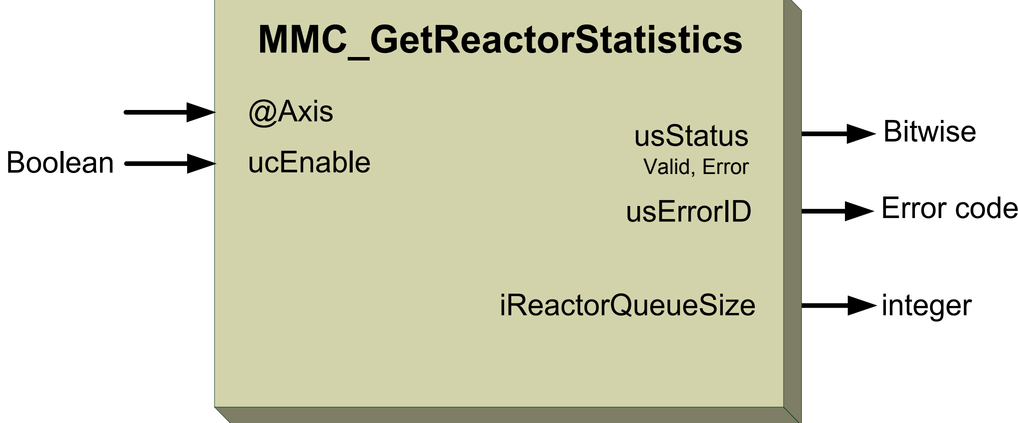


Figure 16-64: MMC_GetReactorStatistics function block

16.7.12.2 Function Block Code Example

```
int rc;
MMC_GETREACTORSTATISTICS_IN         stGetReactorStats_in;
MMC_GETREACTORSTATISTICS_OUT        stGetReactorStats_out;
//
// Inserting the structure parameters:
stGetReactorStats_in.ucDummy    = 1; //Dummy data input
//
rc = MMC_GetReactorStatistics (hConn, iAxisRef, &stGetReactorStats_in,
&stGetReactorStats_out);
printf("Reactor Statistics[%ld] ErrId[%d]\n", (long
int)stGetReactorStats_out.iReactorQueueSize, (short)stGetReactorStats_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.7.13 MMC_IsEthercatConfigMode

Defines whether the EtherCAT configuration mode is operational or not.

```
MMC_LIB_API int MMC_IsEthercatConfigMode(
IN MMC_CONNECT_HNDL hConn,
IN MMC_IS_ECATCONFIGMODE_IN* pInParam
OUT MMC_IS_ECATCONFIGMODE_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed - Supported

Source C:\GMAS\includes\MMC_drive_comm_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_IS_ECATCONFIGMODE_IN** input data structure using the MMC_IsEthercatConfigMode function.

pOutParam

Points to the **MMC_IS_ECATCONFIGMODE_OUT** output structure receiving information as a result of calling the MMC_IsEthercatConfigMode function.

Remarks

Extension of the CANopen technology questioning the Gateway communication between a host system and the Maestro.

Scope

All

MMC_IS_ECATCONFIGMODE_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_IS_ECATCONFIGMODE_IN;
```

Parameters

ucDummy

Dummy input data. Any +ve character value.

MMC_IS_ECATCONFIGMODE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned char ucResult;
}MMC_IS_ECATCONFIGMODE_OUT;
```

Parameters

ucResult

Returned answer to question “Is the EtherCAT Config mode operational?” Yes or No.

Character but Boolean, 0 or 1

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-65 describes the function block for MMC_IsEthercatConfigMode

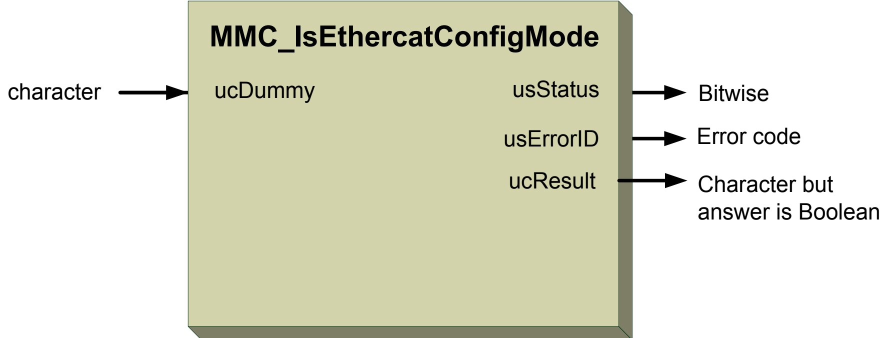


Figure 16-65: MMC_IsEthercatConfigMode function block

16.7.13.2 Function Block Code Example

```
int rc;
MMC_IS_ECATCONFIGMODE_IN        stIsEcatConfigMode_in;
MMC_IS_ECATCONFIGMODE_OUT       stIsEcatConfigMode_out;
//
// Inserting the structure parameters:
stIsEcatConfigMode_in.ucDummy        = 1;      // Dummy input
//
rc = MMC_IsEthercatConfigMode (hConn, &stIsEcatConfigMode_out);
printf("Is EtherCAT Config Mode Status[%ld] ErrId[%d]\n", (long
int)stIsEcatConfigMode_out.ucResult, (short)stIsEcatConfigMode_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.7.14 MMC_ResetCommDiagnostics

Resets the CRC counters registers of all slaves on the bus to 0. The CRC counters registers can be retrieved via the *GetCommDiagnostics* function.

```
MMC_LIB_API int MMC_ResetCommDiagnostics(
IN MMC_CONNECT_HNDL hConn,
IN MMC_RESETCOMMDIAGNOSTICS_IN* pInParam,
OUT MMC_RESETCOMMDIAGNOSTICS_OUT* pOutParam
);
```

Motion Mode	NC – Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoGlobal	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_RESETCOMMDIAGNOSTICS_IN** input data structure using the MMC_ResetCommDiagnostics function.

pOutParam

Points to the **MMC_RESETCOMMDIAGNOSTICS_OUT** output structure receiving information as a result of calling the MMC_ResetCommDiagnostics function.

Remarks

None

Scope

All

MMC_RESETCOMMDIAGNOSTICS_IN Structure

typedef struct{
unsigned char ucDummy;
}MMC_RESETCOMMDIAGNOSTICS_IN;

Parameters

ucDummy

Dummy input data. Any +ve character value.

MMC_RESETCOMMDIAGNOSTICS_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_RESETCOMMDIAGNOSTICS_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-66 describes the function block for MMC_ResetCommDiagnostics as applied within the IEC 61131 programming.

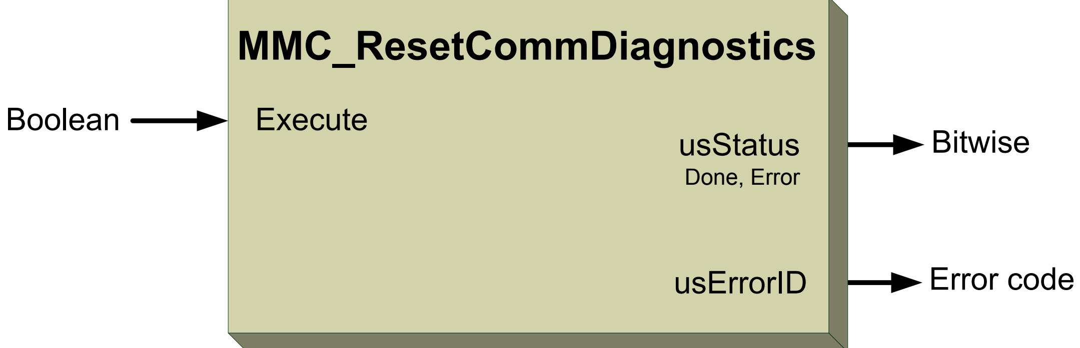


Figure 16-66: MMC_ResetCommDiagnostics function block

16.7.14.2 Function Block Code Example

```
int rc;
MMC_RESETCOMMDIAGNOSTICS_IN         stResetCOMMDiags_in;
MMC_RESETCOMMDIAGNOSTICS_OUT        stResetCOMMDiags_out;
//
// Inserting the structure parameters:
stResetCOMMDiags_in.ucDummy     = 1; //Dummy data input
//
rc = MMC_ResetCommDiagnostics (hConn, &stResetCOMMDiags_in, &stResetCOMMDiags_out);
if (rc != 0)
{
    HandleError();
}
```

16.7.15 MMC_ResetCommStatistics

Reset all communication statistics. Resets the communication error counters.

```
MMC_LIB_API int MMC_ResetCommStatistics(
IN MMC_CONNECT_HNDL hConn,
IN MMC_RESETCOMMSTATISTICS_IN* pInParam,
OUT MMC_RESETCOMMSTATISTICS_OUT* pOutParam
);
```

Motion Mode NC – Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoGlobal

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pInParam

Points to the **MMC_RESETCOMMSTATISTICS_IN** input data structure using the MMC_ResetCommStatistics function.

pOutParam

Points to the **MMC_RESETCOMMSTATISTICS_OUT** output structure receiving information as a result of calling the MMC_ResetCommStatistics function.

Remarks

None

Scope

All

MMC_RESETCOMMSTATISTICS_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_RESETCOMMSTATISTICS_IN;
```

Parameters

ucDummy

Dummy input data. Any +ve character value.

MMC_RESETCOMMSTATISTICS_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_RESETCOMMSTATISTICS_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Displays an error code as -ve or +ve integers.

Figure 16-67 describes the function block for MMC_ResetCommStatistics as applied within the IEC 61131 programming.

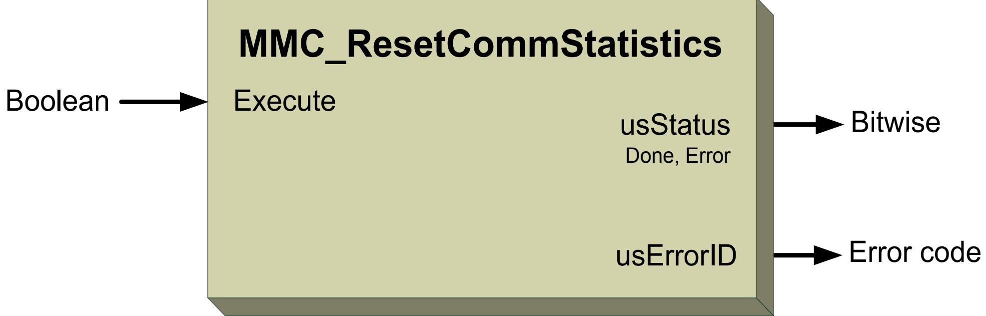


Figure 16-67: MMC_ResetCommStatistics function block

16.7.15.2 Function Block Code Example

```
int rc;
MMC_RESETCOMMSTATISTICS_IN      stResetCOMMStats_in;
MMC_RESETCOMMSTATISTICS_OUT     stResetCOMMStats_out;
//
// Inserting the structure parameters:
stResetCOMMStats_in.dummy  = 1; //Dummy data input
//
rc = MMC_ResetCommStatistics (hConn, &stResetCOMMStats_in, &stResetCOMMStats_out);
if (rc != 0)
{
    HandleError();
}
```

16.7.16 MMC_SendSDO

Sends SDO message command, in units of 1, 2, or 4 bytes.

```
MMC_LIB_API int MMC_SendSdoCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SENDSDO_IN* pInParam,
OUT MMC_SENDSDO_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SENDSDO_IN** input data structure using the MMC_SendSdo function.

pOutParam

Points to the **MMC_SENDSDO_OUT** output structure receiving information, as a result of calling the MMC_SendSdo function.

Remarks

The data length parameter MMC_SendSDO receives in bytes. However, if a user sends a request to upload/download an SDO object that is not 1, 2 or 4 bytes, an error is returned.

Scope

All

MMC_SENDSDO_IN Structure

```
typedef struct{
long lData;
unsigned long ulDataLength;
unsigned short usSlaveID;
unsigned short usIndex;
unsigned char ucSubIndex;
unsigned char ucService;
}MMC_SENDSDO_IN;
```

Parameters

lData

Data. Unlimited +ve or –ve values (long).

ulDataLength

Length of the data with 1, 2, or 4 values in bytes

usSlaveID

The slave ID. Any +ve integer value. This variable is redundant and no value should be entered.

usIndex

COB index. Any +ve integer values (2 bytes).

ucSubIndex

Defines which index value signifies the group of events to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping**. From events group 7 and above, the values represent the RPDO output. This parameter should mirror the enumerator value of the ucEventGroup variable, where applicable.

Any +ve character values.

ucService

Defines whether the input is a download or upload. Accepted integer values of:

1 - Download

2 - Upload

MMC_SENDSDO_OUT Structure

```
typedef struct{
long lData;
unsigned long ulDataLength;
unsigned short usStatus;
short sErrorID;
}MMC_SENDSDO_OUT;
```

Parameters

lData

Data. Unlimited +ve or –ve values (long).

ulDataLength

Length of the data. Unlimited +ve values in bytes.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-68 describes the function block for MMC_SendSdo as applied within the IEC 61131 programming.

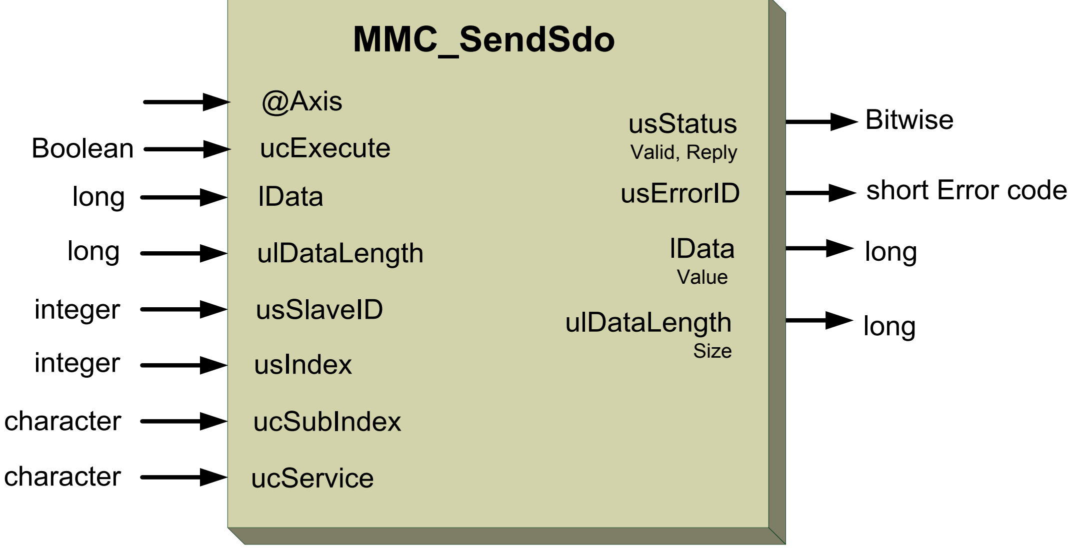


Figure 16-68: MMC_SendSdo function block

16.7.16.2 Function Block Code Example

```
int rc;
MMC_SENDSDO_IN        stSendSDO_in;
MMC_SENDSDO_OUT       stSendSDO_out;
//
// Inserting the structure parameters:
stSendSDO_in.lData          = 1234; //Data
stSendSDO_in.ulDataLength   = 2; //Length of the data
stSendSDO_in.usSlaveID      = 4; //The slave ID ---unused
stSendSDO_in.usIndex        = 0x6085; //COB index
stSendSDO_in.ucSubIndex     = 0;     //Defines which index value signifies the group of events
stSendSDO_in.ucService      = 1;     //Defines whether the input is a download or upload
//
rc = MMC_SendSdoCmd (hConn, iAxisRef, &stSendSDO_in, &stSendSDO_out);
if (rc != 0)
{
    HandleError();
}
```

16.7.17 MMC_SendSdoAsync

Sends SDO asynchronized message command, in units of 1, 2, or 4 bytes.

```
MMC_LIB_API int MMC_SendSdoAsyncCmd(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_SENDSDO_IN* pInParam,
OUT MMC_SENDSDO_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_SENDSDO_IN** input data structure using the MMC_SendSdoAsync function.

pOutParam

Points to the **MMC_SENDSDO_OUT** output structure receiving information, as a result of calling the MMC_SendSdoAsync function.

Remarks

The data length parameter MMC_SendSdoAsync receives in bytes. However, if a user sends a request to upload/download an SDO object that is not 1, 2 or 4 bytes, an error is returned.

Scope

All

MMC_SENDSDO_IN Structure

```
typedef struct{
long lData;
unsigned long ulDataLength;
unsigned short usSlaveID;
unsigned short usIndex;
unsigned char ucSubIndex;
unsigned char ucService;
}MMC_SENDSDO_IN;
```

Parameters

lData

Data. Unlimited +ve or –ve values (long).

ulDataLength

Length of the data with 1, 2, or 4 values in bytes

usSlaveID

The slave ID. Any +ve integer value. This variable is redundant and no value should be entered.

usIndex

COB index. Any +ve integer values (2 bytes).

ucSubIndex

Defines which index value signifies the group of events to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping**. From events group 7 and above, the values represent the RPDO output. This parameter should mirror the enumerator value of the ucEventGroup variable, where applicable.

Any +ve character values.

ucService

Defines whether the input is a download or upload. Accepted integer values of:

1 - Download

2 - Upload

MMC_SENDSDO_OUT Structure

```
typedef struct{
long lData;
unsigned long ulDataLength;
unsigned short usStatus;
short sErrorID;
}MMC_SENDSDO_OUT;
```

Parameters

lData

Data. Unlimited +ve or –ve values (long).

ulDataLength

Length of the data. Unlimited +ve values in bytes.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-69 describes the function block for MMC_SendSdoAsync as applied within the IEC 61131 programming.

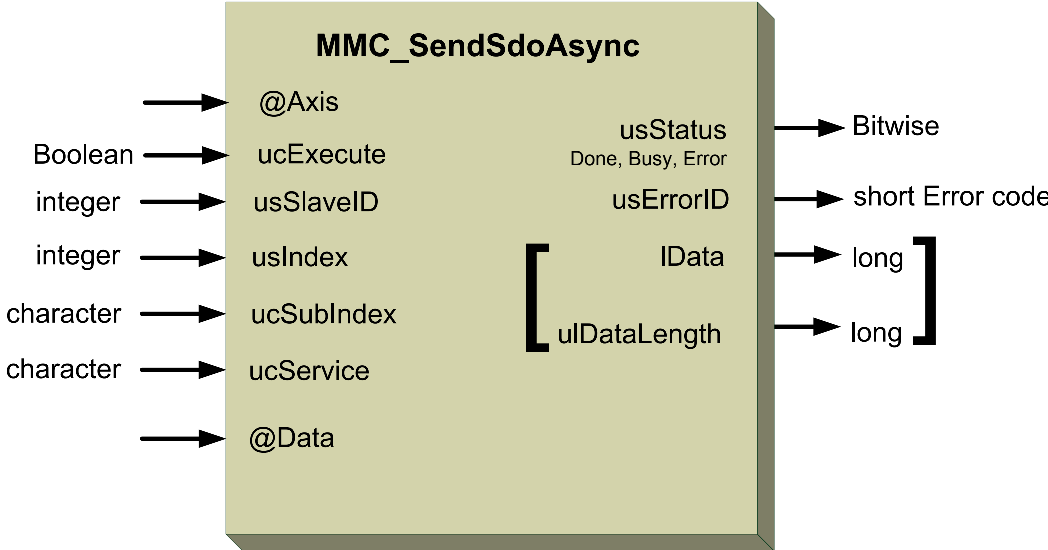


Figure 16-69: MMC_SendSdoAsync function block

16.8 Interpreter Command Functions

This section describes the functions (these are not function blocks) that are downloaded to the servo driver via Bin Interpreter or OS Interpreter mechanism. These functions may use RPC or IPC communication to perform the download. The purpose of these functions is to allow users of a command interpreter to access the servo driver and Maestro via direct commands, and to move the axis via a list of commands. However, the use of the command interpreter is limited to single axis and restricts the Maestro's capabilities.

The Get type functions perform in two ways:

- **Synchronously**
  The function does not return to the Maestro server until a response is received from the servo driver.
- **Asynchronously**
  The function returns immediately to the Maestro server, without waiting for a response from the servo driver. In the Maestro, when the response from the servo driver is received, it is sent via a UDP message to the library, where the connection listener thread processes the message.
  If the message is a Binary Interpreter Get command, it is processed by checking whether the Query operation mode is set to asynchronous query mode. If so, the uploaded data is kept per axis FIFO (the asynchronous query FIFO) and the axis asynchronous query FIFO index is incremented.
  However, if the Query operation mode is set to synchronous query mode, the response data is copied to the user out structure parameter.
  For every data stored in the FIFO, the FIFO index is incremented. The user can access the axis asynchronous query FIFO whenever, and retrieve data from the axis FIFO according to the axis FIFO index progress, whose value can be monitored via the *ElmoQueryOperationFIFOIndex* function.

The data returned from the Get functions, may be long or float. Therefore, storing the uploaded library data in the axes' FIFOs and enable retrieving of the data by the user is dependently performed on the specific data type saved.

Note: **For the asynchronous Get operation, a FIFO with size 1 is managed for each axis.**
If an error is sent from the servo driver because of sending an Interpreter command, the queue will be emptied and an error will be sent to the library.

16.8.1 Get Function – Asynchronous Mode

Can operate in sync and async modes. To operate in this mode, the user sets the Query operation mode to asynchronous mode,.

When a UDP message arrives to the connection UDP listener thread, the thread checks that it is a *SendRawData* – Get response, and if so, will place the data as a specific entry in a dedicated FIFO of the designated axis.

When it reaches the last entry at the FIFO, the index will not be incremented, until the user will reset the index via the ElmoQueryOperationFIFOIndexReset() function. It is responsibility of the user, to reset the Query Operation FIFO at the right time, to prevent the last record overrunning in the FIFO. The user can retrieve data from the axis Query Operation FIFO at location index in any time, via the ElmoQueryOperationFIFORetrieveData(index) function.

In order to perform Get operation in asynchronous mode, for example: et[1] for 4 axes, the following should be performed:

- ElmoGetArray(axisref1, "et", 1) to axis 1.
- ElmoGetArray(axisref2, "et", 1) to axis 2.
- ElmoGetArray(axisref3, "et", 1) to axis 3.
- ElmoGetArray(axisref4, "et", 1) to axis 4.
- ElmoQueryOperationFIFOIndex(axisref) - to return to the current index buffer location which occupies the asynchronous returned replies for a specific axis. When the current index location for one of the 1-4 axes is 1, then for that axis the asynchronous reply was received and we can start retrieving it via the, ElmoQueryOperationFIFORetrieveData(axisref, index) function.
- ElmoQueryOperationFIFORetrieveData(axisref1, 1) – Get asynchronous reply of axis 1 which located at the first entry in the axis FIFO.

The following Interpreter Command functions are described:

Interpreter Command
MMC_ElmoExecuteLabel
MMC_ElmoSetParameter
MMC_ElmoGetParameter
MMC_ElmoGetArray
MMC_ElmoGetArrayAndRetrieveData
MMC_ElmoGetParameterAndRetrieveData
MMC_ElmoSetArray
MMC_ElmoQueryOperationFIFOIndex
MMC_ElmoQueryOperationFIFORetrieveData
MMC_ElmoQueryOperationFIFOIndexReset
MMC_ElmoCall

16.8.2 MMC_ElmoExecuteLabel

Executes the user program downloaded via the EASII application.

```
MC_LIB_API int MMC_ElmoExecuteLabel(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_INTERPEXECUTECMD_IN* pInParam,
OUT MMC_INTERPEXECUTECMD_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_INTERPEXECUTECMD_IN** input data structure using the MMC_ElmoExecuteLabel function.

pOutParam

Points to the **MMC_INTERPEXECUTECMD_OUT** output structure receiving information, as a result of calling the MMC_ElmoExecuteLabel function.

Remarks

These functions may communicate via an RPC or IPC connection.

Scope

Note: When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).

MMC_INTERPEXECUTECMD_IN Structure

```
typedef struct{
unsigned char ucLength;
unsigned char pData[NODE_ASCII_ARRAY_MAX_LENGTH];
}MMC_INTERPEXECUTECMD_IN;
```

Parameters

ucLength

Length of the label string. Length with the precursor format of [Metronome command]##[label]. Any +ve character values.

pData

String Data with the precursor format of [Metronome command]##[label]. Any +ve character values with a maximum length of 80 bytes

[NODE_ASCII_ARRAY_MAX_LENGTH] is the node ASCII array integers with a maximum length of 80 bytes.

MMC_INTERPEXECUTECMD_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_INTERPEXECUTECMD_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-70 describes the function for MMC_ElmoExecuteLabel as applied within the IEC 61131 programming for MC_ElmoExecute.

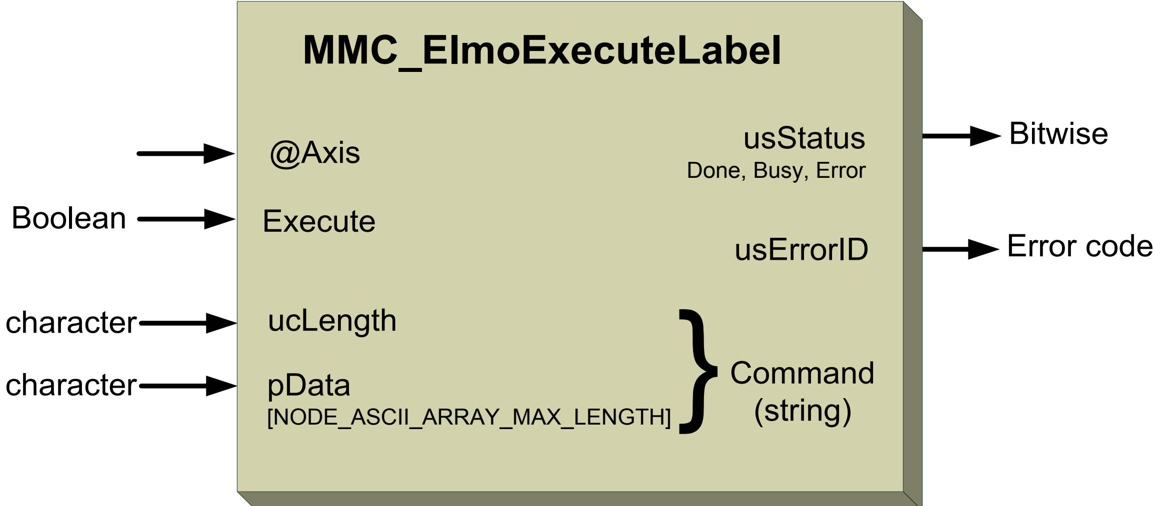


Figure 16-70: MMC_ElmoExecuteLabel function

16.8.2.2 Function Code Example

```
int rc;
MMC_INTERPEXECUTECMD_IN    stInterpExCmd_in;
MMC_INTERPEXECUTECMD_OUT   stInterpExCmd_out;
//
// Inserting the structure parameters:

stInterpExCmd_in.ucLength = sizeof ("XQ##start"); //Length of the data
strcpy((char*) stInterpExCmd_in.pData,"XQ##start"); //Data

//
rc = MMC_ElmoExecuteLabel (hConn, iAxisRef, &stInterpExCmd_in, &stInterpExCmd_out);
if (rc != 0)
{
    HandleError();
}
```

16.8.3 MMC_ElmoSetParameter

Sets the Elmo drive parameter with a specific name in the servo drive.

```
MMC_LIB_API int MMC_ElmoSetParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN char cCmd[3],
IN unsigned char ucValType,
IN void* pVal
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

ucValType

Data value type, whether integer or float. Integer or float with values of 0 or 1.

pVal

Pointer to data that is to be set. Point value of a maximum of 4 bytes (Void)

Remarks

These functions may communicate via an RPC or IPC connection.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

Figure 16-73 and Figure 16-72 describe the function block for MMC_ElmoSetParam as applied within the IEC 61131 programming for ElmoSetFloatParam and ElmoSetIntParam. The MMC_ElmoSetXXXXParam parameters differ in C language to the IEC.

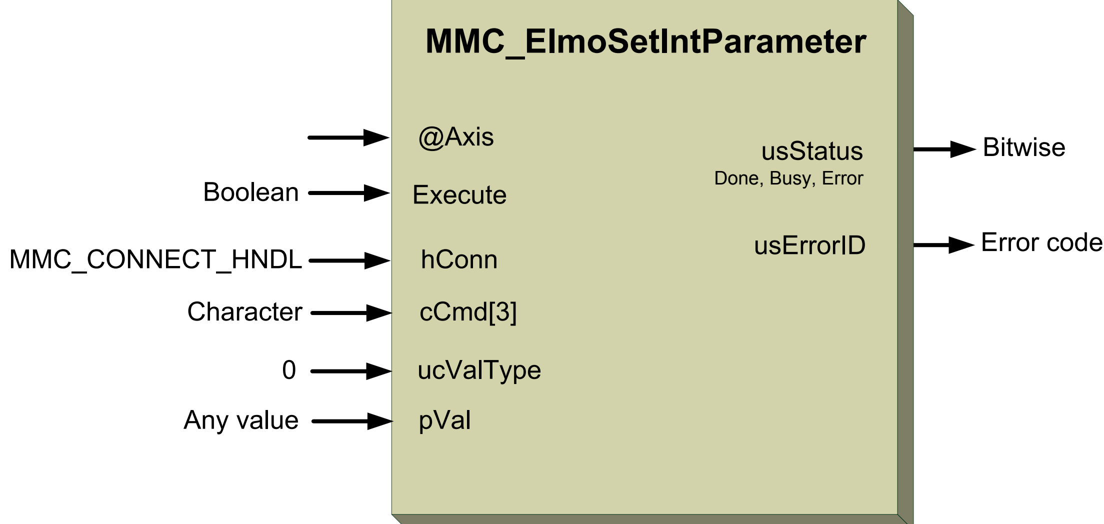


Figure 16-71: MMC_ElmoSetIntParam function block

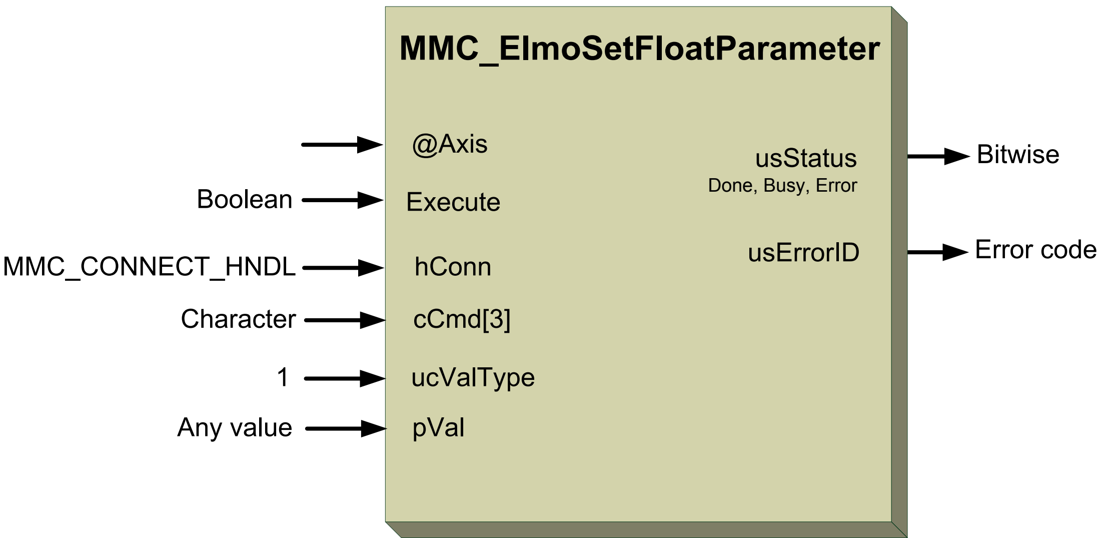


Figure 16-72: MMC_ElmoSetFloatParam function block

16.8.3.1 Function Code Example

```
int rc;
//
// Inserting the structure parameters:
strcpy(cCmd,"MO");      //Name of the parameter
iArrayIdx     = 1;      //Index of array element
ucValType     = 0;      //Data value type
strcpy(pVal, "1");      //Pointer to data that is to be set
//
rc = MMC_ElmoSetParameter (hConn, iAxisRef, cCmd, iArrayIdx, ucValType, pVal);
if (rc != 0)
{
    HandleError();
}
```

16.8.4 MMC_ElmoGetParameter

Request to receive the Elmo parameters from the servo drive.

```
MMC_LIB_API int MMC_ElmoGetParameter(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN char cCmd[3],
OUT unsigned char ucValType
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

ucValType

Data value type, whether integer or float. Integer or float with values of 0 or 1.

Remarks

These functions may communicate via an RPC or IPC connection.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

Figure 16-75 and Figure 16-74 describes the function block for MMC_ElmoGetParameter as applied within the IEC 61131 programming for ElmoGetFloatParam and ElmoGetIntParam. The MMC_ElmoGetXXXXParam parameters differ in C language to the IEC. The C version waits for the sync MMC_GetParameterAndRetrieveData to produce the pVal output. However the IEC version automatically stores the pVal output.

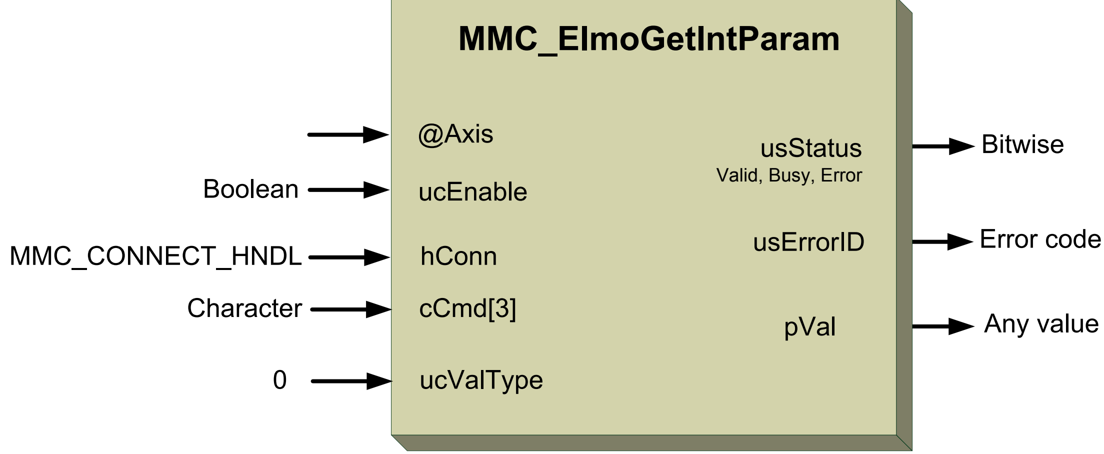


Figure 16-73: MMC_ElmoGetIntParam function block

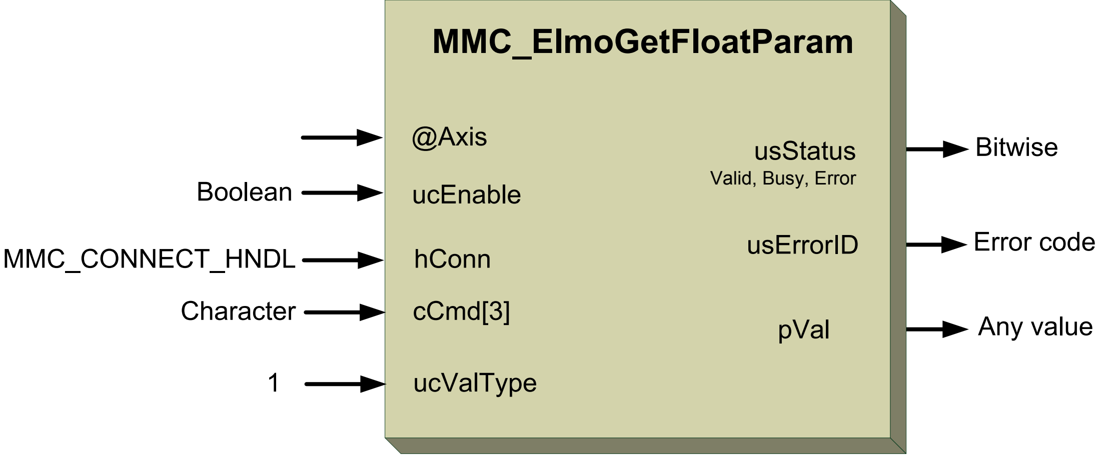


Figure 16-74: MMC_ElmoGetFloatParam function block

16.8.4.1 Function Code Example

```
int rc;
//
// Inserting the structure parameters:
strcpy(cCmd,"MO");      //Name of the parameter
ucValType     = 0;      //Data value type
//
rc = MMC_ElmoGetParameter (hConn, iAxisRef, cCmd, ucValType);
if (rc != 0)
{
    HandleError();
}
```

16.8.5 MMC_ElmoGetArray

Request to receive an element from the asynchronous array of parameters in the servo drive.

```
MMC_LIB_API int MMC_ElmoGetArray(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN char cCmd[3],
IN short iArrayIdx,
IN unsigned char ucValType
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

cCmd[3]

[IN] Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iArrayIdx

[IN] Index of array element. Any +ve or –ve short integer value with a maximum of 2 bytes.

ucValType

[OUT] Data value type, whether integer or float. Integer or float with values of 0 or 1.

Remarks

These functions may communicate via an RPC or IPC connection.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

Figure 16-75 and Figure 16-76 describe the function block for MMC_ElmoGetArray as applied within the IEC 61131 programming for ElmoGetFloatArr and ElmoGetIntArr. The MMC_ElmoGetXXXXArray parameters differ in C language to the IEC. The C version waits for the sync MMC_GetArrayAndRetrieveData to produce the pVal output. However the IEC version automatically stores the pVal output.

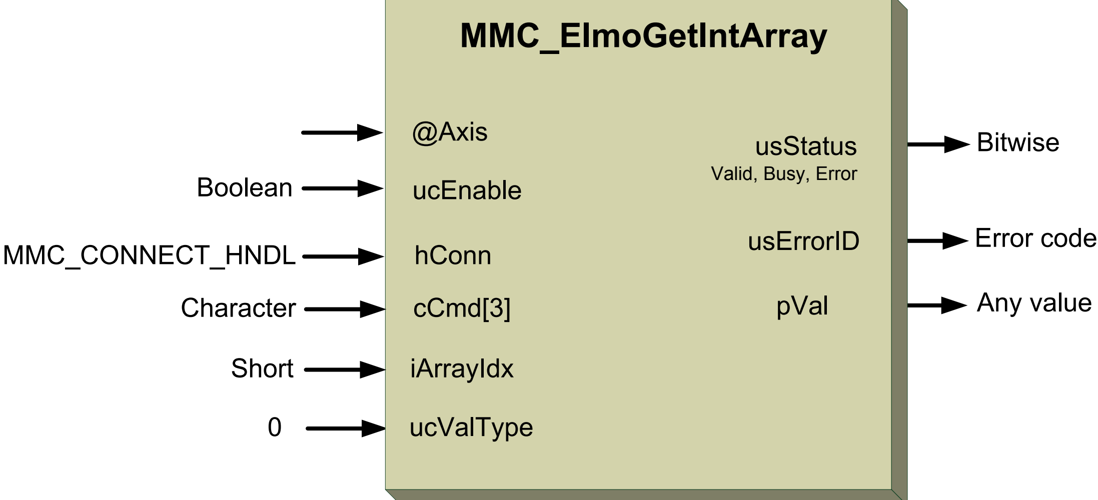


Figure 16-75: MMC_ElmoGetIntArray function block

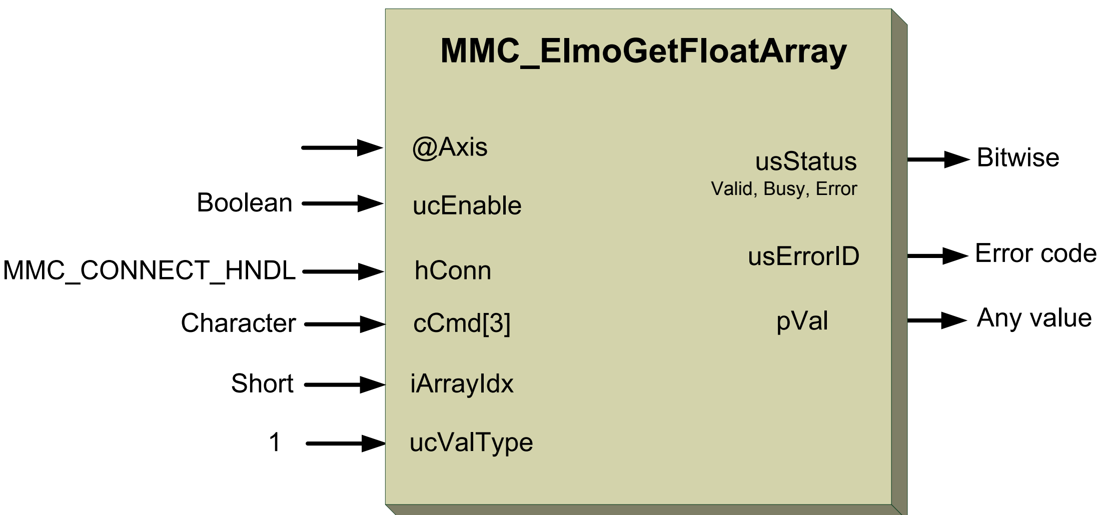


Figure 16-76: MMC_ElmoGetFloatArray function block

16.8.5.1 Function Code Example

```
int rc;
//
// Inserting the structure parameters:
strcpy(cCmd,"MO");          //Name of the parameter
iArrayIdx     = 1;          //Index of array element
ucValType     = 0;          //Data value type
//
rc = MMC_ElmoGetArray (hConn, iAxisRef, cCmd, iArrayIdx, ucValType);
if (rc != 0)
{
    HandleError();
}
```

16.8.6 MMC_ElmoGetArrayAndRetrieveData

Synchronously requests an element from the array parameters in the servo drive and retrieves it.

```
MMC_LIB_API int MMC_ElmoGetArrayAndRetrieveData(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN char cCmd[3],
IN short iArrayIdx,
IN unsigned char ucValType,
OUT void *pVal,
OUT unsigned int* uiErrorID
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

[IN] Axis/group reference handle type returned by GetAxisRef command

cCmd[3]

[IN] Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iArrayIdx

[IN] Index of array element. Any +ve or –ve short integer value with a maximum of 2 bytes.

ucValType

[IN] Data value type, whether integer or float. Integer or float with values of 0 or 1.

pVal

[OUT] Copies a point from where to retrieve the data. Point value of a maximum of 4 bytes (Void)

uiErrorID

[OUT] Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Function init with values of 0 or +ve error_id.

Remarks

This command will necessitate waiting for the data element to be retrieved or an error returned. No other process command may be sent meanwhile.

Scope

For synchronous communication only

16.8.6.1 Function Code Example

```
int rc;
//
// Inserting the structure parameters:
strcpy(cCmd,"MO");     //Name of the parameter
iArrayIdx     = 1;     //Index of array element
ucValType     = 0;     //Data value type
//
rc = MMC_ElmoGetArrayAndRetrieveData (hConn, iAxisRef, cCmd, iArrayIdx, ucValType, pVal,
uiErrorID);
printf("Elmo Get Array and Retrieve Data Status[%ld] ErrId[%d]\n", (long int)pVal,
(int)uiErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.8.7 MMC_ElmoGetParameterAndRetrieveData

Synchronously requests a parameter in the servo drive and retrieves it.

```
MMC_LIB_API int MMC_ElmoGetParameterAndRetrieveData(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN char cCmd[3],
IN unsigned char ucValType,
OUT void *pVal,
OUT unsigned int* uiErrorID);
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

[IN] Axis/group reference handle type returned by GetAxisRef command

cCmd[3]

[IN] Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

ucValType

[IN] Data value type, whether integer or float. Integer or float with values of 0 or 1.

pVal

[OUT] Copies a point from where to retrieve the data. Point value of a maximum of 4 bytes (Void)

uiErrorID

[OUT] Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**. Function init with values of 0 or +ve error_id.

Remarks

This command will necessitate waiting for the data parameter to be retrieved or an error returned. No other process command may be sent meanwhile.

Scope

For synchronous communication only

16.8.7.1 Function Code Example

```
int rc;
//
// Inserting the structure parameters:
strcpy(cCmd,"MO");     //Name of the parameter
ucValType     = 0;     //Data value type
//
rc = MMC_ElmoGetParameterAndRetrieveData (hConn, iAxisRef, cCmd, ucValType, pVal, uiErrorID);
printf("Elmo Get Parameter and Retrieve Data Status[%ld] ErrId[%d]\n", (long int)pVal,
(int)uiErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.8.8 MMC_ElmoSetArray

Sets an element of the array parameters in the servo drive.

```
MMC_LIB_API int MMC_ElmoSetArray(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN char cCmd[3],
IN short iArrayIdx,
IN unsigned char ucValType,
IN void* pVal
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

[IN] Axis/group reference handle type returned by GetAxisRef command

cCmd[3]

[IN] Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iArrayIdx

[IN] Index of array element. Any +ve or –ve short integer value with a maximum of 2 bytes.

ucValType

[IN] Data value type, whether integer or float. Integer or float with values of 0 or 1.

pVal

[IN] Pointer to data that is to be set. Point value of a maximum of 4 bytes (Void)

Remarks

This command will necessitate waiting for the data element to be retrieved or an error returned. No other process command may be sent meanwhile.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

Figure 16-77 and Figure 16-78 describe the function block for MMC_ElmoSetArray as applied within the IEC 61131 programming for ElmoSetFloatArr and ElmoSetIntArr. The MMC_ElmoSetXXXXArray parameters differ in C language to the IEC.

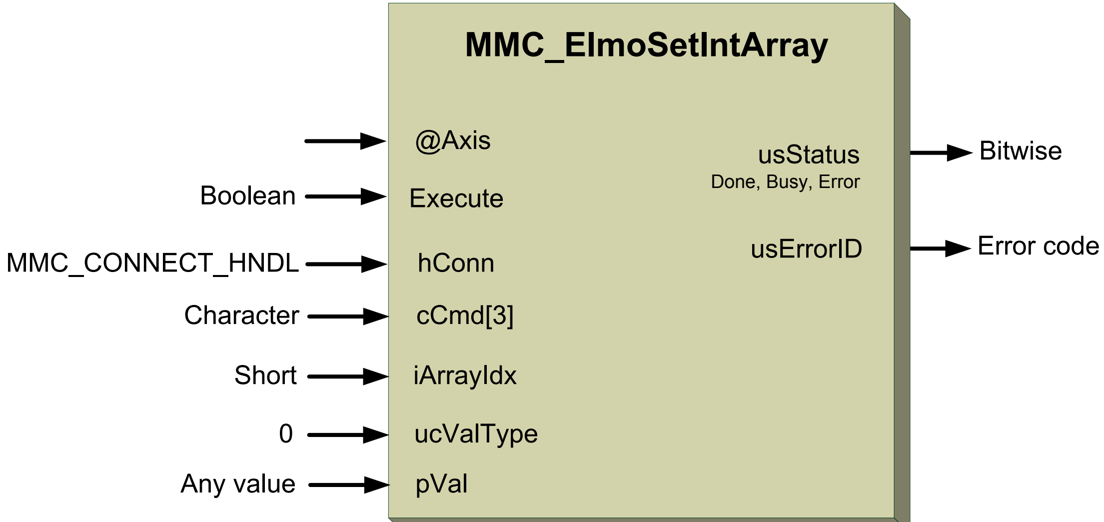


Figure 16-77: MMC_ElmoSetIntArray function block

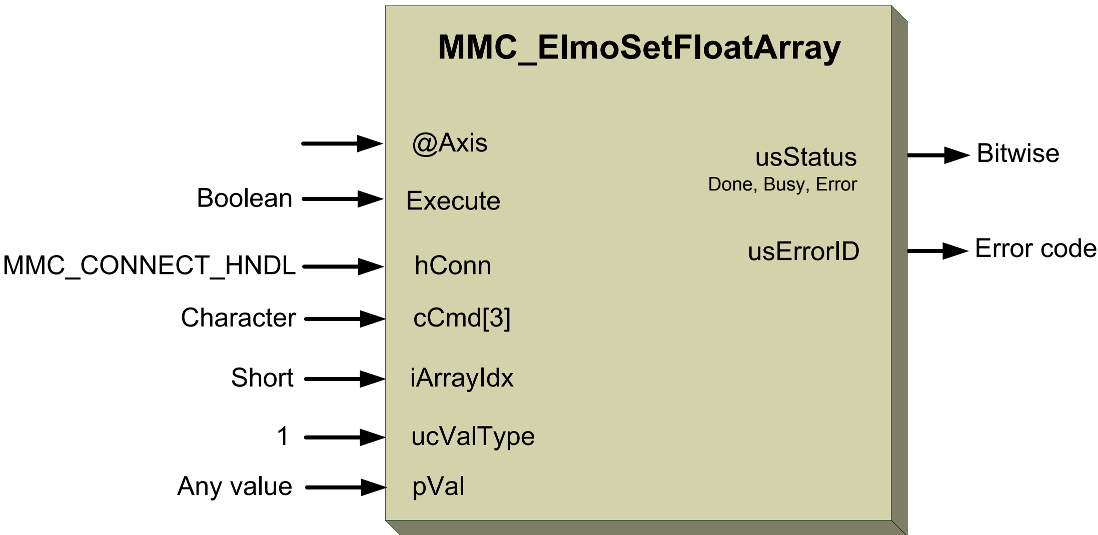


Figure 16-78: MMC_ElmoSetFloatArray function block

16.8.8.1 Function Code Example

```
int rc;
//
// Inserting the structure parameters:
strcpy(cCmd,"IL");      //Name of the parameter
iArrayIdx     = 1;      //Index of array element
ucValType     = 0;      //Data value type
strcpy(pVal, "1");      //Pointer to data that is to be set
//
rc = MMC_ElmoSetArray (hConn, iAxisRef, cCmd, iArrayIdx, ucValType, pVal);
if (rc != 0)
{
    HandleError();
}
```

16.8.9 MMC_ElmoQueryOperationFIFOIndex

Returns the FIFO index.

MMC_LIB_API int MMC_ElmoQueryOperationFIFOIndex(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
OUT int* iReceivedMsgIdx
);

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

[IN] Axis/group reference handle type returned by GetAxisRef command

iReceivedMsgIdx

[OUT] Index of the number of received messages. Function init with values 0 or error_id.

Remarks

None

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

16.8.9.1 Function Code Example

```
int rc;
//
rc = MMC_ElmoQueryOperationFIFOIndex (hConn, iAxisRef, iReceivedMsgIdx);
printf("Elmo Query Operation FIFO Index Status[%ld]\n", (long int)iReceivedMsgIdx);
if (rc != 0)
{
    HandleError();
}
```

16.8.10 MMC_ElmoQueryOperationFIFORetrieveData

Request the FIFO index to retrieve data.

MMC_LIB_API MMC_LIB_API int MMC_ElmoQueryOperationFIFORetrieveData(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
OUT void *pVal,
OUT unsigned int* uiErrorID
);

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

[IN] Axis/group reference handle type returned by GetAxisRef command

pVal

[OUT] Pointer to data that is to be set. Point value of a maximum of 4 bytes (Void).

uiErrorID

[OUT] Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**. Function init with values of 0 or +ve error_id.

Remarks

If the FIFO index is 0, either an error or no data is received. If 1, data is received.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

16.8.10.1 Function Code Example

```
int rc;
//
rc = MMC_ElmoQueryOperationFIFORetrieveData (hConn, iAxisRef, pVal, uiErrorID);
printf("Elmo Query Operation FIFO Retrieve Data Status[%ld] ErrId[%d]\n", (long int)pVal,
(int)uiErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.8.11 MMC_ElmoQueryOperationFIFOIndexReset

Erases the message FIFO to 0.

MMC_LIB_API int MMC_ElmoQueryOperationFIFOIndexReset(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef
);

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_drive_comm_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

[IN] Axis/group reference handle type returned by GetAxisRef command

Remarks

None

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

16.8.11.1 Function Code Example

```
void MMC_ElmoQueryOperationFIFOIndexReset_wrapper(int iAxisRef)
{
int rc;
//
rc = MMC_ElmoQueryOperationFIFOIndexReset (hConn, iAxisRef);
if (rc != 0)
{
    HandleError();
}
}
```

16.8.12 MMC_ElmoCall

ElmoCall is used to call a subroutine, a user program, where cCmd[3] is the name of the program

```
MMC_LIB_API int MMC_ElmoCall(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN char cCmd[3]
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_drive_comm_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

[IN] Axis/group reference handle type returned by GetAxisRef command

cCmd[3]

[IN] Name of the program limited to three characters. Any +ve character value with a maximum of 2 bytes

Remarks

Some parameters are not true variables but are direct commands to operate the servo drive. ElmoCall uses these commands to call a specific parameter from the servo drive.

Scope

All

Figure 16-79 describes the function block for MMC_ElmoCall as applied within the IEC 61131 programming.

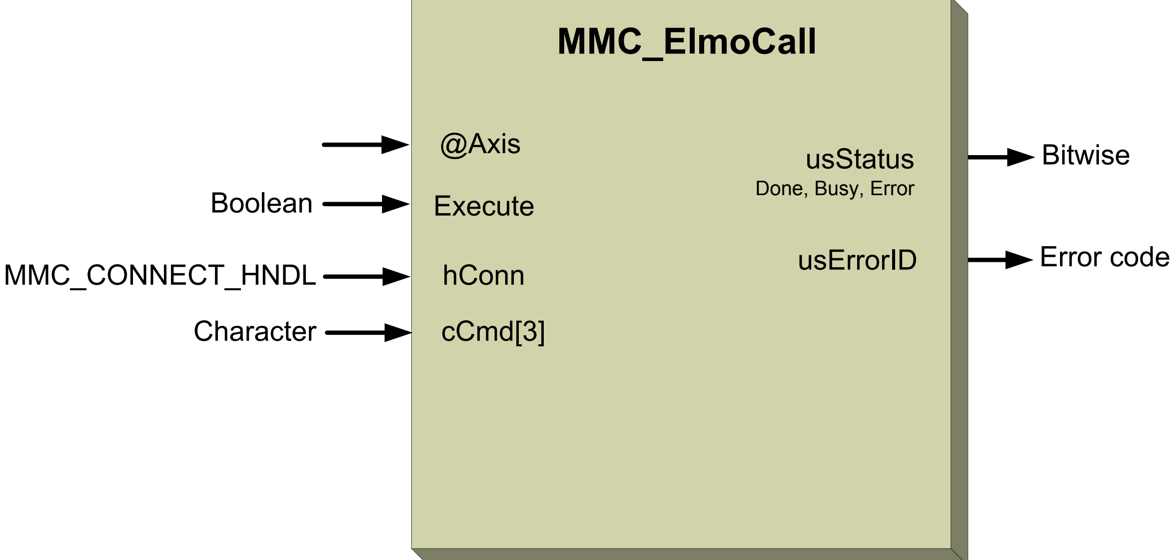


Figure 16-79: MMC_ElmoCall function block

16.8.12.1 Function Block Code Example

```
int rc;
//
// Inserting the structure parameters:
strcpy(cCmd,"BG");     //Name of the parameter
//
rc = MMC_ElmoCall (hConn, iAxisRef, cCmd);
if (rc != 0)
{
    HandleError();
}
```

16.9 EtherNetIP Communication

This section describes the EtherNet/IP (Ethernet Industrial Protocol), which is a communications protocol developed by Rockwell Automation, managed by ODVA and designed for use in process control and other industrial automation applications e.g. water processing plants, manufacturing facilities, and other utilities. It can be used in automation networks, which tolerate some amount of non-determinism. This is because the Ethernet physical media, by definition, is not deterministic. EtherNet/IP is most commonly used in the US and Asia for communication to and from Rockwell Automation's Allen-Bradley-control systems.

ODVA manages the protocol and assures multi-vendor system interoperability. The transfer of basic I/O data is via the UDP-based implicit messaging (port 2222). Uploading and downloading of parameters via TCP (i.e. explicit messaging port 44818). The EtherNet/IP application layer protocol is based on the Common Industrial Protocol (CIP) standard used in DeviceNet. Elmo uses the PyramidSolutions EIP adapter stack (EADK) used in the Maestro. It was compiled for Elmo’s Linux toolchain.

16.9.1 Terminology

The devices are:

Devices	Explanation
Master (Scanner)	Initiates communications with adapter devices. A scanner is typically the most complex type of EtherNet/IP device, as it must manage issues such as configuration, connections, of the adapter device. A Rockwell PLC is an example of a scanner.
Slave (Adapter)	Receives communication connection requests from a scanner and then produces its I/O data at the requested rate. An adapter can be a simple digital input device or more complex such as a modular pneumatic valve system, the Maestro.

The message types involved in EtherNetIP are:

Messages	Explanation
Explicit messaging (information)	Non time-critical data transfers that are a relatively large packet size for short-lived connections between originator and single target device. The data is transferred via TCP/IP.
Implicit messaging (I/O data)	Time-critical data transfers usually consist of small packets. I/O data exchanges are long term lived connections between single originator to one or more target devices. I/O data packets are UDP packets.

16.9.1.1 Assemblies

An Ethernet /IP object is used for I/O message communication. There are three types of Assemblies, which use implicit messaging:

Assembly	Explanation
Production	produce (sends) data
Consumption	consumes (receives) data
Configuration	general configuration

16.9.1.2 Tags

A Tag defines the name for a single or array of variable(s). there are two types of Tags:

TAGs	Explanation
Adapter	Data that resides in the Maestro and is available for other devices on the Ethernet/IP.
Device	Data that resides on other devices on the Ethernet/IP, e.g. PLC.

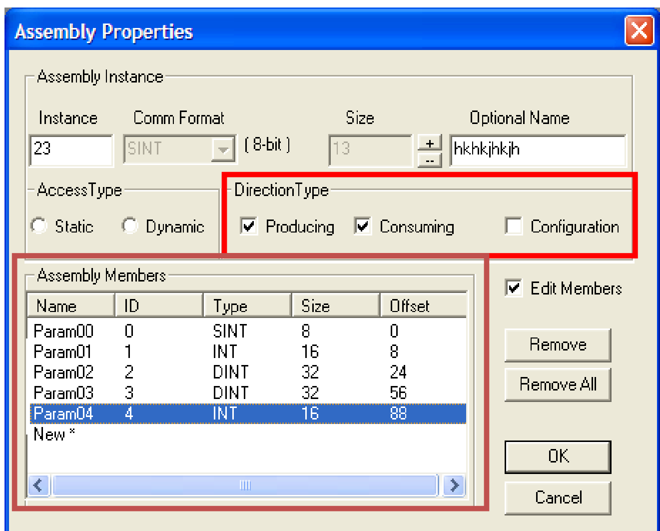
16.9.1.3 Data Types

The Following data Types are supported (Assemblies and Tags):

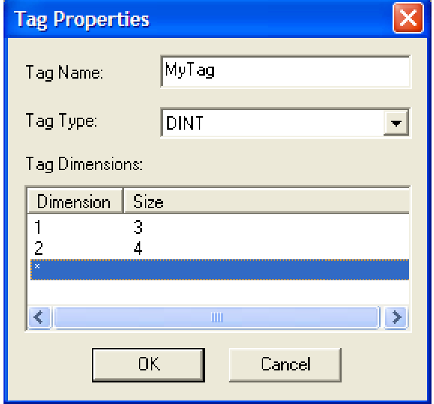
Data Type	Explanation
BOOL(Tag only)	unsigned char (1 byte)
SINT	signed char (1 byte)
INT	short int (2 bytes)
DINT	int (4 bytes)
REAL	float (4 bytes)

16.9.2 Configuring the Ethernet IP Device as Adapter

Identical Configurations must reside on the Master and Slave (Maestro) side. The assembly parameters should be of one to three directional types: Consumer, Producer or Configuration, operating in their specific directions. The opposing direction must reside on the Master side. The assemblies are identified by the InstanceID, and have member data. The member data is exchanged every I/O data exchange (initiated by scanner). The Assembly definition is created in the Properties window as shown in the Maestro assembly configuration window.



Tags (Slave (Adapter) or Device) may be defined as arrays or single dimension, and are sent as soon as they are modified. Tags have names, which are defined in the XML Configuration File.



This XML configuration file contains all of the Ethernet/IP data, e.g. Paths, Tags and Assembly data, and is similar to the structure of the Maestro XML File.

```
<?xml version="1.0" ?>
- <EIPADAPTER>
  - <EIPDEVICES>
    - <EIPDEVICE NAME="ggg" NETWORKPATH="192.168.1.1">
        <EIPTAG TYPE="REAL" NAME="jjjjj" SIZE="4,6" />
      </EIPDEVICE>
    </EIPDEVICES>
  - <EIPASSEMBLIES>
    - <EIPASSEMBLY NAME="outAss" INSTANCE="1" SIZE="4" OFFSET="4294967295" COMFORMAT="DINT" TYPE="8">
        <EIPASSEMBLY_MEMBER ID="0" SIZE="32" OFFSET="0" COMFORMAT="DINT" NAME="Param00" />
        <EIPASSEMBLY_MEMBER ID="1" SIZE="32" OFFSET="32" COMFORMAT="DINT" NAME="Param01" />
        <EIPASSEMBLY_MEMBER ID="2" SIZE="32" OFFSET="64" COMFORMAT="DINT" NAME="Param02" />
        <EIPASSEMBLY_MEMBER ID="3" SIZE="32" OFFSET="96" COMFORMAT="DINT" NAME="Param03" />
      </EIPASSEMBLY>
    - <EIPASSEMBLY NAME="inAss" INSTANCE="2" SIZE="4" OFFSET="4294967295" COMFORMAT="DINT" TYPE="4">
        <EIPASSEMBLY_MEMBER ID="0" SIZE="32" OFFSET="0" COMFORMAT="DINT" NAME="Param00" />
        <EIPASSEMBLY_MEMBER ID="0" SIZE="32" OFFSET="0" COMFORMAT="DINT" NAME="Param00" />
        <EIPASSEMBLY_MEMBER ID="0" SIZE="32" OFFSET="0" COMFORMAT="DINT" NAME="Param00" />
        <EIPASSEMBLY_MEMBER ID="0" SIZE="32" OFFSET="0" COMFORMAT="DINT" NAME="Param00" />
      </EIPASSEMBLY>
    </EIPASSEMBLIES>
  - <EIPADAPTERTAGS>
      <EIPADAPTERTAG TYPE="INT" NAME="KUKU111" SIZE="2,3" />
      <EIPADAPTERTAG TYPE="BOOL" NAME="dsfsdf" SIZE="1" />
    </EIPADAPTERTAGS>
  </EIPADAPTER>
```


Figure 16-80 describes the Maestro operation block diagram using EtherNetIP.

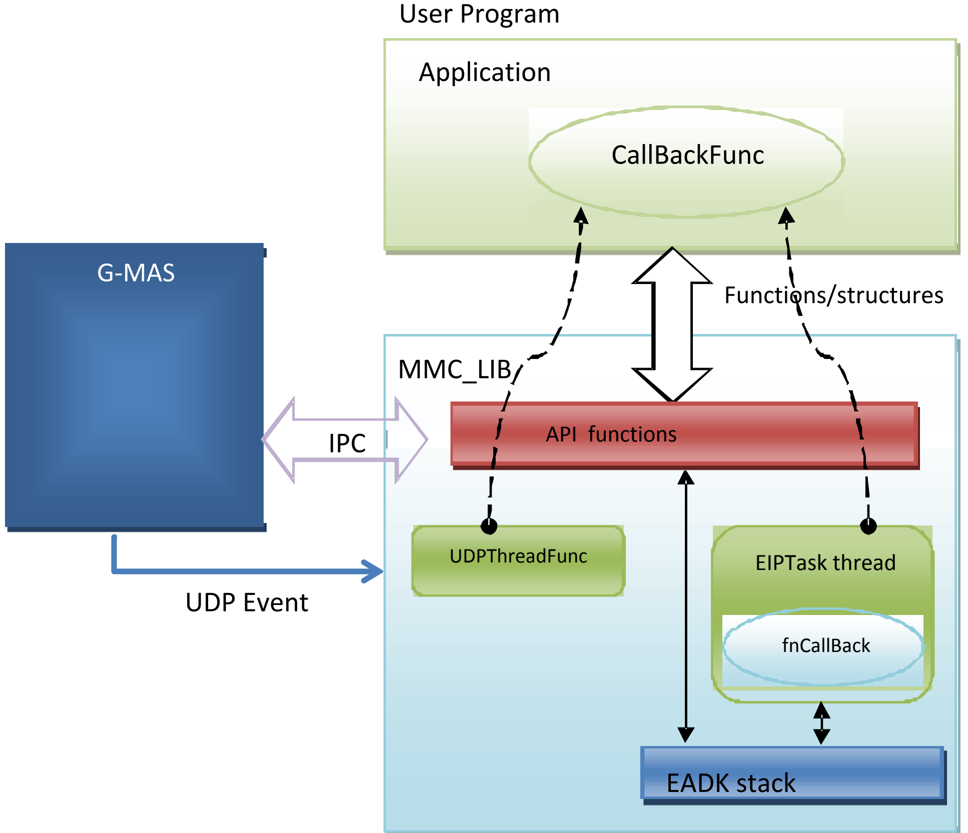


Figure 16-80: Maestro Operation block diagram

The EADK is part of the EIP_LIB (Maestro Library), with the following advantages:

- Less Overhead when reading/writing data. Faster.
- Less events, context switching, etc …
- Easy debugging.

As such, the Assembly Tags reside both in the Maestro and other device memory according to the following:

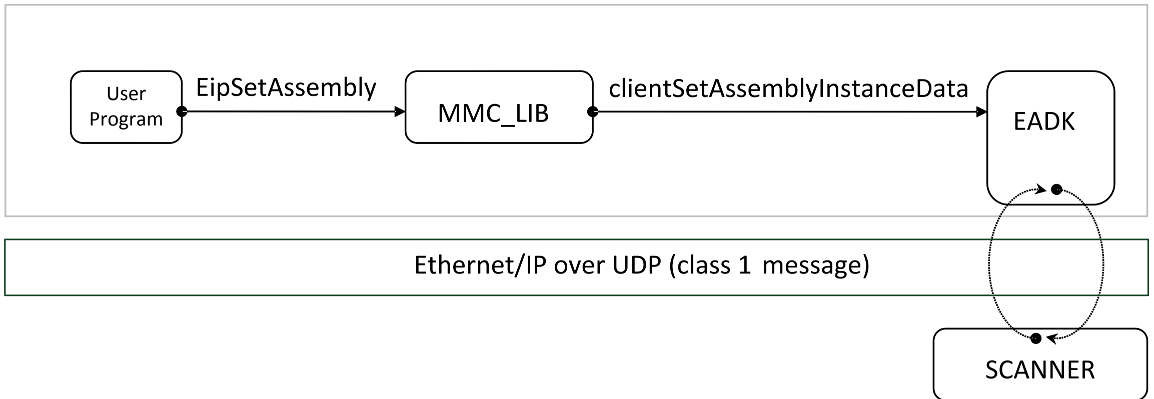
Tags	Explanation
Device	Reside at the other device memory
Adapter	Reside at the Maestro memory

When Initialized, the MMC_EipInit performs the following:

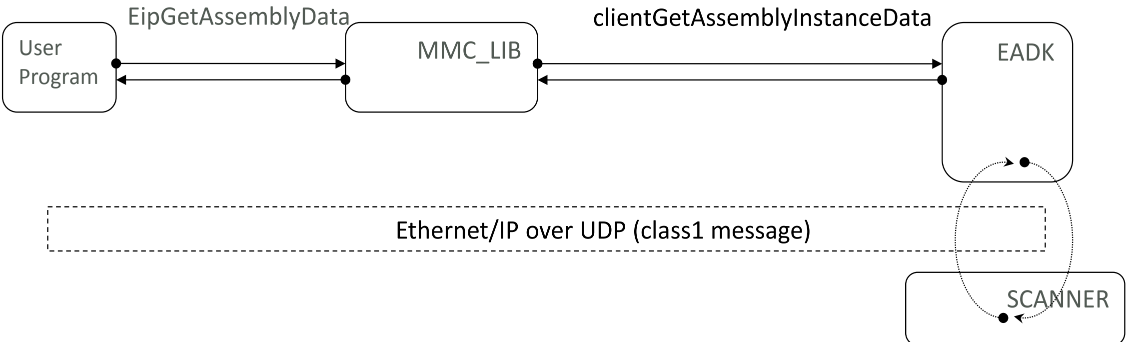
1. Calls relevant EIP stack APIs that:
    a. Initialize callbacks
    b. Creates listening thread
2. Initializes assemblies and Tags by parsing XML.
3. Sets assemblies / member structures in the stack.

16.9.2.1 Assembly Setting Data

The user calls MMC_EipSetAssembly() – with the input tag reference associated with the producing assembly instance, and then sends the assembly data structure. If an error occurs – an event is sent. The message is sent at the next scanner cycle.

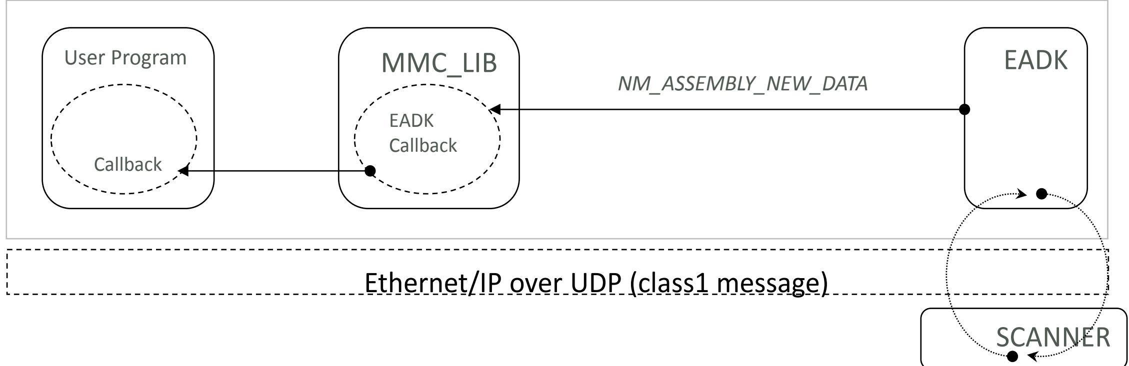


Then user then calls MMC_EipGetAssembly() – with input tag reference associated with the consuming assembly instance. The function returns the data to a local data structure.



16.9.2.2 New Assembly Received Event

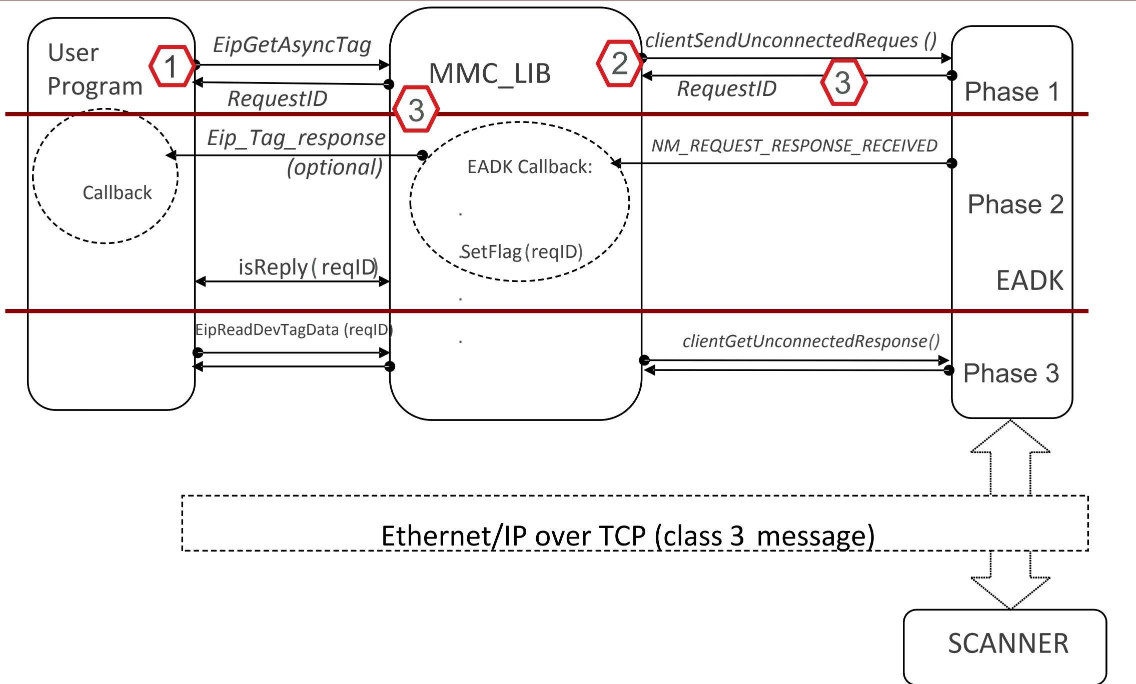
The EADK invokes a callback when new data (modified) is received. This callback calls the same default callback as in the User program. User program can then read the assembly data (Instead of polling).



The device Tags are set by calling the EipSetDevTag function (an asynchronous call). If no immediate error occurs, then the message is sent. If an error occurs while sending the message – an event is invoked and the user is notified.

16.9.2.2.1 Modes

Mode	Explanation
Async.	Does not wait for the EipReadTag. The EipGetDevTag function sends a request to the EIP device to read specific device tag data but not in the immediate sequence. EipReadDevTagData reads the data when it arrives.
Sync.	Waits for data to arrive and employs the EipGetAsyncTag function to read the EipReadDevTagData in phases 1 to 3 in sequence



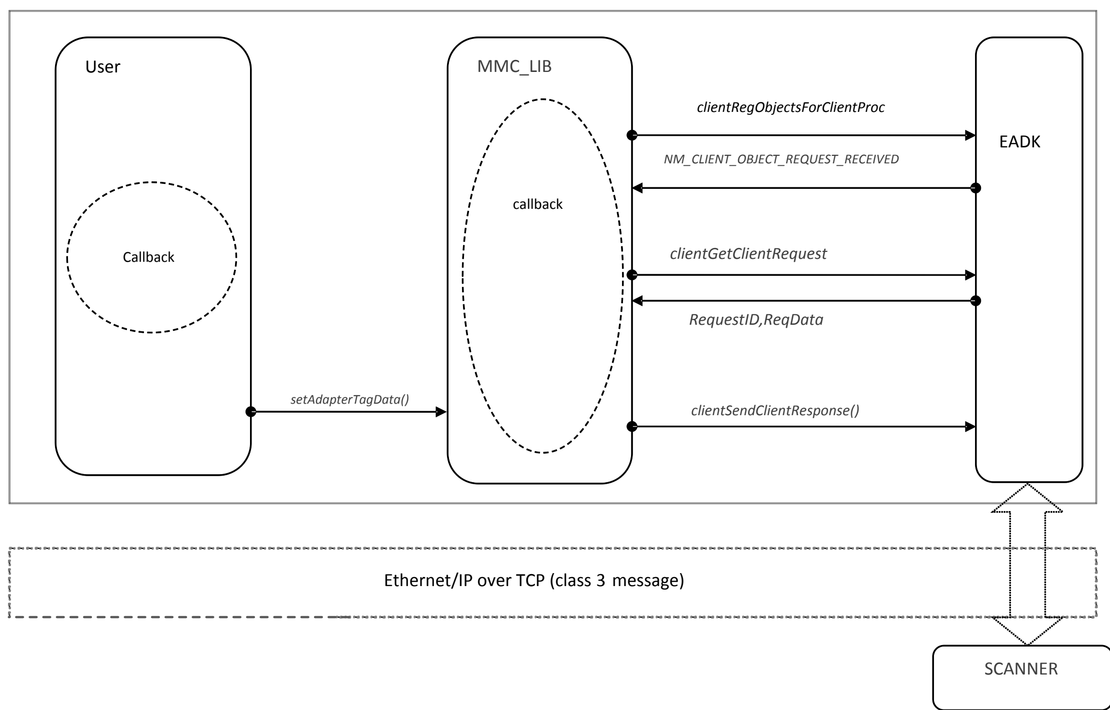
The User calls the EipGetDevTag function (asynchronous). If no immediate error occurs, then the message is sent and a ReqID is returned (Max 100 concurrent messages). Once the stack replies, a flag is set that the message arrived. User may poll or receive an event that a reply has arrived, but is not pending.

User calls the EIPReadDevTagData(ReqID) to retrieve the reply. If an error occurs while sending the message, an event is invoked and the user is notified. This is similar to the Async mode, except that the API function waits until the requested tag data arrives.

Once a tag arrives and it is pending an event, an event is set to the calling function. Similar to the ElmoGetSyncParameter.

16.9.2.3 Timeouts and errors

Timeouts and errors are also handled using Adapter Tags. These are tags that reside in the Maestro memory space. A device writes/reads to the adapter tags at the stack level and does not need User Program Intervention. These tags are built at EIP initialization, and their responses are handled in the EADK callback function.



1. The User calls the clientRegObjectsForClientProc API and allocates memory accordingly.
2. Stack sends NM_CLIENT_OBJECT_REQUEST_RECEIVED events. Once the event is received by the stack, service type is requested by clientGetClientRequest API
3. Stack sends requested service (read/write) with ID. Service is completed and allocated data sent to stack.
4. User can update/read allocated data asynchronically. If an error occurs while sending the message – an event is invoked and the user is notified.

16.9.3 Ethernet/IP Setup

GMAS operation as an Ethernet/IP adapter (Slave) is actually an application, which runs on the Maestro. As an initial stage it loads and parses an XML file, which defines a specific configuration on the Scanner (e.g. Rockwell PLC). The name of the file can be any name as long as the extension is xml, the scheme is correct (see XML Setup File Sample below), and the adapter program is aware of that name and the file full path.

```
<?xml version="1.0" encoding="utf-8"?>
<EIPCONFIG>
<EIPDEVICES DEVNUM="2">
         <EIPDEVICE NAME="dev_plc2gmas" NETWORKPATH="10.10.20.229,1,0" TYPE="DINT" SIZE="1"/>
         <EIPDEVICE NAME="dev_gmas2plc" NETWORKPATH="10.10.20.229,1,0" TYPE="DINT" SIZE="1"/>
    </EIPDEVICES>
    <EIPASSEMBLIES NUM="3">
         <EIPASSEMBLY NAME="asm_gmas2plc" INSTANCE="1" SIZE="1" OFFSET="0" COMFORMAT="DINT" TYPE="9"/>
         <EIPASSEMBLY NAME="asm_plc2gmas" INSTANCE="2" SIZE="1" OFFSET="4" COMFORMAT="DINT" TYPE="5"/>
         <EIPASSEMBLY NAME="cfgAss" INSTANCE="3" SIZE="1" OFFSET="8" COMFORMAT="DINT" TYPE="17"/>
    </EIPASSEMBLIES>
    <EIPADAPTERS NUM="2">
         <EIPADAPTER TYPE="DINT" NAME="adp_plc2gmas" SIZE="1"/>
         <EIPADAPTER TYPE="DINT" NAME="adp_gmas2plc" SIZE="1"/>
    </EIPADAPTERS>
</EIPCONFIG>
```

16.9.3.1 XML Structure

The XML file consist of three blocks for devices (EIPDEVICES), assemblies (EIPASSEMBLIES) and adapters (EIPADAPTERS). Each of them has an attribute NUM, which declares the amount of elements it contains.

EIPDEVICE Attributes

NAME

Name of variable bound to PLC tag. The EIP program on the Maestro retrieves a tag reference by this name.

NETWORKPATH

<IP address>,1,0. The 1,0 is specifically required by the Rockwell PLC scanner. It may be redundant for other scanners.

TYPE

Variable type: SINT (char, 1 byte), INT (short, 2 bytes), DINT (int, 4 bytes), REAL (float, 4 bytes)

SIZE

Variable dimension. One may declare a device tag as an array.

EIPASSEMBLY Attributes

NAME

Name of variable bound to PLC assembly (IO). The EIP program on the Maestro retrieves a tag reference by this name.

INSTANCE

Unlike other tags (adapter, device), this tag reference may also be retrieved by instance (optionally).

SIZE

Variable dimension. One may declare an assembly tag as an array.

OFFSET

Unlike other tags (adapter, device) for all assemblies which are mapped into one flat buffer in memory, the user should map each **assembly** variable into this buffer according to its type (see COMFORMAT) and dimension. The two variables of type int (DINT) should be mapped. For example, if the first offset is 0 then the second will be 4, since DINT is four byte length.

COMFORMAT

Variable type format: SINT (char, 1 byte), INT (short, 2 bytes), DINT (int, 4 bytes), REAL (float, 4 bytes).

TYPE

This attribute sets the EIP library mode of operation and it should be constant for this stage:

9 - Assembly that produces data on the network (GMAS 2 PLC)

5 - Assembly that consumes data from the network (PLC 2 GMAS)

17 - Assembly that will be used to store configuration.

EIPADAPTER Attributes

NAME

Name of variable bound to the PLC tag. The EIP program on the Maestro retrieves the tag reference by this name.

TYPE

Variable type: SINT (char, 1 byte), INT (short, 2 bytes), DINT (int, 4 bytes), REAL (float, 4 bytes)

SIZE

Variable dimension. One may declare an adapter tag as an array.

16.10 EtherNetIP Functions

The following EtherNetIP communication functions are described:

EtherNetIP Communication
EipGetAdpTagRefByName
EipWriteAdpTag
EipReadAdpTag
EipGetAssemblyRefByInstance
EipGetAssemblyRefByName
EipSetAssembly
EipGetAssembly
EipGetDevTagRefByName
EipSetDevTag
EipGetDevTag
EipReadDevTagData
EipSyncGetDevTag
EipCheckDevTagReply
EipOpenSession
EIPCloseSession
EipCreate
EipDestroy

16.10.1 EipGetAdpTagRefByName

Returns adapter tag reference index according to its name.

```
int EipGetAdpTagRefByName(
IN EIP_REFBYNAME_IN *pInParam,
OUT EIP_REFBYNAME_OUT *pOutParam
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\EIP_API.h

GMAS Programming(IEC 61331 Program)\ElmoEIP

Function Parameters

pInParam

Points to the **EIP_REFBYNAME_IN** input data structure using the EipGetAdpTagRefByName function. References the adapter tag name as declared in the XML configuration file.

pOutParam

Points to the **EIP_REFBYNAME_OUT** output structure receiving information as a result of calling the EipGetAdpTagRefByName function. Index reference to the adapter tag.

Remarks

None

Scope

All

EIP_REFBYNAME_IN Structure

```
typedef struct {
char cName[NAME_MAX_LENGTH];
}EIP_REFBYNAME_IN;
```

Parameters

cName

Tag/assembly name as declared in XML configuration file. The variable [NAME_MAX_LENGTH] is limited to 80 characters.

EIP_REFBYNAME_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned short usTagRef;
}EIP_REFBYNAME_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

usTagRef

Tag/Assembly index reference. Any +ve value accepted.

Figure 16-81 describes the function for EipGetAdpTagRefByName, reflected also in the IEC 61131-3 programming as ELMOEipGetTagRef function.

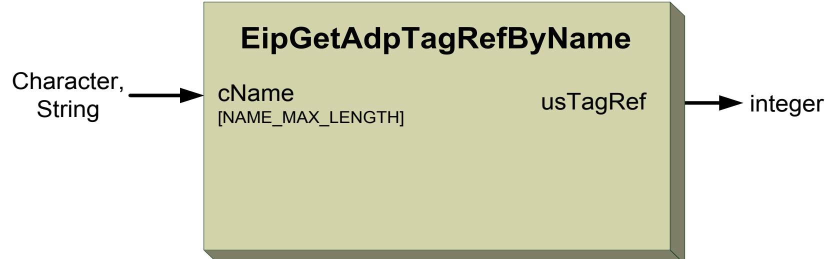


Figure 16-81: EipGetAdpTagRefByName function

16.10.2 EipWriteAdpTag

Writes adapter tag data according to the tag type.

```
int EipWriteAdpTag(
IN EIP_WRITEDATA_IN *pInParam,
OUT EIP_WRITEDATA_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_WRITEDATA_IN** input data structure using the EipWriteAdpTag function. Reference index to the adapter tag.

pOutParam

Points to the **EIP_WRITEDATA_OUT** output structure receiving information as a result of calling the EipWriteAdpTag function. Data to be written to the referenced adapter tag.

Remarks

None

Scope

All

EIP_WRITEDATA_IN Structure

```
typedef struct{
unsigned short usTagRef;
char buffer[MAX_REQUEST_DATA_SIZE];
}EIP_WRITEDATA_IN;
```

Parameters

usTagRef, TagRef

Tag/Assembly index reference. Any +ve value accepted.

buffer

Buffer that holds returned data. The variable [MAX_REQUEST_DATA_SIZE] is limited to 504 characters.

EIP_WRITEDATA_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
int iReqid;
}EIP_WRITEDATA_OUT;
```

Parameters

UsStatus, Done, Active, Error

Bitwise returned command status with the following values:

Aborted Done CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

iReqid

Adapter tag request id

Figure 16-82 describes the function for EipWriteAdpTag, reflected also in the IEC 61131-3 programming.

Figure 16-82: EipWriteAdpTag function

Enable

Exexcute

TagType

Tag type can be Adapter, Developer, Assembly

RW

Read, Write

@Data, Data[]

Any form of data or array of data

16.10.3 EipReadAdpTag

Reads the adapter tag data according to the tag type. Copies adapter tag data from memory into input buffer.

```
int EipReadAdpTag(
IN EIP_READADPTAG_IN *pInParam,
OUT EIP_READADPTAG_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoEIP	

Function Parameters

pInParam

Points to the **EIP_READADPTAG_IN** input data structure using the EipReadAdpTag function. Reference index to adapter tag.

pOutParam

Points to the **EIP_READADPTAG_OUT** output structure receiving information as a result of calling the EipReadAdpTag function. Buffer to hold adapter data.

Remarks

None

Scope

All

EIP_READADPTAG_IN Structure

```
typedef struct{
unsigned short usTagRef;
}EIP_READADPTAG_IN;
```

Parameters

usTagRef

Tag/Assembly index reference. Any +ve value accepted.

EIP_READADPTAG_OUT Structure

```
typedef struct {
unsigned short usStatus;
short sErrorID;
char buffer[MAX_REQUEST_DATA_SIZE];
}EIP_READADPTAG_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

buffer

Buffer that holds returned data. The variable [MAX_REQUEST_DATA_SIZE] is limited to 504 characters.

Figure 16-83 describes the function for EipReadAdpTag, reflected also in the IEC 61131-3 programming for the ELMOEipTag function.

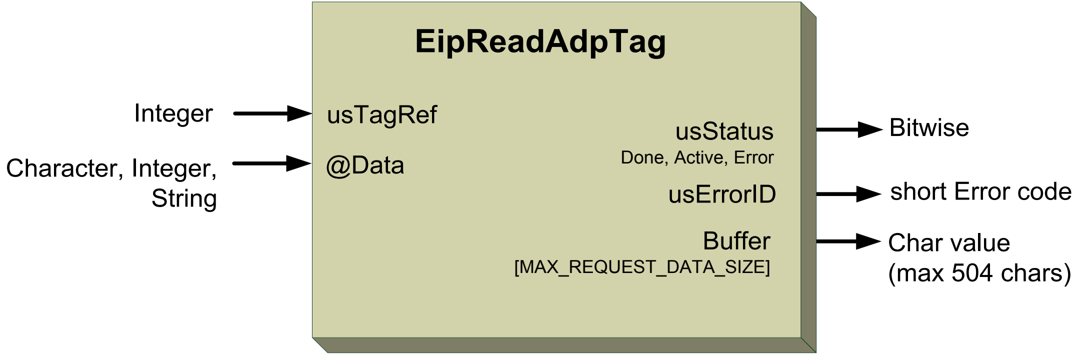


Figure 16-83: EipReadAdpTag function

16.10.4 EipGetAssemblyRefByInstance

Reads the assembly information according to the instance reference .Locates the asm_instance and applies a reference to this instance.

```
int EipGetAssemblyRefByInstance(
IN EIP_REFBYINSTANCE_IN *pInParam,
OUT EIP_REFBYINSTANCE_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_REFBYINSTANCE_IN** input data structure using the EipGetAssemblyRefByInstance function, where asm_instance is an existing assembly instance.

pOutParam

Points to the **EIP_REFBYINSTANCE_OUT** output structure receiving information as a result of calling the EipGetAssemblyRefByInstance function, where the referenced index relates to the requested assembly.

Remarks

None

Scope

All

EIP_REFBYINSTANCE_IN Structure

```
typedef struct{
int iInstance;
}EIP_REFBYINSTANCE_IN;
```

Parameters

iInstance

Assembly instance as declared in the XML configuration file.

EIP_REFBYINSTANCE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned short usTagRef;
}EIP_REFBYINSTANCE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

usTagRef

Tag/Assembly index reference. Any +ve value accepted.

Figure 16-84 describes the function block for EipGetAssemblyRefByInstance

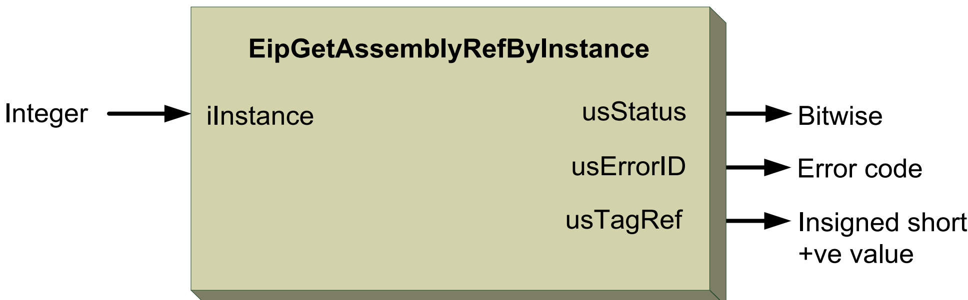


Figure 16-84: EipGetAssemblyRefByInstance function

16.10.5 EipGetAssemblyRefByName

Reads the assembly information according to the name reference .This function returns the assembly reference index according to its name.

```
int EipGetAssemblyRefByName(
IN EIP_REFBYNAME_IN *pInParam,
OUT EIP_REFBYNAME_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_REFBYNAME_IN** input data structure using the EipGetAssemblyRefByName function, where asm_name is the assembly name as declared in the XML configuration file.

pOutParam

Points to the **EIP_REFBYNAME_OUT** output structure receiving information as a result of calling the EipGetAssemblyRefByName function, where the referenced index relates to the adapter tag.

Remarks

None

Scope

All

EIP_REFBYNAME_IN Structure

```
typedef struct{
char cName[NAME_MAX_LENGTH];
}EIP_REFBYNAME_IN;
```

Parameters

cName

Tag/assembly name as declared in the XML configuration file. The variable [NAME_MAX_LENGTH] is limited to 80 characters.

EIP_REFBYNAME_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned short usTagRef;
}EIP_REFBYNAME_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

usTagRef

Tag/Assembly index reference. Any +ve value accepted.

Figure 16-85 describes the function block for EipGetAssemblyRefByName

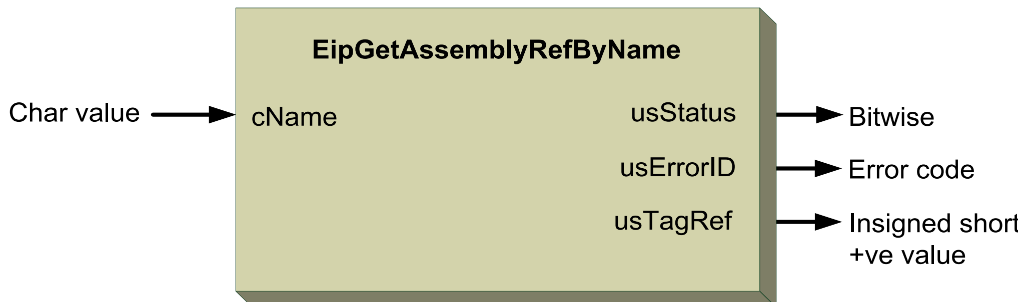


Figure 16-85: EipGetAssemblyRefByName function

16.10.6 EipSetAssembly

Fills the assembly data with out_buff data and sends it through EthernetIP.

```
int EipSetAssembly(
IN EIP_SETASSEMBLY_IN *pInParam,
OUT EIP_SETASSEMBLY_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_SETASSEMBLY_IN** input data structure using the E EipSetAssembly function, where the input is an instance of the requested assembly.

pOutParam

Points to the **EIP_SETASSEMBLY_OUT** output structure receiving information as a result of calling the EipSetAssembly function, where the output relates to the buffer which holds data sent to the assembly.

Remarks

None

Scope

All

EIP_SETASSEMBLY_IN Structure

```
typedef struct{
unsigned short usTagRef;
char buffer[MAX_REQUEST_DATA_SIZE];
}EIP_SETASSEMBLY_IN;
```

Parameters

usTagRef

Index reference to specified assembly (tag)

buffer

Buffer that holds returned data. The variable [MAX_REQUEST_DATA_SIZE] is limited to 504 characters.

EIP_SETASSEMBLY_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
int iReqid;
}EIP_SETASSEMBLY_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

iReqid

Assembly request ID. Integer value.

Figure 16-86 describes the function block for EipSetAssembly

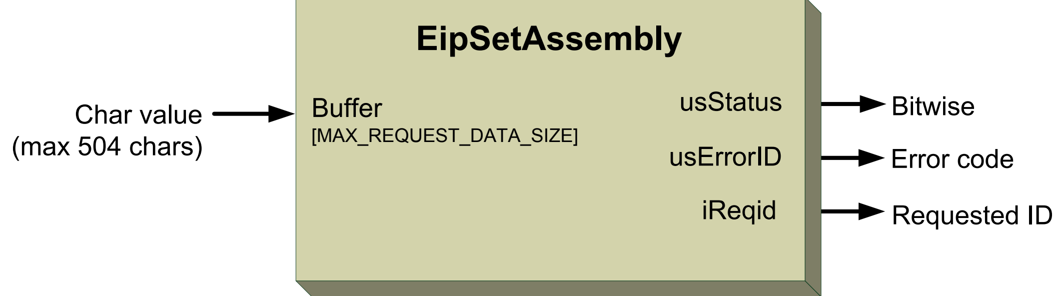


Figure 16-86: EipSetAssembly function

16.10.7 EipGetAssembly

Copies an assembly data identified by instance to in_buff.

```
int EipGetAssembly(
IN EIP_GETASSEMBLY_IN *pInParam,
OUT EIP_GETASSEMBLY_OUT *pOutParam
);
```

Motion Mode NC – Not relevant Distributed – not relevant

Source GMAS\includes\EIP_API.h

Function Parameters

pInParam

Points to the **EIP_GETASSEMBLY_IN** input data structure using the EipGetAssembly function identified by the index reference for requested assembly.

pOutParam

Points to the **EIP_GETASSEMBLY_OUT** output structure receiving information as a result of calling the EipGetAssembly function using the buffer to hold incoming assembly data.

Remarks

None

Scope

All

EIP_GETASSEMBLY_IN Structure

```
typedef struct{
unsigned short usInstance;
}EIP_GETASSEMBLY_IN;
```

Parameters

usInstance

Assembly instance reference. Any +ve value accepted.

EIP_GETASSEMBLY_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
char buffer[MAX_REQUEST_DATA_SIZE];
}EIP_GETASSEMBLY_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

buffer

Buffer that holds returned data. The variable [MAX_REQUEST_DATA_SIZE] is limited to 504 characters.

Figure 16-87 describes the function block for EipGetAssembly

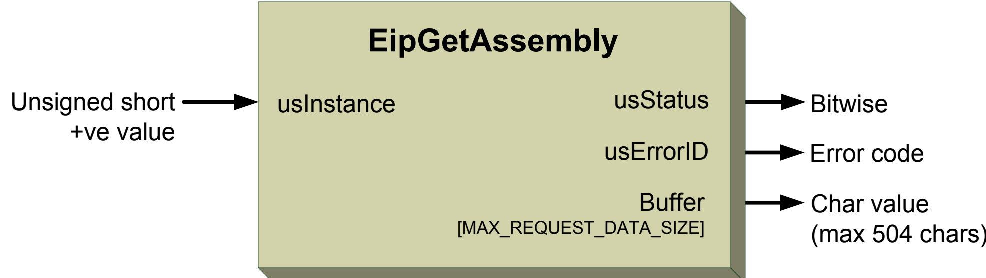


Figure 16-87: EipGetAssembly function

16.10.8 EipGetDevTagRefByName

This function returns device tag reference index according to its name.

```
int EipGetDevTagRefByName(
IN EIP_REFBYNAME_IN *pInParam,
OUT EIP_REFBYNAME_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h GMAS Programming(IEC 61331 Program)\ElmoEIP	

Function Parameters

pInParam

Points to the **EIP_REFBYNAME_IN** input data structure using the EipGetDevTagRefByName function. References the device tag name as declared in XML configuration file.

pOutParam

Points to the **EIP_REFBYNAME_OUT** output structure receiving information as a result of calling the EipGetDevTagRefByName function, and the index reference to the device tag.

Remarks

None

Scope

All

EIP_REFBYNAME_IN Structure

```
typedef struct {
char cName[NAME_MAX_LENGTH];
}EIP_REFBYNAME_IN;
```

Parameters

cName

Tag/assembly name as declared in XML configuration file. The variable [NAME_MAX_LENGTH] is limited to 80 characters.

EIP_REFBYNAME_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
unsigned short usTagRef;
}EIP_REFBYNAME_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

usTagRef

Tag/Assembly index reference. Any +ve value accepted.

Figure 16-88 describes the function block for EipGetDevTagRefByName as applied within the IEC 61131 programming for ELMOEipGetTagRef function.

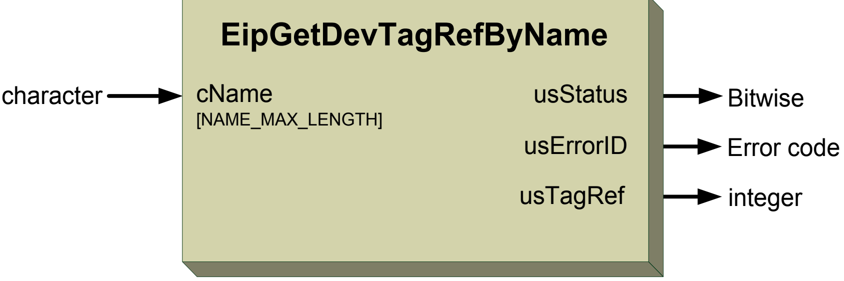


Figure 16-88: EipGetDevTagRefByName function

16.10.9 EipSetDevTag

Writes the device tag data according to the tag type. Updates device tag data and sends it to the EIP device.

```
int EipSetDevTag(
IN EIP_SETDEVTAG_IN *pInParam,
OUT EIP_SETDEVTAG_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_SETDEVTAG_IN** input data structure using the EipSetDevTag function that is index referenced to the device tag.

pOutParam

Points to the **EIP_SETDEVTAG_OUT** output structure receiving information as a result of calling the EipSetDevTag function using the buffer which holds data to be sent to the device tag.

Remarks

None

Scope

All

EIP_SETDEVTAG_IN Structure

```
typedef struct{
unsigned short usTagRef;
char buffer[MAX_REQUEST_DATA_SIZE];
}EIP_SETDEVTAG_IN;
```

Parameters

usTagRef

Index reference to a device tag. Any +ve value accepted.

buffer

Buffer that holds returned data. The variable [MAX_REQUEST_DATA_SIZE] is limited to 504 characters.

EIP_SETDEVTAG_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
int iReqid;
}EIP_SETDEVTAG_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

iReqid

Device tag request id. Integer value.

Figure 16-89 describes the function for EipSetDevTag, reflected also in the IEC 61131-3 programming.

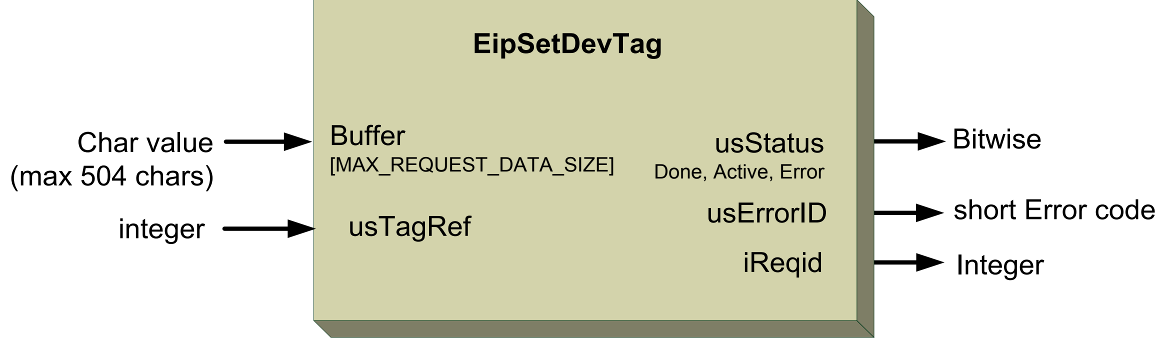


Figure 16-89: EipSetDevTag function

16.10.10 EipGetDevTag

Reads the device tag data according to the tag type. Sends request to the EIP device to read specific device tag.

```
int EipGetDevTag(
IN EIP_GETDEVTAG_IN *pInParam,
OUT EIP_GETDEVTAG_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h GMAS Programming(IEC 61331 Program)\ElmoEIP	

Function Parameters

pInParam

Points to the **EIP_GETDEVTAG_IN** input data structure using the EipGetDevTag function that is index referenced to the device tag.

pOutParam

Points to the **EIP_GETDEVTAG_OUT** output structure receiving information as a result of calling the EipGetDevTag function using the requested ID to be used to read data when received.

Remarks

None

Scope

All

EIP_GETDEVTAG_IN Structure

```
typedef struct{
unsigned short usTagRef;
}EIP_GETDEVTAG_IN;
```

Parameters

usTagRef

T Index reference to a device tag. Any +ve (??) value accepted.

EIP_GETDEVTAG_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
int iReqid;
}EIP_GETDEVTAG_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

iReqid

Device tag request ID. Integer value.

Figure 16-90 describes the function for EipGetDevTag, reflected also in the IEC 61131-3 programming.

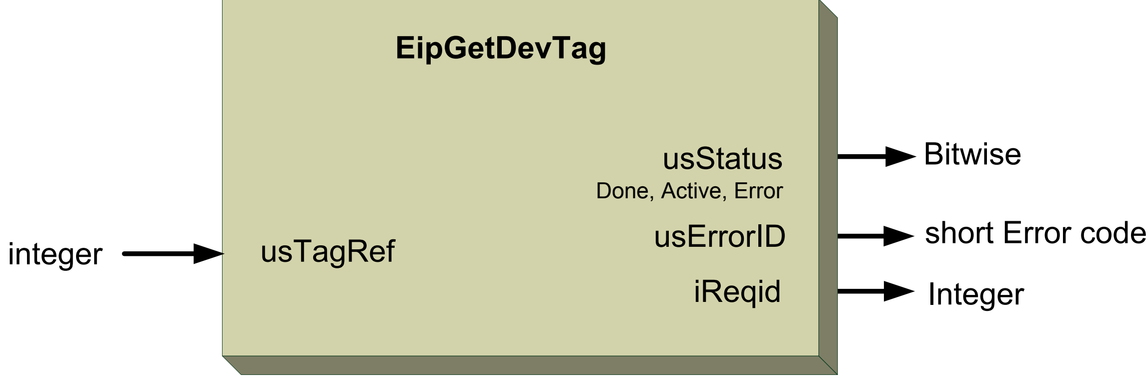


Figure 16-90: EipGetDevTag function

16.10.11 EipReadDevTagData

Reads and stores device tag data received from an EIP device, as a response to user request.

```
int EipReadDevTagData(
IN EIP_READDEVTAG_IN *pInParam,
OUT EIP_READDEVTAG_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h GMAS Programming(IEC 61331 Program)\ElmoEIP	

Function Parameters

pInParam

Points to the **EIP_READDEVTAG_IN** input data structure using the EipReadDevTagData function that is index referenced to the device tag.

pOutParam

Points to the **EIP_READDEVTAG_OUT** output structure receiving information as a result of calling the EipReadDevTagData function using the buffer to hold the incoming device tag data.

Remarks

None

Scope

All

EIP_READDEVTAG_IN Structure

```
typedef struct{
unsigned short usTagRef;
}EIP_READDEVTAG_IN;
```

Parameters

usTagRef

Index reference to a device tag. Any +ve value accepted.

EIP_READDEVTAG_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
char buffer[MAX_REQUEST_DATA_SIZE];
}EIP_READDEVTAG_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

buffer

Buffer that holds returned data. The variable [MAX_REQUEST_DATA_SIZE] is limited to 504 characters.

Figure 16-91 describes the function block for EipReadDevTagData reflected also in the IEC 61131-3 programming.

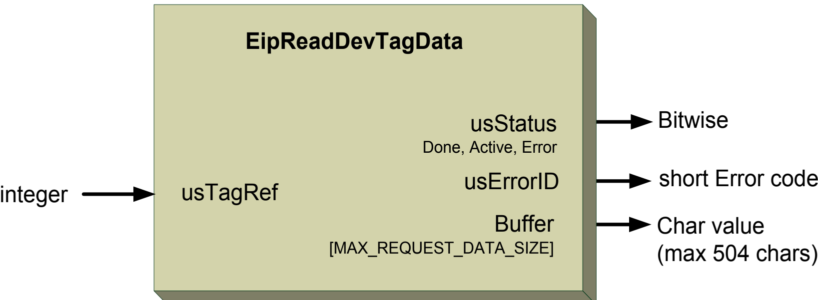


Figure 16-91: EipReadDevTagData function

16.10.12 EipSyncGetDevTag

Sends a request to read device tag data, and waits for a response to be received.

```
int EipSyncGetDevTag(
IN EIP_GETSYNC_IN *pInParam,
OUT EIP_GETSYNC_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_GETSYNC_IN** input data structure using the EipSyncGetDevTag function that is index referenced to the device tag.

pOutParam

Points to the **EIP_GETSYNC_OUT** output structure receiving information as a result of calling the EipSyncGetDevTag function using the buffer to hold the incoming device tag data.

Remarks

None

Scope

All

EIP_GETSYNC_IN Structure

typedef struct{
unsigned short usTagRef;
}EIP_GETSYNC_IN;

Parameters

usTagRef

Tag/Assembly index reference. Any +ve value accepted.

EIP_GETSYNC_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
char buffer[MAX_REQUEST_DATA_SIZE];
}EIP_GETSYNC_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

buffer

Buffer that holds returned data. The variable [MAX_REQUEST_DATA_SIZE] is limited to 504 characters.

Figure 16-92 describes the function block for EipSyncGetDevTag.

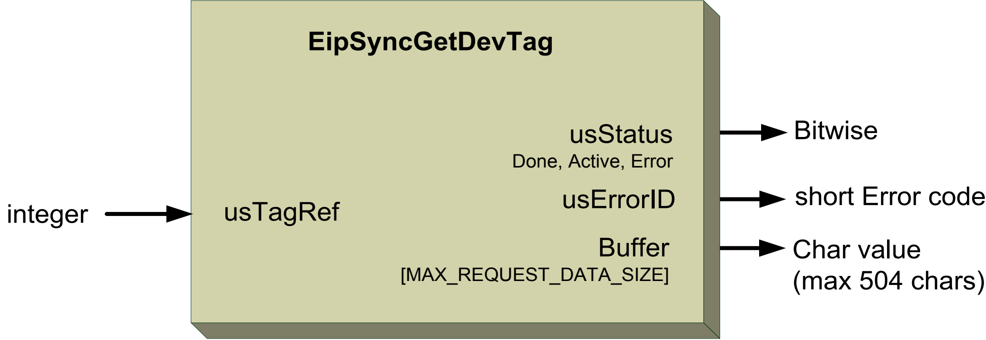


Figure 16-92: EipSyncGetDevTag function

16.10.13 EipCheckDevTagReply

Check that a reply has been received for a specific device tag request.

```
void EipCheckDevTagReply(
IN EIP_CHECKREPLY_IN *pInParam,
OUT EIP_CHECKREPLY_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_CHECKREPLY_IN** input data structure using the EipCheckDevTagReply function that is the device tag request ID.

pOutParam

Points to the **EIP_CHECKREPLY_OUT** output structure receiving information as a result of calling the EipCheckDevTagReply function.

Remarks

None

Scope

All

EIP_CHECKREPLY_IN Structure

```
typedef struct{
int iReqid;
}EIP_CHECKREPLY_IN;
```

Parameters

iReqid

Device tag request ID. Integer value. This request ID submitted as an output value by previous call to EIpGetDevTag API.

EIP_CHECKREPLY_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
short sReplyStatus;
}EIP_CHECKREPLY_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

sReplyStatus

Returned reply status flag

Figure 16-93 describes the function block for EipCheckDevTagReply.

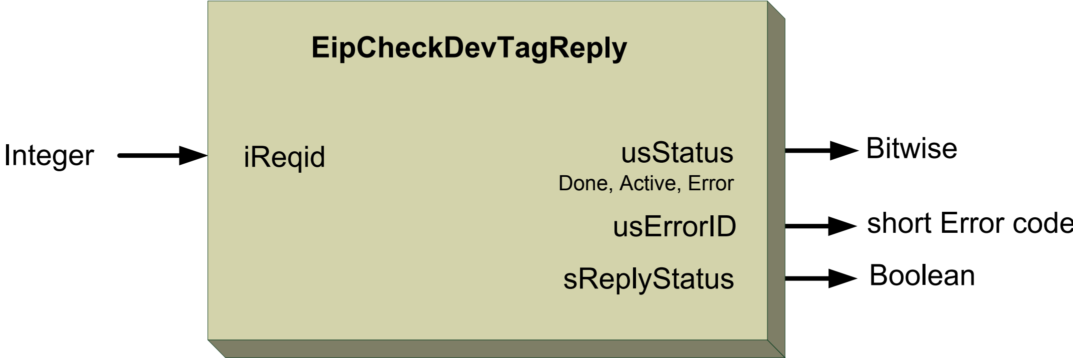


Figure 16-93: EipCheckDevTagReply function

16.10.14 EipOpenSession

Initialize and start an EIP session in order to be able to use EthernetIP.

```
int EipOpenSession(
IN EIP_CALLBACK_FUNC pCallBackFunc,
IN EIP_OPEN_SESSION_IN *pInParam,
OUT EIP_OPEN_SESSION_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h GMAS Programming(IEC 61331 Program)\ElmoEIP	

Function Parameters

pCallBackFunc

The callback function is returned from the stack of the Ethernet IP library originating from the Maestro, together with an EIP type event. The Buffer index may be 0, but the EIP event data is positioned at index 20.

pInParam

Points to the **EIP_OPEN_SESSION_IN** input data structure using the EipOpenSession function.

pOutParam

Points to the **EIP_OPEN_SESSION_OUT** output structure receiving information as a result of calling the EipOpenSession function.

Remarks

None

Scope

All

EIP_OPEN_SESSION_IN Structure

```
typedef struct{
char cNotifyEvant;
}EIP_OPEN_SESSION_IN;
```

Parameters

cNotifyEvant

Notification of an event occurring. Character values of +1 and above

EIP_OPEN_SESSION_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}EIP_OPEN_SESSION_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

Figure 16-94 describes the function block for EipOpenSession as applied within the IEC 61131 programming.

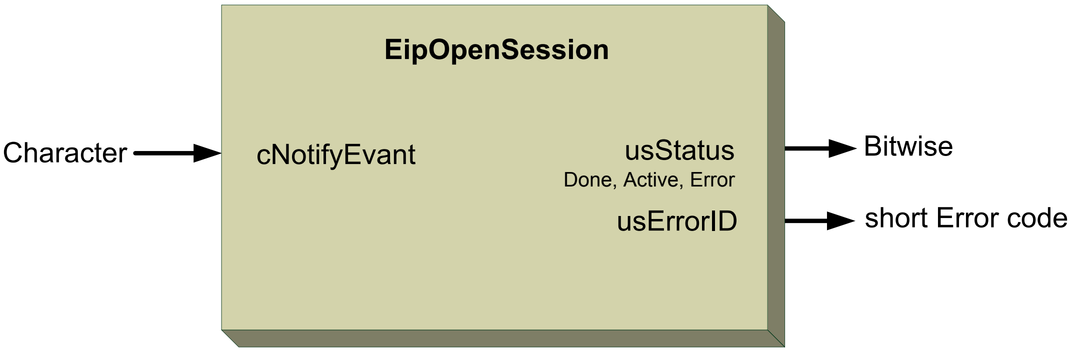


Figure 16-94: EipOpenSession function

16.10.14.2 Implementation Example

When an EIP session is opened, the following EIPCallback may be called to provide error information when the connnection proves faulty.

```
CMMCEIPSession::Instance()->EipOpenSession(EIPCallback, true);

int EIPCallback(unsigned char* ucBuffer, short sReqID, void* pSock)
{
     unsigned char ucEventID = ucBuffer[2];
     switch (ucEventID)
     {
       case NM_REQUEST_RESPONSE_RECEIVED:
//printf("NM_REQUEST_RESPONSE_RECEIVED: sReqID = %d\n", sReqID);
         break;
         case NM_ASSEMBLY_NEW_INSTANCE_DATA:
//printf("NM_ASSEMBLY_NEW_INSTANCE_DATA: assembly instance = %d\n", sReqID);
              break;
         case NM_ASSEMBLY_NEW_MEMBER_DATA:
//New data received for the specified assembly member. sReqID contains assembly instance.
              printf("NM_ASSEMBLY_NEW_MEMBER_DATA: assembly instance = %d\n", sReqID);
             break;
         case NM_REQUEST_FAILED_INVALID_NETWORK_PATH:
              break;
         case NM_REQUEST_TIMED_OUT:
              printf("NM_REQUEST_TIMED_OUT: sReqID = %d\n", sReqID);
              break;
         case NM_CONNECTION_ESTABLISHED:
              printf("\033[1;45m\tEIPTestAll: New connection opened with instance %d\033[0m\n",
sReqID);
              break;
         case NM_CONNECTION_VERIFICATION:
              printf("NM_CONNECTION_VERIFICATION\n");
              break;
         case NM_CONNECTION_RECONFIGURED:
              printf("NM_CONNECTION_RECONFIGURED\n");
              break;
         case NM_CONNECTION_TIMED_OUT:
              printf("\033[1;42m\tEIPTestAll: Connection with instance %d timed out\033[0m\n",
sReqID);
              break;
         case NM_CONNECTION_CLOSED:
              printf("\033[1;43m\tEIPTestAll: Connection with instance %d closed\033[0m\n",
sReqID);
              break;
         case NM_CLIENT_OBJECT_REQUEST_RECEIVED:
              printf("NM_CLIENT_OBJECT_REQUEST_RECEIVED\n");
              break;
         case NM_PENDING_REQUESTS_LIMIT_REACHED:
              printf("NM_PENDING_REQUESTS_LIMIT_REACHED\n");
              break;

         default:
              printf("%s Unhandled(unknown) response event. %d\n", __func__, ucEventID);
              break;
     }
     return 0;
}
```

16.10.15 EIPCloseSession

Close an EtherNETIP session and free allocated memory before terminating program.

```
int EIPCloseSession(
IN EIP_CLOSE_SESSION_IN *pInParam,
OUT EIP_CLOSE_SESSION_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_CLOSE_SESSION_IN** input data structure using the EIPCloseSession function.

pOutParam

Points to the **EIP_CLOSE_SESSION_OUT** output structure receiving information as a result of calling the EIPCloseSession function.

Remarks

None

Scope

All

EIP_CLOSE_SESSION_IN Structure

```
typedef struct{
char cDummy;
}EIP_OPEN_SESSION_IN;
```

Parameters

cDummy

Dummy value

EIP_CLOSE_SESSION_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}EIP_OPEN_SESSION_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

Figure 16-95 describes the function block for EIPCloseSession.

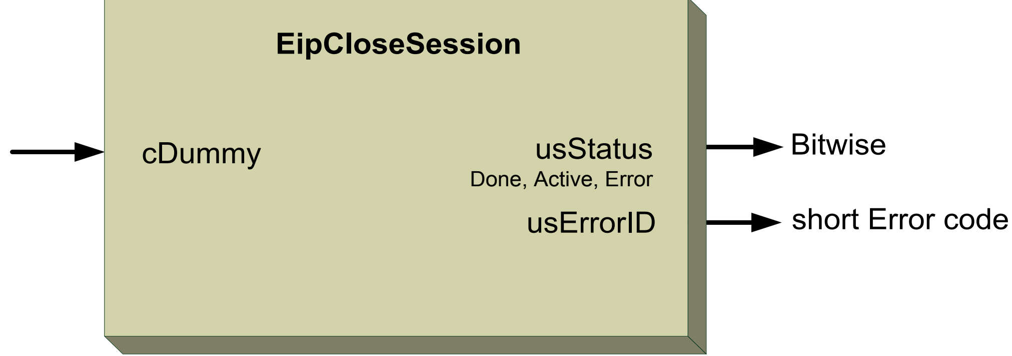


Figure 16-95: EIPCloseSession function

16.10.16 EipCreate

Create an EtherNETIP session.

```
int EipCreate(
IN EIP_CREATE_IN *pInParam,
OUT EIP_CREATE_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_CREATE_IN** input data structure using the EipCreate function.

pOutParam

Points to the **EIP_CREATE_OUT** output structure receiving information as a result of calling the EipCreate function.

Remarks

None

Scope

All

EIP_CREATE_IN Structure

```
typedef struct{
char cPath[80];
}EIP_CREATE_IN;
```

Parameters

cPath[80]

Path of the EtherNETIP session. Value to maximum of 80 characters.

EIP_CREATE_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}EIP_CREATE_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section **4.14 EtherNetIP Communication Error IDs on page 125**. Displays an error code as -ve integers.

Figure 16-96 describes the function block for EipCreate.

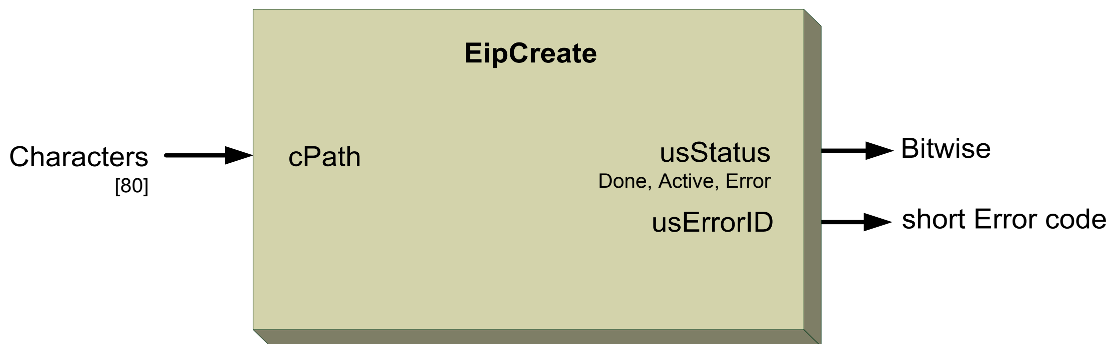


Figure 16-96: EipCreate function

16.10.17 EipDestroy

Kills the EtherNETIP session.

```
int EipDestroy(
IN EIP_DESTROY_IN *pInParam,
OUT EIP_DESTROY_OUT *pOutParam
);
```

Motion Mode	NC – Not relevant	Distributed – not relevant
Source	GMAS\includes\EIP_API.h	

Function Parameters

pInParam

Points to the **EIP_DESTROY_IN** input data structure using the EipDestroy function.

pOutParam

Points to the **EIP_DESTROY_OUT** output structure receiving information as a result of calling the EipDestroy function.

Remarks

None

Scope

All

EIP_DESTROY_IN Structure

```
typedef struct{
char dummy;
}EIP_DESTROY_IN;
```

Parameters

dummy

Dummy value

EIP_DESTROY_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}EIP_DESTROY_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in section 4.14 EtherNetIP Communication Error IDs on page 125. Displays an error code as -ve integers.

Figure 16-97 describes the function block for EipDestroy.

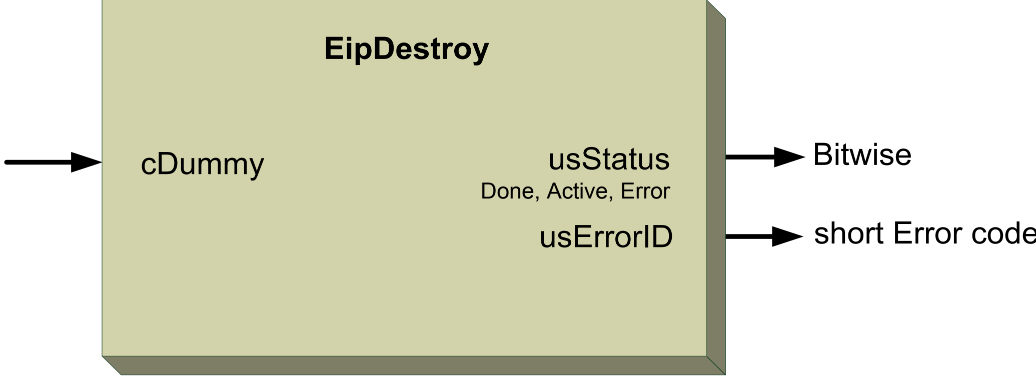


Figure 16-97: EipDestroy function

16.10.18 Functions and Implementation Example

```
/*
* EIPSample.c
*/

#include "EIPSample.h"
#include <expat.h>
#include "EIP_API.h"
#include "datastructs.h"

/* ////////////////////////////////////////////////////////////////////// */
/*                              Globals                                   */
/* ////////////////////////////////////////////////////////////////////// */
float new_pos,position, velocity;
int axis_ref;
int  pos_cmd_id ,servo_cmd_id;
char new_servo_cmd,servo_cmd;

unsigned long  ulState ;
unsigned int err_stat=0;
int val;

/* ////////////////////////////////////////////////////////////////////// */
/*                          external     Globals                          */
/* ////////////////////////////////////////////////////////////////////// */
extern char g_cfg_name[256];

/* ////////////////////////////////////////////////////////////////////// */
/*                              Function Prototypes                       */
/* ////////////////////////////////////////////////////////////////////// */
int EIPSample();
int EipReadInputs();
void EipInit();
int EipWriteStatus();

/* ////////////////////////////////////////////////////////////////////// */
/*                              External Functions                        */
/* ////////////////////////////////////////////////////////////////////// */
extern int MyCallbackFunc (unsigned char*, short,void*) ;
extern int PowerCmdOn();
extern int PowerCmdOff();
extern int MoveAbsolute(int axis_idx,double position , float velocity);


/* ////////////////////////////////////////////////////////////////////// */
/*                              Function Implementation                   */
/* ////////////////////////////////////////////////////////////////////// */

/*****************************************************************************
* FUNCTION:   EIPSample()
* DESCRIPTION:
* INPUTS:     N/A
* OUTPUTS:
* RETURN VALUE:
*****************************************************************************/
int EIPSample()
{
    int indx;

    axis_ref=0;
    sr_data_ref=0;
    pos_cmd_id =0;
    servo_cmd_id =0;
    velocity = 2000;

    EipInit();

    indx=0;
```

```
    while(1)
    {
    indx++;

    EipReadInputs();

        if (!(ulState & NC_AXIS_VALID_MASK))
        {
            usleep(10000);
            continue;
        }

    if (new_servo_cmd != servo_cmd)
    {
        if ((ulState & NC_AXIS_STAND_STILL_MASK) && (!new_servo_cmd))
        {
            printf("Power OFF\n");
            PowerCmdOff();
        }

        if ((ulState & NC_AXIS_DISABLED_MASK) && (new_servo_cmd))
        {
            printf("Power ON\n");
            PowerCmdOn();
        }
    }

    if (new_pos !=position)
    {
        if (ulState & NC_AXIS_STAND_STILL_MASK)
        {
            printf("Move Absolute from %f to %f\n",position,new_pos);
            MoveAbsolute(axis_ref, new_pos , velocity);
            position = new_pos;
        }
    }

    EipWriteStatus();

        if(gTerminateFlag)
        {
            break;
        }
    }

    return 0;
}

/*****************************************************************************
* FUNCTION:    EipReadInputs()
* DESCRIPTION:
* INPUTS:      N/A
* OUTPUTS:
* RETURN VALUE:
*****************************************************************************/
int EipReadInputs()
{

    static int read_position =1;
    static int read_servo =1;

    /* ----------------------------- */
    /* Asynchronous position request */
    /* ----------------------------- */
    if (read_position)
    {
    get_devtag_in.usTagRef = pos_cmd_ref;
        EipGetDevTag(&get_devtag_in, &get_devtag_out);
```

```
    if(get_devtag_out.sErrorID !=0)
{
        printf("position request error\n");
        gTerminateFlag =1;
    return -1;
    }

    pos_cmd_id = get_devtag_out.iReqid;
    read_position =0;
}

/* ------------------------------- */
/* Asynchronous stop command check */
/* ------------------------------- */
if (read_servo)
{
get_devtag_in.usTagRef = servo_cmd_ref;
EipGetDevTag(&get_devtag_in, &get_devtag_out);

    if(get_devtag_out.sErrorID !=0)
{
        printf("servo request error\n");
        gTerminateFlag =1;
    return -1;
    }
    servo_cmd_id = get_devtag_out.iReqid;
    read_servo=0;
}

if (MMC_ElmoGetParameterAndRetrieveData(g_conn_hndl,axis_ref,"SR",0,&val,&err_stat)!=0)
{
    printf("error reading SR paramter\n");
    return -1;
}

/* ----------------------------- */
/* position request reply  check */
/* ----------------------------- */
check_reply_in.iReqid = pos_cmd_id;
EipCheckDevTagReply(&check_reply_in,&check_reply_out);

if (check_reply_out.sReplyStatus)
{

read_dev_in.usTagRef = pos_cmd_ref;
EipReadDevTagData(&read_dev_in,&read_dev_out);

if(read_dev_out.sErrorID!=0)
{
        printf("reading position data failure %d\n",read_dev_out.sErrorID);

        gTerminateFlag =1;
        return -1;
    }

new_pos = *(float *)(read_dev_out.buffer);
read_position =1;
}

/* ------------------------------ */
/* stop flag request reply  check */
/* ------------------------------ */
check_reply_in.iReqid = servo_cmd_id;
EipCheckDevTagReply(&check_reply_in,&check_reply_out);

if (check_reply_out.sReplyStatus)
{
    read_dev_in.usTagRef = servo_cmd_ref;
EipReadDevTagData(&read_dev_in,&read_dev_out);
```

```
    if(read_dev_out.sErrorID!=0)
        {
            printf("reading stop flag failure %d\n",read_dev_out.sErrorID);
            gTerminateFlag =1;
            return -1;
        }

        new_servo_cmd = *read_dev_out.buffer;
        read_servo =1;
    }


    if (MMC_ReadStatusCmd(g_conn_hndl, axis_ref, &read_status_in,&read_status_out) != 0)
    {
        printf("%s Read Status Failed\n",__func__);
        return -1;
    }
    ulState = read_status_out.ulState ;

    return 0;

}

/*****************************************************************************
* FUNCTION:   EipReadInputs()
* DESCRIPTION:
* INPUTS:     N/A
* OUTPUTS:
* RETURN VALUE:
*****************************************************************************/
int EipWriteStatus()
{
    MMC_ReadActualPositionCmd(g_conn_hndl,  axis_ref, &read_pos_in,    &read_pos_out);
    MMC_ReadActualVelocityCmd(g_conn_hndl,  axis_ref, &read_vel_in, &read_vel_out);

    write_adp_in.usTagRef =axis_data_ref ;
    *(float*)&write_adp_in.buffer[0] = read_pos_out.dbPosition;
    EipWriteAdpTag (&write_adp_in,&write_adp_out);


    write_adp_in.usTagRef =sr_data_ref ;
    *(int*)&write_adp_in.buffer[0] = val;
    EipWriteAdpTag (&write_adp_in,&write_adp_out);

    //position=new_pos;
    servo_cmd = new_servo_cmd;

    set_asm_in.usInstance =1;
    *(float*)&set_asm_in.buffer = read_vel_out.dVelocity;
    EipSetAssembly(&set_asm_in, &set_asm_out);

    return 0;
}

/*****************************************************************************
* FUNCTION:   EipReadInputs()
* DESCRIPTION:
* INPUTS:     N/A
* OUTPUTS:
* RETURN VALUE:
*****************************************************************************/
void EipInit()
{

    int (*fun_ptr)(unsigned char*, short,void*);
    fun_ptr = NULL;//MyCallbackFunc;

    /* ---------------- */
```

```
    /* open EIP session */
    /* ---------------- */
    open_session_in.cNotifyEvant=1;
    if (EipOpenSession(fun_ptr, &open_session_in, &open_session_out)!=0)
    {
        printf("open session error %d\n",open_session_out.sErrorID);
    }

    /* ----------------------------------------------------------- */
    /* allocate EIP session memory according to configuration file */
    /* ----------------------------------------------------------- */
    strcpy(create_in.cPath,g_cfg_name);
    if (EipCreate(&create_in,&create_out)!=0)
    {
        printf("create session error %d\n",create_out.sErrorID);
    }

    /* -------------------------- */
    /* get device tags references */
    /* -------------------------- */
    strcpy(ref_by_name_in.cName,"ServoCmd");
    EipGetDevTagRefByName(&ref_by_name_in, &ref_by_name_out);
    servo_cmd_ref = ref_by_name_out.usTagRef;

    strcpy(ref_by_name_in.cName,"PosCmd");
    EipGetDevTagRefByName(&ref_by_name_in, &ref_by_name_out);
    pos_cmd_ref = ref_by_name_out.usTagRef;

    /* --------------------------- */
    /* get adapter tags references */
    /* --------------------------- */
    strcpy(ref_by_name_in.cName,"AxisParams");
    EipGetAdpTagRefByName(&ref_by_name_in,&ref_by_name_out);
    axis_data_ref = ref_by_name_out.usTagRef;

    strcpy(ref_by_name_in.cName,"SR_data");
    EipGetAdpTagRefByName(&ref_by_name_in,&ref_by_name_out);
    sr_data_ref = ref_by_name_out.usTagRef;
    printf("sr_data_ref %d\n",sr_data_ref);

    /* Assemblies instance references */
    ref_by_instance_in.iInstance =1;
    EipGetAssemblyRefByInstance (&ref_by_instance_in, &ref_by_instance_out);
    velocity_ref = ref_by_instance_out.usTagRef;

    /*Init position to current actual position */
    MMC_ReadActualPositionCmd(g_conn_hndl,  axis_ref, &read_pos_in,   &read_pos_out);
    position = read_pos_out.dbPosition;
    printf("start position is %f\n",position);

}
```

16.11 DS-401 CANbus I/O Communications

This section details the CANbus input and output communication to the Maestro (DS-401 digital and analog input/output modules). This form of communication uses the CANopen protocol and device profile specification for embedded systems used in automation.

The purpose of Input/Output modules is to connect sensors and actuators to CANopen networks. In operational mode, input data can be transmitted from the inputs via TPDOs. By default, the PDO transmission is triggered by an interrupt (event). Optionally, PDOs may be transmitted synchronously or remotely requested. In addition, it is possible to read input data via SDO communication from another module, or to write data via SDO to the network, if the module provides SDO client functions. Output data can be received via RPDO by those Input/Output modules that have output capabilities. Output data can also be received via SDO communication services. However, the main purpose of SDO communication is to configure an Input/Output module. Via SDO, the module can receive Input/Output configuration data, and parameters for converting data into meaningful measurements and so on. Input/Output modules compliant with this device profile use pre-defined PDOs. The default mapping of application objects into TPDO and respectively RPDO may be changed via SDO, if variable PDO mapping is supported. An Input/Output module may provide optionally Sync producer/consumer, Time-Stamp producer/consumer, and Emergency producer/consumer functions. For new servo driver designs, it is highly recommended to support Heartbeat functions.

It should be noted that the valid bit and additional logic added to the PDO mapping sequence require that the MMC_CancelGeneralXXPDOX functions must be called prior to calling the relevant MMC_ConfigGeneralXXPDOX function.

16.12 DS-401 Function Blocks

The following DS-401 I/O communication function blocks are described:

DS-401 I/O Communications
MMC_CancelGeneralRPDO3
MMC_CancelGeneralRPDO4
MMC_CancelGeneralTPDO3
MMC_CancelGeneralTPDO4
MMC_ConfigGeneralRPDO3
MMC_ConfigGeneralRPDO4
MMC_ConfigGeneralTPDO3
MMC_ConfigGeneralTPDO4
MMC_DisableDS401DIChangedEvent
MMC_EnableDS401DIChangedEvent
MMC_ReadDS401DIGroup
MMC_ReadDS401DInput
MMC_WriteDS401DOGroup
MMC_WriteDS401DOutput

16.12.1 MMC_CancelGeneralRPDO3

Cancels the general configuration of the DS-401 node or Maestro for RX at PDO3.

```
MMC_LIB_API int MMC_CancelGeneralRPDO3(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CANCELGENERALRPDO3_IN* pInParam,
OUT MMC_CANCELGENERALRPDO3_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_DS401_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CANCELGENERALRPDO3_IN** input data structure using the MMC_CancelGeneralRPDO3 function.

pOutParam

Points to the **MMC_CANCELGENERALRPDO3_OUT** output structure receiving information, as a result of calling the MMC_CancelGeneralRPDO3 function.

Remarks

Cancels communications to read/write PDO’s sent from the Maestro or host.

Scope

All

MMC_CANCELGENERALRPDO3_IN Structure

typedef struct{
unsigned char ucDummy;
}MMC_CANCELGENERALRPDO3_IN;

Parameters

ucDummy

Dummy values. Any +ve character value.

MMC_CANCELGENERALRPDO3_OUT Structure

typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CANCELGENERALRPDO3_OUT;

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-98 describes the function block for MMC_CancelGeneralRPDO3 as applied within the IEC 61131 programming.

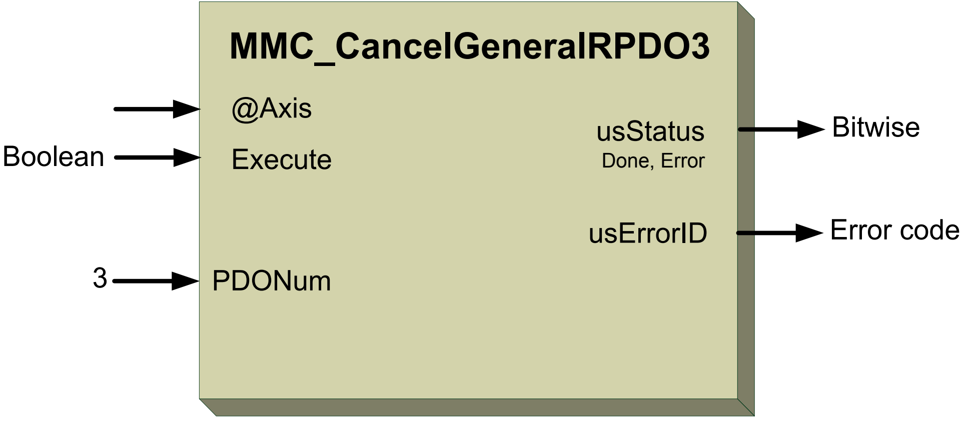


Figure 16-98: MMC_CancelGeneralRPDO3 function block

16.12.1.2 Function Block Code Example

Refer to the example in section **16.12.4.2**.

16.12.2 MMC_CancelGeneralRPDO4

Cancels the general configuration of the DS-401 node or Maestro for RX at PDO4.

```
MMC_LIB_API int MMC_CancelGeneralRPDO4(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CANCELGENERALRPDO4_IN* pInParam,
OUT MMC_CANCELGENERALRPDO4_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_DS401_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CANCELGENERALRPDO4_IN** input data structure using the MMC_CancelGeneralRPDO4 function.

pOutParam

Points to the **MMC_CANCELGENERALRPDO4_OUT** output structure receiving information, as a result of calling the MMC_CancelGeneralRPDO4 function.

Remarks

Cancels communications to read/write PDO’s sent from the Maestro or host.

Scope

All

MMC_CANCELGENERALRPDO4_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_CANCELGENERALRPDO4_IN;
```

Parameters

ucDummy

Dummy values. Any +ve character value.

MMC_CANCELGENERALRPDO4_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CANCELGENERALRPDO4_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-99 describes the function block for MMC_CancelGeneralRPDO4 as applied within the IEC 61131 programming.

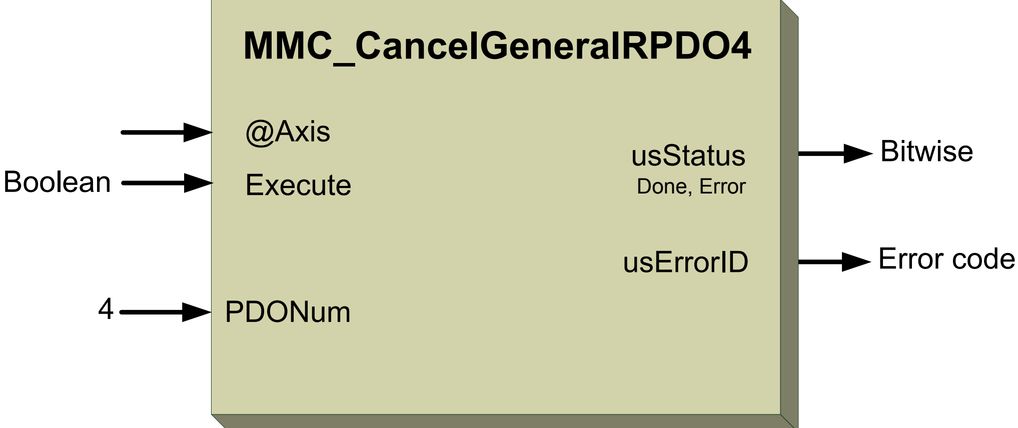


Figure 16-99: MMC_CancelGeneralRPDO4 function block

16.12.2.2 Function Block Code Example

```
MMC_CANCELGENERALRPDO3_IN Cancel3InParam;
MMC_CANCELGENERALRPDO3_OUT Cancel3OutParam;
MMC_CANCELGENERALRPDO4_IN Cancel4InParam;
MMC_CANCELGENERALRPDO4_OUT Cancel4OutParam;

rc = MMC_CancelGeneralRPDO3(ConnHndl,hAxisRef & Cancel3InParam,&Cancel3OutParam);
if(rc != 0)
{
 printf("MMC_CancelGeneralRPDO3 failed, error %d\n",Cancel3OutParam.sErrorID);
}

rc = MMC_CancelGeneralRPDO4(ConnHndl,hAxisRef,&Cancel4InParam,&Cancel4OutParam);
if(rc != 0)
{
 printf("MMC_CancelGeneralRPDO4 failed, error %d\n",Cancel4OutParam.sErrorID);
}
```

16.12.3 MMC_CancelGeneralTPDO3

Cancels the general configuration of the DS-401 node or Maestro for TX at PDO3.

```
MMC_LIB_API int MMC_CancelGeneralTPDO3(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CANCELGENERALTPDO3_IN* pInParam,
OUT MMC_CANCELGENERALTPDO3_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_DS401_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CANCELGENERALTPDO3_IN** input data structure using the MMC_CancelGeneralTPDO3 function.

pOutParam

Points to the **MMC_CANCELGENERALTPDO3_OUT** output structure receiving information, as a result of calling the MMC_CancelGeneralTPDO3 function.

Remarks

Cancels communications to read/write PDO's sent from the Maestro or host.

Scope

All

MMC_CANCELGENERALTPDO3_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_CANCELGENERALTPDO3_IN;
```

Parameters

ucDummy

Dummy values. Any +ve character value.

MMC_CANCELGENERALTPDO3_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CANCELGENERALTPDO3_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-100 describes the function block for MMC_CancelGeneralTPDO3 as applied within the IEC 61131 programming.

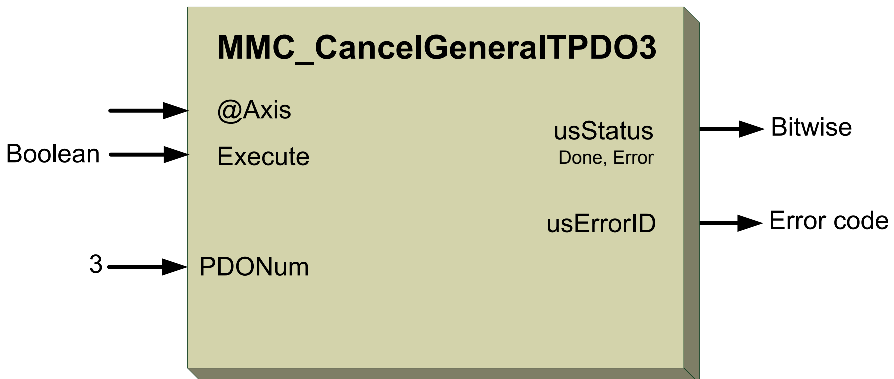


Figure 16-100: MMC_CancelGeneralTPDO3 function block

16.12.3.2 Function Block Code Example

Refer to the example in section **16.12.4.2**.

16.12.4 MMC_CancelGeneralTPDO4

Cancels the general configuration of the DS-401 node or Maestro for TX at PDO4.

```
MMC_LIB_API int MMC_CancelGeneralTPDO4(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CANCELGENERALTPDO4_IN* pInParam,
OUT MMC_CANCELGENERALTPDO4_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_DS401_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CANCELGENERALTPDO4_IN** input data structure using the MMC_CancelGeneralTPDO4 function.

pOutParam

Points to the **MMC_CANCELGENERALTPDO4_OUT** output structure receiving information, as a result of calling the MMC_CancelGeneralTPDO4 function.

Remarks

Cancels communications to read/write PDO’s sent from the Maestro or host.

Scope

All

MMC_CANCELGENERALTPDO4_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_CANCELGENERALTPDO4_IN;
```

Parameters

ucDummy

Dummy values. Any +ve character value.

MMC_CANCELGENERALTPDO4_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CANCELGENERALTPDO4_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-101 describes the function block for MMC_CancelGeneralTPDO4 as applied within the IEC 61131 programming.

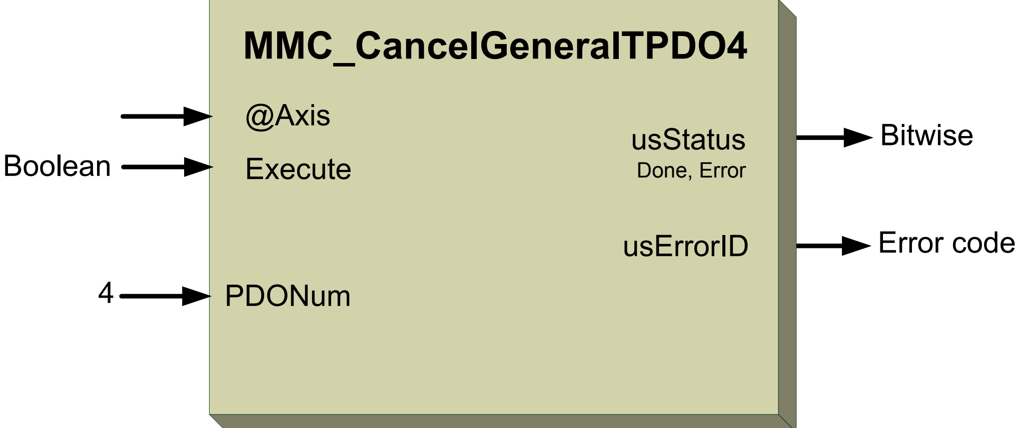


Figure 16-101: MMC_CancelGeneralTPDO4 function block

16.12.4.2 Function Block Code Example

```
MMC_CANCELGENERALTPDO3_IN Cancel3InParam;
MMC_CANCELGENERALTPDO3_OUT Cancel3OutParam;
MMC_CANCELGENERALTPDO4_IN Cancel4InParam;
MMC_CANCELGENERALTPDO4_OUT Cancel4OutParam;

rc = MMC_CancelGeneralTPDO3(ConnHndl,hAxisRef & Cancel3InParam,&Cancel3OutParam);
if(rc != 0)
{
 printf("MMC_CancelGeneralTPDO3 failed, error %d\n",Cancel3OutParam.sErrorID);
}

rc = MMC_CancelGeneralTPDO4(ConnHndl,hAxisRef,&Cancel4InParam,&Cancel4OutParam);
if(rc != 0)
{
 printf("MMC_CancelGeneralTPDO4 failed, error %d\n",Cancel4OutParam.sErrorID);
}
```

16.12.5 MMC_ConfigGeneralRPDO3

Generally configures the DS-401 node or Maestro for RX at PDO3.

```
MMC_LIB_API int MMC_ConfigGeneralRPDO3(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGGENERALRPDO3_IN* pInParam,
OUT MMC_CONFIGGENERALRPDO3_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_DS401_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGGENERALRPDO3_IN** input data structure using the MMC_ConfigGeneralRPDO3 function.

pOutParam

Points to the **MMC_CONFIGGENERALRPDO3_OUT** output structure receiving information, as a result of calling the MMC_ConfigGeneralRPDO3 function.

Remarks

Opens communications to read/write PDO's sent from the Maestro or host. Make sure to map the PDOs by itself before using these APIs.

Scope

All

MMC_CONFIGGENERALRPDO3_IN Structure

```
typedef struct{
unsigned char ucEventType;
unsigned char ucPDOCommParam;
unsigned char ucPDOLength;
}MMC_CONFIGGENERALRPDO3_IN;
```

Parameters

ucEventType

Defines which group of events are to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping** the correct definition to be used. Any +ve character values are acceptable.

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC 0x01

PDO_COM_PARAM_ASYNC 0xFF

ucPDOLength

Indicates the number of bytes to be sent as an RPDO, RPDO message. It can contain 1-8 bytes of data.

MMC_CONFIGGENERALRPDO3_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGGENERALRPDO3_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-102 describes the function block for MMC_ConfigGeneralRPDO3 as applied within the IEC 61131 programming.

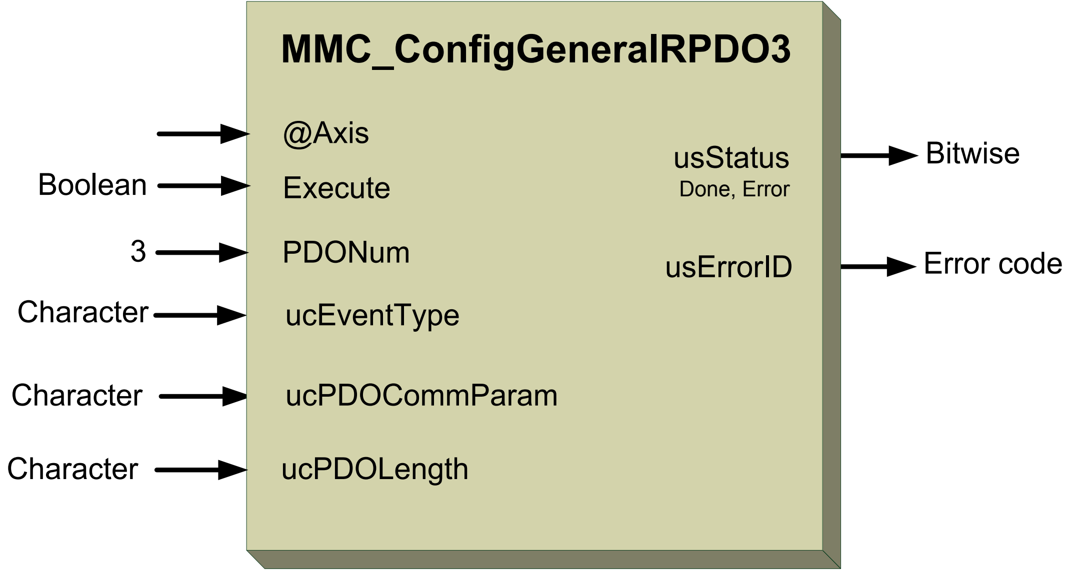


Figure 16-102: MMC_ConfigGeneralRPDO3 function block

16.12.5.2 Function Block Code Example

Refer to the example in section **16.12.6.2**.

16.12.6 MMC_ConfigGeneralRPDO4

Generally configures the DS-401 node or Maestro for RX at PDO4.

```
MMC_LIB_API int MMC_ConfigGeneralRPDO4(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGGENERALRPDO4_IN* pInParam,
OUT MMC_CONFIGGENERALRPDO4_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_DS401_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGGENERALRPDO4_IN** input data structure using the MMC_ConfigGeneralRPDO4 function.

pOutParam

Points to the **MMC_CONFIGGENERALRPDO4_OUT** output structure receiving information, as a result of calling the MMC_ConfigGeneralRPDO4 function.

Remarks

Opens communications to read/write PDO’s sent from the Maestro or host. Make sure to map the PDOs by itself before using these APIs.

Scope

All

MMC_CONFIGGENERALRPDO4_IN Structure

```
typedef struct{
unsigned char ucEventType;
unsigned char ucPDOCommParam;
unsigned char ucPDOLength;
}MMC_CONFIGGENERALRPDO4_IN;
```

Parameters

ucEventType

Defines which group of events are to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping** the correct definition to be used. Any +ve character values are acceptable.

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC	0x01
PDO_COM_PARAM_ASYNC	0xFF
PDO_COM_PARAM_EVENT	0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

ucPDOLength

Indicates the number of bytes to be sent as an RPDO, RPDO message. It can contain 1-8 bytes of data.

MMC_CONFIGGENERALRPDO4_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGGENERALRPDO4_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-103 describes the function block for MMC_ConfigGeneralRPDO4

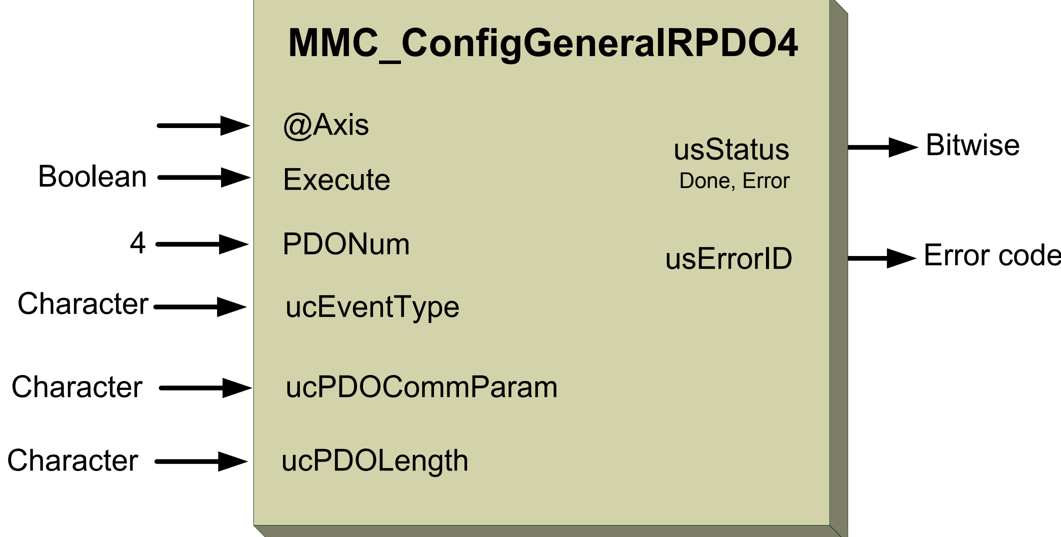


Figure 16-103: MMC_ConfigGeneralRPDO4 function block

16.12.6.2 Function Block Code Example

```
MMC_CONFIGGENERALRPDO3_IN ConfigRPDO3InParam;
MMC_CONFIGGENERALRPDO3_OUT ConfigRPDO3OutParam;
MMC_CONFIGGENERALRPDO4_IN ConfigRPDO4InParam;
MMC_CONFIGGENERALRPDO4_OUT ConfigRPDO4OutParam;

        ConfigRPDO3InParam.ucEventType = 16;
        ConfigRPDO4InParam.ucEventType = 17;
        ConfigRPDO3InParam.ucPDOCommParam = 0x01;
        ConfigRPDO4InParam.ucPDOCommParam = 0x01;
        ConfigRPDO3InParam.ucPDOLength = 2;
        ConfigRPDO4InParam.ucPDOLength = 2;

rc = MMC_ConfigGeneralRPDO3(ConnHndl,hAxisRef,& ConfigRPDO3InParam,&ConfigRPDO3OutParam);
 if(rc != 0)
 {
  printf("MMC_ConfigGeneralRPDO3 failed, error %d\n",ConfigRPDOOutParam.sErrorID);
 }

rc = MMC_ConfigGeneralRPDO4(ConnHndl,hAxisRef,&ConfigRPDO4InParam,&ConfigRPDO4OutParam);
 if(rc != 0)
 {
  printf("MMC_ConfigGeneralRPDO4 failed, error %d\n",ConfigRPDO4OutParam.sErrorID);
 }
```

16.12.7 MMC_ConfigGeneralTPDO3

Generally configures the DS-401 node or Maestro for TX at PDO3.

```
MMC_LIB_API int MMC_ConfigGeneralTPDO3(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGGENERALTPDO3_IN* pInParam,
OUT MMC_CONFIGGENERALTPDO3_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_DS401_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGGENERALTPDO3_IN** input data structure using the MMC_ConfigGeneralTPDO3 function.

pOutParam

Points to the **MMC_CONFIGGENERALTPDO3_OUT** output structure receiving information, as a result of calling the MMC_ConfigGeneralTPDO3 function.

Remarks

Opens communications to read/write PDO's sent from the Maestro or host. Make sure to map the PDOs by itself before using these APIs.

Scope

All

MMC_CONFIGGENERALTPDO3_IN Struzcture

```
typedef struct{
unsigned char ucEventType;
}MMC_CONFIGGENERALTPDO3_IN;
```

Parameters

ucEventType

Defines which group of events are to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping** the correct definition to be used. Any +ve character values are acceptable.

MMC_CONFIGGENERALTPDO3_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGGENERALTPDO3_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-104 describes the function block for MMC_ConfigGeneralTPDO3 as applied within the IEC 61131 programming.

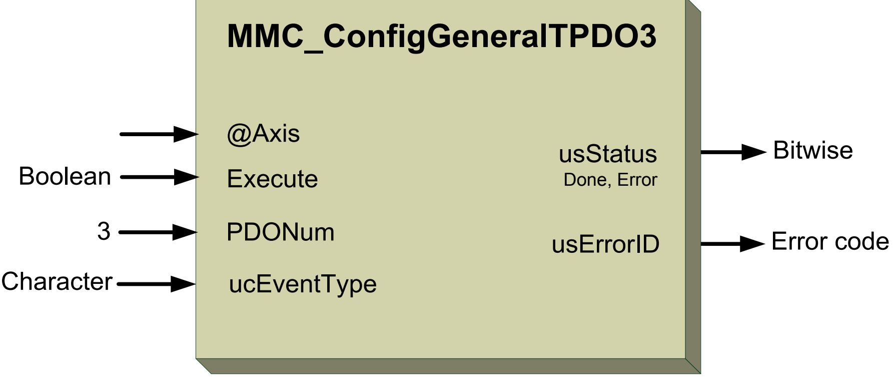


Figure 16-104: MMC_ConfigGeneralTPDO3 function block

16.12.7.2 Function Block Code Example

Refer to the example in section 16.12.8.2.

16.12.8 MMC_ConfigGeneralTPDO4

Generally configures the DS-401 node or Maestro for TX at PDO4.

```
MMC_LIB_API int MMC_ConfigGeneralTPDO4(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_CONFIGGENERALTPDO4_IN* pInParam,
OUT MMC_CONFIGGENERALTPDO4_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_DS401_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_CONFIGGENERALTPDO4_IN** input data structure using the MMC_ConfigGeneralTPDO4 function.

pOutParam

Points to the **MMC_CONFIGGENERALTPDO4_OUT** output structure receiving information, as a result of calling the MMC_ConfigGeneralTPDO4 function.

Remarks

Opens communications to read/write PDO's sent from the Maestro or host. Make sure to map the PDOs by itself before using these APIs.

Scope

All

MMC_CONFIGGENERALTPDO4_IN Structure

```
typedef struct{
unsigned char ucEventType;
}MMC_CONFIGGENERALTPDO4_IN;
```

Parameters

ucEventType

Defines which group of events are to be transferred from the Maestro. Refer to the section 16.4.3 **PDO Mapping on page 1239** the correct definition to be used. Any +ve character values are acceptable.

MMC_CONFIGGENERALTPDO4_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_CONFIGGENERALTPDO4_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs, and 4.9 NC Profiler Error IDs**.

Figure 16-105 describes the function block for MMC_ConfigGeneralTPDO4 as applied within the IEC 61131 programming.

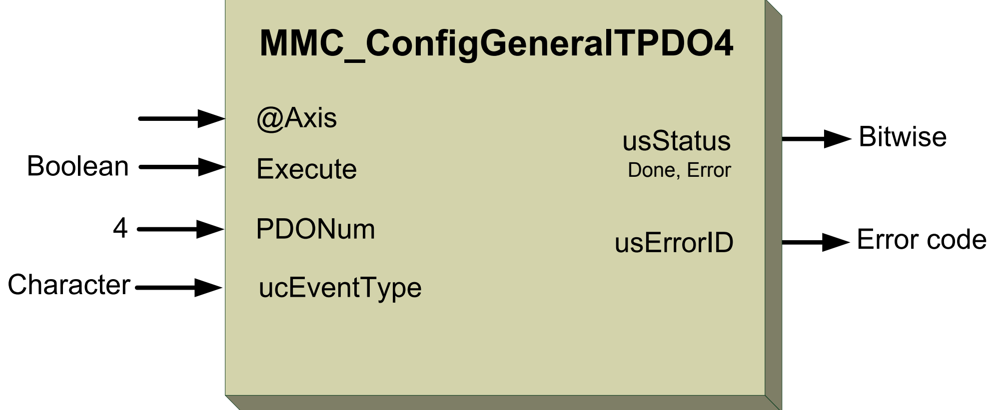


Figure 16-105: MMC_ConfigGeneralTPDO4 function block

16.12.8.2 Function Block Code Example

```
MMC_CONFIGGENERALTPDO3_IN ConfigTPDO3InParam;
MMC_CONFIGGENERALTPDO3_OUT ConfigTPDO3OutParam;
MMC_CONFIGGENERALTPDO4_IN ConfigTPDO4InParam;
MMC_CONFIGGENERALTPDO4_OUT ConfigTPDO4OutParam;

    ConfigTPDO3OutParam.ucEventType = 16;
    ConfigTPDO4InParam.ucEventType = 17;
rc = MMC_ConfigGeneralTPDO3(ConnHndl,hAxisRef,& ConfigTPDO3InParam,&ConfigTPDO3OutParam);
 if(rc != 0)
 {
  printf("MMC_ConfigGeneralTPDO3 failed, error %d\n",ConfigTPDO3OutParam.sErrorID);
 }

rc = MMC_ConfigGeneralTPDO4(ConnHndl,hAxisRef,&ConfigTPDO4InParam,&ConfigTPDO4OutParam);
 if(rc != 0)
 {
  printf("MMC_ConfigGeneralTPDO4 failed, error %d\n",ConfigTPDO4OutParam.sErrorID);
 }
```

16.12.9 MMC_DisableDS401DIChangedEvent

Disables a DS401 digital input event change against an I/O module.

```
MMC_LIB_API int MMC_DisableDS401DIChangedEvent(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_DISABLEDICHANGEDEVENT_IN* pInParam,
OUT MMC_DISABLEDICHANGEDEVENT_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_DS401_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_DISABLEDICHANGEDEVENT_IN** input data structure using the MMC_DisableDS401DIChangedEvent function.

pOutParam

Points to the **MMC_DISABLEDICHANGEDEVENT_OUT** output structure receiving information, as a result of calling the MMC_DisableDS401DIChangedEvent function.

Remarks

When disabled, prevents any DS401 digital input event change being sent from the I/O module to the Maestro and then host server (if connected).

Scope

All

MMC_DISABLEDICHANGEDEVENT_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_DISABLEDICHANGEDEVENT_IN;
```

Parameters

ucDummy

Dummy data input. Any +ve character value.

MMC_DISABLEDICHANGEDEVENT_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_DISABLEDICHANGEDEVENT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-106 describes the function block for MMC_DisableDS401DIChangedEvent as applied within the IEC 61131 programming.

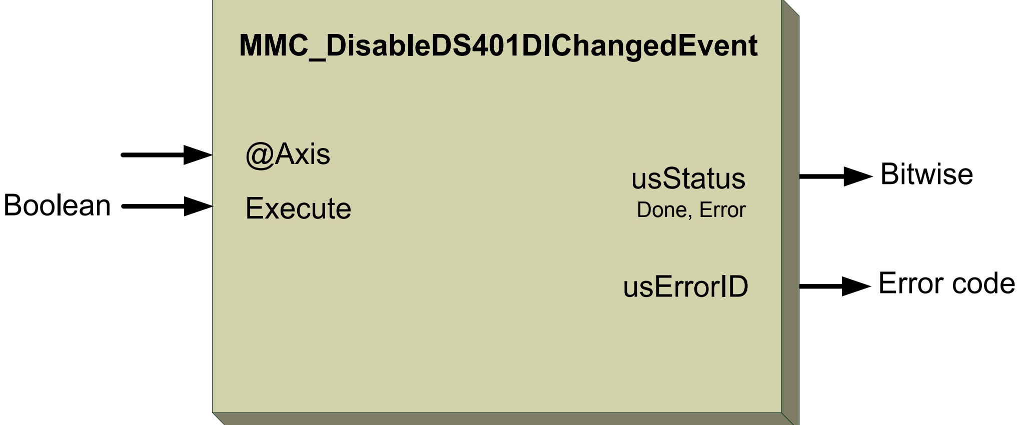


Figure 16-106: MMC_DisableDS401DIChangedEvent function block

16.12.9.2 Function Block Code Example

```
int rc;
MMC_DISABLEDICHANGEDEVENT_IN        stDisableDIChangeEv_in;
MMC_DISABLEDICHANGEDEVENT_OUT       stDisableDIChangeEv_out;
//
// Inserting the structure parameters:
stDisableDIChangeEv_in.ucDummy  = 1;     //Dummy data input
//
rc = MMC_DisableDS401DIChangedEvent (hConn, iAxisRef, &stDisableDIChangeEv_in,
&stDisableDIChangeEv_out);
if (rc != 0)
{
    HandleError();
}
```

16.12.10 MMC_EnableDS401DIChangedEvent

Enables an DS401 digital input event change.

```
MMC_LIB_API int MMC_EnableDS401DIChangedEvent (
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_ENABLEDICHANGEDEVENT_IN* pInParam,
OUT MMC_ENABLEDICHANGEDEVENT_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_DS401_API.h	
	GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_ENABLEDICHANGEDEVENT_IN** input data structure using the MMC_EnableDS401DIChangedEvent function.

pOutParam

Points to the **MMC_ENABLEDICHANGEDEVENT_OUT** output structure receiving information, as a result of calling the MMC_EnableDS401DIChangedEvent function.

Remarks

When enabled, any DS401 digital input event change is sent from the I/O module to the Maestro and then host server (if connected).

Scope

All

MMC_ENABLEDICHANGEDEVENT_IN Structure

```
typedef struct{
unsigned char ucDummy;
}MMC_ENABLEDICHANGEDEVENT_IN;
```

Parameters

ucDummy

Dummy data input. Any +ve character value.

MMC_ENABLEDICHANGEDEVENT_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_ENABLEDICHANGEDEVENT_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-107 describes the function block for MMC_EnableDS401DIChangedEvent as applied within the IEC 61131 programming.

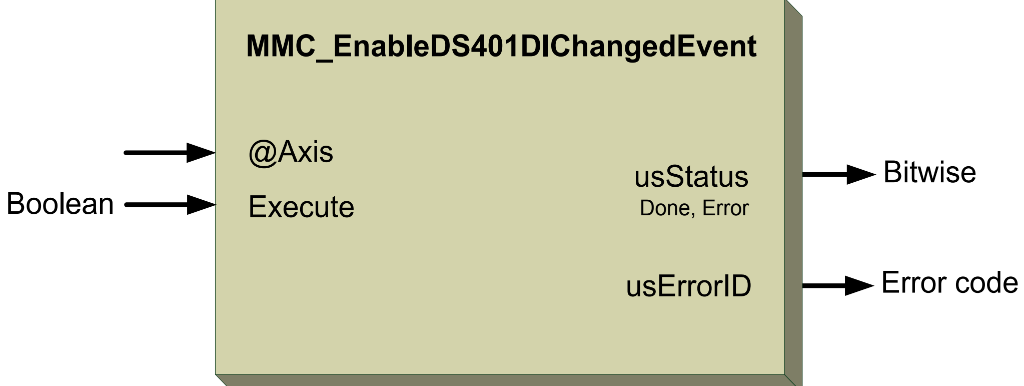


Figure 16-107: MMC_EnableDS401DIChangedEvent function block

16.12.10.2 Function Block Code Example

```
int rc;
MMC_ENABLEDICHANGEDEVENT_IN         stEnableDIChangeEv_in;
MMC_ENABLEDICHANGEDEVENT_OUT        stEnableDIChangeEv_out;
//
// Inserting the structure parameters:
stEnableDIChangeEv_in.ucDummy   = 1;     //Dummy data input
//
rc = MMC_EnableDS401DIChangedEvent (hConn, iAxisRef, &stEnableDIChangeEv_in,
&stEnableDIChangeEv_out);
if (rc != 0)
{
    HandleError();
}
```

16.12.11 MMC_ReadDS401DIGroup

Reads the DS-401 digital inputs of a group of 8 digital I/Os.

```
MMC_LIB_API int MMC_ReadDS401DIGroup(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READDIGROUP_IN* pInParam,
OUT MMC_READDIGROUP_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_DS401_API.h

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READDIGROUP_IN** input data structure using the MMC_ReadDIGroup function.

pOutParam

Points to the **MMC_READDIGROUP_OUT** output structure receiving information, as a result of calling the MMC_ReadDIGroup function.

Remarks

A group consists of 8 I/O connections with a possible maximum of 8 groups and therefore 64 I/O connections.

Scope

All

MMC_READDIGROUP_IN Structure

```
typedef struct{
unsigned char ucGroupIndex;
}MMC_READDIGROUP_IN;
```

Parameters

ucGroupIndex

Group index of 8 I/O’s up to a max of 64 I/O’s. +ve Integer (character) values of [9 onwards]

Note that It is possible to write to the lower 8 groups (64 bits) using the function MMC_WriteDS401DOutput that writes to the first 64 bits (long long variable) if they are valid. MMC_WriteDS401DOGroup only writes to the upper bits.

MMC_READDIGROUP_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_READDIGROUP_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-108 describes the function block for MMC_Read DS401DIGroup

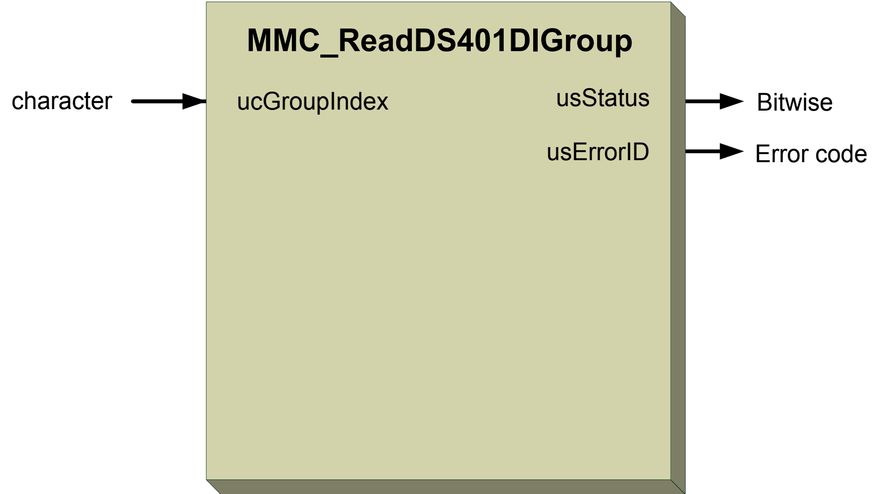


Figure 16-108: MMC_ReadDS401DIGroup function block

16.12.11.2 Function Block Code Example

```
int rc;
MMC_READDIGROUP_IN          stReadDIGroup_in;
MMC_READDIGROUP_OUT         stReadDIGroup_out;
//
// Inserting the structure parameters:
stReadDIGroup_in.ucGroupIndex   = 21;         //Group index
//
rc = MMC_ReadDS401DIGroup (hConn, iAxisRef, &stReadDIGroup_in, &stReadDIGroup_out);
printf("ErrId[%d]\n", (short)stReadDIGroup_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.12.12 MMC_ReadDS401DInput

Reads the DS-401 digital input of all 64 bit I/O’s in one action, increasing the communication speed proportionately versus reading 8 x groups of 8 I/O’s.

```
MMC_LIB_API int MMC_ReadDS401DInput(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_READDI_IN* pInParam,
OUT MMC_READDI_OUT* pOutParam
);
```

Motion Mode	NC - Supported	Distributed - Supported
Source	GMAS\includes\MMC_DS401_API.h GMAS Programming(IEC 61331 Program)\ElmoSingleAxis	

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_READDI_IN** input data structure using the MMC_ReadDS401DInput function.

pOutParam

Points to the **MMC_READDI_OUT** output structure receiving information, as a result of calling the MMC_ReadDS401DInput function.

Remarks

None

Scope

All

MMC_READDI_IN Structure

```
typedef struct{
unsigned char dummy;
}MMC_READDI_IN;
```

Parameters

dummy

Dummy data input. Any +ve character value.

MMC_READDI_OUT Structure

```
typedef struct{
#ifdef WIN32
unsigned __int64 ulliDI;
#else
unsigned long long int ulliDI;
#endif
unsigned short usStatus;
short sErrorID;
}MMC_READDI_OUT;
```

Parameters

__int64 ulliDI or ulliDI

If function is defined for WIN32 then use *__int64 ulliDI*, else use *ulliDI*. Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-109 describes the function block for MMC_ReadDS401DInput as applied within the IEC 61131 programming.

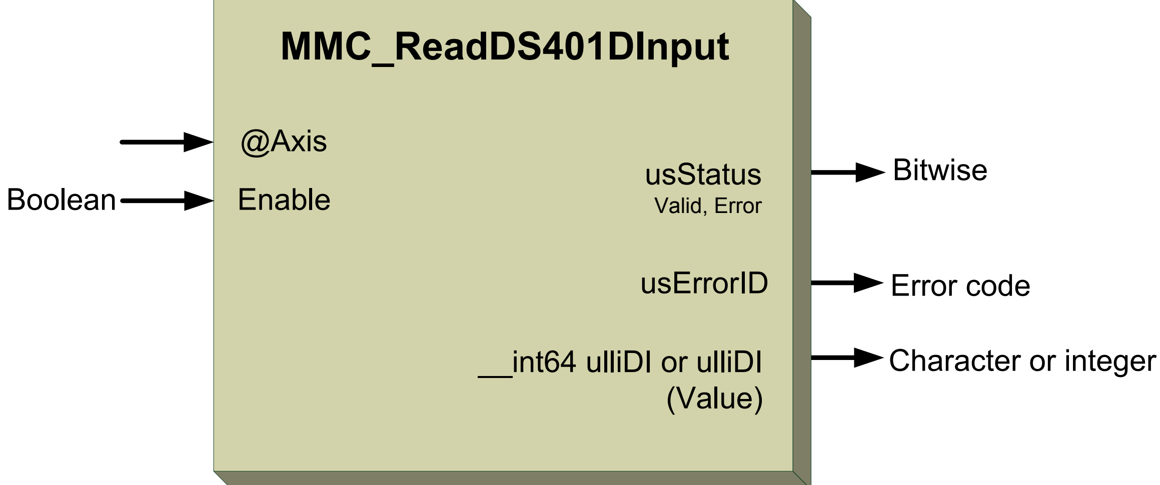


Figure 16-109: MMC_ReadDS401DInput function block

16.12.12.2 Function Block Code Example

```
int rc;
MMC_READDI_IN      stReadDI_in;
MMC_READDI_OUT     stReadDI_out;
//
// Inserting the structure parameters:
stReadDI_in.dummy      = 1;      //dummy input
//
rc = MMC_ReadDS401DInput (hConn, iAxisRef, &stReadDI_in, &stReadDI_out);
printf("DS-401 Input Status[%ld] ErrId[%d]\n", (long int)stReadDI_out.ulliDI,
(short)stReadDI_out.sErrorID);
if (rc != 0)
{
    HandleError();
}
```

16.12.13 MMC_WriteDS401DOGroup

Writes the DS-401 digital outputs of a group of 8 I/O's to the Maestro.

```
MMC_LIB_API int MMC_WriteDS401DOGroup(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEDOGROUP_IN* pInParam,
OUT MMC_WRITEDOGROUP_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_DS401_API.h

GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEDOGROUP_IN** input data structure using the MMC_WriteDS401DOGroup function.

pOutParam

Points to the **MMC_WRITEDOGROUP_OUT** output structure receiving information, as a result of calling the MMC_WriteDS401DOGroup function.

Remarks

A group consists of 8 I/O connections with a possible maximum of 8 groups and therefore 64 I/O connections.

Scope

All

MMC_WRITEDOGROUP_IN Structure

```
typedef struct{
unsigned char ucGroupIndex;
unsigned char ucVal;
}MMC_WRITEDOGROUP_IN;
```

Parameters

ucGroupIndex

Group index of 8 I/O's up to a max of 64 I/O's. +ve Integer (character) values of [9 onwards]

Note that It is possible to write to the lower 8 groups (64 bits) using the function MMC_WriteDS401DOutput that writes to the first 64 bits (long long variable) if they are valid. MMC_WriteDS401DOGroup only writes to the upper bits.

ucVal

Digital output value of the 0 – 8 bit data in a group, ranging from 0 – 255. Any +ve character value.

MMC_WRITEDOGROUP_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEDOGROUP_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-110 describes the function block for MMC_WriteDS401DOGroup as applied within the IEC 61131 programming.

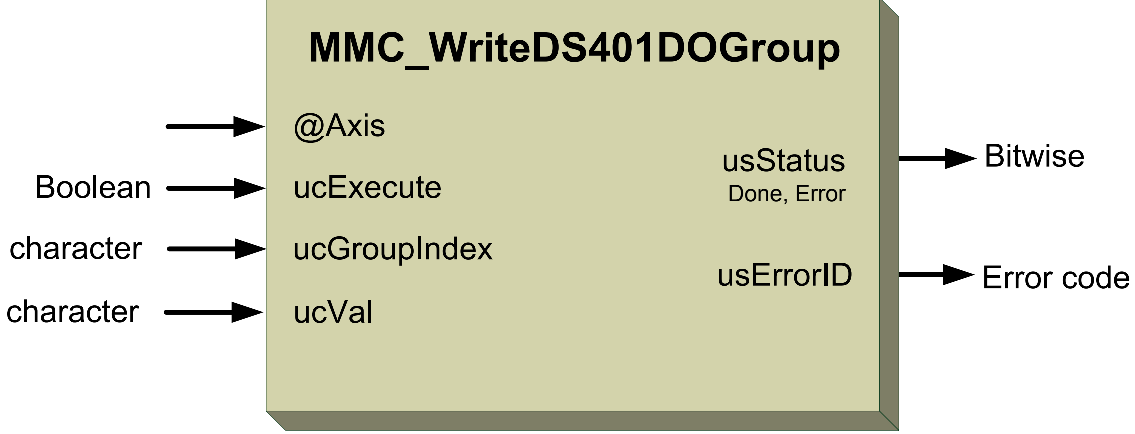


Figure 16-110: MMC_WriteDS401DOGroup function block

16.12.13.2 Function Block Code Example

```
int rc;
MMC_WRITEDOGROUP_IN         stWriteDOGroup_in;
MMC_WRITEDOGROUP_OUT        stWriteDOGroup_out;
//
// Inserting the structure parameters:
stWriteDOGroup_in.ucGroupIndex = 10;         //Index of the group axes
stWriteDOGroup_in.ucVal         = 9;         //Digital output value
//
rc = MMC_WriteDS401DOGroup (hConn, iAxisRef, &stWriteDOGroup_in, &stWriteDOGroup_out);
if (rc != 0)
{
    HandleError();
}
```

16.12.14 MMC_WriteDS401DOutput

Writes to all of the DS-401 digital outputs assigned to TPDO1 at once, up to 64 bit I/O's in one action, increasing the communication speed proportionately versus writing to 8 x groups of 8 I/O's.

```
MMC_LIB_API int MMC_WriteDS401DOutput(
IN MMC_CONNECT_HNDL hConn,
IN MMC_AXIS_REF_HNDL hAxisRef,
IN MMC_WRITEDO_IN* pInParam,
OUT MMC_WRITEDO_OUT* pOutParam
);
```

Motion Mode NC - Supported Distributed - Supported

Source GMAS\includes\MMC_DS401_API.h
GMAS Programming(IEC 61331 Program)\ElmoSingleAxis

Function Parameters

hConn

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

hAxisRef

Axis/group reference handle type returned by GetAxisRef command

pInParam

Points to the **MMC_WRITEDO_IN** input data structure using the MMC_WriteDS401DOutput function.

pOutParam

Points to the **MMC_WRITEDO_OUT** output structure receiving information, as a result of calling the MMC_WriteDS401DOutput function.

Remarks

None

Scope

All

MMC_WRITEDO_IN Structure

```
typedef struct{
#ifdef WIN32
unsigned __int64 ulliDO;
#else
unsigned long long int ulliDO;
#endif
}MMC_WRITEDO_IN;
```

Parameters

__int64 ulliDO or ulliDO

If function is defined for WIN32 then use *__int64 ulliDO*, else use *ulliDO*. Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

MMC_WRITEDO_OUT Structure

```
typedef struct{
unsigned short usStatus;
short sErrorID;
}MMC_WRITEDO_OUT;
```

Parameters

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Figure 16-111 describes the function block for MMC_WriteDS401DOutput as applied within the IEC 61131 programming.

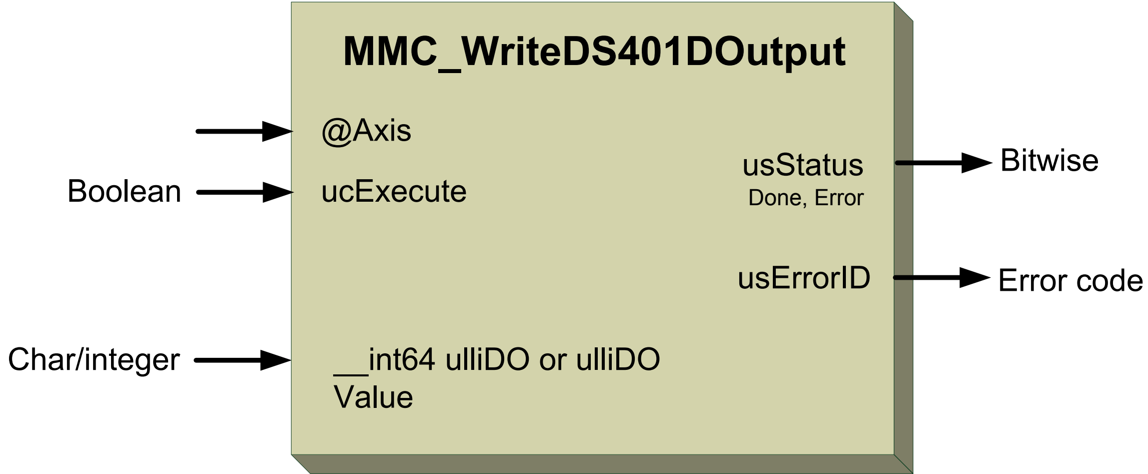


Figure 16-111: MMC_WriteDS401DOutput function block

16.12.14.2 Function Block Code Example

```
int rc;
MMC_WRITEDO_IN          stWriteDO_in;
MMC_WRITEDO_OUT         stWriteDO_out;
//
// Inserting the structure parameters:
stWriteDO_in.ulliDO    = 1;      //Value to write to digital outputs
//
rc = MMC_WriteDS401DOutput (hConn, iAxisRef, &stWriteDO_in, &stWriteDO_out);
if (rc != 0)
{
    HandleError();
}
```

Chapter 17: Programming in C++

17.1 Introduction

C++ began as enhancements to C, first adding classes, then virtual functions, operator overloading, multiple inheritance, templates, and exception handling among other features. The class has both an interface and a structure. The interface describes how to interact with the class and its instances using methods, while the structure describes how the data is partitioned into attributes within an instance. A class may also have a representation (metaobject) at run time, which provides run time support for manipulating the class-related metadata.

While Elmo has created a series of sophisticated function blocks in C, in C++ these functions are wrapped, thereby dividing the C functions into better-structured categories (classes), where these classes have methods (similar to the C functions) and parameters.

C++ has the following advantages:

- Overloading functions. User can call for example; MoveAbsolute(pos) and also MoveAbsolute(pos,vel)
- Try-catch error handling mechanisms. Instead of using the 'if' condition continuously, per function, there can be a single error handler per function, in place of numerous 'ifs' that perform the same error handling.
- Eclipse Drop-Down function list. The relevant function can be chosen from the dropdown list.
- It is not necessary to learn the parameters of each function. Once the default parameters are set, only position, and velocity, for instance, need be updated. However, it will be necessary to copy parameters, as function blocks do not all have same values.

Each Class has a function using the name *SetDefaultValues*, which receives a pointer to a structure, for example, with all motion values of the single axis. After calling this function, **all** motion parameters are copied to a local buffer in the class. After calling, the overloaded functions may be called.

17.1.1 CMMCException

This return in all C++ wrapper functions is based on this CMMCException, which normally returns 0 on success, and throws an exception on error.

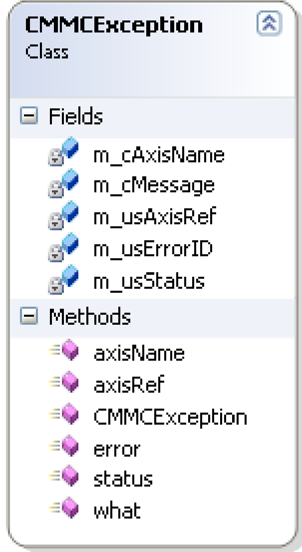


Figure 17-1 Fields and methods of the CMMCException class

When an error is encountered, the following information is returned:

Function Name	Structure name	
Axis reference	Error ID	Status of the axis

17.2 The CMMCNode class

The class CMMCNode wraps the host communication functions detailed in the section **16.2 Host Communication on page 1212**. The class CMMCNode retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

CMMCAxis
Class

Fields
- m_strName
- m_uiConnHndl
- m_usAxisRef

Methods
- ~CMMCAxis
- CMMCAxis
- GetAxisByName
- GetConnHndl
- GetName
- GetRef
- GetStatusRegister
- GetSystemCounter
- InitAxisData
- SetConnHndl
- SetDefaultManufacturerParameters
- SetName
- SetRef

_unnamed_enum_0136_1
Enum
- PDO_NUM_3
- PDO_NUM_4

PDO_NUMBER_ENUM : _unnamed_enum_0136_1
Typedef

_unnamed_enum_0136_2
Enum
- PDO_PARAM_REG
- PDO_PARAM_USE

PDO_PARAM_TYPE_ENUM : _unnamed_enum_0136_2
Typedef

public

CMMCNode
Class
→ CMMCAxis

Methods
- ~CMMCNode
- CMMCNode (+ 1 overload)
- ConfigPDOEventMode
- EthercatPIVarInfo
- EthercatReadMemoryRange
- EthercatReadPIVar (+ 14 overloads)
- EthercatWriteMemoryRange
- EthercatWritePIVar (+ 14 overloads)
- GetAxisError
- GetBoolParameter
- GetParameter
- GetPDOInfo
- PDOGeneralRead
- PDOGeneralWrite (+ 1 overload)
- ReadStatus (+ 1 overload)
- Reset
- RetreiveSdoUploadAsync
- SendSdoCmd
- SendSdoDownload
- SendSdoUpload
- SendSdoUploadAsync
- SetBoolParameter
- SetParameter

public

CMMCMotionAxis
Class
→ CMMCNode

Methods
- ~CMMCMotionAxis
- AppendPVTPoints (+ 2 overloads)
- CMMCMotionAxis (+ 1 overload)
- DisableMotionEndedEvent
- Dwell
- EnableMotionEndedEvent
- GetFbDepth
- GetPVTTableIndex
- InitPVTTable
- KillRepetitive
- LoadPVTTableFromFile
- MovePVT
- ReadGroupOfParameters
- UnloadPVTTable
- WaitUntilConditionFB
- WriteGroupOfParameters
- WriteGroupOfParametersImmediate
- WriteGroupOfParametersQueued

Figure 17-2 Fields and methods of the CMMCNode class

The detailed class view shown in **Figure 17-2** describes the fields and methods associated with the CMMCNode class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.2.1 CMMCNode Class Functions Code Example 1

```
/*
============================================================================
Collection of Gmas API functions (Set #2)
Examples for document: "G-MAS Administrative and Motion API.pdf".
09Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"
#define STAR_DIM                       2


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection          gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCHostComm            cHost ;

     CMMCSingleAxis          AxisA,    AxisB;
     unsigned short               AxisARef, AxisBRef;

     CMMCGroupAxis            Group;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

    char  * delimit = "============================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroMbusfunc(int);
     void mBusReadWriteHoldReg(void);
     void mBusReadWriteCoil(void);
     void mBusReadWriteInput(void);
     void moveAxisAAndRepActualParamOnMbusPos(void);
     void      RepAxisAActualParamOnMbus(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
```

```
     int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus);
     void      EndMotionEventCB(unsigned short usAxisRef);
     void ModbusWrite_Received();
     void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
     int       mapConfigPdo(void);


/*================================= Administration functions STR ===================================*/
int       main(int)
// ==============
{
     int  trace = 1;

     printf("\n %s", delimit);
     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

     try
     {
          SnroMoveAbsolute(trace++);
          SnroEnableDisableMotionEndedEvent(trace++);
          SnroDepthName(trace++);

          SnroMbusfunc(trace++);
     }
     catch (CMMCException excp)
     {
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          exit(0);
     }

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


//   15.11.3.  CMMCNode::ReadStatus     1374
int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//=====================================================================
{
               int end_of = 0;
               int iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
          end_of = 1;
/* Read Axis Status command server for specific Axis */
          ulState = sng_axis->ReadStatus();
          if (!(ulState & break_state))
          {
               end_of = 0;

               WAIT_SLEEP_MILLI(20)
          }
     }

//        MMC_SHOWNODESTAT_IN  showin;
//        MMC_SHOWNODESTAT_OUT showout;
//        MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

     return 0;
}


// 16.6.3.     CMMCConnection::ConnectIPCEx  1241
// 16.6.11.    CMMCConnection::CallbackFunc  1248
```

```
void initAdminSingleAxis(void)
// =============================
{
     int iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;



/* CallbackFunc in ConnectIPCEx call if there    */
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */

/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk     = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection     = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode    = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute      = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


//   16.4.19.  CMMCGroupAxis::AddAxisToGroup 1212
void initAdminMultiAxis()
// ========================
{
//   MMC_CONNECT_HNDL         ComHndl;
//   CMMCSingleAxis           AxisA,    AxisB;
//   CMMCGroupAxis            Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// ============================
{
     MMC_CloseConnection(ComHndl) ;
}


//   16.4.3.   CMMCGroupAxis::RemoveAxisFromGroup 1197
     void endAdminMultiAxis(void)
// ================================
{

//   CMMCGroupAxis             Group;

     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
```

```
    Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================ Administration functions END ====================================*/


/*================================ Scenario functions STR ==========================================*/
void SnroMoveAbsolute(int trace)
// ===============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    MoveAbsoluteMoves();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


//   16.6.12.  CMMCConnection::RegisterEventCallback  1249
void SnroEnableDisableMotionEndedEvent(int trace)
// ================================================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    EnableDisableMotionEndedEvent();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
    endAdminSingleAxis();

    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


void SnroDepthName(int trace)
// ============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    initAdminMultiAxis();

   AxisA.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

   AxisB.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

    Group.GroupEnable();

    DepthName();

    Group.GroupDisable();

    AxisB.PowerOff(MC_BUFFERED_MODE);
    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
```

```
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

     endAdminMultiAxis();
     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


//   16.8.5.   cHost::MbusIsRunning     1258
//   16.8.1.   cHost::MbusStartServer   1255
//   16.8.2.   cHost::MbusStopServer    1256
void SnroMbusfunc(int trace)
// ============================
{
static    bool      mBusServerState;


     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
/* Starting MBus server           */
/* map read/write user req. to Mbus    */
/* area & responsible to Mbus commu.*/
     mBusServerState = cHost.MbusIsRunning(ComHndl);
     if (mBusServerState)
     {
          printf("\n ... ModBus Server is already running ");
     }
     else
     {
          printf("\n ... ModBus Server is NOT running - activate it ");
     }
/* Call to StartServer even if it   */
/* already running, it not start    */
/* the server because it alrady so  */
/* but for initialize init struct */
     cHost.MbusStartServer(ComHndl, 1) ;
                              /* Mode Bus functions examples.        */
     mBusReadWriteHoldReg();
/* Move Axis A and report it poss.  */
/* on Mbus.*/
     moveAxisAAndRepActualParamOnMbusPos();
     mBusReadWriteCoil();
     mBusReadWriteInput();

/* If I was the one who activate the   */
/* Mbus server I stop it when I finish.*/
     if (! mBusServerState)
     {
/* Stop MBus server */
          cHost.MbusStopServer();
          printf("\n ... Stop ModBus Server ");
     }
     else
     {
          printf("\n ... Left ModBus Server running ");
     }

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
/*================================= Scenario functions END ==========================================*/



/*================================= Example functions STR ===========================================*/

void EnableDisableMotionEndedEvent(void)
// ========================================
{
     int  loopInd;
```

```
printf("\n Function: %s:", __func__);
     for (loopInd = 0; loopInd < 2; loopInd++)
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}

void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

// iVal3 = AxisA.GetAxisByName("A01"); /* It case sensitive - Not define - exception... */

     iVal1 = Group.GetGroupAxisByName("v01");    /* Expected 256 */
/*
 *   iVal2 = Group.GetGroupAxisByName("v02");
 */
}


void MoveAbsoluteMoves(void)
// ===========================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:   */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0 */
/* update default speed to 5000000     */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters       */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity */
/* v=100000    which become the new def V    */
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0  */
                              /* new def v=80000                          */
     AxisA.MoveAbsolute(310000.0, 80000.0);
```

```
/* Move abs to: 400000, with parameters:    */
                              /*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=            */
/* MC_BUFFERED_MODE (def)    */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
                              /* Move abs to 350000 with parameters from  */
                              /* above command which become the default:  */
                              /*        Speed 500000, Acc=1000000              */
                              /*        Dec=1500000, Jerk=20000000,       */
                              /*        buffer mode=MC_BUFFERED_MODE (def)   */
     AxisA.MoveAbsolute(350000);
}


//   16.6.11.  CMMCConnection::CallbackFunc  1248
int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ==================================================================================
{
//   int  ind;

    printf("\n *********** STR Func: %s *********** ", __func__);

//    printf("\n");
//    for (ind=0; ind < recvBufferSize; ind++)
//    {
//        printf("   recvBuffer[%d]=%d", ind, recvBuffer[ind]);
//    }
//    printf("\n");

/* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
          printf("\n Underflow event received ");
          break ;
     case MODBUS_WRITE_EVT:
          printf("\n ModBus Write event received ");
          break ;
     case TOUCH_PROBE_ENDED_EVT:
          printf("\n Touch Probe event received ");
          break ;
     default:
          printf("\n Default.... Whatever arrived event received ");
          break;
     }

    printf("\n *********** END Func: %s *********** ", __func__);
     fflush(stdout); fflush(stderr);

     return 1 ;
```

```
}


//
// Needs registeration, E.g:  CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);
//
int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
//
=========================================================================================================
{
     printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
             msg, usAxisRef, sErrorID, usStatus);
     fflush(stdout); fflush(stderr);

     MMC_CloseConnection(uiConnHndl);
     exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
     printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
     printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
     fflush(stdout); fflush(stderr);
}

/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ==========================
{
     printf("\n %s Received ", __func__) ;
     fflush(stdout); fflush(stderr);
}

/* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ==================================================================
{
     printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
     fflush(stdout); fflush(stderr);
}


#define   MODBUS_UPDATE_START_INDEX              0
#define   MODBUS_UPDATE_CNT                 4
                              /* WRITE INDEXS extend along two indexs */
#define MODBUS_STR_REP_ACT_VAL_INDX         (MODBUS_UPDATE_START_INDEX+MODBUS_UPDATE_CNT)
#define   MBUS_WAIT_FOR_USER                     30000

//   16.8.3.   cHost::MbusReadHoldingRegisterTable         1257
//   16.8.4.   cHost::MbusWriteHoldingRegisterTable   1258
void mBusReadWriteHoldReg(void)
// ==============================
{
          int       startRef;
          int       refCnt;
          int            loopCount = 0;

     MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN     mbus_write_in;
     MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT     mbus_read_out;


     printf("\n\n Function: %s: ", __func__);

     memset(mbus_write_in.regArr,0x0, 250) ;

     startRef = MODBUS_UPDATE_START_INDEX;
     refCnt    = MODBUS_UPDATE_CNT;
     mbus_write_in.startRef        = startRef;
     mbus_write_in.refCnt          = refCnt;
     mbus_write_in.regArr[0]  = 0xf0f0;
```

```
     mbus_write_in.regArr[1]  = 0x0f0f;
     mbus_write_in.regArr[2]  = 0x5a5a;
     mbus_write_in.regArr[3]  = 0xa5a5;


     printf("\n ... You can activate the EAS and look on EAS Mbus reg 1-4 ");

     printf("\n ... Going deal with Mbus H.Reg. EAS ind 1-4 ");
     loopCount = 0;
     do
     {
          loopCount++;
/* write into mBus Holding register */
/* data on reg. ind 0-3           */
          cHost.MbusWriteHoldingRegisterTable(mbus_write_in);
          printf("\n ... Mbus Write  H.Reg ind 0-3: %8x %8x %8x %8x ",
                                          (int)mbus_write_in.regArr[0],
                                          (int)mbus_write_in.regArr[1],
                                          (int)mbus_write_in.regArr[2],
                                          (int)mbus_write_in.regArr[3]);
          printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);

          WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
/* read mbus H.Reg start from           */
/* location startRef along refCnt reg*/

          cHost.MbusReadHoldingRegisterTable(startRef, refCnt, mbus_read_out);
          printf("\n ... Mbus Readed H.Reg ind 0-3: %8x %8x %8x %8x ",
                                           (int)mbus_read_out.regArr[0],
                                           (int)mbus_read_out.regArr[1],
                                           (int)mbus_read_out.regArr[2],
                                           (int)mbus_read_out.regArr[3]);

          mbus_write_in.regArr[0] = ~(mbus_read_out.regArr[0]);
          mbus_write_in.regArr[1] = ~(mbus_read_out.regArr[1]);
          mbus_write_in.regArr[2] = ~(mbus_read_out.regArr[2]);
          mbus_write_in.regArr[3] = ~(mbus_read_out.regArr[3]);

     } while (loopCount < 3);
}


//   16.8.6.   cHost::MbusReadCoilsTable    1259
//   16.8.7.   cHost::MbusWriteCoilsTable    1260
void mBusReadWriteCoil(void)
// ============================
{
          int       startRef;
          int       refCnt;

          int            loopCount = 0;

     MMC_MODBUSWRITECOILS_IN       stInParams_Coil;
     MMC_MODBUSREADCOILS_OUT       stOutParams_Coil;


     printf("\n\n Function: %s: ", __func__);

     startRef = MODBUS_UPDATE_START_INDEX;
     refCnt    = MODBUS_UPDATE_CNT;
     stInParams_Coil.startRef= startRef;
     stInParams_Coil.refCnt    = refCnt;
     stInParams_Coil.coilsArr[0] = (char)0xf0;
     stInParams_Coil.coilsArr[1] = (char)0x0f;
     stInParams_Coil.coilsArr[2] = (char)0x5a;
     stInParams_Coil.coilsArr[3] = (char)0xa5;

     printf("\n ... Going deal with Mbus Coils EAS ind 1-4 ");
     loopCount = 0;
     do
     {
          loopCount++;
/* write into mBus Coil register */
/* data on reg. ind 0-3            */
```

```
        cHost.MbusWriteCoilsTable(stInParams_Coil);
        printf("\n ... Mbus Write  Coil ind 0-3: %8x %8x %8x %8x ",
                                           (int)stInParams_Coil.coilsArr[0],
                                           (int)stInParams_Coil.coilsArr[1],
                                           (int)stInParams_Coil.coilsArr[2],
                                           (int)stInParams_Coil.coilsArr[3]);
        printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
        WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
/* read mbus Coil start from       */
/* location startRef along refCnt reg*/
        cHost.MbusReadCoilsTable(stInParams_Coil.startRef, stInParams_Coil.refCnt, stOutParams_Coil);
        printf("\n ... Mbus Readed Coil ind 0-3: %8x %8x %8x %8x ",
                                           (int)stOutParams_Coil.coilsArr[0],
                                           (int)stOutParams_Coil.coilsArr[1],
                                           (int)stOutParams_Coil.coilsArr[2],
                                           (int)stOutParams_Coil.coilsArr[3]);

        stInParams_Coil.coilsArr[0] = ~(stOutParams_Coil.coilsArr[0]);
        stInParams_Coil.coilsArr[1] = ~(stOutParams_Coil.coilsArr[1]);
        stInParams_Coil.coilsArr[2] = ~(stOutParams_Coil.coilsArr[2]);
        stInParams_Coil.coilsArr[3] = ~(stOutParams_Coil.coilsArr[3]);

    } while (loopCount < 3);
}


//   16.8.8.   cHost::MbusReadInputsTable   1261
void mBusReadWriteInput(void)
// =============================
{
    int       startRef;
    int       refCnt;

    int            loopCount = 0;

    MMC_MODBUSREADINPUTS_OUT stOutParams_Input;


    printf("\n\n Function: %s: ", __func__);

    printf("\n ... Going deal with Mbus Inputs EAS ind 1-4 ");
    startRef= MODBUS_UPDATE_START_INDEX;
    refCnt    = MODBUS_UPDATE_CNT;
    loopCount = 0;
    do
    {
        loopCount++;
        cHost.MbusReadInputsTable(startRef, refCnt, stOutParams_Input);
        printf("\n ... Mbus Readed inputs ind 0-3: %8x %8x %8x %8x ",
                  (int)stOutParams_Input.inputsArr[0], (int)stOutParams_Input.inputsArr[1],
(int)stOutParams_Input.inputsArr[2], (int)stOutParams_Input.inputsArr[3]);

        printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
        WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
    } while (loopCount < 3);
}


#define   UL_TO_MBUS_STRUCT(modbus_write_val, aux_ul_p, mod_bus_idx)              \
    {                                                                             \
            aux_ul_p = (unsigned long *)(& mbus_write_in.regArr[mod_bus_idx]);   \
            * aux_ul_p = modbus_write_val;                                        \
    }                                                                             \
//   16.3.20.  CMMCSingleAxis::GetActualPosition      1164
//   16.3.21.  CMMCSingleAxis::GetActualVelocity      1164
//   16.3.22.  CMMCSingleAxis::GetActualTorque        1165
//   16.6.7.   CMMCSingleAxis::GetGlobalBoolParameter 1245
```

```
void RepAxisAActualParamOnMbus(void)
// ================================
{
     MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN     mbus_write_in; /* Auxiliary var for write to ModBus */
     unsigned  long*     ul_p;
     unsigned  long ul_val;
     double    dActul;

     dActul = AxisA.GetActualPosition();
     ul_val = (unsigned long)dActul; /* Put value into ModBus write array use 2 index poss (0 & 1).
     */
     UL_TO_MBUS_STRUCT(ul_val, ul_p, 0);

/* Remmber about mapping & config !!! */
/* if the motor connected above ehercat it needs apropirate */
/* config setting (EAS). */
     dActul = AxisA.GetActualVelocity();
     ul_val = (unsigned long)dActul;
                         /* Put value into ModBus write array use 2 index poss (2 & 3).  */
     UL_TO_MBUS_STRUCT(ul_val, ul_p, 2);


/* Remmber about mapping & config (see GetActualVelocity) !!!   */
     dActul = AxisA.GetActualTorque();
     ul_val = (unsigned long)dActul;
/* Put value into ModBus write array use 2 index poss (4 & 5).  */
     UL_TO_MBUS_STRUCT(ul_val, ul_p, 4);


/* Index of modbus H.reg. for start write actal values. */
     mbus_write_in.startRef       = MODBUS_STR_REP_ACT_VAL_INDX;
/* Number of indexes to write - each long extend along     */
/* 2 indexs (2 for Pos, 2 for Vel, 2 for Torq              */
     mbus_write_in.refCnt = 6;
     cHost.MbusWriteHoldingRegisterTable(mbus_write_in) ;

     return ;
}


#define   eCOMM_TYPE_ETHERCAT      1
#define   eCOMM_TYPE_CAN           2
/* Move Axis A and report it Actual */
/* parameters on Mbus.                  */
void moveAxisAAndRepActualParamOnMbusPos(void)
// ==============================================
{
               int ind,
                        rt;
     unsigned int   uiState,
                        uiDoneFlg;

     printf("\n\n Function: %s: ", __func__);

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


/* Connection type - CAN/EtherCAT */
     rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);
     if (rt == eCOMM_TYPE_ETHERCAT)
     {
          printf("\n --- Motor connection is via ETHERCAT rembber map & config for get actul param
(Vel. Torq.) ");
     }
     else if (rt == eCOMM_TYPE_CAN)
     {
          printf("\n --- Motor connection is via CAN ");
          if ( mapConfigPdo() )
          {
               printf("\n --- ERROR FAIL IN MAP PDO FOR CAN  !!! ");
          }
     }
     else
```

```
     {
          printf("\n --- ERROR UNKNOWN motor connection  !!! ");
     }

     printf("\n ... Start rep AxisA pos on Mbus");
     for(ind=0; ind<3; ind++)
     {
/* Move abs to: 400000, with parameters:         */
/*            Speed 300000, Acc=500000           */
/* Dec=1500000, Jerk=20000000, buffer mode=      */
/* MC_BUFFERED_MODE (def)         */
          AxisA.MoveAbsolute(400000*ind, 300000, 500000, 1500000, 20000000);
          do
          {
                              /* Write AxisA pos. to Modbus H.Reg */
               RepAxisAActualParamOnMbus();
               uiState = AxisA.ReadStatus();
               uiDoneFlg = uiState & NC_AXIS_STAND_STILL_MASK;
               if ( ! uiDoneFlg)
               {
                    WAIT_SLEEP_MILLI(40)
               }
          } while(! uiDoneFlg);
     }
     printf("\n ... End rep AxisA pos on Mbus");

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
}


int  mapConfigPdo(void)
// ===================
{
     int  rc;
     unsigned short                AxisARef;


     MMC_CONFIGREGULARPARAMEVENTPDO3_IN CfgReg3In;
    MMC_CONFIGREGULARPARAMEVENTPDO3_OUT CfgReg3Out;

    CfgReg3In.ucEventGroup    = NC_COMM_EVENT_GROUP6;
    CfgReg3In.ucPDOCommParam= PDO_COM_PARAM_SYNC;

     AxisARef = AxisA.GetRef();

    rc = MMC_CfgRegParamEvPDO3Cmd(ComHndl, AxisARef, &CfgReg3In, &CfgReg3Out);
    if(rc != 0)
    {
     printf("\n **** config error %d \n",CfgReg3Out.sErrorID);
/* ... goto lbl_motor_off; */
    }

     return (rc);
}

/*================================= Example functions END ==========================================*/
```

17.2.2 ConfigPDOEventMode

Refer to the sections **16.5.10 - 16.5.11** for details of the description, scope, and motion mode.

```
void ConfigPDOEventMode(
unsigned char ucPDOEventMode
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

ucPDOEventMode

PDO event mode. This enumerator has the following values:

MC_PDO_EVENT_NO_NOTIF = 0,

MC_PDO_EVENT_CYCLIC_NOTIF,

MC_PDO_EVENT_IMMIDIATE_NOTIF

of which the default event type is MC_PDO_EVENT_NO_NOTIF. When using MC_PDO_EVENT_IMMIDIATE_NOTIF mode, no endian swap is created on the data.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

17.2.3 EtherCATPIVarInfo

This function returns the detailed information about a required Processing Image variable, reading the variable according to its index. Refer to the sections **9.7.27 MMC_GetPIVarInfo** for details of the description, scope, and motion mode.

```
void EthercatPIVarInfo(
unsigned short usPIVarIndex,
unsigned char ucDirection,
NC_PI_ENTRY &VarInfo
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

usPIVarIndex

The required PI variable index to be read from the processed image. Any +ve integer values.

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

NC_PI_ENTRY &VarInfo

```
typedef struct pientry{
unsigned int uiBitSize;
unsigned int uiBitOffset;
unsigned short usCanOpenIndex;
unsigned char ucCanOpenSubIndex;
unsigned char ucVarType;
char pAliasing[PI_ALIASING_LENGTH];
unsigned char ucPadding;
} NC_PI_ENTRY;
```

uiBitSize

The size of the Bit. Any +ve integer values

uiBitOffset

The Offset of the Bit. Any +ve integer values.

usCanOpenIndex

CANopen index. 2 Byte +ve values

ucCanOpenSubIndex

CANopen subindex. +ve character values

ucVarType

Variable type according to the following enumerator values:

ePI_BOOL	= 0,
ePI_SIGNED_CHAR	= 1,
ePI_UNSIGNED_CHAR	= 2,
ePI_SIGNED_SHORT	= 3,
ePI_UNSIGNED_SHORT	= 4,
ePI_SIGNED_INT	= 5,
ePI_UNSIGNED_INT	= 6,
ePI_SIGNED_LONG_LONG	= 7,
ePI_UNSIGNED_LONG_LONG	= 8,
ePI_FLOAT	= 9,
ePI_DOUBLE	= 10,
ePI_BITWISE	= 11,
ePI_8MULTIPLE	= 12,

pAliasing [PI_ALIASING_LENGTH]

The alias name of the variable with a maximum name length of PI_ALIASING_LENGTH characters complemented with the IO name.

ucPadding

Alignment padding of data. Unsigned +ve character values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

17.2.4 EtherCATReadMemoryRange

Describes the read memory range for the EtherCAT process image. Refer to the section **8.1.48 MMC_ReadMemoryRange** for details of the description, scope, and motion mode.

```
void EthercatReadMemoryRange(
unsigned short usRegAddr,
unsigned char ucLength,
unsigned char pData[ETHERCAT_MEMORY_READ_MAX_SIZE]
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

usRegAddr

Registry address of the EtherCAT memory for the slave.

ucLength

Length of the label string. Length with the precursor format of [Metronome command]##[label]. Any +ve character values.

pData[ETHERCAT_MEMORY_READ_MAX_SIZE]

String Data with the precursor format of [Metronome command]##[label]. Any +ve character values with a maximum length of 80 bytes

[ETHERCAT_MEMORY_READ_MAX_SIZE] is the array EtherCAT memory read max size limited to 1400 bytes.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

17.2.5 EthercatReadPIVar

This function reads a Processing Image input\output variable according to its index. The variable data may be Boolean, Signed/Unsigned/Short Character, Short, Unsigned/Signed Integer, Float, or Double. Refer to the sections **9.7.1 MMC_ReadPIVarBOOL - 9.7.13 MMC_ReadLargePIVarRaw** for details of the description, scope, and motion mode.

```
void EthercatReadPIVar(
unsigned short usIndex,
unsigned char ucDirection,
unsigned char ucByteLength
[unsigned short usByteLength]
[bool &bData]
[signed char &cData]
[unsigned char &ucData]
[unsigned short &usData]
[short &sData]
[unsigned int &uiData]
[int &iData]
[float &fData]
[double &dbData]
unsigned char pRawData[PI_REG_VAR_SIZE]
[unsigned char pRawData[PI_LARGE_VAR_SIZE]]
[#ifdef WIN32 unsigned __int64 &ullData, __int64 &llData,
#else unsigned long long &ullData, long long &llData]
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

usIndex

COB index. Any +ve integer values (2 bytes).

ucDirection

This variable has two possible PI directions, output or input according to the enumerator values:

ePI_INPUT = 0,

ePI_OUTPUT = 1

unsigned char ucByteLength
[unsigned short usByteLength]

Length of the RAW data including the BitStart offset. The BitStart offset is the PI variable bit offset Modulo 8, e.g. Bit Offset = 25 with the BitStart = 1. The Byte length is (BitStart + Bit size) Modulo 8.

[bool &bData]
[signed char &cData]
[unsigned char &ucData]
[unsigned short &usData]
[short &sData]
[unsigned int &uiData]
[int &iData]
[float &fData]
[double &dbData]
unsigned char pRawData[PI_REG_VAR_SIZE]
[unsigned char pRawData[PI_LARGE_VAR_SIZE]]
[#ifdef WIN32 unsigned __int64 &ullData, __int64 &llData,
#else unsigned long long &ullData, long long &llData]

bool &bData signed char &cData unsigned char &ucData unsigned short &usData short &sData int &uiData int &iData float &fData double &dbData	The boolean, character, unsigned character, unsigned short, short, usigned integer, integer,float, or double parameter requested from the processed image.
pRawData[PI_REG_VAR_SIZE] pRawData[PI_LARGE_VAR_SIZE]	The RAW parameter requested from the processed image. The byte size of the RAW data is (BitStart + Bit size) Modulo 8, which is inserted as the ucByteLength. Dependant on the variable type array PI_REG_VAR_SIZE which is a maximum of 4, or the variable type array PI_LARGE_VAR_SIZE which is a maximum of 1400.
unsigned __int64 &ullData, __int64 &llData, unsigned long long &ullData, long long &llData	The long long parameter requested from the processed image. If the function is defined for WIN32 then use *__int64 ullData*, else use *long long ullData*. Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

17.2.6 EtherCATWriteMemoryRange

Describes the write memory range for the EtherCAT process image. Refer to the section **8.1.46 MMC_WaitUntilConditionFBEx** for details of the description, scope, and motion mode.

```
void EthercatWriteMemoryRange(
unsigned short usRegAddr,
unsigned char ucLength,
unsigned char pData[ETHERCAT_MEMORY_READ_MAX_SIZE]
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

usRegAddr

Registry address of the EtherCAT memory for the slave.

ucLength

Length of the label string. Length with the precursor format of [Metronome command]##[label]. Any +ve character values.

pData[ETHERCAT_MEMORY_READ_MAX_SIZE]

String Data with the precursor format of [Metronome command]##[label]. Any +ve character values with a maximum length of 80 bytes

[ETHERCAT_MEMORY_READ_MAX_SIZE] is the array EtherCAT memory read max size limited to 1400 bytes.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

17.2.7 EthercatWritePIVar

This function writes a Processing Image input\output variable according to its index. The variable data may be Boolean, Signed/Unsigned/Short Character, Short, Unsigned/Signed Integer, Float, or Double. Refer to the sections **9.7.14 MMC_WritePIVarBool - 9.7.26 MMC_WriteLargePIVarRaw** for details of the description, scope, and motion mode.

void EthercatWritePIVar(
unsigned short usIndex,
unsigned char ucByteLength
[unsigned short usByteLength]
[bool bData]
[signed char cData]
[unsigned char ucData]
[unsigned short usData]
[short sData]
[unsigned int uiData]
[int iData]
[float fData]
[double dbData]
unsigned char pRawData[PI_REG_VAR_SIZE]
[unsigned char pRawData[PI_LARGE_VAR_SIZE]]
[#ifdef WIN32 unsigned __int64 &ullData, __int64 &llData,
#else unsigned long long &ullData, long long &llData]
) throw (CMMCException);

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

usIndex

COB index. Any +ve integer values (2 bytes).

unsigned char ucByteLength
[unsigned short usByteLength]

Length of the RAW data including the BitStart offset. The BitStart offset is the PI variable bit offset Modulo 8, e.g. Bit Offset = 25 with the BitStart = 1. The Byte length is (BitStart + Bit size) Modulo 8.

[bool bData]
[signed char cData]
[unsigned char ucData]
[unsigned short usData]
[short sData]
[unsigned int uiData]
[int iData]
[float fData]
[double dbData]
unsigned char pRawData[PI_REG_VAR_SIZE]
[unsigned char pRawData[PI_LARGE_VAR_SIZE]]
[#ifdef WIN32 unsigned __int64 ullData, __int64 llData,
#else unsigned long long ullData, long long llData]

bool bData signed char cData unsigned char ucData unsigned short usData short sData int uiData int iData float fData double dbData	The boolean, character, unsigned character, unsigned short, short, usigned integer, integer,float, or double parameter requested from the processed image.
pRawData[PI_REG_VAR_SIZE] pRawData[PI_LARGE_VAR_SIZE]	The RAW parameter requested from the processed image. The byte size of the RAW data is (BitStart + Bit size) Modulo 8, which is inserted as the ucByteLength. Dependant on the variable type array PI_REG_VAR_SIZE which is a maximum of 4, or the variable type array PI_LARGE_VAR_SIZE which is a maximum of 1400.
unsigned __int64 ullData, __int64 llData, unsigned long long ullData, long long llData	The long long parameter requested from the processed image. If the function is defined for WIN32 then use *__int64 ullData*, else use *long long ullData*. Any +ve, –ve (Win32) or +ve 64bit (8 bytes) character and/or integer.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

17.2.8 Reset

Refer to the section **5.9.22 MMC_Reset** for details of the description, scope, and communication mode.

```
void Reset(
) throw(CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.2.9 ReadStatus

This function obtains the axis status for a specific axis. Refer to the section **5.9.21 MMC_ReadStatus** for details of the description, and scope.

unsigned long ReadStatus(
unsigned short& usAxisErrorID,
unsigned short& usStatusWord
) throw (CMMCException);

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

usAxisErrorID

Returns the axis error bitwise ID defined by the following enumerators. Bitwise ID error code:

Bit ID	Enumerator
0x1	MMC_ERR_TYPE_FAULT_BIT
0x2	MMC_ERR_TYPE_HEARTBEAT
0x4	MMC_ERR_TYPE_EMERGENCY
0x8	MMC_ERR_TYPE_COMM
0x10	MMC_ERR_TYPE_CFG_FILE

usStatusWord

Drive Status text. Any text characters.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section**17.2.1**.

17.2.10 SendSDOCmd

This function send SDO message command. Refer to the section **16.7.16 MMC_SendSDO** for details of the description, and scope.

```
void SendSdoCmd(
long lData,
unsigned char ucService,
unsigned char ucSubIndex,
unsigned long ulDataLength,
unsigned short usIndex,
unsigned short usSlaveID
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

lData

Data. Unlimited +ve or –ve values (long).

ucService

Defines whether the input is a download or upload. Accepted integer values of:

1 - Download

2 - Upload

ucSubIndex

Defines which index value signifies the group of events to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping**. From events group 7 and above, the values represent the RPDO output. This parameter should mirror the enumerator value of the ucEventGroup variable, where applicable.

Any +ve character values.

ulDataLength

Length of the data. Unlimited +ve values in bytes.

usSlaveID

The slave ID. Any +ve integer value.

usIndex

COB index. Any +ve integer values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name Axis reference Error ID
Status of the axis.

17.2.11 SendSDODownload

Sends a command to download an SDO. Refer to the section **16.7.16 MMC_SendSDO** for details of the description, and scope.

```
void SendSdoDownload(
long lData,
unsigned char ucSubIndex,
unsigned long ulDataLength,
unsigned short usIndex,
unsigned short usSlaveID
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

lData

Data. Unlimited +ve or –ve values (long).

ucSubIndex

Defines which index value signifies the group of events to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping**. From events group 7 and above, the values represent the RPDO output. This parameter should mirror the enumerator value of the ucEventGroup variable, where applicable.

Any +ve character values.

ulDataLength

Length of the data. Unlimited +ve values in bytes.

usSlaveID

The slave ID. Any +ve integer value.

usIndex

COB index. Any +ve integer values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.2.12 SendSDOUpload

Sends a command to upload an SDO. Refer to the section **16.7.16 MMC_SendSDO** for details of the description, and scope.

```
long SendSdoUpload(
unsigned char ucSubIndex,
unsigned long ulDataLength,
unsigned short usIndex,
unsigned short usSlaveID
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

ucSubIndex

Defines which index value signifies the group of events to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping**. From events group 7 and above, the values represent the RPDO output. This parameter should mirror the enumerator value of the ucEventGroup variable, where applicable.

Any +ve character values.

ulDataLength

Length of the data. Unlimited +ve values in bytes.

usSlaveID

The slave ID. Any +ve integer value.

usIndex

COB index. Any +ve integer values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.2.13 SendSDOUploadAsync

Sends a command to upload an async SDO. Refer to the section **16.7.16 MMC_SendSDO** for details of the description, and scope.

```
void SendSdoUploadAsync(
unsigned char ucSubIndex,
unsigned long ulDataLength,
unsigned short usIndex,
unsigned short usSlaveID
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

ucSubIndex

Defines which index value signifies the group of events to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping**. From events group 7 and above, the values represent the RPDO output. This parameter should mirror the enumerator value of the ucEventGroup variable, where applicable.

Any +ve character values.

ulDataLength

Length of the data. Unlimited +ve values in bytes.

usSlaveID

The slave ID. Any +ve integer value.

usIndex

COB index. Any +ve integer values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.2.14 RetreiveSDOUploadAsync

Sends a command to retrieve an async SDO. Refer to the section **16.7.16 MMC_SendSDO** for details of the description, and scope.

```
void RetreiveSdoUploadAsync(
long& lData
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

lData

Data. Unlimited +ve or –ve values (long).

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.2.15 PDOGeneralRead

Refer to the section **16.5.16 MMC_PDOGeneralRead** for details of the description, and scope.

```
MMCPPULL_T PDOGeneralRead(
unsigned char ucParam
) throw(CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

ucParam

Defines which of the general group of 32bit assigned events 16 or 17 is to be transferred to the Maestro. Refer to the section **16.4.3 PDO Mapping**. Use the values 0, 1 denoted below to represent the assignment:

NC_COMM_EVENT_GROUP16 0

NC_COMM_EVENT_GROUP17 1

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.2.16 PDOGeneralWrite

Refer to the section **16.5.17 MMC_PDOGeneralWrite** for details of the description, and scope.

```
void PDOGeneralWrite(
unsigned char ucParam,
MMCPPULL_T ulliVal
) throw(CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

ulliVal

Use the *ulliVal* parameter to describe the value of the general PDO parameter. Values are +ve 64bit (8 bytes) character and/or integer.

ucParam

Defines which of the general group of 32bit assigned events 16 or 17 is to be transferred to the Maestro. Refer to the section **16.4.3 PDO Mapping**. Use the values 0, 1 denoted below to represent the assignment:

NC_COMM_EVENT_GROUP16 0

NC_COMM_EVENT_GROUP17 1

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.2.17 GetPDOInfo

Refer to the section **16.5.14 MMC_GetPDOInfo** for details of the description, and scope.

```
void GetPDOInfo(
unsigned char uiPDONumber,
int &iPDOEventMode,
unsigned char &ucPDOCommType,
unsigned char &ucTPDOCommEventGroup,
unsigned char &ucRPDOCommEventGroup
) throw(CMMCException);
```

Source GMAS\includes\CPP\CMMCNode.h

.NET Definition

Function Parameters

uiPDONumber

The PDO index number. Changes a specific PDO's communication from sync to async and visa versa. Allowed values are 1 to 4, representing the PDO1, PDO2, PDO3, and PDO4.

&iPDOEventMode

The PDO event mode integer

&ucPDOCommType

PDO communication type as a +ve character value

&ucTPDOCommEventGroup

TPDO communication event group as a +ve character value

&ucRPDOCommEventGroup

RPDO communication event group as a +ve character value

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.2.18 SetBoolParameter

Sets the Boolean Parameter. Refer to the section **5.9.28 MMC_WriteBoolParameter** for details of the description, scope, and motion mode.

```
void SetBoolParameter(
unsigned long ulValue,
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNode.h

GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

ulValue

Any integer value. +ve numeric value.

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.4 Axis Status** for the appropriate integer parameter to be used as enumerator.

iIndex

Index array (only relevant for array situations). Any +ve integer values

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.2.19 SetParameter

Sets an array Parameter. Refer to the section **5.9.32 MMC_WriteParameter** for details of the description, scope, and motion mode.

```
void SetParameter(
double dbValue,
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNode.h

GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

dbValue

Array parameter with double value.

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.4 Axis Status** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.2.20 GetBoolParameter

Obtains a Boolean Parameter. Refer to the section **5.9.14 MMC_ReadBoolParameter** for details of the description, scope, and motion mode.

```
Long GetBoolParameter(
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
)throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNode.h

GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.4 Axis Status** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values

Return

lValue Boolean parameters integer value

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.2.21 GetParameter

Obtains any Parameter. Refer to the section **5.9.19 MMC_ReadParameter** for details of the description, scope, and motion mode.

```
double GetParameter(
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
)throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNode.h

GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MMC_PARAMETER_LIST_ENUM eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3 Axis, Group, Global, Parameters** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values

Return

dbValue

Output of the specific parameter. Any Double value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.2.22 GetAxisError

Obtains the last axis error. Refer to the section 5.9.13 MMC_ReadAxisError for details of the description, scope, and motion mode.

```
unsigned short GetAxisError(
unsigned short* usLastEmergencyErrCode
)throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNode.h

.NET Definition

Function Parameters

usLastEmergencyErrCode

Returns the axis error bitwise ID defined by the following enumerators. Bitwise ID error code:

Bit ID	Enumerator
0x1	MMC_ERR_TYPE_FAULT_BIT
0x2	MMC_ERR_TYPE_HEARTBEAT
0x4	MMC_ERR_TYPE_EMERGENCY
0x8	MMC_ERR_TYPE_COMM
0x10	MMC_ERR_TYPE_CFG_FILE

Refer to the chapter Chapter 12: API Events for an explanation of the various bit errors.

Return

throw (CMMCException)

Refer to the section 17.1.1 CMMCException. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.3 The CMMCAxis class

This class is the main header for the single and multiple axes wrapper functions as described in the diagram in **Figure 17-3**. The diagram displays the hierarchical structure of the main classes and their type definitions associated with CMMCAxis.Each class consist of an *.h file containing functions which correspond to similar functions in C, and are referenced with their similar explanations and details. However, this is appropriate only to Public functions. Private functions are internal functions which are transparent to the user.

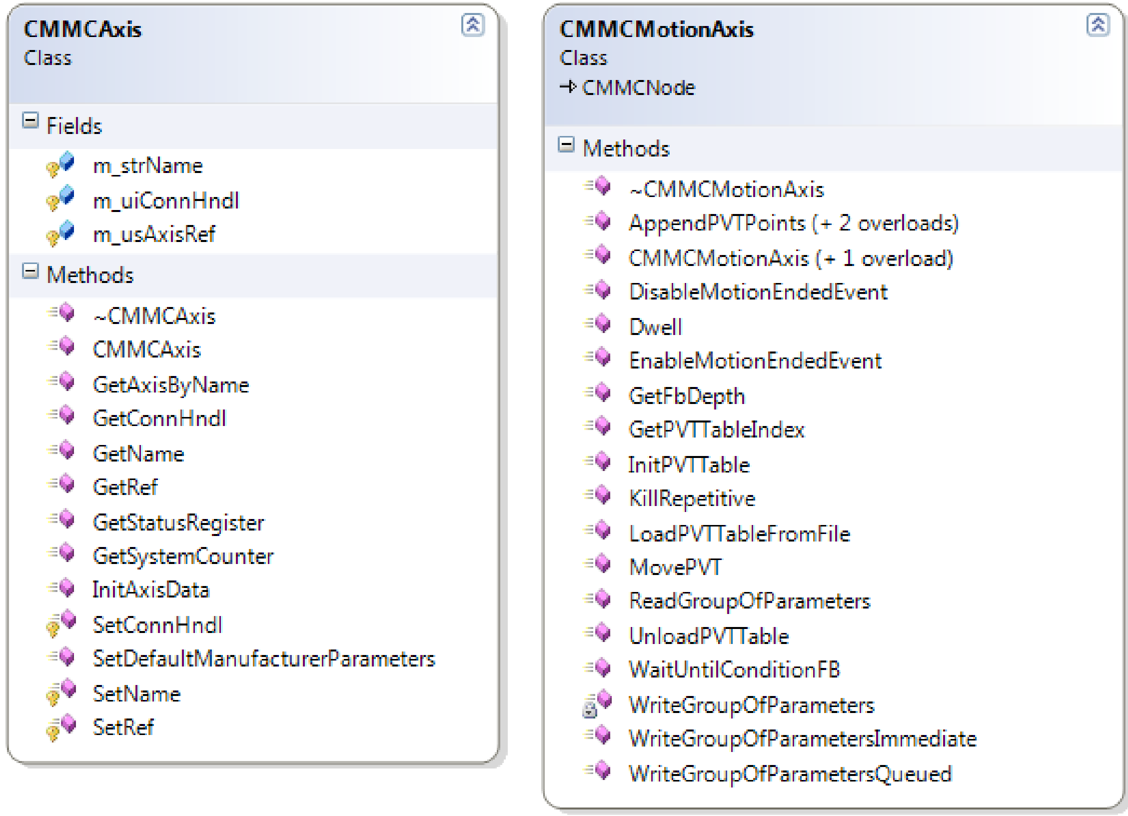


Figure 17-3 CMMCAxis class diagram

17.3.1 DisableMotionEndedEvent

This function disables the event of Ended Motion. The same callbacks are called as in the Singe Axis, however, the group axis reference is called.

void DisableMotionEndedEvent(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCAxis.h

.NET Definition

Function Parameters

DisableMotionEndedEvent

Refer to the section **12.22.3 MMC_DisableMotionEndedEvent** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.3.2 EnableMotionEndedEvent

This function enables the event of Ended Motion. The same callbacks are called as in the Singe Axis, however, the group axis reference is called.

void EnableMotionEndedEvent(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCAxis.h

.NET Definition

Function Parameters

EnableMotionEndedEvent

Refer to the section **12.22.4 MMC_EnableMotionEndedEvent** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.3.2.1 Functions Code Example

```
// 16.2.2. DisableMotionEndedEvent
// 16.2.3. EnableMotionEndedEvent
void  EnableDisableMotionEndedEvent(void)
// ========================================
{
 int   loopInd;

printf("\n     %s:", __func__);
 for (loopInd=0; loopInd<2; loopInd++)
 {
  printf("\n +++++++++ On end of motion ");
  if (loopInd==0)
  {
   AxisA.EnableMotionEndedEvent();
   printf("EXPECT:");
  }
  else
  {
   AxisA.DisableMotionEndedEvent();
   printf("NOT EXPECT:");
  }
  printf(" <%s> (call back func) ", EndMotionEventCB_MESSAGE);

  MoveAbsoluteMoves();
  printf("\n +++++++++ Motion started...");
  WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
  printf("\n +++++++++ Motion End \n");
 }

 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
}
```

17.3.3 SetDefaultManufacturerParameters

This function restores the axis\group parameters to their original factory defaults values.

void SetDefaultManufacturerParameters(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCAxis.h

.NET Definition

Function Parameters

SetDefaultManufacturerParameters

Refer to the section **8.1.36 MMC_SetDefaultParametersGlobal on page 866** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.3.3.1 Function Code Example

```
// 16.2.4.    SetDefaultManufacturerParameters
//  CMMCAxis::SetDefaultManufacturerParameters   10.3.36. MMC_SetDefaultParametersCmd
// CMMCConnection::SetDefaultManufacturerParameters  10.3.37.MMC_SetDefaultParametersGlobalCmd
void  SetDefManufact(void)
// =========================
{
unsigned long     ulong;
printf("\n     %s:", __func__);
 Group.GroupDisable();
 do
 {
  ulong = Group.GroupReadStatus();
 } while  (!(ulong & NC_GROUP_DISABLED_MASK));
 AxisB.PowerOff(MC_BUFFERED_MODE);
 AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

 AxisA.SetDefaultManufacturerParameters();
 AxisB.SetDefaultManufacturerParameters();
 Group.SetDefaultManufacturerParameters();
 cConn.SetDefaultManufacturerParameters();
    AxisA.PowerOn(MC_BUFFERED_MODE);
    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
 Group.GroupEnable();
 do
 {
  ulong = Group.GroupReadStatus();
 } while  (!(NC_GROUP_STANDBY_MASK & ulong));
}
```

17.3.4 GetAxisByName

Refer to the section **8.1.16 MMC_GetAxisByName** for details of the description, scope, and motion mode.

```
int GetAxisByName(
const char* cName
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCAxis.h

.NET Definition

Function Parameters

cName

Tag/assembly name as declared in XML configuration file.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

Refer to the function example in section **17.3.6**

17.3.5 InitAxisData

This function initiates an axis name and retrieves a session handler. Refer to the section **8.1.16 MMC_GetAxisByName** for details of the description, scope, and motion mode.

```
ivirtual void InitAxisData(
const char* cName,
 MMC_CONNECT_HNDL uHandle
) throw (CMMCException)
```

Source GMAS\includes\CPP\MMCAxis.h

.NET Definition

Function Parameters

cName

Tag/assembly name as declared in XML configuration file.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

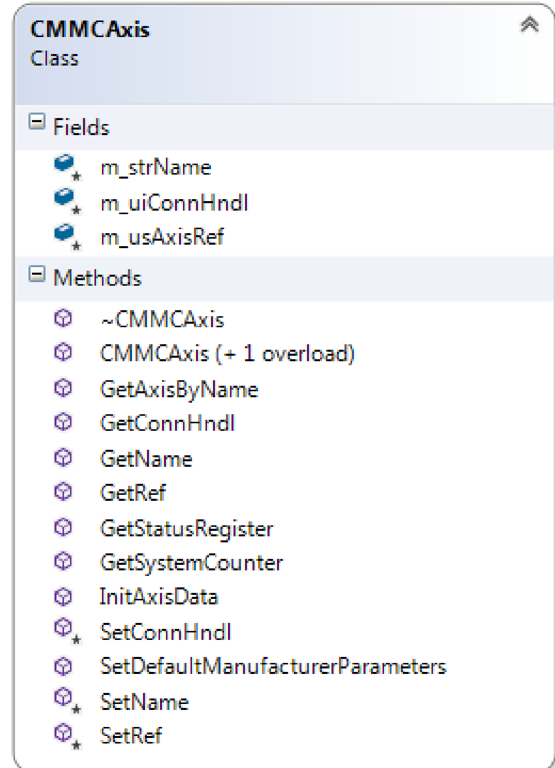
Refer to the function example in section **17.3.6**

17.3.6 CMMCAxis Class Functions Code Examples

```
// 16.2.5.  GetFbDepth   5.7.3. MMC_GetFbDepth Cmd
// unsigned int CMMCMotionAxis::GetFbDepth()
//    16.2.6.     GetAxisByName     10.3.17. MMC_GetAxisByName Cmd
//int CMMCAxis::GetAxisByName(const char* cName)
//    16.2.7.     GetGroupAxisByName  10.3.18. MMC_GetGroupByName Cmd
// int CMMCGroupAxis::GetGroupAxisByName(const char* cName)
void  DepthName(void)
// ====================
{
unsigned int iVal1, iVal2, iVal3;


printf("\n     %s:", __func__);
 iVal1 = AxisB.GetFbDepth();

 Group.GroupDisable();
 AxisB.PowerOff(MC_BUFFERED_MODE);

 iVal2 = AxisB.GetFbDepth();
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
 iVal3 = AxisB.GetFbDepth();

 printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
 Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");   /* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

    // iVal3 = AxisA.GetAxisByName("A01"); /* It case sensitive - Not define - exception... */

/*
*   iVal2 = Group.GetGroupAxisByName("v02");
*/
}
```

17.4 The CMMCMotionAxis class

The base class for PVT (ECAM and splines, in future) is the class CMMCMotionAxis, which is inherited from CMMCAxis, i.e. it contains all CMMCAxis members and methods. The class CMMCMotionAxis functions are detailed in **Chapter 6: Position, Velocity, Time (PVT) Motion**. The class CMMCMotionAxis retains the same field parameter properties and values described in this document for the C function blocks.



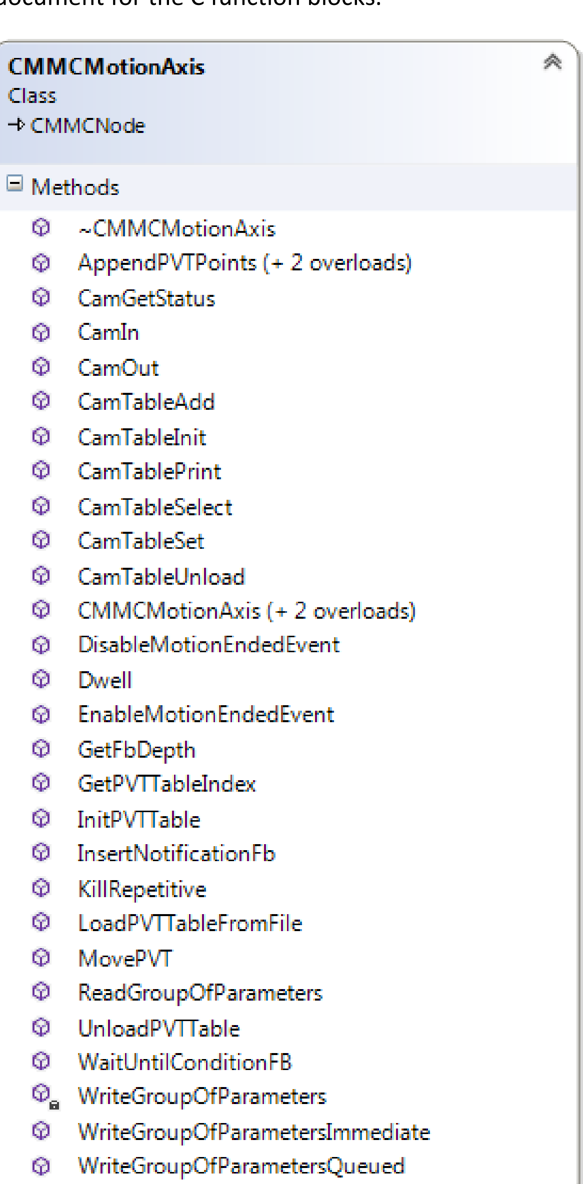


Figure 17-4 Fields and methods of the CMMCMotionAxis class

The detailed class view shown in **Figure 17-25** describes the fields and methods associated with the CMMCMotionAxis class.

It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

The base class for PVT and ECAM called CMMCMotionAxis is inherited from CMMCAxis, i.e. it contains all CMMCAxis members and methods.

This method is inherited from CMMCAxis class, but overloaded in the CMMCMotionAxis class, since the method does not suit to the requirements of the CMMCAxis implementation. The BindAxis method is used instead, and InitAxisData will just throw an exception.

The C++ PVT functions wrap the motion axis parameter reading functions detailed in **Chapter 6: Position, Velocity, Time (PVT) Motion** and retains similar field parameter properties and values described in this document for the C function blocks.

The PVT contains the following methods:

Function	Explanation
InitPVTTable	The method is a wrapper for the MMC_InitTableCmd() C command. InitTableCmd().
LoadPVTTable	The method loads PVT table from file
AppendPointsToPVTTable	This method appends points to the current PVT table (in automatic mode).
AppendPointsToPVTTable	This method appends points to the current PVT table (in manual mode).
MovePVT	This method inserts PVT function block.
UnloadPVTTable	Unloads PVT table

Similarly, the C++ ECAM functions wrap the motion axis parameter reading functions detailed in **Chapter 7: Electronic CAM** and retains similar field parameter properties and values described in this document for the C function blocks. The ECAM contains the following methods:

Function	Explanation
CamGetStatus	Retrieves the significant parameters of the CAM process
CamIn	Executes the CAM process
CamOut	Performs a Stop function procedure on the slave axis
CamTableAdd	Appends points to an existing table
CamTableInit	Allocates memory for the ECAM table, prepares and initializes the function block in journal
CamTablePrint	Ouputs the CAM Table data to a prepared device
CamTableSelect	Selects a table by input handler
CamTableSet	Used for loading a table from memory
CamTableUnload	Unloads a ECAM table from the Maestro and frees a memory segment in the Maestro shared memory according to the dimension and number of points given in a file

17.4.1 CMMCMotionAxis Source Code Examples

```
eTableType = eNC_TABLE_PVT_ARRAY;
ucIsCyclic = 1;
ucIsDynamicMode = 1;
ucIsPosAbsolute = 0;
ulMaxNumberOfPoints = 50;
ulUnderflowThreshold = 6;

usAxisRef = g_cGroupObject.GetRef();
usDimension = NUMBER_OF_AXES;

try
{
g_cPVTObject.BindAxis(g_cGroupObject);
 g_cPVTObject.InitPVTTable(ulMaxNumberOfPoints, ulUnderflowThreshold, ucIsCyclic,
ucIsDynamicMode, ucIsPosAbsolute);
g_cPVTObject.AppendPointsToPVTTable(stAppendIn.dTable, stAppendIn.ulNumberOfPoints, (unsigned
char )0);
g_cPVTObject.MovePVT(MC_ACS_COORD);
}

catch (CMMCException ex)
{
 cout << "Failed to init, error [" << ex.error() << "]" << endl;
}
```

Or, using file mode:

```
try
{
g_cPVTObject.BindAxis(g_cGroupObject);
g_cPVTObject.LoadPVTTable(pFileName);
g_cPVTObject.MovePVT(MC_ACS_COORD);
}

catch (CMMCException ex)
{
 cout << "Failed to init, error [" << ex.error() << "]" << endl;
}
```

17.4.2 CAM Process Example

```
_axes[0].CamTableUnload(-1);                                  //clear all tables if so desired.
_CamTable.dbGap = 1000;                                       //relevant only on fixed gap (see CAM table header)
_CamTable.dbMasterStartPosition = 0;                          //relevant only on absolute master and fixed gap
_CamTable.eCoordSystem = MC_NONE_COORD;                       //for future use
_CamTable.iCamTableID = -1;                                   //may be retrieved by MC_CamTableInit
_CamTable.pPathToTableFile[0] = "/mnt/jffs/usr/table.cm"      //table path is ignored on memory loading.
_CamTable.eCurveType = eQuinticInterp;                        //eQuinticInterp is the default interpolation type.
_CamTable.eTableMode = eCAMT_RWMode;                          //modifiable table
_uiTableID = _axes[2].CamTableSelect(_CamTable, 0, 1, 0);     //master absolute, slave/relative

uiMaster = _axes[1].GetRef();
_axes[2].CamIn(uiMaster, MC_BUFFERED_MODE, _uiTableID, 0, eCAM_PERIODIC);
While (TRUE)
{
        If (<user program condition>) {
                _axes[2].MC_CamOut();
                Break;
        }
        usleep(50000);
}
```

17.4.3 GetFbDepth

This function sends a command to receive the number of function blocks in the Node Queue. Refer to the sections **5.9.6 MMC_GetFBDepth** and **5.9.7 MMC_GetTotalFbDepth** for details of the description, scope, and motion mode.

```
unsigned int GetFbDepth(
const unsigned int uiHndl
)throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

GetFbDepth

Refer to the section

on page 269 for details of the function.

uiHndl

Returned function block handle. Integer with any +ve value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

Refer to the function example in section **17.6.7.1**

17.4.4 EnableMotionEndedEvent

This function enables the event of Ended Motion. The same callbacks are called as in the Singe Axis, however, the group axis reference is called.

```
void EnableMotionEndedEvent(
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

EnableMotionEndedEvent

Refer to the section **12.22.4 MMC_EnableMotionEndedEvent** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.4.5 DisableMotionEndedEvent

This function disables the event of Ended Motion. The same callbacks are called as in the Singe Axis, however, the group axis reference is called.

```
void DisableMotionEndedEvent(
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

DisableMotionEndedEvent

Refer to the section **12.22.3 MMC_DisableMotionEndedEvent** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.4.6 Dwell

This function inserts a Dwell function block to the Function Block queue of the axis. Refer to the section **5.9.5 MMC_Dwell** for details of the description, scope, and motion mode.

```
void Dwell (
unsigned long ulDwellTimeMs
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

ulDwellTimeMs

Refer to the **MMC_DWELL_IN Structure** for details of this parameter.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.4.7 InsertNotificationFb

This function inserts a notification function block that will generate an event when called. Refer to the section **12.22.1 MMC_InsertNotificationFb** for details of the description, scope, and motion mode.

void InsertNotificationFb(
[int iEventCode]
[MMC_INSNOTIFICATIONFB_IN stInParams]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

iEventCode

Refer to the section **MMC_INSNOTIFICATIONFB_IN Structure** for details.

MMC_INSNOTIFICATIONFB_IN stInParams

Refer to the section **MMC_INSNOTIFICATIONFB_IN Structure** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.4.8 WaitUntilConditionFB

This function inserts a condition function block to the queue of the node. Refer to the section **8.1.45 MMC_WaitUntilConditionFB** for details of the description, scope, and motion mode.

void WaitUntilConditionFB(
MMC_WAITUNTILCONDITIONFB_IN stInput
) throw (CMMCException);

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

MMC_WAITUNTILCONDITIONFB_IN stInput

Refer to the section **MMC_WAITUNTILCONDITIONFB_IN Structure** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.4.9 WaitUntilConditionFBEX

The operation of this function block applies to both static and PI functions, and allows synchronization of numerous axes that are not part of a group, to start their motion together. Refer to the section **8.1.46 MMC_WaitUntilConditionFBEx** for details of the description, scope, and motion mode.

void WaitUntilConditionFBEX(
MMC_WAITUNTILCONDITIONFBEX_IN stInput
) throw (CMMCException);

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

MMC_WAITUNTILCONDITIONFBEX_IN stInput

Refer to the section **MMC_WAITUNTILCONDITIONFBEx_IN Structure** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.4.10 KillRepetitive

This function stops the repetitive motion after the current function block. Refer to the section **8.1.50 MMC_KillRepetitive** for details of the description, scope, and motion mode.

void KillRepetitive (
) throw (CMMCException);

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.4.11 InitPVTTable

This function loads the PVT table from file. Refer to the section **6.8.1 MMC_InitTable** for details of the description, and scope.

```
virtual MC_PATH_REF InitPVTTable (
unsigned long ulMaxPoints,
unsigned long ulUnderflowThreshold,
unsigned char ucIsCyclic,
unsigned char ucIsPosAbsolute,
unsigned short usDimension,
MC_COORD_SYSTEM_ENUM eCoordSystem,
NC_MOTION_TABLE_TYPE_ENUM eTableMode = eNC_TABLE_PVT_ARRAY
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

ulMaxPoints

The maximal number of points that the table will be able to contain (in non-cyclic mode). Any +ve values accepted.

ulUnderflowThreshold

An event will be generated if the number of points between the current index, and the end index falls below this value. Value cannot be greater that the ulMaxPoints value. Any +ve values accepted.

ucIsCyclic

Is the table supposed to be cyclic? i.e. when the index reaches the end of the table, will it roll over and start from the beginning. Boolean answer 0 or 1

ucIsPosAbsolute

Is position absolute? Boolean answer 0 or 1

usDimension

Dimensions of the PVT table. +ve value

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator value. The options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

NC_MOTION_TABLE_TYPE_ENUM eTableType

The Table type defined according to the enumerator parameter NC_MOTION_TABLE_TYPE_ENUM defined by:

eNC_TABLE_NONE	= 0	
eNC_TABLE_SPLINE	= 1	
eNC_TABLE_PVT_FILE	= 2	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_ARRAY	= 3	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_FILE_QUINTIC_CUB	= 4	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_PVT_ARRAY_QUINTIC_CUB	= 5	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_ECAM_FILE	= 6	
eNC_TABLE_ECAM_ARRAY	= 7	
eNC_TABLE_OLSPLN_FILE	= 8	
eNC_TABLE_OLSPLN_ARRAY	= 9	
eNC_TABLE_MAX	= 10	

This enumeration is used as the input for these functions, to distinguish between ECAM and PVT. Currently only eNC_TABLE_PVT_FILE and eNC_TABLE_PVT_ARRAY can be applied.

throw (CMMCException)

Refer to the **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.12 InitPTTable

This function allocates and initiates the PT (online spline) table on Maestro. Refer to the section **6.8.1 MMC_InitTable** for details of the description, and scope.

```
virtual MC_PATH_REF InitPTTable (
unsigned long ulMaxPoints,
unsigned long ulUnderflowThreshold,
unsigned char ucIsCyclic,
unsigned char ucIsDynamic,
unsigned char ucIsPosAbsolute,
unsigned short usDimension,
MC_COORD_SYSTEM_ENUM eCoordSystem,
NC_ONLINE_SPLINE_MODE_ENUM eSplineMode = MC_QUINTIC_ON_PARAB_VT_DWELL,
double ConstVelocity = 1000.0,
double FixedTime = 2.0
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

ulMaxPoints

The maximal number of points that the table will be able to contain (in non-cyclic mode). Any +ve values accepted.

ulUnderflowThreshold

An event will be generated if the number of points between the current index, and the end index falls below this value. Value cannot be greater that the ulMaxPoints value. Any +ve values accepted.

ucIsCyclic

Is the table supposed to be cyclic? i.e. when the index reaches the end of the table, will it roll over and start from the beginning. Boolean answer 0 or 1

ucIsDynamic

Is table to be dynamic? Boolean answer 0 or 1

ucIsPosAbsolute

Is position absolute? Boolean answer 0 or 1

usDimension

Dimensions of the PVT table. +ve value

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator value. The options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

NC_ONLINE_SPLINE_MODE_ENUM eSplineMode = MC_QUINTIC_ON_PARAB_VT_DWELL

Define the types of spline mode. The NC_ONLINE_SPLINE_MODE_ENUM enumerator options are:

MC_NONE_ONLINE_SPLINE_MODE = 0

MC_QUINTIC_ON_PARAB_FT_DWELL = 1 //defines fixed time

MC_QUINTIC_ON_PARAB_VT_DWELL = 2 //defines variable time

MC_QUINTIC_ON_PARAB_CV_DWELL = 3 //defines constant velocity

ConstVelocity = 1000.0

Defines the constant velocity in this situation fixed to 1000.0

FixedTime = 2.0

Defines the fixed time, in this situation fixed to 2.0.

throw (CMMCException)

Refer to the **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.13 LoadPVTTableFromFile

This method loads a PVT table from file. Refer to the section **6.8.3 MMC_LoadTableFromFile** for details of the description, and scope.

```
virtual MC_PATH_REF LoadPVTTableFromFile(
char* pFileName,
MC_COORD_SYSTEM_ENUM eCoordSystem,
NC_MOTION_TABLE_TYPE_ENUM eTableMode = eNC_TABLE_PVT_FILE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

pFileName

The file name in standard UNIX format. String value (null-terminated).

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator value. The options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

NC_MOTION_TABLE_TYPE_ENUM eTableMode = eNC_TABLE_PVT_FILE

The Table type defined according to the enumerator parameter NC_MOTION_TABLE_TYPE_ENUM defined by:

```
eNC_TABLE_NONE                  = 0
eNC_TABLE_SPLINE                = 1
eNC_TABLE_PVT_FILE              = 2  //old method eCUBIC_POLYNOM
eNC_TABLE_PVT_ARRAY             = 3  //old method eCUBIC_POLYNOM
eNC_TABLE_PVT_FILE_QUINTIC_CUB  = 4  //new method eQUINTIC_ON_CUBIC
eNC_TABLE_PVT_ARRAY_QUINTIC_CUB = 5  //new method eQUINTIC_ON_CUBIC
eNC_TABLE_ECAM_FILE             = 6
eNC_TABLE_ECAM_ARRAY            = 7
eNC_TABLE_OLSPLN_FILE           = 8
eNC_TABLE_OLSPLN_ARRAY          = 9
eNC_TABLE_MAX                   = 10
```

This enumeration is used as the input for these functions, to distinguish between ECAM and PVT. Currently only eNC_TABLE_PVT_FILE and eNC_TABLE_PVT_ARRAY can be applied.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.14 LoadPTTableFromFile

This function loads a PT (online spline) table from file. Refer to the section **6.8.3 MMC_LoadTableFromFile** for details of the description, and scope.

```
virtual MC_PATH_REF LoadPTTableFromFile(
char* pFileName,
MC_COORD_SYSTEM_ENUM eCoordSystem
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

pFileName

The file name in standard UNIX format. String value (null-terminated).

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems using the MC_COORD_SYSTEM_ENUM enumerator value. The options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.15 AppendPVTPoints

The method appends points to the current PVT table (in automatic and manual modes). Refer to the section 6.8.6 MMC_AppendPointsToTable for details of the description, and scope.

Append automatically

```
void AppendPVTPoints(
MC_PATH_REF hMemHandle,
double dTable[NC_PVT_ECAM_MAX_ARRAY_SIZE],
unsigned long ulNumberOfPoints,
unsigned char ucIsTimeAbsolute = 0,
NC_MOTION_TABLE_TYPE_ENUM eTableType = eNC_TABLE_PVT_ARRAY
) throw (CMMCException);
```

Append Manually

```
void AppendPVTPoints(
MC_PATH_REF hMemHandle,
double dTable[NC_PVT_ECAM_MAX_ARRAY_SIZE],
unsigned long ulNumberOfPoints,
unsigned long ulStartIndex,
unsigned char ucIsTimeAbsolute = 0,
NC_MOTION_TABLE_TYPE_ENUM eTableType = eNC_TABLE_PVT_ARRAY
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

MC_PATH_REF hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle can have integer values.

dTable [NC_PVT_ECAM_MAX_ARRAY_SIZE]

Pointer to the Table of array values. Table of NC_PVT_ECAM_MAX_ARRAY_SIZE limited length in values, limited to 170 values max.

ulNumberOfPoints

Number of points in a row to append. Any +ve values accepted.

ulStartIndex

User specifies the row index for inserting a table segment in the G-MAS, from which to start the append process. Any +ve values accepted.

ucIsTimeAbsolute = 0

Boolean result to the question, Is the Time Absolute or not? Select the mode of the parameter:

0 - absolute mode

Every "time" input is used as absolute time when the current point will end the motion and reach the desired position.

1 - relative mode
Every "time" input is used as the time that will take to move from previous point to the end of current point.

NC_MOTION_TABLE_TYPE_ENUM eTableType = eNC_TABLE_PVT_ARRAY

The Table type defined according to the enumerator parameter NC_MOTION_TABLE_TYPE_ENUM defined by:

eNC_TABLE_NONE	= 0	
eNC_TABLE_SPLINE	= 1	
eNC_TABLE_PVT_FILE	= 2	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_ARRAY	= 3	//old method eCUBIC_POLYNOM
eNC_TABLE_PVT_FILE_QUINTIC_CUB	= 4	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_PVT_ARRAY_QUINTIC_CUB	= 5	//new method eQUINTIC_ON_CUBIC
eNC_TABLE_ECAM_FILE	= 6	
eNC_TABLE_ECAM_ARRAY	= 7	
eNC_TABLE_OLSPLN_FILE	= 8	
eNC_TABLE_OLSPLN_ARRAY	= 9	
eNC_TABLE_MAX	= 10	

This enumeration is used as the input for these functions, to distinguish between ECAM and PVT. Currently only eNC_TABLE_PVT_FILE and eNC_TABLE_PVT_ARRAY can be applied.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.16 AppendPTPoints

The method appends points to the current PT table. Refer to the section **6.8.6 MMC_AppendPointsToTable** for details of the description, and scope.

Append automatically

void AppendPTPoints(
MC_PATH_REF hMemHandle,
double dTable[NC_PVT_ECAM_MAX_ARRAY_SIZE],
unsigned long ulNumberOfPoints,
unsigned char ucAutoAppend,
unsigned long ulStartIndex,
unsigned char ucIsTimeAbsolute = 0
) throw (CMMCException);

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

MC_PATH_REF hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.
hMemHandle can have integer values.

dTable [NC_PVT_ECAM_MAX_ARRAY_SIZE]

Pointer to the Table of array values. Table of NC_PVT_ECAM_MAX_ARRAY_SIZE limited length in values, limited to 170 values max.

ulNumberOfPoints

Number of points in a row to append. Any +ve values accepted.

ucAutoAppend

Whether appended automatically or not?. Boolean value 0 or 1.

ulStartIndex

Index to manually append from. Any +ve values accepted. Used only in manual mode.

ucIsTimeAbsolute = 0

Boolean result to the question, Is the Time Absolute or not? Select the mode of the parameter:
0 - absolute mode
Every "time" input is used as absolute time when the current point will end the motion and reach the desired position.
1 - relative mode
Every "time" input is used as the time that will take to move from previous point to the end of current point.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.17 GetTableList

This function provides a list of tables for given table type. Refer to the section **6.7.5 MMC_GetTableList** for details of the description, and scope.

```
int GetTableList(
NC_MOTION_TABLE_TYPE_ENUM eTableType,
unsigned int (&uiHandlers)[MMC_MAX_JOURNAL_ENTRIES],
int &iNum
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

NC_MOTION_TABLE_TYPE_ENUM eTableType

The Table type defined according to the enumerator parameter NC_MOTION_TABLE_TYPE_ENUM defined by:

eNC_TABLE_NONE = 0

eNC_TABLE_SPLINE = 1

eNC_TABLE_PVT_FILE = 2 //old method eCUBIC_POLYNOM

eNC_TABLE_PVT_ARRAY = 3 //old method eCUBIC_POLYNOM

eNC_TABLE_PVT_FILE_QUINTIC_CUB = 4 //new method eQUINTIC_ON_CUBIC

eNC_TABLE_PVT_ARRAY_QUINTIC_CUB = 5 //new method eQUINTIC_ON_CUBIC

eNC_TABLE_ECAM_FILE = 6

eNC_TABLE_ECAM_ARRAY = 7

eNC_TABLE_OLSPLN_FILE = 8

eNC_TABLE_OLSPLN_ARRAY = 9

eNC_TABLE_MAX = 10

This enumeration is used as the input for these functions, to distinguish between ECAM and PVT. Currently only eNC_TABLE_PVT_FILE and eNC_TABLE_PVT_ARRAY can be applied.

(&uiHandlers)[MMC_MAX_JOURNAL_ENTRIES]

Defines the list of table handlers with a maximum limited to the number of MMC_MAX_JOURNAL_ENTRIES, i.e. 20. This is the buffer of table handlers to be retrieved by eTableType.

&iNum

Defines the number of tables found. Stores the number of loaded tables of eTableType when successfully returned. Positive integer value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.18 GetTableInfo

This function provides a list of tables for given table type. Refer to the section **6.7.5 MMC_GetTableList** for details of the description, and scope.

```
int GetTableInfo(
unsigned int uiHandler,
char (&name)[MMC_TABLE_FILE_LENGTH+1]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

uiHandler

Defines the list of table handlers. +ve integer value.

(&name)[MMC_TABLE_FILE_LENGTH+1]

Defines the name of the table with a max length for the table file name (exluded from path) of 20+1 i.e. MMC_TABLE_FILE_LENGTH+1

Note: The length of file's name refers only to file name and not the length of the whole path. If file name is longer than twenty only the first twenty characters shall be returned.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.19 MovePVT

This method inserts a PVT function block. Refer to the section **6.8.5 MMC_MoveTable** for details of the description, and scope.

void MovePVT(
MC_COORD_SYSTEM_ENUM eCoordSystem = MC_NONE_COORD
);

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

17.4.20 UnloadPVTTable

This method unloads the PVT table from the Maestro. Refer to the section **6.8.4 MMC_UnloadTable** for details of the description, and scope.

void UnloadPVTTable(
void
);

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

void

Function takes no parameters

17.4.21 CamGetStatus

MC_Cam GetStatus retrieves parameters, which are significant to CAM process. Refer to the section **7.6.9 MMC_CamStatus** for details of the description, and scope.

```
int CamGetStatus(
unsigned long& ulEndOfProfile,
unsigned long& ulCurrentIndex,
unsigned long& ulCycle,
unsigned short& usStatus,
short& sErrorID
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

ulEndOfProfile

Counts number of exists from cam table.

ulCurrentIndex

Segment index in CAM table, which is currently processed.

ucIsCyclic

Is the table supposed to be cyclic? i.e. when the index reaches the end of the table, will it roll over and start from the beginning. Boolean answer 0 or 1

usStatus

Bitwise returned command status with the following values:

Aborted
Done
CommandError

sErrorID

Returned command error ID. Signals where an error has occurred within function block. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

throw (CMMCException)

Refer to the **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.22 CamTableInit

Allocate memory for table, prepare and initializes function block in journal. Refer to the section **7.6.1 MMC_CamTableInit** for details of the description, and scope.

```
MC_PATH_REF CamTableInit(
unsigned long ulMaxPoints,
unsigned short usDimension,
unsigned char ucIsFixedGap=0
CURVE_TYPE_ENUM eCurveType=eQuinticInterp
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

ulMaxPoints

Maximal number of points for this table.

usDimension

Number of slaves. Group is not supported presently, and therfore the dimension is always 1.

ucIsFixedGap=0

The gap is variable with a value of 0. It can optionally be Fixed (1)

CURVE_TYPE_ENUM eCurveType

Interpolation type. User defined type allows different type for each segment. Any other type sets that type (same type) for all segments. ***CurveType of each segment defines the curve type between current segment and its predecessor.***

CurveType defines interpolation type. If CurveType is 'UserDefineInterp', then CurveType must be supplied by user array (via MC_CamTableAdd/Set) or table's file on the last column of each row. Otherwise the supplied CurveType will be used for all segments and CurveType must not be supplied by user array (via MC_CamTableAdd/Set). File loader ignores CurveType column if CurveType is not 'UserDefineInterp'.

CurveType of each segment defines the curve type between current segment and its predecessor. Defined by the enumerator CURVE_TYPE_ENUM with Interpolation type values:

eTableDefInterp = 0

eLinearInterp = 1

ePolynom5Interp = 2

ePolynom7Interp = 3,

eCycloidPositionInterp = 4

eCycloidVelocityModified1Interp = 5

eCycloidVelocityModified2Interp = 6

throw (CMMCException)

Refer to the **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.23 CamTableSelect

This function selects a table by input handler. Refer to the section **7.6.2 MMC_CamTableSelect** for details of the description, and scope.

```
MC_PATH_REF CamTableSelect(
const MC_CamRef& CamTableDescr,
unsigned int uiStartMode=0,
unsigned char ucIsMasterPosAbsolute,
unsigned char ucIsSlavePosAbsolute
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

MC_CamRef& CamTableDescr

MC_CamRef Input/output parameter of MC_CamTableSelect is an Elmo specific data type. Refer to MC_CAMREF for deails of this data type.

uiStartMode

Overrides uiStartMode of MC_CamTableSelect. Reserved for future use of Ramp-In and other options.

ucIsMasterPosAbsolute

Boolean parameter value. If 1 GMAS refers to master column as absolute values, otherwise 0 as relative values.

ucIsSlavePosAbsolute

Boolean parameter value. If 1 GMAS refers to slave column as absolute values, otherwise 0 as relative values.

throw (CMMCException)

Refer to the **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.24 CamIn

MC_CamIn executes the CAM process. Refer to the examples in section **7.7 Application Example** for details of the description, and scope.

```
int CamIn(
unsigned short usMaster,
MC_BUFFERED_MODE_ENUM eBufferMode,
unsigned int uiCamTableID,
unsigned int ucAutoOffset = 0,
ECAM_PERIODIC_ENUM ePeriodic=eCAM_NON_PERIODIC,
double dbMasterSyncPosition=0.0,
double dbMasterStartDistance = 0,
unsigned int uiStartMode = 0,
double dbMasterOffset = 0,
double dbSlaveOffset = 0,
double dbMasterScaling = 1,
double dbSlaveScaling = 1,
ECAM_VALUE_SRC_ENUM eMasterValueSource = eECAM_SET_VALUE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

usMaster

The master axis reference.

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows, but only the Buffered Mode is supported:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered The next function block affects the axis as soon as the previous movement is completed.

BlendingLow The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-

position (1).

BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

ucAutoOffset = 0

Auto offset. Adjust slave position in table to axis position when master reaches Sync Position.

ECAM_PERIODIC_ENUM ePeriodic=eCAM_NON_PERIODIC

Describes the perodicity mode of the function, according to the following options:

eCAM_NON_PERIODIC	= 0	One shot
eCAM_PERIODIC	= 1	periodic
eCAM_PERIODIC_LINEAR	= 2	periodic-linear

dbMasterSyncPosition=0.0

Defined as relative to first phase of master in CAM table. If table is relative then it is defined as relative master position just like any other phase in table.

dbMasterStartDistance = 0

Backward distance from dbMasterSyncPosition to allow Ramp-In. Untill we have Ram-In implemented it is always zero.

uiStartMode = 0

Overrides uiStartMode of MC_CamTableSelect

dbMasterOffset = 0

Master offset from the master definition in the CAM table.

dbSlaveOffset = 0

Slave offset from slave definition in CAM table.

dbMasterScaling = 1

Master scaling of the master definition in the CAM table.

dbSlaveScaling = 1

Slave scaling of the master definition in the CAM table

ECAM_VALUE_SRC_ENUM eMasterValueSource = eECAM_SET_VALUE

The Master source value is defined by the eMasterValueSource input parameter. It may be a Maestro parameter for target position, actual position (integer) or some kind of auxiliary. If the master axis operates in modulo mode, then the target position uses the

Maestro parameter as a source for the target modulated position (UU).

The Master value source dependant on whether set, actual or based on another value, according the ECAM_VALUE_SRC_ENUM enumerator.

eECAM_SET_VALUE = 0

eECAM_ACTUAL_VALUE = 1

eECAM_AUX_VALUE = 2

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.25 CamOut

MC_CamOut performs an MC_Stop on the slave axis. Refer to the section **7.6.8 MMC_CamOut** for details of the description, and scope.

int CamOut(
) throw (CMMCException)

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.26 CamTableUnload

This method unloads the ECAM table from the Maestro. Refer to the section **6.8.4 MMC_UnloadTable** for details of the description, and scope.

void CamTableUnload (
void
);

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

void

Function takes no parameters

17.4.27 CamTableAdd

This method appends points to the current ECAM table. Refer to the section **7.6.4 MMC_CamTableAdd** for details of the description, and scope.

```
int CamTableAdd(
MC_PATH_REF hMemHandle,
double *dbTable,
unsigned short usColumns,
unsigned long ulStartIndex,
unsigned long ulNumberOfPoints
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Remarks

The prerequisite to using this function is a call to MC_CamTableInit.

Scope

Loads CAM tables from an array in a user program into the Maestro. The user should be aware of the amount of columns used for each row (a point).

- Use an array of type double.
- The array must contain a sequence of rows (points), one by one.
- The columns order must be as follows: master, slave, curve type.
- Each row must contain the slave position. Master position and curve type are optional.
- If the Master gap is fixed then a row contains no master column, otherwise it does.
- If the curve type parameter is defined by user then a special column for curve type must be supplied, otherwise it must not.

Function Parameters

MC_PATH_REF hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle can have integer values.

**dbTable*

Pointer to the Table of values. Table of limited length in values, limited to 170 values max.

usColumns

Number of columns in the array. +ve number

ulStartIndex

Index to manually append from. Any +ve values accepted. Used only in manual mode.

ulNumberOfPoints

Number of points in the rows to append. Any +ve values accepted.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.28 CamTableAddEx

The CamTableAddEx function is used to add an unlimited number of rows to an existing table.

```
int CMMCMotionAxis::CamTableAddEx(
MC_PATH_REF hMemHandle,
double *dbTable,
unsigned shor usColumns,
unsigned long ulNumberOfPoints
) throw (CMMCException)
```

Source GMAS\includes\CPP\MMCMotionAxis.h

Remarks

The prerequisite to using this function is a call to MC_CamTableInit. This API allows users to add unlimited number of rows.

Scope

Loads CAM tables from an array in a user program into the Maestro. The user should be aware of the amount of columns used for each row (a point).

- Use an array of type double.
- The array must contain a sequence of rows (points), one by one.
- The columns order must be as follows: master, slave, curve type.
- Each row must contain the slave position. Master position and curve type are optional.
- If the Master gap is fixed then a row contains no master column, otherwise it does.
- If the curve type parameter is defined by user then a special column for curve type must be supplied, otherwise it must not.

Function Parameters

MC_PATH_REF hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle can have integer values.

**dbTable*

Pointer to the Table of values. Table of unlimited limited length in values.

usColumns

Number of columns depends on ucIsFixedGap and eCurveType as input parameters of CamTableInit.

ulNumberOfPoints

Number of points in the rows to append. Any +ve values accepted.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.4.29 CamTableSet

This method sets the number of points and number of columns in the array of the ECAM table. Refer to the section **7.6.6 MC_CamTableSet** for details of the description, and scope.

```
int CamTableSet(
MC_PATH_REF hMemHandle,
double *dbTable,
unsigned short usColumns,
unsigned long ulStartIndex,
unsigned long ulNumberOfPoints
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCMotionAxis.h

.NET Definition

Function Parameters

MC_PATH_REF hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle can have integer values.

dTable

Pointer to the Table of values. Table of limited length in values, limited to 170 values max.

usColumns

Number of columns in the array. +ve number

ulStartIndex

Index to manually append from. Any +ve values accepted. Used only in manual mode.

ulNumberOfPoints

Number of points in the rows to append. Any +ve values accepted.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis

17.5 The DLLMMCPP_API MMC_MOTIONPARAMS_GROUP class

The class DLLMMCPP_API MMC_MOTIONPARAMS_GROUP wraps the multiple axes functions detailed in the section **5.10 Multiple Axes Motion Control**. The diagram in **Figure 17-6** describes the heirarchial structure of the classes and type definitions associated with the CMMCGroupAxis.

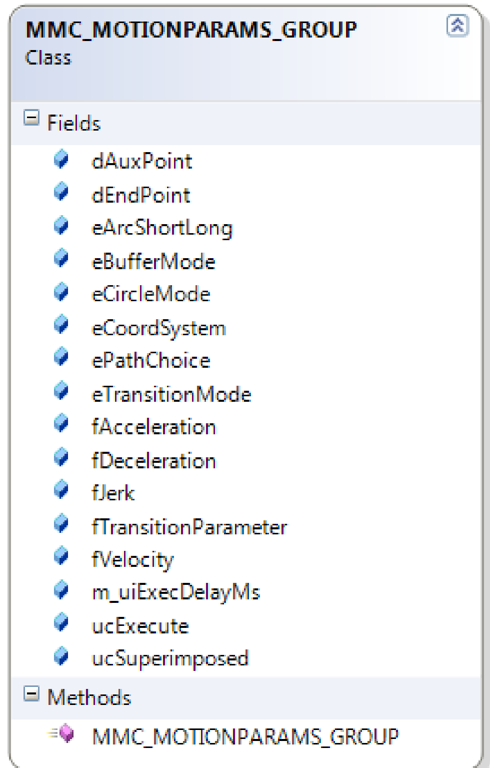


Figure 17-5 MMC_MOTIONPARAMS_GROUP class Fields diagram

The class DLLMMCPP_API MMC_MOTIONPARAMS_GROUP retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

It should be noted that Private functions and their operation should be transparent to the user, and are not for general application by the user.

The detailed class view shown in Figure 17-5, describes the fields and methods associated with the DLLMMCPP_API MMC_MOTIONPARAMS_GROUP class. These are generally default parameters, which can be operated using their default values. However if the user wishes to change the defaults, refer to the relevant parameter section in the manual.

17.5.1 MMC_MOTIONPARAMS_GROUP()

Defines the group motion parameters for arrays of drives.

```
public:
        MMC_MOTIONPARAMS_GROUP();
double dAuxPoint[NC_MAX_NUM_AXES_IN_NODE];
double dEndPoint[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
NC_ARC_SHORT_LONG_ENUM eArcShortLong;
NC_PATH_CHOICE_ENUM ePathChoice;
NC_CIRC_MODE_ENUM eCircleMode;
unsigned int m_uiExecDelayMs;
unsigned char ucSuperimposed;
unsigned char ucExecute;
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

dAuxPoint[NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dAuxPoint can have vector array [1….3] double values in a technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dEndPoint[NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of absolute end point positions for each dimension in the coordinate system specified by the input signal CoordSystem, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dEndPoint is a 2D or 3D double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fVelocity

Value of the maximum velocity (not necessarily reached) in which the path is defined. Any positive float value in u/s

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^{2}.

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

fTransitionParameter [NC_MAX_NUM_AXES_IN_NODE]

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section **5.10.1 Coordinate System and kinematic transformation**.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

NC_TRANSITION_MODE_ENUM eTransitionMode

Define the supported NC_TRANSITION_MODE_ENUM enumerator transition modes. Refer to the section **5.11 Multiple Axes Motion Control - Transition and Buffer Modes** and options below. The options are:

MC_TM_NONE_MODE = 0,
MC_TM_MAX_VELOCITY_MODE = 1, Not supported at this time
MC_TM_DEFINED_VELOCITY_MODE = 2,
MC_TM_CORNER_DISTANCE_MODE = 3,
MC_TM_MAX_CORNER_DEVIATION_MODE = 4,
MC_TM_SWITCH_RADIUS_MODE = 5,
MC_TM_CORNER_DIST_TC_POLYNOM = 6,
MC_TM_CORNER_DIST_CV_POLYNOM3 = 7,
MC_TM_CORNER_DIST_CV_POLYNOM5 = 8,
MC_TM_CORNER_DEVIATION_MODE_PLN6 = 9,
MC_TM_CORNER_DIST_CV_POLYNOM5_NAXES = 10,
MC_TM_LAST_MODE

MC_BUFFERED_MODE_ENUM eBufferMode

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1
MC_BUFFERED_MODE = 2
MC_BLENDING_LOW_MODE = 3
MC_BLENDING_PREVIOUS_MODE = 4
MC_BLENDING_NEXT_MODE = 5
MC_BLENDING_HIGH_MODE = 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

NC_ARC_SHORT_LONG_ENUM eArcShortLong

Defines the types of supported arc length. The NC_ARC_SHORT_LONG_ENUM enumerator options are:

MC_NONE_ARC_CHOICE = 0
MC_SHORT = 1
MC_LONG = 2

NC_PATH_CHOICE_ENUM ePathChoice

Defines the NC_PATH_CHOICE_ENUM enumerator types of supported path choice. The option are:

MC_NONE_PATH_CHOICE = 0
MC_CLOCKWISE = 1
MC_COUNTERCLOCKWISE = 2

NC_CIRC_MODE_ENUM eCircleMode

Defines the types of supported circular modes in 2D. Refer to the section **5.10.1 Coordinate System and kinematic transformation**. The NC_CIRC_MODE_ENUM enumerator options are:

MC_NONE_CIRC_MODE = 0

MC_BORDER_CIRC_MODE= 1

MC_CENTER_CIRC_MODE = 2

MC_RADIUS_CIRC_MODE = 3

MC_ANGLE_CIRC_MODE = 4

m_uiExecDelayMs

The delay in execution of the next action (in msecs). Any +ve integer value.

ucSuperimposed

Whether the option to superimpose is operated or not. Values accepted are Boolean TRUE/FALSE.

ucExecute

Start the execution command. Boolean TRUE/FALSE values.

17.6 The CMMCGroupAxis class

The class CMMCGroupAxis wraps the multiple axes functions detailed in the section **5.10 Multiple Axes Motion Control**.

```
class DLLMMCPP_API CMMCGroupAxis: public CMMCAxis {
public:
     CMMCGroupAxis();
     virtual ~CMMCGroupAxis();

     CMMCGroupAxis(CMMCGroupAxis& axis);
```

The diagram in **Figure 17-6** describes the heirarchial structure of the classes and type definitions associated with the CMMCGroupAxis.

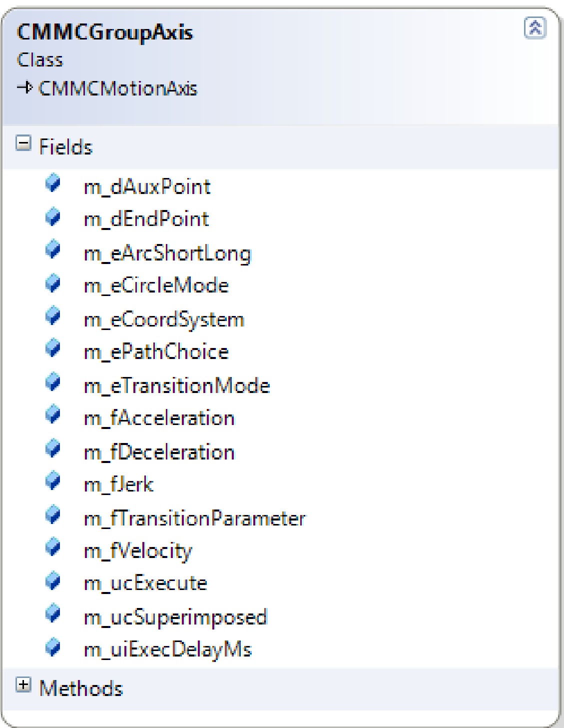


Figure 17-6 CMMCGroupAxis class Fields diagram

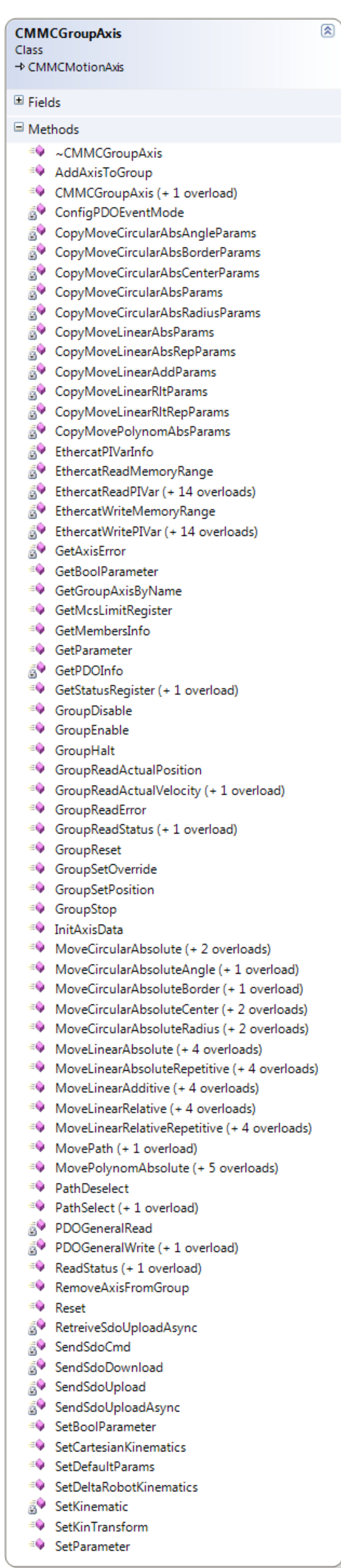


Figure 17-7 CMMCGroupAxis class Methods diagram

The class CMMCGroupAxis retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

It should be noted that Private functions and their operation should be transparent to the user, and are not for general application by the user.

The detailed class view shown in Figure 17-7, describes the fields and methods associated with the CMMCGroupAxis class. These are generally default parameters, which can be operated using their default values. However if the user wishes to change the defaults, refer to the relevant parameter section in the manual.

17.6.1 CMMCGroupAxis Class Functions Code Example 1

```
/*
============================================================================
Collection of Gmas API functions (Set #3)
Examples for document: "G-MAS Administrative and Motion API.pdf".
12Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection          gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCSingleAxis          AxisA,    AxisB;
     unsigned short               AxisARef, AxisBRef;

     CMMCGroupAxis           Group;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

    char  * delimit = "============================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroConnection(int);
     void      ConnectionTypeAndNum(void);
     void      SendReciveFromEthercat(int NumAmp);
     void      SetGetDefDigOutput(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
     int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus);
     void      EndMotionEventCB(unsigned short usAxisRef);
     void ModbusWrite_Received();
     void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
```

```
/*================================ Administration functions STR ===================================*/

int       main(int)
// ==============
{
     int  trace = 1;

     printf("\n %s", delimit);
     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

     try
     {
          SnroConnection(trace++);
          SnroMoveAbsolute(trace++);
          SnroEnableDisableMotionEndedEvent(trace++);
          SnroDepthName(trace++);

     }
     catch (CMMCException excp)
     {
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          exit(0);
     }

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//=====================================================================
{
                int end_of = 0;
                int iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
          end_of = 1;
/* Read Axis Status command server for specific Axis */
          ulState = sng_axis->ReadStatus();
          if (!(ulState & break_state))
          {
               end_of = 0;

               WAIT_SLEEP_MILLI(20)
          }
     }

//        MMC_SHOWNODESTAT_IN   showin;
//        MMC_SHOWNODESTAT_OUT showout;
//        MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

     return 0;
}


//   15.5.2.   RegisterRTE    Page 1274
void initAdminSingleAxis(void)
// ==============================
{
     int       iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;
```

```
/* CallbackFunc in ConnectIPCEx call if there    */
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Put Null param Val for no CallbackFunc        */
/*   ComHndl = gConn.ConnectIPCEx(iEventMask, NULL); */
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */


/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk     = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute     = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


void initAdminMultiAxis()
// =========================
{
//  Source class:
//       MMC_CONNECT_HNDL         ComHndl;
//       CMMCSingleAxis           AxisA,    AxisB;
//       CMMCGroupAxis            Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// =============================
{
     MMC_CloseConnection(ComHndl) ;
}


     void endAdminMultiAxis(void)
// ================================
{
//  Source class:
//       CMMCGroupAxis  Group;

     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
     Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================= Administration functions END ====================================*/

/*================================= Scenario functions STR ========================================*/
```

```
void SnroMoveAbsolute(int trace)
// ================================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

     MoveAbsoluteMoves();

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void SnroEnableDisableMotionEndedEvent(int trace)
// =================================================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);
                              /* Register the callback function for Modbus and Emergency: */
     gConn.RegisterEventCallback(MMCPP_MODBUS_WRITE,(void*)ModbusWrite_Received) ;
     gConn.RegisterEventCallback(MMCPP_EMCY,(void*)Emergency_Received) ;

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

     EnableDisableMotionEndedEvent();

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
     endAdminSingleAxis();

     gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

//   15.4.14.  GroupEnable    Page 1258
//   15.4.15.  GroupDisable   Page 1259
void SnroDepthName(int trace)
// ============================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     initAdminMultiAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

     Group.GroupEnable();

     DepthName();

     Group.GroupDisable();

     AxisB.PowerOff(MC_BUFFERED_MODE);
     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

     endAdminMultiAxis();
```

```
    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


void SnroConnection(int trace)
// ==============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();

    ConnectionTypeAndNum();

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

/*================================= Example functions STR =========================================*/


void EnableDisableMotionEndedEvent(void)
// ========================================
{
    int  loopInd;

printf("\n Function: %s:", __func__);
    for (loopInd = 0; loopInd < 2; loopInd++)
    {
        if ((loopInd % 2) == 0)
        {
            printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
            AxisA.EnableMotionEndedEvent();
        }
        else
        {
            printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
            AxisA.DisableMotionEndedEvent();
        }

        printf("\n +++++++++ Motion started...");
        MoveAbsoluteMoves();
        WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
        printf("\n +++++++++ Motion End \n");
    }
}


//   15.4.14.  GroupEnable    1258
//   15.4.15.  GroupDisable   1259
void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
    iVal1 = AxisB.GetFbDepth();

    Group.GroupDisable();
    AxisB.PowerOff(MC_BUFFERED_MODE);

    iVal2 = AxisB.GetFbDepth();
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
    iVal3 = AxisB.GetFbDepth();

    printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
    Group.GroupEnable();
```

```
    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

     iVal3 = Group.GetGroupAxisByName("v01");    /* Expected 256 */

     printf("\n +++++ Reff: a01=%d a02=%d, v01=%d +++++", iVal1, iVal2, iVal3);
/*
*    iVal2 = Group.GetGroupAxisByName("v02");
*/
}


void MoveAbsoluteMoves(void)
// ============================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:        */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0      */
/* update default speed to 5000000          */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters            */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity      */
/* v=100000    which become the new def V        */
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0       */
/* new def v=80000                               */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:    */
/*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=            */
/* MC_BUFFERED_MODE (def)    */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
/* Move abs to 350000 with parameters from  */
/* above command which become the default:  */
/*       Speed 500000, Acc=1000000               */
/*       Dec=1500000, Jerk=20000000,        */
/*       buffer mode=MC_BUFFERED_MODE (def)  */
     AxisA.MoveAbsolute(350000);
}


int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{

    printf("\n *********** STR Func: %s *********** ", __func__);

                              /* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
```

```
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
          printf("\n Underflow event received ");
          break ;
     case MODBUS_WRITE_EVT:
          printf("\n ModBus Write event received ");
          break ;
     case TOUCH_PROBE_ENDED_EVT:
          printf("\n Touch Probe event received ");
          break ;
     default:
          printf("\n Default.... Whatever arrived event received ");
          break;
     }

    printf("\n *********** END Func: %s *********** ", __func__);
     fflush(stdout); fflush(stderr);

     return 1 ;
}

int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short sErrorID,
unsigned short usStatus)
// =========================================================================================================
{
     printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d, bye\n",
             msg, usAxisRef, sErrorID, usStatus);
     fflush(stdout); fflush(stderr);

     MMC_CloseConnection(uiConnHndl);
     exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
     printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
     printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
     fflush(stdout); fflush(stderr);
}


/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ===========================
{
     printf("\n %s Received ", __func__) ;
     fflush(stdout); fflush(stderr);
}


                              /* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ==================================================================
{
     printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
     fflush(stdout); fflush(stderr);
}
```

```
//   16.6.10. SetHeartBeatConsumer     1248
//   16.7.3.  GetNetworkInfo           1253
enum
{
     eCOMM_TYPE_NONE = 0,
     eCOMM_TYPE_ETHERCAT,
     eCOMM_TYPE_CAN
};


void ConnectionTypeAndNum(void)
// ===============================
//
{
     unsigned int                 uiHeartbeatTimeFactor;
     int                               rt;
     int                               NumFoundAmp;
     char *                            cErrotStr;
     CMMCNetwork                       CNet;
     MMC_NETWORKINFO_OUT          CanOutParams;       // Can drv connection
     MMC_GETCOMMSTATISTICS_OUT    EthCatOutParams;    // Ethcat drv connection


     printf("\n Function: %s ", __func__);

     CNet.SetConnHndl(ComHndl);
                              /* Connection type - CAN/EtherCAT */
     rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);

     if (rt == eCOMM_TYPE_ETHERCAT)
     {
                              /* !!! ETERCAT !!!*/
          rt = CNet.GetCommStatistic(EthCatOutParams);
          if (rt != 0)
          {
               cErrotStr = "EtherCat failed, get statistic";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = (int)EthCatOutParams.usNumOfSlaves;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through ETERCAT net. ", NumFoundAmp);

                              /* Send and recive from UDP soket to driv */
          SendReciveFromEthercat(NumFoundAmp);
     }
     else if (rt == eCOMM_TYPE_CAN)
     {
/* !!! CAN !!!*/
/* !!!  hard bit should be set in resource file*/
/* take from the resource file and actual connected...   */
          uiHeartbeatTimeFactor = 1;    /* On for Every Cycle */
          gConn.SetHeartBeatConsumer(uiHeartbeatTimeFactor);

         rt = CNet.GetNetworkInfo  (CanOutParams);
          if (rt != 0)
          {
               cErrotStr = "Can failed, get NetworkInfo";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = CanOutParams.iNumOfActiveNodes;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through CAN net. ", NumFoundAmp);
     }

/* Get & Set Default Mapping Digital Output */
     SetGetDefDigOutput();

     return;

ConnectionTypeAndNum_exit_err:
     printf("\n>>> %s: *** %s %d ", __func__, cErrotStr, rt);
     return;
}

/* Should create sockets according to actual number of amplifire (param NumAmp)     */
```

```
/* ...for demo (examples etc...) assume at least two Amp. exist.                    */
//   16.18.1.  Create         1302
//   16.18.2.  SendTo         1303
//   16.18.3.  ReceiveFrom    1304
//
//   16.18.11. ElmoGetArray        1310
//   16.18.12. ElmoGetParameter    1311

void SendReciveFromEthercat(int NumAmp)
// =======================================
{
     int            rt_val = 0;
    int             iBv;
     bool           bWait;
     float          fValue;
                         /* IPC type of app. */
     char           sAxisName[50] ;
     CMMCUDP   cUDP1,
                    cUDP2;
     CMMCEoE        gEoe;



     printf("\n Function: %s ", __func__);

     rt_val = cUDP1.Create("192.168.1.5", 5001, 0); /* Ip of first drive (Gmas "last part IP" + 1) */
     rt_val = cUDP1.SendTo("vr\r", 3);
     rt_val = cUDP1.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 0 Version: <%s> ", sAxisName);

     rt_val = cUDP2.Create("192.168.1.6", 5001); /* Ip of second Gmas     */
     rt_val = cUDP2.SendTo("vr\r", 3) ;
     rt_val = cUDP2.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 1 Version: <%s> ", sAxisName);


     rt_val = gEoe.Connect("192.168.1.5", 5001, bWait);
     if (bWait == true)
     {
          usleep(10000);                        /* Wait 10 mili */
          if (gEoe.IsWritable() == false)
          {
               printf("\n>>> %s: *** Amp (Drv) connection it not writable... ", __func__);
          }
     }
/* AN[6] is command for get PsHv   */
/* return 0 on succceed otherwise 1.*/
     rt_val = gEoe.ElmoGetArray("AN", 6, fValue);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get EthCat Driver psHv AN[6] ", __func__);
     }
     else
     {
          printf("\n>>> Max Power Supplay High Voltage for this driver (ip=192.168.1.5) is %3.1f ", fValue);
     }

/* BV - Maximum Motor DC Voltage (return int)   */
/* return 0 on succeed, otherwise 1.             */
     rt_val = gEoe.ElmoGetParameter("BV", iBv);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get Bus Driver HV ", __func__);
     }
     else
     {
          printf("\n>> Actual Bus Driver (ip=192.168.1.5) Hv is %d ", iBv);
     }

     gEoe.Close();
}
```

```
//   16.3.27.  GetDigOutputs32Bit  1168
//   16.3.29.  SetDigOutputs32Bit  1169
                              /* Get & Set Default Mapping Digital Output */
void SetGetDefDigOutput(void)
// =============================
{
unsigned long       ulDigOutputs32bit;



     printf("\n Function: %s ", __func__);

                              /* Read Digital output group 0 state */
     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: B4 action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned int)ulDigOutputs32bit);


/* Change specific bit state of digital Output group 0 */
     if ((ulDigOutputs32bit & 0x10000) != 0x00000)
     {
/* ReSet specific bit of Digital output group 0 to state "0" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit & 0xfffeffff);   /* E.g: disconnedt ps... */
     }
     else
     {
/* Set specific bit of Digital output group 0 to state "1" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit | 0x10000);     /* E.g: connect ps... */
     }


     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: Aft action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned int)ulDigOutputs32bit);
}

/*================================= Example functions END ===========================================*/

/*================================= Output STR ============================================*/
#ifdef PROGRAM_OUTPUT
#endif /* PROGRAM_OUTPUT */
/*================================= Output END ============================================*/
```

17.6.2 CMMCGroupAxis Class Functions Code Example 2

```
/*
============================================================================
Collection of Gmas API functions (Set #2)
Examples for document: "G-MAS Administrative and Motion API.pdf".
09Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"
#define STAR_DIM                       2


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection          gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCHostComm            cHost ;

     CMMCSingleAxis          AxisA,    AxisB;
     unsigned short               AxisARef, AxisBRef;

     CMMCGroupAxis            Group;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

    char  * delimit = "===========================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroMbusfunc(int);
     void mBusReadWriteHoldReg(void);
     void mBusReadWriteCoil(void);
     void mBusReadWriteInput(void);
     void moveAxisAAndRepActualParamOnMbusPos(void);
     void      RepAxisAActualParamOnMbus(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
```

```
	int	OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short 
sErrorID, unsigned short usStatus);
	void	EndMotionEventCB(unsigned short usAxisRef);
	void ModbusWrite_Received();
	void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
	int	mapConfigPdo(void);


/*=============================== Administration functions STR ===================================*/
int	main(int)
// ==============
{
	int  trace = 1;

	printf("\n %s", delimit);
	printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

	try
	{
		SnroMoveAbsolute(trace++);
		SnroEnableDisableMotionEndedEvent(trace++);
		SnroDepthName(trace++);

		SnroMbusfunc(trace++);
	}
	catch (CMMCException excp)
	{
		printf("\n %s", delimit);
		printf("\n %s", delimit);
		printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(), 
(short)excp.error(), excp.status());
		printf("\n %s", delimit);
		printf("\n %s", delimit);
		exit(0);
	}

	printf("\n End of %s ", __FILE__);
	printf("\n %s\n\n", delimit);
	return 0;
}


//	15.11.3.  CMMCNode::ReadStatus	1374
int	WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//======================================================================
{
			int end_of = 0;
			int iCount = 0;
	unsigned int	ulState;

	while( ! end_of)
	{
		iCount ++;
		end_of = 1;
/* Read Axis Status command server for specific Axis */
		ulState = sng_axis->ReadStatus();
		if (!(ulState & break_state))
		{
			end_of = 0;

			WAIT_SLEEP_MILLI(20)
		}
	}

//		MMC_SHOWNODESTAT_IN  showin;
//		MMC_SHOWNODESTAT_OUT showout;
//		MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

	return 0;
}
```

```
// 15.6.5.     CMMCConnection::ConnectIPCEx  1335

// 15.6.13.    CMMCConnection::CallbackFunc  1345
void initAdminSingleAxis(void)
// ==============================
{
     int iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;



/* CallbackFunc in ConnectIPCEx call if there    */
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */

/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk     = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute     = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


//   15.4.21.  CMMCGroupAxis::AddAxisToGroup 1285
void initAdminMultiAxis()
// =========================
{
//   MMC_CONNECT_HNDL         ComHndl;
//   CMMCSingleAxis           AxisA,    AxisB;
//   CMMCGroupAxis            Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// =============================
{
     MMC_CloseConnection(ComHndl) ;
}
```

```
//   15.4.5.   CMMCGroupAxis::RemoveAxisFromGroup     1269
     void endAdminMultiAxis(void)
// ================================
{

//   CMMCGroupAxis   Group;

     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
     Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================ Administration functions END ====================================*/


/*================================ Scenario functions STR ==========================================*/
void SnroMoveAbsolute(int trace)
// ================================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

     MoveAbsoluteMoves();

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


//   15.6.14.  CMMCConnection::RegisterEventCallback  1249
void SnroEnableDisableMotionEndedEvent(int trace)
// ================================================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

     EnableDisableMotionEndedEvent();

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
     endAdminSingleAxis();

     gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


void SnroDepthName(int trace)
// ============================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     initAdminMultiAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
```

```
     Group.GroupEnable();

     DepthName();

     Group.GroupDisable();

     AxisB.PowerOff(MC_BUFFERED_MODE);
     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

     endAdminMultiAxis();
     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


//   16.8.5.   cHost::MbusIsRunning     1258
//   16.8.1.   cHost::MbusStartServer   1255
//   16.8.2.   cHost::MbusStopServer    1256
void SnroMbusfunc(int trace)
// ============================
{
static    bool      mBusServerState;


     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
/* Starting MBus server            */
/* map read/write user req. to Mbus     */
/* area & responsible to Mbus commu.*/
     mBusServerState = cHost.MbusIsRunning(ComHndl);
     if (mBusServerState)
     {
          printf("\n ... ModBus Server is already running ");
     }
     else
     {
          printf("\n ... ModBus Server is NOT running - activate it ");
     }
/* Call to StartServer even if it   */
/* already running, it not start    */
/* the server because it alrady so  */
/* but for initialize init struct */
     cHost.MbusStartServer(ComHndl, 1) ;
                              /* Mode Bus functions examples.        */
     mBusReadWriteHoldReg();
/* Move Axis A and report it poss.  */
/* on Mbus.*/
     moveAxisAAndRepActualParamOnMbusPos();
     mBusReadWriteCoil();
     mBusReadWriteInput();

/* If I was the one who activate the   */
/* Mbus server I stop it when I finish.*/
     if (! mBusServerState)
     {
/* Stop MBus server */
          cHost.MbusStopServer();
          printf("\n ... Stop ModBus Server ");
     }
     else
     {
          printf("\n ... Left ModBus Server running ");
     }

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
```

```
/*================================ Scenario functions END ==========================================*/


/*================================ Example functions STR ===========================================*/

void EnableDisableMotionEndedEvent(void)
// ========================================
{
     int  loopInd;

printf("\n Function: %s:", __func__);
     for (loopInd = 0; loopInd < 2; loopInd++)
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}

void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

// iVal3 = AxisA.GetAxisByName("A01"); /* It case sensitive - Not define - exception... */

     iVal1 = Group.GetGroupAxisByName("v01");    /* Expected 256 */
/*
 *   iVal2 = Group.GetGroupAxisByName("v02");
 */
}


void MoveAbsoluteMoves(void)
// ============================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:    */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0 */
```

```
/* update default speed to 5000000     */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters       */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity */
/* v=100000    which become the new def V    */
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0   */
                              /* new def v=80000                     */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:    */
                              /*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=            */
/* MC_BUFFERED_MODE (def)    */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
                              /* Move abs to 350000 with parameters from  */
                              /* above command which become the default:  */
                              /*        Speed 500000, Acc=1000000              */
                              /*        Dec=1500000, Jerk=20000000,       */
                              /*        buffer mode=MC_BUFFERED_MODE (def)  */
     AxisA.MoveAbsolute(350000);
}


//   16.6.11.  CMMCConnection::CallbackFunc  1248
int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{
//   int  ind;

    printf("\n *********** STR Func: %s *********** ", __func__);

//    printf("\n");
//    for (ind=0; ind < recvBufferSize; ind++)
//    {
//        printf("   recvBuffer[%d]=%d", ind, recvBuffer[ind]);
//    }
//    printf("\n");

/* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
          printf("\n Underflow event received ");
          break ;
     case MODBUS_WRITE_EVT:
          printf("\n ModBus Write event received ");
          break ;
```

```
    case TOUCH_PROBE_ENDED_EVT:
         printf("\n Touch Probe event received ");
         break ;
    default:
         printf("\n Default.... Whatever arrived event received ");
         break;
    }

   printf("\n *********** END Func: %s *********** ", __func__);
    fflush(stdout); fflush(stderr);

    return 1 ;
}


//
// Needs registeration, E.g:  CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);
//
int      OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short sErrorID,
unsigned short usStatus)
// ==========================================================================================================
{
    printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d, bye\n",
            msg, usAxisRef, sErrorID, usStatus);
    fflush(stdout); fflush(stderr);

    MMC_CloseConnection(uiConnHndl);
    exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
    printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
    printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
    fflush(stdout); fflush(stderr);
}

/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ===========================
{
    printf("\n %s Received ", __func__) ;
    fflush(stdout); fflush(stderr);
}

/* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ==================================================================
{
    printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
    fflush(stdout); fflush(stderr);
}


#define   MODBUS_UPDATE_START_INDEX             0
#define   MODBUS_UPDATE_CNT                4
                              /* WRITE INDEXS extend along two indexs */
#define MODBUS_STR_REP_ACT_VAL_INDX          (MODBUS_UPDATE_START_INDEX+MODBUS_UPDATE_CNT)
#define   MBUS_WAIT_FOR_USER                    30000

//   15.8.4.   cHost::MbusReadHoldingRegisterTable         1257
//   15.8.5.   cHost::MbusWriteHoldingRegisterTable   1258
void mBusReadWriteHoldReg(void)
// ===============================
{
         int       startRef;
         int       refCnt;
         int            loopCount = 0;

    MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN     mbus_write_in;
    MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT     mbus_read_out;
```

```
	printf("\n\n Function: %s: ", __func__);

	memset(mbus_write_in.regArr,0x0, 250) ;

	startRef = MODBUS_UPDATE_START_INDEX;
	refCnt    = MODBUS_UPDATE_CNT;
	mbus_write_in.startRef        = startRef;
	mbus_write_in.refCnt          = refCnt;
	mbus_write_in.regArr[0]  = 0xf0f0;
	mbus_write_in.regArr[1]  = 0x0f0f;
	mbus_write_in.regArr[2]  = 0x5a5a;
	mbus_write_in.regArr[3]  = 0xa5a5;


	printf("\n ... You can activate the EAS and look on EAS Mbus reg 1-4 ");

	printf("\n ... Going deal with Mbus H.Reg. EAS ind 1-4 ");
	loopCount = 0;
	do
	{
		loopCount++;
/* write into mBus Holding register */
/* data on reg. ind 0-3            */
		cHost.MbusWriteHoldingRegisterTable(mbus_write_in);
		printf("\n ... Mbus Write  H.Reg ind 0-3: %8x %8x %8x %8x ",
								(int)mbus_write_in.regArr[0],
								(int)mbus_write_in.regArr[1],
								(int)mbus_write_in.regArr[2],
								(int)mbus_write_in.regArr[3]);
		printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);

		WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
/* read mbus H.Reg start from            */
/* location startRef along refCnt reg*/

		cHost.MbusReadHoldingRegisterTable(startRef, refCnt, mbus_read_out);
		printf("\n ... Mbus Readed H.Reg ind 0-3: %8x %8x %8x %8x ",
								(int)mbus_read_out.regArr[0],
								(int)mbus_read_out.regArr[1],
								(int)mbus_read_out.regArr[2],
								(int)mbus_read_out.regArr[3]);

		mbus_write_in.regArr[0] = ~(mbus_read_out.regArr[0]);
		mbus_write_in.regArr[1] = ~(mbus_read_out.regArr[1]);
		mbus_write_in.regArr[2] = ~(mbus_read_out.regArr[2]);
		mbus_write_in.regArr[3] = ~(mbus_read_out.regArr[3]);

	} while (loopCount < 3);
}


//   16.8.6.   cHost::MbusReadCoilsTable     1259
//   16.8.7.   cHost::MbusWriteCoilsTable    1260
void mBusReadWriteCoil(void)
// ============================
{
		int       startRef;
		int       refCnt;

		int            loopCount = 0;

	MMC_MODBUSWRITECOILS_IN       stInParams_Coil;
	MMC_MODBUSREADCOILS_OUT       stOutParams_Coil;


	printf("\n\n Function: %s: ", __func__);

	startRef = MODBUS_UPDATE_START_INDEX;
	refCnt    = MODBUS_UPDATE_CNT;
	stInParams_Coil.startRef= startRef;
	stInParams_Coil.refCnt   = refCnt;
```

```
	stInParams_Coil.coilsArr[0] = (char)0xf0;
	stInParams_Coil.coilsArr[1] = (char)0x0f;
	stInParams_Coil.coilsArr[2] = (char)0x5a;
	stInParams_Coil.coilsArr[3] = (char)0xa5;

	printf("\n ... Going deal with Mbus Coils EAS ind 1-4 ");
	loopCount = 0;
	do
	{
		loopCount++;
/* write into mBus Coil register */
/* data on reg. ind 0-3           */
		cHost.MbusWriteCoilsTable(stInParams_Coil);
		printf("\n ... Mbus Write  Coil ind 0-3: %8x %8x %8x %8x ",
						(int)stInParams_Coil.coilsArr[0],
						(int)stInParams_Coil.coilsArr[1],
						(int)stInParams_Coil.coilsArr[2],
						(int)stInParams_Coil.coilsArr[3]);
		printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
		WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
/* read mbus Coil start from       */
/* location startRef along refCnt reg*/
		cHost.MbusReadCoilsTable(stInParams_Coil.startRef, stInParams_Coil.refCnt, stOutParams_Coil);
		printf("\n ... Mbus Readed Coil ind 0-3: %8x %8x %8x %8x ",
						(int)stOutParams_Coil.coilsArr[0],
						(int)stOutParams_Coil.coilsArr[1],
						(int)stOutParams_Coil.coilsArr[2],
						(int)stOutParams_Coil.coilsArr[3]);

		stInParams_Coil.coilsArr[0] = ~(stOutParams_Coil.coilsArr[0]);
		stInParams_Coil.coilsArr[1] = ~(stOutParams_Coil.coilsArr[1]);
		stInParams_Coil.coilsArr[2] = ~(stOutParams_Coil.coilsArr[2]);
		stInParams_Coil.coilsArr[3] = ~(stOutParams_Coil.coilsArr[3]);

	} while (loopCount < 3);
}


//   16.8.8.   cHost::MbusReadInputsTable   1261
void mBusReadWriteInput(void)
// =============================
{
	int		startRef;
	int		refCnt;

	int			loopCount = 0;

	MMC_MODBUSREADINPUTS_OUT stOutParams_Input;


	printf("\n\n Function: %s: ", __func__);

	printf("\n ... Going deal with Mbus Inputs EAS ind 1-4 ");
	startRef= MODBUS_UPDATE_START_INDEX;
	refCnt	= MODBUS_UPDATE_CNT;
	loopCount = 0;
	do
	{
		loopCount++;
		cHost.MbusReadInputsTable(startRef, refCnt, stOutParams_Input);
		printf("\n ... Mbus Readed inputs ind 0-3: %8x %8x %8x %8x ",
				(int)stOutParams_Input.inputsArr[0], (int)stOutParams_Input.inputsArr[1],
(int)stOutParams_Input.inputsArr[2], (int)stOutParams_Input.inputsArr[3]);

		printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
		WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
	} while (loopCount < 3);
}


#define	UL_TO_MBUS_STRUCT(modbus_write_val, aux_ul_p, mod_bus_idx)			\
	{										\
			aux_ul_p = (unsigned long *)(& mbus_write_in.regArr[mod_bus_idx]);	\
```

```
                * aux_ul_p = modbus_write_val;                                    \
        }                                                                         \
//   16.3.20.  CMMCSingleAxis::GetActualPosition 1164
//   16.3.21.  CMMCSingleAxis::GetActualVelocity 1164
//   16.3.22.  CMMCSingleAxis::GetActualTorque       1165
//   16.6.7.       CMMCSingleAxis::GetGlobalBoolParameter  1245
void RepAxisAActualParamOnMbus(void)
// ================================
{
     MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN     mbus_write_in;
                                   /* Auxiliary var for write to ModBus */
     unsigned  long*     ul_p;
     unsigned  long ul_val;
                    double     dActul;



     dActul = AxisA.GetActualPosition();
     ul_val = (unsigned long)dActul;
/* Put value into ModBus write array use 2 index poss (0 & 1).  */
     UL_TO_MBUS_STRUCT(ul_val, ul_p, 0);

/* Remmber about mapping & config !!! */
/* if the motor connected above ehercat it needs apropirate */
/* config setting (EAS). */
     dActul = AxisA.GetActualVelocity();
     ul_val = (unsigned long)dActul;
                              /* Put value into ModBus write array use 2 index poss (2 & 3).  */
     UL_TO_MBUS_STRUCT(ul_val, ul_p, 2);


/* Remmber about mapping & config (see GetActualVelocity) !!!   */
     dActul = AxisA.GetActualTorque();
     ul_val = (unsigned long)dActul;
/* Put value into ModBus write array use 2 index poss (4 & 5).  */
     UL_TO_MBUS_STRUCT(ul_val, ul_p, 4);


/* Index of modbus H.reg. for start write actal values. */
     mbus_write_in.startRef        = MODBUS_STR_REP_ACT_VAL_INDX;
/* Number of indexes to write - each long extend along     */
/* 2 indexs (2 for Pos, 2 for Vel, 2 for Torq              */
     mbus_write_in.refCnt = 6;
     cHost.MbusWriteHoldingRegisterTable(mbus_write_in) ;

     return ;
}


#define   eCOMM_TYPE_ETHERCAT      1
#define   eCOMM_TYPE_CAN           2
/* Move Axis A and report it Actual */
/* parameters on Mbus.                */
void moveAxisAAndRepActualParamOnMbusPos(void)
// ==============================================
{
                int ind,
                         rt;
     unsigned int   uiState,
                         uiDoneFlg;

     printf("\n\n Function: %s: ", __func__);

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


/* Connection type - CAN/EtherCAT */
     rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);
     if (rt == eCOMM_TYPE_ETHERCAT)
     {
          printf("\n --- Motor connection is via ETHERCAT rembber map & config for get actul param (Vel.
Torq.) ");
```

```
    }
    else if (rt == eCOMM_TYPE_CAN)
    {
        printf("\n --- Motor connection is via CAN ");
        if ( mapConfigPdo() )
        {
            printf("\n --- ERROR FAIL IN MAP PDO FOR CAN  !!! ");
        }
    }
    else
    {
        printf("\n --- ERROR UNKNOWN motor connection  !!! ");
    }

    printf("\n ... Start rep AxisA pos on Mbus");
    for(ind=0; ind<3; ind++)
    {
/* Move abs to: 400000, with parameters:         */
/*              Speed 300000, Acc=500000         */
/* Dec=1500000, Jerk=20000000, buffer mode=      */
/* MC_BUFFERED_MODE (def)         */
        AxisA.MoveAbsolute(400000*ind, 300000, 500000, 1500000, 20000000);
        do
        {
                        /* Write AxisA pos. to Modbus H.Reg */
            RepAxisAActualParamOnMbus();
            uiState = AxisA.ReadStatus();
            uiDoneFlg = uiState & NC_AXIS_STAND_STILL_MASK;
            if ( ! uiDoneFlg)
            {
                WAIT_SLEEP_MILLI(40)
            }
        } while(! uiDoneFlg);
    }
    printf("\n ... End rep AxisA pos on Mbus");

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
}


int  mapConfigPdo(void)
// ===================
{
    int  rc;
    unsigned short                AxisARef;


    MMC_CONFIGREGULARPARAMEVENTPDO3_IN CfgReg3In;
    MMC_CONFIGREGULARPARAMEVENTPDO3_OUT CfgReg3Out;

    CfgReg3In.ucEventGroup    = NC_COMM_EVENT_GROUP6;
    CfgReg3In.ucPDOCommParam= PDO_COM_PARAM_SYNC;

    AxisARef = AxisA.GetRef();

    rc = MMC_CfgRegParamEvPDO3Cmd(ComHndl, AxisARef, &CfgReg3In, &CfgReg3Out);
    if(rc != 0)
    {
    printf("\n **** config error %d \n",CfgReg3Out.sErrorID);
/* ... goto lbl_motor_off; */
    }

    return (rc);
}

/*================================ Example functions END ==========================================*/
```

17.6.3 CMMCGroupAxis Class Functions Code Example 3

The following is an example of group axes motion in **2-D** using the MoveCircularXXXXXXX set of functions.

```
/*
============================================================================
Collection of Gmas API functions (Set #4)
Examples for document: "G-MAS Administrative and Motion API.pdf".
12Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection      gConn;
     MMC_CONNECT_HNDL         ComHndl;

     CMMCSingleAxis      AxisX, AxisY, AxisZ;
     unsigned short           AxisXRef, AxisYRef, AxisZRef;

     CMMCGroupAxis       Group;
     double              S_Factor_For_Polynomial_Transition;
     MMC_MOTIONPARAMS_GROUP   ParamsGroup;

     MMC_SETKINTRANSFORM_IN   SetKin;

     double    StarPoi[][2] =
                              {
                              { 7500.,      0.},
                              {12500.,  13100.},
                              {  200.,   5600.},
                              {14700.,   5600.},
                              { 3340.,  13750.},
                              { 7500.,   7500.}   /* the Center cordination */
                              };

        char   * delimit = "==============================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int  WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
```

```
	void	endAdminSingleAxis(void);

	void initAdminMultiAxis2d();
	void endAdminMultiAxis2d(void);

	void SnroMoveComplex2d(int);
	void MoveLinearRelative(void);
	void MoveCircularLinarAbs(void);
	void MoveCircularAbsolute2d(void);
	void GroupOverrideAndPosition(void);

	int  CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
	int  OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short sErrorID, 
unsigned short usStatus);

/*======================== Administration functions STR ========================*/

int		main(int)
// ==============
{
	int  trace = 1;

	printf("\n %s", delimit);
	printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

	try
	{
		SnroMoveComplex2d(trace++);
	}
	catch (CMMCException excp)
	{
		printf("\n %s", delimit);
		printf("\n %s", delimit);
		printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(), (short)excp.error(), 
excp.status());
		printf("\n %s", delimit);
		printf("\n %s", delimit);
		exit(0);
	}

	printf("\n End of %s ", __FILE__);
	printf("\n %s\n\n", delimit);
	return 0;
}


int		WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//=======================================================================
{
			int end_of = 0;
			int iCount = 0;
	unsigned int   ulState;

	while( ! end_of)
	{
		iCount ++;
		end_of = 1;
/* Read Axis Status command server for specific Axis */
		ulState = sng_axis->ReadStatus();
		if (!(ulState & break_state))
		{
```

```
                end_of = 0;

                WAIT_SLEEP_MILLI(20)
            }
        }

//    MMC_SHOWNODESTAT_IN  showin;
//    MMC_SHOWNODESTAT_OUT showout;
//    MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

      return 0;
}


void WaitGrpDone(unsigned int groupStatusMsk)
// =============================================
{
      unsigned int   uiStatusRegister;

      uiStatusRegister = Group.GroupReadStatus();
      while((uiStatusRegister & groupStatusMsk) != groupStatusMsk)
      {
            WAIT_SLEEP_MILLI(2)
            uiStatusRegister = Group.GroupReadStatus();
      }
}


void initAdminSingleAxis(void)
// ==============================
{
      int  iEventMask;

      MMC_MOTIONPARAMS_SINGLE stSingleDefault;

      printf("\n Function: %s: ", __func__);

/* CallbackFunc in ConnectIPCEx call if there is no calling to 'RegisterEventCallback'*/
      iEventMask = 0x7fffffff;
      ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Put Null param Val for no CallbackFunc ComHndl = gConn.ConnectIPCEx(iEventMask, NULL); */
/* Should Not calling, called inside 'ConnectIPCEx'   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in,
&openudp_param_out); */


/* Register Run Time Error Callback function*/
      CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);


      AxisX.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
      stSingleDefault.fEndVelocity  =    0.0;
      stSingleDefault.dbDistance    =    100000.0;
      stSingleDefault.dbPosition    =    0.0;
      stSingleDefault.fVelocity     =    100000.0;
      stSingleDefault.fAcceleration =   2000000.0;
      stSingleDefault.fDeceleration =  10000000.0;
      stSingleDefault.fJerk     = 200000000.0;
/* MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
```

```
    stSingleDefault.eDirection     = MC_POSITIVE_DIRECTION;
    stSingleDefault.eBufferMode    = MC_BUFFERED_MODE;
    stSingleDefault.ucExecute      = 1;

    AxisX.SetDefaultParams(stSingleDefault);
}

//   15.4.4.   SetKinTransform
//   15.3.53.  SetParameter
void initAdminMultiAxis2d()
// ===========================
{
/* Source class:*/
/*   MMC_CONNECT_HNDL    ComHndl;
/*   CMMCSingleAxis      AxisX, AxisY;  */
/*   CMMCGroupAxis       Group;         */


    printf("\n Function: %s: ", __func__);

    AxisY.InitAxisData("a02", ComHndl);
    Group.InitAxisData("v01", ComHndl);

    AxisXRef = AxisX.GetRef();
    AxisYRef = AxisY.GetRef();
/* Set by default the EndPoint=StartPoint */
    ParamsGroup.dEndPoint[0] = 0.0;
    ParamsGroup.dEndPoint[1] = 0.0;


    ParamsGroup.fVelocity =   100000.0;
    ParamsGroup.fAcceleration =   8000000.0;
    ParamsGroup.fDeceleration =   8000000.0;
    ParamsGroup.fJerk =    100000000.0;
    ParamsGroup.eCoordSystem = MC_MCS_COORD;
    ParamsGroup.eBufferMode = MC_BUFFERED_MODE;
//   ParamsGroup.eTransitionMode = MC_TM_CORNER_DEVIATION_MODE_PLN6;
    ParamsGroup.eTransitionMode = MC_TM_NONE_MODE;
    ParamsGroup.fTransitionParameter[0] = 2000.0;
    ParamsGroup.ucExecute = 1;

    Group.SetDefaultParams(ParamsGroup);

/* Parameters for Kinematic Transformation */
    SetKin.eBufferMode = MC_BUFFERED_MODE;
    SetKin.eType[0] = NC_X_AXIS_TYPE;
    SetKin.eType[1] = NC_Y_AXIS_TYPE;

    SetKin.hNode[0] = AxisX.GetRef();
    SetKin.hNode[1] = AxisY.GetRef();

    SetKin.iMcsToAcsFuncID[0] = NC_TR_SHIFT_FUNC;
    SetKin.iMcsToAcsFuncID[1] = NC_TR_SHIFT_FUNC;

    SetKin.iNumAxes = 2;
    SetKin.ucExecute = 1;

    SetKin.ulTrCoef[0][0] = 1;
    SetKin.ulTrCoef[0][1] = 1;
    SetKin.ulTrCoef[0][2] = 0;
```

```
     SetKin.ulTrCoef[1][0] = 1;
     SetKin.ulTrCoef[1][1] = 1;
     SetKin.ulTrCoef[1][2] = 0;

     Group.SetKinTransform(SetKin);

/* Set the factor for Polynominal Transition*/
     S_Factor_For_Polynomial_Transition = 0.4;   // The default is 1.4

/* SetParameter(double dbValue, MMC_PARAMETER_LIST_ENUM eNumber, int iIndex); */
     Group.SetParameter(S_Factor_For_Polynomial_Transition, MMC_S_FACTOR, 0);
}


void endAdminSingleAxis(void)
// =============================
{
     printf("\n Function: %s: ", __func__);

     MMC_CloseConnection(ComHndl) ;
}


     void endAdminMultiAxis(void)
// ================================
{
/*  Source class is: CMMCGroupAxis Group;    */

     printf("\n Function: %s: ", __func__);

//The two function below can be called as shown below, or called from the EAS (configuration file) but, if defined
in configuration (EAS show axis in V01 group)
//  they should not be called here! (the axis is already in group...)
//   Group.RemoveAxisFromGroup(NC_NODE_1_ID);
//   Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*===================== Administration functions END ===========================*/


/*======================== Scenario functions STR =============================*/

void SnroMoveComplex2d(int trace)
// ==============================
{

     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     initAdminMultiAxis2d();

     AxisX.PowerOn(MC_BUFFERED_MODE);
    AxisY.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisX);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisY);

     Group.GroupEnable();

     MoveLinearRelative();
     MoveCircularLinarAbs();
     MoveCircularAbsolute2d();
     GroupOverrideAndPosition();
```

```
	Group.GroupDisable();

	AxisY.PowerOff(MC_BUFFERED_MODE);
	AxisX.PowerOff(MC_BUFFERED_MODE);
	WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisY);
	WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisX);

	endAdminMultiAxis();
	endAdminSingleAxis();

	printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

/*==================== Example functions STR ==================================*/

//	15.4.12.  MoveLinearAbsolute
//	15.4.11.  MoveCircularAbsoluteAngle
/* While looking X Y => Drawing 5 points star encircle by circle */
void MoveCircularLinarAbs(void)
// ==============================
{
	printf("\n Function: %s: ", __func__);

	Group.m_fVelocity	  =    50000.0;
	Group.m_fAcceleration =   500000.0;
	Group.m_fDeceleration =   500000.0;
	Group.m_fJerk		  =  1000000.0;

// Move #1
	Group.m_dEndPoint[0] = StarPoi[0][0];
	Group.m_dEndPoint[1] = StarPoi[0][1];
	Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #2
	Group.m_ucExecute = 0;		// Overwrite the default
//	Group.m_ucExecute = 1;		// or not...
	Group.m_dEndPoint[0] = StarPoi[1][0];
	Group.m_dEndPoint[1] = StarPoi[1][1];
	Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #3
	Group.m_ucExecute = 1;		// Set (returned) the default to 1
	Group.m_dEndPoint[0] = StarPoi[2][0];
	Group.m_dEndPoint[1] = StarPoi[2][1];
	Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #4
	Group.m_dEndPoint[0] = StarPoi[3][0];
	Group.m_dEndPoint[1] = StarPoi[3][1];
	Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #5
	Group.m_dEndPoint[0] = StarPoi[4][0];
	Group.m_dEndPoint[1] = StarPoi[4][1];
	Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #6 Back to the started point - end drowing star.
	Group.m_dEndPoint[0] = StarPoi[0][0];
	Group.m_dEndPoint[1] = StarPoi[0][1];
```

```
    Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #7 drawing circle
    Group.m_dAuxPoint[0] = StarPoi[5][0];  // The Center Cordinate
    Group.m_dAuxPoint[1] = StarPoi[5][1];
    double dAngle = 360;
// Only 2D
    Group.MoveCircularAbsoluteAngle(dAngle, MC_BLENDING_LOW_MODE);
    WaitGrpDone(NC_GROUP_STANDBY_MASK);
}

//   15.4.7.   MoveCircularAbsolute
//   15.4.8.   MoveCircularAbsoluteCenter
//   15.4.9.   MoveCircularAbsoluteBorder
void MoveCircularAbsolute2d(void)
//   ================================
{
    int                         rt_val;
    NC_ARC_SHORT_LONG_ENUM   eArcShortLong;
    NC_PATH_CHOICE_ENUM      ePathChoice;
    NC_CIRC_MODE_ENUM        eCircleMode;
    double                      dAuxPoint[NC_MAX_NUM_AXES_IN_NODE];
    double                      dCenterPoint[NC_MAX_NUM_AXES_IN_NODE];
    double                      dBorderPoint[NC_MAX_NUM_AXES_IN_NODE];
    MC_BUFFERED_MODE_ENUM    eBufferMode;

    printf("\n Function: %s: ", __func__);

    ParamsGroup.eTransitionMode = MC_TM_NONE_MODE;
    Group.SetDefaultParams(ParamsGroup);

    ParamsGroup.fVelocity = 100000.0;
    Group.m_dEndPoint[0] = 0.0;      // Start point of MoveCircularAbs is End point of MoveLinearAbs...
    Group.m_dEndPoint[1] = 0.0;
    Group.MoveLinearAbsolute(MC_ABORTING_MODE);

    eArcShortLong = MC_NONE_ARC_CHOICE;     // MC_LONG;
    ePathChoice = MC_NONE_PATH_CHOICE; // MC_CLOCKWISE  MC_COUNTERCLOCKWISE   MC_CLOCKWISE;
    eCircleMode = MC_BORDER_CIRC_MODE;
    eBufferMode = MC_BUFFERED_MODE;        // MC_ABORTING_MODE = the defalut

    Group.m_dAuxPoint[0] = 10000.0;        // Point on border
    Group.m_dAuxPoint[1] = 10000.0;
    Group.m_dEndPoint[0] = 10000.0;        // End point
    Group.m_dEndPoint[1] =-10000.0;
    rt_val = Group.MoveCircularAbsolute(eArcShortLong, ePathChoice, eCircleMode, eBufferMode);

    dAuxPoint[0] =  15000.0;           // Circular Center point
    dAuxPoint[1] = -10000.0;
    Group.m_dEndPoint[0] = 15000.0;        // Circular End point, Circular start point = (end of last motion)
10000,-10000
    Group.m_dEndPoint[1] = -5000.0;
    eCircleMode = MC_CENTER_CIRC_MODE;
    eArcShortLong = MC_LONG;
    rt_val = Group.MoveCircularAbsolute(eArcShortLong, ePathChoice, eCircleMode, dAuxPoint, eBufferMode);

    Group.m_dAuxPoint[0] = 20000.0;        // Circular center
    Group.m_dAuxPoint[1] = -5000.0;
    Group.m_dEndPoint[0] = 20000.0;        // Circular end point
    Group.m_dEndPoint[1] =     0.0;
    rt_val = Group.MoveCircularAbsoluteCenter(eArcShortLong, eBufferMode);
```

```
    Group.m_dEndPoint[0] = 10000.0;   // Start point of MoveCircularAbs... (End point of MoveLinearAbs)
    Group.m_dEndPoint[1] =  6000.0;
    Group.MoveLinearAbsolute(MC_BUFFERED_MODE);
    eArcShortLong = MC_LONG;
    dCenterPoint[0] = 10000.0;        // Circular center
    dCenterPoint[1] =  8000.0;
    Group.m_dEndPoint[0] = 12000.0;        // Circular end point
    Group.m_dEndPoint[1] =  8000.0;
    eBufferMode = MC_BUFFERED_MODE;
    rt_val = Group.MoveCircularAbsoluteCenter(eArcShortLong, dCenterPoint, eBufferMode);

// Start point, end of previous move (12000, 8000)
    dBorderPoint[0] = 15000.0;        // Circular border (end of previous).
    dBorderPoint[1] = 11000.0;
    Group.m_dEndPoint[0] = 15000.0;        // Circular end point
    Group.m_dEndPoint[1] =  5000.0;
    rt_val = Group.MoveCircularAbsoluteBorder(dBorderPoint, eBufferMode);

    Group.m_dEndPoint[0] = 10000.0;   // Start point of MoveCircularAbs... (End point of MoveLinearAbs)
    Group.m_dEndPoint[1] = 0.0;
    Group.MoveLinearAbsolute(MC_BUFFERED_MODE);

    WaitGrpDone(NC_GROUP_STANDBY_MASK);
}


//   15.4.13.MoveLinearRelative
void MoveLinearRelative(void)
// ============================
{
    double         dbDistance[NC_MAX_NUM_AXES_IN_NODE];
    float          fAcceleration,
              fDeceleration,
              fJerk;

    printf("\n Function: %s: ", __func__);

    Group.MoveLinearAbsolute(MC_ABORTING_MODE);
    WaitGrpDone(NC_GROUP_STANDBY_MASK);

    Group.m_dEndPoint[0] = 1000.0;
    Group.m_dEndPoint[1] =  500.0;
    Group.MoveLinearRelative(MC_BUFFERED_MODE);

    WAIT_SLEEP_MILLI(10);
/*...broke previous straight line when looking x relative to y*/
    Group.m_dEndPoint[0] =  250.0;
    Group.MoveLinearRelative(MC_BUFFERED_MODE);

    WAIT_SLEEP_MILLI(10);

/*...broke previous straight line when looking x relative to y*/
    Group.m_dEndPoint[1] =  250.0;
    Group.m_eTransitionMode = MC_TM_DEFINED_VELOCITY_MODE;
    Group.MoveLinearRelative(MC_BLENDING_PREVIOUS_MODE);

    WAIT_SLEEP_MILLI(10);

    Group.m_dEndPoint[0] =  800.0;
    Group.MoveLinearRelative(MC_BLENDING_PREVIOUS_MODE);
```

```
    WAIT_SLEEP_MILLI(10);

    Group.m_dEndPoint[1] =  1000.0;
    Group.MoveLinearRelative(50000.0, MC_BLENDING_PREVIOUS_MODE);

    WAIT_SLEEP_MILLI(10);

    Group.m_dEndPoint[0] =  1000.0;
    Group.MoveLinearRelative(75000.0, MC_BLENDING_PREVIOUS_MODE);

    dbDistance[0] = 1000.0;
    dbDistance[1] = 1500.0;
    Group.MoveLinearRelative(100000.0, dbDistance, MC_BLENDING_PREVIOUS_MODE);
    WaitGrpDone(NC_GROUP_STANDBY_MASK);

    dbDistance[1] = 100.0;
    Group.MoveLinearRelative( 50000.0, dbDistance, MC_BLENDING_PREVIOUS_MODE);
    WaitGrpDone(NC_GROUP_STANDBY_MASK);

    fAcceleration = 20000000.0;
    fDeceleration = fAcceleration;
    dbDistance[0] = 100.0;
    Group.MoveLinearRelative(125000.0, dbDistance, fAcceleration, fDeceleration, MC_BLENDING_PREVIOUS_MODE);
    Group.MoveLinearRelative(250000.0, dbDistance, fAcceleration, fDeceleration, MC_BLENDING_PREVIOUS_MODE);
    WaitGrpDone(NC_GROUP_STANDBY_MASK);

    fJerk = 200000000.0;
    dbDistance[0] = 1000.0;
    Group.MoveLinearRelative(150000.0, dbDistance, fAcceleration, fDeceleration, fJerk, MC_BLENDING_PREVIOUS_MODE);
    Group.MoveLinearRelative(200000.0, dbDistance, fAcceleration, fDeceleration, fJerk, MC_BLENDING_PREVIOUS_MODE);
    WaitGrpDone(NC_GROUP_STANDBY_MASK);
}

//   15.4.14.GroupSetOverride
//   15.4.15.GroupSetPosition
 void     GroupOverrideAndPosition(void)
// ===================================
{
    double        dbPosition[3];
    float         fVelFactor,
                  fAccFactor,
                  fJerkFactor;

        float     fVelocity,
                  fAcceleration,
                  fDeceleration,
                  fJerk;

    unsigned short usUpdateVelFactorIdx;
    int           ind,
                  rt_val;

    printf("\n Function: %s: ", __func__);

//   GroupSetPosition is Not released yet... - Not supported
//
//   MC_COORD_SYSTEM_ENUM     eCoordSystem;
//   unsigned char       ucMode;
//
//   Group.m_dEndPoint[0] = 1000000.0;
```

```
//   Group.m_dEndPoint[1] = 2000000.0;
//   Group.m_dEndPoint[2] = 0.0;
//   ParamsGroup.eTransitionMode = MC_TM_NONE_MODE;
//   Group.SetDefaultParams(ParamsGroup);
//   Group.MoveLinearAbsolute(MC_ABORTING_MODE);

//   dbPosition[0] =     500000.0;
//   dbPosition[1] =     750000.0;
//   dbPosition[2] =     0.0;
//   eCoordSystem = MC_MCS_COORD;
/* RELATIVE =True, ABSOLUTE = False (Default) */
//   ucMode = ABSOLUTE;
/* MC_ABORTING_MODE is the defalut mode */
//   Group.GroupSetPosition(dbPosition, eCoordSystem, ucMode);

    fVelocity = 100000.0;
     fAcceleration = fDeceleration = 10000000;
     fJerk = 100000000.0;
     usUpdateVelFactorIdx = 0; /* Meanwhile only 0 is support */
     dbPosition[2] = 0.0;

     for (ind=0; ind<6; ind++)
     {
          if (ind==2)
          {
          WaitGrpDone(NC_GROUP_STANDBY_MASK);
          fVelFactor = 0.5;
          fAccFactor = 0.5;
          fJerkFactor= 0.5;
          rt_val = Group.GroupSetOverride(fVelFactor, fAccFactor, fJerkFactor, usUpdateVelFactorIdx);
          }

          if (ind==4)
          {
          WaitGrpDone(NC_GROUP_STANDBY_MASK);
          fVelFactor = 1.0;
          fAccFactor = 1.0;
          fJerkFactor= 1.0;
          rt_val = Group.GroupSetOverride(fVelFactor, fAccFactor, fJerkFactor, usUpdateVelFactorIdx);
          }

          dbPosition[0] = 0.0;
          dbPosition[1] = 0.0;
          Group.MoveLinearAbsolute(fVelocity, dbPosition, fAcceleration, fDeceleration, fJerk, MC_BUFFERED_MODE);
          dbPosition[0] = 100000.0;
          dbPosition[1] = 100000.0;
          Group.MoveLinearAbsolute(fVelocity, dbPosition, fAcceleration, fDeceleration, fJerk, MC_BUFFERED_MODE);
     }

     WaitGrpDone(NC_GROUP_STANDBY_MASK);
}

int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{
    printf("\n *********** STR Func: %s *********** ", __func__);

/* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
```

```
        printf("\n ASYNC event Reply ");
        break ;
    case EMCY_EVT:
        printf("\n Emergency Event received ");
        break ;
    case MOTIONENDED_EVT:
        printf("\n Motion Ended Event received ");
        break ;
    case HBEAT_EVT:
        printf("\n H Beat Fail Event received ");
        break ;
    case PDORCV_EVT:
        printf("\n PDO Received Event received - Updating Inputs ");
        break ;
    case DRVERROR_EVT:
        printf("\n Drive Error Received Event received ");
        break ;
    case HOME_ENDED_EVT:
        printf("\n Home Ended Event received ");
        break ;
    case SYSTEMERROR_EVT:
        printf("\n System Error Event received ");
    case TABLE_UNDERFLOW_EVT:
        printf("\n Underflow event received ");
        break ;
    case MODBUS_WRITE_EVT:
        printf("\n ModBus Write event received ");
        break ;
    case TOUCH_PROBE_ENDED_EVT:
        printf("\n Touch Probe event received ");
        break ;
    default:
        printf("\n Default.... Whatever arrived event received ");
        break;
    }

   printf("\n *********** END Func: %s *********** ", __func__);
    fflush(stdout); fflush(stderr);

    return 1 ;
}

int  OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short sErrorID, unsigned
short usStatus)
// ===================================================================================
{
    printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d, bye\n",
    msg, usAxisRef, sErrorID, usStatus);
    fflush(stdout); fflush(stderr);
    MMC_CloseConnection(uiConnHndl);
    exit(0);
}
/*======================= Example functions END ===============================*/

/*======================= Output STR ==========================================*/
#ifdef PROGRAM_OUTPUT
Output example EAS movment record:

#endif /* PROGRAM_OUTPUT */
/*========================= Output END ========================================*/
```

17.6.4 CMMCGroupAxis Class Functions Code Example 4

The following is an example of group axes motion in **3-D** using the MoveCircularXXXXXXX set of functions.

```
/*
============================================================================
Collection of Gmas API functions (Set #5)
Examples for document: "G-MAS Administrative and Motion API.pdf".
12Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define  EndMotionEventCB_MESSAGE  "!!!!END MOTION EVENT MESSAGE!!!"


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

    CMMCConnection             gConn;
    MMC_CONNECT_HNDL       ComHndl;

    CMMCSingleAxis             AxisX,    AxisY,  AxisZ;
    unsigned short             AxisXRef, AxisYRef, AxisZRef;

    CMMCGroupAxis          Group;
    double                     S_Factor_For_Polynomial_Transition;
    MMC_MOTIONPARAMS_GROUP     ParamsGroup;

    MMC_SETKINTRANSFORM_IN     SetKin;

    double   StarPoi[][2] =
                           {
                               { 7500.,     0.},
                               {12500., 13100.},
                               {  200.,  5600.},
                               {14700.,  5600.},
                               { 3340., 13750.},
                               { 7500.,  7500.}      // the Center cordination
                           };

    char * delimit =
"===========================================================================";
    char* strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
    char* strEndSnro1 = "\n End ";
    char* strEndSnro2 = " >>>>>>>>>>>>> ";

    int      WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

    void initAdminMultiAxis();
    void endAdminMultiAxis(void);

    void SnroMoveComplex3d(int);
    void     MoveCircularAbsolute3d(void);
```

```
    int      CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
    int      OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus);

/*========================== Administration functions STR ====================================*/

int      main(int)
// =============
{
    int trace = 1;

    printf("\n %s", delimit);
    printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

    try
    {
        SnroMoveComplex3d(trace++);

    }
    catch (CMMCException excp)
    {
        printf("\n %s", delimit);
        printf("\n %s", delimit);
        printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
        printf("\n %s", delimit);
        printf("\n %s", delimit);
        exit(0);
    }

    printf("\n End of %s ", __FILE__);
    printf("\n %s\n\n", delimit);
    return 0;
}


int      WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//=================================================================
{
             int end_of = 0;
             int iCount = 0;
    unsigned int ulState;

    while( ! end_of)
    {
        iCount ++;
        end_of = 1;
/* Read Axis Status command server for specific Axis */
        ulState = sng_axis->ReadStatus();
        if (!(ulState & break_state))
        {
            end_of = 0;

            WAIT_SLEEP_MILLI(20)
        }
    }

//       MMC_SHOWNODESTAT_IN  showin;
//       MMC_SHOWNODESTAT_OUT showout;
//       MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

    return 0;
}
```

```
void WaitGrpDone(unsigned int groupStatusMsk)
// =============================================
{
    unsigned int uiStatusRegister;

    uiStatusRegister = Group.GroupReadStatus();
    while((uiStatusRegister & groupStatusMsk) != groupStatusMsk)
    {
        WAIT_SLEEP_MILLI(2)
        uiStatusRegister = Group.GroupReadStatus();
    }
}


void initAdminMultiAxis()
// =========================
{
/*  Source class:                               */
/*      MMC_CONNECT_HNDL      ComHndl;          */
/*      CMMCSingleAxis        AxisX,    AxisY;  */
/*      CMMCGroupAxis         Group;            */
    int      iEventMask;


    printf("\n Function: %s: ", __func__);


/* CallbackFunc in ConnectIPCEx call if there   */
/* is no calling to 'RegisterEventCallback'     */
    iEventMask = 0x7fffffff;
    ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Put Null param Val for no CallbackFunc       */
/*  ComHndl = gConn.ConnectIPCEx(iEventMask, NULL); */
/* Should Not calling, called inside 'ConnectIPCEx' */
/*  rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */


/* Register Run Time Error Callback function    */
    CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

    AxisX.InitAxisData("a01", ComHndl);
    AxisY.InitAxisData("a02", ComHndl);
    AxisZ.InitAxisData("a03", ComHndl);

    Group.InitAxisData("v01", ComHndl);

    AxisXRef = AxisX.GetRef();
    AxisYRef = AxisY.GetRef();
    AxisZRef = AxisZ.GetRef();

/* Set by default the EndPoint=StartPoint */
    ParamsGroup.dEndPoint[0] = 0.0;
    ParamsGroup.dEndPoint[1] = 0.0;
    ParamsGroup.dEndPoint[2] = 0.0;


    ParamsGroup.fVelocity =   100000.0;
    ParamsGroup.fAcceleration =   8000000.0;
    ParamsGroup.fDeceleration =   8000000.0;
    ParamsGroup.fJerk =    100000000.0;
    ParamsGroup.eCoordSystem = MC_MCS_COORD;
    ParamsGroup.eBufferMode = MC_BUFFERED_MODE;
//  ParamsGroup.eTransitionMode = MC_TM_CORNER_DEVIATION_MODE_PLN6;
    ParamsGroup.eTransitionMode = MC_TM_NONE_MODE;
```

```
    ParamsGroup.fTransitionParameter[0] = 2000.0;
    ParamsGroup.ucExecute = 1;

    Group.SetDefaultParams(ParamsGroup);

/* Parameters for Kinematic Transformation */
    SetKin.eBufferMode = MC_BUFFERED_MODE;
    SetKin.eType[0] = NC_X_AXIS_TYPE;
    SetKin.eType[1] = NC_Y_AXIS_TYPE;
    SetKin.eType[2] = NC_Z_AXIS_TYPE;

    SetKin.hNode[0] = AxisX.GetRef();
    SetKin.hNode[1] = AxisY.GetRef();
    SetKin.hNode[2] = AxisZ.GetRef();

    SetKin.iMcsToAcsFuncID[0] = NC_TR_SHIFT_FUNC;
    SetKin.iMcsToAcsFuncID[1] = NC_TR_SHIFT_FUNC;
    SetKin.iMcsToAcsFuncID[2] = NC_TR_SHIFT_FUNC;

    SetKin.iNumAxes = 3;
    SetKin.ucExecute = 1;

    SetKin.ulTrCoef[0][0] = 1;
    SetKin.ulTrCoef[0][1] = 1;
    SetKin.ulTrCoef[0][2] = 1;

    SetKin.ulTrCoef[1][0] = 1;
    SetKin.ulTrCoef[1][1] = 1;
    SetKin.ulTrCoef[1][2] = 1;

    SetKin.ulTrCoef[2][0] = 1;
    SetKin.ulTrCoef[2][1] = 1;
    SetKin.ulTrCoef[2][2] = 1;

    Group.SetKinTransform(SetKin);

/* Set the factor for Polynominal Transition    */
    S_Factor_For_Polynomial_Transition = 0.4;   // The default is 1.4

/* SetParameter(double dbValue, MMC_PARAMETER_LIST_ENUM eNumber, int iIndex); */
    Group.SetParameter(S_Factor_For_Polynomial_Transition, MMC_S_FACTOR, 0);
}



    void endAdminMultiAxis(void)
// ================================
{
//  Source class:
//  CMMCGroupAxisGroup;


    printf("\n Function: %s: ", __func__);

    MMC_CloseConnection(ComHndl) ;

//  The two functions below can be called as shown, or called from the EAS application
(configuration file), but, if define in configuration (EAS show axis in V01 group)
//  they shuld not be call here! (the axis is already in group...)
//  Group.RemoveAxisFromGroup(NC_NODE_1_ID);
//  Group.RemoveAxisFromGroup(NC_NODE_2_ID);
//  Group.RemoveAxisFromGroup(NC_NODE_3_ID);
}
/*======================== Administration functions END ====================================*/
```

```
/*========================== Scenario functions STR ==========================================*/

void SnroMoveComplex3d(int trace)
// ==============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminMultiAxis();


    AxisX.PowerOn(MC_BUFFERED_MODE);
    AxisY.PowerOn(MC_BUFFERED_MODE);
    AxisZ.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisX);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisY);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisZ);

    Group.GroupEnable();

    MoveCircularAbsolute3d();

    Group.GroupDisable();

    AxisY.PowerOff(MC_BUFFERED_MODE);
    AxisX.PowerOff(MC_BUFFERED_MODE);
    AxisZ.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisY);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisX);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisZ);

    endAdminMultiAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

/*========================= Example functions STR ============================================*/

//   15.4.7.  MoveCircularAbsolute
//   15.4.8.  MoveCircularAbsoluteCenter
//   15.4.9.  MoveCircularAbsoluteBorder
//   15.4.10.MoveCircularAbsoluteRadius
void MoveCircularAbsolute3d(void)
//   ================================
{
    int                         rt_val;
    NC_ARC_SHORT_LONG_ENUM      eArcShortLong;
    NC_PATH_CHOICE_ENUM     ePathChoice;
    NC_CIRC_MODE_ENUM           eCircleMode;
    double                      dAuxPoint[NC_MAX_NUM_AXES_IN_NODE];
    double                      dCenterPoint[NC_MAX_NUM_AXES_IN_NODE];
    double                      dBorderPoint[NC_MAX_NUM_AXES_IN_NODE];
    MC_BUFFERED_MODE_ENUM       eBufferMode;


    printf("\n Function: %s: ", __func__);


    ParamsGroup.fVelocity = 100000.0;

    Group.m_dEndPoint[0] = 0.0;
    Group.m_dEndPoint[1] = 0.0;
    Group.m_dEndPoint[2] = 0.0;

    Group.MoveLinearAbsolute(MC_ABORTING_MODE);
```

```
    eArcShortLong = MC_NONE_ARC_CHOICE;     // MC_LONG;
    ePathChoice   = MC_NONE_PATH_CHOICE;    // MC_CLOCKWISE  MC_COUNTERCLOCKWISE   MC_CLOCKWISE;
    eCircleMode   = MC_BORDER_CIRC_MODE;
    eBufferMode   = MC_BUFFERED_MODE;       // MC_ABORTING_MODE = the defalut

    Group.m_dAuxPoint[0] = 20000.0;         // Point on border
    Group.m_dAuxPoint[1] =     0.0;
    Group.m_dAuxPoint[2] =     0.0;

    Group.m_dEndPoint[0] = 10000.0;         // End point
    Group.m_dEndPoint[1] =     0.0;
    Group.m_dEndPoint[2] =-10000.0;

    rt_val = Group.MoveCircularAbsolute(eArcShortLong, ePathChoice, eCircleMode, eBufferMode);

    dAuxPoint[0] = 10000.0;                 // Circular Center point
    dAuxPoint[1] =     0.0;
    dAuxPoint[2] =     0.0;

    Group.m_dEndPoint[0] =  10000.0; // Circular End point, Circular start point = (end of last
motion)
    Group.m_dEndPoint[1] = -10000.0;
    Group.m_dEndPoint[2] = 0.0;

    eCircleMode = MC_CENTER_CIRC_MODE;
    eArcShortLong = MC_LONG;
    rt_val = Group.MoveCircularAbsolute(eArcShortLong, ePathChoice, eCircleMode, dAuxPoint,
eBufferMode);

    Group.m_dAuxPoint[0] =  10000.0;        // Circular center
    Group.m_dAuxPoint[1] = -10000.0;
    Group.m_dAuxPoint[2] =   5000.0;

    Group.m_dEndPoint[0] =   5000.0;        // Circular end point
    Group.m_dEndPoint[1] = -10000.0;
    Group.m_dEndPoint[2] =   5000.0;

    // 2D & 3D
    rt_val = Group.MoveCircularAbsoluteCenter(eArcShortLong, eBufferMode);

    Group.m_dEndPoint[0] = -20000.0;        // Start point of MoveCircularAbsoluteCenter... (End
point of MoveLinearAbs)
    Group.m_dEndPoint[1] = 0.0;
    Group.m_dEndPoint[2] = -20000.0;
    rt_val = Group.MoveLinearAbsolute(MC_BUFFERED_MODE);

    eArcShortLong = MC_LONG;

    dCenterPoint[0] = 0.0;                  // Circular center
    dCenterPoint[1] = 0.0;
    dCenterPoint[2] = 0.0;

    Group.m_dEndPoint[0] = -20000.0;        // Circular end point
    Group.m_dEndPoint[1] = -20000.0;
    Group.m_dEndPoint[2] = 0.0;

    eBufferMode = MC_BUFFERED_MODE;
    // 2D & 3D
    rt_val = Group.MoveCircularAbsoluteCenter(eArcShortLong, dCenterPoint, eBufferMode);

                                            // Start point, end of previous move
    dBorderPoint[0] = -20000.0;             // Circular border (end of previous).
    dBorderPoint[1] = -40000.0;
    dBorderPoint[2] = 0.0;
```

```
    Group.m_dEndPoint[0] = -20000.0;       // Circular end point
    Group.m_dEndPoint[1] = -10000.0;
    Group.m_dEndPoint[2] = -20000.0;
    // 2D & 3D
    rt_val = Group.MoveCircularAbsoluteBorder(dBorderPoint, eBufferMode);

    Group.m_dEndPoint[0] = 10000.0;        // Start point
    Group.m_dEndPoint[1] =     0.0;
    Group.m_dEndPoint[2] =     0.0;
    Group.MoveLinearAbsolute(MC_BUFFERED_MODE);

    Group.m_dEndPoint[0] =     0.0;        // End point
    Group.m_dEndPoint[1] = 10000.0;
    Group.m_dEndPoint[2] =     0.0;

    Group.m_dAuxPoint[0] =     0.0;        //  Radios location
    Group.m_dAuxPoint[1] =       0.0;
    Group.m_dAuxPoint[2] = 10000.0;

    ePathChoice = MC_CLOCKWISE;
// 3D (not 2D)
    rt_val = Group.MoveCircularAbsoluteRadius(eArcShortLong, ePathChoice, eBufferMode);

    WaitGrpDone(NC_GROUP_STANDBY_MASK);
}


int      CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{
    printf("\n *********** STR Func: %s *********** ", __func__);

/* Which function ID was received ... */
    switch(recvBuffer[1])
    {
    case ASYNC_REPLY_EVT:
        printf("\n ASYNC event Reply ");
        break ;
    case EMCY_EVT:
        printf("\n Emergency Event received ");
        break ;
    case MOTIONENDED_EVT:
        printf("\n Motion Ended Event received ");
        break ;
    case HBEAT_EVT:
        printf("\n H Beat Fail Event received ");
        break ;
    case PDORCV_EVT:
        printf("\n PDO Received Event received - Updating Inputs ");
        break ;
    case DRVERROR_EVT:
        printf("\n Drive Error Received Event received ");
        break ;
    case HOME_ENDED_EVT:
        printf("\n Home Ended Event received ");
        break ;
    case SYSTEMERROR_EVT:
        printf("\n System Error Event received ");
    case TABLE_UNDERFLOW_EVT:
        printf("\n Underflow event received ");
        break ;
    case MODBUS_WRITE_EVT:
        printf("\n ModBus Write event received ");
        break ;
```

```
    case TOUCH_PROBE_ENDED_EVT:
        printf("\n Touch Probe event received ");
        break ;
    default:
        printf("\n Default.... Whatever arrived event received ");
        break;
    }

    printf("\n *********** END Func: %s *********** ", __func__);
    fflush(stdout); fflush(stderr);

    return 1 ;
}


int     OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus)
//
=================================================================================================
{
    printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
            msg, usAxisRef, sErrorID, usStatus);
    fflush(stdout); fflush(stderr);

    MMC_CloseConnection(uiConnHndl);
    exit(0);
}

/*============================= Example functions END =======================================*/

/*================================= Output STR =============================================*/
#ifdef PROGRAM_OUTPUT
Output example EAS movment record:



#endif /* PROGRAM_OUTPUT */
/*================================= Output END =============================================*/
```

17.6.5 CMMCGroupAxis(CMMCGroupAxis& axis)

This function initiates the Group axis and includes the function InitAxisData that initiates the axis name and retrieves a session handler, and GetGroupAxisByName which accesses the group function name. Refer to the section **8.1.16 MMC_GetAxisByName** for details of the description, scope, and motion mode.

```
public:
CMMCGroupAxis();
virtual ~CMMCGroupAxis();
CMMCGroupAxis(CMMCGroupAxis& axis);
void InitAxisData(const char* cName, MMC_CONNECT_HNDL uHandle) throw (CMMCException);
int GetGroupAxisByName(const char* cName) throw (CMMCException)
double m_dAuxPoint[NC_MAX_NUM_AXES_IN_NODE];
double m_dEndPoint[NC_MAX_NUM_AXES_IN_NODE];
float m_fVelocity;
float m_fAcceleration;
float m_fDeceleration;
float m_fJerk;
float m_fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM m_eCoordSystem;
NC_TRANSITION_MODE_ENUM m_eTransitionMode;
NC_ARC_SHORT_LONG_ENUM m_eArcShortLong;
NC_PATH_CHOICE_ENUM m_ePathChoice;
NC_CIRC_MODE_ENUM m_eCircleMode;
unsigned char m_ucSuperimposed;
unsigned char m_ucExecute;
unsigned int m_uiExecDelayMs;
```

Source GMAS\includes\CPP\CMMCGroupAxis.h

.NET Definition

Function Parameters

void InitAxisData(const char cName, MMC_CONNECT_HNDL uHandle) throw (CMMCException)*

Refer to the next function **17.6.6 InitAxisData** for details.

int GetGroupAxisByName(const char cName) throw (CMMCException)*

Refer to the next function **17.6.7 GetGroupAxisByName** for details.

double m_dAuxPoint[NC_MAX_NUM_AXES_IN_NODE];
double m_dEndPoint[NC_MAX_NUM_AXES_IN_NODE];
float m_fVelocity;
float m_fAcceleration;
float m_fDeceleration;
float m_fJerk;
float m_fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM m_eCoordSystem;
NC_TRANSITION_MODE_ENUM m_eTransitionMode;
NC_ARC_SHORT_LONG_ENUM m_eArcShortLong;
NC_PATH_CHOICE_ENUM m_ePathChoice;
NC_CIRC_MODE_ENUM m_eCircleMode;
unsigned char m_ucSuperimposed;
unsigned char m_ucExecute;
unsigned int m_uiExecDelayMs;

Refer to the section **17.5.1 MMC_MOTIONPARAMS_GROUP()** for details of the parameters.

17.6.6 InitAxisData

This function initiates an axis name and retrieves a session handler. Refer to the section **8.1.16 MMC_GetAxisByName** for details of the description, scope, and motion mode.

```
ivirtual void InitAxisData(
const char* cName,
 MMC_CONNECT_HNDL uHandle
) throw (CMMCException)
```

Source GMAS\includes\CPP\CMMCGroupAxis.h

.NET Definition

Function Parameters

cName

Tag/assembly name as declared in XML configuration file.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

Refer to the function example in section **17.3.6**

17.6.7 GetGroupAxisByName

Refer to the section 8.1.16 MMC_GetAxisByName for details of the description, scope, and motion mode. This function accesses the group function name.

int GetGroupAxisByName(
const char* cName
) throw (CMMCException);

Source GMAS\includes\CPP\MMCAxis.h

.NET Definition

Function Parameters

cName

Tag/assembly name as declared in XML configuration file.

throw (CMMCException)

Refer to the section 17.1.1 CMMCException. Produces details of the error including:

Function Name	Structure name	Axis reference
Error ID	Status of the axis.	

17.6.7.1 Functions Code Example

```
// 16.2.5.  GetFbDepth   5.7.3. MMC_GetFbDepth Cmd
// unsigned int CMMCMotionAxis::GetFbDepth()
//    16.2.6.     GetAxisByName     10.3.17. MMC_GetAxisByName Cmd
//int CMMCAxis::GetAxisByName(const char* cName)
//    16.2.7.     GetGroupAxisByName  10.3.18. MMC_GetGroupByName Cmd
// int CMMCGroupAxis::GetGroupAxisByName(const char* cName)
void  DepthName(void)
// ====================
{
unsigned int iVal1, iVal2, iVal3;

printf("\n      %s:", __func__);
 iVal1 = AxisB.GetFbDepth();

 Group.GroupDisable();
 AxisB.PowerOff(MC_BUFFERED_MODE);

 iVal2 = AxisB.GetFbDepth();
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
 iVal3 = AxisB.GetFbDepth();

 printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
 Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");   /* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */
    // iVal3 = AxisA.GetAxisByName("A01"); /* It case sensitive - Not define - exception... */
 iVal1 = Group.GetGroupAxisByName("v01");  /* Expected 256 */
/*
*   iVal2 = Group.GetGroupAxisByName("v02");
*/
}
```

17.6.8 SetDefaultParams

Sets the multiple axes' default parameters, and overwrites the class default parameters.

```
void SetDefaultParams(
const MMC_MOTIONPARAMS_GROUP& stGroupAxisParams
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

stGroupAxisParams

stGroupAxisParams references the structure MMC_MOTIONPARAMS_GROUP with default parameters, and either returns none, or throws CMMCException on failure.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

MMC_MOTIONPARAMS_GROUP Structure

```
typedef struct{
double dAuxPoint[NC_MAX_NUM_AXES_IN_NODE];
double dEndPoint[NC_MAX_NUM_AXES_IN_NODE];
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE];
MC_COORD_SYSTEM_ENUM eCoordSystem;
NC_TRANSITION_MODE_ENUM eTransitionMode;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucSuperimposed;
unsigned int m_uiExecDelayMs;
unsigned char ucExecute;
}MMC_MOTIONPARAMS_GROUP;
```

Parameters

All parameters

Refer to the section **17.5.1 MMC_MOTIONPARAMS_GROUP()** for details of the parameters.

17.6.8.2 Function Code Example

```
// 16.4.1. void CMMCGroupAxis:: SetDefaultParams(const MMC_MOTIONPARAMS_GROUP&
stGroupAxisParams)  (- no corresponding MMC_ C func)
void  SetGrpKinDef(void)
// ======================
{
printf("\n      %s:", __func__);
        /* Parameters for Kinematic Transformation */
 SetKin.eBufferMode = MC_BUFFERED_MODE;
 SetKin.eType[0] = NC_X_AXIS_TYPE;
 SetKin.eType[1] = NC_Y_AXIS_TYPE;

 SetKin.hNode[0] = AxisA.GetRef();
 SetKin.hNode[1] = AxisB.GetRef();

 SetKin.iMcsToAcsFuncID[0] = NC_TR_SHIFT_FUNC;
 SetKin.iMcsToAcsFuncID[1] = NC_TR_SHIFT_FUNC;

 SetKin.iNumAxes = 2;
 SetKin.ucExecute = 1;

 SetKin.ulTrCoef[0][0] = 1;
 SetKin.ulTrCoef[0][1] = 1;
 SetKin.ulTrCoef[0][2] = 0;

 SetKin.ulTrCoef[1][0] = 1;
 SetKin.ulTrCoef[1][1] = 1;
 SetKin.ulTrCoef[1][2] = 0;
        /* Set by default the EndPoint=StartPoint */
 ParamsGroup.dEndPoint[0] = StarPoi[0][0];
 ParamsGroup.dEndPoint[1] = StarPoi[0][1];


 ParamsGroup.fVelocity    = 100000;
 ParamsGroup.fAcceleration     = 2000000;
 ParamsGroup.fDeceleration     = 2000000;
 ParamsGroup.fJerk     = 10000000;
 ParamsGroup.eCoordSystem = MC_MCS_COORD;
 ParamsGroup.eBufferMode  = MC_BUFFERED_MODE;
 ParamsGroup.eTransitionMode = MC_TM_CORNER_DEVIATION_MODE_PLN6;
 ParamsGroup.fTransitionParameter[0] = 2000;
 ParamsGroup.ucExecute    = 1;


 Group.SetKinTransform(SetKin);
 Group.SetDefaultParams(ParamsGroup);
}
```

17.6.9 SetCartesianKinematics

Refer to **5.15.16 MMC_SetKinTransformCartesian** for details of the description, scope, and motion mode.

This function will be deprecated in the future.

```
void SetCartesianKinematics(
MC_KIN_REF_CARTESIAN stCart
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MC_KIN_REF_CARTESIAN stCart

Refer to the parameter definition in the structure **5.15.3 MC_KIN_REF_CARTESIAN** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.10 SetDeltaRobotKinematics

Refer to **5.15.15 MMC_SetKinTransformDelta** for details of the description, scope, and motion mode.

This function will be deprecated in the future.

```
void SetDeltaRobotKinematics(
MC_KIN_REF_DELTA stDelta
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MC_KIN_REF_DELTA stDelta

Refer to the parameter definition in the structure **5.15.4 MC_KIN_REF_DELTA** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.11 SetKinematic

Refer to **5.15.11 MMC_SetKinTransformEx** for details of the description, scope, and motion mode.

This function will be deprecated in the future.

```
void SetKinematic(
MC_KIN_REF stInput,
NC_KIN_TYPE eKinType
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MC_KIN_REF stInput

Refer to the parameter definition in the function **MMC_SETKINTRANSFORMEX_IN Structure** for details.

NC_KIN_TYPE eKinType

Refer to the parameter definition in the function **MMC_SETKINTRANSFORMEX_IN Structure** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.12 SetKinTransform

Sets the multiple axes' default parameters, and overwrites the class default parameters.

```
void SetKinTransform(
MMC_SETKINTRANSFORM_IN& stInParam
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

stInParam

stInParam references the structure MMC_SETKINTRANSFORM_IN with default parameters, and either returns none, or throws CMMCException on failure.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name   Structure name
Axis reference   Error ID
Status of the axis.

MMC_SETKINTRANSFORM_IN Structure

Parameters

All parameters

Refer to the section **MMC_SETKINTRANSFORM_IN Structure** for details of the parameters.

For code example, refer to the section **17.6.1**.

17.6.13 SetCartesianTransform

Sets the MCS to PCS parameters for group's kinematic transformation.

int SetCartesianTransform(
[MMC_SETCARTESIANTRANSFORM_IN* stInParam]

[(double (&dbOffset)[3],
double (&dbRotAngle)[3],
PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits=PCS_DEGREE,
MC_BUFFERED_MODE_ENUM eBufferMode=MC_BUFFERED_MODE,
MC_EXECUTION_MODE eExecutionMode=eMMC_EXECUTION_MODE_IMMEDIATE]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MMC_SETCARTESIANTRANSFORM_IN stInParam*

stInParam references the structure MMC_SETCARTESIANTRANSFORM_IN with default parameters, and either returns none, or throws CMMCException on failure.

(&dbOffset)[3]

X,Y,Z translation components' offsets. Any +ve or –ve values

(&dbRotAngle)[3]

U,V,W rotation angles. Any +ve or –ve values

PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits=PCS_DEGREE

The rotational units used to defined the angle units used in PCS to MCS transformations

The enumerator PCS_ROTATION_ANGLE_UNITS_ENUM is defined by the following:

```
PCS_DEGREE   = 0
PCS_RADIAN   = 1
```

MC_BUFFERED_MODE_ENUM eBufferMode=MC_BUFFERED_MODE

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Buffered The next function block affects the axis as soon as the previous movement is completed.

MC_EXECUTION_MODE eExecutionMode=eMMC_EXECUTION_MODE_IMMEDIATE

Execution mode enumerator defining whether the execution is immediate or queued, with the following values:

eMMC_EXECUTION_MODE_IMMEDIATE = 0,

eMMC_EXECUTION_MODE_QUEUED

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

MMC_SETCARTESIANTRANSFORM_IN Structure

Parameters

All parameters

Refer to the section **MMC_SETCARTESIANTRANSFORM_IN Structure** for details of the parameters.

17.6.14 ReadCartesianTransform

Read parameters which was previouslly set by SetCartesianTransform.

```
int ReadCartesianTransform(
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.15 TrackConveyorBelt

This function block provides an abstraction layer for a conveyor, allowing the user to track objects moving on a straight line in space on a conveyor belt. In short, a dynamic MCS to PCS transition depends on the conveyor axis position.

```
int TrackConveyorBelt(
MMC_TRACKCONVEYORBELT_IN* pInParam

[unsigned short usRefAxis,
double (&dbConveyorBeltOrigin)[6],
double (&dbInitialObjectPosition)[6],
double dInitialRefAxisPosition,
double dRefAxisScaling = 1.0,
PCS_REF_AXIS_SRC_ENUM eSourceType = NC_PCS_TARGET_POS,
PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits = PCS_RADIAN,
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE,
MC_COORD_SYSTEM_ENUM eCoordSystem = MC_PCS_COORD,
MC_EXECUTION_MODE eExecutionMode = eMMC_EXECUTION_MODE_QUEUED]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MMC_TRACKCONVEYORBELT_IN pInParam*

pInParam references the structure MMC_TRACKCONVEYORBELT_IN with default parameters, and either returns none, or throws CMMCException on failure.

usRefAxis

The selected axis reference. Any +ve value.

(&dbConveyorBeltOrigin)[6]

X,Y,Z,U,V,W conveyor belt origin. Any +ve –ve values

(&dbInitialObjectPosition)[6]

X,Y,Z translation components' offsets. Any +ve or –ve values

dInitialRefAxisPosition

Initial reference axis position of the conveyor belt. Any +ve or –ve values

dRefAxisScaling = 1.0

The scaling of the referenece axis relative to the position of the object on the conveyor belt and the length of the conveyor belt. Any +ve or –ve values

PCS_REF_AXIS_SRC_ENUM eSourceType = NC_PCS_TARGET_POS

This is a reference type for dynamic PCS/MCS transformation. The enumerator PCS_REF_AXIS_SRC_ENUM has the following values:

NC_PCS_TARGET_POS = 0

NC_PCS_ACTUAL_POS = 1

NC_PCS_AUX_POS = 2

PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits = PCS_RADIAN

The rotational units used to defined the angle units used in PCS to MCS transformations

The enumerator PCS_ROTATION_ANGLE_UNITS_ENUM is defined by the following:

PCS_DEGREE = 0

PCS_RADIAN = 1

MC_BUFFERED_MODE_ENUM eBufferMode=MC_BUFFERED_MODE

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1
MC_BUFFERED_MODE = 2
MC_BLENDING_LOW_MODE = 3
MC_BLENDING_PREVIOUS_MODE = 4
MC_BLENDING_NEXT_MODE = 5
MC_BLENDING_HIGH_MODE = 6

Buffered The next function block affects the axis as soon as the previous movement is completed.

MC_COORD_SYSTEM_ENUM eCoordSystem = MC_PCS_COORD

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

MC_EXECUTION_MODE eExecutionMode = eMMC_EXECUTION_MODE_QUEUED

Execution mode enumerator defining whether the execution is immediate or queued, with the following values:

eMMC_EXECUTION_MODE_IMMEDIATE = 0,

eMMC_EXECUTION_MODE_QUEUED

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name

Axis reference Error ID

Status of the axis.

MMC_SETCARTESIANTRANSFORM_IN Structure

Parameters

All parameters

Refer to the section **MMC_TRACKCONVEYORBELT_IN Structure** for details of the parameters.

17.6.16 TrackRotaryTable

This function block offers an abstraction layer for a rotary table, allowing the user to track objects moving on a cyclic space. In short, a dynamic MCS to PCS transition depends on rotary table axis position.

```
int TrackRotaryTable(
MMC_TRACKROTARYTABLE_IN* pInParam

double (&dbRotaryTableOrigin)[6],
double (&dbInitialObjectPosition)[6],
double dInitialRefAxisPosition,
double dRefAxisScaling = 1.0,
PCS_REF_AXIS_SRC_ENUM eSourceType = NC_PCS_TARGET_POS,
PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits = PCS_RADIAN,
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE,
MC_COORD_SYSTEM_ENUM eCoordSystem = MC_PCS_COORD,
MC_EXECUTION_MODE eExecutionMode = eMMC_EXECUTION_MODE_QUEUED
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MMC_TRACKROTARYTABLE_IN pInParam*

pInParam references the structure MMC_TRACKROTARYTABLE_IN with default parameters, and either returns none, or throws CMMCException on failure.

(&dbRotaryTableOrigin)[6]

X,Y,Z,U,V,W rotary table origin. Any +ve –ve values

(&dbInitialObjectPosition)[6]

X,Y,Z translation components' offsets. Any +ve or –ve values

dInitialRefAxisPosition

Initial reference axis position of the rotary table. Any +ve or –ve values

dRefAxisScaling = 1.0

The scaling of the referenece axis relative to the position of the object on the rotary table and the diameter of the table. Any +ve or –ve values

PCS_REF_AXIS_SRC_ENUM eSourceType = NC_PCS_TARGET_POS

This is a reference type for dynamic PCS/MCS transformation. The enumerator PCS_REF_AXIS_SRC_ENUM has the following values:

NC_PCS_TARGET_POS = 0

NC_PCS_ACTUAL_POS = 1

NC_PCS_AUX_POS = 2

PCS_ROTATION_ANGLE_UNITS_ENUM eRotAngleUnits = PCS_RADIAN

The rotational units used to defined the angle units used in PCS to MCS transformations

The enumerator PCS_ROTATION_ANGLE_UNITS_ENUM is defined by the following:

PCS_DEGREE = 0

PCS_RADIAN = 1

MC_BUFFERED_MODE_ENUM eBufferMode=MC_BUFFERED_MODE

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows:

MC_ABORTING_MODE = 1
MC_BUFFERED_MODE = 2
MC_BLENDING_LOW_MODE = 3
MC_BLENDING_PREVIOUS_MODE = 4
MC_BLENDING_NEXT_MODE = 5
MC_BLENDING_HIGH_MODE = 6

Buffered The next function block affects the axis as soon as the previous movement is completed.

MC_COORD_SYSTEM_ENUM eCoordSystem = MC_PCS_COORD

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

MC_EXECUTION_MODE eExecutionMode = eMMC_EXECUTION_MODE_QUEUED

Execution mode enumerator defining whether the execution is immediate or queued, with the following values:

eMMC_EXECUTION_MODE_IMMEDIATE = 0,

eMMC_EXECUTION_MODE_QUEUED

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

MMC_TRACKROTARYTABLE_IN Structure

Parameters

All parameters

Refer to the section **MMC_TRACKROTARYTABLE_IN Structure** for details of the parameters.

17.6.17 SetKinTransformDelta

Sets the kinematic transformation parameters (MSC to ACS) for the Delta robot. Refer to **5.15.15 MMC_SetKinTransformDelta** for details of the description, scope, and motion mode.

```
int SetKinTransformDelta(
IN MMC_KINTRANSFORM_DELTA_IN& pInParam
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Input Function Parameters

MMC_KINTRANSFORM_DELTA_IN& pInParam

pInParam references the structure MMC_KINTRANSFORM_DELTA_IN with default parameters, and either returns none, or throws CMMCException on failure.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

MMC_KINTRANSFORM_DELTA_IN Structure

Parameters

All parameters

Refer to the section **MMC_KINTRANSFORM_DELTA_IN Structure** for details of the parameters.

17.6.18 SetKinTransformCartesian

Sets the parameters kinematic transformation (MSC to ACS) for Cartesian system. Refer to **5.15.16 MMC_SetKinTransformCartesian** for details of the description, scope, and motion mode.

```
int SetKinTransformCartesian(
IN MMC_KINTRANSFORM_CARTESIAN_IN& pInParam
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Input Function Parameters

MMC_KINTRANSFORM_CARTESIAN_IN& pInParam

pInParam references the structure MMC_KINTRANSFORM_CARTESIAN_IN with default parameters, and either returns none, or throws CMMCException on failure.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

MMC_KINTRANSFORM_CARTESIAN_IN Structure

Parameters

All parameters

Refer to the section **MMC_KINTRANSFORM_CARTESIAN_IN Structure** for details of the parameters.

17.6.19 SetKinTransformScara

Sets the kinematic transformation parameters (MSC to ACS) for the SCARA robot. Refer to **5.15.17 MMC_SetKinTransformScara** for details of the description, scope, and motion mode.

```
int SetKinTransformScara(
IN MMC_KINTRANSFORM_SCARA_IN& pInParam
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Input Function Parameters

MMC_KINTRANSFORM_SCARA_IN& pInParam

pInParam references the structure MMC_KINTRANSFORM_SCARA_IN with default parameters, and either returns none, or throws CMMCException on failure.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

MMC_KINTRANSFORM_SCARA_IN Structure

Parameters

All parameters

Refer to the section **MMC_KINTRANSFORM_SCARA_IN Structure** for details of the parameters.

17.6.20 SetKinTransformThreeLink

Sets the kinematic transformation parameters (MSC to ACS) for the THREELINK robot. Refer to **5.15.18 MMC_SetKinTransformThreeLink** for details of the description, scope, and motion mode.

```
int SetKinTransformThreeLink(
IN MMC_KINTRANSFORM_THREELINK_IN& pInParam
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Input Function Parameters

MMC_KINTRANSFORM_THREELINK_IN& pInParam

pInParam references the structure MMC_KINTRANSFORM_THREELINK_IN with default parameters, and either returns none, or throws CMMCException on failure.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

MMC_KINTRANSFORM_THREELINK_IN Structure

Parameters

All parameters

Refer to the section **MMC_KINTRANSFORM_THREELINK_IN Structure** for details of the parameters.

17.6.21 RemoveAxisFromGroup

Refer to the section **5.15.29 MMC_RemoveAxisFromGroup** for details of the description, scope, and motion mode.

```
void RemoveAxisFromGroup(
NC_IDENT_IN_GROUP_ENUM eIdentInGroup
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

eIdentInGroup

The NC_IDENT_IN_GROUP_ENUM enumerator identifies the order and Nodes in the group of the added axis. Performed via an enumerator to give the different axes a name in the order, which can be coupled to the names in the kinematic model. The options are:

NC_NODE_1_ID = 0

……………

NC_NODE_16_ID = 15

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID

Status of the axis.

For code example, refer to the section **17.2.1, 17.6.2.**

17.6.22 MoveCircularAbsolute

Refer to the section **5.14.4 MMC_MoveCircularAbsolute** for details of the description, scope, and motion mode.

```
int MoveCircularAbsolute(
NC_ARC_SHORT_LONG_ENUM eArcShortLong,
NC_PATH_CHOICE_ENUM ePathChoice,
NC_CIRC_MODE_ENUM eCircleMode,
[double dAuxPoint[]]
[double dEndPoint[]]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

eArcShortLong

Defines the types of supported arc length. The NC_ARC_SHORT_LONG_ENUM enumerator options are:

MC_NONE_ARC_CHOICE = 0

MC_SHORT = 1

MC_LONG = 2

ePathChoice

Defines the NC_PATH_CHOICE_ENUM enumerator types of supported path choice. The option are:

MC_NONE_PATH_CHOICE = 0

MC_CLOCKWISE = 1

MC_COUNTERCLOCKWISE = 2

eCircleMode

Defines the types of supported circular modes in 2D. efer to the section **5.10.1 Coordinate System and kinematic transformation**, and the definitions below.

The NC_CIRC_MODE_ENUM enumerator options are:

MC_NONE_CIRC_MODE = 0

MC_BORDER_CIRC_MODE= 1

MC_CENTER_CIRC_MODE = 2

MC_RADIUS_CIRC_MODE = 3

MC_ANGLE_CIRC_MODE = 4

dAuxPoint[]

Absolute position for a dimension in the coordinate system specified by the input signal CoordSystem.

dAuxPoint can have double values in a technical unit [u].

dEndPoint[]

Absolute end point position for a dimension in the coordinate system specified by the input signal CoordSystem. *dEndPoint* is a 2D or 3D double in technical unit [u].

eBufferMode

Refer to the structure **MMC_MOVECIRCULARABSOLUTE_IN on page 453** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.6.1**.

17.6.23 MoveCircularAbsoluteCenter

Refer to the section **5.14.5 MMC_MoveCircularAbsoluteCenter** for details of the description, scope, and motion mode.

```
int MoveCircularAbsoluteCenter(
NC_ARC_SHORT_LONG_ENUM eArcShortLong,
double dBorderPoint[]
[double dCenterPoint[]]
[double dEndPoint[]]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

eArcShortLong

Defines the types of supported arc length. The NC_ARC_SHORT_LONG_ENUM enumerator options are:

MC_NONE_ARC_CHOICE = 0

MC_SHORT = 1

MC_LONG = 2

dBorderPoint

Absolute border position for a dimension in the coordinate system specified by the input signal CoordSystem. *dBorderPoint* can have double values in a technical unit [u].

dCenterPoint

Absolute position for a dimension in the coordinate system specified by the input signal CoordSystem. *dCenterPoint* can have double values in a technical unit [u].

dEndPoint

Absolute end point position for a dimension in the coordinate system specified by the input signal CoordSystem.*dEndPoint* is a 2D or 3D double vector in technical unit [u].

eBufferMode

Refer to the structure **MMC_MOVECIRCULARABSOLUTECENTER_IN Structure on page 461** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.6.1**.

17.6.24 MoveCircularAbsoluteBorder

Refer to the section **5.14.6 MMC_MoveCircularAbsoluteBorder** for details of the description, scope, and motion mode.

```
int MoveCircularAbsoluteBorder(
double dBorderPoint[],
[double dEndPoint[],]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

dBorderPoint

Absolute border position for a dimension in the coordinate system specified by the input signal CoordSystem. *dBorderPoint* can have double values in a technical unit [u].

dEndPoint

Absolute end point position for a dimension in the coordinate system specified by the input signal CoordSystem.*dEndPoint* is a 2D or 3D double vector in technical unit [u].

eBufferMode

Refer to the structure **MMC_MOVECIRCULARABSOLUTEBORDER_IN Structure on page 467** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.6.1**.

17.6.25 MoveCircularAbsoluteRadius

Refer to the section **5.14.7 MMC_MoveCircularAbsoluteRadius** for details of the description, scope, and motion mode.

```
int MoveCircularAbsoluteRadius(
NC_ARC_SHORT_LONG_ENUM eArcShortLong,
NC_PATH_CHOICE_ENUM ePathChoice,
[double dSpearHeadPoint[],]
[double dEndPoint[],]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

eArcShortLong

Defines the types of supported arc length. The NC_ARC_SHORT_LONG_ENUM enumerator options are:

MC_NONE_ARC_CHOICE = 0

MC_SHORT = 1

MC_LONG = 2

ePathChoice

Defines the NC_PATH_CHOICE_ENUM enumerator types of supported path choice. The option are:

MC_NONE_PATH_CHOICE = 0

MC_CLOCKWISE = 1

MC_COUNTERCLOCKWISE = 2

dSpearHeadPoint

Absolute radius position for a dimension in the coordinate system specified by the input signal CoordSystem. *dSpearHeadPoint* can have double values in a technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dEndPoint[]

Absolute end point position for a dimension in the coordinate system specified by the input signal CoordSystem. *dEndPoint* is a 2D or 3D double in technical unit [u].

eBufferMode

Refer to the structure **MOVECIRCULARABSOLUTERADIUS_IN Structure on page 473** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name Axis reference Error ID
Status of the axis.

17.6.26 MoveCircularAbsoluteAngle

Refer to the section **5.14.7 MMC_MoveCircularAbsoluteRadius** for details of the description, scope, and motion mode.

```
int MoveCircularAbsoluteAngle(
[double dAngle,]
[double dCenterPoint[],]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

dAngle

Relative angular position for the coordinate system specified by the input signal CoordSystem. Angular double value in degrees [u], which may be +ve or –ve without restriction.

dCenterPoint

Absolute position for a dimension in the coordinate system specified by the input signal CoordSystem. *dCenterPoint* can have double values in a technical unit [u].

eBufferMode

Refer to the structure **MOVECIRCULARABSOLUTERADIUS_IN Structure on page 473** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

For code example, refer to the section **17.6.1**.

17.6.27 MoveLinearAbsolute

Refer to the section **5.14.9 MMC_MoveLinearAbsolute** for details of the description, scope, and motion mode.

```
int MoveLinearAbsolute(
[float fVelocity,]
[double dbPosition[NC_MAX_NUM_AXES_IN_NODE],]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

fVelocity

Value of the maximum velocity (not necessarily reached). Any -ve or +ve double values in technical unit [u].

dbPosition

Target position for the motion of the axis when conditions are met. Any -ve or +ve double values in technical unit [u].

Array of coordinates, incl. positions and orientations (Distance if Mode = RELATIVE). The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eBufferMode

Refer to the structure **MMC_MOVELINEARABSOLUTE_IN Structure on page 486** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.6.1**.

17.6.28 MoveLinearRelative

Refer to the section **5.14.10 MMC_MoveLinearRelative** for details of the description, scope, and motion mode.

int MoveLinearRelative(
[float fVelocity,]
[double dbDistance[NC_MAX_NUM_AXES_IN_NODE],]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

fVelocity

Value of the maximum velocity (not necessarily reached). Any -ve or +ve double values in technical unit [u].

dbDistance

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eBufferMode

Refer to the structure **MMC_MOVELINEARRELATIVE_IN Structure on page 494** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.6.1**.

17.6.29 GroupSetOverride

Refer to the section **5.15.27 MMC_GroupSetOverride** for details of the description, scope, and motion mode.

```
int GroupSetOverride(
float fVelFactor,
float fAccFactor,
float fJerkFactor,
unsigned short usUpdateVelFactorIdx
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

fVelFactor

New override factor for the velocity. Any +ve float value between [0 – 1].

fAccFactor

New override factor for the acceleration/deceleration. ACC/Jerk Factors are NOT supported at this time. For future compatibility, enter "1" in the function call.

fJerkFactor

New override factor for the jerk. ACC/Jerk Factors are NOT supported at this time. For future compatibility, enter "1" in the function call.

usUpdateVelFactorIdx

Index of changed velocity factor. Vendor defined. The default is 0. Has integer values of 0 - 2

This variable is not in use at this moment.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

For code example, refer to the section **17.6.1**.

17.6.30 SetBoolParameter

Sets the Boolean Parameter for Group Axis. Refer to the section **5.9.28 MMC_WriteBoolParameter** for details of the description, scope, and motion mode.

```
void SetBoolParameter(
unsigned long ulValue,
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNode.h

GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

ulValue

Any integer value. +ve numeric value.

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.4 Axis Status on page 176** for the appropriate integer parameter to be used as enumerator.

iIndex

Index array (only relevant for array situations). Any +ve integer values

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.6.31 SetParameter

Sets an array Parameter for group axes. Refer to the section **5.9.32 MMC_WriteParameter** for details of the description, scope, and motion mode.

```
void SetParameter(
double dbValue,
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNode.h

GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

dbValue

Array parameter with double value.

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.4 Axis Status on page 176** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.6.32 GetBoolParameter

Obtains a Boolean Parameter for group axes. Refer to the section **5.9.14 MMC_ReadBoolParameter** for details of the description, scope, and motion mode.

```
Long GetBoolParameter(
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
)throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNode.h

GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.4 Axis Status** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values

Return

lValue Boolean parameters integer value

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.6.33 GetParameter

Obtains any group axes Parameter. Refer to the section **5.9.19 MMC_ReadParameter** for details of the description, scope, and motion mode.

```
double GetParameter(
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
)throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNode.h

GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3 Axis, Group, Global, Parameters** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values

Return

dbValue

Output of the specific parameter. Any Double value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.6.34 GroupSetPosition

Refer to the section **5.15.28 MMC_GroupSetPosition on page 620** for details of the description, scope, and motion mode.

```
int GroupSetPosition(
double dbPosition[],
MC_COORD_SYSTEM_ENUM eCoordSystem,
unsigned char ucMode,
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

dbPosition

Target position for the motion of the axis when conditions are met. Any -ve or +ve double values in technical unit [u].

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

ucMode

RELATIVE =True, ABSOLUTE = False (Default)

RELATIVE means that Position is added to the actual position value of the axis at the time of execution. This results in a recalibration by a specified distance. ABSOLUTE means that the actual position value of the axis is set to the value specified in the Position parameter.

Values accepted are Boolean, TRUE/FALSE.

eBufferMode

Refer to the structure **MMC_GROUPSETPOSITION_IN Structure on page 621** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.6.1**.

17.6.35 GroupReadStatus

Refer to the section **5.15.25 MMC_GroupReadStatus** for details of the description, scope, and motion mode.

unsigned long GroupReadStatus(
unsigned short& usGroupErrorID
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

usGroupErrorID

Returned command group error ID. Signals where an group error has occurred within function block. These values are vendor specific. Refer to the errors listed in sections **4.3 Maestro Error IDs,** and **4.9 NC Profiler Error IDs**.

Displays an error code integer.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.36 GetStatusRegister

Provides the status register. Refer to the section **8.1.19 MMC_GetStatusRegister** for details of the description, scope, and motion mode.

unsigned int GetStatusRegister(
[MMC_GETSTATUSREGISTER_OUT& sOutput]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MMC_GETSTATUSREGISTER_OUT& sOutput

Refer to the **MMC_GETSTATUSREGISTER_OUT Structure** for details of the parameters.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.37 GetMcsLimitRegister

This function returns the MCS limit register. This is the MCS Limit Register is a 32 bit representation of the software limit status of all kinematic directions, 16 directions * 2 limits (High\Low) = 32.

Refer to the section **8.1.19 MMC_GetStatusRegister** for details of the description, scope, and motion mode.

```
unsigned int GetMcsLimitRegister(
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.38 ReadStatus

This function obtains the axis status for a specific axis. Refer to the section **5.9.21 MMC_ReadStatus** for details of the description, and scope.

unsigned long ReadStatus(
[unsigned short& usAxisErrorID]
[unsigned short& usStatusWord]
) throw (CMMCException){return GroupReadStatus(usAxisErrorID);}

Source GMAS\includes\CPP\CMMCNode.h
GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

usAxisErrorID

Returns the axis error bitwise ID defined by the following enumerators. Bitwise ID error code:

Bit ID	Enumerator
0x1	MMC_ERR_TYPE_FAULT_BIT
0x2	MMC_ERR_TYPE_HEARTBEAT
0x4	MMC_ERR_TYPE_EMERGENCY
0x8	MMC_ERR_TYPE_COMM
0x10	MMC_ERR_TYPE_CFG_FILE

usStatusWord

Drive Status text. Any text characters.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section**17.2.1**.

17.6.39 Reset

Refer to the section 5.9.22 MMC_Reset for details of the description, scope, and communication mode.

```
void Reset(
)throw (CMMCException){GroupReset();}
```

Source GMAS\includes\CPP\CMMCNode.h
GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section 17.1.1 CMMCException. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.6.40 GetMembersInfo

Returns information about a specific group and its members. Refer to the section 5.15.34 MMC_GetGroupMembersInfo for details of the description, scope, and motion mode.

```
void GetMembersInfo(
MMC_GETGROUPMEMBERSINFO_OUT* stOutput
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MMC_GETGROUPMEMBERSINFO_OUT stOutput*

Refer to the MMC_GETGROUPMEMBERSINFO_OUT Structure for details of the parameters.

throw (CMMCException)

Refer to the section 17.1.1 CMMCException. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section 17.6.2.

17.6.41 GroupEnable

Refer to the section **5.15.21 MMC_GroupEnable** for details of the description, scope, and motion mode.

void GroupEnable(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

For code example, refer to the section **17.6.1**.

17.6.42 GroupDisable

Refer to the section **5.15.20 MMC_GroupDisable** for details of the description, scope, and motion mode.

void GroupDisable(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

For code example, refer to the section **17.6.1**.

17.6.43 GroupReset

Refer to the section **5.15.26 MMC_GroupReset** for details of the description, scope, and motion mode.

void GroupReset(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.44 GroupReadError

Refer to the section **5.15.24 MMC_GroupReadError** for details of the description, scope, and motion mode.

unsigned short GroupReadError(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.45 GroupReadActualVelocity

Refer to the section **5.15.23 MMC_GroupReadActualVelocity** for details of the description, scope, and motion mode.

double GroupReadActualVelocity(
MC_COORD_SYSTEM_ENUM eCoordSystem,
[double dVelocity[]]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

ePathChoice

Defines the NC_PATH_CHOICE_ENUM enumerator types of supported path choice. The option are:

MC_NONE_PATH_CHOICE = 0

MC_CLOCKWISE = 1

MC_COUNTERCLOCKWISE = 2

dVelocity[]

Current velocity of the group:

- For ACS the velocities of the different axes
- For MCS it provides the velocity of the TCP

dVelocity any -ve or +ve array double value in the axis's unit [u/s].

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

17.6.46 AddAxisToGroup

Refer to the section **5.15.19 MMC_AddAxisToGroup** for details of the description, scope, and motion mode.

```
void AddAxisToGroup(
NC_NODE_HNDL_T hNode,
NC_IDENT_IN_GROUP_ENUM eIdentInGroup
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

hNode

The NC_NODE_HNDL_T enumerator defines the Node handle transition. The axis ref parameter.

hNode can have any positive numeric value.

eIdentInGroup

The NC_IDENT_IN_GROUP_ENUM enumerator identifies the order and Nodes in the group of the added axis. Performed via an enumerator to give the different axes a name in the order, which can be coupled to the names in the kinematic model. The options are:

NC_NODE_1_ID = 0

……………

NC_NODE_16_ID = 15

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.6.2**.

17.6.47 GroupReadActualPosition

Refer to the section **5.15.22 MMC_GroupReadActualPosition** for details of the description, scope, and motion mode.

```
int GroupReadActualPosition(
MC_COORD_SYSTEM_ENUM eCoordSystem,
double dbPosition[]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

dbPosition[]

Target position for the motion of the axis when conditions are met. Any -ve or +ve double values in technical unit [u].

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name

Axis reference Error ID

Status of the axis.

17.6.48 GroupStop

Refer to the section **5.14.2 MMC_GroupStop on page 444** for details of the description, scope, and motion mode.

```
void GroupStop(
float fDeceleration,
float fJerk,
MC_BUFFERED_MODE_ENUM eBufferMode
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eBufferMode

Refer to the structure **MMC_GROUPSTOP_IN Structure on page 445** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.6.49 GroupHalt

Refer to the section **5.14.3 MMC_GroupHalt** for details of the description, scope, and motion mode.

```
void GroupHalt(
float fDeceleration,
float fJerk,
MC_BUFFERED_MODE_ENUM eBufferMode
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eBufferMode

Refer to the structure **MMC_GROUPHALT_IN Structure on page 449** for further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.6.50 MoveLinearAbsoluteRepetitive

Refer to the function block in section **5.14.13 MMC_MoveLinearAbsoluteRepetitive on page 511** for details of the description, scope, and motion mode.

```
int MoveLinearAbsoluteRepetitive(
[float fVelocity,]
[double dbPosition[NC_MAX_NUM_AXES_IN_NODE],]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

fVelocity

Value of the maximum velocity (not necessarily reached). Any -ve or +ve double values in technical unit [u].

dbPosition

Target position for the motion of the axis when conditions are met. Any -ve or +ve double values in technical unit [u].

Array of coordinates, incl. positions and orientations (Distance if Mode = RELATIVE). The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eBufferMode

Refer to the **MMC_MOVELINEARABSOLUTEREPETITIVE_IN Structure on page 512**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.6.51 MoveLinearRelativeRepetitive

Refer to **5.14.14 MMC_MoveLinearRelativeRepetitive on page 516** for details of the description, scope, and motion mode.

```
int MoveLinearRelativeRepetitive(
[float fVelocity,]
[double dbDistance[NC_MAX_NUM_AXES_IN_NODE],]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

fVelocity

Value of the maximum velocity (not necessarily reached). Any -ve or +ve double values in technical unit [u].

dbDistance

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

eBufferMode

Refer to the **MMC_MOVELINEARRELATIVEREPETITIVE_IN Structure**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID

Status of the axis.

17.6.52 MovePolynomAbsolute

Refer to **5.14.15 MMC_MovePolynomAbsolute** for details of the description, scope, and motion mode.

int MovePolynomAbsolute(
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
[float fVelocity,]
[double dbAuxPoint[NC_MAX_NUM_AXES_IN_NODE]]
[double dbDistance[NC_MAX_NUM_AXES_IN_NODE]]
[double dbPosition[NC_MAX_NUM_AXES_IN_NODE]]
[float fAcceleration]
[float fDeceleration]
[float fJerk]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE

Refer to the **MMC_MOVELINEARRELATIVEREPETITIVE_IN Structure**.

fVelocity

Value of the maximum velocity (not necessarily reached). Any -ve or +ve double values in technical unit [u].

dbAuxPoint[NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dbDistance[NC_MAX_NUM_AXES_IN_NODE]

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

dbPosition[NC_MAX_NUM_AXES_IN_NODE]

Target position for the motion of the axis when conditions are met. Any -ve or +ve double values in technical unit [u].

Array of coordinates, incl. positions and orientations (Distance if Mode = RELATIVE). The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in u/s^{2}.

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.6.53 MoveLinearAdditive

Refer to the sections **5.14.11 MMC_MoveLinearAdditive** and **5.14.12 MMC_MoveLinearAdditiveEx on page 501 - 506** for details of the description, scope, and motion mode. The MMC_MoveLinearAdditiveEx parameter is for further accuracy in setting the parameters. The double parmeters allow setting of an 8 bit value.

```
int MoveLinearAdditive(
[float fVelocity,]
[double dbDistance[NC_MAX_NUM_AXES_IN_NODE],]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

fVelocity

Value of the maximum velocity (not necessarily reached). Any -ve or +ve double values in technical unit [u].

dbDistance

Array [1..N] of relative distances for each dimension in the specified coordinate system, with N being vendor specific. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

dbDistance is a double vector array in technical unit [u].

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE

Refer to the **MMC_MOVELINEARADDITIVE_IN Structure on page 502**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.6.54 MovePath

Refer to section **5.14.17 MMC_MovePath** for details of the description, scope, and motion mode.

void MovePath(
MC_PATH_REF hMemHandle,
[float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE]]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
[MC_COORD_SYSTEM_ENUM eCoordSystem = MC_MCS_COORD]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle has integer values.

float fTransitionParameter[NC_MAX_NUM_AXES_IN_NODE

Depending on the transition mode, different supplier specific transition parameters can be used which characterize the contour curve. The array parameter NC_MAX_NUM_AXES_IN_NODE is limited to 16, and defined as the maximum number of axis in a group.

fTransitionParameter can have any positive float value in appropriate units, dependant on the TransitionMode parameter. Refer to the section ***5.10.9 Special Robot Transformations.***

[NC_MAX_NUM_AXES_IN_NODE] is an array of values [2….15].

MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Refer to the **MMC_MOVEPATH_IN Structure on page 530** for further details. Modes are as follows, but only the Buffered Mode is supported:

MC_ABORTING_MODE = 1
MC_BUFFERED_MODE = 2,
MC_BLENDING_LOW_MODE = 3
MC_BLENDING_PREVIOUS_MODE = 4
MC_BLENDING_NEXT_MODE = 5
MC_BLENDING_HIGH_MODE = 6

Aborting Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared

Buffered The next function block affects the axis as soon as the previous movement is completed.

BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD	= 0
MC_ACS_COORD	= 1
MC_MCS_COORD	= 2
MC_PCS_COORD	= 3

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.6.55 PathDeselect

Refer to **5.14.18 MMC_PathUnselect** for details of the description, scope, and motion mode.

int PathDeselect(
MC_PATH_REF hMemHandle
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MC_PATH_REF hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle has integer values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name	Axis reference
Error ID	Status of the axis.	

17.6.56 PathSelect

Refer to the function **5.14.16 MMC_PathSelect** for details of the description, scope, and motion mode.

unsigned int PathSelect(
MC_PATH_DATA_REF pPathToSplineFile
[MC_COORD_SYSTEM_ENUM eCoordSystem]
[unsigned char ucExecute = 1]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MC_PATH_DATA_REF pPathToSplineFile

This string describes where the splines data file is located. Values accepted are any characters describing a file path.

MC_PATH_DATA_REF Where the enumerator MC_PATH_DATA_REF describes the I/O definition of the path data reference using the array [NC_MAX_SPLINES_FILE_PATH_LENGTH] that defines the maximum length of the splines file path data.

MC_PATH_DATA_REF can have values of any characters.

NC_MAX_SPLINES_FILE_PATH_LENGTH can have any numeric value.

MC_COORD_SYSTEM_ENUM eCoordSystem

Define the types of supported coordinate systems. The MC_COORD_SYSTEM_ENUM enumerator options are:

MC_NONE_COORD = 0

MC_ACS_COORD = 1

MC_MCS_COORD = 2

MC_PCS_COORD = 3

unsigned char ucExecute = 1

Start the execution command. Boolean TRUE/FALSE values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.6.57 PathGetLengths

Retrieves the length values of specified segments in a spline table. The buffer must comply to the number of values, which the programmer expects to recieve and cannot be greater than 170 elements.

```
unsigned int PathGetLengths(
MC_PATH_REF hMemHandle,
unsigned int uiStartIndex,
unsigned int uiNumOfSegments,
double *dbValues
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MC_PATH_REF hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle has integer values.

uiStartIndex

Specifies the table segment from which to start collecting the length values of specified segments in a spline table. Any +ve value.

uiNumOfSegments

Specifies the number of segments for the length values collection. Any +ve value.

dbValues

The buffer in which this function stores the collected values. Any +ve values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.6.58 EthercatWriteMemoryRange

Refer to **5.15.11 MMC_SetKinTransformEx** for details of the description, scope, and motion mode.

```
void EthercatWriteMemoryRange(
unsigned short usRegAddr,
unsigned char ucLength,
unsigned char pData[ETHERCAT_MEMORY_WRITE_MAX_SIZE]
) throw (CMMCException){return;}
```

Source GMAS\includes\CPP\MMCGroupAxis.h

.NET Definition

Function Parameters

MC_KIN_REF_DELTA stDelta

Refer to the parameter definition in the function **5.15.11 MMC_SetKinTransformEx** for details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.7 The CMMCSingleAxis class

The class CMMCSingleAxis wraps the single axis functions detailed in the section 5.4 **Axis Status on page 176**. The diagram in **Figure 17-8** describes the heirarchial structure of the classes and type definitions associated with the CMMCSingleAxis.

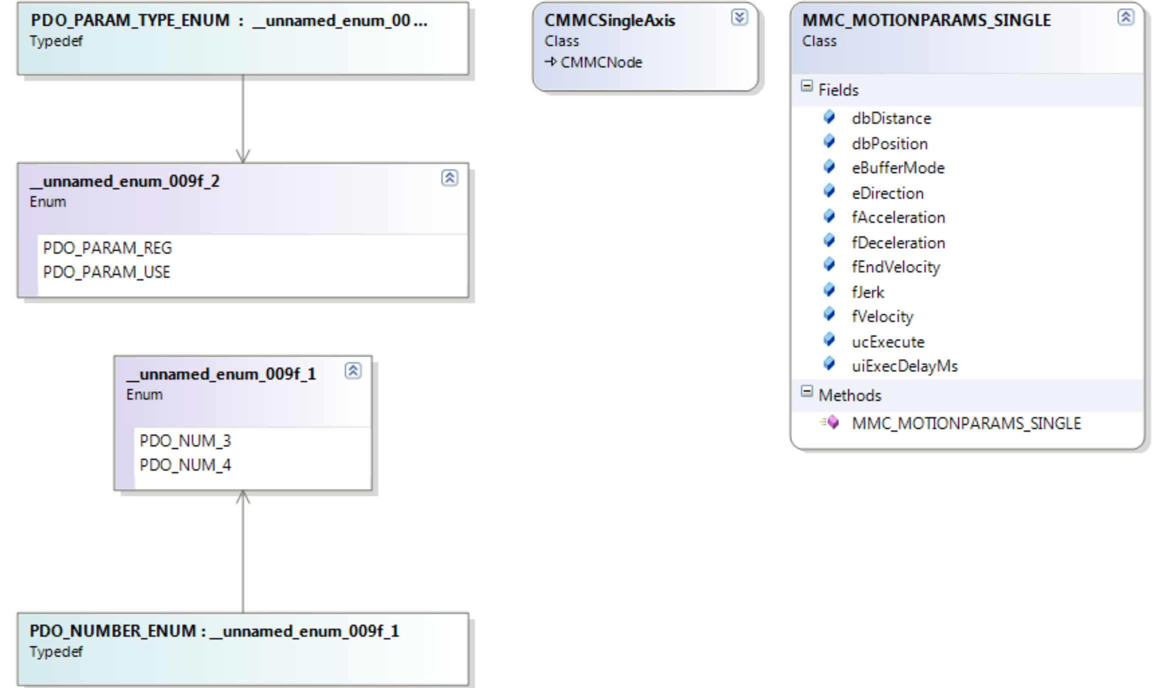


Figure 17-8 CMMCSingleAxis class diagram

The class CMMCSingle retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

It should be noted that Private functions and their operation should be transparent to the user, and are not for general application by the user.

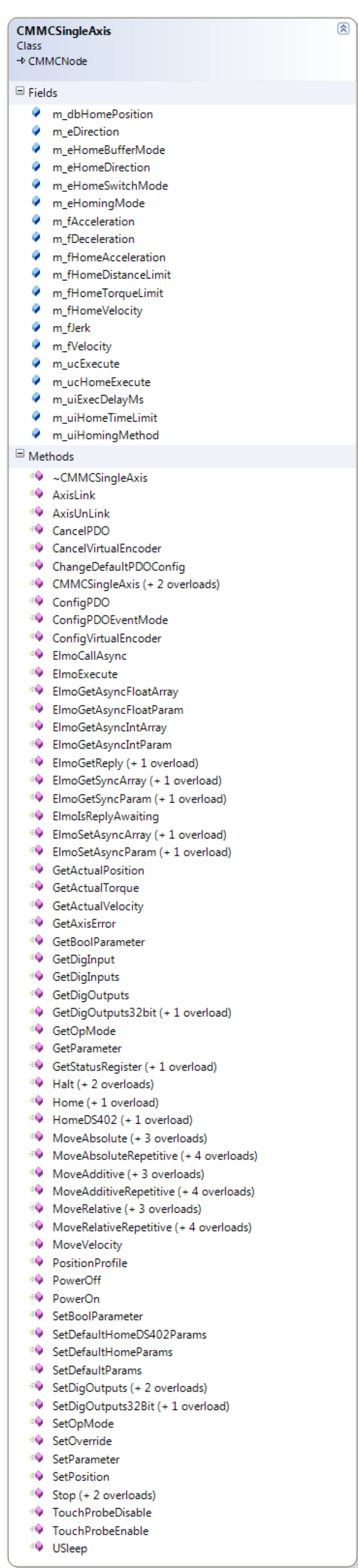


Figure 17-9 Fields and methods of the CMCSingleAxis class

The detailed class view shown in **Figure 17-9**, describes the fields and methods associated with the CMMCSingleAxis class. These are generally default parameters, which can be operated using their default values. However if the user wishes to change the defaults, refer to the relevant parameter section in the manual.

17.7.1 CMMCSingleAxis Class Functions Code Example 1

```
#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define      EndMotionEventCB_MESSAGE    "!!!!END MOTION EVENT MESSAGE!!!"
#define STAR_DIM   2


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     MMC_CONNECT_HNDL         ComHndl;
     CMMCSingleAxis           AxisA, AxisB;
     CMMCConnection           cConn;
     CMMCGroupAxis            Group;
     MMC_MOTIONPARAMS_GROUP   ParamsGroup;
     MMC_SETKINTRANSFORM_IN   SetKin;
     unsigned short           AxisARef, AxisBRef;

     double    StarPoi[][STAR_DIM] = {
                                   { 7500,        0},
                                   {12500,    13100},
                                   {  200,     5600},
                                   {14700,     5600},
                                   { 3340,    13750},
                                   { 7500,     7500}/* ~ the Center cordination */
                                   };
     char * delimit = "===========================================================================";
     char  * strStrSnro = "\n <<<<<<<<<<<<< Start ";
     char  * strEndSnro1 = "\n End ";
     char  * strEndSnro2 = " >>>>>>>>>>>>> ";

     int     WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);
     void   initAdminSingleAxis(void);
     void   initAdminMultiAxis();
     void   endAdminSingle(void);
     void   endAdminMultiAxis(void);

     void   SnroMoveCombinations(int);
     void   SnroSetGetParameters(int);
     void   SnroEnableDisableMotionEndedEvent(int);
     void   SnroSetDefParamHome(int);
     void   SnroMoveAbsolute(int);
     void   SnroDepthName(int);

     void   EnableDisableMotionEndedEvent(void);

     void   SetGrpKinDef(void);
     void   SetGetParameters(void);
     void   SetGetGroupParam(void);
     void   SetDefManufact(void);
     void   SetGetDrvParameters(void);
     void   DepthName(void);

     void SetDefParamsHome(void);
```

```
     void  MoveCombinations(void);
     void   MoveAdditiveMoves(void);
     void   MoveRelativeVelMoves(void);

     void   MoveAbsRepetiveMoves(void);
     void   MoveAdditiveRepetiveMoves(void);
     void   MoveRelativeRepetiveMoves(void);

     void  MoveAbsoluteMoves(void);
     int     CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
     int     OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus);
     void EndMotionEventCB(unsigned short usAxisRef);
     void  ModbusWrite_Received();
     void  Emergency_Received(unsigned short usAxisRef, short sEmcyCode);

/*========== Administration functions STR ============================================*/
int     main(int)
// ==============
{
     int   trace = 1;

     printf("\n %s", delimit);

     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);
try
{

     SnroMoveCombinations(trace++);
     SnroMoveAbsolute(trace++);

     SnroSetDefParamHome(trace++);
     SnroSetGetParameters(trace++);
     SnroEnableDisableMotionEndedEvent(trace++);
     SnroDepthName(trace++);
}
catch (CMMCException excp)
{
     printf("\n %s", delimit);
     printf("\n %s", delimit);
     printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
     printf("\n %s", delimit);
     printf("\n %s", delimit);
     exit(0);
}

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


int     WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//=====================================================================
{
               int  end_of = 0;
               int  iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
```

```
        end_of = 1;
                  /* Read Axis Status command server for specific Axis */
        ulState = sng_axis->ReadStatus();
        if (!(ulState & break_state))
        {
            end_of = 0;

            WAIT_SLEEP_MILLI(20)

        }
    }

    if(0)
    {
        MMC_SHOWNODESTAT_IN showin;
        MMC_SHOWNODESTAT_OUT showout;
        MMC_ShowNodeStatCmd(ComHndl,sng_axis->GetRef(),&showin,&showout);
    }

    return 0;
}

// 16.3.2. void CMMCSingleAxis::SetDefaultHomeDS402Params(const MMC_HOMEDS402_IN& stSingleParams)  (-
no corresponding MMC_ C func)
// 16.3.3. void CMMCSingleAxis::SetDefaultHomeParams(const MMC_HOME_IN& stSingleParams)          (- no
corresponding MMC_ C func)
//
// 16.3.5.  void CMMCSingleAxis::HomeDS402()                          5.6.3. MMC_HomeDS402
// 16.3.5.  void CMMCSingleAxis::HomeDS402(MMC_HOMEDS402_IN stHomeDS402Params) 5.6.3. MMC_HomeDS402
//
// 16.3.16. SetOpMode int CMMCSingleAxis::SetOpMode(OPM402 eMode) 11.5.9. MMC_ChngOpMode
// 16.3.17. GetOpMode OPM402 CMMCSingleAxis::GetOpMode()     MMC_ReadBoolParameter
    void SetDefParamsHome(void)
//  ==============================
{
    MMC_HOMEDS402_IN  stDS402Home ;
    MMC_HOME_IN          stSingleParams;
    OPM402                   drvMode;
    short                     sErrorID;

printf("\n                %s:", __func__);
    stDS402Home.dbPosition     = -1000000 ;
    stDS402Home.eBufferMode   = MC_BUFFERED_MODE;
    stDS402Home.fAcceleration      = 100000;
    stDS402Home.fDistanceLimit    = 100000;
    stDS402Home.fTorqueLimit = 1;
    stDS402Home.fVelocity       = 100000;
    stDS402Home.uiHomingMethod    = 35; // Homing method immediate
    stDS402Home.uiTimeLimit   = 100000;
    stDS402Home.ucExecute       = 1;
    AxisA.SetDefaultHomeDS402Params(stDS402Home);

    stSingleParams.dbPosition      = 2000000;
    stSingleParams.eBufferMode    = MC_BUFFERED_MODE; /* MC_ABORTING_MODE; */
    stSingleParams.fAcceleration = 200000;
    stSingleParams.fDistanceLimit= 400000;
    stSingleParams.fTorqueLimit  = 1;
    stSingleParams.fVelocity  = 200000;
    stSingleParams.ucExecute  = 1;
    stSingleParams.uiTimeLimit    = 50000;
    stSingleParams.eHomingMode    = MC_DIRECT;
    stSingleParams.eDirection      = MC_POSITIVE;
    stSingleParams.eSwitchMode      = MC_ON;
    AxisA.SetDefaultHomeParams(stSingleParams);
```

```
    drvMode = AxisA.GetOpMode();
                                        /* Cannot be done on Virtual Axis */
    AxisA.SetOpMode(OPM402_HOMING_MODE);

    stDS402Home.dbPosition     = 3000000;
    stDS402Home.fAcceleration      =  300000;
    stDS402Home.fVelocity      =  300000;
    stDS402Home.fDistanceLimit    =  300000;
    stDS402Home.uiTimeLimit   =    30000;

    AxisA.HomeDS402(stDS402Home);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    AxisA.HomeDS402();
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

                    /* Retrive the keeping mode */
    AxisA.SetOpMode(drvMode);

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    initAdminMultiAxis();
    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

    Group.GroupEnable();

    SetGrpKinDef();

    Group.GroupDisable();
    AxisB.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);

    endAdminMultiAxis();
}
```

17.7.2 CMMCSingleAxis Class Functions Code Example 2

```
/*
============================================================================
Collection of Gmas API functions (Set #2)
Examples for document: "G-MAS Administrative and Motion API.pdf".
09Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"
#define STAR_DIM                       2


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection          gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCHostComm            cHost ;

     CMMCSingleAxis          AxisA,    AxisB;
     unsigned short               AxisARef, AxisBRef;

     CMMCGroupAxis                Group;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

     char * delimit = "============================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void  endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void      EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void      MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroMbusfunc(int);
     void mBusReadWriteHoldReg(void);
     void mBusReadWriteCoil(void);
     void mBusReadWriteInput(void);
     void moveAxisAAndRepActualParamOnMbusPos(void);
     void      RepAxisAActualParamOnMbus(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
```

```
    int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus);
    void      EndMotionEventCB(unsigned short usAxisRef);
    void ModbusWrite_Received();
    void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
    int       mapConfigPdo(void);


/*================================ Administration functions STR ===================================*/
int       main(int)
// ==============
{
    int  trace = 1;

    printf("\n %s", delimit);
    printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

    try
    {
        SnroMoveAbsolute(trace++);
        SnroEnableDisableMotionEndedEvent(trace++);
        SnroDepthName(trace++);

        SnroMbusfunc(trace++);
    }
    catch (CMMCException excp)
    {
        printf("\n %s", delimit);
        printf("\n %s", delimit);
        printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
        printf("\n %s", delimit);
        printf("\n %s", delimit);
        exit(0);
    }

    printf("\n End of %s ", __FILE__);
    printf("\n %s\n\n", delimit);
    return 0;
}


//   15.11.3.  CMMCNode::ReadStatus     1374
int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//================================================================
{
              int end_of = 0;
              int iCount = 0;
    unsigned int   ulState;

    while( ! end_of)
    {
        iCount ++;
        end_of = 1;
/* Read Axis Status command server for specific Axis */
        ulState = sng_axis->ReadStatus();
        if (!(ulState & break_state))
        {
            end_of = 0;

            WAIT_SLEEP_MILLI(20)
        }
    }

//        MMC_SHOWNODESTAT_IN  showin;
//        MMC_SHOWNODESTAT_OUT showout;
//        MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

    return 0;
}


// 15.6.5.     CMMCConnection::ConnectIPCEx  1335
```

```
// 15.6.13.    CMMCConnection::CallbackFunc  1345
void initAdminSingleAxis(void)
// ==============================
{
     int iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;



/* CallbackFunc in ConnectIPCEx call if there    */
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */

/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters             */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk     = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute     = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


//   15.4.21.  CMMCGroupAxis::AddAxisToGroup
void initAdminMultiAxis()
// =========================
{
//   MMC_CONNECT_HNDL        ComHndl;
//   CMMCSingleAxis          AxisA,    AxisB;
//   CMMCGroupAxis           Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// =============================
{
     MMC_CloseConnection(ComHndl) ;
}


//   15.4.5.   CMMCGroupAxis::RemoveAxisFromGroup 1197
     void endAdminMultiAxis(void)
// ================================
{

//   CMMCGroupAxis            Group;
```

```
     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
     Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================= Administration functions END ====================================*/


/*================================= Scenario functions STR ==========================================*/
void SnroMoveAbsolute(int trace)
// ================================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

     MoveAbsoluteMoves();

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

//   15.6.14.  CMMCConnection::RegisterEventCallback  1358

void SnroEnableDisableMotionEndedEvent(int trace)
// =================================================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

     EnableDisableMotionEndedEvent();

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
     endAdminSingleAxis();

     gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


void SnroDepthName(int trace)
// =============================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     initAdminMultiAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

     Group.GroupEnable();

     DepthName();

     Group.GroupDisable();

     AxisB.PowerOff(MC_BUFFERED_MODE);
     AxisA.PowerOff(MC_BUFFERED_MODE);
```

```
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminMultiAxis();
    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


//   15.8.6.   cHost::MbusIsRunning     1389

//   15.8.2.   cHost::MbusStartServer   1386

//   15.8.3.   cHost::MbusStopServer    1387

void SnroMbusfunc(int trace)
// ============================
{
static    bool      mBusServerState;


    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
/* Starting MBus server           */
/* map read/write user req. to Mbus    */
/* area & responsible to Mbus commu.*/
    mBusServerState = cHost.MbusIsRunning(ComHndl);
    if (mBusServerState)
    {
        printf("\n ... ModBus Server is already running ");
    }
    else
    {
        printf("\n ... ModBus Server is NOT running - activate it ");
    }
/* Call to StartServer even if it   */
/* already running, it not start    */
/* the server because it alrady so  */
/* but for initialize init struct */
    cHost.MbusStartServer(ComHndl, 1) ;
                              /* Mode Bus functions examples.        */
    mBusReadWriteHoldReg();
/* Move Axis A and report it poss.  */
/* on Mbus.*/
    moveAxisAAndRepActualParamOnMbusPos();
    mBusReadWriteCoil();
    mBusReadWriteInput();

/* If I was the one who activate the   */
/* Mbus server I stop it when I finish.*/
    if (! mBusServerState)
    {
/* Stop MBus server */
        cHost.MbusStopServer();
        printf("\n ... Stop ModBus Server ");
    }
    else
    {
        printf("\n ... Left ModBus Server running ");
    }

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
/*================================ Scenario functions END ========================================*/



/*================================ Example functions STR =========================================*/

void EnableDisableMotionEndedEvent(void)
// ========================================
```

```
{
     int  loopInd;

printf("\n Function: %s:", __func__);
     for (loopInd = 0; loopInd < 2; loopInd++)
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}

void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

// iVal3 = AxisA.GetAxisByName("A01"); /* It case sensitive - Not define - exception... */

     iVal1 = Group.GetGroupAxisByName("v01");    /* Expected 256 */
/*
 *   iVal2 = Group.GetGroupAxisByName("v02");
 */
}


void MoveAbsoluteMoves(void)
// ============================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:    */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0 */
/* update default speed to 5000000     */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters       */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity */
/* v=100000    which become the new def V    */
     AxisA.MoveAbsolute(-300000.0);
```

```
/* Move to 310000 at velocity 80000.0  */
/* new def v=80000                       */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:    */
/*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=            */
/* MC_BUFFERED_MODE (def)    */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
/* Move abs to 350000 with parameters from  */
/* above command which become the default:  */
/*        Speed 500000, Acc=1000000         */
/*        Dec=1500000, Jerk=20000000,       */
/*        buffer mode=MC_BUFFERED_MODE (def)  */
     AxisA.MoveAbsolute(350000);
}


//   15.6.13.  CMMCConnection::CallbackFunc
int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{
//   int  ind;

    printf("\n *********** STR Func: %s *********** ", __func__);

//    printf("\n");
//    for (ind=0; ind < recvBufferSize; ind++)
//    {
//        printf("   recvBuffer[%d]=%d", ind, recvBuffer[ind]);
//    }
//    printf("\n");

/* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
          printf("\n Underflow event received ");
          break ;
     case MODBUS_WRITE_EVT:
          printf("\n ModBus Write event received ");
          break ;
     case TOUCH_PROBE_ENDED_EVT:
          printf("\n Touch Probe event received ");
          break ;
     default:
          printf("\n Default.... Whatever arrived event received ");
          break;
     }

    printf("\n *********** END Func: %s *********** ", __func__);
```

```
    fflush(stdout); fflush(stderr);

    return 1 ;
}


//
// Needs registeration, E.g:  CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);
//
int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
//
===========================================================================================================
{
    printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
            msg, usAxisRef, sErrorID, usStatus);
    fflush(stdout); fflush(stderr);

    MMC_CloseConnection(uiConnHndl);
    exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
    printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
    printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
    fflush(stdout); fflush(stderr);
}

/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ===========================
{
    printf("\n %s Received ", __func__) ;
    fflush(stdout); fflush(stderr);
}

/* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ================================================================
{
    printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
    fflush(stdout); fflush(stderr);
}


#define   MODBUS_UPDATE_START_INDEX              0
#define   MODBUS_UPDATE_CNT                  4
                            /* WRITE INDEXS extend along two indexs */
#define MODBUS_STR_REP_ACT_VAL_INDX          (MODBUS_UPDATE_START_INDEX+MODBUS_UPDATE_CNT)
#define   MBUS_WAIT_FOR_USER                     30000

//   15.8.4.   cHost::MbusReadHoldingRegisterTable    1388
//   15.8.5.   cHost::MbusWriteHoldingRegisterTable   1389
void mBusReadWriteHoldReg(void)
// ================================
{
        int       startRef;
        int       refCnt;
        int           loopCount = 0;

    MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN     mbus_write_in;
    MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT     mbus_read_out;

    printf("\n\n Function: %s: ", __func__);

    memset(mbus_write_in.regArr,0x0, 250) ;

    startRef = MODBUS_UPDATE_START_INDEX;
    refCnt    = MODBUS_UPDATE_CNT;
    mbus_write_in.startRef        = startRef;
```

```
    mbus_write_in.refCnt         = refCnt;
    mbus_write_in.regArr[0]  = 0xf0f0;
    mbus_write_in.regArr[1]  = 0x0f0f;
    mbus_write_in.regArr[2]  = 0x5a5a;
    mbus_write_in.regArr[3]  = 0xa5a5;

    printf("\n ... You can activate the EAS and look on EAS Mbus reg 1-4 ");

    printf("\n ... Going deal with Mbus H.Reg. EAS ind 1-4 ");
    loopCount = 0;
    do
    {
        loopCount++;
/* write into mBus Holding register */
/* data on reg. ind 0-3            */
        cHost.MbusWriteHoldingRegisterTable(mbus_write_in);
        printf("\n ... Mbus Write  H.Reg ind 0-3: %8x %8x %8x %8x ",
                                          (int)mbus_write_in.regArr[0],
                                          (int)mbus_write_in.regArr[1],
                                          (int)mbus_write_in.regArr[2],
                                          (int)mbus_write_in.regArr[3]);
        printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);

        WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
/* read mbus H.Reg start from            */
/* location startRef along refCnt reg*/

        cHost.MbusReadHoldingRegisterTable(startRef, refCnt, mbus_read_out);
        printf("\n ... Mbus Readed H.Reg ind 0-3: %8x %8x %8x %8x ",
                                           (int)mbus_read_out.regArr[0],
                                           (int)mbus_read_out.regArr[1],
                                           (int)mbus_read_out.regArr[2],
                                           (int)mbus_read_out.regArr[3]);

        mbus_write_in.regArr[0] = ~(mbus_read_out.regArr[0]);
        mbus_write_in.regArr[1] = ~(mbus_read_out.regArr[1]);
        mbus_write_in.regArr[2] = ~(mbus_read_out.regArr[2]);
        mbus_write_in.regArr[3] = ~(mbus_read_out.regArr[3]);

    } while (loopCount < 3);
}

//   15.8.7.   cHost::MbusReadCoilsTable    1390
//   15.8.8.   cHost::MbusWriteCoilsTable   1391
void mBusReadWriteCoil(void)
// ============================
{
        int  startRef;
        int  refCnt;

        int  loopCount = 0;

    MMC_MODBUSWRITECOILS_IN      stInParams_Coil;
    MMC_MODBUSREADCOILS_OUT      stOutParams_Coil;


    printf("\n\n Function: %s: ", __func__);

    startRef = MODBUS_UPDATE_START_INDEX;
    refCnt   = MODBUS_UPDATE_CNT;
    stInParams_Coil.startRef= startRef;
    stInParams_Coil.refCnt   = refCnt;
    stInParams_Coil.coilsArr[0] = (char)0xf0;
    stInParams_Coil.coilsArr[1] = (char)0x0f;
    stInParams_Coil.coilsArr[2] = (char)0x5a;
    stInParams_Coil.coilsArr[3] = (char)0xa5;

    printf("\n ... Going deal with Mbus Coils EAS ind 1-4 ");
    loopCount = 0;
    do
    {
        loopCount++;

/* write into mBus Coil register data on reg. ind 0-3    */
```

```
        cHost.MbusWriteCoilsTable(stInParams_Coil);
        printf("\n ... Mbus Write  Coil ind 0-3: %8x %8x %8x %8x ",
                                   (int)stInParams_Coil.coilsArr[0],
                                   (int)stInParams_Coil.coilsArr[1],
                                   (int)stInParams_Coil.coilsArr[2],
                                   (int)stInParams_Coil.coilsArr[3]);
        printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
        WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)

/* read mbus Coil start from location startRef along refCnt reg*/

        cHost.MbusReadCoilsTable(stInParams_Coil.startRef, stInParams_Coil.refCnt, stOutParams_Coil);
        printf("\n ... Mbus Readed Coil ind 0-3: %8x %8x %8x %8x ",
                                   (int)stOutParams_Coil.coilsArr[0],
                                   (int)stOutParams_Coil.coilsArr[1],
                                   (int)stOutParams_Coil.coilsArr[2],
                                   (int)stOutParams_Coil.coilsArr[3]);

        stInParams_Coil.coilsArr[0] = ~(stOutParams_Coil.coilsArr[0]);
        stInParams_Coil.coilsArr[1] = ~(stOutParams_Coil.coilsArr[1]);
        stInParams_Coil.coilsArr[2] = ~(stOutParams_Coil.coilsArr[2]);
        stInParams_Coil.coilsArr[3] = ~(stOutParams_Coil.coilsArr[3]);

    } while (loopCount < 3);
}


//   15.8.9.   cHost::MbusReadInputsTable   1392

void mBusReadWriteInput(void)
// =============================
{
    int  startRef;
    int  refCnt;

    int  loopCount = 0;

    MMC_MODBUSREADINPUTS_OUT stOutParams_Input;

    printf("\n\n Function: %s: ", __func__);

    printf("\n ... Going deal with Mbus Inputs EAS ind 1-4 ");
    startRef= MODBUS_UPDATE_START_INDEX;
    refCnt    = MODBUS_UPDATE_CNT;
    loopCount = 0;
    do
    {
        loopCount++;
        cHost.MbusReadInputsTable(startRef, refCnt, stOutParams_Input);
        printf("\n ... Mbus Readed inputs ind 0-3: %8x %8x %8x %8x ",
                  (int)stOutParams_Input.inputsArr[0], (int)stOutParams_Input.inputsArr[1],
(int)stOutParams_Input.inputsArr[2], (int)stOutParams_Input.inputsArr[3]);

        printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
        WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
    } while (loopCount < 3);
}

#define   UL_TO_MBUS_STRUCT(modbus_write_val, aux_ul_p, mod_bus_idx)                 \
    {                                                                                \
            aux_ul_p = (unsigned long *)(& mbus_write_in.regArr[mod_bus_idx]);   \
            * aux_ul_p = modbus_write_val;                                                \
    }                                                                                \

//   15.3.22.  CMMCSingleAxis::GetActualPosition      1212

//   15.3.23.  CMMCSingleAxis::GetActualVelocity      1212

//   15.3.24.  CMMCSingleAxis::GetActualTorque        1213

//   15.6.9.   CMMCConnection::GetGlobalBoolParameter 1245

void RepAxisAActualParamOnMbus(void)
// ================================
{
    MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN     mbus_write_in;
                              /* Auxiliary var for write to ModBus */
```

```
    unsigned  long*     ul_p;
    unsigned  long ul_val;
                   double    dActul;

    dActul = AxisA.GetActualPosition();
    ul_val = (unsigned long)dActul;
/* Put value into ModBus write array use 2 index poss (0 & 1).  */
    UL_TO_MBUS_STRUCT(ul_val, ul_p, 0);

/* Remmber about mapping & config !!! */
/* if the motor connected above ehercat it needs apropriate config setting (EAS).  */
    dActul = AxisA.GetActualVelocity();
    ul_val = (unsigned long)dActul;
                              /* Put value into ModBus write array use 2 index poss (2 & 3).  */
    UL_TO_MBUS_STRUCT(ul_val, ul_p, 2);


/* Remmber about mapping & config (see GetActualVelocity) !!!   */
    dActul = AxisA.GetActualTorque();
    ul_val = (unsigned long)dActul;
/* Put value into ModBus write array use 2 index poss (4 & 5).  */
    UL_TO_MBUS_STRUCT(ul_val, ul_p, 4);


/* Index of modbus H.reg. for start write actal values. */
    mbus_write_in.startRef        = MODBUS_STR_REP_ACT_VAL_INDX;
/* Number of indexes to write - each long extend along     */
/* 2 indexs (2 for Pos, 2 for Vel, 2 for Torq              */
    mbus_write_in.refCnt = 6;
    cHost.MbusWriteHoldingRegisterTable(mbus_write_in) ;

    return ;
}


#define   eCOMM_TYPE_ETHERCAT      1
#define   eCOMM_TYPE_CAN           2
/* Move Axis A and report its Actual parameters on Mbus.*/
void moveAxisAAndRepActualParamOnMbusPos(void)
// ==============================================
{
                int ind,
                    rt;
    unsigned int   uiState,
                    uiDoneFlg;

    printf("\n\n Function: %s: ", __func__);

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


/* Connection type - CAN/EtherCAT */
    rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);
    if (rt == eCOMM_TYPE_ETHERCAT)
    {
        printf("\n --- Motor connection is via ETHERCAT remmber map & config for get actul param 
(Vel. Torq.) ");
    }
    else if (rt == eCOMM_TYPE_CAN)
    {
        printf("\n --- Motor connection is via CAN ");
        if ( mapConfigPdo() )
        {
            printf("\n --- ERROR FAIL IN MAP PDO FOR CAN  !!! ");
        }
    }
    else
    {
        printf("\n --- ERROR UNKNOWN motor connection  !!! ");
    }

    printf("\n ... Start rep AxisA pos on Mbus");
    for(ind=0; ind<3; ind++)
```

```
     {
/* Move abs to: 400000, with parameters:         */
/*            Speed 300000, Acc=500000           */
/* Dec=1500000, Jerk=20000000, buffer mode=      */
/* MC_BUFFERED_MODE (def)        */
          AxisA.MoveAbsolute(400000*ind, 300000, 500000, 1500000, 20000000);
          do
          {
                              /* Write AxisA pos. to Modbus H.Reg */
               RepAxisAActualParamOnMbus();
               uiState = AxisA.ReadStatus();
               uiDoneFlg = uiState & NC_AXIS_STAND_STILL_MASK;
               if ( ! uiDoneFlg)
               {
                    WAIT_SLEEP_MILLI(40)
               }
          } while(! uiDoneFlg);
     }
     printf("\n ... End rep AxisA pos on Mbus");

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
}


int  mapConfigPdo(void)
// ===================
{
     int  rc;
     unsigned short                AxisARef;


     MMC_CONFIGREGULARPARAMEVENTPDO3_INCfgReg3In;
    MMC_CONFIGREGULARPARAMEVENTPDO3_OUT CfgReg3Out;

    CfgReg3In.ucEventGroup    = NC_COMM_EVENT_GROUP6;
    CfgReg3In.ucPDOCommParam= PDO_COM_PARAM_SYNC;

     AxisARef = AxisA.GetRef();

    rc = MMC_CfgRegParamEvPDO3Cmd(ComHndl, AxisARef, &CfgReg3In, &CfgReg3Out);
    if(rc != 0)
    {
     printf("\n **** config error %d \n",CfgReg3Out.sErrorID);
/* ... goto lbl_motor_off; */
    }

     return (rc);
}

/*================================= Example functions END ===========================================*/
```

17.7.3 CMMCSingleAxis Class Functions Code Example 3

```
/*
============================================================================
Collection of Gmas API functions (Set #4)
Examples for document: "G-MAS Administrative and Motion API.pdf".
12Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection     gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCSingleAxis     AxisX, AxisY, AxisZ;
     unsigned short          AxisXRef, AxisYRef, AxisZRef;

     CMMCGroupAxis      Group;
     double             S_Factor_For_Polynomial_Transition;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

     double    StarPoi[][2] =
                             {
                             { 7500.,      0.},
                             {12500.,  13100.},
                             {  200.,   5600.},
                             {14700.,   5600.},
                             { 3340.,  13750.},
                             { 7500.,   7500.}   /* the Center cordination */
                             };

        char   * delimit = "==================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int  WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis2d();
```

```
     void endAdminMultiAxis2d(void);

     void SnroMoveComplex2d(int);
     void MoveLinearRelative(void);
     void MoveCircularLinarAbs(void);
     void MoveCircularAbsolute2d(void);
     void GroupOverrideAndPosition(void);

     int  CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
     int  OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus);

/*======================== Administration functions STR ========================*/

int       main(int)
// ==============
{
     int  trace = 1;

     printf("\n %s", delimit);
     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

     try
     {
          SnroMoveComplex2d(trace++);
     }
     catch (CMMCException excp)
     {
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          exit(0);
     }

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//=====================================================================
{
                int end_of = 0;
                int iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
          end_of = 1;
/* Read Axis Status command server for specific Axis */
          ulState = sng_axis->ReadStatus();
          if (!(ulState & break_state))
          {
               end_of = 0;

               WAIT_SLEEP_MILLI(20)
```

```
        }
    }

//   MMC_SHOWNODESTAT_IN  showin;
//   MMC_SHOWNODESTAT_OUT showout;
//   MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

    return 0;
}


void WaitGrpDone(unsigned int groupStatusMsk)
// =============================================
{
    unsigned int   uiStatusRegister;

    uiStatusRegister = Group.GroupReadStatus();
    while((uiStatusRegister & groupStatusMsk) != groupStatusMsk)
    {
        WAIT_SLEEP_MILLI(2)
        uiStatusRegister = Group.GroupReadStatus();
    }
}


void initAdminSingleAxis(void)
// ==============================
{
    int  iEventMask;

    MMC_MOTIONPARAMS_SINGLE stSingleDefault;

    printf("\n Function: %s: ", __func__);

/* CallbackFunc in ConnectIPCEx call if there is no calling to 'RegisterEventCallback'*/
    iEventMask = 0x7fffffff;
    ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Put Null param Val for no CallbackFunc ComHndl = gConn.ConnectIPCEx(iEventMask, NULL); */
/* Should Not calling, called inside 'ConnectIPCEx'   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl,
&openudp_param_in, &openudp_param_out); */


/* Register Run Time Error Callback function*/
    CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);


    AxisX.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
    stSingleDefault.fEndVelocity  =    0.0;
    stSingleDefault.dbDistance    =    100000.0;
    stSingleDefault.dbPosition    =    0.0;
    stSingleDefault.fVelocity     =    100000.0;
    stSingleDefault.fAcceleration =   2000000.0;
    stSingleDefault.fDeceleration =  10000000.0;
    stSingleDefault.fJerk     = 200000000.0;
/* MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
    stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
    stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
    stSingleDefault.ucExecute     = 1;
```

```
    AxisX.SetDefaultParams(stSingleDefault);
}

//   15.4.4.   SetKinTransform
//   15.3.53.  SetParameter
void initAdminMultiAxis2d()
// ==========================
{
/* Source class:*/
/*    MMC_CONNECT_HNDL    ComHndl;
/*    CMMCSingleAxis AxisX, AxisY;  */
/*    CMMCGroupAxis  Group;         */


    printf("\n Function: %s: ", __func__);

    AxisY.InitAxisData("a02", ComHndl);
    Group.InitAxisData("v01", ComHndl);

    AxisXRef = AxisX.GetRef();
    AxisYRef = AxisY.GetRef();
/* Set by default the EndPoint=StartPoint */
    ParamsGroup.dEndPoint[0] = 0.0;
    ParamsGroup.dEndPoint[1] = 0.0;


    ParamsGroup.fVelocity =   100000.0;
    ParamsGroup.fAcceleration =   8000000.0;
    ParamsGroup.fDeceleration =   8000000.0;
    ParamsGroup.fJerk =    100000000.0;
    ParamsGroup.eCoordSystem = MC_MCS_COORD;
    ParamsGroup.eBufferMode = MC_BUFFERED_MODE;
//    ParamsGroup.eTransitionMode = MC_TM_CORNER_DEVIATION_MODE_PLN6;
    ParamsGroup.eTransitionMode = MC_TM_NONE_MODE;
    ParamsGroup.fTransitionParameter[0] = 2000.0;
    ParamsGroup.ucExecute = 1;

    Group.SetDefaultParams(ParamsGroup);

/* Parameters for Kinematic Transformation */
    SetKin.eBufferMode = MC_BUFFERED_MODE;
    SetKin.eType[0] = NC_X_AXIS_TYPE;
    SetKin.eType[1] = NC_Y_AXIS_TYPE;

    SetKin.hNode[0] = AxisX.GetRef();
    SetKin.hNode[1] = AxisY.GetRef();

    SetKin.iMcsToAcsFuncID[0] = NC_TR_SHIFT_FUNC;
    SetKin.iMcsToAcsFuncID[1] = NC_TR_SHIFT_FUNC;

    SetKin.iNumAxes = 2;
    SetKin.ucExecute = 1;

    SetKin.ulTrCoef[0][0] = 1;
    SetKin.ulTrCoef[0][1] = 1;
    SetKin.ulTrCoef[0][2] = 0;

    SetKin.ulTrCoef[1][0] = 1;
    SetKin.ulTrCoef[1][1] = 1;
    SetKin.ulTrCoef[1][2] = 0;
```

```
     Group.SetKinTransform(SetKin);

/* Set the factor for Polynominal Transition*/
     S_Factor_For_Polynomial_Transition = 0.4;   // The default is 1.4

/* SetParameter(double dbValue, MMC_PARAMETER_LIST_ENUM eNumber, int iIndex); */
     Group.SetParameter(S_Factor_For_Polynomial_Transition, MMC_S_FACTOR, 0);
}


void endAdminSingleAxis(void)
// =============================
{
     printf("\n Function: %s: ", __func__);

     MMC_CloseConnection(ComHndl) ;
}


     void endAdminMultiAxis(void)
// ================================
{
/*  Source class is: CMMCGroupAxis Group;    */

     printf("\n Function: %s: ", __func__);

//The two function below can be called as shown below, or called from the EAS (configuration file) but,
if defined in configuration (EAS show axis in V01 group)
//  they should not be called here! (the axis is already in group...)
//   Group.RemoveAxisFromGroup(NC_NODE_1_ID);
//   Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*===================== Administration functions END ===========================*/


/*======================== Scenario functions STR ==============================*/

void SnroMoveComplex2d(int trace)
// ==============================
{

     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     initAdminMultiAxis2d();

     AxisX.PowerOn(MC_BUFFERED_MODE);
    AxisY.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisX);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisY);

     Group.GroupEnable();

     MoveLinearRelative();
     MoveCircularLinarAbs();
     MoveCircularAbsolute2d();
     GroupOverrideAndPosition();

     Group.GroupDisable();
```

```
    AxisY.PowerOff(MC_BUFFERED_MODE);
    AxisX.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisY);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisX);

    endAdminMultiAxis();
    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

/*===================== Example functions STR ==================================*/

//   15.4.12.  MoveLinearAbsolute
//   15.4.11.  MoveCircularAbsoluteAngle
/* While looking X Y => Drawing 5 points star encircle by circle */
void MoveCircularLinarAbs(void)
// ==============================
{
    printf("\n Function: %s: ", __func__);

    Group.m_fVelocity        =    50000.0;
    Group.m_fAcceleration    =   500000.0;
    Group.m_fDeceleration    =   500000.0;
    Group.m_fJerk            =  1000000.0;

// Move #1
    Group.m_dEndPoint[0] = StarPoi[0][0];
    Group.m_dEndPoint[1] = StarPoi[0][1];
    Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #2
    Group.m_ucExecute = 0;        // Overwrite the default
//  Group.m_ucExecute = 1;        // or not...
    Group.m_dEndPoint[0] = StarPoi[1][0];
    Group.m_dEndPoint[1] = StarPoi[1][1];
    Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #3
    Group.m_ucExecute = 1;        // Set (returned) the default to 1
    Group.m_dEndPoint[0] = StarPoi[2][0];
    Group.m_dEndPoint[1] = StarPoi[2][1];
    Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #4
    Group.m_dEndPoint[0] = StarPoi[3][0];
    Group.m_dEndPoint[1] = StarPoi[3][1];
    Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #5
    Group.m_dEndPoint[0] = StarPoi[4][0];
    Group.m_dEndPoint[1] = StarPoi[4][1];
    Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #6 Back to the started point - end drowing star.
    Group.m_dEndPoint[0] = StarPoi[0][0];
    Group.m_dEndPoint[1] = StarPoi[0][1];
    Group.MoveLinearAbsolute(MC_BLENDING_LOW_MODE);

// Move #7 drowing circle
    Group.m_dAuxPoint[0] = StarPoi[5][0];   // The Center Cordinate
```

```
     Group.m_dAuxPoint[1] = StarPoi[5][1];
     double dAngle = 360;
// Only 2D
     Group.MoveCircularAbsoluteAngle(dAngle, MC_BLENDING_LOW_MODE);
     WaitGrpDone(NC_GROUP_STANDBY_MASK);
}

//   15.4.7.   MoveCircularAbsolute
//   15.4.8.   MoveCircularAbsoluteCenter
//   15.4.9.   MoveCircularAbsoluteBorder
void MoveCircularAbsolute2d(void)
//   ================================
{
     int                          rt_val;
     NC_ARC_SHORT_LONG_ENUM   eArcShortLong;
     NC_PATH_CHOICE_ENUM      ePathChoice;
     NC_CIRC_MODE_ENUM        eCircleMode;
     double                       dAuxPoint[NC_MAX_NUM_AXES_IN_NODE];
     double                       dCenterPoint[NC_MAX_NUM_AXES_IN_NODE];
     double                       dBorderPoint[NC_MAX_NUM_AXES_IN_NODE];
     MC_BUFFERED_MODE_ENUM    eBufferMode;

     printf("\n Function: %s: ", __func__);

     ParamsGroup.eTransitionMode = MC_TM_NONE_MODE;
     Group.SetDefaultParams(ParamsGroup);

     ParamsGroup.fVelocity = 100000.0;
     Group.m_dEndPoint[0] = 0.0;      // Start point of MoveCircularAbs is End point of
MoveLinearAbs...
     Group.m_dEndPoint[1] = 0.0;
     Group.MoveLinearAbsolute(MC_ABORTING_MODE);

     eArcShortLong = MC_NONE_ARC_CHOICE;     // MC_LONG;
     ePathChoice = MC_NONE_PATH_CHOICE;// MC_CLOCKWISE  MC_COUNTERCLOCKWISE   MC_CLOCKWISE;
     eCircleMode = MC_BORDER_CIRC_MODE;
     eBufferMode = MC_BUFFERED_MODE;         // MC_ABORTING_MODE = the defalut

     Group.m_dAuxPoint[0] = 10000.0;         // Point on border
     Group.m_dAuxPoint[1] = 10000.0;
     Group.m_dEndPoint[0] = 10000.0;         // End point
     Group.m_dEndPoint[1] =-10000.0;
     rt_val = Group.MoveCircularAbsolute(eArcShortLong, ePathChoice, eCircleMode, eBufferMode);

     dAuxPoint[0] =  15000.0;           // Circular Center point
     dAuxPoint[1] = -10000.0;
     Group.m_dEndPoint[0] = 15000.0;         // Circular End point, Circular start point = (end of last
motion) 10000,-10000
     Group.m_dEndPoint[1] = -5000.0;
     eCircleMode = MC_CENTER_CIRC_MODE;
     eArcShortLong = MC_LONG;
     rt_val = Group.MoveCircularAbsolute(eArcShortLong, ePathChoice, eCircleMode, dAuxPoint,
eBufferMode);

     Group.m_dAuxPoint[0] = 20000.0;         // Circular center
     Group.m_dAuxPoint[1] = -5000.0;
     Group.m_dEndPoint[0] = 20000.0;         // Circular end point
     Group.m_dEndPoint[1] =     0.0;
     rt_val = Group.MoveCircularAbsoluteCenter(eArcShortLong, eBufferMode);

     Group.m_dEndPoint[0] = 10000.0;         // Start point of MoveCircularAbs... (End point of
MoveLinearAbs)
```

```
    Group.m_dEndPoint[1] =  6000.0;
    Group.MoveLinearAbsolute(MC_BUFFERED_MODE);
    eArcShortLong = MC_LONG;
    dCenterPoint[0] = 10000.0;        // Circular center
    dCenterPoint[1] =  8000.0;
    Group.m_dEndPoint[0] = 12000.0;        // Circular end point
    Group.m_dEndPoint[1] =  8000.0;
    eBufferMode = MC_BUFFERED_MODE;
    rt_val = Group.MoveCircularAbsoluteCenter(eArcShortLong, dCenterPoint, eBufferMode);

// Start point, end of previous move (12000, 8000)
    dBorderPoint[0] = 15000.0;        // Circular border (end of previous).
    dBorderPoint[1] = 11000.0;
    Group.m_dEndPoint[0] = 15000.0;        // Circular end point
    Group.m_dEndPoint[1] =  5000.0;
    rt_val = Group.MoveCircularAbsoluteBorder(dBorderPoint, eBufferMode);

    Group.m_dEndPoint[0] = 10000.0;   // Start point of MoveCircularAbs... (End point of
MoveLinearAbs)
    Group.m_dEndPoint[1] = 0.0;
    Group.MoveLinearAbsolute(MC_BUFFERED_MODE);

    WaitGrpDone(NC_GROUP_STANDBY_MASK);
}


//   15.4.13.MoveLinearRelative
void MoveLinearRelative(void)
// ============================
{
    double        dbDistance[NC_MAX_NUM_AXES_IN_NODE];
    float         fAcceleration,
              fDeceleration,
              fJerk;

    printf("\n Function: %s: ", __func__);

    Group.MoveLinearAbsolute(MC_ABORTING_MODE);
    WaitGrpDone(NC_GROUP_STANDBY_MASK);

    Group.m_dEndPoint[0] = 1000.0;
    Group.m_dEndPoint[1] =  500.0;
    Group.MoveLinearRelative(MC_BUFFERED_MODE);

    WAIT_SLEEP_MILLI(10);
/*...broke previous straight line when looking x relative to y*/
    Group.m_dEndPoint[0] =  250.0;
    Group.MoveLinearRelative(MC_BUFFERED_MODE);

    WAIT_SLEEP_MILLI(10);

/*...broke previous straight line when looking x relative to y*/
    Group.m_dEndPoint[1] =  250.0;
    Group.m_eTransitionMode = MC_TM_DEFINED_VELOCITY_MODE;
    Group.MoveLinearRelative(MC_BLENDING_PREVIOUS_MODE);

    WAIT_SLEEP_MILLI(10);

    Group.m_dEndPoint[0] =  800.0;
    Group.MoveLinearRelative(MC_BLENDING_PREVIOUS_MODE);
```

```
     WAIT_SLEEP_MILLI(10);

     Group.m_dEndPoint[1] =  1000.0;
     Group.MoveLinearRelative(50000.0, MC_BLENDING_PREVIOUS_MODE);

     WAIT_SLEEP_MILLI(10);

     Group.m_dEndPoint[0] =  1000.0;
     Group.MoveLinearRelative(75000.0, MC_BLENDING_PREVIOUS_MODE);

     dbDistance[0] = 1000.0;
     dbDistance[1] = 1500.0;
     Group.MoveLinearRelative(100000.0, dbDistance, MC_BLENDING_PREVIOUS_MODE);
     WaitGrpDone(NC_GROUP_STANDBY_MASK);

     dbDistance[1] = 100.0;
     Group.MoveLinearRelative( 50000.0, dbDistance, MC_BLENDING_PREVIOUS_MODE);
     WaitGrpDone(NC_GROUP_STANDBY_MASK);

     fAcceleration = 20000000.0;
     fDeceleration = fAcceleration;
     dbDistance[0] = 100.0;
     Group.MoveLinearRelative(125000.0, dbDistance, fAcceleration, fDeceleration,
MC_BLENDING_PREVIOUS_MODE);
     Group.MoveLinearRelative(250000.0, dbDistance, fAcceleration, fDeceleration,
MC_BLENDING_PREVIOUS_MODE);
     WaitGrpDone(NC_GROUP_STANDBY_MASK);

     fJerk = 200000000.0;
     dbDistance[0] = 1000.0;
     Group.MoveLinearRelative(150000.0, dbDistance, fAcceleration, fDeceleration, fJerk,
MC_BLENDING_PREVIOUS_MODE);
     Group.MoveLinearRelative(200000.0, dbDistance, fAcceleration, fDeceleration, fJerk,
MC_BLENDING_PREVIOUS_MODE);
     WaitGrpDone(NC_GROUP_STANDBY_MASK);
}

//   15.4.14.GroupSetOverride
//   15.4.15.GroupSetPosition
 void     GroupOverrideAndPosition(void)
// ===================================
{
     double         dbPosition[3];
     float          fVelFactor,
               fAccFactor,
               fJerkFactor;

        float       fVelocity,
               fAcceleration,
               fDeceleration,
               fJerk;

     unsigned short usUpdateVelFactorIdx;
     int       ind,
               rt_val;

     printf("\n Function: %s: ", __func__);

//   GroupSetPosition is Not release yet... - Not support
//
//   MC_COORD_SYSTEM_ENUM     eCoordSystem;
//   unsigned char       ucMode;
```

```
//
//   Group.m_dEndPoint[0] = 1000000.0;
//   Group.m_dEndPoint[1] = 2000000.0;
//   Group.m_dEndPoint[2] = 0.0;
//   ParamsGroup.eTransitionMode = MC_TM_NONE_MODE;
//   Group.SetDefaultParams(ParamsGroup);
//   Group.MoveLinearAbsolute(MC_ABORTING_MODE);

//   dbPosition[0] =     500000.0;
//   dbPosition[1] =     750000.0;
//   dbPosition[2] =     0.0;
//   eCoordSystem = MC_MCS_COORD;
/* RELATIVE =True, ABSOLUTE = False (Default) */
//   ucMode = ABSOLUTE;
/* MC_ABORTING_MODE is the defalut mode */
//   Group.GroupSetPosition(dbPosition, eCoordSystem, ucMode);

    fVelocity = 100000.0;
     fAcceleration = fDeceleration = 10000000;
     fJerk = 100000000.0;
     usUpdateVelFactorIdx = 0; /* Meanwhile only 0 is support */
     dbPosition[2] = 0.0;

     for (ind=0; ind<6; ind++)
     {
          if (ind==2)
          {
          WaitGrpDone(NC_GROUP_STANDBY_MASK);
          fVelFactor = 0.5;
          fAccFactor = 0.5;
          fJerkFactor= 0.5;
          rt_val = Group.GroupSetOverride(fVelFactor, fAccFactor, fJerkFactor, usUpdateVelFactorIdx);
          }

          if (ind==4)
          {
          WaitGrpDone(NC_GROUP_STANDBY_MASK);
          fVelFactor = 1.0;
          fAccFactor = 1.0;
          fJerkFactor= 1.0;
          rt_val = Group.GroupSetOverride(fVelFactor, fAccFactor, fJerkFactor, usUpdateVelFactorIdx);
          }

          dbPosition[0] = 0.0;
          dbPosition[1] = 0.0;
          Group.MoveLinearAbsolute(fVelocity, dbPosition, fAcceleration, fDeceleration, fJerk,
MC_BUFFERED_MODE);
          dbPosition[0] = 100000.0;
          dbPosition[1] = 100000.0;
          Group.MoveLinearAbsolute(fVelocity, dbPosition, fAcceleration, fDeceleration, fJerk,
MC_BUFFERED_MODE);
     }

     WaitGrpDone(NC_GROUP_STANDBY_MASK);
}

int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ===============================================================================
{
    printf("\n *********** STR Func: %s *********** ", __func__);
```

```
/* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
          printf("\n Underflow event received ");
          break ;
     case MODBUS_WRITE_EVT:
          printf("\n ModBus Write event received ");
          break ;
     case TOUCH_PROBE_ENDED_EVT:
          printf("\n Touch Probe event received ");
          break ;
     default:
          printf("\n Default.... Whatever arrived event received ");
          break;
     }

    printf("\n *********** END Func: %s *********** ", __func__);
     fflush(stdout); fflush(stderr);

     return 1 ;
}

int  OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
// ===================================================================================
{
     printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
     msg, usAxisRef, sErrorID, usStatus);
     fflush(stdout); fflush(stderr);
     MMC_CloseConnection(uiConnHndl);
     exit(0);
}

/*====================== Example functions END ===============================*/

/*====================== Output STR ==========================================*/
```

```
#ifdef PROGRAM_OUTPUT
Output example EAS movment record:

#endif /* PROGRAM_OUTPUT */
/*========================= Output END ========================================*/
```

17.7.4 SetDefaultParams

This function initiates default parameters for motion for a specific axis and sets the single axis' default parameters, overwriting the class default parameters.

void SetDefaultParams(
const MMC_MOTIONPARAMS_SINGLE& stSingleAxisParams
);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

stSingleAxisParams

stSingleAxisParams is the structure of motion parameters for a single axis. stSingleAxisParams references the structure MMC_MOTIONPARAMS_SINGLE with default parameters, and either returns none, or throws CMMCException on failure.

MMC_MOTIONPARAMS_SINGLE Structure

typedef struct{
double dbPosition;
double dbDistance;
float fEndVelocity;
float fVelocity;
float fAcceleration;
float fDeceleration;
float fJerk;
MC_DIRECTION_ENUM eDirection ;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned char ucExecute;
unsigned int uiExecDelayMs;
}MMC_MOTIONPARAMS_SINGLE;

Parameters

All parameters

Refer to the **MMC_MOVEABSOLUTE_IN structure on page 219** and **MMC_MOVEABSOLUTEREPETITIVE_IN** structure **on page 247** for details of the parameters.

17.7.4.2 Functions Code Example

```
// 16.3.1. void CMMCSingleAxis::SetDefaultParams(const MMC_MOTIONPARAMS_SINGLE&
stSingleParams)     (- no corresponding MMC_ C func)
void  initAdminSingleAxis(void)
// ==============================
{
 int iEventMask;

 MMC_MOTIONPARAMS_SINGLE stSingleDefault;
```

```
    /* Init default Gmas Parameters       */
 stSingleDefault.fEndVelocity  = 0;
 stSingleDefault.dbDistance    = 100000;
 stSingleDefault.dbPosition    = 0;
 stSingleDefault.fVelocity     = 100000;
 stSingleDefault.fAcceleration = 2000000;
 stSingleDefault.fDeceleration = 10000000;
 stSingleDefault.fJerk     = 200000000;
    /*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
    /* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
 stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
 stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
 stSingleDefault.ucExecute     = 1;
       /* CallbackFunc in ConnectIPCEx call if there   */
      /* is no calling to 'RegisterEventCallback'      */
 iEventMask = 0x7fffffff;
 ComHndl = cConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
      /* Should Not calling, called inside 'ConnectIPCEx'   */
/* rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */


 AxisA.InitAxisData("a01", ComHndl);
 AxisA.SetDefaultParams(stSingleDefault);
}


void  initAdminMultiAxis()
// =========================
{

 AxisB.InitAxisData("a02", ComHndl);
 Group.InitAxisData("v01", ComHndl);

 AxisARef = AxisA.GetRef();
 AxisBRef = AxisB.GetRef();

 Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void  endAdminSingle(void)
// =========================
{
 MMC_CloseConnection(ComHndl) ;
}


 void  endAdminMultiAxis(void)
// ================================
{

 Group.RemoveAxisFromGroup(NC_NODE_1_ID);
 Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}

/*========= Administration functions END =============================================*/


/*================ Scenario functions STR =============================================*/
void  SnroMoveCombinations(int trace)
// ===================================
{
 printf("%s%s -%d- ", strStrSnro, __func__, trace);

 initAdminSingleAxis();
```

```
 AxisA.PowerOn(MC_BUFFERED_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

 MoveCombinations();

 AxisA.PowerOff(MC_BUFFERED_MODE);
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
 endAdminSingle();

 printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);

}

void  SnroEnableDisableMotionEndedEvent(int trace)
// ==================================================
{
 printf("%s%s -%d- ", strStrSnro, __func__, trace);

 initAdminSingleAxis();
 cConn.RegisterEventCallback(MMCPP_MOTIONENDED,  (void* )EndMotionEventCB);

 AxisA.PowerOn(MC_BUFFERED_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

 EnableDisableMotionEndedEvent();

 AxisA.PowerOff(MC_BUFFERED_MODE);
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
 endAdminSingle();

 cConn.RegisterEventCallback(MMCPP_MOTIONENDED,  NULL);

 printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void    SnroSetGetParameters(int trace)
// ====================================
{
 printf("%s%s -%d- ", strStrSnro, __func__, trace);

 initAdminSingleAxis();
 initAdminMultiAxis();

       /* Before power on !*/
 SetGetDrvParameters();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

 Group.GroupEnable();

 SetGetParameters();
 SetGetGroupParam();
 SetDefManufact();

 Group.GroupDisable();

 AxisB.PowerOff(MC_BUFFERED_MODE);
 AxisA.PowerOff(MC_BUFFERED_MODE);
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

 endAdminMultiAxis();
 endAdminSingle();
```

```
 printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void  SnroDepthName(int trace)
// ============================
{
 printf("%s%s -%d- ", strStrSnro, __func__, trace);

 initAdminSingleAxis();
 initAdminMultiAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

 Group.GroupEnable();

 DepthName();

 Group.GroupDisable();

 AxisB.PowerOff(MC_BUFFERED_MODE);
 AxisA.PowerOff(MC_BUFFERED_MODE);
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

 endAdminMultiAxis();
 endAdminSingle();

 printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void  SnroSetDefParamHome(int trace)
// ==================================
{
 printf("%s%s -%d- ", strStrSnro, __func__, trace);

 initAdminSingleAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

 SetDefParamsHome();

 AxisA.PowerOff(MC_BUFFERED_MODE);
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

 endAdminSingle();

 printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void  SnroMoveAbsolute(int trace)
// ===============================
{
 printf("%s%s -%d- ", strStrSnro, __func__, trace);

 initAdminSingleAxis();

 AxisA.PowerOn(MC_BUFFERED_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

 MoveAbsoluteMoves();
```

```
 AxisA.PowerOff(MC_BUFFERED_MODE);
 WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

 endAdminSingle();

 printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
/*============ Scenario functions END ============================================*/
```

17.7.5 SetDefaultHomeDS402Params

Sets the default home (DS-402) parameters, and overwrites the class default parameters.

```
void SetDefaultHomeDS402Params(
const MMC_HOMEDS402_IN& stSingleParams
);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

stSingleAxisParams

stSingleAxisParams references the structure MMC_MOTIONPARAMS_SINGLE with default parameters, and either returns none, or throws CMMCException on failure.

MMC_HOMEDS402_IN Structure

```
typedef struct{
double dbPosition;
float fAcceleration;
float fVelocity;
float fDistanceLimit;
float fTorqueLimit;
MC_BUFFERED_MODE_ENUM eBufferMode;
unsigned int uiHomingMethod;
unsigned int uiTimeLimit;
unsigned char ucExecute;
}MMC_HOMEDS402_IN;
```

Parameters

All parameters

Refer to the **MMC_HOMEDS402_IN Structure** for details of the parameters.

Refer to section **17.7.8.1** for the function example.

17.7.6 SetDefaultHomeParams

Sets the default home parameters, and overwrites the class default parameters.

void SetDefaultHomeParams(
const MMC_HOME_IN& stSingleParams
);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

MMC_HOME_IN& stSingleParams

stSingleParams references the structure MMC_HOME_IN with default parameters, and either returns none, or throws CMMCException on failure.

Refer to the section describing the function **MMC_HOME_IN Structure on page 198**.

Refer to section **17.7.8.1** for the function example.

17.7.7 Home

Refer to the section 5.8.2 **MMC_Home on page 197** for details of the description, scope, and motion mode.

```
unsigned short Home(
[MMC_HOME_IN stHomeParams]
short* sErrorID,
unsigned int* uiHndl
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

MMC_HOME_IN stHomeParams

Refer to the section describing the function **MMC_HOME_IN Structure on page 198**.

sErrorID

Returned command error ID as -ve or +ve integers. Signals where an error has occurred within function block:

- MC_TimeLimitExceeded
- MC_DistanceLimitExceeded
- MC_TorqueLimitExceeded

Refer to the errors listed in sections 4.3 **Maestro Error IDs,** and 4.9 **NC Profiler Error IDs on page 57- 108.**

uiHndl

Returned function block handle. Any +ve value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.8 HomeDS402

Refer to the section 5.8.3 **MMC_HomeDS402 on page 213** for details of the description, scope, and motion mode.

void HomeDS402(
[MMC_HOMEDS402_IN stHomeDS402Params]
) throw (CMMCException)

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

MMC_HOMEDS402_IN stHomeDS402Params

Refer to function parameter MMC_HOMEDS402_IN in section **MMC_HOMEDS402_IN Structure on page 213.**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.8.1 Functions Code Example

```
// 16.3.2. void CMMCSingleAxis::SetDefaultHomeDS402Params(const MMC_HOMEDS402_IN&
stSingleParams)  (- no corresponding MMC_ C func)
// 16.3.3. void CMMCSingleAxis::SetDefaultHomeParams(const MMC_HOME_IN& stSingleParams)   (-
no corresponding MMC_ C func)
//
// 16.3.5. void CMMCSingleAxis::HomeDS402()   5.6.3. MMC_HomeDS402
// 16.3.5. void CMMCSingleAxis::HomeDS402(MMC_HOMEDS402_IN stHomeDS402Params) 5.6.3.
MMC_HomeDS402
//
// 16.3.16. SetOpMode int CMMCSingleAxis::SetOpMode(OPM402 eMode) 11.5.9. MMC_ChngOpMode
// 16.3.17. GetOpMode OPM402 CMMCSingleAxis::GetOpMode()      MMC_ReadBoolParameter
 void SetDefParamsHome(void)
//  ==============================
{
 MMC_HOMEDS402_IN  stDS402Home ;
 MMC_HOME_IN   stSingleParams;
 OPM402     drvMode;
 short      sErrorID;

printf("\n     %s:", __func__);
 stDS402Home.dbPosition    = -1000000 ;
 stDS402Home.eBufferMode   = MC_BUFFERED_MODE;
 stDS402Home.fAcceleration      = 100000;
 stDS402Home.fDistanceLimit     = 100000;
 stDS402Home.fTorqueLimit = 1;
 stDS402Home.fVelocity     = 100000;
 stDS402Home.uiHomingMethod     = 35; // Homing method immediate
 stDS402Home.uiTimeLimit   = 100000;
 stDS402Home.ucExecute     = 1;
 AxisA.SetDefaultHomeDS402Params(stDS402Home);
```

```
stSingleParams.dbPosition     = 2000000;
stSingleParams.eBufferMode    = MC_BUFFERED_MODE; /* MC_ABORTING_MODE; */
stSingleParams.fAcceleration = 200000;
stSingleParams.fDistanceLimit= 400000;
stSingleParams.fTorqueLimit  = 1;
stSingleParams.fVelocity     = 200000;
stSingleParams.ucExecute     = 1;
stSingleParams.uiTimeLimit   = 50000;
stSingleParams.eHomingMode   = MC_DIRECT;
stSingleParams.eDirection    = MC_POSITIVE;
stSingleParams.eSwitchMode    = MC_ON;
AxisA.SetDefaultHomeParams(stSingleParams);

drvMode = AxisA.GetOpMode();
        /* Cannot be done on Virtual Axis */
AxisA.SetOpMode(OPM402_HOMING_MODE);


stDS402Home.dbPosition   = 3000000;
stDS402Home.fAcceleration     =  300000;
stDS402Home.fVelocity     =  300000;
stDS402Home.fDistanceLimit    =  300000;
stDS402Home.uiTimeLimit  =     30000;

AxisA.HomeDS402(stDS402Home);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

AxisA.HomeDS402();
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    /* Retrive the keeping mode */
AxisA.SetOpMode(drvMode);

AxisA.PowerOff(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

initAdminMultiAxis();
   AxisA.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

   AxisB.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

Group.GroupEnable();

SetGrpKinDef();

Group.GroupDisable();
AxisB.PowerOff(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);

endAdminMultiAxis();
}
```

17.7.9 MoveAbsolute

Refer to the section 5.8.4 **MMC_MoveAbsolute on page 218** for details of the description, scope, and motion mode.

```
int MoveAbsolute(
double dPos,
[float fVel],
[float fAcceleration],
[float fDeceleration],
[float fJerk],
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbPos

Target position for the motion. Any -ve or +ve double values in technical unit [u]

fVel

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s.

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

eBufferMode

Refer to the section **MMC_MOVEABSOLUTE_IN Structure**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.9.1 Function Code Examples

Example 1

```
{
CMMCConnection      MyConn;
CMMCSingleAxis      cScanAxis;
MMC_MOTIONPARAMS_SINGLE stSingleDefault;
//
unsigned    int      conn_hndl = MyConn.ConnectIPCEx(0x7fffffff, NULL);
//
    /* Init default Gmas Parameters       */
stSingleDefault.fEndVelocity  = 0;
stSingleDefault.dbDistance    = 100000;
stSingleDefault.dbPosition    = 0;
stSingleDefault.fVelocity     = 100000;
stSingleDefault.fAcceleration = 1000000;
stSingleDefault.fDeceleration = 1000000;
stSingleDefault.fJerk     = 20000000;
    /*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
    /* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
stSingleDefault.ucExecute     = 1;
//
cScanAxis.InitAxisData("a01", conn_hndl);
cScanAxis.SetDefaultParams(stSingleDefault);
//
ScanAxis.PowerOn(MC_BUFFERED_MODE);
    /* Move to -100000 at default speed:   */
ScanAxis.MoveAbsolute(-100000.0);
    /* Move to -200000 at  speed 250000.0: */
ScanAxis.MoveAbsolute(-200000.0, 250000.0);
//
    /* Change the default parameters  */
cScanAxis.m_fAcceleration = 10000000.0;
cScanAxis.m_fDeceleration = 40000.0;
cScanAxis.m_fVelocity = 10000.0;
//
    /* Move to -300000 at default velocity */
ScanAxis.MoveAbsolute(-300000.0);
    /* Move to 310000 at velocity 10000.0  */
ScanAxis.MoveAbsolute(310000.0, 10000.0);
    /* Move: Pos=400000, speed 20000, Acc=1000000  */
    /* Dec=1500000, Jerk=20000000, buffer mode=    */
    /* MC_BUFFERED_MODE         */
ScanAxis.MoveAbsolute(400000, 20000, 1000000, 1500000 , 20000000);
    /* Move abs to 350000 with default parameters  */
ScanAxis.MoveAbsolute(350000);
}
```

Example 2

```
/* 16.3.6. MoveAbsolute (5.6.4. MMC_MoveAbsolute) *\
\* ============================================== */
void  MoveAbsoluteMoves(void)
// ============================
{
printf("\n     %s:", __func__);
    /* Move to -400000 at default speed: */
 AxisA.MoveAbsolute(-40000.0);
    /* Move to -200000 at  speed 5000000.0    */
 AxisA.MoveAbsolute(-200000.0, 5000000.0);
    /* Change the default parameters      */
 AxisA.m_fAcceleration = 1000000.0;
 AxisA.m_fDeceleration = 5000000.0;
 AxisA.m_fVelocity = 100000.0;
    /* Move to -300000 at default velocity */
 AxisA.MoveAbsolute(-300000.0);
    /* Move to 310000 at velocity 80000.0  */
 AxisA.MoveAbsolute(310000.0, 80000.0);
```

```
    /* Move: Pos=400000, speed 500000, Acc=100000   */
    /* Dec=1500000, Jerk=20000000, buffer mode=     */
    /* MC_BUFFERED_MODE          */
 AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
    /* Move abs to 35000 with default parameters    */
 AxisA.MoveAbsolute(350000);
}


int    CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock)
// ===============================================================================
{
    printf("\n ******************** STR %s ************************* ", __func__);
    printf("\n >>>>> UDP connection: recvBuffer=<%s> recvBufferSize=%d ", recvBuffer,
recvBufferSize);

      /* Which function ID was received ... */
 switch(recvBuffer[1])
 {
 case ASYNC_REPLY_EVT:
  printf("\n ASYNC event Reply ");
  break ;
 case EMCY_EVT:
  printf("\n Emergency Event received ");
  break ;
 case MOTIONENDED_EVT:
  printf("\n Motion Ended Event received ");
  break ;
 case HBEAT_EVT:
  printf("\n H Beat Fail Event received ");
  break ;
 case PDORCV_EVT:
  printf("\n PDO Received Event received - Updating Inputs ");
  break ;
 case DRVERROR_EVT:
  printf("\n Drive Error Received Event received ");
  break ;
 case HOME_ENDED_EVT:
  printf("\n Home Ended Event received ");
  break ;
 case SYSTEMERROR_EVT:
  printf("\n System Error Event received ");
 case TABLE_UNDERFLOW_EVT:
  printf("\n Underflow event received ");
  break ;
 case MODBUS_WRITE_EVT:
  printf("\n ModBus Write event received ");
  break ;
 case TOUCH_PROBE_ENDED_EVT:
  printf("\n Touch Probe event received ");
  break ;
 default:
  printf("\n Default.... Whatever arrived event received ");
  break;
 }

    printf("\n ******************** END %s ************************* ", __func__);
 fflush(stdout); fflush(stderr);

 return 1 ;
}


int    OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus)
// =====================================================================================
{
```

```
 printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d, 
bye\n",
     msg, usAxisRef, sErrorID, usStatus);
 fflush(stdout); fflush(stderr);

 MMC_CloseConnection(uiConnHndl);
 exit(0);
}


void EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
 printf("\n\t\t %s: Received usAxisRef=%d ", __func__, (int)usAxisRef);
 printf("\n\t\t %s: %s \n", __func__, EndMotionEventCB_MESSAGE);
 fflush(stdout); fflush(stderr);
}

     /* Callback Function once a Modbus message is received. */
void  ModbusWrite_Received()
// ===========================
{
 printf("\n %s Received ", __func__) ;
 fflush(stdout); fflush(stderr);
}

     /* Callback Function once an Emergency is received. */
void  Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ===================================================================
{
 printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
 fflush(stdout); fflush(stderr);
}
/*================ Example functions END ==============================================*/
```

17.7.10 MoveAdditive

Refer to the section 5.8.5 **MMC_MoveAdditive on page 224** for details of the description, scope, and motion mode.

```
int MoveAdditive(
double dbDistance,
[float fVel],
[float fAcceleration],
[float fDeceleration],
[float fJerk],
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbDistance

Target distancefor the motion. Any -ve or +ve double values in technical unit [u]

fVel

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s.

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

eBufferMode

Refer to **MMC_MOVEADDITIVE_IN Structure on page 225**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.10.1 Function Code Example

```
/*============== Example functions STR =============================================*/
void  MoveCombinations(void)
// ===========================
{
printf("\n Group of function: %s", __func__);
  MoveAdditiveMoves();
  MoveRelativeVelMoves();

  MoveAbsRepetiveMoves();
  MoveAdditiveRepetiveMoves();
  MoveRelativeRepetiveMoves();
}

//    16.3.7.     MoveAdditive     5.6.5. MMC_MoveAdditiveCmd
//    int CMMCSingleAxis::MoveAdditive(double dbDistance, MC_BUFFERED_MODE_ENUM eBufferMode)
//    int CMMCSingleAxis::MoveAdditive(double dbDistance, float fVel, MC_BUFFERED_MODE_ENUM
eBufferMode)
//    int CMMCSingleAxis::MoveAdditive(double dbDistance, float fVel, float fAcceleration,
float fDeceleration, MC_BUFFERED_MODE_ENUM eBufferMode)
//    int CMMCSingleAxis::MoveAdditive(double dbDistance, float fVel, float fAcceleration,
float fDeceleration, float fJerk, MC_BUFFERED_MODE_ENUM eBufferMode)
void  MoveAdditiveMoves(void)
// ============================
{
 MC_BUFFERED_MODE_ENUM eBufferMode;
 double dbDistance;
 float  fVel,
   fAcceleration,
   fDeceleration,
   fJerk;

printf("\n     %s:", __func__);
    /* Move: Pos=0, speed 500000, Acc=1000000   */
    /* Dec=1500000, Jerk=20000000, buffer mode=    */
    /* MC_BUFFERED_MODE (default)    */
 AxisA.MoveAbsolute(0.0, 500000, 1000000, 1500000 , 20000000);

     /* Distance additional to the most recent commanded position */
 dbDistance =  100000.0;
 eBufferMode = MC_BUFFERED_MODE;
 AxisA.MoveAdditive(dbDistance, eBufferMode);
 fVel = 200000.0;
 AxisA.MoveAdditive(dbDistance, fVel, eBufferMode);
 fAcceleration = 400000.0;
 fDeceleration = fAcceleration;
 eBufferMode = MC_BLENDING_LOW_MODE;
 AxisA.MoveAdditive(dbDistance, fVel, fAcceleration, fDeceleration, eBufferMode);
 fJerk = 800000000.0;
 AxisA.MoveAdditive(dbDistance, fVel, fAcceleration, fDeceleration, fJerk, eBufferMode);
 AxisA.MoveAdditive(dbDistance, eBufferMode);
 AxisA.MoveAdditive(dbDistance, eBufferMode);
}
```

17.7.11 MoveRelative

Refer to the section 5.8.6 **MMC_MoveRelative on page 230** for details of the description, scope, and motion mode.

int MoveRelative(
double dbDistance,
[float fVel],
[float fAcceleration],
[float fDeceleration],
[float fJerk],
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbDistance

Target distancefor the motion. Any -ve or +ve double values in technical unit [u]

fVel

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s.

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

eBufferMode

Refer to **MMC_MOVERELATIVE_IN Structure on page 231**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.11.1 Function Code Example

```
//    16.3.8.     MoveRelative     5.6.6. MMC_MoveRelativeCmd
//  int CMMCSingleAxis::MoveRelative(double dbDistance, MC_BUFFERED_MODE_ENUM eBufferMode)
//  int CMMCSingleAxis::MoveRelative(double dbDistance, float fVel, MC_BUFFERED_MODE_ENUM
eBufferMode)
//  int CMMCSingleAxis::MoveRelative(double dbDistance, float fVel,  float fAcceleration,
float fDeceleration, MC_BUFFERED_MODE_ENUM eBufferMode)
//  int CMMCSingleAxis::MoveRelative(double dbDistance, float fVel,  float fAcceleration,
float fDeceleration, float fJerk, MC_BUFFERED_MODE_ENUM eBufferMode)

//    16.3.9.     MoveVelocity     5.6.7. MMC_MoveVelocityCmd
//    int CMMCSingleAxis::MoveVelocity(float fVelocity, MC_BUFFERED_MODE_ENUM eBufferMode)
throw (CMMCException)
void  MoveRelativeVelMoves(void)
// ================================
{
 MC_BUFFERED_MODE_ENUM eBufferMode;
 OPM402 drvMode;
 double dbDistance;
 float  fVel,
   fAcceleration,
   fDeceleration,
   fJerk;


printf("\n     %s:", __func__);
 dbDistance =  100000.0;
 eBufferMode = MC_BUFFERED_MODE;
      /* Distance relative to the set position at the time of the execution */
 AxisA.MoveRelative(dbDistance,
        eBufferMode);
 fVel = 400000.0;
 AxisA.MoveRelative(dbDistance, fVel,
        eBufferMode);
 fAcceleration = 800000.0;
 fDeceleration = fAcceleration;
 AxisA.MoveRelative(dbDistance, fVel, fAcceleration, fDeceleration,
        eBufferMode);
 fJerk = 400000.0;
 AxisA.MoveRelative(dbDistance, fVel, fAcceleration, fDeceleration, fJerk,
        eBufferMode);

 drvMode = AxisA.GetOpMode();
      /* Set sutiable mode for MoveVelocity */
 AxisA.SetOpMode(OPM402_INTERPOLATED_POSITION_MODE);
 fVel = 100000.0;
 AxisA.MoveVelocity(fVel, eBufferMode);
      /* Sleep for 2 Sec */
 WAIT_SLEEP_MILLI(2000)

 eBufferMode = MC_ABORTING_MODE;
 AxisA.Stop(100000000.0, 1000000000.0,eBufferMode);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
      /* Retrive the keeping mode */
 AxisA.SetOpMode(drvMode);
}
```

17.7.12 MoveVelocity

Refer to the section 5.8.7 **MMC_MoveVelocity on page 236** for details of the description, scope, and motion mode.

```
int MoveVelocity(
float fVelocity,
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

fVelocity

Value of the maximum velocity (not necessarily reached). Any -ve or +ve double values in technical unit [u].

eBufferMode

Refer to **MMC_MOVEVELOCITY_IN Structure on page 237**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.13 MoveAbsoluteRepetitive

Refer to the section **5.8.8 MMC_MoveContinuous** for details of the description, scope, and motion mode.

```
int MoveAbsoluteRepetitive(
double dPos,
[float fVel],
[float fAcceleration],
[float fDeceleration],
[float fJerk],
[unsigned int uiExecDelayMs],
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbPos

Target position for the motion. Any -ve or +ve double values in technical unit [u]

fVel

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s.

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

uiExecDelayMs

The delay in execution of the next action (in msecs). Any +ve integer value.

eBufferMode

Refer to the **MMC_MOVEABSOLUTEREPETITIVE_IN Structure on page 247**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name Axis reference Error ID
Status of the axis.

17.7.13.1 Function Code Example

```
//    16.3.10. MoveAbsoluteRepetitive    5.6.8. MMC_MoveAbsoluteRepetitive (Cmd)
```

```
//  int CMMCSingleAxis::MoveAbsoluteRepetitive(double dPos, MC_BUFFERED_MODE_ENUM eBufferMode)
//  int CMMCSingleAxis::MoveAbsoluteRepetitive(double dPos, float fVel, MC_BUFFERED_MODE_ENUM
eBufferMode)
//  int CMMCSingleAxis::MoveAbsoluteRepetitive(double dPos, float fVel, float fAcceleration,
float fDeceleration, MC_BUFFERED_MODE_ENUM eBufferMode)
//  int CMMCSingleAxis::MoveAbsoluteRepetitive(double dPos, float fVel, float fAcceleration,
float fDeceleration, float fJerk, MC_BUFFERED_MODE_ENUM eBufferMode)
//  int CMMCSingleAxis::MoveAbsoluteRepetitive(double dPos, float fVel, unsigned int
uiExecDelayMs, MC_BUFFERED_MODE_ENUM eBufferMode)
void  MoveAbsRepetiveMoves(void)
// ===============================
{
 MC_BUFFERED_MODE_ENUM eBufferMode;
 double  dbPos,
   minMov;
 float  fVel,
   fAcceleration,
   fDeceleration,
   fJerk;
unsigned int uiExecDelayMs;
int   rtVal;


printf("\n     %s:", __func__);
 AxisA.MoveAbsolute(0.0);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 eBufferMode = MC_BUFFERED_MODE;
 dbPos = 200000.0;
 AxisA.MoveAbsoluteRepetitive(dbPos,  eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
         /* Ensure known starting position  */
 minMov = 100.0;
 AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

 dbPos = 300000.0;
 fVel = 200000.0;
 AxisA.MoveAbsoluteRepetitive(dbPos, fVel, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 rtVal = AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


 dbPos = 400000.0;
 fAcceleration = 800000.0;
 fDeceleration = fAcceleration;
 AxisA.MoveAbsoluteRepetitive(dbPos, fVel, fAcceleration, fDeceleration, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 rtVal = AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


 dbPos = 600000.0;
 fJerk = 400000.0;
 AxisA.MoveAbsoluteRepetitive(dbPos, fVel, fAcceleration, fDeceleration, fJerk, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
```

```
 dbPos = 1000000.0;
 uiExecDelayMs = 10;
 AxisA.MoveAbsoluteRepetitive(dbPos,  fVel, uiExecDelayMs, eBufferMode);
 WAIT_SLEEP_MILLI(10000);
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
}
```

17.7.14 MoveRelativeRepetitive

Refer to the section **5.8.10 MMC_MoveRelativeRepetitive on page 251** for details of the description, scope, and motion mode.

int MoveRelativeRepetitive(
double dPos,
[float fVel],
[float fAcceleration],
[float fDeceleration],
[float fJerk],
[unsigned int uiExecDelayMs]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbPos

Target position for the motion. Any -ve or +ve double values in technical unit [u]

fVel

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s.

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

uiExecDelayMs

The delay in execution of the next action (in msecs). Any +ve integer value.

eBufferMode

Refer to the **MMC_MOVERELATIVEREPETITIVE_IN Structure on page 252**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name Axis reference Error ID
Status of the axis.

17.7.14.1 Function Code Example

```
//    16.3.11. MoveRelativeRepetitive     5.6.9. MMC_MoveRelativeRepetitiveCmd
//  int CMMCSingleAxis::MoveRelativeRepetitive(double dPos,  MC_BUFFERED_MODE_ENUM
eBufferMode)
//  int CMMCSingleAxis::MoveRelativeRepetitive(double dPos, float fVel, MC_BUFFERED_MODE_ENUM
eBufferMode)
//  int CMMCSingleAxis::MoveRelativeRepetitive(double dPos, float fVel, float fAcceleration,
float fDeceleration, MC_BUFFERED_MODE_ENUM eBufferMode)
//  int CMMCSingleAxis::MoveRelativeRepetitive(double dPos, float fVel, float fAcceleration,
float fDeceleration, float fJerk, MC_BUFFERED_MODE_ENUM eBufferMode)
//  int CMMCSingleAxis::MoveRelativeRepetitive(double dPos, float fVel, unsigned int
uiExecDelayMs, MC_BUFFERED_MODE_ENUM eBufferMode)
void  MoveRelativeRepetiveMoves(void)
// ====================================
{
 MC_BUFFERED_MODE_ENUM eBufferMode;
 double  dbPos,
   minMov;
 float  fVel,
   fAcceleration,
   fDeceleration,
   fJerk;
unsigned int uiExecDelayMs;
int    rtVal;


printf("\n     %s:", __func__);
 AxisA.MoveAbsolute(0.0);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 eBufferMode = MC_BUFFERED_MODE;
 dbPos = 200000.0;
    AxisA.MoveRelativeRepetitive(dbPos,  eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
         /* Ensure known starting position  */
 minMov = 100.0;
 AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

 dbPos = 300000.0;
 fVel = 200000.0;
    AxisA.MoveRelativeRepetitive(dbPos, fVel, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 rtVal = AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


 dbPos = 400000.0;
 fAcceleration = 800000.0;
 fDeceleration = fAcceleration;
 AxisA.MoveRelativeRepetitive(dbPos, fVel, fAcceleration, fDeceleration, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 rtVal = AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


 dbPos = 700000.0;
 fJerk = 400000.0;
 AxisA.MoveRelativeRepetitive(dbPos, fVel, fAcceleration, fDeceleration, fJerk, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
```

```
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


 dbPos = 1000000.0;
 uiExecDelayMs = 10;
    AxisA.MoveRelativeRepetitive(dbPos, fVel, uiExecDelayMs, eBufferMode);
 WAIT_SLEEP_MILLI(10000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
}
```

17.7.15 MoveAdditiveRepetitive

Refer to the section **5.8.11 above MMC_MoveAdditiveRepetitive on page 256** for details of the description, scope, and motion mode.

```
int MoveAdditiveRepetitive(
double dPos,
[float fVel],
[float fAcceleration],
[float fDeceleration],
[float fJerk],
[unsigned int uiExecDelayMs]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbPos

Target position for the motion. Any -ve or +ve double values in technical unit [u]

fVel

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s.

fAcceleration

Value of the acceleration (increasing energy of the motor). Any positive float value in $u/s^{2.}$

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Maximum float value of the Jerk. Any positive value in u/s^3

uiExecDelayMs

The delay in execution of the next action (in msecs). Any +ve integer value.

eBufferMode

Refer to the **MMC_MOVEADDITIVEREPETITIVE_IN Structure on page 257**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name Axis reference Error ID
Status of the axis.

17.7.15.1 Function Code Example

```
//    16.3.12. MoveAdditiveRepetitive     5.6.10. MMC_MoveAdditiveRepetitiveCmd
//    int CMMCSingleAxis::MoveAdditiveRepetitive(double dPos, MC_BUFFERED_MODE_ENUM
eBufferMode)
//    int CMMCSingleAxis::MoveAdditiveRepetitive(double dPos, float fVel,
MC_BUFFERED_MODE_ENUM eBufferMode)
//    int CMMCSingleAxis::MoveAdditiveRepetitive(double dPos, float fVel, float fAcceleration,
float fDeceleration, MC_BUFFERED_MODE_ENUM eBufferMode)
//    int CMMCSingleAxis::MoveAdditiveRepetitive(double dPos, float fVel, float fAcceleration,
float fDeceleration, float fJerk, MC_BUFFERED_MODE_ENUM eBufferMode)
//    int CMMCSingleAxis::MoveAdditiveRepetitive(double dPos, float fVel, unsigned int
uiExecDelayMs, MC_BUFFERED_MODE_ENUM eBufferMode)
void  MoveAdditiveRepetiveMoves(void)
// ====================================
{
 MC_BUFFERED_MODE_ENUM eBufferMode;
 double  dbPos,
   minMov;
 float  fVel,
   fAcceleration,
   fDeceleration,
   fJerk;
unsigned int uiExecDelayMs;
int   rtVal;


printf("\n     %s:", __func__);
 AxisA.MoveAbsolute(0.0);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 eBufferMode = MC_BUFFERED_MODE;
 dbPos = 200000.0;
 AxisA.MoveAdditiveRepetitive(dbPos, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
         /* Ensure known starting position  */
 minMov = 100.0;
 AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

 dbPos = 300000.0;
 fVel = 200000.0;
    AxisA.MoveAdditiveRepetitive(dbPos, fVel, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 rtVal = AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


 dbPos = 400000.0;
 fAcceleration = 800000.0;
 fDeceleration = fAcceleration;
 AxisA.MoveAdditiveRepetitive(dbPos, fVel, fAcceleration, fDeceleration, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 rtVal = AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


 dbPos = 600000.0;
 fJerk = 400000.0;
 AxisA.MoveAdditiveRepetitive(dbPos, fVel, fAcceleration, fDeceleration, fJerk, eBufferMode);
 WAIT_SLEEP_MILLI(5000)
```

```
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


 dbPos = 1000000.0;
 uiExecDelayMs = 10;
    AxisA.MoveAdditiveRepetitive(dbPos, fVel, uiExecDelayMs, eBufferMode);
 WAIT_SLEEP_MILLI(10000)
 AxisA.Stop(100000000.0, 1000000000.0,MC_ABORTING_MODE);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
 AxisA.MoveAbsolute(dbPos+minMov);
 WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
}
```

17.7.16 MoveTorque

This function send Move Torque command to MMC server for specific Axis. Refer to the section **5.9.11 MMC_ReadActualTorque** for details of the description, scope, and motion mode.

```
void MoveTorque(
double dbTargetTorque
[double dbTorqeVelocity]
[double dbTorqueAcceleration]
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbTargetTorque

Target torque to move. Any -ve or +ve double values in technical unit [u]

dbTorqeVelocity

The target torque velocity. Any -ve or +ve double values in technical unit [u]

dbTorqueAcceleration

The target torque acceleration. Any -ve or +ve double values in technical unit [u]

MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE

The MC_BUFFERED_MODE_ENUM enumerator defines the behavior of the axis. Modes are as follows, but only the Buffered Mode is supported:

MC_ABORTING_MODE	= 1
MC_BUFFERED_MODE	= 2
MC_BLENDING_LOW_MODE	= 3
MC_BLENDING_PREVIOUS_MODE	= 4
MC_BLENDING_NEXT_MODE	= 5
MC_BLENDING_HIGH_MODE	= 6

Aborting	Default mode without buffering. The next function block aborts an ongoing motion and the command affects the axis immediately. The buffer is cleared
Buffered	The next function block affects the axis as soon as the previous movement is completed.
BlendingLow	The next function block controls the axis after the previous function block has finished (equivalent to buffered), but the axis will not stop between the movements. The velocity is blended with the lowest velocity of both commands (1 and 2) at the first end-position (1).
BlendingPrevious	Blending with the velocity of function block 1 at the end-position

	of this block
BlendingNext	Blending with the velocity of function block 2 at end-position of function block1
BlendingHigh	Blending with highest velocity of function block 1 and function block 2 at end-position of function block1.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Return 0 if success, otherwise error_id in case of error. Produces details of the error including:

Function Name Structure name Axis reference Error ID

Status of the axis.

17.7.17 GetFbDepth

Refer to the sections **5.9.6 MMC_GetFBDepth** and **5.9.7 MMC_GetTotalFbDepth** for details of the description, scope, and motion mode.

unsigned int GetFbDepth(
const unsigned int uiHndl
)throw (CMMCException);

Source GMAS\includes\CPP\MMCAxis.h

.NET Definition

Function Parameters

GetFbDepth

Refer to the section

on page 269 for details of the function.

uiHndl

Returned function block handle. Integer with any +ve value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID

Status of the axis.

Refer to the function example in section **17.6.7.1**

17.7.18 PositionProfile

Refer to the section **5.9.9 MMC_PositionProfile on page 287** for details of the description, scope, and motion mode.

```
int PositionProfile(
MC_PATH_REF hMemHandle,
MC_BUFFERED_MODE_ENUM eBufferMode = MC_BUFFERED_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

hMemHandle

MC_PATH_REF enumerator handle to a journal entry where the pointer to the shared memory is located. MC_PATH_REF is the journal entry path reference.

hMemHandle can have integer values.

fVel

Value of the maximum velocity (not necessarily reached). Any positive float value in u/s.

eBufferMode

Refer to the **MMC_POSITIONPROFILE_IN Structure on page 288.**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.19 TouchProbeDisable

Refer to the section **5.9.27 MMC_TouchProbeDisable on page 344** for details of the description, scope, and motion mode.

int TouchProbeDisable(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.20 TouchProbeEnable

Refer to the section **5.9.26 MMC_TouchProbeEnable on page 341** for details of the description, scope, and motion mode.

int TouchProbeEnable(
unsigned char ucTriggerType
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

ucTriggerType

Trigger enables the touch proble. Reference to the trigger signal source, where the trigger input may be specified by the AXIS_REF. Has the following enumerator values:

eMMC_TOUCHPROBE_POS_EDGE = 0,

eMMC_TOUCHPROBE_NEG_EDGE

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.21 SetOpMode

Changes the motion mode between NC and Distributed. This is previous determined in the DS-402 mode. Refer to the similar function block described in section **16.5.9 MMC_ChngOpMode on page 1277** for details of the description, scope, and motion mode.

```
int SetOpMode(
OPM402 eMode
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

OPM402 eMode

Refer to the section 16.4.2 **CANopen DS-402 Modes of Operation on page 1238** for further details and similar useage of the parameter *ucMotionMode*.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

Refer to section17.7.22.1 for the function example.

17.7.22 GetOpMode

Changes the motion mode between NC and Distributed. This is previous determined in the DS-402 mode. Refer to the similar function block described in section **16.5.9 MMC_ChngOpMode on page 1277** for details of the description, scope, and motion mode.

OPM402 GetOpMode(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.22.1 Functions Code Example

```
// 16.3.2. void CMMCSingleAxis::SetDefaultHomeDS402Params(const MMC_HOMEDS402_IN&
stSingleParams)  (- no corresponding MMC_ C func)
// 16.3.3. void CMMCSingleAxis::SetDefaultHomeParams(const MMC_HOME_IN& stSingleParams)   (-
no corresponding MMC_ C func)
//
// 16.3.5. void CMMCSingleAxis::HomeDS402()   5.6.3. MMC_HomeDS402
// 16.3.5. void CMMCSingleAxis::HomeDS402(MMC_HOMEDS402_IN stHomeDS402Params) 5.6.3.
MMC_HomeDS402
//
// 16.3.16. SetOpMode int CMMCSingleAxis::SetOpMode(OPM402 eMode) 11.5.9. MMC_ChngOpMode
// 16.3.17. GetOpMode OPM402 CMMCSingleAxis::GetOpMode()      MMC_ReadBoolParameter
 void SetDefParamsHome(void)
//  ==============================
{
 MMC_HOMEDS402_IN  stDS402Home ;
 MMC_HOME_IN   stSingleParams;
 OPM402     drvMode;
 short      sErrorID;

printf("\n      %s:", __func__);
 stDS402Home.dbPosition    = -1000000 ;
 stDS402Home.eBufferMode   = MC_BUFFERED_MODE;
 stDS402Home.fAcceleration      = 100000;
 stDS402Home.fDistanceLimit     = 100000;
 stDS402Home.fTorqueLimit = 1;
 stDS402Home.fVelocity     = 100000;
 stDS402Home.uiHomingMethod     = 35; // Homing method immediate
 stDS402Home.uiTimeLimit   = 100000;
 stDS402Home.ucExecute     = 1;
 AxisA.SetDefaultHomeDS402Params(stDS402Home);

 stSingleParams.dbPosition      = 2000000;
 stSingleParams.eBufferMode     = MC_BUFFERED_MODE; /* MC_ABORTING_MODE; */
 stSingleParams.fAcceleration = 200000;
 stSingleParams.fDistanceLimit= 400000;
```

```
stSingleParams.fTorqueLimit  = 1;
stSingleParams.fVelocity     = 200000;
stSingleParams.ucExecute     = 1;
stSingleParams.uiTimeLimit   = 50000;
stSingleParams.eHomingMode   = MC_DIRECT;
stSingleParams.eDirection    = MC_POSITIVE;
stSingleParams.eSwitchMode    = MC_ON;
AxisA.SetDefaultHomeParams(stSingleParams);

drvMode = AxisA.GetOpMode();
        /* Cannot be done on Virtual Axis */
AxisA.SetOpMode(OPM402_HOMING_MODE);


stDS402Home.dbPosition   = 3000000;
stDS402Home.fAcceleration     =  300000;
stDS402Home.fVelocity    =  300000;
stDS402Home.fDistanceLimit    =  300000;
stDS402Home.uiTimeLimit  =     30000;

AxisA.HomeDS402(stDS402Home);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

AxisA.HomeDS402();
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    /* Retrive the keeping mode */
AxisA.SetOpMode(drvMode);

AxisA.PowerOff(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

initAdminMultiAxis();
   AxisA.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

   AxisB.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

Group.GroupEnable();

SetGrpKinDef();

Group.GroupDisable();
AxisB.PowerOff(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);

endAdminMultiAxis();
}
```

17.7.23 PowerOn

Sets the drive to Power On mode. Refer to the similar function block described in section **5.9.8 MMC_Power** for details of the description, scope, and motion mode.

```
void PowerOn(
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

eBufferMode

Refer to the **MMC_POWER_IN Structure on page 284**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.24 PowerOff

Sets the drive to Power Off mode. Refer to the similar function block described in section **5.9.8 MMC_Power** for details of the description, scope, and motion mode.

```
void PowerOff(
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

eBufferMode

Refer to the **MMC_POWER_IN Structure on page 284**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including as for PowerOn:
Function Name Structure name Axis reference
Error ID Status of the axis

17.7.25 GetActualPosition

Refer to the similar function block described in section **5.9.10 MMC_ReadActualPosition on page 290** for details of the description, scope, and motion mode.

```
double GetActualPosition(
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

For code example, refer to the section17.2.1 and **17.7.2.**

17.7.26 GetActualVelocity

Refer to the similar function block described in section **5.9.12 MMC_ReadActualVelocity on page 296** for details of the description, scope, and motion mode.

double GetActualVelocity(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

For code example, refer to the section17.2.1 **17.7.2.**

17.7.27 GetActualTorque

Refer to the similar function block described in section **5.9.11 MMC_ReadActualTorque on page 293** for details of the description, scope, and motion mode.

double GetActualTorque(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section**17.2.1** and **17.7.2.**

17.7.28 Halt

Refer to the section **5.8.1 MMC_Halt** for details of the description, scope, and motion mode.

```
void Halt(
[float fDeceleration],
[float fJerk],
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eBufferMode

Refer to the **MMC_HALT_IN Structure on page 194**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name

Axis reference Error ID Status of the axis.

17.7.29 Stop

Refer to the section **5.8.12 MMC_Stop on page 261** for details of the description, scope, and motion mode.

```
void Stop(
[float fDeceleration],
[float fJerk],
MC_BUFFERED_MODE_ENUM eBufferMode = MC_ABORTING_MODE
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

fDeceleration

Float value of the deceleration when stopping (decreasing energy of the motor). Any positive float value in u/s^2

fJerk

Float value of the Jerk. Any positive value in u/s^3

eBufferMode

Refer to the **MMC_STOP_IN Structure on page 262**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.30 GetAxisError

Refer to the section **5.9.13 MMC_ReadAxisError on page 299** for details of the description, scope, and motion mode.

unsigned short GetAxisError(
unsigned short* usLastEmergencyErrCode
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

usLastEmergencyErrCode

Last emergency DS-402 error code that occurred in the system. Refer to the DS-402 Elmo emergency codes in the following:

- CANopen DS-301 Implementation Guide Chapter 6
- CANopen DSP-402 Implementation Guide

SimplIQ Command Reference Guide or Gold Command Reference Guide

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

17.7.31 GetDigInput[s]

Refer to the section **5.9.16 MMC_ReadDigitalInput(s) on page 309** for details of the description, scope, and motion mode.

unsigned char GetDigInput[s](
[int iInputNumber]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

iInputNumber

Selects the input depending on the drive. Can be part of MMC_Axis_Ref, if only one single input is referenced. +ve integer value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name Axis reference Error ID
Status of the axis.

17.7.32 GetDigOutputs32Bit

Refer to the section **5.9.18 MMC_ReadDigitalOutputs32Bit on page 317** for details of the description, scope, and motion mode.

```
unsigned long GetDigOutputs32bit(
int iOutputNumber
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

iOutputNumber

Selects the output. Can be part of MC_OutputRef, if only one single output is referenced. +ve integer value

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For an example, refer to the section 17.7.2.

17.7.33 GetDigOutputs

Refer to the section **5.9.17 MMC_ReadDigitalOutputs on page 314** for details of the description, scope, and motion mode.

```
unsigned char GetDigOutputs(
int iOutputNumber
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

iOutputNumber

Selects the output. Can be part of MC_OutputRef, if only one single output is referenced. +ve integer value

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.7.34 SetDigOutputs32Bit

Refer to the section **5.9.31 MMC_WriteDigitalOutputs32Bit on page 357** for details of the description, scope, and motion mode.

```
void SetDigOutputs32Bit(
[const int iOutputNumber,]
const unsigned long ulValue
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

iOutputNumber

Selects the output. Can be part of MC_OutputRef, if only one single output is referenced. +ve integer value

ulValue

Total value of all the inputs together. +ve 32 bit bitwise numeric value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID Status of the axis.

For an example, refer to the section 17.7.2.

17.7.35 SetDigOutputs

Refer to the section **5.9.30 MMC_WriteDigitalOutputs on page 354** for details of the description, scope, and motion mode.

```
void SetDigOutputs(
[const int iOutputNumber,]
const unsigned char ucValue
[unsigned char ucEnable]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

iOutputNumber

Selects the output. Can be part of MC_OutputRef, if only one single output is referenced. +ve integer value

ucValue

Selected value of the input signal. +ve integer value

ucEnable

Get the value of the parameter continuously while enabled. As long as Enable is true, power is enabled. Character values of 0, 1 integers.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name Axis reference
Error ID Status of the axis.

17.7.36 SetOverride

Refer to the sections **5.9.24 MMC_SetOverride** and **5.15.27 MMC_GroupSetOverride** for details of the description, scope, and motion mode.

```
void SetOverride(
float fAccFactor,
float fJerkFactor,
float fVelFactor
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

fAccFactor

New override factor for the acceleration/deceleration. ACC/Jerk Factors are NOT supported at this time. For future compatibility, enter "1" in the function call.

fJerkFactor

New override factor for the jerk. ACC/Jerk Factors are NOT supported at this time. For future compatibility, enter "1" in the function call.

fVelFactor

New override factor for the velocity. Any +ve float value between [0 – 1].

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

17.7.37 ConfigPDO

Refer to the sections 16.5.4 - 16.5.7 for details of the description, scope, and motion mode.

```
void ConfigPDO(
PDO_NUMBER_ENUM ePDONum,
PDO_PARAM_TYPE_ENUM eParamType,
unsigned int uiPDOCommParamEvent,
unsigned short usEventTimer,
unsigned char ucEventGroup,
unsigned char ucPDOCommParam,
unsigned char ucSubIndex,
unsigned char ucPDOType
)throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

ePDONum

Defines an enumerator whose structure PDO_NUMBER_ENUM is the PDO number with values:

PDO_NUM_3 = 3

PDO_NUM_4 = 4

eParamType

Defines an enumerator whose structure PDO_PARAM_TYPE_ENUM has the following values:

PDO_PARAM_REG = 1

PDO_PARAM_USE = 2

uiPDOCommParamEvent

This parameter inserts a PDO Events mechanism in the GMAS for servo drive operations like emit, motion complete, motion started etc. When the operation is performed, an event is sent back to the Maestro, and thereon to any connected host.

Any +ve integer in bitwise form is accepted.

usEventTimer

The timer for the PDO event. Has the following conditions depending on the drive specifications. Acceptable values are any integer in millisecs:

0 for Synchronous data

>0 for Asynchronous data

For the following PDO event times:

MC_PDO_TIMER_NON = 0

MC_PDO_TIMER_1_MILISEC = 1

MC_PDO_TIMER_2_MILISEC = 2

MC_PDO_TIMER_3_MILISEC = 3

MC_PDO_TIMER_4_MILISEC = 4

MC_PDO_TIMER_5_MILISEC = 5

MC_PDO_TIMER_10_MILISEC = 10

MC_PDO_TIMER_20_MILISEC = 20

MC_PDO_TIMER_25_MILISEC = 25

MC_PDO_TIMER_50_MILISEC = 50

MC_PDO_TIMER_100_MILISEC = 100

MC_PDO_TIMER_150_MILISEC = 150

MC_PDO_TIMER_200_MILISEC = 200

MC_PDO_TIMER_250_MILISEC = 250

MC_PDO_TIMER_255_MILISEC = 255

ucEventGroup

Defines which group of events are to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping** the correct definition to be used. Any +ve character values are acceptable.

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC 0x01

PDO_COM_PARAM_ASYNC 0xFF

PDO_COM_PARAM_EVENT 0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

ucSubIndex

Defines which index value signifies User Integer and User Float values of the servo drive. Refer to the section **16.4.4 Using Event Groups 16 and 17**. Any +ve character integers are accepted as values

ucPDOType

The direction of the PDO, according to the MC_PDO_TYPE_ENUM enumerator:

MC_PDO_TYPE_RPDO

MC_PDO_TYPE_TXPDO

This will be dependent on the value of the parameters ucEventGroup, and ucSubIndex.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name Axis reference Error ID
Status of the axis.

17.7.38 CancelPDO

Refer to the sections **16.12.1 - 16.12.4** for details of the description, scope, and motion mode.

```
void CancelPDO(
PDO_NUMBER_ENUM ePDONum
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

ePDONum

Defines an enumerator whose structure PDO_NUMBER_ENUM is the PDO number with values:

PDO_NUM_3 = 3

PDO_NUM_4 = 4

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

17.7.39 ChangeDefaultPDOConfig

Refer to the section **16.5.8 MMC_ChangeDefaultPDOConfiguration on page 1274** for details of the description, scope, and motion mode.

```
void ChangeDefaultPDOConfig(
unsigned char ucPDONum,
unsigned char ucPDODir,
unsigned char ucPDOCommParam
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

ucPDONum

Changes a specific PDO's communication from sync to async and visa versa. Allowed values are 3 and 4, representing the PDO3 and PDO4.

ucPDODir

Changes the RX or TX specific PDO communication. Allowed values are:

RXPDO=0

TXPDO=1

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC	0x01
PDO_COM_PARAM_ASYNC	0xFF
PDO_COM_PARAM_EVENT	0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID

Status of the axis.

17.7.40 ElmoSetAsyncArray

Refer to the section **16.8.8 MMC_ElmoSetArray on page 1415** for details of the description, and motion mode.

```
void ElmoSetAsyncArray(
char cCmd[3],
short iArrayIdx,
float& fVal
[int& iVal]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iArrayIdx

Index of array element. Any +ve or –ve short integer value with a maximum of 2 bytes.

iVal or fVal

Integer value of the data that is to be set. Optionally, float value of the data.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.41 ElmoSetAsyncParam

Refer to the section 16.8.3 MMC_ElmoSetParameter on page 1405 for details of the description, and motion mode.

void ElmoSetAsyncParam(
char cCmd[3],
int& iVal
[float& fVal]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iVal or fVal

Integer value of the data that is to be set.

Optionally, float value of the data.

throw (CMMCException)

Refer to the section 17.1.1 CMMCException. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.41.1 Function Code Example

```
// 16.3.38.ElmoGetAsyncIntParam       11.8.4. MMC_ElmoGetParameter
// 16.3.37.ElmoSetAsyncParam     11.8.3.     MMC_ElmoSetParameter
//    CMMCSingleAxis::ElmoGetAsyncIntParam(char cCmd[3])
//    CMMCSingleAxis::ElmoSetAsyncParam    (char cCmd[3], int  & iVal)
//    CMMCSingleAxis::ElmoSetAsyncParam    (char cCmd[3], float& fVal)
//
// 16.3.42.ElmoGetSyncParam      11.8.7. MMC_ElmoGetParameterAndRetrieveData
//    CMMCSingleAxis::ElmoGetSyncParam(char cCmd[3], float& fVal)
//    CMMCSingleAxis::ElmoGetSyncParam(char cCmd[3], int  & iVal)
void  SetGetDrvParameters(void)
// ==============================
{

 int   iValGet,
   iValSet;
    float   fValGet;
 int    loopCount,
   iMax;
 char* cPdrvCmd;
```

```
printf("\n      %s:", __func__);

iValSet = 5000;
loopCount = 0;
iMax = 100;
cPdrvCmd = "px";

      /* Before power set ON !*/
AxisA.ElmoSetAsyncParam(cPdrvCmd, iValSet);
AxisA.ElmoGetAsyncIntParam(cPdrvCmd);
while (loopCount < iMax)
{
 if(AxisA.ElmoIsReplyAwaiting())
 {
    AxisA.ElmoGetReply(iValGet) ;
    break;
 }
 loopCount++;
 WAIT_SLEEP_MILLI(10)
}
if (loopCount < iMax)
{
  printf("\n ++++ DRIVER AxisA %s Position=%d (set to %d)", cPdrvCmd, iValGet, iValSet);
}
else
{
 printf("\n ---- Fail get DRIVER AxisA %s Position...", cPdrvCmd);
}



iValSet = 8000;
AxisA.ElmoSetAsyncParam(cPdrvCmd, iValSet);
AxisA.ElmoGetSyncParam (cPdrvCmd, iValGet);
printf("\n ++++ DRIVER AxisA %s Position=%d (set to %d)", cPdrvCmd, iValGet, iValSet);

cPdrvCmd = "iq";
AxisA.ElmoGetSyncParam(cPdrvCmd, fValGet);
printf("\n ++++ DRIVER AxisA %s Current=%f", cPdrvCmd, fValGet);

}
```

17.7.42 ElmoGetAsyncIntParam

Refer to the section **16.8.4 MMC_ElmoGetParameter on page 1407** for details of the description, and motion mode.

void ElmoGetAsyncIntParam(
char cCmd[3]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.42.1 Function Code Example

```
// 16.3.37.ElmoGetAsyncIntParam       11.8.4. MMC_ElmoGetParameter
// 16.18.6.ElmoSetAsyncParam    11.8.3.     MMC_ElmoSetParameter
//    CMMCSingleAxis::ElmoGetAsyncIntParam(char cCmd[3])
//    CMMCSingleAxis::ElmoSetAsyncParam   (char cCmd[3], int  & iVal)
//    CMMCSingleAxis::ElmoSetAsyncParam   (char cCmd[3], float& fVal)
//
// 16.3.42.ElmoGetSyncParam      11.8.7. MMC_ElmoGetParameterAndRetrieveData
//    CMMCSingleAxis::ElmoGetSyncParam(char cCmd[3], float& fVal)
//    CMMCSingleAxis::ElmoGetSyncParam(char cCmd[3], int  & iVal)
void  SetGetDrvParameters(void)
// ==============================
{

 int   iValGet,
   iValSet;
    float    fValGet;
 int     loopCount,
   iMax;
 char* cPdrvCmd;


 printf("\n     %s:", __func__);

 iValSet = 5000;
 loopCount = 0;
 iMax = 100;
 cPdrvCmd = "px";


       /* Before power set ON !*/
 AxisA.ElmoSetAsyncParam(cPdrvCmd, iValSet);
```

```
AxisA.ElmoGetAsyncIntParam(cPdrvCmd);
while (loopCount < iMax)
{
 if(AxisA.ElmoIsReplyAwaiting())
 {
    AxisA.ElmoGetReply(iValGet) ;
    break;
 }
 loopCount++;
 WAIT_SLEEP_MILLI(10)
}
if (loopCount < iMax)
{
 printf("\n ++++ DRIVER AxisA %s Position=%d (set to %d)", cPdrvCmd, iValGet, iValSet);
}
else
{
 printf("\n ---- Fail get DRIVER AxisA %s Position...", cPdrvCmd);
}



iValSet = 8000;
AxisA.ElmoSetAsyncParam(cPdrvCmd, iValSet);
AxisA.ElmoGetSyncParam (cPdrvCmd, iValGet);
printf("\n ++++ DRIVER AxisA %s Position=%d (set to %d)", cPdrvCmd, iValGet, iValSet);

cPdrvCmd = "iq";
AxisA.ElmoGetSyncParam(cPdrvCmd, fValGet);
printf("\n ++++ DRIVER AxisA %s Current=%f", cPdrvCmd, fValGet);

}
```

17.7.43 ElmoGetAsyncFloatParam

Refer to the section **16.8.4 MMC_ElmoGetParameter on page 1407** for details of the description, and motion mode.

void ElmoGetAsyncFloatParam(
char cCmd[3]
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.44 ElmoGetAsyncIntArray

Refer to the section **16.8.5 MMC_ElmoGetArray on page 1409** for details of the description, and motion mode.

```
void ElmoGetAsyncIntArray(
char cCmd[3],
short iArrayIdx
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iArrayIdx

Index of array element. Any +ve or –ve short integer value with a maximum of 2 bytes.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.45 ElmoGetAsyncFloatArray

Refer to the section **16.8.5 MMC_ElmoGetArray** for details of the description, and motion mode.

```
void ElmoGetAsyncFloatArray(
char cCmd[3],
short iArrayIdx
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iArrayIdx

Index of array element. Any +ve or –ve short integer value with a maximum of 2 bytes.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.46 ElmoGetSyncParam

Refer to the section **16.8.4 MMC_ElmoGetParameter on page 1407** for details of the description, and motion mode.

```
void ElmoGetSyncParam(
char cCmd[3],
float& fVal
[int& iVal]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iVal or fVal

Integer value of the data that is to be set.

Optionally, float value of the data.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.47 ElmoGetSyncArray

Refer to the section **16.8.5 MMC_ElmoGetArray on page 1409** for details of the description, and motion mode.

```
void ElmoGetSyncArray(
char cCmd[3],
short iArrayIdx,
float& fVal
[int& iVal]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iArrayIdx

Index of array element. Any +ve or –ve short integer value with a maximum of 2 bytes.

iVal or fVal

Integer value of the data that is to be set.

Optionally, float value of the data.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.48 ElmoCallAsync

Refer to the section **16.8.12 MMC_ElmoCall** for details of the description, and motion mode.

```
void ElmoCallAsync(
char cCmd[3]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.49 ElmoExecute

Refer to the section **16.8.2 MMC_ElmoExecuteLabel on page 1401** for details of the description, and motion mode.

```
void ElmoExecute(
unsigned char* pData,
unsigned char ucLength
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

pData

String Data with the precursor format of [Metronome command]##[label]. Any +ve character values with a maximum length of 80 bytes.

ucLength

Length of the label string. Length with the precursor format of [Metronome command]##[label]. Any +ve character values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.50 ElmoIsReplyAwaiting

Refer to the section **16.8.9 MMC_ElmoQueryOperationFIFOIndex on page 1417** for details of the description, and motion mode.

```
int ElmoIsReplyAwaiting(
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**.

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.51 ElmoGetReply

Refer to the section **16.8.10 MMC_ElmoQueryOperationFIFORetrieveData on page 1418** for details of the description, and motion mode.

```
void ElmoGetReply(
float& fVal
[int& iVal]
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

iVal or fVal

Integer value of the data that is to be set.

Optionally, float value of the data.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

Scope

Note: **When using the binary interpreter, first open a UDP channel. Otherwise, the binary interpreter functions will return error -10(Lib error).**

17.7.52 ConfigVirtualEncoder

Refer to the section 16.5.12 MMC_ConfigVirtualEncoder for details of the description, scope, and motion mode.

void ConfigVirtualEncoder(
double dbLowPos,
double dbHighPos,
float fFactor,
unsigned char ucMode,
unsigned char ucGroupID
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbLowPos

Low range of the virtual encoder. Any -ve or +ve double values in technical unit [u]

dbHighPos

High range of the virtual encoder. Any -ve or +ve double values in technical unit [u]

fFactor

Encoder factor. Any +ve integer value accepted.

ucMode

Defines the virtual encoder mode of the encoder with the following options:

NC_NODE_VIRTUAL_ENC_MODE_DISABLED = 0

NC_NODE_VIRTUAL_ENC_MODE_TARGET_POS

NC_NODE_VIRTUAL_ENC_MODE_ACTUAL_POS

ucGroupID

Group CAN ID. +ve integer accepted.

throw (CMMCException)

Refer to the section 17.1.1 CMMCException. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

Example

```
#define  ENC_LOW_POS   0
#define  ENC_HIGH_POS  0
#define  ENC_ACTUAL_POS     2
#defineENC_GROUP_ID 80

try
{
cAxis[eAxis_a01_Treadmill].ConfigVirtualEncoder(ENC_LOW_POS,ENC_HIGH_POS,1
```

```
.0,ENC_ACTUAL_POS,ENC_GROUP_ID);
   state = STT_DONE;
  }
  catch (CMMCException exp)
  {
   state = STT_ERR;   }
```

17.7.53 CancelVirtualEncoder

Refer to the section **16.5.1 MMC_CancelVirtualEncoder on page 1246** for details of the description, scope, and motion mode.

void CancelVirtualEncoder(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

17.7.54 SetPosition

Refer to the section **5.9.25 MMC_SetPosition on page 338** for details of the description, scope, and motion mode.

void SetPosition(
double dbPosition,
double dbModulus,
unsigned char ucPosMode
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbPosition

Target position for the motion when conditions are met. Any -ve or +ve double values in technical unit [u].

dbModulus

The relative modulus of the axis. Any -ve or +ve double values in technical unit [u].

ucPosMode

Boolean values for the position mode can be absolute mode = 0, or relative mode = 1

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name Axis reference Error ID
Status of the axis.

17.7.55 SetParameter

Refer to the section **5.9.32 MMC_WriteParameter** for details of the description, scope, and motion mode.

```
void SetParameter(
double dbValue,
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

dbValue

Array parameter with double value.

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.4 Axis Status on page 176** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

For code example, refer to the section **17.7.3**.

17.7.56 SetBoolParameter

Refer to the section **5.9.28 MMC_WriteBoolParameter** for details of the description, scope, and motion mode.

```
void SetBoolParameter(
long lValue,
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

lValue

Input value. Any +ve integer.

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.4 Axis Status on page 176** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.7.57 AxisLink

Refer to the section **5.9.3 MMC_AxisLink** for details of the description, scope, and motion mode.

```
void AxisLink(
unsigned short usAxisRef,
unsigned char ucMode = 0
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

usAxisRef

Axis reference. Any +ve bitwise integer.

ucMode = 0

Defines the virtual encoder mode of the encoder with value 0. This parameter has the following options:

NC_NODE_VIRTUAL_ENC_MODE_DISABLED = 0

NC_NODE_VIRTUAL_ENC_MODE_TARGET_POS

NC_NODE_VIRTUAL_ENC_MODE_ACTUAL_POS

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID

Status of the axis.

17.7.58 AxisUnLink

Refer to the section **5.9.4 MMC_AxisUnLink** for details of the description, scope, and motion mode.

void AxisUnLink(
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

17.7.59 GetBoolParameter

Refer to the section **5.9.28 MMC_WriteBoolParameter** for details of the description, scope, and motion mode.

long GetBoolParameter(
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException);

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3 Axis, Group, Global, Parameters** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

17.7.60 GetParameter

Refer to the section **5.9.32 MMC_WriteParameter** for details of the description, scope, and motion mode.

```
double GetParameter(
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCSingleAxis.h

.NET Definition

Function Parameters

eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.4 Axis Status on page 176** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values

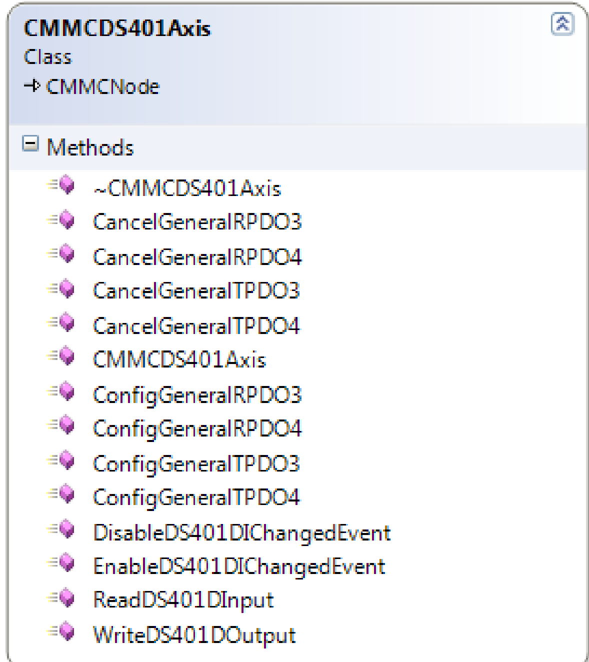
throw (CMMCException)

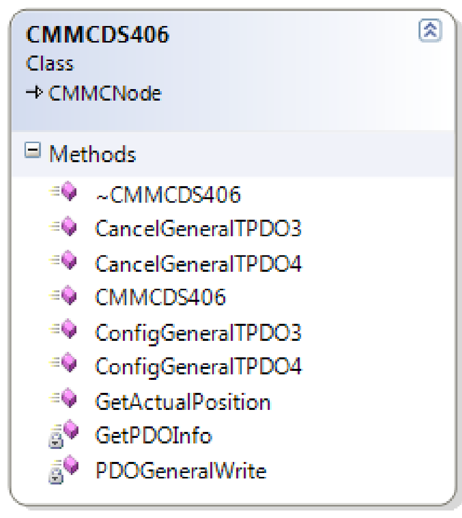
Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.8 The CMMCDS401Axis class

The class CMMCDS401Axis wraps the DS-401 parameter functions detailed in section **16.11 DS-401 CANbus I/O Communications**. The class CMMCDS401Axis retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.





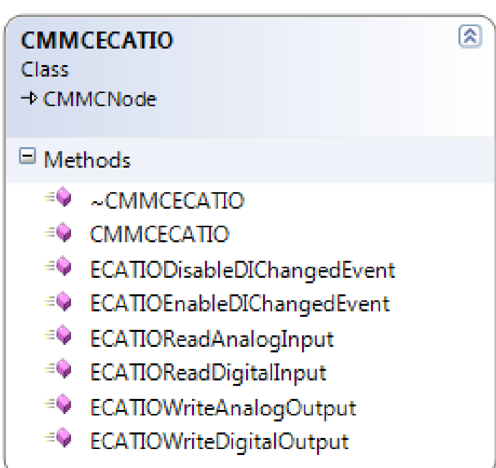


Figure 17-10 Fields and methods of the CMMCDS401Axis class

The detailed class view shown in **Figure 17-24** describes the fields and methods associated with the CMMCDS401Axis class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.8.1 ConfigGeneralRPDO3

This function configures the Maestro to receive a general PDO3 message. Refer to the section **16.12.5 MMC_ConfigGeneralRPDO3** for details of the description, and scope.

```
void ConfigGeneralRPDO3(
unsigned char ucEventType,
unsigned char ucPDOCommParam,
unsigned char ucPDOLength
) throw(CMMCException)
```

Source GMAS\includes\CPP\ CMMCDS401Axis.h

.NET Definition

Function Parameters

ucEventType

Defines which group of events are to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping** the correct definition to be used. Any +ve character values are acceptable.

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC 0x01

PDO_COM_PARAM_ASYNC 0xFF

PDO_COM_PARAM_EVENT 0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

ucPDOLength

Indicates the number of bytes to be sent as an RPDO, RPDO message. It can contain 1-8 bytes of data.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.8.2 ConfigGeneralRPDO4

This function configures the Maestro to receive a general PDO3 message. Refer to the section **16.12.8 MMC_ConfigGeneralTPDO4** for details of the description, and scope.

```
void ConfigGeneralRPDO4(
unsigned char ucEventType,
unsigned char ucPDOCommParam,
unsigned char ucPDOLength
) throw(CMMCException)
```

Source GMAS\includes\CPP\ CMMCDS401Axis.h

.NET Definition

Function Parameters

ucEventType

Defines which group of events are to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping** the correct definition to be used. Any +ve character values are acceptable.

ucPDOCommParam

PDO communications parameter. Has the following +ve character values:

PDO_COM_PARAM_SYNC 0x01

PDO_COM_PARAM_ASYNC 0xFF

PDO_COM_PARAM_EVENT 0xFE

PDO events are only possible when the input argument *ucPDOCommParam*, is PDO_COM_PARAM_EVENT.

ucPDOLength

Indicates the number of bytes to be sent as an RPDO, RPDO message. It can contain 1-8 bytes of data.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.8.3 CancelGeneralRPDO3

This function cancels the Maestro configuration from receiving general PDO3 messages. Refer to the section **16.12.1 MMC_CancelGeneralRPDO3** for details of the description, and scope.

void CancelGeneralRPDO3(
) throw(CMMCException)

Source GMAS\includes\CPP\ CMMCDS401Axis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.8.4 CancelGeneralRPDO4

This function cancels the Maestro configuration from receiving general PDO4 messages. Refer to the section **16.12.2 MMC_CancelGeneralRPDO4** for details of the description, and scope.

void CancelGeneralRPDO4(
) throw(CMMCException)

Source GMAS\includes\CPP\ CMMCDS401Axis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.8.5 ConfigGeneralRPDO4

This function cancels the Maestro configuration from receiving general PDO4 messages. Refer to the section **16.12.2 MMC_CancelGeneralRPDO4** for details of the description, and scope.

```
void CancelGeneralRPDO4(
) throw(CMMCException)
```

Source GMAS\includes\CPP\ CMMCDS401Axis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.8.6 ConfigGeneralTPDO3

This function configures the Maestro to transmit general PDO3 messages. Refer to the section **16.12.7 MMC_ConfigGeneralTPDO3** for details of the description, and scope.

```
void ConfigGeneralTPDO3(
unsigned char ucEventType
) throw(CMMCException)
```

Source GMAS\includes\CPP\ CMMCDS401Axis.h

.NET Definition

Function Parameters

ucEventType

Defines which group of events are to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping** the correct definition to be used. Any +ve character values are acceptable.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.8.7 ConfigGeneralTPDO4

This function configures the Maestro to transmit general PDO43 messages. Refer to the section **16.12.8 MMC_ConfigGeneralTPDO4** for details of the description, and scope.

```
void ConfigGeneralTPDO4(
unsigned char ucEventType
) throw(CMMCException)
```

Source GMAS\includes\CPP\ CMMCDS401Axis.h

.NET Definition

Function Parameters

ucEventType

Defines which group of events are to be transferred from the Maestro. Refer to the section **16.4.3 PDO Mapping** the correct definition to be used. Any +ve character values are acceptable.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.8.8 CancelGeneralTPDO3

This function cancels the Maestro configuration from transmitting general PDO3 messages. Refer to the section **16.12.3 MMC_CancelGeneralTPDO3** for details of the description, and scope.

```
void CancelGeneralTPDO3(
) throw(CMMCException)
```

Source GMAS\includes\CPP\ CMMCDS401Axis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.8.9 CancelGeneralTPDO4

This function cancels the Maestro configuration from transmitting general PDO4 messages. Refer to the section **16.12.4 MMC_CancelGeneralTPDO4** for details of the description, and scope.

```
void CancelGeneralRPDO3(
) throw(CMMCException)
```

Source GMAS\includes\CPP\ CMMCDS401Axis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.9 The CMMCDS406Axis class

The purpose of encoders is to detect positions of any kind of machine tools. Encoders detect positions and transmit the position values or provide speed, acceleration, and jerk values, across the CANopen network. The encoder may receive configuration information via SDO, and in the NMT state operation, the position value may be transmitted using synchronous PDO. Additionally, the encoders may transmit a PDO asynchronously, scheduled by the elapsing of the event timer.

The CANopen device profile defines two encoder classes, a standard device class 1 (C1) and an extended device class 2 (C2). The standard device C1 specifies basic functions provided by each device. The C2 extended device provides a variety of features with mandatory and optional functions. The mandatory functions of both, C1 and C2, are necessary to ensure non-manufacturer specific operations of a device.

By defining mandatory device characteristics in C1, the operation of the basic network and encoder is ensured. By defining extended C2, a degree of defined flexibility may be built-in. By leaving ‘hooks’ for optional and manufacturer-specific functions, the device developer is not constrained to an out-of-date standard.

The CiA DS406 device profile for encoders specifies the CANopen interface of absolute linear and rotary encoders. Besides position and velocity output, the profile describes also acceleration and jerk outputs, and specifies several configuration parameters, e.g. the code sequence (complement) that determines the counting direction, in which the output code increases or decreases. The resolution parameter is used to configure a given number of steps for each revolution. The profile specification covers complete cam functionality with hysteresis, and it is possible to describe multi-sensor modules implemented in a single CANopen encoder device.

The encoder profile specifies the following operation modes:

Mode	Profile
Event-timer	Current position value is sampled and transmitted periodically.
Synchronous	Current position is sampled and transmitted after the reception of the Sync message.

The remote mode based on remotely requested PDOs is not recommended due to several general problems that occur when CAN remote frames are used.

The class CMMCDS406Axis wraps the parameter functions. The class CMMCDS406Axis retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

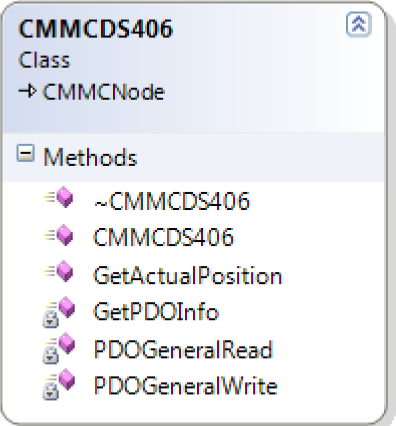


Figure 17-11 Fields and methods of the CMMCDS406Axis class

The detailed class view shown in **Figure 17-24** describes the fields and methods associated with the CMMCDS406Axis class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.9.1 GetActualPosition

Refer to the similar function block described in section **5.9.10 MMC_ReadActualPosition on page 290** for details of the description, scope, and motion mode.

```
double GetActualPosition(
) throw (CMMCException)
```

Source GMAS\includes\CPP\ MMCDS406Axis.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

For code example, refer to the section **17.7.2.**

17.10 The CMMCECATIO class

The class CMMCECATIO wraps the parameter functions. The class CMMCECATIO retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

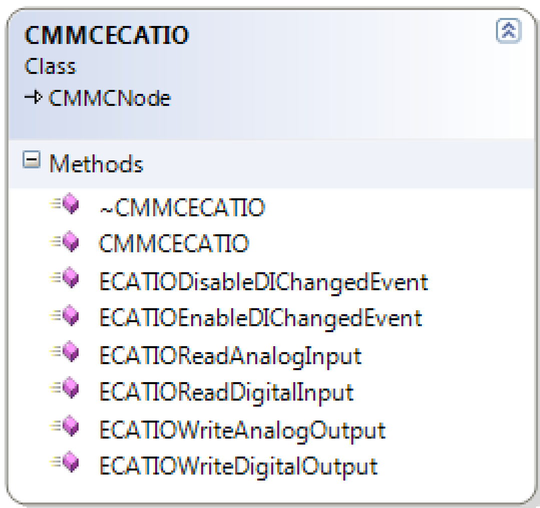


Figure 17-12 Fields and methods of the CMMCECATIO class

The detailed class view shown in **Figure 17-24** describes the fields and methods associated with the CMMCECATIO class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.10.1 ECATIOEnableDIChangedEvent

Refer to the similar function block described in section **16.7.4 MMC_ECATIOEnableDIChangedEvent** for details of the description, scope, and motion mode.

void ECATIOEnableDIChangedEvent(
) throw (CMMCException)

Source GMAS\includes\CPP\CMMCECATIO.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.10.2 ECATIODisableDIChangedEvent

Refer to the similar function block described in section **16.7.3 MMC_ECATIODisableDIChangedEvent** for details of the description, scope, and motion mode.

void ECATIODisableDIChangedEvent(
) throw (CMMCException)

Source GMAS\includes\CPP\CMMCECATIO.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.10.3 ECATIOReadDigitalInput

Refer to the similar function block described in section **16.7.5 MMC_ECATIOReadDigitalInput** for details of the description, scope, and motion mode.

(Unsigned Long Long) MMCPPULL_T ECATIOReadDigitalInput(
) throw (CMMCException)

Source GMAS\includes\CPP\CMMCECATIO.h

.NET Definition

Function Parameters

(Unsigned Long Long) MMCPPULL_T

Parameter is an 8 Byte variable with ECATIOReadDigitalInput having +ve values ranging from 0 to unlimited

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.10.4 ECATIOWriteDigitalOutput

Refer to the similar function block described in section **16.7.8 MMC_ECATIOWriteDigitalOutput** for details of the description, scope, and motion mode.

void ECATIOWriteDigitalOutput(
(Unsigned Long Long) MMCPPULL_T ulliDO
) throw (CMMCException)

Source GMAS\includes\CPP\CMMCECATIO.h

.NET Definition

Function Parameters

(Unsigned Long Long) MMCPPULL_T ulliDO

Parameter is an 8 Byte variable with ulliDO having +ve values ranging from 0 to unlimited

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name
Structure name
Axis reference
Error ID
Status of the axis.

17.10.5 ECATIOReadAnalogInput

Refer to the section **16.7.6 MMC_ECATIOReadAnalogInput** for details of the description, and scope.

short ECATIOReadAnalogInput(
unsigned char ucIndex
) throw(CMMCException)

Source GMAS\includes\CPP\CMMCECATIO.h

.NET Definition

Function Parameters

ucIndex

Analog input index. Any +ve character value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

17.10.6 ECATIOWriteAnalogOutput

Refer to the section**16.7.7 MMC_ECATIOWriteAnalogOutput** for details of the description, and scope.

void ECATIOWriteAnalogOutput(
unsigned char ucIndex,
short sAOValue
) throw(CMMCException)

Source GMAS\includes\CPP\CMMCECATIO.h

.NET Definition

Function Parameters

ucIndex

Analog input index. Any +ve character value.

sAOValue

Analog Output value. Any +ve value.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

17.11 The CMMCPPGlobal class

The class MMCPPGlobal wraps the global functions detailed in various sections of this API document. The diagram in Figure 17-13 describes the heirarchial structure of the classes and type definitions associated with the MMCPPGlobal.

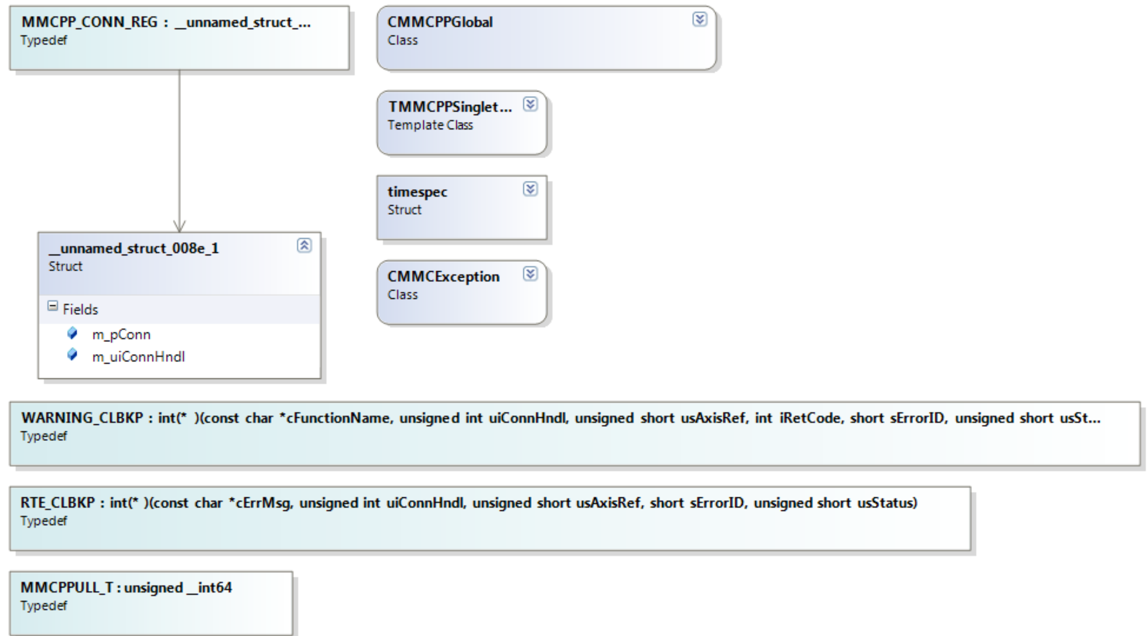


Figure 17-13 MMCPPGlobal class diagram

The class MMCPPGlobal retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

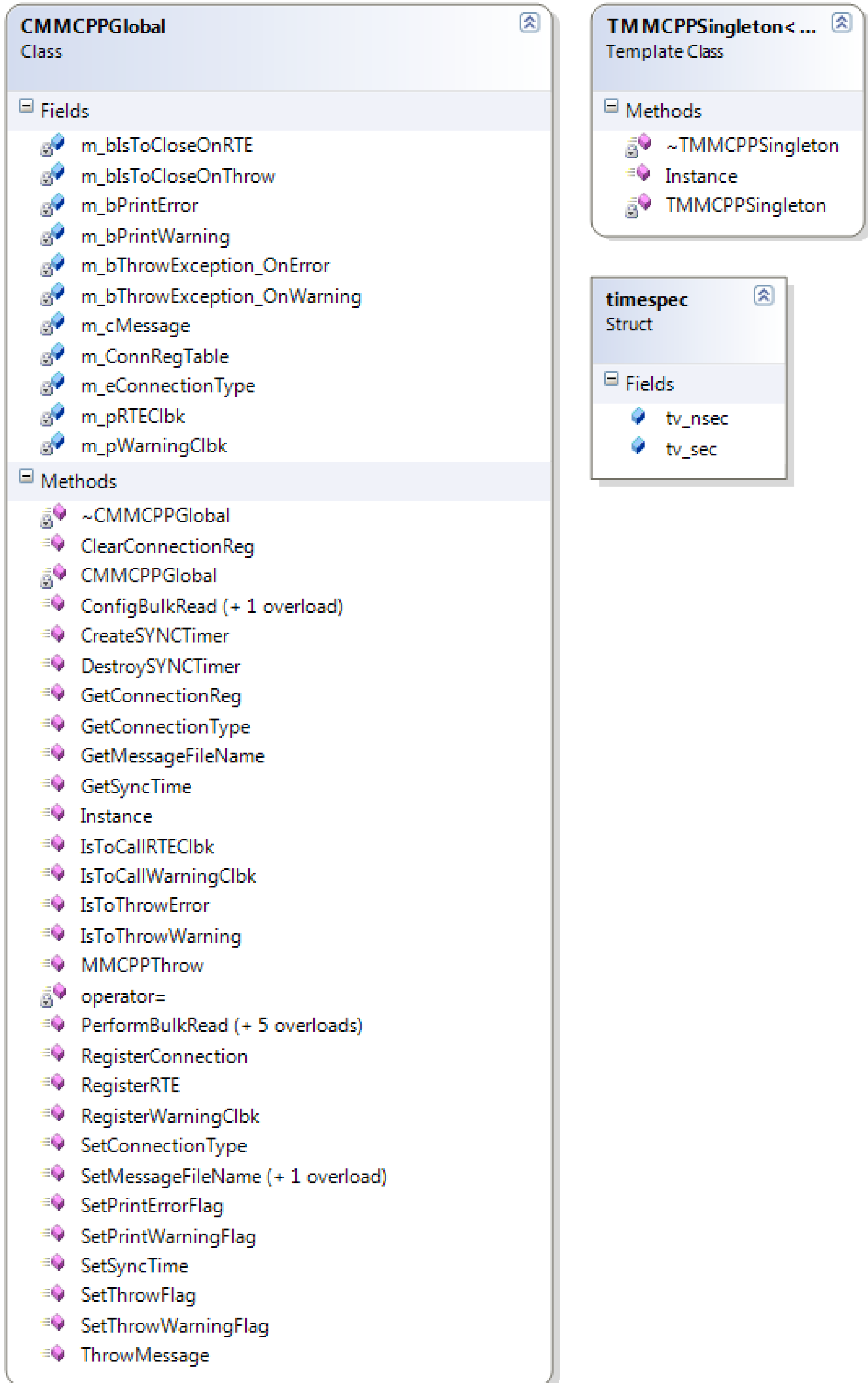


Figure 17-14 Fields and methods of the MMCPPGlobal class

The detailed class view shown in Figure 17-14 describes the fields and methods associated with the MMCPPGlobal class. These are generally default parameters, which can be operated using their default values. However if the user wishes to change the defaults, refer to the relevant parameter section in the manual.

It should be noted that Protected and Private and Protected functions together with their operations, should be transparent to the user, and are not for general application by the user.

17.11.1 CMMCPPGlobal Class Functions Code Example

```
/*
============================================================================
Collection of Gmas API functions (Set #3)
Examples for document: "G-MAS Administrative and Motion API.pdf".
12Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection          gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCSingleAxis          AxisA,    AxisB;
     unsigned short               AxisARef, AxisBRef;

     CMMCGroupAxis           Group;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

    char  * delimit = "===========================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroConnection(int);
     void      ConnectionTypeAndNum(void);
     void      SendReciveFromEthercat(int NumAmp);
     void      SetGetDefDigOutput(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
     int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus);
     void      EndMotionEventCB(unsigned short usAxisRef);
     void ModbusWrite_Received();
     void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
```

```
/*================================ Administration functions STR ===================================*/

int       main(int)
// ==============
{
     int  trace = 1;

     printf("\n %s", delimit);
     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

     try
     {
          SnroConnection(trace++);
          SnroMoveAbsolute(trace++);
          SnroEnableDisableMotionEndedEvent(trace++);
          SnroDepthName(trace++);

     }
     catch (CMMCException excp)
     {
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          exit(0);
     }

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//=====================================================================
{
                int end_of = 0;
                int iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
          end_of = 1;
/* Read Axis Status command server for specific Axis */
          ulState = sng_axis->ReadStatus();
          if (!(ulState & break_state))
          {
               end_of = 0;

               WAIT_SLEEP_MILLI(20)
          }
     }

//        MMC_SHOWNODESTAT_IN  showin;
//        MMC_SHOWNODESTAT_OUT showout;
//        MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

     return 0;
}


//   15.5.2.   RegisterRTE    Page 1274
void initAdminSingleAxis(void)
// ==============================
{
     int       iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;

/* CallbackFunc in ConnectIPCEx call if there    */
```

```
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Put Null param Val for no CallbackFunc        */
/*   ComHndl = gConn.ConnectIPCEx(iEventMask, NULL); */
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */


/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk    = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute     = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


void initAdminMultiAxis()
// =========================
{
//  Source class:
//       MMC_CONNECT_HNDL         ComHndl;
//       CMMCSingleAxis           AxisA,    AxisB;
//       CMMCGroupAxis            Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// =============================
{
     MMC_CloseConnection(ComHndl) ;
}


     void endAdminMultiAxis(void)
// ================================
{
//  Source class:
//       CMMCGroupAxis  Group;

     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
     Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================= Administration functions END ====================================*/

/*================================= Scenario functions STR =========================================*/
void SnroMoveAbsolute(int trace)
// ================================
{
```

```
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    MoveAbsoluteMoves();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void SnroEnableDisableMotionEndedEvent(int trace)
// ================================================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);
                            /* Register the callback function for Modbus and Emergency: */
    gConn.RegisterEventCallback(MMCPP_MODBUS_WRITE,(void*)ModbusWrite_Received) ;
    gConn.RegisterEventCallback(MMCPP_EMCY,(void*)Emergency_Received) ;

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    EnableDisableMotionEndedEvent();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
    endAdminSingleAxis();

    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

//   15.4.14.  GroupEnable    Page 1208
//   15.4.15.  GroupDisable   Page 1209
void SnroDepthName(int trace)
// ============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    initAdminMultiAxis();

   AxisA.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

   AxisB.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

    Group.GroupEnable();

    DepthName();

    Group.GroupDisable();

    AxisB.PowerOff(MC_BUFFERED_MODE);
    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminMultiAxis();
    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
```

```
void SnroConnection(int trace)
// ==============================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();

     ConnectionTypeAndNum();

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

/*================================= Example functions STR =========================================*/


void EnableDisableMotionEndedEvent(void)
// ========================================
{
     int  loopInd;

printf("\n Function: %s:", __func__);
     for (loopInd = 0; loopInd < 2; loopInd++)
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}


//   16.4.14.  GroupEnable         1208
//   16.4.15.  GroupDisable   1209
void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

     iVal3 = Group.GetGroupAxisByName("v01");    /* Expected 256 */
```

```
     printf("\n +++++ Reff: a01=%d a02=%d, v01=%d +++++", iVal1, iVal2, iVal3);
/*
*    iVal2 = Group.GetGroupAxisByName("v02");
*/
}


void MoveAbsoluteMoves(void)
// ============================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:        */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0      */
                             /* update default speed to 5000000          */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters            */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity      */
                             /* v=100000    which become the new def V        */
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0       */
                             /* new def v=80000                               */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:    */
                             /*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=            */
/* MC_BUFFERED_MODE (def)     */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
/* Move abs to 350000 with parameters from  */
/* above command which become the default:  */
/*        Speed 500000, Acc=1000000              */
/*        Dec=1500000, Jerk=20000000,       */
/*        buffer mode=MC_BUFFERED_MODE (def)  */
     AxisA.MoveAbsolute(350000);
}


int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{

    printf("\n *********** STR Func: %s *********** ", __func__);

                             /* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
```

```
            printf("\n Underflow event received ");
            break ;
        case MODBUS_WRITE_EVT:
            printf("\n ModBus Write event received ");
            break ;
        case TOUCH_PROBE_ENDED_EVT:
            printf("\n Touch Probe event received ");
            break ;
        default:
            printf("\n Default.... Whatever arrived event received ");
            break;
        }

       printf("\n *********** END Func: %s *********** ", __func__);
        fflush(stdout); fflush(stderr);

        return 1 ;
}

int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
//
=========================================================================================================
{
        printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
                msg, usAxisRef, sErrorID, usStatus);
        fflush(stdout); fflush(stderr);

        MMC_CloseConnection(uiConnHndl);
        exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
        printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
        printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
        fflush(stdout); fflush(stderr);
}


/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ==========================
{
        printf("\n %s Received ", __func__) ;
        fflush(stdout); fflush(stderr);
}


                              /* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// =================================================================
{
        printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
        fflush(stdout); fflush(stderr);
}



//   16.6.10. SetHeartBeatConsumer 1248
//   16.7.3.    GetNetworkInfo              1253
enum
{
        eCOMM_TYPE_NONE = 0,
        eCOMM_TYPE_ETHERCAT,
        eCOMM_TYPE_CAN
};


void ConnectionTypeAndNum(void)
// ==============================
```

```
//
{
     unsigned int                  uiHeartbeatTimeFactor;
     int                                rt;
     int                                NumFoundAmp;
     char *                             cErrotStr;
     CMMCNetwork                        CNet;
     MMC_NETWORKINFO_OUT           CanOutParams;       // Can drv connection
     MMC_GETCOMMSTATISTICS_OUT     EthCatOutParams;    // Ethcat drv connection


     printf("\n Function: %s ", __func__);

     CNet.SetConnHndl(ComHndl);
                              /* Connection type - CAN/EtherCAT */
     rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);

     if (rt == eCOMM_TYPE_ETHERCAT)
     {
                              /* !!! ETERCAT !!!*/
          rt = CNet.GetCommStatistic(EthCatOutParams);
          if (rt != 0)
          {
               cErrotStr = "EtherCat failed, get statistic";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = (int)EthCatOutParams.usNumOfSlaves;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through ETERCAT net. ",
NumFoundAmp);

                              /* Send and recive from UDP soket to driv */
          SendReciveFromEthercat(NumFoundAmp);
     }
     else if (rt == eCOMM_TYPE_CAN)
     {
/* !!! CAN !!!*/
/* !!!  hard bit should be set in resource file*/
/* take from the resource file and actual connected...   */
          uiHeartbeatTimeFactor = 1;    /* On for Every Cycle */
          gConn.SetHeartBeatConsumer(uiHeartbeatTimeFactor);

         rt = CNet.GetNetworkInfo  (CanOutParams);
          if (rt != 0)
          {
               cErrotStr = "Can failed, get NetworkInfo";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = CanOutParams.iNumOfActiveNodes;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through CAN net. ", NumFoundAmp);
     }

/* Get & Set Default Mapping Digital Output */
     SetGetDefDigOutput();

     return;

ConnectionTypeAndNum_exit_err:
     printf("\n>>> %s: *** %s %d ", __func__, cErrotStr, rt);
     return;
}
```

```
/* Should create sockets according to actual number of amplifire (param NumAmp)     */
/* ...for demo (examples etc...) assume at least two Amp. exist.                    */
//   16.18.1.  Create         1302
//   16.18.2.  SendTo         1303
//   16.18.3.  ReceiveFrom    1304
//
//   16.18.11. ElmoGetArray       1310
//   16.18.12. ElmoGetParameter   1311

void SendReciveFromEthercat(int NumAmp)
// =======================================
{
     int            rt_val = 0;
    int             iBv;
     bool           bWait;
     float          fValue;
                         /* IPC type of app. */
     char           sAxisName[50] ;
     CMMCUDP    cUDP1,
                    cUDP2;
     CMMCEoE        gEoe;

     printf("\n Function: %s ", __func__);

     rt_val = cUDP1.Create("192.168.1.5", 5001, 0); /* Ip of first drive (Gmas "last part IP" + 1) */
     rt_val = cUDP1.SendTo("vr\r", 3);
     rt_val = cUDP1.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 0 Version: <%s> ", sAxisName);

     rt_val = cUDP2.Create("192.168.1.6", 5001); /* Ip of second Gmas     */
     rt_val = cUDP2.SendTo("vr\r", 3) ;
     rt_val = cUDP2.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 1 Version: <%s> ", sAxisName);


     rt_val = gEoe.Connect("192.168.1.5", 5001, bWait);
     if (bWait == true)
     {
          usleep(10000);                         /* Wait 10 mili */
          if (gEoe.IsWritable() == false)
          {
               printf("\n>>> %s: *** Amp (Drv) connection it not writable... ", __func__);
          }
     }
/* AN[6] is command for get PsHv   */
/* return 0 on succceed otherwise 1.*/
     rt_val = gEoe.ElmoGetArray("AN", 6, fValue);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get EthCat Driver psHv AN[6] ", __func__);
     }
     else
     {
          printf("\n>>> Max Power Supply High Voltage for this driver (ip=192.168.1.5) is %3.1f ",
fValue);
     }

/* BV - Maximum Motor DC Voltage (return int)    */
/* return 0 on succeed, otherwise 1.              */
     rt_val = gEoe.ElmoGetParameter("BV", iBv);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get Bus Driver HV ", __func__);
     }
     else
     {
          printf("\n>> Actual Bus Driver (ip=192.168.1.5) Hv is %d ", iBv);
     }

     gEoe.Close();
}
```

```
//   16.3.27.  GetDigOutputs32Bit  1168
//   16.3.29.  SetDigOutputs32Bit  1169
                              /* Get & Set Default Mapping Digital Output */
void SetGetDefDigOutput(void)
// ============================
{
unsigned long       ulDigOutputs32bit;



     printf("\n Function: %s ", __func__);

                              /* Read Digital output group 0 state */
     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: B4 action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned
int)ulDigOutputs32bit);


/* Change specific bit state of digital Output group 0 */
     if ((ulDigOutputs32bit & 0x10000) != 0x00000)
     {
/* ReSet specific bit of Digital output group 0 to state "0" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit & 0xfffeffff);   /* E.g: disconnedt ps... */
     }
     else
     {
/* Set specific bit of Digital output group 0 to state "1" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit | 0x10000);     /* E.g: connect ps... */
     }


     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: Aft action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned
int)ulDigOutputs32bit);
}

/*================================= Example functions END ===========================================*/

/*================================= Output STR ============================================*/
#ifdef PROGRAM_OUTPUT
#endif /* PROGRAM_OUTPUT */
/*================================= Output END ============================================*/
```

17.11.2 RegisterRTE

Registers the Run Time Error.

```
void RegisterRTE(
RTE_CLBKP pRTEClbk,
bool bIsToCloseOnRTE = true
)
{m_pRTEClbk = pRTEClbk;
m_bIsToCloseOnRTE = bIsToCloseOnRTE;}
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

pRTEClbk

Run Time Error callback parameter

bIsToCloseOnRTE = true

Should the operation be closed on Run Time Error? 0 or 1 option, answer is True (1)

For code example, refer to the section **17.11.1**.

17.11.3 RegisterWarningClbk

Registers the warning of possible error as a callback.

```
void RegisterWarningClbk(
WARNING_CLBKP pWarningClbk
)
{m_pWarningClbk =  pWarningClbk;}
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

pWarningClbk

Warning callback parameter. Boolean 0 or 1 option

17.11.4 SetThrowFlag

Set whether a throw error is flagged or not.

```
void SetThrowFlag(
bool bThrow,
bool bIsToCloseOnThrow = true
)
m_bThrowException_OnError =  bThrow;
m_bIsToCloseOnThrow = bIsToCloseOnThrow;}
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

bThrow

Use option to throw error? Boolean 0 or 1 option.

bIsToCloseOnThrow = true

If error thrown and flagged then close operation. This will be True.

17.11.5 SetThrowWarningFlag

Set whether a throw warning is flagged or not.

```
void SetThrowWarningFlag(
bool bThrowWarining
)
{m_bThrowException_OnWarning =  bThrowWarining;}
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

bThrowWarining

Use option to throw warning? Boolean 0 or 1 option.

17.11.6 SetPrintErrorFlag

Set whether a print error is flagged or not.

```
Void SetPrintErrorFlag(
bool bPrintError
)
{m_bPrintError =  bPrintError;}
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

bPrintError

Use option to print error? Boolean 0 or 1 option.

17.11.7 SetPrintWarningFlag

Set whether a print warning is flagged or not.

```
void SetPrintWarningFlag(
bool bPrintWarining
)
{m_bPrintWarning =  bPrintWarining;}
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

bPrintWarining

Use option to print warning? Boolean 0 or 1 option.

17.11.8 ThrowMessage

Details of the throw message when a function produces an error

```
Void ThrowMessage(
int iRetval,
int iErrorID,
const char* cFunctionName
);
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

iRetval

Returned value of throw message. Integer value.

iErrorID

Error ID. Integer values.

cFunctionName

Name of the Function causing the error. Character value.

17.11.9 SetConnectionType

Sets the connection type globally.

```
Void SetConnectionType(
MMCPP_CONN_TYPE eConnectionType_IN
){m_eConnectionType = eConnectionType_IN;}
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

eConnectionType_IN

```
MMCPP_CONN_TYPE GetConnectionType(
)
{return m_eConnectionType;}
```

GetConnectionType

[IN] Inputs the connection type details.

17.11.10 SetMessageFileName

Sets the file name message sent to the Maestro

```
Void SetMessageFileName(
string sMessageFileName_IN
){m_cMessage.SetMessageFileName(sMessageFileName_IN);}

void SetMessageFileName(
char* cMessageFileName_IN
){m_cMessage.SetMessageFileName(cMessageFileName_IN);}

String GetMessageFileName(){return m_cMessage.GetMessageFileName();}
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

sMessageFileName_IN

[IN] File name message. Can be string or char

cMessageFileName_IN

[IN] File name message. Can be string or char

17.11.11 GetSyncTime

Refer to the section **16.5.15 MMC_GetSyncTime** for details of the description, scope, and communication mode.

```
unsigned short GetSyncTime(
unsigned int uiConnHndl
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

uiConnHndl

[IN] Connection handle input, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.11.12 SetSyncTime

Refer to the section **8.1.7 MMC_CreateSYNCTimer on page 787** for details of the description, scope, and communication mode.

```
int SetSyncTime(
unsigned int uiConnHndl,
unsigned short usSync
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

uiConnHndl

[IN] Connection handle input, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

usSync

Synchronous period setting. Any +ve value in milliseconds

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID

Status of the axis.

17.11.13 CreateSYNCTimer

Refer to the section **8.1.7 MMC_CreateSYNCTimer on page 787** for details of the description, scope, and communication mode.

```
void CreateSYNCTimer(
unsigned int uiConnHndl,
MMC_SYNC_TIMER_CB_FUNC pfClbk,
unsigned short usSYNCTimerTime
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

uiConnHndl

[IN] Connection handle input, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pfClbk

[IN] Points to the callback function MMC_SYNC_TIMER_CB_FUNC using MMC_CreateSYNCTimer.

usSYNCTimerTime

[IN] Defines the time between which a synchronization message is sent as an event.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.11.14 DestroySYNCTimer

Refer to the section **8.1.8 MMC_DestroySYNCTimer on page 788** for details of the description, scope, and communication mode.

void DestroySYNCTimer(
unsigned int uiConnHndl
) throw (CMMCException);

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

uiConnHndl

[IN] Connection handle input, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.11.15 GetConnectionReg

Obtains the connection register call back value.

void* GetConnectionReg(
unsigned int uiConnHndl);

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

uiConnHndl

[IN] Connection handle input, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

17.11.16 ConfigBulkRead

Refer to the section **11.1.1 MMC_ConfigBulkRead** for details of the description, scope, and communication mode.

```
void ConfigBulkRead(
MMC_CONNECT_HNDL hConnHndl,
NC_BULKREAD_CONFIG_ENUM eConfig,
[NC_BULKREAD_PRESET_ENUM ePreset,]
[unsigned long ulParameters[],]
unsigned long ulAxisRefMask,
unsigned char ucAxisRefGroup,
float* fFactorsArray
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

hConnHndl

IN] Connection handle input, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

eConfig

Defines the reading source. eBULKREAD_CONFIG_1 is reserved to the EAS application. Acceptable values are eBULKREAD_CONFIG_1 and eBULKREAD_CONFIG_2.

NC_BULKREAD_CONFIG_ENUM defines the following values:

eBULKREAD_CONFIG_NONE = -1

eBULKREAD_CONFIG_1 = 0

eBULKREAD_CONFIG_2 = 1

eBULKREAD_CONFIG_MAX,

ePreset

Whether the preset parameters is used or not. Values accepted are 0, or 1. The enumerator NC_BULKREAD_PRESET_ENUM has values defind according to the parameter**NC_BULKREAD_PRESET_ENUM on page 1067**.

ulParameters

Describes the user defined bulk parameters to be read. Any +ve value.

ulAxisRefMask

Defines the axes, with the bitwise information read from (within a group). For example, if this value is set to 0x03 – only axes with axisref 0 and 1 are read.

ucAxisRefGroup

Defines the "group" where the axes are located. For example, if this value is set to 0,

and ulAxisRefMask value is set to 0x03 – Axes 0 and 1 will be read. If the group value is set to 1 – axes 25 and 26 will be read. Acceptable values – 0 – 3.

fFactorsArray

The *fFactorsArray* returns an array of factors for each parameter, with enumerator values:

NC_BULKREAD_PRESET_1 stOutputData[]

NC_BULKREAD_PRESET_2 stOutputData[]

NC_BULKREAD_PRESET_3 stOutputData[]

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.11.17 PerformBulkRead

Refer to the section **11.1.2 MMC_PerformBulkRead on page 1073** for details of the description, scope, and communication mode.

```
void PerformBulkRead(
MMC_CONNECT_HNDL hConnHndl,
int iNumberOfAxes,
NC_BULKREAD_CONFIG_ENUM eConfiguration,
NC_BULKREAD_PRESET_ENUM& eChosenPreset,
NC_BULKREAD_PRESET_1 stOutputData[]
[NC_BULKREAD_PRESET_2 stOutputData[]]
[NC_BULKREAD_PRESET_3 stOutputData[]]
[NC_BULKREAD_PRESET_4 stOutputData[]]
[NC_BULKREAD_PRESET_5 stOutputData[]]
[unsigned long* ulOutputData]
);
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

hConnHndl

IN] Connection handle input, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

iNumberOfAxes

The number of axes whose parameters are to be read. An integer.

eConfiguration

Refer to the definition of the input parameter in the section **MMC_PERFORMBULKREAD_IN on page 1074**.

eChosenPreset

Refer to the definition of the output parameter in the section **MMC_PERFORMBULKREAD_OUT on page 1074**.

stOutputData[]

Output data selection for NC_BULKREAD_PRESET_1 to 5 from the **NC_BULKREAD_PRESET_ENUM**. Refer to the section **NC_BULKREAD_PRESET_ENUM on page 1067**.

ulOutputData

User defined output data option. Entered as unlimited set of values.

17.11.18 RegisterConnection

Registers the connection with the Maestro.

```
int RegisterConnection(
unsigned int uiConnHndl,
void * pConn
);
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

uiConnHndl

[IN] Connection handle input, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

pConn

Connection check. Boolean options 0 or 1

17.11.19 GetConnectionReg ClearConnectionReg

Obtains or clears the connection registry from the Maestro

```
void* GetConnectionReg(
unsigned int uiConnHndl
);
void ClearConnectionReg(
unsigned int uiConnHndl
);
```

Source GMAS\includes\CPP\MMCPPGlobal.h

.NET Definition

Function Parameters

uiConnHndl

[IN] Connection handle input, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

17.12 The CMMCConnection class

The class MMCConnection wraps the global functions detailed in various sections of this API document. The diagram in Figure 17-15 describes the heirarchial structure of the classes and type definitions associated with the MMCConnection.

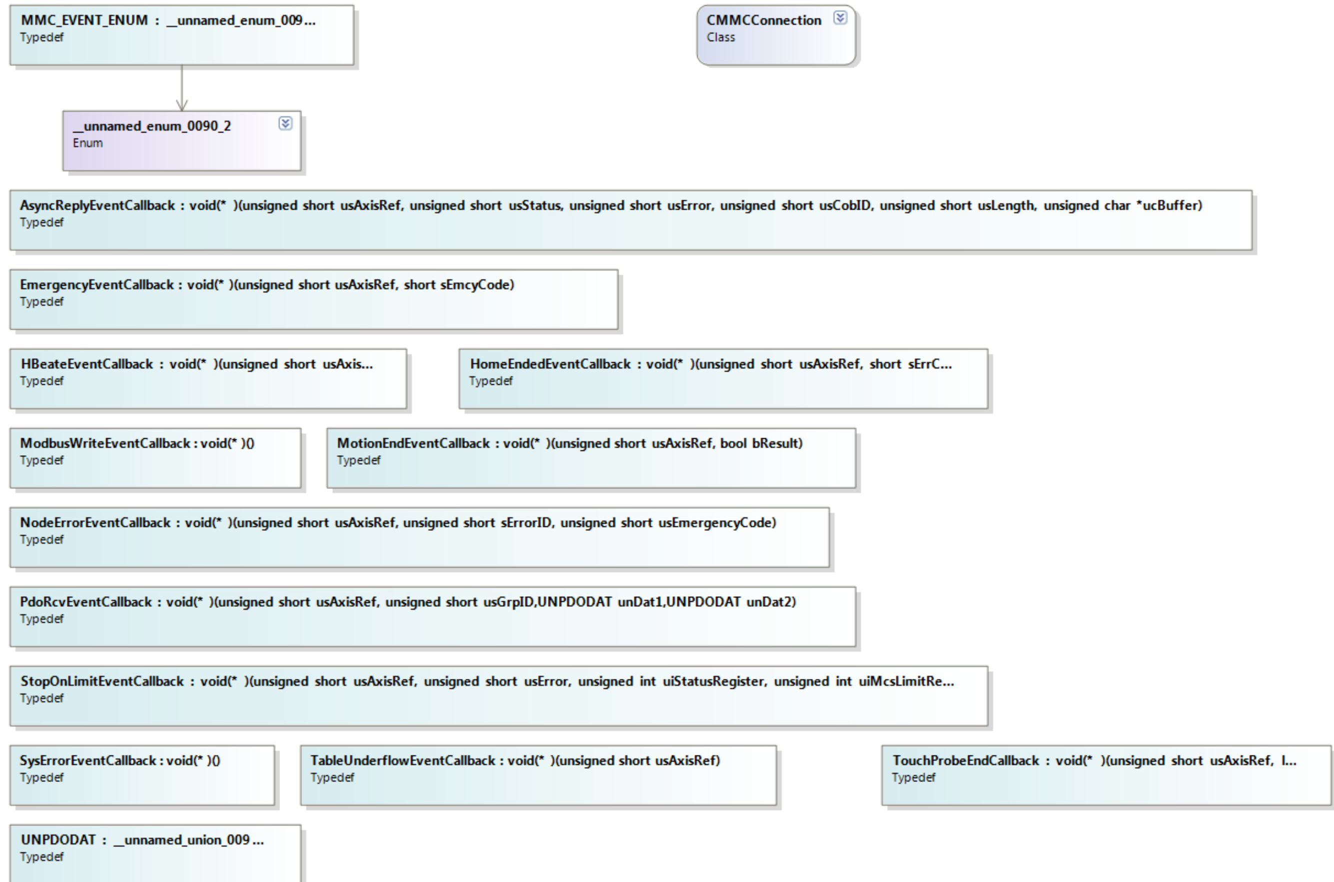


Figure 17-15 MMCConnection class diagram

The class MMCConnection retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

[

Figure 17-16 Fields and methods of the MMCConnection class

The detailed class view shown in **Figure 17-16** describes the fields and methods associated with the MMCPPGlobal class. These are generally default parameters, which can be operated using their default values. However if the user wishes to change the defaults, refer to the relevant parameter section in the manual.

Figure 17-17 Events enumerators

Figure 17-17 describes the events enumerator described and called by the function RegisterEventCallback used in conjunction with the relevant type definition.

It should be noted that Protected and Private and Protected functions together with their operations, should be transparent to the user, and are not for general application by the user.

17.12.1 CMMCConnection Class Functions Code Example 1

```
/*
============================================================================
Collection of Gmas API functions (Set #3)
Examples for document: "G-MAS Administrative and Motion API.pdf".
12Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection          gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCSingleAxis          AxisA,    AxisB;
     unsigned short               AxisARef, AxisBRef;

     CMMCGroupAxis           Group;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

    char  * delimit = "===========================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroConnection(int);
     void      ConnectionTypeAndNum(void);
     void      SendReciveFromEthercat(int NumAmp);
     void      SetGetDefDigOutput(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
     int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus);
     void      EndMotionEventCB(unsigned short usAxisRef);
     void ModbusWrite_Received();
     void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
```

```
/*================================ Administration functions STR ===================================*/

int       main(int)
// ==============
{
     int  trace = 1;

     printf("\n %s", delimit);
     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

     try
     {
          SnroConnection(trace++);
          SnroMoveAbsolute(trace++);
          SnroEnableDisableMotionEndedEvent(trace++);
          SnroDepthName(trace++);

     }
     catch (CMMCException excp)
     {
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          exit(0);
     }

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//======================================================================
{
                int end_of = 0;
                int iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
          end_of = 1;
/* Read Axis Status command server for specific Axis */
          ulState = sng_axis->ReadStatus();
          if (!(ulState & break_state))
          {
               end_of = 0;

               WAIT_SLEEP_MILLI(20)
          }
     }

//        MMC_SHOWNODESTAT_IN  showin;
//        MMC_SHOWNODESTAT_OUT showout;
//        MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

     return 0;
}


//   15.5.2.   RegisterRTE    Page 1274
void initAdminSingleAxis(void)
// ==============================
{
     int       iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;

/* CallbackFunc in ConnectIPCEx call if there    */
```

```
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Put Null param Val for no CallbackFunc        */
/*   ComHndl = gConn.ConnectIPCEx(iEventMask, NULL); */
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */


/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk    = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection     = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode    = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute      = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


void initAdminMultiAxis()
// =========================
{
//  Source class:
//       MMC_CONNECT_HNDL         ComHndl;
//       CMMCSingleAxis           AxisA,    AxisB;
//       CMMCGroupAxis            Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// =============================
{
     MMC_CloseConnection(ComHndl) ;
}


     void endAdminMultiAxis(void)
// ================================
{
//  Source class:
//       CMMCGroupAxis  Group;

     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
     Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================ Administration functions END ====================================*/

/*================================= Scenario functions STR =========================================*/
void SnroMoveAbsolute(int trace)
// ================================
{
```

```
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    MoveAbsoluteMoves();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void SnroEnableDisableMotionEndedEvent(int trace)
// ================================================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);
                            /* Register the callback function for Modbus and Emergency: */
    gConn.RegisterEventCallback(MMCPP_MODBUS_WRITE,(void*)ModbusWrite_Received) ;
    gConn.RegisterEventCallback(MMCPP_EMCY,(void*)Emergency_Received) ;

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    EnableDisableMotionEndedEvent();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
    endAdminSingleAxis();

    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

//   15.4.14.  GroupEnable    Page 1208
//   15.4.15.  GroupDisable   Page 1209
void SnroDepthName(int trace)
// ============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    initAdminMultiAxis();

   AxisA.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

   AxisB.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

    Group.GroupEnable();

    DepthName();

    Group.GroupDisable();

    AxisB.PowerOff(MC_BUFFERED_MODE);
    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminMultiAxis();
    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
```

```
void SnroConnection(int trace)
// ==============================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();

     ConnectionTypeAndNum();

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

/*================================= Example functions STR =========================================*/


void EnableDisableMotionEndedEvent(void)
// ========================================
{
     int  loopInd;

printf("\n Function: %s:", __func__);
     for (loopInd = 0; loopInd < 2; loopInd++)
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}


//   16.4.14.  GroupEnable        1208
//   16.4.15.  GroupDisable   1209
void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

     iVal3 = Group.GetGroupAxisByName("v01");    /* Expected 256 */
```

```
     printf("\n +++++ Reff: a01=%d a02=%d, v01=%d +++++", iVal1, iVal2, iVal3);
/*
*    iVal2 = Group.GetGroupAxisByName("v02");
*/
}


void MoveAbsoluteMoves(void)
// ============================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:        */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0      */
                              /* update default speed to 5000000          */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters            */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity      */
                              /* v=100000    which become the new def V        */
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0       */
                              /* new def v=80000                              */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:    */
                              /*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=            */
/* MC_BUFFERED_MODE (def)     */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
/* Move abs to 350000 with parameters from  */
/* above command which become the default:  */
/*       Speed 500000, Acc=1000000               */
/*       Dec=1500000, Jerk=20000000,        */
/*       buffer mode=MC_BUFFERED_MODE (def)  */
     AxisA.MoveAbsolute(350000);
}


int      CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{

    printf("\n *********** STR Func: %s *********** ", __func__);

                              /* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
```

```
            printf("\n Underflow event received ");
            break ;
        case MODBUS_WRITE_EVT:
            printf("\n ModBus Write event received ");
            break ;
        case TOUCH_PROBE_ENDED_EVT:
            printf("\n Touch Probe event received ");
            break ;
        default:
            printf("\n Default.... Whatever arrived event received ");
            break;
        }

    printf("\n *********** END Func: %s *********** ", __func__);
    fflush(stdout); fflush(stderr);

    return 1 ;
}

int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
//
=========================================================================================================
{
    printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
            msg, usAxisRef, sErrorID, usStatus);
    fflush(stdout); fflush(stderr);

    MMC_CloseConnection(uiConnHndl);
    exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
    printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
    printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
    fflush(stdout); fflush(stderr);
}


/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ==========================
{
    printf("\n %s Received ", __func__) ;
    fflush(stdout); fflush(stderr);
}


                             /* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ===================================================================
{
    printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
    fflush(stdout); fflush(stderr);
}
```

```
//   15.6.11. SetHeartBeatConsumer     1248
//   15.7.3.  GetNetworkInfo           1253
enum
{
     eCOMM_TYPE_NONE = 0,
     eCOMM_TYPE_ETHERCAT,
     eCOMM_TYPE_CAN
};


void ConnectionTypeAndNum(void)
// ===============================
//
{
     unsigned int                 uiHeartbeatTimeFactor;
     int                               rt;
     int                               NumFoundAmp;
     char *                            cErrotStr;
     CMMCNetwork                       CNet;
     MMC_NETWORKINFO_OUT          CanOutParams;       // Can drv connection
     MMC_GETCOMMSTATISTICS_OUT    EthCatOutParams;    // Ethcat drv connection


     printf("\n Function: %s ", __func__);

     CNet.SetConnHndl(ComHndl);
                              /* Connection type - CAN/EtherCAT */
     rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);

     if (rt == eCOMM_TYPE_ETHERCAT)
     {
                              /* !!! ETERCAT !!!*/
          rt = CNet.GetCommStatistic(EthCatOutParams);
          if (rt != 0)
          {
               cErrotStr = "EtherCat failed, get statistic";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = (int)EthCatOutParams.usNumOfSlaves;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through ETERCAT net. ",
NumFoundAmp);

                              /* Send and recive from UDP soket to driv */
          SendReciveFromEthercat(NumFoundAmp);
     }
     else if (rt == eCOMM_TYPE_CAN)
     {
/* !!! CAN !!!*/
/* !!!  hard bit should be set in resource file*/
/* take from the resource file and actual connected...  */
          uiHeartbeatTimeFactor = 1;    /* On for Every Cycle */
          gConn.SetHeartBeatConsumer(uiHeartbeatTimeFactor);

         rt = CNet.GetNetworkInfo  (CanOutParams);
          if (rt != 0)
          {
               cErrotStr = "Can failed, get NetworkInfo";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = CanOutParams.iNumOfActiveNodes;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through CAN net. ", NumFoundAmp);
     }

/* Get & Set Default Mapping Digital Output */
     SetGetDefDigOutput();

     return;

ConnectionTypeAndNum_exit_err:
     printf("\n>>> %s: *** %s %d ", __func__, cErrotStr, rt);
     return;
}
```

```
/* Should create sockets according to actual number of amplifire (param NumAmp)     */
/* ...for demo (examples etc...) assume at least two Amp. exist.                     */
//   16.18.1.  Create         1302
//   16.18.2.  SendTo         1303
//   16.18.3.  ReceiveFrom    1304
//
//   16.18.11. ElmoGetArray       1310
//   16.18.12. ElmoGetParameter   1311

void SendReciveFromEthercat(int NumAmp)
// =======================================
{
     int            rt_val = 0;
    int             iBv;
     bool           bWait;
     float          fValue;
                         /* IPC type of app. */
     char           sAxisName[50] ;
     CMMCUDP   cUDP1,
                    cUDP2;
     CMMCEoE        gEoe;

     printf("\n Function: %s ", __func__);

     rt_val = cUDP1.Create("192.168.1.5", 5001, 0); /* Ip of first drive (Gmas "last part IP" + 1) */
     rt_val = cUDP1.SendTo("vr\r", 3);
     rt_val = cUDP1.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 0 Version: <%s> ", sAxisName);

     rt_val = cUDP2.Create("192.168.1.6", 5001); /* Ip of second Gmas     */
     rt_val = cUDP2.SendTo("vr\r", 3) ;
     rt_val = cUDP2.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 1 Version: <%s> ", sAxisName);


     rt_val = gEoe.Connect("192.168.1.5", 5001, bWait);
     if (bWait == true)
     {
          usleep(10000);                         /* Wait 10 mili */
          if (gEoe.IsWritable() == false)
          {
               printf("\n>>> %s: *** Amp (Drv) connection it not writable... ", __func__);
          }
     }
/* AN[6] is command for get PsHv   */
/* return 0 on succceed otherwise 1.*/
     rt_val = gEoe.ElmoGetArray("AN", 6, fValue);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get EthCat Driver psHv AN[6] ", __func__);
     }
     else
     {
          printf("\n>>> Max Power Supply High Voltage for this driver (ip=192.168.1.5) is %3.1f ",
fValue);
     }

/* BV - Maximum Motor DC Voltage (return int)    */
/* return 0 on succeed, otherwise 1.              */
     rt_val = gEoe.ElmoGetParameter("BV", iBv);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get Bus Driver HV ", __func__);
     }
     else
     {
          printf("\n>> Actual Bus Driver (ip=192.168.1.5) Hv is %d ", iBv);
     }

     gEoe.Close();
}
```

```
//   16.3.27.  GetDigOutputs32Bit  1168
//   16.3.29.  SetDigOutputs32Bit  1169
                              /* Get & Set Default Mapping Digital Output */
void SetGetDefDigOutput(void)
// ============================
{
unsigned long       ulDigOutputs32bit;



     printf("\n Function: %s ", __func__);

                              /* Read Digital output group 0 state */
     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: B4 action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned
int)ulDigOutputs32bit);


/* Change specific bit state of digital Output group 0 */
     if ((ulDigOutputs32bit & 0x10000) != 0x00000)
     {
/* ReSet specific bit of Digital output group 0 to state "0" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit & 0xfffeffff);   /* E.g: disconnedt ps... */
     }
     else
     {
/* Set specific bit of Digital output group 0 to state "1" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit | 0x10000);     /* E.g: connect ps... */
     }


     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: Aft action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned
int)ulDigOutputs32bit);
}

/*================================= Example functions END ===========================================*/

/*================================= Output STR =============================================*/
#ifdef PROGRAM_OUTPUT
#endif /* PROGRAM_OUTPUT */
/*================================= Output END =============================================*/
```

17.12.2 CMMCConnection Class Functions Code Example 2

```
/*
============================================================================
Collection of Gmas API functions (Set #2)
Examples for document: "G-MAS Administrative and Motion API.pdf".
09Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define  EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"
#define STAR_DIM                        2


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection           gConn;
     MMC_CONNECT_HNDL         ComHndl;

     CMMCHostComm             cHost ;

     CMMCSingleAxis           AxisA,    AxisB;
     unsigned short                AxisARef, AxisBRef;

     CMMCGroupAxis             Group;
     MMC_MOTIONPARAMS_GROUP   ParamsGroup;

     MMC_SETKINTRANSFORM_IN   SetKin;

    char  * delimit = "============================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroMbusfunc(int);
     void mBusReadWriteHoldReg(void);
     void mBusReadWriteCoil(void);
     void mBusReadWriteInput(void);
     void moveAxisAAndRepActualParamOnMbusPos(void);
     void      RepAxisAActualParamOnMbus(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
```

```
	int 	OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, 
short sErrorID, unsigned short usStatus);
	void 	EndMotionEventCB(unsigned short usAxisRef);
	void ModbusWrite_Received();
	void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
	int 	mapConfigPdo(void);


/*================================ Administration functions STR ===================================*/
int 	main(int)
// ==============
{
	int  trace = 1;

	printf("\n %s", delimit);
	printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

	try
	{
		SnroMoveAbsolute(trace++);
		SnroEnableDisableMotionEndedEvent(trace++);
		SnroDepthName(trace++);

		SnroMbusfunc(trace++);
	}
	catch (CMMCException excp)
	{
		printf("\n %s", delimit);
		printf("\n %s", delimit);
		printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(), 
(short)excp.error(), excp.status());
		printf("\n %s", delimit);
		printf("\n %s", delimit);
		exit(0);
	}

	printf("\n End of %s ", __FILE__);
	printf("\n %s\n\n", delimit);
	return 0;
}


//	15.11.3.  CMMCNode::ReadStatus	  1374
int 	WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//===================================================================
{
			int end_of = 0;
			int iCount = 0;
	unsigned int	ulState;

	while( ! end_of)
	{
		iCount ++;
		end_of = 1;
/* Read Axis Status command server for specific Axis */
		ulState = sng_axis->ReadStatus();
		if (!(ulState & break_state))
		{
			end_of = 0;

			WAIT_SLEEP_MILLI(20)
		}
	}

//		MMC_SHOWNODESTAT_IN  showin;
//		MMC_SHOWNODESTAT_OUT showout;
//		MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

	return 0;
}


// 15.6.5.	CMMCConnection::ConnectIPCEx  1335
```

```
// 15.6.13.    CMMCConnection::CallbackFunc  1345
void initAdminSingleAxis(void)
// ==============================
{
     int iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;


/* CallbackFunc in ConnectIPCEx call if there is no calling to 'RegisterEventCallback' */

     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Should Not calling, called inside 'ConnectIPCEx'    */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */

/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk     = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,      */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION   */
     stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute     = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}



//   15.4.21.  CMMCGroupAxis::AddAxisToGroup 1212

void initAdminMultiAxis()
// ========================
{
//   MMC_CONNECT_HNDL        ComHndl;
//   CMMCSingleAxis          AxisA,    AxisB;
//   CMMCGroupAxis           Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// =============================
{
     MMC_CloseConnection(ComHndl) ;
}


//   15.4.5.   CMMCGroupAxis::RemoveAxisFromGroup 1197
     void endAdminMultiAxis(void)
// ================================
{

//   CMMCGroupAxis            Group;

     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
```

```
    Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================= Administration functions END ====================================*/


/*================================= Scenario functions STR =========================================*/
void SnroMoveAbsolute(int trace)
// ===============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    MoveAbsoluteMoves();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


//   15.6.14.  CMMCConnection::RegisterEventCallback  1358
void SnroEnableDisableMotionEndedEvent(int trace)
// ================================================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    EnableDisableMotionEndedEvent();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
    endAdminSingleAxis();

    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


void SnroDepthName(int trace)
// ============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    initAdminMultiAxis();

   AxisA.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

   AxisB.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

    Group.GroupEnable();

    DepthName();

    Group.GroupDisable();

    AxisB.PowerOff(MC_BUFFERED_MODE);
    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
```

```
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminMultiAxis();
    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


//   15.8.6.   cHost::MbusIsRunning
//   15.8.2.   cHost::MbusStartServer
//   15.8.3.   cHost::MbusStopServer
void SnroMbusfunc(int trace)
// ============================
{
static    bool      mBusServerState;


    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
/* Starting MBus server, map read/write user req. to Mbus area & responsible to Mbus commu.*/
    mBusServerState = cHost.MbusIsRunning(ComHndl);
    if (mBusServerState)
    {
        printf("\n ... ModBus Server is already running ");
    }
    else
    {
        printf("\n ... ModBus Server is NOT running - activate it ");
    }
/* Call to StartServer even if it is already running, it does not start the server */
/* because already started, but initialize init struct */
    cHost.MbusStartServer(ComHndl, 1) ;
/* Mode Bus functions examples.        */
    mBusReadWriteHoldReg();
/* Move Axis A and report it poss.   on Mbus.*/
    moveAxisAAndRepActualParamOnMbusPos();
    mBusReadWriteCoil();
    mBusReadWriteInput();

/* If I am the one who activates the Mbus server, I wish to stop it when I finish.  */
    if (! mBusServerState)
    {
/* Stop MBus server */
        cHost.MbusStopServer();
        printf("\n ... Stop ModBus Server ");
    }
    else
    {
        printf("\n ... Left ModBus Server running ");
    }

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
/*================================ Scenario functions END =========================================*/


/*================================= Example functions STR ==========================================*/

void EnableDisableMotionEndedEvent(void)
// ========================================
{
    int  loopInd;

printf("\n Function: %s:", __func__);
    for (loopInd = 0; loopInd < 2; loopInd++)
```

```
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}

void DepthName(void)
// ===================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

// iVal3 = AxisA.GetAxisByName("A01"); /* It case sensitive - Not define - exception... */

     iVal1 = Group.GetGroupAxisByName("v01");    /* Expected 256 */
/*
 *   iVal2 = Group.GetGroupAxisByName("v02");
 */
}


void MoveAbsoluteMoves(void)
// ===========================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:   */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0 */
/* update default speed to 5000000     */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters       */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity */
/* v=100000    which become the new def V    */
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0  */
/* new def v=80000  */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:    */
/*    Speed=500000, Acc=1000000, Dec=1500000,*/
```

```
/*   Jerk=20000000, buffer mode=            */
/* MC_BUFFERED_MODE (def)    */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
                              /* Move abs to 350000 with parameters from  */
                              /* above command which become the default:  */
                              /*        Speed 500000, Acc=1000000              */
                              /*        Dec=1500000, Jerk=20000000,       */
                              /*        buffer mode=MC_BUFFERED_MODE (def)  */
     AxisA.MoveAbsolute(350000);
}


//   15.6.13.  CMMCConnection::CallbackFunc      1357
int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{
//   int  ind;

    printf("\n *********** STR Func: %s *********** ", __func__);

//    printf("\n");
//    for (ind=0; ind < recvBufferSize; ind++)
//    {
//        printf("   recvBuffer[%d]=%d", ind, recvBuffer[ind]);
//    }
//    printf("\n");

/* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
          printf("\n Underflow event received ");
          break ;
     case MODBUS_WRITE_EVT:
          printf("\n ModBus Write event received ");
          break ;
     case TOUCH_PROBE_ENDED_EVT:
          printf("\n Touch Probe event received ");
          break ;
     default:
          printf("\n Default.... Whatever arrived event received ");
          break;
     }

    printf("\n *********** END Func: %s *********** ", __func__);
     fflush(stdout); fflush(stderr);

     return 1 ;
}
```

```
//
// Needs registeration, E.g:  CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);
//
int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
//
===================================================================================================
{
     printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
             msg, usAxisRef, sErrorID, usStatus);
     fflush(stdout); fflush(stderr);

     MMC_CloseConnection(uiConnHndl);
     exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
     printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
     printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
     fflush(stdout); fflush(stderr);
}

/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ==========================
{
     printf("\n %s Received ", __func__) ;
     fflush(stdout); fflush(stderr);
}

/* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ===================================================================
{
     printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
     fflush(stdout); fflush(stderr);
}


#define   MODBUS_UPDATE_START_INDEX              0
#define   MODBUS_UPDATE_CNT                 4
                              /* WRITE INDEXS extend along two indexs */
#define MODBUS_STR_REP_ACT_VAL_INDX         (MODBUS_UPDATE_START_INDEX+MODBUS_UPDATE_CNT)
#define   MBUS_WAIT_FOR_USER                     30000

//   15.8.4.   cHost::MbusReadHoldingRegisterTable
//   15.8.5.   cHost::MbusWriteHoldingRegisterTable
void mBusReadWriteHoldReg(void)
// ==============================
{
          int       startRef;
          int       refCnt;
          int            loopCount = 0;

     MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN     mbus_write_in;
     MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT     mbus_read_out;


     printf("\n\n Function: %s: ", __func__);

     memset(mbus_write_in.regArr,0x0, 250) ;

     startRef = MODBUS_UPDATE_START_INDEX;
     refCnt    = MODBUS_UPDATE_CNT;
     mbus_write_in.startRef        = startRef;
     mbus_write_in.refCnt          = refCnt;
     mbus_write_in.regArr[0]  = 0xf0f0;
     mbus_write_in.regArr[1]  = 0x0f0f;
     mbus_write_in.regArr[2]  = 0x5a5a;
```

```
    mbus_write_in.regArr[3]  = 0xa5a5;


    printf("\n ... You can activate the EAS and look on EAS Mbus reg 1-4 ");

    printf("\n ... Going deal with Mbus H.Reg. EAS ind 1-4 ");
    loopCount = 0;
    do
    {
         loopCount++;
/* write into mBus Holding register */
/* data on reg. ind 0-3           */
         cHost.MbusWriteHoldingRegisterTable(mbus_write_in);
         printf("\n ... Mbus Write  H.Reg ind 0-3: %8x %8x %8x %8x ",
                                           (int)mbus_write_in.regArr[0],
                                           (int)mbus_write_in.regArr[1],
                                           (int)mbus_write_in.regArr[2],
                                           (int)mbus_write_in.regArr[3]);
         printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);

         WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
/* read mbus H.Reg start from            */
/* location startRef along refCnt reg*/

         cHost.MbusReadHoldingRegisterTable(startRef, refCnt, mbus_read_out);
         printf("\n ... Mbus Readed H.Reg ind 0-3: %8x %8x %8x %8x ",
                                            (int)mbus_read_out.regArr[0],
                                            (int)mbus_read_out.regArr[1],
                                            (int)mbus_read_out.regArr[2],
                                            (int)mbus_read_out.regArr[3]);

         mbus_write_in.regArr[0] = ~(mbus_read_out.regArr[0]);
         mbus_write_in.regArr[1] = ~(mbus_read_out.regArr[1]);
         mbus_write_in.regArr[2] = ~(mbus_read_out.regArr[2]);
         mbus_write_in.regArr[3] = ~(mbus_read_out.regArr[3]);

    } while (loopCount < 3);
}


//   15.8.7.   cHost::MbusReadCoilsTable
//   15.8.8.   cHost::MbusWriteCoilsTable
void mBusReadWriteCoil(void)
// ============================
{
         int       startRef;
         int       refCnt;

         int            loopCount = 0;

    MMC_MODBUSWRITECOILS_IN      stInParams_Coil;
    MMC_MODBUSREADCOILS_OUT      stOutParams_Coil;


    printf("\n\n Function: %s: ", __func__);

    startRef = MODBUS_UPDATE_START_INDEX;
    refCnt    = MODBUS_UPDATE_CNT;
    stInParams_Coil.startRef= startRef;
    stInParams_Coil.refCnt   = refCnt;
    stInParams_Coil.coilsArr[0] = (char)0xf0;
    stInParams_Coil.coilsArr[1] = (char)0x0f;
    stInParams_Coil.coilsArr[2] = (char)0x5a;
    stInParams_Coil.coilsArr[3] = (char)0xa5;

    printf("\n ... Going deal with Mbus Coils EAS ind 1-4 ");
    loopCount = 0;
    do
    {
         loopCount++;
/* write into mBus Coil register */
/* data on reg. ind 0-3           */
         cHost.MbusWriteCoilsTable(stInParams_Coil);
         printf("\n ... Mbus Write  Coil ind 0-3: %8x %8x %8x %8x ",
```

```
                                   (int)stInParams_Coil.coilsArr[0],
                                   (int)stInParams_Coil.coilsArr[1],
                                   (int)stInParams_Coil.coilsArr[2],
                                   (int)stInParams_Coil.coilsArr[3]);
          printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
          WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
/* read mbus Coil start from       */
/* location startRef along refCnt reg*/
          cHost.MbusReadCoilsTable(stInParams_Coil.startRef, stInParams_Coil.refCnt, stOutParams_Coil);
          printf("\n ... Mbus Readed Coil ind 0-3: %8x %8x %8x %8x ",
                                   (int)stOutParams_Coil.coilsArr[0],
                                   (int)stOutParams_Coil.coilsArr[1],
                                   (int)stOutParams_Coil.coilsArr[2],
                                   (int)stOutParams_Coil.coilsArr[3]);

          stInParams_Coil.coilsArr[0] = ~(stOutParams_Coil.coilsArr[0]);
          stInParams_Coil.coilsArr[1] = ~(stOutParams_Coil.coilsArr[1]);
          stInParams_Coil.coilsArr[2] = ~(stOutParams_Coil.coilsArr[2]);
          stInParams_Coil.coilsArr[3] = ~(stOutParams_Coil.coilsArr[3]);

     } while (loopCount < 3);
}


//   15.8.9.   cHost::MbusReadInputsTable
void mBusReadWriteInput(void)
// =============================
{
     int       startRef;
     int       refCnt;

     int            loopCount = 0;

     MMC_MODBUSREADINPUTS_OUT stOutParams_Input;


     printf("\n\n Function: %s: ", __func__);

     printf("\n ... Going deal with Mbus Inputs EAS ind 1-4 ");
     startRef= MODBUS_UPDATE_START_INDEX;
     refCnt    = MODBUS_UPDATE_CNT;
     loopCount = 0;
     do
     {
          loopCount++;
          cHost.MbusReadInputsTable(startRef, refCnt, stOutParams_Input);
          printf("\n ... Mbus Readed inputs ind 0-3: %8x %8x %8x %8x ",
                    (int)stOutParams_Input.inputsArr[0], (int)stOutParams_Input.inputsArr[1],
(int)stOutParams_Input.inputsArr[2], (int)stOutParams_Input.inputsArr[3]);

          printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
          WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
     } while (loopCount < 3);
}


#define   UL_TO_MBUS_STRUCT(modbus_write_val, aux_ul_p, mod_bus_idx)                     \
     {                                                                                   \
               aux_ul_p = (unsigned long *)(& mbus_write_in.regArr[mod_bus_idx]);   \
               * aux_ul_p = modbus_write_val;                                                 \
     }                                                                                   \
//   16.3.20.  CMMCSingleAxis::GetActualPosition 1164
//   16.3.21.  CMMCSingleAxis::GetActualVelocity 1164
//   16.3.22.  CMMCSingleAxis::GetActualTorque       1165
//   15.6.9.       CMMCConnection::GetGlobalBoolParameter  1353
void RepAxisAActualParamOnMbus(void)
// ================================
{
     MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN      mbus_write_in;
                              /* Auxiliary var for write to ModBus */
     unsigned  long*     ul_p;
     unsigned  long ul_val;
          double    dActul;
```

```
    dActul = AxisA.GetActualPosition();
    ul_val = (unsigned long)dActul;
/* Put value into ModBus write array use 2 index poss (0 & 1).  */
    UL_TO_MBUS_STRUCT(ul_val, ul_p, 0);

/* Remmber about mapping & config !!! */
/* if the motor connected above ehercat it needs apropriate config setting (EAS).   */
    dActul = AxisA.GetActualVelocity();
    ul_val = (unsigned long)dActul;
                            /* Put value into ModBus write array use 2 index poss (2 & 3).  */
    UL_TO_MBUS_STRUCT(ul_val, ul_p, 2);


/* Remmber about mapping & config (see GetActualVelocity) !!!   */
    dActul = AxisA.GetActualTorque();
    ul_val = (unsigned long)dActul;
/* Put value into ModBus write array use 2 index poss (4 & 5).  */
    UL_TO_MBUS_STRUCT(ul_val, ul_p, 4);


/* Index of modbus H.reg. for start write actal values. */
    mbus_write_in.startRef       = MODBUS_STR_REP_ACT_VAL_INDX;
/* Number of indexes to write - each long extend along 2 indexs (2 for Pos, 2 for Vel, 2 for Torq) */
    mbus_write_in.refCnt = 6;
    cHost.MbusWriteHoldingRegisterTable(mbus_write_in) ;

    return ;
}


#define   eCOMM_TYPE_ETHERCAT      1
#define   eCOMM_TYPE_CAN           2
/* Move Axis A and report its Actual parameters on Mbus.                  */
void moveAxisAAndRepActualParamOnMbusPos(void)
// ==============================================
{
                int ind,
                        rt;
    unsigned int   uiState,
                        uiDoneFlg;

    printf("\n\n Function: %s: ", __func__);

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


/* Connection type - CAN/EtherCAT */
    rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);
    if (rt == eCOMM_TYPE_ETHERCAT)
    {
        printf("\n --- Motor connection is via ETHERCAT rembber map & config for get actul param
(Vel. Torq.) ");
    }
    else if (rt == eCOMM_TYPE_CAN)
    {
        printf("\n --- Motor connection is via CAN ");
        if ( mapConfigPdo() )
        {
            printf("\n --- ERROR FAIL IN MAP PDO FOR CAN  !!! ");
        }
    }
    else
    {
        printf("\n --- ERROR UNKNOWN motor connection  !!! ");
    }

    printf("\n ... Start rep AxisA pos on Mbus");
    for(ind=0; ind<3; ind++)
    {
/* Move abs to: 400000, with parameters:         */
/*   Speed 300000, Acc=500000 , Dec=1500000, Jerk=20000000, buffer mode= MC_BUFFERED_MODE (def) */
        AxisA.MoveAbsolute(400000*ind, 300000, 500000, 1500000, 20000000);
        do
```

```
          {
/* Write AxisA pos. to Modbus H.Reg */
               RepAxisAActualParamOnMbus();
               uiState = AxisA.ReadStatus();
               uiDoneFlg = uiState & NC_AXIS_STAND_STILL_MASK;
               if ( ! uiDoneFlg)
               {
                    WAIT_SLEEP_MILLI(40)
               }
          } while(! uiDoneFlg);
     }
     printf("\n ... End rep AxisA pos on Mbus");

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
}


int  mapConfigPdo(void)
// ===================
{
     int  rc;
     unsigned short                AxisARef;


     MMC_CONFIGREGULARPARAMEVENTPDO3_IN CfgReg3In;
    MMC_CONFIGREGULARPARAMEVENTPDO3_OUT CfgReg3Out;

    CfgReg3In.ucEventGroup    = NC_COMM_EVENT_GROUP6;
    CfgReg3In.ucPDOCommParam= PDO_COM_PARAM_SYNC;

     AxisARef = AxisA.GetRef();

    rc = MMC_CfgRegParamEvPDO3Cmd(ComHndl, AxisARef, &CfgReg3In, &CfgReg3Out);
    if(rc != 0)
    {
     printf("\n **** config error %d \n",CfgReg3Out.sErrorID);
/* ... goto lbl_motor_off; */
    }

     return (rc);
}

/*================================ Example functions END ==========================================*/
```

17.12.3 Event Type Definitions

The following table defines the type definitions for the events special cases. Each of the special events provides a range of parameters

Event	Parameters provided
PdoRcvEventCallback	unsigned short usAxisRef, unsigned short usGrpID, UNPDODAT unDat1 - any form of value type included in the PDO data UNPDODAT unDat2 - any form of value type included in the PDO data
HBeateEventCallback	unsigned short usAxisRef
MotionEndEventCallback	unsigned short usAxisRef, bool bResult
EmergencyEventCallback	unsigned short usAxisRef, short sEmcyCode
HomeEndedEventCallback	unsigned short usAxisRef, short sErrCode,...
ModbusWriteEventCallback	Values only
SysErrorEventCallback	Values only
AsyncReplyEventCallback	unsigned short usAxisRef, unsigned short usStatus, unsigned short usError, unsigned short usCobID, unsigned short usLength, unsigned char ucBuffer
TouchProbeEndCallback	unsigned short usAxisRef, long lPosition
NodeErrorEventCallback	unsigned short usAxisRef, unsigned short sErrorID, unsigned short usEmergencyCode
StopOnLimitEventCallback	unsigned short usAxisRef,

Event	Parameters provided
	unsigned short usError, unsigned int uiStatusRegister, unsigned int uiMcsLimitRegister
TableUnderflowEventCallback	unsigned short usAxisRef

17.12.4 ConnectIPC

Creates an IPC connection for WIN32 only. Any +ve integer values accepted for the connection. Refer to the the functions **8.1.24 MMC_GetLastError**, and **8.1.26 MMC_IPCInitConnection**.

unsigned int ConnectIPC(
int iEventMask,
MMC_CB_FUNC fpClbk
)

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

iEventMask

Defined according to the event IDs described in the section **12.20 Events Mask and Enumeration**. Bitwise +ve integer ID.

MMC_CB_FUNC fpClbk

The type definition MMC_CB_FUNC is part of this header file and is similar to the callback prototype described in section 12.21.1, which also contains the fpClbk callback function.

17.12.5 ConnectIPCEx

Creates an IPC connection for WIN32 only. Any +ve integer values accepted for the connection. Refer to the the functions **8.1.24 MMC_GetLastError**, and **8.1.26 MMC_IPCInitConnection**.

unsigned int ConnectIPCEx(
int iEventMask,
MMC_MB_CLBK fpClbk
)

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

iEventMask

Defined according to the event IDs described in the section **12.20 Events Mask and Enumeration**. Bitwise +ve integer ID.

MMC_MB_CLBK fpClbk

The type definition MMC_MB_CLBK is part of this header file and is similar to the callback prototype described in section 12.21.1, which also contains the fpClbk callback function. However, the MMC_MB_CLBK is sa specific type of callback definition used to describe functions with a variable number of parameters.

For code example, refer to the section17.2.1 **17.12.2**.

17.12.6 ConnectRPC

Creates an RPC connection. Any +ve integer values accepted for the connection. Refer to the the functions **8.1.24 MMC_GetLastError**, and **8.1.26 MMC_IPCInitConnection**.

```
unsigned int ConnectRPC(
char* cHostIP,
char* cDestIP,
int iEventMask,
MMC_CB_FUNC fpClbk
)
```

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

cHostIP

Host IP address. Character value in the format of an IP address.

cDestIP

Destination IP address. Character value in the format of an IP address.

iEventMask

Defined according to the event IDs described in the section 12.20 **Events Mask** and Enumeration **on page 1093**. Bitwise +ve integer ID.

MMC_CB_FUNC fpClbk

The type definition MMC_CB_FUNC is part of this header file and is similar to the callback prototype described in section 12.21.1, which also contains the fpClbk callback function.

17.12.7 ConnectRPCEx

Creates an RPC connection. Any +ve integer values accepted for the connection. Refer to the the functions **8.1.24 MMC_GetLastError**, and **8.1.26 MMC_IPCInitConnection**.

```
unsigned int ConnectRPCEx(
char* cHostIP,
char* cDestIP,
int iEventMask,
MMC_MB_CLBK fpClbk
)
```

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

cHostIP

Host IP address. Character value in the format of an IP address.

cDestIP

Destination IP address. Character value in the format of an IP address.

iEventMask

Defined according to the event IDs described in the section **12.20 Events Mask and Enumeration**. Bitwise +ve integer ID.

MMC_MB_CLBK fpClbk

The type definition MMC_MB_CLBK is part of this header file and is similar to the callback prototype described in section 12.21.1, which also contains the fpClbk callback function. However, the MMC_MB_CLBK is sa specific type of callback definition used to describe functions with a variable number of parameters.

17.12.8 SetGlobalBoolParameter

Sets the global Boolean parameters. Refer to the section **5.9.29 MMC_GlobalWriteBoolParameter** for further details.

void SetGlobalBoolParameter(
[long lValue,]
[double dbValue,]
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException)

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

lValue

Input parameter. Any integer value.

dbValue

Array parameter with double value.

MMC_PARAMETER_LIST_ENUM eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3 Axis, Group, Global, Parameters** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values.

Return

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID

Status of the axis.

17.12.9 GetGlobalBoolParameter

Obtain the global Boolean parameters. Refer to the section **5.9.29 MMC_GlobalWriteBoolParameter** for further details.

long GetGlobalBoolParameter(
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException)

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

MMC_PARAMETER_LIST_ENUM eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3 Axis, Group, Global, Parameters** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values.

Return

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section17.2.1 **17.12.2.**

17.12.9.1 Functions Example

```
//    16.6.7. GetGlobalBoolParameter   5.7.12. MMC_GlobalReadBoolParameter
//    16.6.6. SetGlobalBoolParameter     5.7.25. MMC_GlobalWriteBoolParameter

//  16.2.10.  GetBoolParameter
//  16.2.8.   SetBoolParameter
//  CMMCNode::GetBoolParameter   5.7.11. MMC_ReadBoolParameter
//  CMMCNode::SetBoolParameter   5.7.24. MMC_WriteBoolParameter
//  CMMCGroupAxis::GetBoolParameter   5.9.15. MMC_GroupReadBoolParameter
//  CMMCGroupAxis::SetBoolParameter   5.9.17. MMC_GroupWriteBoolParameter


// 16.2.11.  GetParameter
// CMMCGroupAxis::GetParameter 5.9.14. MMC_GroupReadParameter
// CMMCNode::GetParameter  5.7.16. MMC_ReadParameter
// 16.2.9.  SetParameter
// CMMCGroupAxis::SetParameter  5.9.16. MMC_GroupWriteParameter
// CMMCNode::SetParameter     5.7.28. MMC_WriteParameter
```

```
// 16.4.13. GroupReadStatus     5.9.7.  MMC_GroupReadStatusCmd

// 16.6.8.  GetGlobalParameter    5.7.17. MMC_GlobalReadParameter
// accordint to doc. table under 5.3.2:"Real Global Parameters table - Description" - !!! NOT
IN USE !!!
void  SetGetParameters(void)
// ===========================
{
unsigned long     ulong;
 long   lValue1, lValue2, lValue3, lValue4, lValue5;
 double  db1, db2, db3;
 int     iIndex;    /* index into parameters array if axis group */

 iIndex = 0;


printf("\n     %s:", __func__);
      /* Execution time limit before FB become active,     */
          /* when performing EndVelocities Recalcuation [ms]  */
lValue1 = cConn.GetGlobalBoolParameter( MMC_EST_TIME_TO_BE_ACTIVEFB_THRSHLD, iIndex);
     cConn.SetGlobalBoolParameter(1000, MMC_EST_TIME_TO_BE_ACTIVEFB_THRSHLD, iIndex);

        /* GMAS - Drive uf pARAMETER #3          */
lValue1 = AxisA.GetBoolParameter(     MMC_I_COMM_EV_USR_3_PARAM, iIndex);
     AxisA.SetBoolParameter(300, MMC_I_COMM_EV_USR_3_PARAM, iIndex);

/*
* See "Boolean Group Parameters table - Description" (Prg: 5.3.1. & 5.3.2.)
*  Symbol Name      Val   Bitwise Permission
*  ===========      === ==================
*    MMC_AXIS_GROUP_ID_PARAM =  4    5
*    MMC_SPATIAL_OPTION_PARAM = 28   6
*    MMC_END_MOTION_REASON   = 71   5
*    MMC_FB_DEPTH    = 75   5
*    MMC_STATUS_REGISTER      = 91   5
*    MMC_MCS_LIMIT_REGISTER  = 92   5
*/
 lValue1 = Group.GetBoolParameter(   MMC_AXIS_GROUP_ID_PARAM,  0); /* Read Only */
 lValue2 = Group.GetBoolParameter(   MMC_AXIS_GROUP_ID_PARAM,  1); /* Read Only */
 lValue3 = Group.GetBoolParameter(   MMC_AXIS_GROUP_ID_PARAM,  2); /* Read Only */
 lValue4 = Group.GetBoolParameter(   MMC_AXIS_GROUP_ID_PARAM,  3); /* Read Only */
 lValue5 = Group.GetBoolParameter(   MMC_AXIS_GROUP_ID_PARAM,  4); /* Read Only */

 lValue2 = Group.GetBoolParameter(   MMC_END_MOTION_REASON,    iIndex); /* Read Only */
 lValue3 = Group.GetBoolParameter(   MMC_FB_DEPTH,    iIndex); /* Read Only */
 lValue4 = Group.GetBoolParameter(   MMC_SPATIAL_OPTION_PARAM, iIndex);
    Group.SetBoolParameter(1, MMC_SPATIAL_OPTION_PARAM, iIndex); /* Max=1     */

/*
 * See "Real Group Parameters table - Description" (Prg: 5.3.2.)
*  Symbol Name   Val    Bitwise Permission (bit0=Read only(LSB), bit1=Read/Write...)
*  ===========      ====    ==================
*  MMC_MAX_VELOCITY_PARAM      = 15  14
*  MMC_SW_MAX_VELOCITY_PARAM  = 16
*  MMC_SET_ACCELERATION_PARAM  = 18  21
*  MMC_MAX_ACCELERATION_PARAM  = 19     22
*  MMC_SW_MAX_ACCELERATION_PARAM    = 20
*  MMC_SET_DECELERATION_PARAM  = 22     21
*  MMC_MAX_DECELERATION_PARAM  = 23     22
*  MMC_SW_MAX_DECELERATION_PARAM    = 24
*  MMC_MAX_JERK_PARAM      = 26     30
*  MMC_SW_MAX_JERK_PARAM        = 27
*  MMC_F_COMM_EV_USR_1_PARAM  = 35
*/


 db1 = AxisA.GetParameter(  MMC_MAX_VELOCITY_PARAM, iIndex);
```

```
    AxisA.SetParameter(20000000.0,MMC_MAX_VELOCITY_PARAM,     iIndex);

 db2 = AxisB.GetParameter(  MMC_MAX_ACCELERATION_PARAM,iIndex);
  AxisB.SetParameter(22200000.0, MMC_MAX_ACCELERATION_PARAM,iIndex);
 db3 = AxisB.GetParameter(  MMC_MAX_ACCELERATION_PARAM,iIndex); /* Not necessary (Test, E.g.
etc...) */

 db1 = Group.GetParameter(  MMC_MAX_DECELERATION_PARAM,0);
    Group.SetParameter(3000000.0,MMC_MAX_DECELERATION_PARAM,0);

 db2 = Group.GetParameter(  MMC_MAX_VELOCITY_PARAM, 1);
    Group.SetParameter(3330000.0,MMC_MAX_VELOCITY_PARAM,     1);

 ulong = Group.GroupReadStatus();
 if(ulong & NC_GROUP_ERROR_STOP_MASK)
 {
  Group.GroupReset();
 }

 Group.GroupDisable();
 do
 {
  ulong = Group.GroupReadStatus();
 } while (!(ulong & NC_GROUP_DISABLED_MASK));

 Group.GroupEnable();
 do
 {
  ulong = Group.GroupReadStatus();
 } while (!(NC_GROUP_STANDBY_MASK & ulong));

}


void  SetGetGroupParam(void)
// ===========================
{
 int     iIndex;   /* index into parameters array if axis group */

 iIndex = 0;

MMC_READGROUPOFPARAMETERSMEMBER GroupOfPrmRed[GROUP_OF_PARAMETERS_MAXIMUM_SIZE] =
    {
     {MMC_MAX_VELOCITY_PARAM, iIndex, AxisARef, 0},
     {MMC_MAX_ACCELERATION_PARAM,iIndex, AxisARef, 0},
     {MMC_MAX_DECELERATION_PARAM,iIndex, AxisARef, 0},
     {MMC_MAX_JERK_PARAM,     iIndex, AxisARef, 0},
     {MMC_SW_MAX_JERK_PARAM,  iIndex, AxisARef, 0}
    };
 MMC_WRITEGROUPOFPARAMETERSMEMBER    GroupOfPrmWrt[GROUP_OF_PARAMETERS_MAXIMUM_SIZE] =
    {
     {11110000, MMC_MAX_VELOCITY_PARAM,   iIndex, AxisARef, 0,0,0},
     {22220000, MMC_MAX_ACCELERATION_PARAM,    iIndex, AxisARef, 0,0,0},
     {33330000, MMC_MAX_DECELERATION_PARAM,    iIndex, AxisARef, 0,0,0},
     {44440000, MMC_MAX_JERK_PARAM,   iIndex, AxisARef, 0,0,0},
     {55550000, MMC_MAX_VELOCITY_PARAM,   iIndex, AxisBRef, 0,0,0}
    };
 double GroupOfPrmRet[GROUP_OF_PARAMETERS_MAXIMUM_SIZE];

printf("\n     %s:", __func__);
         /* Read the startup def. Val.*/
 AxisA.ReadGroupOfParameters  (GroupOfPrmRed, 5, GroupOfPrmRet);
       /* Set and change one value (above init array) */
 AxisA.WriteGroupOfParametersImmediate(GroupOfPrmWrt, 1);
       /* Only for demo.. - for check the new setting is in effect */
 AxisA.ReadGroupOfParameters  (GroupOfPrmRed, 5, GroupOfPrmRet);
```

```
        /* Change one of value in above init array */
 GroupOfPrmRed[1].usAxisRef = AxisBRef;
        /* Write  param - one is updated, (reff by AxisB).  */
 AxisB.WriteGroupOfParametersImmediate(GroupOfPrmWrt, 5);
    /* read 5 param. Expec: 11110000,22220000,33330000,44440000,55550000    */
 AxisB.ReadGroupOfParameters  (GroupOfPrmRed, 5, GroupOfPrmRet);
   /* read 5 param. Expec: 11110000,22220000,33330000,44440000,55550000    */
 AxisA.ReadGroupOfParameters  (GroupOfPrmRed, 5, GroupOfPrmRet);

          /* AxisB default Val are diff. from AxisA... */
 GroupOfPrmWrt[0].dbValue = 11111100.0;    /* MMC_MAX_VELOCITY_PARAM      */
 GroupOfPrmWrt[2].dbValue = 33331100.0;   /* MMC_MAX_DECELERATION_PARAM  */
 GroupOfPrmWrt[4].dbValue = 55551100.0;   /* MMC_MAX_VELOCITY_PARAM  */
}
```

17.12.10 GetGlobalParameter

Obtain the global parameters. Refer to the section **5.9.33 MMC_GlobalWriteParameter** for further details.

double GetGlobalParameter(
MMC_PARAMETER_LIST_ENUM eNumber,
int iIndex
) throw (CMMCException)

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

MMC_PARAMETER_LIST_ENUM eNumber

Number of the parameter. One can also use symbolic parameter names, which are declared as VAR CONST.

Refer to the section **5.3 Axis, Group, Global, Parameters** for the appropriate integer parameter to be used as enumerator.

iIndex

Array index parameter (only relevant for array situations). Any +ve integer values.

Return

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.12.11 SetIsToLoadGlobalParams

Defines a flag whether to load or not, the global parameters, when updating the global parameters from a file to the Maestro. Refer to the section **8.1.37 MMC_SetIsToLoadGlobalParams** for a detailed explanation.

```
void SetIsToLoadGlobalParams(
unsigned char ucVal
) throw (CMMCException)
```

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

ucVal

The function recieves either 0 (not required to load the set global parameters) or 1 (required to load the set global parameters). +ve integer value

Return

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.12.12 SetHeartBeatConsumer

Sets the consumer heartbeat as an event to the user. Refer to the section **16.5.22 MMC_SetHeartBeatConsumer for a detailed explanation.**

```
void SetHeartBeatConsumer(
unsigned int uiHeartbeatTimeFactor
) throw (CMMCException)
```

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

uiHeartbeatTimeFactor

Heart beat time factor is a multiple of 1 ms. The calculation of the basic cycle time (predetermined in the Resource file), multiplied by this heartbeat time factor, and 1 ms, will set the Heartbeat time.

Values accepted are:

0, not in use

>0, any +ve value

Return

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

For code example, refer to the section17.2.1 **17.12.12**.

17.12.13 CallbackFunc

Defines the callback function to be an integer with +ve values.

```
int CallbackFunc(
unsigned char*, short, void*
)
```

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

unsigned char, short, void**

Character, short, or void with a +ve value

For code example, refer to the section17.2.1 **17.12.2**.

17.12.14 RegisterEventCallback

Registers the event callback for specific type callbacks. Refer to the section **12.21 Asynchronous Events Callback** for a detailed explanation of callback events.

```
void RegisterEventCallback(
MMC_EVENT_ENUM eClbType,
void * pfClbk
)
```

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

MMC_EVENT_ENUM eClbType

The parameter MMC_EVENT_ENUM is a specialist enumerator variable with values:

MMCPP_PDORCV, MMCPP_HBEAT	0
MMCPP_MOTIONENDED	1
MMCPP_EMCY	2
MMCPP_ASYNC_REPLY	3
MMCPP_HOME_ENDED	4
MMCPP_MODBUS_WRITE	5
MMCPP_TOUCH_PROBE_ENDED	6
MMCPP_NODE_ERROR	7
MMCPP_STOP_ON_LIMIT	8
MMCPP_TABLE_UNDERFLOW	9

The eClbType parameter is called according to the above enumerator, and has values:

PdoRcvEventCallback
HBeateEventCallback
MotionEndEventCallback
EmergencyEventCallback
HomeEndedEventCallback
ModbusWriteEventCallback
SysErrorEventCallback
AsyncReplyEventCallback
TouchProbeEndCallback
NodeErrorEventCallback
StopOnLimitEventCallback
TableUnderflowEventCallback

The enumerator value must complement the callback type. Refer to the table in section **17.12.3 Event Type Definitions** for details of the Event type definitions.

pfClbk

Points to the callback function with no value returned.

For code example, refer to the section17.2.1 **17.12.2**.

17.12.15 RegisterSyncTimerFunction

```
void RegisterSyncTimerFunction(
MMC_SYNC_TIMER_CB_FUNC func,
unsigned short usSYNCTimerTime
)
```

Source GMAS\includes\CPP\MMCConnection.h

.NET Definition

Function Parameters

MMC_SYNC_TIMER_CB_FUNC func

[IN] Points to the callback function MMC_SYNC_TIMER_CB_FUNC using MMC_CreateSYNCTimer.

usSYNCTimerTime

[IN] Defines the time between which a synchronization message is sent as an event.

17.13 The CMMCNetwork class

The class MMCNetwork wraps the network communication functions detailed in the section **16.1 Network Function Blocks on page 1167**. The diagram **in Figure 17-18** describes the heirarchial structure of the classes and type definitions associated with the MMCNetwork.

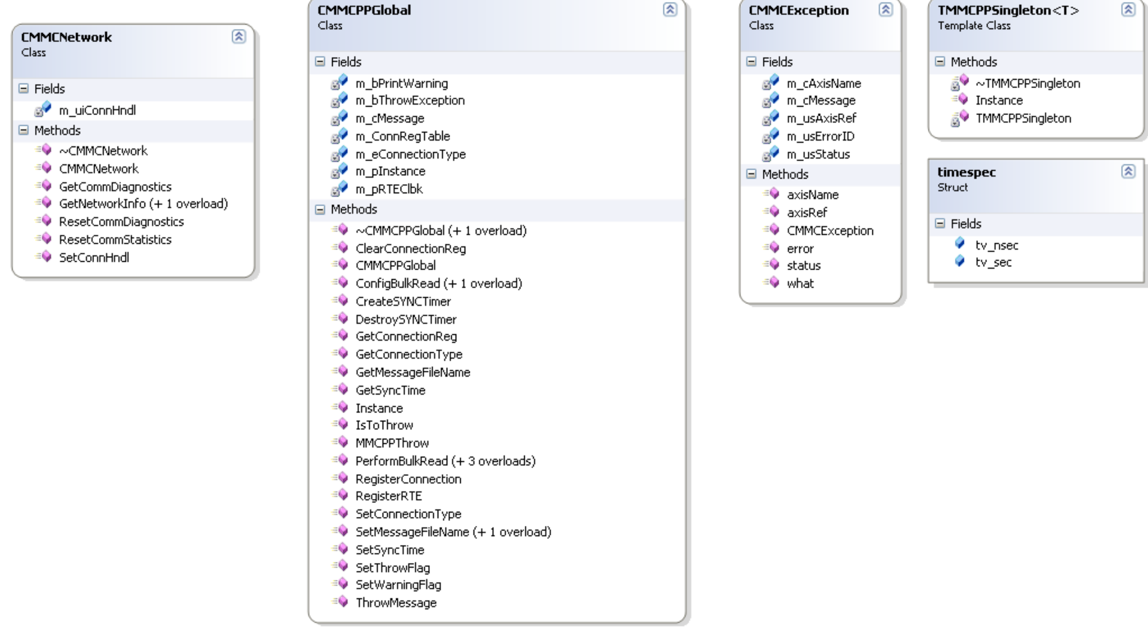


Figure 17-18 MMCNetwork class diagram

The class MMCNetwork retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

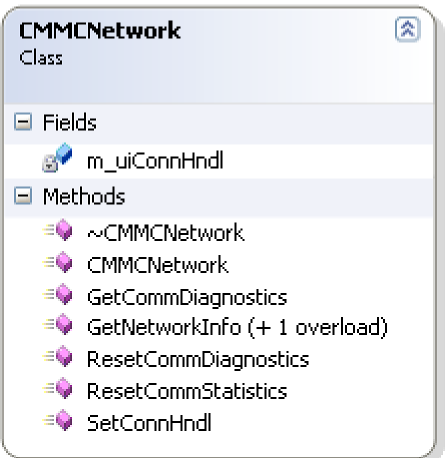


Figure 17-19 Fields and methods of the MMCNetwork class

The detailed class view shown in **Figure 17-19** describes the fields and methods associated with the MMCNetwork class. It should be noted that Protected and Private and Protected functions together with their operations, should be transparent to the user, and are not for general application by the user.

17.13.1 CMMCNetwork Class Functions Code Example

```
/*
============================================================================
Collection of Gmas API functions (Set #3)
Examples for document: "G-MAS Administrative and Motion API.pdf".
12Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection          gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCSingleAxis          AxisA,    AxisB;
     unsigned short               AxisARef, AxisBRef;

     CMMCGroupAxis           Group;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

    char  * delimit = "===========================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroConnection(int);
     void      ConnectionTypeAndNum(void);
     void      SendReciveFromEthercat(int NumAmp);
     void      SetGetDefDigOutput(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
     int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus);
     void      EndMotionEventCB(unsigned short usAxisRef);
     void ModbusWrite_Received();
     void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
```

```
/*================================ Administration functions STR ===================================*/

int      main(int)
// ==============
{
     int  trace = 1;

     printf("\n %s", delimit);
     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

     try
     {
          SnroConnection(trace++);
          SnroMoveAbsolute(trace++);
          SnroEnableDisableMotionEndedEvent(trace++);
          SnroDepthName(trace++);

     }
     catch (CMMCException excp)
     {
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          exit(0);
     }

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


int      WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//======================================================================
{
                int end_of = 0;
                int iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
          end_of = 1;
/* Read Axis Status command server for specific Axis */
          ulState = sng_axis->ReadStatus();
          if (!(ulState & break_state))
          {
               end_of = 0;

               WAIT_SLEEP_MILLI(20)
          }
     }

//        MMC_SHOWNODESTAT_IN  showin;
//        MMC_SHOWNODESTAT_OUT showout;
//        MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

     return 0;
}


//   15.5.2.   RegisterRTE    Page 1274
void initAdminSingleAxis(void)
// ==============================
{
     int       iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;

/* CallbackFunc in ConnectIPCEx call if there    */
```

```
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Put Null param Val for no CallbackFunc        */
/*   ComHndl = gConn.ConnectIPCEx(iEventMask, NULL); */
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */


/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk     = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute     = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


void initAdminMultiAxis()
// =========================
{
//  Source class:
//       MMC_CONNECT_HNDL        ComHndl;
//       CMMCSingleAxis          AxisA,    AxisB;
//       CMMCGroupAxis           Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// =============================
{
     MMC_CloseConnection(ComHndl) ;
}


     void endAdminMultiAxis(void)
// ================================
{
//  Source class:
//       CMMCGroupAxis  Group;

     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
     Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================ Administration functions END ====================================*/

/*================================= Scenario functions STR =========================================*/
void SnroMoveAbsolute(int trace)
// ================================
{
```

```
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    MoveAbsoluteMoves();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void SnroEnableDisableMotionEndedEvent(int trace)
// ================================================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);
                                /* Register the callback function for Modbus and Emergency: */
    gConn.RegisterEventCallback(MMCPP_MODBUS_WRITE,(void*)ModbusWrite_Received) ;
    gConn.RegisterEventCallback(MMCPP_EMCY,(void*)Emergency_Received) ;

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    EnableDisableMotionEndedEvent();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
    endAdminSingleAxis();

    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

//   15.4.14.  GroupEnable    Page 1208
//   15.4.15.  GroupDisable   Page 1209
void SnroDepthName(int trace)
// ============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    initAdminMultiAxis();

   AxisA.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

   AxisB.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

    Group.GroupEnable();

    DepthName();

    Group.GroupDisable();

    AxisB.PowerOff(MC_BUFFERED_MODE);
    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminMultiAxis();
    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
```

```
void SnroConnection(int trace)
// ==============================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();

     ConnectionTypeAndNum();

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

/*================================= Example functions STR =========================================*/


void EnableDisableMotionEndedEvent(void)
// ========================================
{
     int  loopInd;

printf("\n Function: %s:", __func__);
     for (loopInd = 0; loopInd < 2; loopInd++)
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}


//   16.4.14.  GroupEnable        1208
//   16.4.15.  GroupDisable   1209
void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

     iVal3 = Group.GetGroupAxisByName("v01");     /* Expected 256 */
```

```
     printf("\n +++++ Reff: a01=%d a02=%d, v01=%d +++++", iVal1, iVal2, iVal3);
/*
*    iVal2 = Group.GetGroupAxisByName("v02");
*/
}


void MoveAbsoluteMoves(void)
// ============================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:         */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0       */
                              /* update default speed to 5000000          */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters             */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity       */
                              /* v=100000    which become the new def V        */
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0        */
                              /* new def v=80000                              */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:     */
                              /*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=             */
/* MC_BUFFERED_MODE (def)     */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
/* Move abs to 350000 with parameters from  */
/* above command which become the default:  */
/*        Speed 500000, Acc=1000000                */
/*        Dec=1500000, Jerk=20000000,        */
/*        buffer mode=MC_BUFFERED_MODE (def)  */
     AxisA.MoveAbsolute(350000);
}


int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{

    printf("\n *********** STR Func: %s *********** ", __func__);

                              /* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
```

```
            printf("\n Underflow event received ");
            break ;
        case MODBUS_WRITE_EVT:
            printf("\n ModBus Write event received ");
            break ;
        case TOUCH_PROBE_ENDED_EVT:
            printf("\n Touch Probe event received ");
            break ;
        default:
            printf("\n Default.... Whatever arrived event received ");
            break;
        }

       printf("\n *********** END Func: %s *********** ", __func__);
        fflush(stdout); fflush(stderr);

        return 1 ;
}

int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
//
=====================================================================================================
{
        printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
                msg, usAxisRef, sErrorID, usStatus);
        fflush(stdout); fflush(stderr);

        MMC_CloseConnection(uiConnHndl);
        exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
        printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
        printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
        fflush(stdout); fflush(stderr);
}


/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ==========================
{
        printf("\n %s Received ", __func__) ;
        fflush(stdout); fflush(stderr);
}


                              /* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ===============================================================
{
        printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
        fflush(stdout); fflush(stderr);
}
```

```
//   15.6.11. SetHeartBeatConsumer      1248
//   15.7.4.  GetNetworkInfo            1337
enum
{
     eCOMM_TYPE_NONE = 0,
     eCOMM_TYPE_ETHERCAT,
     eCOMM_TYPE_CAN
};


void ConnectionTypeAndNum(void)
// ===============================
//
{
     unsigned int                  uiHeartbeatTimeFactor;
     int                                rt;
     int                                NumFoundAmp;
     char *                             cErrotStr;
     CMMCNetwork                        CNet;
     MMC_NETWORKINFO_OUT           CanOutParams;       // Can drv connection
     MMC_GETCOMMSTATISTICS_OUT     EthCatOutParams;    // Ethcat drv connection


     printf("\n Function: %s ", __func__);

     CNet.SetConnHndl(ComHndl);
                              /* Connection type - CAN/EtherCAT */
     rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);

     if (rt == eCOMM_TYPE_ETHERCAT)
     {
                              /* !!! ETERCAT !!!*/
          rt = CNet.GetCommStatistic(EthCatOutParams);
          if (rt != 0)
          {
               cErrotStr = "EtherCat failed, get statistic";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = (int)EthCatOutParams.usNumOfSlaves;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through ETERCAT net. ",
NumFoundAmp);

                              /* Send and recive from UDP soket to driv */
          SendReciveFromEthercat(NumFoundAmp);
     }
     else if (rt == eCOMM_TYPE_CAN)
     {
/* !!! CAN !!!*/
/* !!!  hard bit should be set in resource file*/
/* take from the resource file and actual connected...  */
          uiHeartbeatTimeFactor = 1;    /* On for Every Cycle */
          gConn.SetHeartBeatConsumer(uiHeartbeatTimeFactor);

         rt = CNet.GetNetworkInfo  (CanOutParams);
          if (rt != 0)
          {
               cErrotStr = "Can failed, get NetworkInfo";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = CanOutParams.iNumOfActiveNodes;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through CAN net. ", NumFoundAmp);
     }

/* Get & Set Default Mapping Digital Output */
     SetGetDefDigOutput();

     return;

ConnectionTypeAndNum_exit_err:
     printf("\n>>> %s: *** %s %d ", __func__, cErrotStr, rt);
     return;
}
```

```
/* Should create sockets according to actual number of amplifire (param NumAmp)     */
/* ...for demo (examples etc...) assume at least two Amp. exist.                    */
//   16.18.1.  Create         1302
//   16.18.2.  SendTo         1303
//   16.18.3.  ReceiveFrom    1304
//
//   16.18.11. ElmoGetArray        1310
//   16.18.12. ElmoGetParameter    1311

void SendReciveFromEthercat(int NumAmp)
// ======================================
{
     int            rt_val = 0;
    int             iBv;
     bool           bWait;
     float          fValue;
                         /* IPC type of app. */
     char           sAxisName[50] ;
     CMMCUDP    cUDP1,
                    cUDP2;
     CMMCEoE        gEoe;

     printf("\n Function: %s ", __func__);

     rt_val = cUDP1.Create("192.168.1.5", 5001, 0); /* Ip of first drive (Gmas "last part IP" + 1) */
     rt_val = cUDP1.SendTo("vr\r", 3);
     rt_val = cUDP1.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 0 Version: <%s> ", sAxisName);

     rt_val = cUDP2.Create("192.168.1.6", 5001); /* Ip of second Gmas     */
     rt_val = cUDP2.SendTo("vr\r", 3) ;
     rt_val = cUDP2.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 1 Version: <%s> ", sAxisName);


     rt_val = gEoe.Connect("192.168.1.5", 5001, bWait);
     if (bWait == true)
     {
          usleep(10000);                         /* Wait 10 mili */
          if (gEoe.IsWritable() == false)
          {
               printf("\n>>> %s: *** Amp (Drv) connection it not writable... ", __func__);
          }
     }
/* AN[6] is command for get PsHv   */
/* return 0 on succceed otherwise 1.*/
     rt_val = gEoe.ElmoGetArray("AN", 6, fValue);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get EthCat Driver psHv AN[6] ", __func__);
     }
     else
     {
          printf("\n>>> Max Power Supply High Voltage for this driver (ip=192.168.1.5) is %3.1f ",
fValue);
     }

/* BV - Maximum Motor DC Voltage (return int)    */
/* return 0 on succeed, otherwise 1.              */
     rt_val = gEoe.ElmoGetParameter("BV", iBv);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get Bus Driver HV ", __func__);
     }
     else
     {
          printf("\n>> Actual Bus Driver (ip=192.168.1.5) Hv is %d ", iBv);
     }

     gEoe.Close();
}
```

```
//   16.3.27.  GetDigOutputs32Bit  1168
//   16.3.29.  SetDigOutputs32Bit  1169
                              /* Get & Set Default Mapping Digital Output */
void SetGetDefDigOutput(void)
// =============================
{
unsigned long       ulDigOutputs32bit;



     printf("\n Function: %s ", __func__);

                              /* Read Digital output group 0 state */
     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: B4 action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned 
int)ulDigOutputs32bit);


/* Change specific bit state of digital Output group 0 */
     if ((ulDigOutputs32bit & 0x10000) != 0x00000)
     {
/* ReSet specific bit of Digital output group 0 to state "0" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit & 0xfffeffff);   /* E.g: disconnedt ps... */
     }
     else
     {
/* Set specific bit of Digital output group 0 to state "1" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit | 0x10000);     /* E.g: connect ps... */
     }


     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: Aft action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned 
int)ulDigOutputs32bit);
}

/*================================= Example functions END ============================================*/

/*================================= Output STR ============================================*/
#ifdef PROGRAM_OUTPUT
#endif /* PROGRAM_OUTPUT */
/*================================= Output END ============================================*/
```

17.13.2 GetCommDiagnostics ResetCommDiagnostics

Refer to the section **16.7.10 MMC_GetEthercatCommStatistics and 16.7.14 MMC_ResetCommDiagnostics on page 1367 - 1386** for details of the description, scope, and communication mode.

```
void GetCommDiagnostics(
MMC_GETCOMMDIAGNOSTICS_OUT& stOutParams
) throw (CMMCException);
```

```
void ResetCommDiagnostics(
MMC_RESETCOMMDIAGNOSTICS_OUT& stOutParams
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNetwork.h

.NET Definition

Function Parameters

stOutParams

Defined output parameters function of MMC_GETCOMMDIAGNOSTICS_OUT, and MMC_RESETCOMMDIAGNOSTICS_OUT respectively.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

17.13.3 ResetCommStatistics

Refer to the section **16.7.15 MMC_ResetCommStatistics on page 1389** for details of the description, scope, and communication mode.

```
void ResetCommStatistics(
MMC_RESETCOMMSTATISTICS_OUT& stOutParams
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNetwork.h

.NET Definition

Function Parameters

stOutParams

Defined output parameters function of MMC_RESETCOMMSTATISTICS_OUT.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name Axis reference Error ID
Status of the axis.

17.13.4 GetNetworkInfo

Refer to the section **16.1.7 MMC_NetworkInfo on page 1186** for details of the description, scope, and communication mode.

```
int GetNetworkInfo(
MMC_GETCOMMSTATISTICS_OUT& stOutParams
) throw (CMMCException);
```

```
int GetNetworkInfo(
MMC_NETWORKINFO_OUT& stOutParams
) throw (CMMCException);
```

Source GMAS\includes\CPP\MMCNetwork.h

.NET Definition

Function Parameters

stOutParams

Defined output parameters function of MMC_GETCOMMSTATISTICS_OUT and MMC_NETWORKINFO_OUT.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.13.1**.

17.14 The CMMCHostComm class

The class CMMCHostComm wraps the host communication functions detailed in the section **16.2 Host Communication on page 1212**. The class CMMCHostComm retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

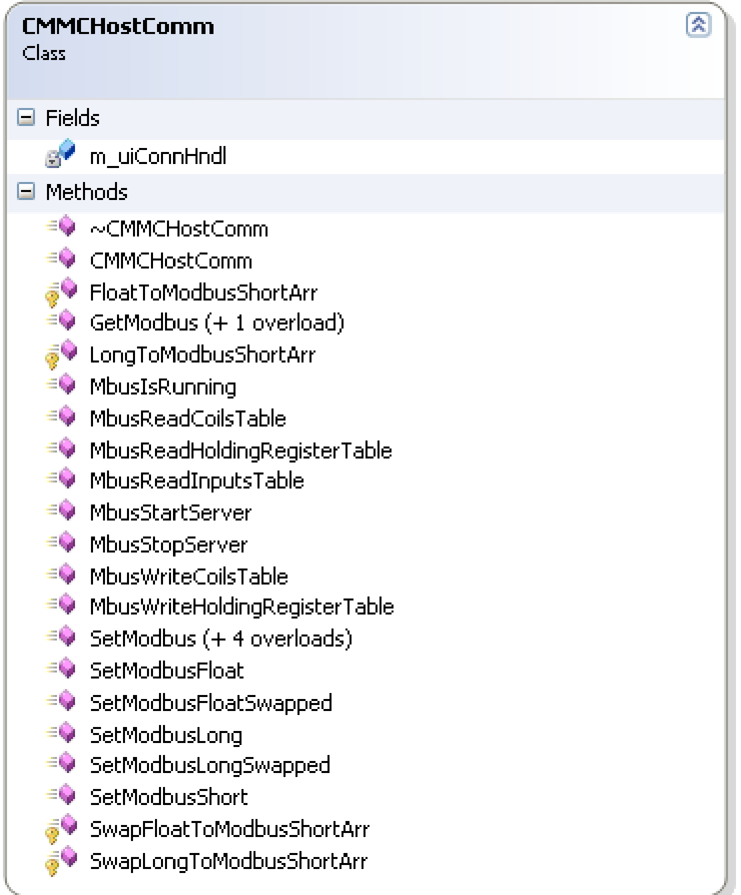


Figure 17-20 Fields and methods of the CMMCHostComm class

The detailed class view shown in **Figure 17-20** describes the fields and methods associated with the CMMCHostComm class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.14.1 CMMCHostComm Class Functions Code Example 1

```
/*
============================================================================
Collection of Gmas API functions (Set #2)
Examples for document: "G-MAS Administrative and Motion API.pdf".
09Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"
#define STAR_DIM                        2


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection          gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCHostComm            cHost ;

     CMMCSingleAxis          AxisA,    AxisB;
     unsigned short               AxisARef, AxisBRef;

     CMMCGroupAxis            Group;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

    char  * delimit = "==========================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroMbusfunc(int);
     void mBusReadWriteHoldReg(void);
     void mBusReadWriteCoil(void);
     void mBusReadWriteInput(void);
     void moveAxisAAndRepActualParamOnMbusPos(void);
     void      RepAxisAActualParamOnMbus(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
```

```
     int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus);
     void      EndMotionEventCB(unsigned short usAxisRef);
     void ModbusWrite_Received();
     void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
     int       mapConfigPdo(void);


/*================================= Administration functions STR ===================================*/
int       main(int)
// ==============
{
     int  trace = 1;

     printf("\n %s", delimit);
     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

     try
     {
          SnroMoveAbsolute(trace++);
          SnroEnableDisableMotionEndedEvent(trace++);
          SnroDepthName(trace++);

          SnroMbusfunc(trace++);
     }
     catch (CMMCException excp)
     {
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          exit(0);
     }

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


//   15.11.3.  CMMCNode::ReadStatus     1374
int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//=======================================================================
{
               int end_of = 0;
               int iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
          end_of = 1;
/* Read Axis Status command server for specific Axis */
          ulState = sng_axis->ReadStatus();
          if (!(ulState & break_state))
          {
               end_of = 0;

               WAIT_SLEEP_MILLI(20)
          }
     }

//        MMC_SHOWNODESTAT_IN  showin;
//        MMC_SHOWNODESTAT_OUT showout;
//        MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

     return 0;
}


// 15.6.5.     CMMCConnection::ConnectIPCEx  1335
```

```
// 15.6.13.    CMMCConnection::CallbackFunc  1345
void initAdminSingleAxis(void)
// ==============================
{
     int iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;



/* CallbackFunc in ConnectIPCEx call if there    */
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */

/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk     = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute     = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


//   15.4.21.  CMMCGroupAxis::AddAxisToGroup
void initAdminMultiAxis()
// =========================
{
//   MMC_CONNECT_HNDL        ComHndl;
//   CMMCSingleAxis          AxisA,    AxisB;
//   CMMCGroupAxis           Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// ============================
{
     MMC_CloseConnection(ComHndl) ;
}


//   15.4.5.   CMMCGroupAxis::RemoveAxisFromGroup 1197
     void endAdminMultiAxis(void)
// ================================
{

//   CMMCGroupAxis            Group;
```

```
    Group.RemoveAxisFromGroup(NC_NODE_1_ID);
    Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================= Administration functions END ====================================*/


/*================================= Scenario functions STR ==========================================*/
void SnroMoveAbsolute(int trace)
// ================================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    MoveAbsoluteMoves();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


//   15.6.14.  CMMCConnection::RegisterEventCallback  1358
void SnroEnableDisableMotionEndedEvent(int trace)
// =================================================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    EnableDisableMotionEndedEvent();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
    endAdminSingleAxis();

    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


void SnroDepthName(int trace)
// =============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    initAdminMultiAxis();

   AxisA.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

   AxisB.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

    Group.GroupEnable();

    DepthName();

    Group.GroupDisable();

    AxisB.PowerOff(MC_BUFFERED_MODE);
    AxisA.PowerOff(MC_BUFFERED_MODE);
```

```
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

     endAdminMultiAxis();
     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}


//   15.8.6.   cHost::MbusIsRunning     1389

//   15.8.2.   cHost::MbusStartServer   1386

//   15.8.3.   cHost::MbusStopServer    1387

void SnroMbusfunc(int trace)
// ============================
{
static    bool      mBusServerState;


     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
/* Starting MBus server           */
/* map read/write user req. to Mbus    */
/* area & responsible to Mbus commu.*/
     mBusServerState = cHost.MbusIsRunning(ComHndl);
     if (mBusServerState)
     {
          printf("\n ... ModBus Server is already running ");
     }
     else
     {
          printf("\n ... ModBus Server is NOT running - activate it ");
     }
/* Call to StartServer even if it   */
/* already running, it not start    */
/* the server because it alrady so  */
/* but for initialize init struct */
     cHost.MbusStartServer(ComHndl, 1) ;
                              /* Mode Bus functions examples.        */
     mBusReadWriteHoldReg();
/* Move Axis A and report it poss.  */
/* on Mbus.*/
     moveAxisAAndRepActualParamOnMbusPos();
     mBusReadWriteCoil();
     mBusReadWriteInput();

/* If I was the one who activate the   */
/* Mbus server I stop it when I finish.*/
     if (! mBusServerState)
     {
/* Stop MBus server */
          cHost.MbusStopServer();
          printf("\n ... Stop ModBus Server ");
     }
     else
     {
          printf("\n ... Left ModBus Server running ");
     }

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
/*================================= Scenario functions END ==========================================*/



/*================================= Example functions STR ==========================================*/

void EnableDisableMotionEndedEvent(void)
```

```
// ========================================
{
     int  loopInd;

printf("\n Function: %s:", __func__);
     for (loopInd = 0; loopInd < 2; loopInd++)
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}

void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

// iVal3 = AxisA.GetAxisByName("A01"); /* It case sensitive - Not define - exception... */

     iVal1 = Group.GetGroupAxisByName("v01");    /* Expected 256 */
/*
 *   iVal2 = Group.GetGroupAxisByName("v02");
 */
}


void MoveAbsoluteMoves(void)
// ============================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:    */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0 */
/* update default speed to 5000000     */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters       */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity */
/* v=100000    which become the new def V    */
```

```
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0  */
/* new def v=80000                     */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:    */
/*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=           */
/* MC_BUFFERED_MODE (def)    */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
/* Move abs to 350000 with parameters from  */
/* above command which become the default:  */
/*       Speed 500000, Acc=1000000          */
/*       Dec=1500000, Jerk=20000000,        */
/*       buffer mode=MC_BUFFERED_MODE (def)  */
     AxisA.MoveAbsolute(350000);
}


//   15.6.13.  CMMCConnection::CallbackFunc
int      CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{
//   int  ind;

    printf("\n *********** STR Func: %s *********** ", __func__);

//    printf("\n");
//    for (ind=0; ind < recvBufferSize; ind++)
//    {
//        printf("   recvBuffer[%d]=%d", ind, recvBuffer[ind]);
//    }
//    printf("\n");

/* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
          printf("\n Underflow event received ");
          break ;
     case MODBUS_WRITE_EVT:
          printf("\n ModBus Write event received ");
          break ;
     case TOUCH_PROBE_ENDED_EVT:
          printf("\n Touch Probe event received ");
          break ;
     default:
          printf("\n Default.... Whatever arrived event received ");
          break;
     }
```

```
    printf("\n *********** END Func: %s *********** ", __func__);
     fflush(stdout); fflush(stderr);

     return 1 ;
}


//
// Needs registeration, E.g:  CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);
//
int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
//
===================================================================================================
{
     printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
             msg, usAxisRef, sErrorID, usStatus);
     fflush(stdout); fflush(stderr);

     MMC_CloseConnection(uiConnHndl);
     exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
     printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
     printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
     fflush(stdout); fflush(stderr);
}

/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ===========================
{
     printf("\n %s Received ", __func__) ;
     fflush(stdout); fflush(stderr);
}

/* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ==============================================================
{
     printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
     fflush(stdout); fflush(stderr);
}


#define   MODBUS_UPDATE_START_INDEX              0
#define   MODBUS_UPDATE_CNT                 4
                              /* WRITE INDEXS extend along two indexs */
#define MODBUS_STR_REP_ACT_VAL_INDX          (MODBUS_UPDATE_START_INDEX+MODBUS_UPDATE_CNT)
#define   MBUS_WAIT_FOR_USER                     30000

//   15.8.4.   cHost::MbusReadHoldingRegisterTable    1388
//   15.8.5.   cHost::MbusWriteHoldingRegisterTable   1389
void mBusReadWriteHoldReg(void)
// ==============================
{
          int       startRef;
          int       refCnt;
          int            loopCount = 0;

     MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN      mbus_write_in;
     MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT      mbus_read_out;

     printf("\n\n Function: %s: ", __func__);

     memset(mbus_write_in.regArr,0x0, 250) ;

     startRef = MODBUS_UPDATE_START_INDEX;
     refCnt   = MODBUS_UPDATE_CNT;
```

```
     mbus_write_in.startRef        = startRef;
     mbus_write_in.refCnt          = refCnt;
     mbus_write_in.regArr[0]  = 0xf0f0;
     mbus_write_in.regArr[1]  = 0x0f0f;
     mbus_write_in.regArr[2]  = 0x5a5a;
     mbus_write_in.regArr[3]  = 0xa5a5;

     printf("\n ... You can activate the EAS and look on EAS Mbus reg 1-4 ");

     printf("\n ... Going deal with Mbus H.Reg. EAS ind 1-4 ");
     loopCount = 0;
     do
     {
          loopCount++;
/* write into mBus Holding register */
/* data on reg. ind 0-3            */
          cHost.MbusWriteHoldingRegisterTable(mbus_write_in);
          printf("\n ... Mbus Write  H.Reg ind 0-3: %8x %8x %8x %8x ",
                                             (int)mbus_write_in.regArr[0],
                                             (int)mbus_write_in.regArr[1],
                                             (int)mbus_write_in.regArr[2],
                                             (int)mbus_write_in.regArr[3]);
          printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);

          WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
/* read mbus H.Reg start from            */
/* location startRef along refCnt reg*/

          cHost.MbusReadHoldingRegisterTable(startRef, refCnt, mbus_read_out);
          printf("\n ... Mbus Readed H.Reg ind 0-3: %8x %8x %8x %8x ",
                                              (int)mbus_read_out.regArr[0],
                                              (int)mbus_read_out.regArr[1],
                                              (int)mbus_read_out.regArr[2],
                                              (int)mbus_read_out.regArr[3]);

          mbus_write_in.regArr[0] = ~(mbus_read_out.regArr[0]);
          mbus_write_in.regArr[1] = ~(mbus_read_out.regArr[1]);
          mbus_write_in.regArr[2] = ~(mbus_read_out.regArr[2]);
          mbus_write_in.regArr[3] = ~(mbus_read_out.regArr[3]);

     } while (loopCount < 3);
}

//   15.8.7.   cHost::MbusReadCoilsTable     1390
//   15.8.8.   cHost::MbusWriteCoilsTable    1391
void mBusReadWriteCoil(void)
// ============================
{
          int  startRef;
          int  refCnt;

          int  loopCount = 0;

     MMC_MODBUSWRITECOILS_IN       stInParams_Coil;
     MMC_MODBUSREADCOILS_OUT       stOutParams_Coil;


     printf("\n\n Function: %s: ", __func__);

     startRef = MODBUS_UPDATE_START_INDEX;
     refCnt    = MODBUS_UPDATE_CNT;
     stInParams_Coil.startRef= startRef;
     stInParams_Coil.refCnt    = refCnt;
     stInParams_Coil.coilsArr[0] = (char)0xf0;
     stInParams_Coil.coilsArr[1] = (char)0x0f;
     stInParams_Coil.coilsArr[2] = (char)0x5a;
     stInParams_Coil.coilsArr[3] = (char)0xa5;

     printf("\n ... Going deal with Mbus Coils EAS ind 1-4 ");
     loopCount = 0;
     do
     {
          loopCount++;
```

```
/* write into mBus Coil register data on reg. ind 0-3    */
         cHost.MbusWriteCoilsTable(stInParams_Coil);
         printf("\n ... Mbus Write  Coil ind 0-3: %8x %8x %8x %8x ",
                                          (int)stInParams_Coil.coilsArr[0],
                                          (int)stInParams_Coil.coilsArr[1],
                                          (int)stInParams_Coil.coilsArr[2],
                                          (int)stInParams_Coil.coilsArr[3]);
         printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
         WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)

/* read mbus Coil start from location startRef along refCnt reg*/
         cHost.MbusReadCoilsTable(stInParams_Coil.startRef, stInParams_Coil.refCnt, stOutParams_Coil);
         printf("\n ... Mbus Readed Coil ind 0-3: %8x %8x %8x %8x ",
                                          (int)stOutParams_Coil.coilsArr[0],
                                          (int)stOutParams_Coil.coilsArr[1],
                                          (int)stOutParams_Coil.coilsArr[2],
                                          (int)stOutParams_Coil.coilsArr[3]);

         stInParams_Coil.coilsArr[0] = ~(stOutParams_Coil.coilsArr[0]);
         stInParams_Coil.coilsArr[1] = ~(stOutParams_Coil.coilsArr[1]);
         stInParams_Coil.coilsArr[2] = ~(stOutParams_Coil.coilsArr[2]);
         stInParams_Coil.coilsArr[3] = ~(stOutParams_Coil.coilsArr[3]);

     } while (loopCount < 3);
}


//   15.8.9.   cHost::MbusReadInputsTable   1392
void mBusReadWriteInput(void)
// =============================
{
     int  startRef;
     int  refCnt;

     int  loopCount = 0;

     MMC_MODBUSREADINPUTS_OUT stOutParams_Input;

     printf("\n\n Function: %s: ", __func__);

     printf("\n ... Going deal with Mbus Inputs EAS ind 1-4 ");
     startRef= MODBUS_UPDATE_START_INDEX;
     refCnt     = MODBUS_UPDATE_CNT;
     loopCount = 0;
     do
     {
         loopCount++;
         cHost.MbusReadInputsTable(startRef, refCnt, stOutParams_Input);
         printf("\n ... Mbus Readed inputs ind 0-3: %8x %8x %8x %8x ",
                    (int)stOutParams_Input.inputsArr[0], (int)stOutParams_Input.inputsArr[1],
(int)stOutParams_Input.inputsArr[2], (int)stOutParams_Input.inputsArr[3]);

         printf("\n ... Waiting %d Sec", MBUS_WAIT_FOR_USER);
         WAIT_SLEEP_MILLI(MBUS_WAIT_FOR_USER)
     } while (loopCount < 3);
}

#define   UL_TO_MBUS_STRUCT(modbus_write_val, aux_ul_p, mod_bus_idx)                   \
     {                                                                                  \
              aux_ul_p = (unsigned long *)(& mbus_write_in.regArr[mod_bus_idx]);   \
              * aux_ul_p = modbus_write_val;                                                  \
     }                                                                                  \
//   15.3.22.  CMMCSingleAxis::GetActualPosition      1164
//   15.3.23.  CMMCSingleAxis::GetActualVelocity      1164
//   15.3.24.  CMMCSingleAxis::GetActualTorque        1165
//   15.6.7.   CMMCConnection::GetGlobalBoolParameter 1245
void RepAxisAActualParamOnMbus(void)
// ================================
{
     MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN     mbus_write_in;
                              /* Auxiliary var for write to ModBus */
     unsigned  long*     ul_p;
     unsigned  long ul_val;
```

```
                double    dActul;

     dActul = AxisA.GetActualPosition();
     ul_val = (unsigned long)dActul;
/* Put value into ModBus write array use 2 index poss (0 & 1).  */
     UL_TO_MBUS_STRUCT(ul_val, ul_p, 0);

/* Remmber about mapping & config !!! */
/* if the motor connected above ehercat it needs apropirate */
/* config setting (EAS). */
     dActul = AxisA.GetActualVelocity();
     ul_val = (unsigned long)dActul;
                              /* Put value into ModBus write array use 2 index poss (2 & 3).  */
     UL_TO_MBUS_STRUCT(ul_val, ul_p, 2);


/* Remmber about mapping & config (see GetActualVelocity) !!!   */
     dActul = AxisA.GetActualTorque();
     ul_val = (unsigned long)dActul;
/* Put value into ModBus write array use 2 index poss (4 & 5).  */
     UL_TO_MBUS_STRUCT(ul_val, ul_p, 4);


/* Index of modbus H.reg. for start write actal values. */
     mbus_write_in.startRef       = MODBUS_STR_REP_ACT_VAL_INDX;
/* Number of indexes to write - each long extend along     */
/* 2 indexs (2 for Pos, 2 for Vel, 2 for Torq              */
     mbus_write_in.refCnt = 6;
     cHost.MbusWriteHoldingRegisterTable(mbus_write_in) ;

     return ;
}


#define   eCOMM_TYPE_ETHERCAT     1
#define   eCOMM_TYPE_CAN          2
/* Move Axis A and report it Actual */
/* parameters on Mbus.                */
void moveAxisAAndRepActualParamOnMbusPos(void)
// ==============================================
{
               int ind,
                        rt;
     unsigned int   uiState,
                        uiDoneFlg;

     printf("\n\n Function: %s: ", __func__);

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);


/* Connection type - CAN/EtherCAT */
     rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);
     if (rt == eCOMM_TYPE_ETHERCAT)
     {
          printf("\n --- Motor connection is via ETHERCAT remmber map & config for get actul param
(Vel. Torq.) ");
     }
     else if (rt == eCOMM_TYPE_CAN)
     {
          printf("\n --- Motor connection is via CAN ");
          if ( mapConfigPdo() )
          {
               printf("\n --- ERROR FAIL IN MAP PDO FOR CAN  !!! ");
          }
     }
     else
     {
          printf("\n --- ERROR UNKNOWN motor connection  !!! ");
     }

     printf("\n ... Start rep AxisA pos on Mbus");
     for(ind=0; ind<3; ind++)
```

```
     {
/* Move abs to: 400000, with parameters:         */
/*           Speed 300000, Acc=500000            */
/* Dec=1500000, Jerk=20000000, buffer mode=      */
/* MC_BUFFERED_MODE (def)        */
          AxisA.MoveAbsolute(400000*ind, 300000, 500000, 1500000, 20000000);
          do
          {
                              /* Write AxisA pos. to Modbus H.Reg */
               RepAxisAActualParamOnMbus();
               uiState = AxisA.ReadStatus();
               uiDoneFlg = uiState & NC_AXIS_STAND_STILL_MASK;
               if ( ! uiDoneFlg)
               {
                    WAIT_SLEEP_MILLI(40)
               }
          } while(! uiDoneFlg);
     }
     printf("\n ... End rep AxisA pos on Mbus");

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
}


int  mapConfigPdo(void)
// ===================
{
     int  rc;
     unsigned short                AxisARef;


     MMC_CONFIGREGULARPARAMEVENTPDO3_IN CfgReg3In;
    MMC_CONFIGREGULARPARAMEVENTPDO3_OUT CfgReg3Out;

    CfgReg3In.ucEventGroup    = NC_COMM_EVENT_GROUP6;
    CfgReg3In.ucPDOCommParam= PDO_COM_PARAM_SYNC;

     AxisARef = AxisA.GetRef();

    rc = MMC_CfgRegParamEvPDO3Cmd(ComHndl, AxisARef, &CfgReg3In, &CfgReg3Out);
    if(rc != 0)
    {
     printf("\n **** config error %d \n",CfgReg3Out.sErrorID);
/* ... goto lbl_motor_off; */
    }

     return (rc);
}

/*================================ Example functions END ==========================================*/
```

17.14.2 MbusStartServer

Refer to the section **16.3.5 MMC_MbusStartServer on page 1225** for details of the description, scope, and communication mode.

```
void MbusStartServer(
unsigned int uiConnHndl,
unsigned short usID
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCHostComm.h

.NET Definition

Function Parameters

uiConnHndl

[IN] Connection handle input using hConn, where MMC_CONNECT_HNDL is the Connection Handle Type. It should be noted that this connection handle is common throughout all Maestro functions. This connection handle is returned by Init Connection command. If error, returns -1 and a MMC_LIB_API error with further details.

usID

Modbus start server enumerator ID has the following values:

MODBUS_NOT_STARTED = 0

MODBUS_RUNNING = 1

MODBUS_STOPPED = 2

After the Maestro is powered-up, Modbus server is in the MODBUS_NOT_STARTED state – Initial state, the Modbus server does not exist.

After the Modbus server state is changed to MODBUS _RUNNING – the Modbus server is created, transmissions from Modbus clients will be handled by the server.

When the Modbus server state is changed to MODBUS_STOPPED – the Modbus server connection is removed, no transmissions will be handled from different Modbus clients.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

For code example, refer to the section **17.14.1**.

17.14.3 MbusStopServer

Refer to the section **16.3.6 MMC_MbusStopServer on page 1228** for details of the description, scope, and communication mode.

```
void MbusStopServer(
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCHostComm.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

For code example, refer to the section **17.14.1**.

17.14.4 MbusReadHoldingRegisterTable

Refer to the section **16.3.3 MMC_MbusReadHoldingRegisterTable on page 1219** for details of the description, scope, and communication mode.

```
void MbusReadHoldingRegisterTable(
int startRef,
int refCnt,
MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT& stOutParams
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCHostComm.h

.NET Definition

Function Parameters

startRef

Start Reference from the base coil table of linear parameters. Any +ve integer values accepted.

refCnt

Reference count. Any +ve integer values.

stOutParams

Refer to the **MMC_MODBUSREADHOLDINGREGISTERSTABLE_OUT Structure on page 1220.**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.14.1**.

17.14.5 MbusWriteHoldingRegisterTable

Refer to the section **16.3.8 MMC_MbusWriteHoldingRegisterTable on page 1234** for details of the description, scope, and communication mode.

```
void MbusWriteHoldingRegisterTable(
MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN& stInParams
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCHostComm.h

.NET Definition

Function Parameters

stInParams

Refer to the **MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN Structure on page 1235.**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.14.1**.

17.14.6 MbusIsRunning

Refer to the section **16.3.1 MMC_MbusIsRunning on page 1213** for details of the description, scope, and communication mode.

```
bool MbusIsRunning(
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCHostComm.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.14.1**.

17.14.7 MbusReadCoilsTable

Refer to the section **16.3.2 MMC_MbusReadCoilsTable on page 1216** for details of the description, scope, and communication mode.

```
void MbusReadCoilsTable(
int startRef,
int refCnt,
MMC_MODBUSREADCOILS_OUT& stOutParams
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCHostComm.h

.NET Definition

Function Parameters

startRef

Start Reference from the base coil table of linear parameters. Any +ve integer values accepted.

refCnt

Reference count. Any +ve integer values.

stOutParams

Refer to the **MMC_MODBUSREADCOILS_OUT Structure on page 1217.**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

For code example, refer to the section **17.14.1**.

17.14.8 MbusWriteCoilsTable

Refer to the section **16.3.7 MMC_MbusWriteCoilsTable on page 1231** for details of the description, scope, and communication mode.

```
void MbusWriteCoilsTable(
MMC_MODBUSWRITECOILS_IN& stInParams
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCHostComm.h

.NET Definition

Function Parameters

stInParams

Refer to the **MMC_MODBUSWRITECOILS_IN Structure on page 1232.**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

For code example, refer to the section **17.14.1**.

17.14.9 MbusReadInputsTable

Refer to the section **16.3.4 MMC_MbusReadInputsTable on page 1222** for details of the description, scope, and communication mode.

```
void MbusReadInputsTable(
int startRef,
int refCnt,
MMC_MODBUSREADINPUTS_OUT& stOutParams
) throw (CMMCException);
```

Source GMAS\includes\CPP\CMMCHostComm.h

.NET Definition

Function Parameters

startRef

Start Reference from the base coil table of linear parameters. Any +ve integer values accepted.

refCnt

Reference count. Any +ve integer values.

stOutParams

Refer to the **MMC_MODBUSREADINPUTS_OUT Structure on page 1223.**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name Structure name
Axis reference Error ID
Status of the axis.

For code example, refer to the section **17.14.1**.

17.14.10 SetModbus[LongSwapped][Short]

Refer to the section 16.3.4 MMC_MbusReadInputsTable on page 1222 for details of the description, scope, and communication mode.

void SetModbus[LongSwapped][Short][Float][FloatSwapped](

MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN& stInParams,

int iOffset,

[iRefCount],

long lPos,[short sPos],[float fPos],

) throw (CMMCException);

Source GMAS\includes\CPP\CMMCHostComm.h

.NET Definition

Function Parameters

iOffset

The is the start reference parameter **StartRef** defined in the **MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN Structure on page 1235**.

iRefCount

The is the reference count parameter **refCnt** defined in the **MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN Structure on page 1235**.

long lPos,[short sPos],[float fPos]

Long, short, or float value of the array parameter described in **MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN Structure on page 1235.**

stInParams

Refer to the **MMC_MODBUSWRITEHOLDINGREGISTERSTABLE_IN Structure on page 1235.**

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.15 The CMMCModbusBuffer class

The class CMMCModbusBuffer wraps the host communication functions detailed in the section 16.2 Host Communication on page 1212. The class CMMCModbusBuffer retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

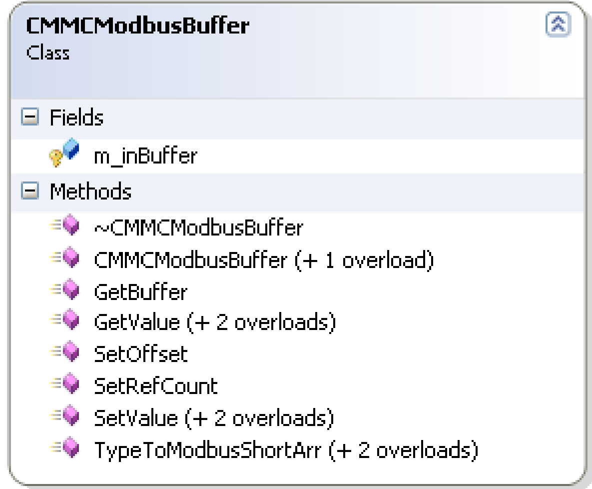


Figure 17-21 Fields and methods of the CMMCModbusBuffer class

The detailed class view shown in Figure 17-21 describes the fields and methods associated with the CMMCModbusBuffer class. These are generally default parameters, which can be operated using their default values. However if the user wishes to change the defaults, refer to the relevant parameter section in the manual.

It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.16 The CMMCModbusSwapBuffer class

The class CMMCModbusSwapBuffer wraps the host communication functions detailed in the section **16.2 Host Communication on page 1212**. The class CMMCModbusSwapBuffer retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

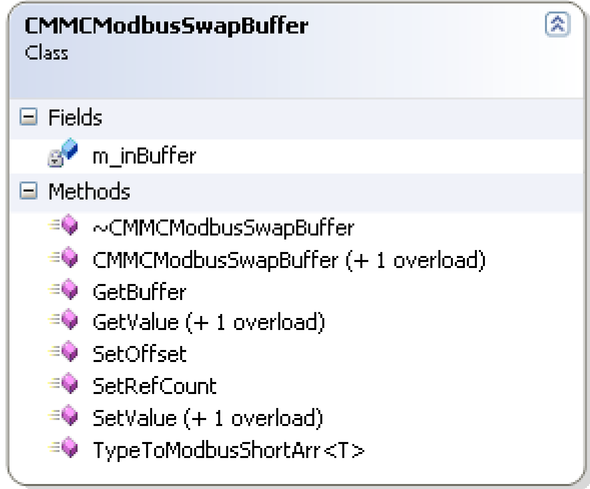


Figure 17-22 Fields and methods of the CMMCModbusSwapBuffer class

The detailed class view shown in **Figure 17-22** describes the fields and methods associated with the CMMCModbusSwapBuffer class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.17 The CMMCErrorCorr class

The class CMMCErrorCorr wraps the network communication functions detailed in the section . The diagram in **Figure 17-26** describes the heirarchial structure of the classes and type definitions associated with the CMMCErrorCorr.

The class CMMCErrorCorr retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

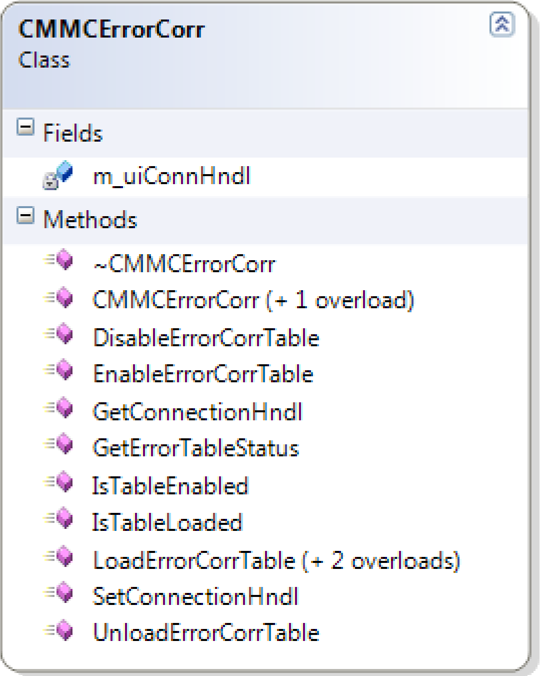


Figure 17-23 Fields and methods of the CMMCErrorCorr class

The detailed class view shown in **Figure 17-26** describes the fields and methods associated with the CMMCErrorCorr class. It should be noted that Protected and Private functions together with their operations, should be transparent to the user, and are not for general application by the user.

17.17.1 LoadErrorCorrTable

Loads an error correction table to memory. Error correction is then performed according to this table. Refer to the section **13.4.1 MMC_LoadErrorCorrTable** for details of the description, scope, and communication mode.

Int(void) LoadErrorCorrTable(
MMC_LOADERRORTABLE_IN& stInParams
[char pPathToETFile[NC_MAX_ET_FILE_PATH_LENGTH]]
[NC_ERROR_TABLE_NUMBER eETNumber]
[double dMaxCorrectionDelta = 0]
) throw (CMMCException)

Source C:\GMAS\includes\CPP\MMCErrorCorr.h

.NET Definition

Function Parameters

MMC_LOADERRORTABLE_IN& stInParams

Refer to the section**MMC_LOADERRORTABLE_IN** structure for details of the input parameters.

pPathToETFile[NC_MAX_ET_FILE_PATH_LENGTH]

Defines the path to the error table file.

[NC_MAX_ET_FILE_PATH_LENGTH] is the maximum size of the error table file path. It is limited to 100.

If you set "NULL" in this input, the default file path will be used. The default path is set to the:

/mnt/jffs/usr/ directory and the filename will be depended on the table index:

ErTBL_#.txt where the # is the table index (A,B,C,D,E,F).

NC_ERROR_TABLE_NUMBER eETNumber

Defines the error table letter assigned. NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

dMaxCorrectionDelta

This parameter define the maximum allowed correction input value. If you try to insert a table where one of the correction values is above the MaxCorrectionDelta, an error is received; **-342**. If you set this value to "0", the max correction delta is unlimited.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis.

17.17.2 UnloadErrorCorrTable

Unloads an error correction table from memory. Refer to the section **13.4.5 MMC_UnloadErrorCorrTable** for details of the description, scope, and communication mode.

int UnloadErrorCorrTable(
NC_ERROR_TABLE_NUMBER eTableNumber
) throw (CMMCException)

Source C:\GMAS\includes\CPP\MMCErrorCorr.h

.NET Definition

Function Parameters

NC_ERROR_TABLE_NUMBER eETNumber

Defines the error table letter assigned. NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis.

17.17.3 EnableErrorCorrTable

EnableErrorCorrTable. Refer to the section **13.4.2 MMC_EnableErrorCorrTable** for details of the description, scope, and communication mode.

```
int EnableErrorCorrTable(
NC_ERROR_TABLE_NUMBER eTableNumber
) throw (CMMCException);
```

Source C:\GMAS\includes\CPP\MMCErrorCorr.h

.NET Definition

Function Parameters

NC_ERROR_TABLE_NUMBER eETNumber

Defines the error table letter assigned. NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis.

17.17.4 DisableErrorCorrTable

Disables the usage of an error table. Refer to the section **13.4.4 MMC_DisableErrorCorrTable** for details of the description, scope, and communication mode.

int DisableErrorCorrTable(
NC_ERROR_TABLE_NUMBER eTableNumber
) throw (CMMCException)

Source C:\GMAS\includes\CPP\MMCErrorCorr.h

.NET Definition

Function Parameters

NC_ERROR_TABLE_NUMBER eETNumber

Defines the error table letter assigned. NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis.

17.17.5 GetErrorTableStatus

Function recieves the table number as input and returns an answer whether the table is loaded and/or enabled. Refer to the section **13.4.3 MMC_GetErrorTableStatus** for details of the description, scope, and communication mode.

```
void GetErrorTableStatus(
NC_ERROR_TABLE_NUMBER eTableNumber,
MMC_GETERRORTABLESTATUS_OUT &stOutParams
);
```

Source C:\GMAS\includes\CPP\MMCErrorCorr.h

.NET Definition

Function Parameters

NC_ERROR_TABLE_NUMBER eETNumber

Defines the error table letter assigned. NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

MMC_GETERRORTABLESTATUS_OUT &stOutParams

Refer to the **MMC_GETERRORTABLESTATUS_OUT Structure** for details of the ouput parameters.

17.17.6 IsTableEnabled

Boolean function requests whether the table is enabled. Refer to the section **13.4.3 MMC_GetErrorTableStatus** for details of the description, scope, and communication mode.

```
bool IsTableEnabled(
NC_ERROR_TABLE_NUMBER eETNumber
) throw (CMMCException);
```

Source C:\GMAS\includes\CPP\MMCErrorCorr.h

.NET Definition

Function Parameters

NC_ERROR_TABLE_NUMBER eETNumber

Defines the error table letter assigned. NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis.

17.17.7 IsTableEnabled

Boolean function requests whether the table is loaded. Refer to the section **13.4.3 MMC_GetErrorTableStatus** for details of the description, scope, and communication mode.

```
bool IsTableLoaded(
NC_ERROR_TABLE_NUMBER eETNumber
) throw (CMMCException);
```

Source C:\GMAS\includes\CPP\MMCErrorCorr.h

.NET Definition

Function Parameters

NC_ERROR_TABLE_NUMBER eETNumber

Defines the error table letter assigned. NC_ERROR_TABLE_NUMBER is an enumerator describing the with the following values:

NC_ERROR_TABLE_A

NC_ERROR_TABLE_B

NC_ERROR_TABLE_C

NC_ERROR_TABLE_D

NC_ERROR_TABLE_E

NC_ERROR_TABLE_F

NC_ERROR_TABLE_MAX

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including; Function Name, Structure name, Axis reference, Error ID, Status of the axis.

17.18 The CMMCBulkRead class

The class CMMCBulkRead wraps the bulk parameters reading functions detailed in **Chapter 11: Bulk Parameters Reading on page 1065**. The class CMMCBulkRead retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

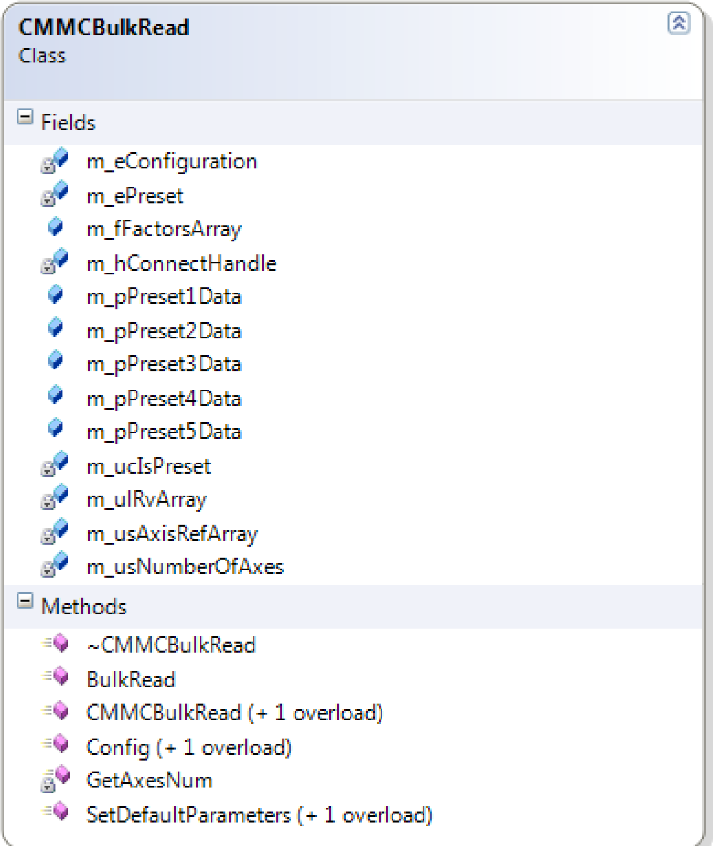


Figure 17-24 Fields and methods of the CMMCBulkRead class

The detailed class view shown in **Figure 17-24** describes the fields and methods associated with the CMMCBulkRead class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.18.1 CMMCBulkRead Source Code Examples

```
MMC_CONFIGBULKREAD_IN stCfgIn;

stCfgIn.eConfiguration = eBULKREAD_CONFIG_2;
stCfgIn.uBulkReadParams.eBulkReadPreset = eNC_BULKREAD_PRESET_2;
stCfgIn.ucIsPreset = 1;
stCfgIn.usAxisRefArray[0] = 0;
stCfgIn.usAxisRefArray[0] = 1;

stCfgIn.usNumberOfAxes = 2;

CMMCBulkRead br(g_hConnect);
try
{
 br.SetDefaultParameters(stCfgIn);
 br.BulkRead();
}

catch (CMMCException e)
{
 cout << e.what() << endl;
}
```

Alternatively:

```
stCfgIn.eConfiguration = eBULKREAD_CONFIG_2;
stCfgIn.uBulkReadParams.eBulkReadPreset = eNC_BULKREAD_PRESET_2;
stCfgIn.ucIsPreset = 1;
stCfgIn.usAxisRefArray[0] = 0;
stCfgIn.usAxisRefArray[0] = 1;

stCfgIn.usNumberOfAxes = 2;

CMMCBulkRead br(g_hConnect);
try
{
 br.Config(stCfgIn);
 br.BulkRead();
}

catch (CMMCException e)
{
 cout << e.what() << endl;
}
```

17.18.2 CMMCBulkRead

Refer to the section **11.1.1 above MMC_ConfigBulkRead** for details of the description, and scope.

```
CMMCBulkRead(
MMC_CONNECT_HNDL hConnectHandle,
NC_BULKREAD_CONFIG_ENUM eConfig,
NC_BULKREAD_PRESET_ENUM ePreset
);
```

Source GMAS\includes\CPP\CMMCBulkRead.h

.NET Definition

Function Parameters

hConnectHandle

Connection handle

eConfig

Configuration enumerator defined by the following:

BULKREAD_DEFAULT_NUMBER_OF_AXES	2
BULKREAD_DEFAULT_FIRST_AXIS_REF	0
BULKREAD_DEFAULT_SECOND_AXIS_REF	1
BULKREAD_DEFAULT_IS_PRESET	1
BULKREAD_DEFAULT_PRESET	eNC_BULKREAD_PRESET_1
BULKREAD_DEFAULT_CONFIG	eBULKREAD_CONFIG_2

ePreset

Preset enumerator defined by the following options:

NC_BULKREAD_PRESET_1 m_pPreset1Data[NC_MAX_AXES_PER_BULK_READ];

NC_BULKREAD_PRESET_2 m_pPreset2Data;

NC_BULKREAD_PRESET_3 m_pPreset3Data[NC_MAX_AXES_PER_BULK_READ];

For further details, refer to **Chapter 11: Bulk Parameters Reading on page 1065**.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:

Function Name	Structure name
Axis reference	Error ID
Status of the axis.	

17.18.3 Config

Refer to the section **11.1.1 above MMC_ConfigBulkRead** for details of the description, and scope.

void Config(
MMC_CONFIGBULKREAD_IN stCfgBulkReadIn
) throw (CMMCException);

Source GMAS\includes\CPP\CMMCBulkRead.h

.NET Definition

Function Parameters

stCfgBulkReadIn

Refer to the section **MMC_CONFIGBULKREAD_IN Structure on page 1067** for details of this parameter.

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

17.18.4 BulkRead

Refer to the section **11.1.1 above MMC_ConfigBulkRead** for details of the description, and scope.

void BulkRead(
) throw (CMMCException);

Source GMAS\includes\CPP\CMMCBulkRead.h

.NET Definition

Function Parameters

throw (CMMCException)

Refer to the section **17.1.1 CMMCException**. Produces details of the error including:
Function Name Structure name
Axis reference Error ID
Status of the axis.

17.19 The CMMCUserParams class

The class CMMCUserParams functions are a group of methods for save and retrieve Maestro data (E.g. GMAS Parameters: Velocity, Acceleration, Location-points...). An XML file is used for managing this Maestro data.

17.19.1 Using the XML structure

The XML tree structure consists of the highest element which are predefined absolutely. The first (highest) element defines the XML version which is used to describe how to read the tree. The second element is the root of the tree which describes the XML Schema origin used for build the tree. This element has a user defined name for point on the XMD file. The two above elements should always exist even when data are not in use.In this case the names of XMD file for second element are "just and only name". The next Element NAME (under the "root") is fixed and is the CATEGORY, which has user a defined value (ATTRIBUTE Value). The Element NAME under CATEGORY elements name is RESOURCES, it has user defined Value (ATTRIBUTE Value). The Element NAME under RESOURCES elements name is user defined and its data is also a user value.

17.19.2 General

The class CMMCUserParams retains the same field parameter properties and values described in this document for the C function blocks.

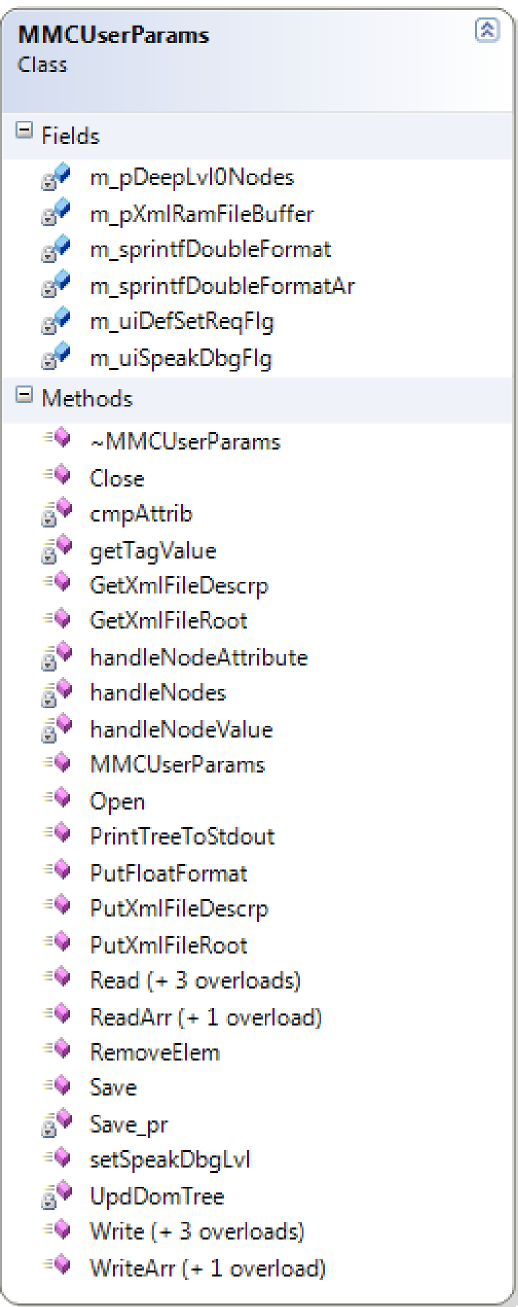


Figure 17-25 Fields and methods of the CMMCMotionAxis class

The detailed class view shown in **Figure 17-25** describes the fields and methods associated with the CMMCUserParams class.

It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.19.3 XML Data File Example

```
<?xml version="1.0" encoding="utf-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="proposed.xsd">
    <FILE_DESCRIPTION NAME="Parameters" VERSION="NovaScan 1236" />
    <CATEGORY NAME="Profiler">
        <RESOURCES NAME="a01">
            <DC>10000000</DC>
            <JERK>1073742336</JERK>
            <DRIVE_ID>'81'</DRIVE_ID>
        </RESOURCES>
        <RESOURCES NAME="a02">
            <AXIS_MODE>0</AXIS_MODE>
            <OP_MODE>8</OP_MODE>
        </RESOURCES>
    </CATEGORY >
    <CATEGORY NAME="Communication">
        <RESOURCES NAME="a01">
            <TIMEOUT>0</TIMEOUT>
        </RESOURCES>
        <RESOURCES NAME="a02">
            <TIMEOUT>0</TIMEOUT>
            <NUM_VARS>8</NUM_VARS>
        </RESOURCES>
    </CATEGORY >
</root>
```

17.19.4 Errors Applicable to the CMMCUserParams Class

Specific Library Errors	Value	Explanation
MMC_LIB_UPXML_ERROR	-3	Error occurred on 'Get User Parameter from XML file'
MMC_LIB_UPXML_WARNING	300	Warning from 'get User Param from XML file library'
Get User Parameter from XML file Errors code:		
MMC_LIB_UPXML_FILE_FORMAT	-3010	XML file Syntax is not well form
MMC_LIB_UPXML_FAIL_ALLOC	-3011	Failed in allocation memory
MMC_LIB_UPXML_OPEN_FAILD	-3012	Failed in Open XML file
MMC_LIB_UPXML_LONG_NAME	-3013	File name (inc path) is too long
MMC_LIB_UPXML_NOT_FOUND	-3020	ENTRY NOT Found
MMC_LIB_UPXML_UNEXP_CHR	-3021	Found but has Unexpected char
MMC_LIB_UPXML_NOT_IN_RANGE	-3022	Found Value outside of Min Max
MMC_LIB_UPXML_SAVE_w_FAILD	-3014	Faild open file for write
MMC_LIB_UPXML_WRITE_FAILD	-3016	Actual len diff from element len
MMC_LIB_UPXML_NUM_FORMAT	-3017	Worng format for number
MMC_LIB_UPXML_NOT_SUPPORTNG	-3018	Not supporting yet
MMC_LIB_UPXML_ILLEGAL_ARRYSIZE	-3019	Illegal array size (E.g: -1)
MMC_LIB_UPXML_ILLEGAL_FLOAT_FORMAT	-3020	Illegal parm for PutFloatFormat()
Warning: Get User Parameter from XML file:		
MMC_LIB_UPXML_DEF_NOT_FOUND	3020	Set Def; NOT Found
MMC_LIB_UPXML_DEF_UNEXP_CHR	3021	Set Def; Found Unexpected char
MMC_LIB_UPXML_DEF_NOT_IN_RANGE	3022	Set Def; Found Val out of Min Max
MMC_LIB_UPXML_UPDATE_L3_ELEM	3023	Update Elem Val. (exist) in Lvl 3
MMC_LIB_UPXML_UPDATE_L2_ELEM	3024	Create (Upd tree) new Elem in Lvl 2
MMC_LIB_UPXML_UPDATE_L1_ELEM	3025	Create (Upd tree) new Elem in Lvl 1

17.19.5 CMMCUserParams class Functions and Examples

Refer to the section **15.2 The MMCUserParams C++ Class** for deails of the functions and function examples.

17.20 The CMMCEIPSession class

The class CMMCEIPSession wraps the network communication functions detailed in the section . The diagram in **Figure 17-26** describes the heirarchial structure of the classes and type definitions associated with the CMMCEIPSession.

The class CMMCEIPSession retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

Figure 17-26 Fields and methods of the MMCEIPSession class

The detailed class view shown in **Figure 17-26** describes the fields and methods associated with the CMMCEIPSession class. It should be noted that Protected and Private functions together with their operations, should be transparent to the user, and are not for general application by the user.

17.20.1 EIPCloseSession

Closes an EthernetIP session. Refer to the section **16.10.15 EIPCloseSession** for details of the description, scope, and communication mode.

```
int EIPCloseSession(
);
```

Source GMAS\includes\CPP\MMCEIPSession.h

.NET Definition

Function Parameters

N/A

17.20.2 EipCreate

Creates a new EthernetIP session. Refer to the section **16.10.16 EipCreate** for details of the description, scope, and communication mode.

```
int EipCreate(
char *strPath
);
```

Source GMAS\includes\CPP\MMCEIPSession.h

.NET Definition

Function Parameters

**strPath*

Path of the tag configuration file on GMAS (XML). Value to maximum of 80 characters. Refer to the similar parameter *cPath* in the section **16.10.16 EipCreate EIP_CREATE_IN Structure**.

17.20.3 EipDestroy

Kills the present EtherNETIP session. Refer to the section **16.10.17 EipDestroy** for details of the description, scope, and communication mode.

```
int EipDestroy(
);
```

Source GMAS\includes\CPP\MMCEIPSession.h

.NET Definition

Function Parameters

17.20.4 EipOpenSession

Opens a new EIP session. Refer to the section **16.10.14 EipOpenSession** for details of the description, scope, and communication mode.

```
int EipOpenSession(
EIP_CALLBACK_FUNC pClbkFunc,
bool bEventNotification = true
);
```

Source GMAS\includes\CPP\MMCEIPSession.h

.NET Definition

Function Parameters

pClbkFunc

The value of the EIP callback function which produces the appropriate event. Refer to the details on the callback function in section 16.10.14.

bEventNotification = true

True for call-back function invocation, otherwise false (0 or 1 as default).

If pClbkFunc is NULL then no event is produces no matter the bEventNotification value.

17.21 The CMMCEIPDataType class

The class CMMCEIPDataType wraps the network communication functions detailed in the section . The diagram in **Figure 17-26** describes the heirarchial structure of the classes and type definitions associated with the CMMCEIPDataType. These define the various EthernetIP data type variables and their possible values.

The class CMMCEIPDataType retains the same field parameter properties and values described in this document for the C function blocks, and while small visual changes may be made to some variables, these are transparent, and do not change the operation of the variable.

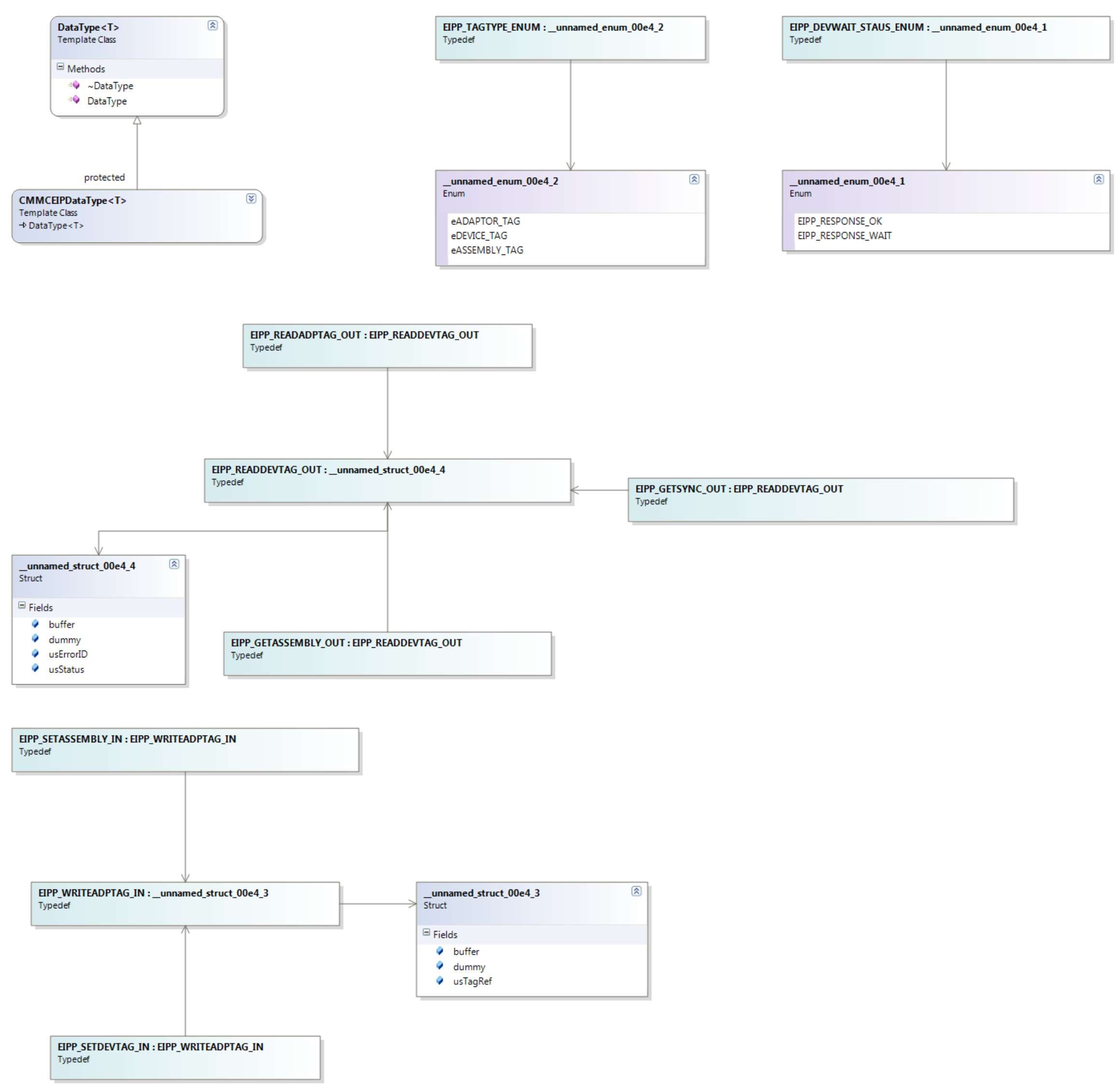


Figure 17-27 Fields and methods of the CMMCEIPDataType class

The detailed class view shown in **Figure 17-26** describes the fields and methods associated with the CMMCEIPDataType class. It should be noted that Protected and Private functions together with their operations, should be transparent to the user, and are not for general application by the user.

17.21.1 EthernetIP Source Code Examples

```
#include "EIP_API.h"
#include "MMCEIPDataType.h"
#include "MMCEIPSession.h"

#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <sys/time.h>

#define NO_CALLBACK_EVENTS          0
#define DO_CALLBACK_EVENTS          1
#define _RESOURCE_XML_FILE          "/mnt/jffs/usr/eippdemo.xml"
//////////////////////////////////////////////////////////////////////////////
//                              Globals                                     //
//////////////////////////////////////////////////////////////////////////////
int gTerminateFlag;

struct timeval tv1, tv2;
struct timezone tz;
long timeOffset = 0;

#define _EIPP 1 //one may want to test simple C interface as well

#ifdef _EIPP
enum _DTYPE_ID {
    eARR_INDEX = 0,
    eSINGLE_INDEX
} DTYPE_ID;

/*
 * Please note type in use must comply with XML type definitions for each tag
 */
CMMCEIPDataType<char> g_devPlc2Gmas[2]; //simply store two tags(single/multidimensional)
CMMCEIPDataType<char> g_devGmas2Plc[2];//simply store two tags (single/multidimensional)
CMMCEIPDataType<short> g_adpPlc2Gmas;
CMMCEIPDataType<short> g_adpGmas2Plc;
CMMCEIPDataType<short> g_adpOne2Gmas;
CMMCEIPDataType<short> g_adpGmas2One;
CMMCEIPDataType<int>       g_asmPlc2Gmas;
CMMCEIPDataType<int>       g_asmGmas2Plc;

#else //not _EIPP
#endif //_EIPP
//////////////////////////////////////////////////////////////////////////////
//                             Function Prototypes                          //
//////////////////////////////////////////////////////////////////////////////
void terminate_application(int signum);

//////////////////////////////////////////////////////////////////////////////
//                             External Functions                           //
//////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////
//                             Implementation                               //
//////////////////////////////////////////////////////////////////////////////

/*
 * general implementation of EtherNetIP call-back function.
 */
int fnCallback(unsigned char* ucBuffer, short sReqID, void* pSock)
{
    unsigned char ucEventID = ucBuffer[2];
    switch (ucEventID)
    {
   case NM_REQUEST_RESPONSE_RECEIVED:
          //printf("NM_REQUEST_RESPONSE_RECEIVED: sReqID = %d\n", sReqID);
```

```
    break;
        case NM_ASSEMBLY_NEW_INSTANCE_DATA:
            //printf("NM_ASSEMBLY_NEW_INSTANCE_DATA: assembly instance = %d\n", sReqID);
            break;
        case NM_ASSEMBLY_NEW_MEMBER_DATA:
            //New data received for the specified assembly member. sReqID contains assembly
instance.
            //printf("NM_ASSEMBLY_NEW_MEMBER_DATA: assembly instance = %d\n", sReqID);
           break;
        case NM_REQUEST_FAILED_INVALID_NETWORK_PATH:
            break;
        case NM_REQUEST_TIMED_OUT:
    printf("NM_REQUEST_TIMED_OUT: sReqID = %d\n", sReqID);
            break;
        case NM_CONNECTION_ESTABLISHED:
            //printf("\033[1;45m\tEIPTestAll: New connection opened with instance
%d\033[0m\n", sReqID);
            break;
        case NM_CONNECTION_VERIFICATION:
            printf("NM_CONNECTION_VERIFICATION\n");
            break;
   case NM_CONNECTION_RECONFIGURED:
    printf("NM_CONNECTION_RECONFIGURED\n");
    break;
        case NM_CONNECTION_TIMED_OUT:
            //printf("\033[1;42m\tEIPTestAll: Connection with instance %d timed
out\033[0m\n", sReqID);
            break;
        case NM_CONNECTION_CLOSED:
            //printf("\033[1;43m\tEIPTestAll: Connection with instance %d closed\033[0m\n",
sReqID);
            break;
        case NM_CLIENT_OBJECT_REQUEST_RECEIVED:
            printf("NM_CLIENT_OBJECT_REQUEST_RECEIVED\n");
            break;
        case NM_PENDING_REQUESTS_LIMIT_REACHED:
            printf("NM_PENDING_REQUESTS_LIMIT_REACHED\n");
            break;

        default:
            break;
    }
    return 0;
}

/*
 * general initialization for these kind of applications
 */
int MainInit() {
    long time;
    struct sigaction stSigAction;

    //whenever a signal is caught, call terminate_application function
    stSigAction.sa_handler = terminate_application;

    sigaction(SIGINT, &stSigAction, NULL);
    sigaction(SIGTERM, &stSigAction, NULL);
    sigaction(SIGABRT, &stSigAction, NULL);
    sigaction(SIGQUIT, &stSigAction, NULL);

    /* Initialise peformance counter */
    /* ----------------------------- */
    for (int i = 0; i < 100; i++)
    {
        gettimeofday(&tv1, &tz);
        usleep(1000);
        gettimeofday(&tv2, &tz);
        time = (tv2.tv_sec - tv1.tv_sec)*1000000 + (tv2.tv_usec - tv1.tv_usec);
        timeOffset += (1000 - time);
```

```
    }
    timeOffset /= 100;

    return 0;
}

#ifndef _EIPP
#else //_EIPP

/*
 * Sample for EtherNetIp opening session via CPP interface
 */
int OpenSessionPP() {
    printf("Opening CPP EIP session\n");
    CMMCEIPSession::Instance()->EipOpenSession(fnCallback, false);
    CMMCEIPSession::Instance()->EipCreate(_RESOURCE_XML_FILE);
    return 0;
}
/*
 * Sample for EtherNetIp closing session via CPP interface
 */
int CloseSessionPP() {
    printf("Free EIP session memory\n");

    CMMCEIPSession::Instance()->EipDestroy();

    printf("Closing CPP EIP session\n");

    CMMCEIPSession::Instance()->EIPCloseSession();
    return 0;
}

#endif //_EIPP

/*
 *  Init device tags reference
 */
#ifndef _EIPP
#else //_EIPP

/*
 * Sample for adapter tags operations
 */
int AdpTESTPP() {
    int iHBCount;
    short iBuffer[7];
    static short iVar=0;

    g_adpPlc2Gmas.EipGetTag(iBuffer);
    g_adpOne2Gmas.EipGetTag(iVar);
    iHBCount = (int)iBuffer[0];
    //set heartbit
    iBuffer[0] =  (iHBCount > 0x7FFFFFFE) ? 0 : iHBCount + 1; //limit to max signed integer
0x7FFFFFFF.
    g_adpGmas2Plc.EipSetTag(iBuffer);
    g_adpGmas2One.EipSetTag(iVar+1);

    return 0;
}
/*
 * Sample for adapter tags initialization. Must be invoked before any other operation on tags.
 */
int AdpInitPP() {
    int rc = g_adpPlc2Gmas.EipTagInit("adp_plc2gmas");
    if (!rc) {
        rc  = g_adpGmas2Plc.EipTagInit("adp_gmas2plc");
        if (!rc) {
            rc  = g_adpOne2Gmas.EipTagInit("adp_one2gmas");
            if (!rc) {
```

```
                    rc  = g_adpGmas2One.EipTagInit("adp_gmas2one");
                }
            }
        }
        return rc;
}
/*
 * Sample for device tags operations
 */
int DevTESTPP() {
        char  iBuffer[7];
        char iVar;
        short sReplyStatus;
        //
        //read device tag asynchronous
        //

        //single example
        if(!g_devPlc2Gmas[eSINGLE_INDEX].EipIsWaiting())
        {
            g_devPlc2Gmas[eSINGLE_INDEX].EipGetTag(NULL);
        }
        else
        {
            g_devPlc2Gmas[eSINGLE_INDEX].EipCheckReply(sReplyStatus);
            if (sReplyStatus == EIPP_RESPONSE_OK)
            {
                g_devPlc2Gmas[eSINGLE_INDEX].EipGetData(iVar);
                g_devGmas2Plc[eSINGLE_INDEX].EipSetTag(iVar+1);//heartbeat
            }
        }

        //multidimensional example
        if(!g_devPlc2Gmas[eARR_INDEX].EipIsWaiting())
        {
            g_devPlc2Gmas[eARR_INDEX].EipGetTag(NULL);
        }
        else
        {
            g_devPlc2Gmas[eARR_INDEX].EipCheckReply(sReplyStatus);
            if (sReplyStatus == EIPP_RESPONSE_OK)
            {
                g_devPlc2Gmas[eARR_INDEX].EipGetData(iBuffer);
                iBuffer[0]++;// heartbeat
                g_devGmas2Plc[eARR_INDEX].EipSetTag(iBuffer);
            }
        }
        return 0;
}

/*
 * Sample for device tags initialization. Must be invoked before any other operation on tags.
 */
int DevInitPP() {
        int rc = g_devPlc2Gmas[eARR_INDEX].EipTagInit("dev_plc2gmas");

        if (!rc) {
            rc  = g_devGmas2Plc[eARR_INDEX].EipTagInit("dev_gmas2plc");
            if (!rc) {
                rc  = g_devPlc2Gmas[eSINGLE_INDEX].EipTagInit("dev_one2gmas");
                if (!rc) {
                    rc  = g_devGmas2Plc[eSINGLE_INDEX].EipTagInit("dev_gmas2one");
                }
            }
        }
        if(rc)
            printf("DevInitPP Failed\n");
        return rc;
}
```

```
/*
 * Sample for Assembly tags initialization. Must be invoked before any other operation on
tags.
 */
int AsmInitPP()
{
     int rc = g_asmPlc2Gmas.EipTagInit("asm_plc2gmas"); //initialization by name
     /*or by instances
      * int rc = g_asmPlc2Gmas.EipTagInit(2); //initialization by instance (must comply to XML
definitions)
      * */
     if (!rc) {
          rc = g_asmGmas2Plc.EipTagInit("asm_gmas2plc");
          /*or by instances
           * int rc = g_asmPlc2Gmas.EipTagInit(1); //initialization by instance (must comply to
XML definitions)
           * */
     }
     return rc;
}

/*
 * Sample for Assembly tags operations
 */
int AsmTESTPP()
{
     static int prev_data = 0;
     static int in_data;
     int iBuffer[7]; //type and dimension must comply to tyep and size definitions in XML

     g_asmPlc2Gmas.EipGetTag(iBuffer);
     in_data = iBuffer[0];

     //set changes only if first cell is different (this is only an application preference)
//   if (prev_data != in_data) {
          prev_data = in_data;
          iBuffer[0] = in_data + 1;
          g_asmGmas2Plc.EipSetTag(iBuffer);
//   }
     return 0;
}

#endif //_EIPP

/*
 * just displays a propeller for progress notification.
 */
void DisplayProgress() {
     static int iProgressIndex = 0;
     static char *cClock[] = {"-", "\\", "|", "/",  "-", "\\", "|", "/"};
     fprintf(stderr, "%s\r", cClock[iProgressIndex]);
     if (++iProgressIndex>7) iProgressIndex = 0;
}
/****************************************************************************
 * FUNCTION:   terminate_application
 * DESCRIPTION:
 * INPUTS:     N/A
 * OUTPUTS:
 * RETURN VALUE:
 ****************************************************************************/
void terminate_application(int signum)
{
     gTerminateFlag = 1;
}
/****************************************************************************
 * FUNCTION:   Initialize
 * DESCRIPTION:
 * INPUTS:     N/A
```

```
 * OUTPUTS:
 * RETURN VALUE:
 ******************************************************************************/
int main(int argc, char** argv)
{
    static int iClock = 0;

    MainInit();

#ifdef _EIPP
    OpenSessionPP();
    //initializations
    if (AdpInitPP())
    {
        printf("Addpters initialization faild\n" );
        exit (1);
    }
    if (DevInitPP())
    {
        printf("Addpters initialization faild\n" );
        exit (1);
    }

    if (AsmInitPP())
    {
        printf("Assemblies initialization faild\n" );
        exit (1);
    }

    while (!gTerminateFlag)
    {
        usleep(5000); //5 ms

        /* Device tags TEST */
        /* ---------------- */
        DevTESTPP();

        /* Adapter tags TEST */
        /* ----------------- */
        AdpTESTPP();

        /* assemblies TEST */
        /* --------------- */
        AsmTESTPP();
        if (!(++iClock%200))
            DisplayProgress();
//        if (iClock > 400)
//            break;
    }

    CloseSessionPP();

#else // not _EIPPP
#endif //_EIPPP
    return 0;
}
```

17.21.2 EipTagInit

Obtains a tag reference by name (*szName) or by instance (iInstance) and initializes this object with tag properties. This uses the C library function to obtain a tag reference by name. For example, refer to the section **16.10.8 EipGetDevTagRefByName** for details of the description, scope, and communication mode.

```
int EipTagInit(
const char *szName
[int iInstance]
);
```

Source GMAS\includes\CPP\ CMMCEIPDataType.h

.NET Definition

Function Parameters

**szName*

Tag name as declared in XML configuration file. Refer to the parameter *cName* in section **16.10.8 EipGetDevTagRefByName EIP_REFBYNAME_IN Structure**.

int iInstance

Tag instance as declared in XML configuration file.

17.21.3 EipSetTag

Writes tag data for a remote device. This uses the C library function to set the tag. For example, refer to the section **16.10.2 EipWriteAdpTag** for details of the description, scope, and communication mode.

```
int EipSetTag(
T tData[]
[T tData]
);
```

Source GMAS\includes\CPP\ CMMCEIPDataType.h

.NET Definition

Function Parameters

T tData[],T tData

[INPUT] The data of the referenced adapter tag either as an array (buffer) or a single data value. For example,refer to the section **16.10.2 EipWriteAdpTag EIP_WRITEDATA_IN Structure**

17.21.4 EipGetTag

Obtains data from a remote device into given tData buffer or tVar. For example, refer to the section **16.10.3 EipReadAdpTag** for details of the description, scope, and communication mode.

```
int EipGetTag(
T tData[],
[T& tVar]
bool bSync=false,
);
```

Source GMAS\includes\CPP\ CMMCEIPDataType.h

.NET Definition

Function Parameters

T tData[]

[OUTPUT] Retrieve data of the referenced adapter tag either as an array (buffer) or a single data value. For example, refer to the sections **16.10.3 EipReadAdpTag.**

T& tVar

[OUTPUT] The variable reference retrieved value. This is a returned output.

bool bSync=false

bSync synchronous flag. This parameter is only relevant for device tags. The default is Asynchronous or false. Otherwise it operates as a synchronized call.

17.21.5 EipCheckReply

Answers whether or not data for the last request has arrived. Refer to the section **16.10.13 EipCheckDevTagReply** for details of the description, scope, and communication mode.

```
int EipCheckReply(
short& sStatus
);
FORCEDINLINE bool EipIsWaiting(
){return _iDevWaitStatus;}
```

Source GMAS\includes\CPP\ CMMCEIPDataType.h

.NET Definition

Function Parameters

sStatus

The status is returned as either 0 meaning OK, or 1, Wait.

FORCEDINLINE bool EipIsWaiting

This answers the question whether the devise tag is in waiting for asynchronous reply or not.

17.21.6 EipGetData

Obtain tag data from a remote device. For example, refer to the sections **16.10.7 EipGetAssembly** for details of the description, scope, and communication mode.

```
int EipGetData(
T tData[]
[T& tVar]
);
```

Source GMAS\includes\CPP\ CMMCEIPDataType.h

.NET Definition

Function Parameters

T tData[]

[OUTPUT] Retrieve array data of referenced tag. For example, refer to the section **16.10.7 EipGetAssembly**

T& tVar

[OUTPUT] The variable reference retrieved value.

17.22 TCP/IP and UDP/IP C++ User Libraries

Historically, in order for the user to communicate to the Maestro, only the following interfaces could be used:

- Modbus
- Ethernet/IP

If the user wished to define a propriety protocol, then managing socket experience was required. In addition, the maximum packet size, for instance, in Modbus is 256 bytes (128 registers). This was found to be limiting.

Elmo is therefore introducing two additional classes to the CPP library:

- CMMCUDP
- CMMCTCP

The user can select to use either Class, as both Classes operate in both the WIn32 and in Maestro Programming environments. The user can also select to be a client or Server and operate the class with or without callbacks. More importantly, no socket experience is required. The definition of the socket port is managed by the Class using a simple interface for the following commands:

- **int Create** (unsigned short usPort, SOCK_CLBK fnClbk=NULL, int iMsgMaxSize=512);
- **int Receive** (void * pData, unsigned short usSize, long lDelay=0L, sockaddr_in* pSockaddr=NULL);
- **int Send** (void * pData, unsigned short usSize, sockaddr_in* pSockaddr=NULL);
- **int Connect** (char* szAddr, unsigned short usPort, bool& bWait);
- bool IsWriteable();
- bool IsReadable(int iTimeOut=0);

In addition to the UDP and TCPIP classes, we also added a class eanbling easy sending / receiving of data to the drive via EoE.

The following C++ functions were added (similar to Binary interpreter):

- Elmo-Get-SetSyncParam (Float + integer)
- Elmo-Get-SetSyncArray (Float + integer)
- Elmo-Get-SetASyncParam (Float + integer)
- Elmo-Get-SetASyncArray (Float + integer)

17.23 The CMMCUDP class

The class CMMCUDP wraps the host communication functions detailed in the section **16.2 Host Communication on page 1212**. This section describes the public interface for UDP.

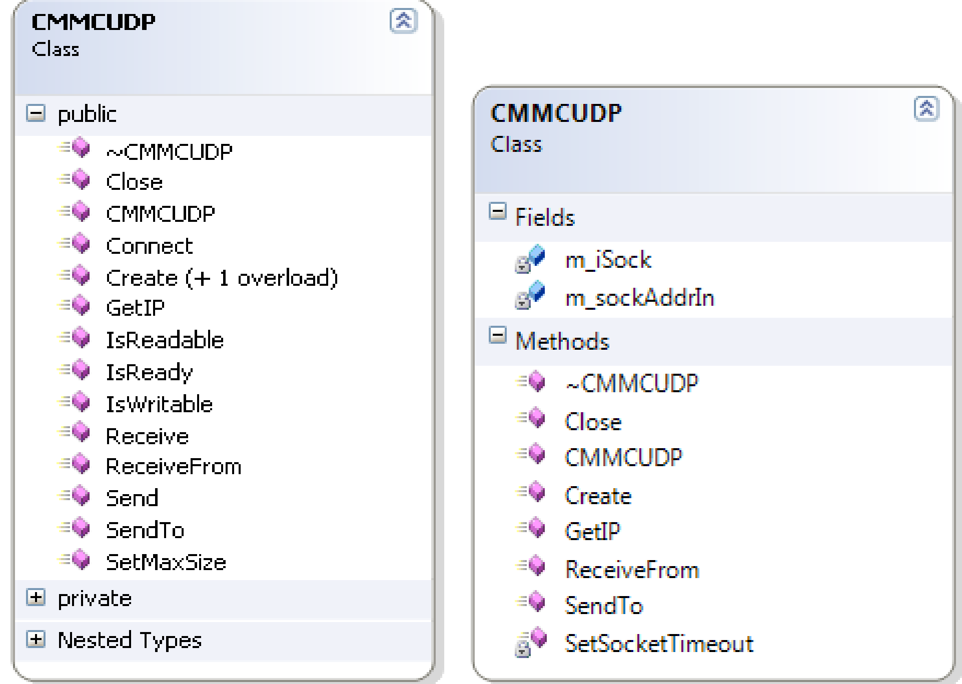


Figure 17-28 Fields and methods of the CMMCUDP class

The detailed class view shown in **Figure 17-28** describes the public user interface associated with the CMMCUDP class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.23.1 Synchronous and Asynchronous Behavior

When using the Receive (or ReceiveFrom) method, one of the user parameters delivered is 'timeout' (for delay). This parameter actually enables a blocking (as for Sync) or non-blocking process.

17.23.2 Mode Of Operation

The CMMCUDP class allows the user to operate in two modes on the server side; Simple and Call-back mode. Using the Simple mode of operation, the user personally manages the calls sequence (Create, Receive, Send, etc…). However, call-back mode is more complicated, from one perspective, the user receives an event for every arriving message via the call-back function he supplies. From the other perspective, the user must be able to synchronize the call-back's thread with his own main thread. The mode of operation is set by the Create method, as long as the call-back parameter is not NULL.

17.23.3 UDP Code Examples

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <c++/4.1.0/iostream>

#include <mmcpplib.h>

using namespace std;

//states enumeration for state machine
typedef enum {eSOCKET_CREAT_STATE=0, eSOCKET_ACCEPT_STATE, eSOCKET_IDLE_STATE,
eSOCKET_READ_STATE, eSOCKET_SEND_STATE, eSOCKET_CONNECT_STATE, eSOCKET_WRITABLE_STATE}
EDEMO_SOCK_STATE;

//message type definition - user proprietary.
typedef struct _sockdemo_msg_t {
     int _iID;
     char _szUserData[128];
} dsock_msg_t ;


/*-----------------------------------------------
 * global variables declaration
 */
static char                    g_szIpAddress[16];
static unsigned short g_usPort;


/**
 * this function demonstrate a UDP none blocking client.
 * it should be called from the main program cyclically (for instance every 5 ms or more).
 */
void UDPDemoClient()
{
     static EDEMO_SOCK_STATE eState = eSOCKET_CONNECT_STATE;
     static CMMCUDP udpClient;
     static dsock_msg_t msg;
     static int iCounter = 0;
     int rc;
     bool bWait;

     switch (eState) {
          case eSOCKET_CONNECT_STATE: //connect
               rc = udpClient.Connect(g_szIpAddress, g_usPort, bWait);
```

```
            cerr << __func__ << ": connect result = " << rc << ", g_usPort = " << g_usPort <<
", bWait = " << bWait << endl;
            if (rc) {
                cerr << __func__ << ": connecting to server failed. g_usPort = " << g_usPort
<< " exiting ..." << endl;
                raise(SIGTERM);
                break;
            }
//   if (bWait==true) //if result OK and wait then wait a second for the connection to
complete.
//       usleep(1000000); another option is to check if socket is writable on the next stage
    eState = (bWait==true) ? eSOCKET_WRITABLE_STATE : eSOCKET_SEND_STATE;
            break;
        case eSOCKET_WRITABLE_STATE: //check if connection completed (writable)
            if (udpClient.IsWritable())
                eState = eSOCKET_SEND_STATE;
            break;

        case eSOCKET_SEND_STATE: //send
            memset(&msg, 0, sizeof (dsock_msg_t));
            msg._iID = iCounter;
            sprintf(msg._szUserData, "message ID is %d", iCounter++);
            rc = udpClient.Send((void *)&msg, sizeof(dsock_msg_t));
            if (rc>0)
                cerr << __func__ << ": sending client id = " << msg._iID << ": " <<
msg._szUserData << endl;
            eState = eSOCKET_READ_STATE;
            break;
        case eSOCKET_READ_STATE: //receive
            memset(&msg, 0, sizeof (dsock_msg_t));
            rc = udpClient.Receive((void *)&msg, sizeof(dsock_msg_t));
            if (rc > 0)
                cerr << __func__ << ": response id = " << msg._iID << ": " <<
msg._szUserData << endl;
            eState = eSOCKET_SEND_STATE;
            break;
        default:
            break;
    }
}


/**
 * this function demonstrate a UDP none blocking server.
 * it should be called from the main program cyclically (for instance every 5 ms or more).
 */
void UDPDemoServer()
{
    static EDEMO_SOCK_STATE eState = eSOCKET_CREAT_STATE;
    static CMMCUDP udpServer;
    static dsock_msg_t msg;
    int rc;

    switch (eState) {
        case eSOCKET_CREAT_STATE:
             //create listener.
            rc = udpServer.Create(g_usPort); //no callback, normal mode of operation
            fprintf(stderr, "%s, Create: rc = %d.\n", __func__, rc);
            eState = eSOCKET_READ_STATE; //normal mode of operation
            break;
        case eSOCKET_READ_STATE: //receive data from available clients (normal mode only)
            memset(&msg, 0, sizeof (dsock_msg_t));
            rc = udpServer.Receive(&msg, sizeof(dsock_msg_t), -1);
            if (rc > 0 ) {
                fprintf(stderr, "%s, receive: client ID = %d, message = %s\n", __func__,
msg._iID, msg._szUserData);
                eState = eSOCKET_SEND_STATE;            //next stage for response available
clients.
            }
            //otherwise repeat on read state.
```

```
            break;
        case eSOCKET_SEND_STATE:                    //send response to  available clients
            rc = udpServer.Send(&msg, sizeof (dsock_msg_t));
            if (rc > 0)
                fprintf(stderr, "%s, sending response id = %d : %s\n", __func__, msg._iID,
msg._szUserData);
            else
                fprintf(stderr, "%s, send: failed\n", __func__);
            eState = eSOCKET_READ_STATE;
            break;
        default:
            break;
    }
}
```

17.23.4 CMMCUDP Class Functions Code Example

```
/*
============================================================================
Collection of Gmas API functions (Set #3)
Examples for document: "G-MAS Administrative and Motion API.pdf".
12Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection          gConn;
     MMC_CONNECT_HNDL        ComHndl;

     CMMCSingleAxis          AxisA,    AxisB;
     unsigned short               AxisARef, AxisBRef;

     CMMCGroupAxis           Group;
     MMC_MOTIONPARAMS_GROUP  ParamsGroup;

     MMC_SETKINTRANSFORM_IN  SetKin;

    char  * delimit = "===========================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroConnection(int);
     void      ConnectionTypeAndNum(void);
     void      SendReciveFromEthercat(int NumAmp);
     void      SetGetDefDigOutput(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
     int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus);
     void      EndMotionEventCB(unsigned short usAxisRef);
     void ModbusWrite_Received();
     void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
```

```
/*================================= Administration functions STR ===================================*/

int       main(int)
// ==============
{
     int  trace = 1;

     printf("\n %s", delimit);
     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

     try
     {
          SnroConnection(trace++);
          SnroMoveAbsolute(trace++);
          SnroEnableDisableMotionEndedEvent(trace++);
          SnroDepthName(trace++);

     }
     catch (CMMCException excp)
     {
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          exit(0);
     }

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//=====================================================================
{
                int end_of = 0;
                int iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
          end_of = 1;
/* Read Axis Status command server for specific Axis */
          ulState = sng_axis->ReadStatus();
          if (!(ulState & break_state))
          {
               end_of = 0;

               WAIT_SLEEP_MILLI(20)
          }
     }

//        MMC_SHOWNODESTAT_IN  showin;
//        MMC_SHOWNODESTAT_OUT showout;
//        MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

     return 0;
}


//   15.5.2.   RegisterRTE    Page 1274
void initAdminSingleAxis(void)
// ==============================
{
     int       iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;

/* CallbackFunc in ConnectIPCEx call if there    */
```

```
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Put Null param Val for no CallbackFunc        */
/*   ComHndl = gConn.ConnectIPCEx(iEventMask, NULL); */
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */


/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk    = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute     = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


void initAdminMultiAxis()
// =========================
{
//  Source class:
//       MMC_CONNECT_HNDL         ComHndl;
//       CMMCSingleAxis           AxisA,    AxisB;
//       CMMCGroupAxis            Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// =============================
{
     MMC_CloseConnection(ComHndl) ;
}


     void endAdminMultiAxis(void)
// ================================
{
//  Source class:
//       CMMCGroupAxis  Group;

     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
     Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================ Administration functions END ====================================*/

/*================================= Scenario functions STR ========================================*/
void SnroMoveAbsolute(int trace)
// ================================
{
```

```
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

     MoveAbsoluteMoves();

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void SnroEnableDisableMotionEndedEvent(int trace)
// ================================================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);
                              /* Register the callback function for Modbus and Emergency: */
     gConn.RegisterEventCallback(MMCPP_MODBUS_WRITE,(void*)ModbusWrite_Received) ;
     gConn.RegisterEventCallback(MMCPP_EMCY,(void*)Emergency_Received) ;

     AxisA.PowerOn(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

     EnableDisableMotionEndedEvent();

     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
     endAdminSingleAxis();

     gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

//   15.4.14.  GroupEnable    Page 1208
//   15.4.15.  GroupDisable   Page 1209
void SnroDepthName(int trace)
// ============================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();
     initAdminMultiAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

     Group.GroupEnable();

     DepthName();

     Group.GroupDisable();

     AxisB.PowerOff(MC_BUFFERED_MODE);
     AxisA.PowerOff(MC_BUFFERED_MODE);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

     endAdminMultiAxis();
     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
```

```
void SnroConnection(int trace)
// ==============================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();

     ConnectionTypeAndNum();

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

/*================================= Example functions STR ==========================================*/


void EnableDisableMotionEndedEvent(void)
// ========================================
{
     int  loopInd;

printf("\n Function: %s:", __func__);
     for (loopInd = 0; loopInd < 2; loopInd++)
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}


//   16.4.14.  GroupEnable        1208
//   16.4.15.  GroupDisable   1209
void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

     iVal3 = Group.GetGroupAxisByName("v01");     /* Expected 256 */
```

```
     printf("\n +++++ Reff: a01=%d a02=%d, v01=%d +++++", iVal1, iVal2, iVal3);
/*
*    iVal2 = Group.GetGroupAxisByName("v02");
*/
}


void MoveAbsoluteMoves(void)
// ============================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:          */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0        */
                              /* update default speed to 5000000          */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters              */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity        */
                              /* v=100000    which become the new def V        */
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0         */
                              /* new def v=80000                             */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:      */
                              /*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=              */
/* MC_BUFFERED_MODE (def)     */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
/* Move abs to 350000 with parameters from  */
/* above command which become the default:  */
/*       Speed 500000, Acc=1000000               */
/*       Dec=1500000, Jerk=20000000,          */
/*       buffer mode=MC_BUFFERED_MODE (def)   */
     AxisA.MoveAbsolute(350000);
}


int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{

    printf("\n *********** STR Func: %s *********** ", __func__);

                              /* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
```

```
          printf("\n Underflow event received ");
          break ;
     case MODBUS_WRITE_EVT:
          printf("\n ModBus Write event received ");
          break ;
     case TOUCH_PROBE_ENDED_EVT:
          printf("\n Touch Probe event received ");
          break ;
     default:
          printf("\n Default.... Whatever arrived event received ");
          break;
     }

    printf("\n *********** END Func: %s *********** ", __func__);
     fflush(stdout); fflush(stderr);

     return 1 ;
}

int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
//
=========================================================================================================
{
     printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
             msg, usAxisRef, sErrorID, usStatus);
     fflush(stdout); fflush(stderr);

     MMC_CloseConnection(uiConnHndl);
     exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
     printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
     printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
     fflush(stdout); fflush(stderr);
}


/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ==========================
{
     printf("\n %s Received ", __func__) ;
     fflush(stdout); fflush(stderr);
}


                              /* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ================================================================
{
     printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
     fflush(stdout); fflush(stderr);
}
```

```
//   15.6.11. SetHeartBeatConsumer      1248
//   15.7.4.  GetNetworkInfo            1337
enum
{
     eCOMM_TYPE_NONE = 0,
     eCOMM_TYPE_ETHERCAT,
     eCOMM_TYPE_CAN
};


void ConnectionTypeAndNum(void)
// ===============================
//
{
     unsigned int                  uiHeartbeatTimeFactor;
     int                                rt;
     int                                NumFoundAmp;
     char *                             cErrotStr;
     CMMCNetwork                        CNet;
     MMC_NETWORKINFO_OUT           CanOutParams;       // Can drv connection
     MMC_GETCOMMSTATISTICS_OUT     EthCatOutParams;    // Ethcat drv connection


     printf("\n Function: %s ", __func__);

     CNet.SetConnHndl(ComHndl);
                              /* Connection type - CAN/EtherCAT */
     rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);

     if (rt == eCOMM_TYPE_ETHERCAT)
     {
                              /* !!! ETERCAT !!!*/
          rt = CNet.GetCommStatistic(EthCatOutParams);
          if (rt != 0)
          {
               cErrotStr = "EtherCat failed, get statistic";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = (int)EthCatOutParams.usNumOfSlaves;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through ETERCAT net. ",
NumFoundAmp);

                              /* Send and recive from UDP soket to driv */
          SendReciveFromEthercat(NumFoundAmp);
     }
     else if (rt == eCOMM_TYPE_CAN)
     {
/* !!! CAN !!!*/
/* !!!  hard bit should be set in resource file*/
/* take from the resource file and actual connected...  */
          uiHeartbeatTimeFactor = 1;    /* On for Every Cycle */
          gConn.SetHeartBeatConsumer(uiHeartbeatTimeFactor);

         rt = CNet.GetNetworkInfo  (CanOutParams);
          if (rt != 0)
          {
               cErrotStr = "Can failed, get NetworkInfo";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = CanOutParams.iNumOfActiveNodes;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through CAN net. ", NumFoundAmp);
     }

/* Get & Set Default Mapping Digital Output */
     SetGetDefDigOutput();

     return;

ConnectionTypeAndNum_exit_err:
     printf("\n>>> %s: *** %s %d ", __func__, cErrotStr, rt);
     return;
}
```

```
/* Should create sockets according to actual number of amplifire (param NumAmp)     */
/* ...for demo (examples etc...) assume at least two Amp. exist.                    */
//   15.18.5.  Create         1397
//   15.18.6.  SendTo         1397
//   15.18.7.  ReceiveFrom    1398
//
//   16.18.11. ElmoGetArray        1310
//   16.18.12. ElmoGetParameter    1311

void SendReciveFromEthercat(int NumAmp)
// ========================================
{
     int            rt_val = 0;
    int             iBv;
     bool           bWait;
     float          fValue;
                         /* IPC type of app. */
     char           sAxisName[50] ;
     CMMCUDP    cUDP1,
                    cUDP2;
     CMMCEoE        gEoe;

     printf("\n Function: %s ", __func__);

     rt_val = cUDP1.Create("192.168.1.5", 5001, 0); /* Ip of first drive (Gmas "last part IP" + 1) */
     rt_val = cUDP1.SendTo("vr\r", 3);
     rt_val = cUDP1.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 0 Version: <%s> ", sAxisName);

     rt_val = cUDP2.Create("192.168.1.6", 5001); /* Ip of second Gmas     */
     rt_val = cUDP2.SendTo("vr\r", 3) ;
     rt_val = cUDP2.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 1 Version: <%s> ", sAxisName);


     rt_val = gEoe.Connect("192.168.1.5", 5001, bWait);
     if (bWait == true)
     {
          usleep(10000);                         /* Wait 10 mili */
          if (gEoe.IsWritable() == false)
          {
               printf("\n>>> %s: *** Amp (Drv) connection it not writable... ", __func__);
          }
     }
/* AN[6] is command for get PsHv   */
/* return 0 on succceed otherwise 1.*/
     rt_val = gEoe.ElmoGetArray("AN", 6, fValue);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get EthCat Driver psHv AN[6] ", __func__);
     }
     else
     {
          printf("\n>>> Max Power Supply High Voltage for this driver (ip=192.168.1.5) is %3.1f ",
fValue);
     }

/* BV - Maximum Motor DC Voltage (return int)    */
/* return 0 on succeed, otherwise 1.              */
     rt_val = gEoe.ElmoGetParameter("BV", iBv);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get Bus Driver HV ", __func__);
     }
     else
     {
          printf("\n>> Actual Bus Driver (ip=192.168.1.5) Hv is %d ", iBv);
     }

     gEoe.Close();
}
```

```
//   16.3.27.  GetDigOutputs32Bit  1168
//   16.3.29.  SetDigOutputs32Bit  1169
                              /* Get & Set Default Mapping Digital Output */
void SetGetDefDigOutput(void)
// ============================
{
unsigned long       ulDigOutputs32bit;



     printf("\n Function: %s ", __func__);

                              /* Read Digital output group 0 state */
     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: B4 action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned
int)ulDigOutputs32bit);


/* Change specific bit state of digital Output group 0 */
     if ((ulDigOutputs32bit & 0x10000) != 0x00000)
     {
/* ReSet specific bit of Digital output group 0 to state "0" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit & 0xfffeffff);   /* E.g: disconnedt ps... */
     }
     else
     {
/* Set specific bit of Digital output group 0 to state "1" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit | 0x10000);     /* E.g: connect ps... */
     }


     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: Aft action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned
int)ulDigOutputs32bit);
}

/*================================= Example functions END ===========================================*/

/*================================= Output STR ============================================*/
#ifdef PROGRAM_OUTPUT
#endif /* PROGRAM_OUTPUT */
/*================================= Output END ============================================*/
```

17.23.5 Create

Create a connection to the Maestro, given the host IP address, and Port number. This is an old convention to obtain UPD client connection with UDP blocking server. See Connect API for connecting UDP non blocking server.

int Create (
char * cIP,
int iPort,
int iGMASPort = 0
);

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Function Parameters

cIP

IP address. Char accepted in xx:xx:xx:xx format

iPort

Port connection used. Integers accepted

iGMASPort = 0

Maestro Port reverted to 0.

For code example, refer to the section **17.23.4**.

17.23.6 SendTo

Sends a message to the Maestro.

```
int SendTo(
char* msg,
short sLength
);
```

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Function Parameters

msg

Message sent. Null terminated string accepted

sLength

Length of the message. String length

For code example, refer to the section **17.23.4**.

17.23.7 ReceiveFrom

Recieves a message from the Maestro.

```
int ReceiveFrom(
char* msg,
short sLength,
short sTimeout
);
```

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Function Parameters

msg

Message sent. Null terminated string accepted

sLength

Length of the message. String length.

sTimeout

Timeout is ending when waiting for a message (milliseconds).

For code example, refer to the section **17.23.4**.

17.23.8 IsWriteable

This verifies that the connection is writable.

```
int IsWritable(
) {}
```

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Return

Returns 1 if writeable, otherwise 0.

17.23.9 IsReady

This verifies that the connection is readable.

```
Int IsReady(
) {}
```

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Return

Returns 1 if writeable, otherwise 0.

17.23.10 Connect

This method creates a UDP connection to remote none blocking UDP server.

```
int Connect (
char* szAddr,
unsigned short usPort,
bool& bWait,
int iMsgMaxSize
) {}
```

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Function Parameters

szAddr

IP address of the server

usPort

Ethernet port number

bWait

Set to TRUE if connection is not fully established (none blocking server on the other side). In this situation, one may have to wait for a couple of milliseconds. It is recommended to is use IsWritable() on the next cycle as part of the main state machine.

iMsgMaxSize

Message maximum size. Default value is 512 bytes.

Return

OK (0) or ERROR

17.23.11 Send

This method sends a UDP message to none blocking UDP server.

int Send (
void * pData,
unsigned short usSize,
sockaddr_in* pSockaddr
)

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Function Parameters

pData

A pointer to data (message) to send

usSize

The message size

pSockaddr

A pointer to the socket address. Default is NULL.

On call-back mode it may point to the socket address with data from the last Receive call.

Returns the number of bytes which has actually been sent, otherwise -1

17.23.12 Receive

This method receives UDP message into the buffer pointed by pData.

```
int Receive (
void * pData,
unsigned short usSize,
long lDelay=0L,
sockaddr_in* pSockaddr=NULL
)
```

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Function Parameters

pData

Pointer to buffer, which will store the received data.

usSize

Message size to read.

lDelay

Delay time. default is 0 (no delay)

pSockaddr

Pointer to socket address. default is NULL. On call-back mode it is used by Send by for synchronous matters.

Return

Number of read bytes, otherwise -1

17.23.13 Create (overloaded)

This method creates none blocking UDP server (listener).

int Create (
unsigned short usPort,
SOCK_CLBK fnClbk,
int iMsgMaxSize
)

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Function Parameters

usPort

Port number to listen on (bind with)

fnClbk

User call-back function. Relevant only for call-back mode of operation.

lDelay

Delay time. default is 0 (no delay)

iMsgMaxSize

Largest possible message (in bytes).

Return

0 on success, -1 otherwise. socket number (iSock) is update on success, otherwise -1;

17.23.14 GetIP

Retrieves inet address structure, which is associated to UDP connection.

in_addr GetIP(
)

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

17.23.15 SetMaxSize

Set default value for message maximum size (byes).

void SetMaxSize(
int iSize
)

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Function Parameters

iSize

Default value for message maximum size (byes)

17.23.16 SetTimeout

Sets the default time out for the UDP socket.

void SetTimeout(
long lTimeOut
)

Source GMAS\includes\CPP\MMCUDP.h

.NET Definition

Function Parameters

lTimeOut

The default time out (milliseconds) for the UDP socket. Any value is acceptable.

17.24 The CMMCTCP class

The class CMMCTCP implements TCP host communication with the Maestro and similarly from the Maestro to the host system. This section describes the public interface for TCP.

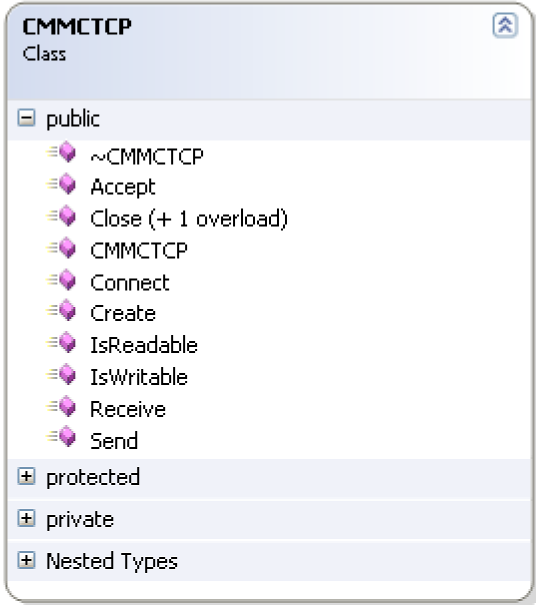


Figure 17-29 Fields and methods of the CMMCTCP class

The detailed class view shown in **Figure 17-29** describes the public user interface associated with the CMMCTCP class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

17.24.1 Synchronous and Asynchronous Behavior

When using the Receive method, one of the user parameters delivered is 'timeout' (for delay). This parameter actually enables a blocking (as for Sync) or non-blocking process.

17.24.2 Mode of Operation

CMMCTCP class allows to user two modes of operation on the server side: 'Simple' & Call-back mode. In the 'Simple' mode of operation, the user manages the calls sequence by himself (Create, Accept, Receive, Send, etc…). Call-back mode is more complicated. On one hand user receives an event for every new connection and/or arriving message via the call-back function he supplies. On the other hand user must be able to synchronize the call-back's thread with his own main thread. Mode of operation is set by Create method if the call-back parameter is not NULL.

17.24.3 Accept

This method obtains a new connection if pending.

int Accept(
)

Source GMAS\includes\CPP\MMCTCPP.h

.NET Definition

Function Parameters

None

Return

Returns a new connection from the socket listener, otherwise -1

17.24.4 IsReadable

This method verifies that a given socket connection is readable.

int IsReadable(
int iSock
)

Source GMAS\includes\CPP\MMCTCPP.h

.NET Definition

Function Parameters

iSock

Socket connection. A unique identifier for connections. Once open the connection amy be as a handler for other functions.

Return

Returns 1 if writeable, otherwise 0.

17.24.5 IsWritable

This method verifies that a given socket connection is writable.

```
int IsWritable(
int iSock
)
```

Source GMAS\includes\CPP\MMCTCPP.h

.NET Definition

Function Parameters

iSock

Socket connection. A unique identifier for connections. Once open the connection amy be as a handler for other functions.

Return

Returns 1 if writeable, otherwise 0.

17.24.6 Connect

This method creates a TCP connection to remote none blocking TCP server.

```
int Connect (
char* szAddr,
unsigned short usPort,
int& iSock,
bool& bWait
) {}
```

Source GMAS\includes\CPP\MMCTCPP.h

.NET Definition

Function Parameters

szAddr

IP address of the server

usPort

Ethernet port number

iSock

Socket connection. A unique identifier for connections. Once open the connection amy be as a handler for other functions. Stores the created socket if returned OK

bWait

Set to TRUE if connection is not fully established (none blocking server on the other side).

In this situation, one may have to wait for couple of milliseconds. It is recommended to use IsWritable() on the next cycle as part of the main state machine.

iMsgMaxSize

Largest possible message (in bytes). Default value is 512 bytes

Return

OK (0) or ERROR

17.24.7 Send

This method receives TCP message from a given socket connection.

int Send (
int iSock,
unsigned short usSize,
void *pData
)

Source GMAS\includes\CPP\MMCTCPP.h

.NET Definition

Function Parameters

iSock

Client socket connection. A unique identifier for connections. Once open the connection amy be as a handler for other functions.

usSize

The message size

pData

A pointer to data (message) to send

Return

Number of bytes, which actually has been sent, -1 otherwise

17.24.8 Receive

This method receives TCP message from a given socket connection.

int Receive (

int iSock,
unsigned short usSize,
void* pData,
long lDelay
)

Source GMAS\includes\CPP\MMCTCPP.h

.NET Definition

Function Parameters

iSock

Socket connection from which message is being read. A unique identifier for connections. Once open the connection amy be as a handler for other functions.

sSize

Expected size (bytes) to read

pData

Message buffer address to read message into.

Return

Number of read bytes. -1 otherwise.

17.24.9 Create

This method creates a none blocking TCP server (listener).

```
int Create (
unsigned short usPort,
SOCK_CLBK fnClbk,
int iMsgMaxSize
)
```

Source GMAS\includes\CPP\MMCTCPP.h

.NET Definition

Function Parameters

usPort

Port number to listen on (bind with)

fnClbk

User call-back function. Relevant only for call-back mode of operation.

iMsgMaxSize

Largest possible message (in bytes).

Return

0 on success, -1 otherwise.

17.24.10 TCP Code Examples

```
/**
 *  TCP client example
 */
#include <string.h>
#include <c++/4.1.0/iostream>
#include <mmcpplib.h>
using namespace std;

/*---------------------------------------
 * User definitions
 */
#define DEMO_DEST_ADDRESS "192.168.1.5"
#define DEMO_PORT 1117

//states enumeration for state machine
typedef enum {eSOCKET_CREAT_STATE=0, eSOCKET_ACCEPT_STATE, eSOCKET_IDLE_STATE, eSOCKET_READ_STATE,
eSOCKET_SEND_STATE, eSOCKET_CONNECT_STATE, eSOCKET_WRITABLE_STATE} EDEMO_SOCK_STATE;

//message type definition - user proprietary.
typedef struct _sockdemo_msg_t {
    int _iID;
    char _szUserData[128];
} dsock_msg_t ;

/*----------------------------------------------
 * global variables declaration
 */
static char        g_szIpAddress[16];
static unsigned short  g_usPort;

/**
*    this function should be called from the main program  periodically (for instance every 5 ms or
more).
*/
void TCPDemoClient()
{
    static int iCounter = 0;
    static int iSock;
    static dsock_msg_t msg;
    static CMMCTCP tcpClient;
    static EDEMO_SOCK_STATE eState = eSOCKET_CONNECT_STATE;
    bool  bWait;
    int rc;

    switch (eState) {
    case eSOCKET_CONNECT_STATE: //connect
        rc = tcpClient.Connect(g_szIpAddress, g_usPort, iSock, bWait);
        cout << __func__ << " Connect " << rc << ", port " << g_usPort << ", destination "<<
g_szIpAddress << endl;
        //if result OK and wait then wait for the connection to complete.
        //this natural behavior of non blocking socket.
        eState = (bWait==true) ? eSOCKET_WRITABLE_STATE : eSOCKET_SEND_STATE;
            break;
    case eSOCKET_WRITABLE_STATE: //check if connection completed (writable)
        if (tcpClient.IsWritable(iSock))
            eState = eSOCKET_SEND_STATE;
        break;
    case eSOCKET_SEND_STATE: //send
        memset(&msg, 0, sizeof (dsock_msg_t));
        msg._iID = iCounter;
        sprintf(msg._szUserData, "message ID is %d", iCounter++);
        rc = tcpClient.Send (iSock, sizeof(dsock_msg_t), (void *)&msg);
        if (rc>0)
    cerr << __func__ << ": sending client id = " << msg._iID << ": " << msg._szUserData <<  endl;
        eState = eSOCKET_READ_STATE;
        break;
```

```
	case eSOCKET_READ_STATE: //receive
		memset(&msg, 0, sizeof (dsock_msg_t));
		rc = tcpClient.Receive (iSock, sizeof(dsock_msg_t), (void *)&msg);
		if (rc>0)
		{
			eState = eSOCKET_SEND_STATE;
	cerr << __func__ << ": response id = " << msg._iID << ": " << msg._szUserData << endl;
		}
		break;
	default:
		break;
	}
}

/**
 * this function invokes the TCP server state machine.
 * it demonstrates control of one single connection. one may want to handle multiple
connection simultaneously. on that case he shall have to manage a global list of connections
and periodically call for Accept API to get pending connections.
 */
void TCPDemoServer() {
	static EDEMO_SOCK_STATE eState = eSOCKET_CREAT_STATE;
	int rc;
	static int iSock;
	static dsock_msg_t msg;
	static CMMCTCP tcpServer;

	switch (eState) {
	case eSOCKET_CREAT_STATE:
		 //create listener for several client connections simultaneously.
		//this sample handles only single one.
		rc = tcpServer.Create(g_usPort);
		cout << __func__ << " Create " << rc << ", port " << g_usPort << endl;
		eState = eSOCKET_ACCEPT_STATE;
		break;
	case eSOCKET_ACCEPT_STATE: //accept pending client connections one at a time.
	//client connection may be closed and new client may connect
	//please note: in case of several connections, one must manage a list of socket file
descriptors.
	//accept none blocking operation. -1 specifies no new client connection is pending.
		if ((rc = tcpServer.Accept()) > 0) {
			iSock = rc;
			cout << __func__ << " Accept " << iSock << endl;
		}
		eState = eSOCKET_READ_STATE;	//same single socket connection in this case.
		break;
	case eSOCKET_READ_STATE:		//receive data from available clients.
		memset(&msg, 0, sizeof (dsock_msg_t));	//clear message buffer.
		rc = tcpServer.Receive(iSock, sizeof (dsock_msg_t), (void*)&msg);
		if (rc>0){ //if anything was read
	cout << __func__ << " Receive: socket = " << iSock << ", rc = " << rc << ", msg "<<
msg._szUserData << endl;
			eState = eSOCKET_SEND_STATE; //next stage for sending response to available clients.
		} else {
			//client may close/reconnect as he wishes
			eState = eSOCKET_ACCEPT_STATE;
		}
		break;
	case eSOCKET_SEND_STATE:	//send response to  available clients
		tcpServer.Send(iSock, sizeof (dsock_msg_t), (void *)&msg);
		eState = eSOCKET_ACCEPT_STATE; //client may close/reconnect as he wishes
		break;
	default:
		break;
	}
}
```

17.25 The CMMCEoE Class

CMMCEoE is a subclass of CMMCUDP. Therefore any public UDP function is operable, and may be used via the CMMCEoE class. Other user interfaces of CMMCEoE implements the C library API for binary interpreter using the C++ method. The EoE class is derived from the CMMCUDP class and uses EoE communication over UDP. This section describes the public interface for EoE. This section describes the public interface for EoE.

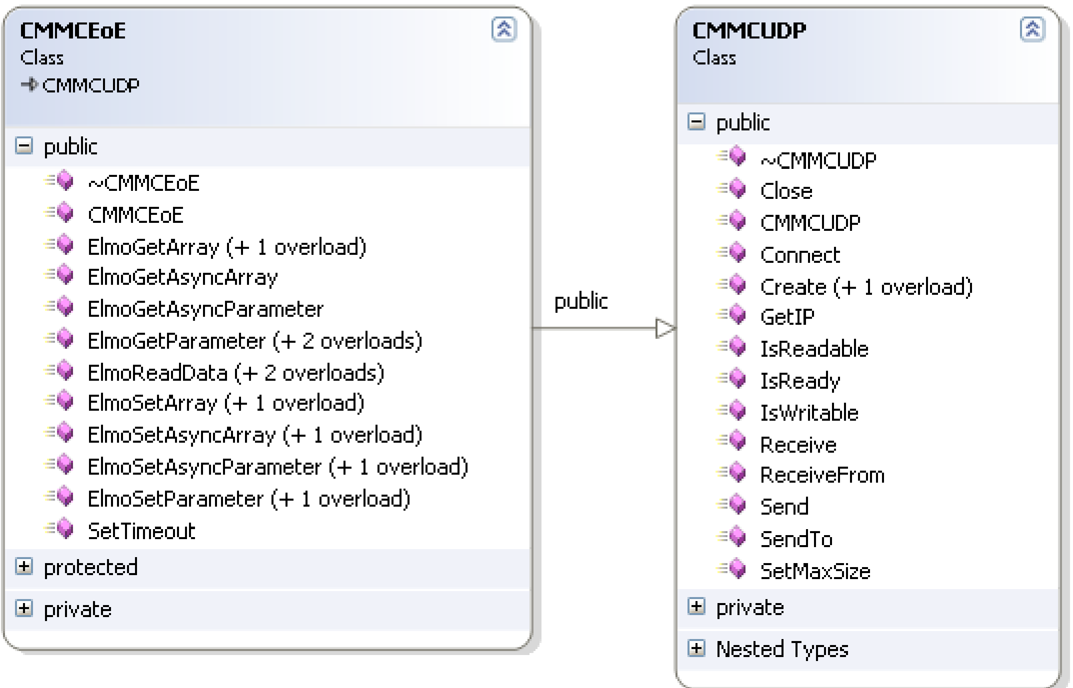


Figure 17-30 Fields and methods of the CMMCEoE class

The detailed class view shown in Figure 17-30 describes the public user interface associated with the CMMCEoE class. It should be noted that Private and Protected functions and their operation should be transparent to the user, and are not for general application by the user.

The following buffer and error definitions apply:

EoE_BUFFER_SIZE 128

EoE_NOERR_ERR 0

EoE_DRV_ERR -1

EoE_SENDCMD_ERR -2

EoE_RETRIES_ERR -3

EoE_SETASYNC_ERR -4

EoE_GETASYNC_ERR -5

EoE_SETSYNC_ERR -6

EoE_GETSYNC_ERR -7

17.25.1 CMMCEoE Class Functions Code Example 1

```
/*
============================================================================
Collection of Gmas API functions (Set #3)
Examples for document: "G-MAS Administrative and Motion API.pdf".
12Sep2013
Haim Hillel
============================================================================
*/

#include <iostream>

#include "MMC_Definitions.h"
#include "mmcpplib.h"

#define   EndMotionEventCB_MESSAGE "!!!!END MOTION EVENT MESSAGE!!!"


#ifdef WIN32
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   Sleep(WAIT_MILLI_SEC);
#else
#define WAIT_SLEEP_MILLI(WAIT_MILLI_SEC)   usleep(WAIT_MILLI_SEC*1000);
#endif


using namespace std;

     CMMCConnection           gConn;
     MMC_CONNECT_HNDL         ComHndl;

     CMMCSingleAxis           AxisA,    AxisB;
     unsigned short                AxisARef, AxisBRef;

     CMMCGroupAxis            Group;
     MMC_MOTIONPARAMS_GROUP   ParamsGroup;

     MMC_SETKINTRANSFORM_IN   SetKin;

    char  * delimit = "===========================================================================";
     char * strStrSnro = "\n\n\n <<<<<<<<<<<< Start ";
     char * strEndSnro1 = "\n End ";
     char * strEndSnro2 = " >>>>>>>>>>>>> ";

     int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis);

     void initAdminSingleAxis(void);
     void     endAdminSingleAxis(void);

     void initAdminMultiAxis();
     void  endAdminMultiAxis(void);

     void SnroEnableDisableMotionEndedEvent(int);
     void     EnableDisableMotionEndedEvent(void);

     void SnroMoveAbsolute(int);
     void     MoveAbsoluteMoves(void);

     void SnroDepthName(int);
     void      DepthName(void);

     void SnroConnection(int);
     void      ConnectionTypeAndNum(void);
     void      SendReciveFromEthercat(int NumAmp);
     void      SetGetDefDigOutput(void);

     int       CallbackFunc(unsigned char* recvBuffer, short recvBufferSize,void* lpsock);
     int       OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef,
short sErrorID, unsigned short usStatus);
     void      EndMotionEventCB(unsigned short usAxisRef);
     void ModbusWrite_Received();
     void Emergency_Received(unsigned short usAxisRef, short sEmcyCode);
```

```
/*================================ Administration functions STR ===================================*/

int       main(int)
// ==============
{
     int  trace = 1;

     printf("\n %s", delimit);
     printf("\n %s %s %s \n", __FILE__, __DATE__, __TIME__);

     try
     {
          SnroConnection(trace++);
          SnroMoveAbsolute(trace++);
          SnroEnableDisableMotionEndedEvent(trace++);
          SnroDepthName(trace++);

     }
     catch (CMMCException excp)
     {
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          printf("\n ERROR:  Axis=%d  <%s> error=%d, status=%d. ",excp.axisRef(), excp.what(),
(short)excp.error(), excp.status());
          printf("\n %s", delimit);
          printf("\n %s", delimit);
          exit(0);
     }

     printf("\n End of %s ", __FILE__);
     printf("\n %s\n\n", delimit);
     return 0;
}


int       WaitFbDone(unsigned int break_state, CMMCSingleAxis * sng_axis)
//===================================================================
{
                int end_of = 0;
                int iCount = 0;
     unsigned int   ulState;

     while( ! end_of)
     {
          iCount ++;
          end_of = 1;
/* Read Axis Status command server for specific Axis */
          ulState = sng_axis->ReadStatus();
          if (!(ulState & break_state))
          {
               end_of = 0;

               WAIT_SLEEP_MILLI(20)
          }
     }

//        MMC_SHOWNODESTAT_IN  showin;
//        MMC_SHOWNODESTAT_OUT showout;
//        MMC_ShowNodeStatCmd(ComHndl, sng_axis->GetRef(), &showin, &showout);

     return 0;
}


//   15.5.2.   RegisterRTE    Page 1274
void initAdminSingleAxis(void)
// ==============================
{
     int       iEventMask;

     MMC_MOTIONPARAMS_SINGLE stSingleDefault;

/* CallbackFunc in ConnectIPCEx call if there    */
```

```
/* is no calling to 'RegisterEventCallback'      */
     iEventMask = 0x7fffffff;
     ComHndl = gConn.ConnectIPCEx(iEventMask, (MMC_MB_CLBK)CallbackFunc);
/* Put Null param Val for no CallbackFunc        */
/*   ComHndl = gConn.ConnectIPCEx(iEventMask, NULL); */
/* Should Not calling, called inside 'ConnectIPCEx'   */
/*   rt_val = MMC_OpenUdpChannelCmdEx(g_ComHndl, &openudp_param_in, &openudp_param_out); */


/* Register Run Time Error Callback function*/
     CMMCPPGlobal::Instance()->RegisterRTE(OnRunTimeError);

     AxisA.InitAxisData("a01", ComHndl);

/* Init default Gmas Parameters       */
     stSingleDefault.fEndVelocity  = 0;
     stSingleDefault.dbDistance    = 100000;
     stSingleDefault.dbPosition    = 0;
     stSingleDefault.fVelocity     = 100000;
     stSingleDefault.fAcceleration = 2000000;
     stSingleDefault.fDeceleration = 10000000;
     stSingleDefault.fJerk    = 200000000;
/*  MC_POSITIVE_DIRECTION, MC_SHORTEST_WAY,     */
/* MC_NEGATIVE_DIRECTION, MC_CURRENT_DIRECTION  */
     stSingleDefault.eDirection    = MC_POSITIVE_DIRECTION;
     stSingleDefault.eBufferMode   = MC_BUFFERED_MODE;
     stSingleDefault.ucExecute     = 1;

     AxisA.SetDefaultParams(stSingleDefault);
}


void initAdminMultiAxis()
// =========================
{
//  Source class:
//       MMC_CONNECT_HNDL        ComHndl;
//       CMMCSingleAxis          AxisA,    AxisB;
//       CMMCGroupAxis           Group;


     AxisB.InitAxisData("a02", ComHndl);
     Group.InitAxisData("v01", ComHndl);

     AxisARef = AxisA.GetRef();
     AxisBRef = AxisB.GetRef();

     Group.AddAxisToGroup(AxisARef, NC_NODE_1_ID);
    Group.AddAxisToGroup(AxisBRef, NC_NODE_2_ID);
}


void endAdminSingleAxis(void)
// =============================
{
     MMC_CloseConnection(ComHndl) ;
}


     void endAdminMultiAxis(void)
// ================================
{
//  Source class:
//       CMMCGroupAxis  Group;

     Group.RemoveAxisFromGroup(NC_NODE_1_ID);
     Group.RemoveAxisFromGroup(NC_NODE_2_ID);
}
/*================================ Administration functions END ===================================*/

/*================================= Scenario functions STR ========================================*/
void SnroMoveAbsolute(int trace)
// ================================
{
```

```
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    MoveAbsoluteMoves();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

void SnroEnableDisableMotionEndedEvent(int trace)
// ================================================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   (void* )EndMotionEventCB);
                            /* Register the callback function for Modbus and Emergency: */
    gConn.RegisterEventCallback(MMCPP_MODBUS_WRITE,(void*)ModbusWrite_Received) ;
    gConn.RegisterEventCallback(MMCPP_EMCY,(void*)Emergency_Received) ;

    AxisA.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

    EnableDisableMotionEndedEvent();

    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);
    endAdminSingleAxis();

    gConn.RegisterEventCallback(MMCPP_MOTIONENDED,   NULL);

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

//   15.4.14.  GroupEnable    Page 1208
//   15.4.15.  GroupDisable   Page 1209
void SnroDepthName(int trace)
// ============================
{
    printf("%s%s -%d- ", strStrSnro, __func__, trace);

    initAdminSingleAxis();
    initAdminMultiAxis();

   AxisA.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);

   AxisB.PowerOn(MC_BUFFERED_MODE);
   WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);

    Group.GroupEnable();

    DepthName();

    Group.GroupDisable();

    AxisB.PowerOff(MC_BUFFERED_MODE);
    AxisA.PowerOff(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
    WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisA);

    endAdminMultiAxis();
    endAdminSingleAxis();

    printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}
```

```
void SnroConnection(int trace)
// ==============================
{
     printf("%s%s -%d- ", strStrSnro, __func__, trace);

     initAdminSingleAxis();

     ConnectionTypeAndNum();

     endAdminSingleAxis();

     printf("%s%s -%d- %s", strEndSnro1, __func__, trace, strEndSnro2);
}

/*================================ Example functions STR =========================================*/


void EnableDisableMotionEndedEvent(void)
// ========================================
{
     int  loopInd;

printf("\n Function: %s:", __func__);
     for (loopInd = 0; loopInd < 2; loopInd++)
     {
          if ((loopInd % 2) == 0)
          {
               printf("\n +++++++++ On end of motion     EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.EnableMotionEndedEvent();
          }
          else
          {
               printf("\n +++++++++ On end of motion NOT EXPECT: <%s> : ", EndMotionEventCB_MESSAGE);
               AxisA.DisableMotionEndedEvent();
          }

          printf("\n +++++++++ Motion started...");
          MoveAbsoluteMoves();
          WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisA);
          printf("\n +++++++++ Motion End \n");
     }
}


//   16.4.14.  GroupEnable         1208
//   16.4.15.  GroupDisable        1209
void DepthName(void)
// ====================
{
unsigned int   iVal1, iVal2, iVal3;


printf("\n Function: %s:", __func__);
     iVal1 = AxisB.GetFbDepth();

     Group.GroupDisable();
     AxisB.PowerOff(MC_BUFFERED_MODE);

     iVal2 = AxisB.GetFbDepth();
     WaitFbDone(NC_AXIS_DISABLED_MASK, &AxisB);
     iVal3 = AxisB.GetFbDepth();

     printf("\n +++++ oldFb=%d B4WaitDis=%d, AftWaitDis=%d +++++", iVal1, iVal2,iVal3);

    AxisB.PowerOn(MC_BUFFERED_MODE);
    WaitFbDone(NC_AXIS_STAND_STILL_MASK, &AxisB);
     Group.GroupEnable();

    iVal1 = AxisA.GetAxisByName("a01");/* Expected 0 */
    iVal2 = AxisB.GetAxisByName("a02"); /* Expected 1 */

     iVal3 = Group.GetGroupAxisByName("v01");    /* Expected 256 */
```

```
     printf("\n +++++ Reff: a01=%d a02=%d, v01=%d +++++", iVal1, iVal2, iVal3);
/*
*    iVal2 = Group.GetGroupAxisByName("v02");
*/
}


void MoveAbsoluteMoves(void)
// ============================
{
printf("\n Function: %s:", __func__);
/* Move to -400000 at default speed:        */
     AxisA.MoveAbsolute(-40000.0);
/* Move to -200000 at  speed 5000000.0      */
                              /* update default speed to 5000000          */
     AxisA.MoveAbsolute(-200000.0, 5000000.0);
/* Change the default parameters            */
     AxisA.m_fAcceleration = 1000000.0;
     AxisA.m_fDeceleration = 5000000.0;
     AxisA.m_fVelocity = 100000.0;
/* Move to -300000 at default velocity      */
                              /* v=100000    which become the new def V        */
     AxisA.MoveAbsolute(-300000.0);
/* Move to 310000 at velocity 80000.0       */
                              /* new def v=80000                           */
     AxisA.MoveAbsolute(310000.0, 80000.0);
/* Move abs to: 400000, with parameters:    */
                              /*    Speed=500000, Acc=1000000, Dec=1500000,*/
/*   Jerk=20000000, buffer mode=            */
/* MC_BUFFERED_MODE (def)     */
     AxisA.MoveAbsolute(400000, 500000, 1000000, 1500000, 20000000);
/* Move abs to 350000 with parameters from  */
/* above command which become the default:  */
/*       Speed 500000, Acc=1000000               */
/*       Dec=1500000, Jerk=20000000,        */
/*       buffer mode=MC_BUFFERED_MODE (def)  */
     AxisA.MoveAbsolute(350000);
}


int      CallbackFunc(unsigned char* recvBuffer, short recvBufferSize, void* lpsock)
// ================================================================================
{

    printf("\n *********** STR Func: %s *********** ", __func__);

                              /* Which function ID was received ... */
     switch(recvBuffer[1])
     {
     case ASYNC_REPLY_EVT:
          printf("\n ASYNC event Reply ");
          break ;
     case EMCY_EVT:
          printf("\n Emergency Event received ");
          break ;
     case MOTIONENDED_EVT:
          printf("\n Motion Ended Event received ");
          break ;
     case HBEAT_EVT:
          printf("\n H Beat Fail Event received ");
          break ;
     case PDORCV_EVT:
          printf("\n PDO Received Event received - Updating Inputs ");
          break ;
     case DRVERROR_EVT:
          printf("\n Drive Error Received Event received ");
          break ;
     case HOME_ENDED_EVT:
          printf("\n Home Ended Event received ");
          break ;
     case SYSTEMERROR_EVT:
          printf("\n System Error Event received ");
     case TABLE_UNDERFLOW_EVT:
```

```
         printf("\n Underflow event received ");
         break ;
     case MODBUS_WRITE_EVT:
         printf("\n ModBus Write event received ");
         break ;
     case TOUCH_PROBE_ENDED_EVT:
         printf("\n Touch Probe event received ");
         break ;
     default:
         printf("\n Default.... Whatever arrived event received ");
         break;
     }

    printf("\n *********** END Func: %s *********** ", __func__);
     fflush(stdout); fflush(stderr);

     return 1 ;
}

int      OnRunTimeError(const char *msg,  unsigned int uiConnHndl, unsigned short usAxisRef, short
sErrorID, unsigned short usStatus)
//
=========================================================================================================
{
     printf("\n APP: MMCPPExitClbk: Run time Error in function %s, axis ref=%d, err=%d, status=%d,
bye\n",
             msg, usAxisRef, sErrorID, usStatus);
     fflush(stdout); fflush(stderr);

     MMC_CloseConnection(uiConnHndl);
     exit(0);
}


void      EndMotionEventCB(unsigned short usAxisRef)
// ===============================================
{
     printf("\n Function: %s: usAxisRef=%d ", __func__, (int)usAxisRef);
     printf("\n\t\t %s \n", EndMotionEventCB_MESSAGE);
     fflush(stdout); fflush(stderr);
}


/* Callback Function once a Modbus message is received. */
void ModbusWrite_Received()
// ==========================
{
     printf("\n %s Received ", __func__) ;
     fflush(stdout); fflush(stderr);
}


                             /* Callback Function once an Emergency is received. */
void Emergency_Received(unsigned short usAxisRef, short sEmcyCode)
// ==================================================================
{
     printf("\n %s: Received on Axis %d. Code: %x ", __func__, usAxisRef, sEmcyCode) ;
     fflush(stdout); fflush(stderr);
}
```

```
//   15.6.11. SetHeartBeatConsumer      1248
//   15.7.4.  GetNetworkInfo            1337
enum
{
     eCOMM_TYPE_NONE = 0,
     eCOMM_TYPE_ETHERCAT,
     eCOMM_TYPE_CAN
};


void ConnectionTypeAndNum(void)
// ===============================
//
{
     unsigned int                  uiHeartbeatTimeFactor;
     int                                rt;
     int                                NumFoundAmp;
     char *                             cErrotStr;
     CMMCNetwork                        CNet;
     MMC_NETWORKINFO_OUT           CanOutParams;       // Can drv connection
     MMC_GETCOMMSTATISTICS_OUT     EthCatOutParams;    // Ethcat drv connection


     printf("\n Function: %s ", __func__);

     CNet.SetConnHndl(ComHndl);
                              /* Connection type - CAN/EtherCAT */
     rt = (int)gConn.GetGlobalBoolParameter(MMC_CONNECTION_TYPE_PARAM, 0);

     if (rt == eCOMM_TYPE_ETHERCAT)
     {
                              /* !!! ETERCAT !!!*/
          rt = CNet.GetCommStatistic(EthCatOutParams);
          if (rt != 0)
          {
               cErrotStr = "EtherCat failed, get statistic";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = (int)EthCatOutParams.usNumOfSlaves;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through ETERCAT net. ",
NumFoundAmp);

                              /* Send and recive from UDP soket to driv */
          SendReciveFromEthercat(NumFoundAmp);
     }
     else if (rt == eCOMM_TYPE_CAN)
     {
/* !!! CAN !!!*/
/* !!!  hard bit should be set in resource file*/
/* take from the resource file and actual connected...   */
          uiHeartbeatTimeFactor = 1;    /* On for Every Cycle */
          gConn.SetHeartBeatConsumer(uiHeartbeatTimeFactor);

         rt = CNet.GetNetworkInfo  (CanOutParams);
          if (rt != 0)
          {
               cErrotStr = "Can failed, get NetworkInfo";
               goto ConnectionTypeAndNum_exit_err;
          }
          NumFoundAmp = CanOutParams.iNumOfActiveNodes;
          printf("\n >>>>>>>>>>> %d drivers are connecting to GMAS through CAN net. ", NumFoundAmp);
     }

/* Get & Set Default Mapping Digital Output */
     SetGetDefDigOutput();

     return;

ConnectionTypeAndNum_exit_err:
     printf("\n>>> %s: *** %s %d ", __func__, cErrotStr, rt);
     return;
}
```

```
/* Should create sockets according to actual number of amplifire (param NumAmp)     */
/* ...for demo (examples etc...) assume at least two Amp. exist.                    */
//   16.18.1.  Create         1302
//   16.18.2.  SendTo         1303
//   16.18.3.  ReceiveFrom    1304
//
//   15.20.8.  ElmoGetArray       1429
//   15.0.9.   ElmoGetParameter   1430

void SendReciveFromEthercat(int NumAmp)
// ======================================
{
     int            rt_val = 0;
    int             iBv;
     bool           bWait;
     float          fValue;
                         /* IPC type of app. */
     char           sAxisName[50] ;
     CMMCUDP    cUDP1,
                    cUDP2;
     CMMCEoE        gEoe;

     printf("\n Function: %s ", __func__);

     rt_val = cUDP1.Create("192.168.1.5", 5001, 0); /* Ip of first drive (Gmas "last part IP" + 1) */
     rt_val = cUDP1.SendTo("vr\r", 3);
     rt_val = cUDP1.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 0 Version: <%s> ", sAxisName);

     rt_val = cUDP2.Create("192.168.1.6", 5001); /* Ip of second Gmas     */
     rt_val = cUDP2.SendTo("vr\r", 3) ;
     rt_val = cUDP2.ReceiveFrom(sAxisName, 50, 100);
     sAxisName[rt_val] = 0;
     printf("\n Axis 1 Version: <%s> ", sAxisName);


     rt_val = gEoe.Connect("192.168.1.5", 5001, bWait);
     if (bWait == true)
     {
          usleep(10000);                         /* Wait 10 mili */
          if (gEoe.IsWritable() == false)
          {
               printf("\n>>> %s: *** Amp (Drv) connection it not writable... ", __func__);
          }
     }
/* AN[6] is command for get PsHv   */
/* return 0 on succceed otherwise 1.*/
     rt_val = gEoe.ElmoGetArray("AN", 6, fValue);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get EthCat Driver psHv AN[6] ", __func__);
     }
     else
     {
          printf("\n>>> Max Power Supply High Voltage for this driver (ip=192.168.1.5) is %3.1f ",
fValue);
     }

/* BV - Maximum Motor DC Voltage (return int)    */
/* return 0 on succeed, otherwise 1.             */
     rt_val = gEoe.ElmoGetParameter("BV", iBv);
     if (rt_val != 0)
     {
          printf("\n>>> %s: *** Can't get Bus Driver HV ", __func__);
     }
     else
     {
          printf("\n>> Actual Bus Driver (ip=192.168.1.5) Hv is %d ", iBv);
     }

     gEoe.Close();
}
//   16.3.27.  GetDigOutputs32Bit      1168
```

```
//   16.3.29.  SetDigOutputs32Bit      1169
                              /* Get & Set Default Mapping Digital Output */
void SetGetDefDigOutput(void)
// ============================
{
unsigned long       ulDigOutputs32bit;



     printf("\n Function: %s ", __func__);

                              /* Read Digital output group 0 state */
     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: B4 action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned
int)ulDigOutputs32bit);


/* Change specific bit state of digital Output group 0 */
     if ((ulDigOutputs32bit & 0x10000) != 0x00000)
     {
/* ReSet specific bit of Digital output group 0 to state "0" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit & 0xfffeffff);   /* E.g: disconnedt ps... */
     }
     else
     {
/* Set specific bit of Digital output group 0 to state "1" */
          AxisA.SetDigOutputs32Bit(ulDigOutputs32bit | 0x10000);     /* E.g: connect ps... */
     }


     ulDigOutputs32bit = AxisA.GetDigOutputs32bit(0);
     printf("\n>>> %s: Aft action: DigOutputs32bit[#0]=0x%x ", __func__, (unsigned
int)ulDigOutputs32bit);
}

/*================================= Example functions END ===========================================*/

/*================================= Output STR ============================================*/
#ifdef PROGRAM_OUTPUT
#endif /* PROGRAM_OUTPUT */
/*================================= Output END ============================================*/
```

17.25.2 CMMCEoE Class Functions Code Example 2

```
/*
 * sample.cpp
 *
 *  Created on: 17/06/2013
 * Author: Elmo
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //system signal mechanism
#include <c++/4.1.0/iostream>
#include <MMCEoE.h>
#include <mmcpplib.h>
using namespace std;

static CMMCEoE g_eoeObject;
/*
 * Call this function from within the main function every 5 milliseconds (for instance)
 */
int TestEoE()
{
     static int iState = 0;
     static float fCount = 17.0;
     static int iCount = 1000;
     int rc, iValue;
     float fValue;
     bool bWait;
     char szVersion[128];

     switch (iState) {
          case 0: //connect
                      rc = g_eoeObject.Connect("192.168.1.6", 5001, bWait); //create a //none
                      blocking connection.
               iState = 10;
               if (bWait)
                    usleep(10000); //wait 10 ms
               break;
          case 1:
               rc = g_eoeObject.ElmoSetParameter("px", iCount++);
               iState = 2;
               break;
          case 2:
               rc = g_eoeObject.ElmoGetParameter("px", iValue);
               cerr << "px = " << iValue << endl;
               iState = 3;
               break;
          case 3:
               rc = g_eoeObject.ElmoSetArray("ui", 1, iCount++);
               iState = 4;
               break;
          case 4:
               rc = g_eoeObject.ElmoGetArray("ui", 1, iValue);
               cerr << "ui[1] = " << iValue << endl;
               iState = 5;
               break;
          case 5:
               rc = g_eoeObject.ElmoSetArray("uf", 3, fCount++);
               iState = 6;
               break;
          case 6:
               rc = g_eoeObject.ElmoGetArray("uf", 3, fValue);
               cerr << "uf[3] = " << fValue << endl;
               iState = 7;
               break;
          case 7:
               rc = g_eoeObject.ElmoSetAsyncArray("uf", 3, fCount++);
               iState = 8;
               break;
          case 8:
               rc = g_eoeObject.ElmoGetAsyncArray("uf", 3);
               iState = 9;
               break;
          case 9:
```

```
            //referes to last ElmoGetAsyncArray("uf", 3) on case 8.
            rc = g_eoeObject.ElmoReadData(fValue);
            cerr << "Async uf[3] = " << fValue << endl;
            iState = 2;
            break;
        case 10:
                    rc = g_eoeObject.ElmoGetParameter("vr", szVersion, sizeof(szVersion));
            cout << "vr = " << szVersion << endl;
            iState = 2;
            break;
        default:
            g_eoeObject.SendTo("bg\r", 3);
            break;
    }

    return 0;
}
```

17.25.3 ElmoSetAsyncArray

Send an EoE array command asynchronously, and sets parameter of type int or float. Refer to the section **16.8.8 MMC_ElmoSetArray** for details of the description, and scope.

```
int ElmoSetAsyncArray(
char cCmd[3],
short iIndex,
[const int iVal]
[const float fVal]
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any character value with a maximum of 2 bytes and '\r' at the end.

iIndex

EoE array index. Array index parameter (only relevant for array situations). Any integer values.

iVal, fVal

Integer value of the data that is to be set. Optionally, float value of the data.

Return

Returns 1 if command was sent OK, or alternative obtained the real value, otherwise 0.

17.25.4 ElmoSetAsyncParameter

Refer to the section **16.8.3 MMC_ElmoSetParameter** for details of the description, and scope.

```
int ElmoSetAsyncParameter(
char cCmd[3],
[const int iVal]
[const float fVal]
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iVal, fVal

Integer value of the data that is to be set. Optionally, float value of the data.

Return

Returns 1 if obtained the integer or alternative real value, otherwise 0.

17.25.5 ElmoSetArray

Send an EoE array command, and sets parameter of type int or float. Refer to the section **16.8.8 MMC_ElmoSetArray** for details of the description, and scope.

```
int ElmoSetArray(
char cCmd[3],
short iIndex,
[const int iVal]
[const float fVal]
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iIndex

EoE array index. Array index parameter (only relevant for array situations). Any +ve integer values.

iVal, fVal

Integer value of the data that is to be set. Optionally, float value of the data.

Return

Return - 0 on success, otherwise 1.

17.25.6 ElmoSetParameter

Set an EoE command, and sets parameter of type int or float. Refer to the section **16.8.3 MMC_ElmoSetParameter** for details of the description, and scope.

int ElmoSetParameter(
char cCmd[3],
[const int iVal]
[const float fVal]
);

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iVal, fVal

Integer value of the data that is to be set. Optionally, float value of the data.

Return

Return - 0 on success, otherwise 1.

17.25.7 ElmoGetAsyncArray

Sends an asynchronous array command to get EoE parameter, with the result that one may get the following result asynchronously; Call IsReady, then ElmoReadData API. Refer to the section **16.8.5 MMC_ElmoGetArray** for details of the description, and scope.

int ElmoGetAsyncArray(
char cCmd[3],
[short iIndex]
int& fVal
);

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iIndex

EoE array index. Array index parameter (only relevant for array situations). Any +ve integer values.

Return

Return - 0 on success, otherwise 1.

17.25.8 ElmoGetAsyncParameter

Sends an asynchronous command to get EoE parameter, with the result that one may get the following result asynchronously; Call IsReady, then ElmoReadData API. Refer to the section **16.8.4 MMC_ElmoGetParameter** for details of the description, and scope.

int ElmoGetAsyncArray(
char cCmd[3],
);

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

Return

Return - 0 on success, otherwise 1.

17.25.9 ElmoGetArray

Sends an array command to get EoE parameter of type int or float, with the result that one may get the following result; Call IsReady, then ElmoReadData API. Refer to the section **16.8.5 MMC_ElmoGetArray** for details of the description, and scope.

```
int ElmoGetArray(char cCmd[3], short iIndex);
char cCmd[3],
short iIndex
[int& iVal]
[float& fiVal]
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

cCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

iIndex

EoE array index. Array index parameter (only relevant for array situations). Any +ve integer values.

Int& iVal, float& fVal

Integer value of the data that is to be set. Optionally, float value of the data.

Return

Return - 0 on success, otherwise 1.

For an example, refer to the section 17.25.1.

17.25.10 ElmoGetParameter

Obtain EoE parameter of type int, float, or string. Refer to the section **16.8.4 MMC_ElmoGetParameter** for details of the description, and scope.

```
int ElmoGetParameter(
char szCmd[3],
[int& iVal,]
[float& fiVal,]
[char szVal[],]
[int iSize]
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

szCmd[3]

Name of the parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

Int& iVal, float& fVal, or szVal[]

Integer value of the data that is to be set. Optionally, float or string value of the data.

iSize

Storage buffer for the returned value.

Return

Return - 0 on success, otherwise 1.

For an example, refer to the section 17.25.1.

17.25.11 ElmoReadData

Read EoE data and set iValue accordingly parameter of type int, float, or string. Refer to the section **16.8.5 MMC_ElmoGetArray** for details of the description, and scope.

```
int ElmoReadData(
[char* cBuffer,]
[int iSize]
[int& iValue,]
[float& fiVal]
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

cBuffer

The buffer size. Any +ve character value with a maximum of 2 bytes

Int& iVal, float& fVal

Integer value of the data that is to be set. Optionally, float value of the data.

iSize

Storage buffer for the returned value.

Return

Return - 0 on success, otherwise 1.

17.25.12 ElmoCall

Sends an EoE command for execution. Refer to the section **16.8.12 MMC_ElmoCall** for details of the description, and scope.

```
int ElmoCall(
const char szCmd[3]
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

szCmd[3]

Name of the constant parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

Return

Return - 0 on success, otherwise 1.

17.25.13 ElmoCallAsync

Sends an asynchronous EoE command for execution. Refer to the section **16.8.12 MMC_ElmoCall** for details of the description, and scope.

```
int ElmoCallAsync(
const char szCmd[3]
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

szCmd[3]

Name of the constant parameter limited to three characters. Any +ve character value with a maximum of 2 bytes

Return

Return - 0 on success, otherwise 1.

17.25.14 ElmoAck

Checks whether or not a previous EoE command is acknowleged. Refer to the section **16.8.12 MMC_ElmoCall** for details of the description, and scope.

```
int ElmoAck(
char* szBuffer,
int iSize
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

szBuffer

The buffer size. Any +ve character value with a maximum of 2 bytes

iSize

Storage buffer for the returned value.

Return

Return - 0 on success, otherwise 1.

17.25.15 ElmoExecuteLabelAsync

Sends an asynchronous EoE command for user program execution. Refer to the section **16.8.2 MMC_ElmoExecuteLabel** for details of the description, and scope.

```
int ElmoExecuteLabelAsync(
const char *szCmd
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

szCmd

Name of the constant parameter with unlimited characters. Any +ve character value.

Return

Return - 0 on success, otherwise 1.

17.25.16 ElmoExecuteLabel

Sends an asynchronous EoE command for user program execution. Refer to the section **16.8.2 MMC_ElmoExecuteLabel** for details of the description, and scope.

```
int ElmoExecuteLabel(
const char *szCmd
);
```

Source GMAS\includes\CPP\ MMCEoE.h

.NET Definition

Function Parameters

szCmd

Name of the constant parameter with unlimited characters. Any +ve character value.

Return

Return - 0 on success, otherwise 1.

Chapter 18: IEC 61131-3 Special Functions

There are two special functions required by IEC in order to confirm the Library and Run-Time versions during initiation. These are:

ElmoIECLibVers	IEC Library version
ElmoIECRTVers	IEC Run time version

18.1.1 ElmoIECLibVers

Reads the Elmo IEC library version during initiation of the IEC 61131 program.

Source IEC61131 Library\ElmoGlobal

Function Parameters

@Version

[INPUT] the present version of the library read during initiation of the IEC program. Has a String value

RC

[OUTPUT] Return code. Short integer value

18.1.2 ElmoIECRTVers

Reads the Elmo IEC run-time version during initiation of the IEC 61131 program.

Source IEC61131 Library\ElmoGlobal

Function Parameters

@Version

[INPUT] the present run-time version read during initiation of the IEC program. Has a String value

RC

[OUTPUT] Return code. Short integer value

18.1.3 Elmo_RetainLoad

The IEC 61131-3 programming allows variables and their values to be Retained, i.e. saved within the Maestro for loading when using a specific function that requests or requires these variables. If not previously loaded, then when function requests them and their values, a popup will appear requesting to run the special function Elmo_RetainLoad.

Not applicable to Maestro API C and C++ applications. Only for IEC applications.

Motion Mode	N/A	N/A
Source	N/A	

Function Parameters

Enable

[INPUT] enabled. Has Boolean value

Result

[OUTPUT] Result code. Has Boolean value

Remarks

None

Scope

N/A

Figure 18-1 describes the function for Elmo_RetainLoad as applied within the IEC 61131 programming.

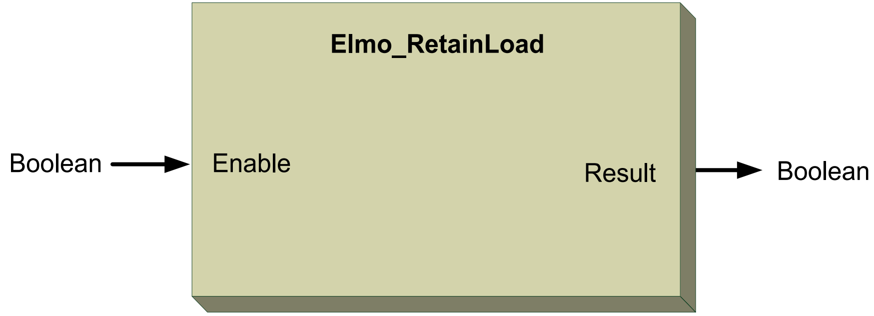


Figure 18-1: Elmo_RetainLoad function

18.1.4 Elmo_RetainSave

The IEC 61131-3 programming allows variables and their values to be Retained, i.e. saved within the Maestro for loading when using a specific function that requests or requires these variables. The function Elmo_RetainSave performs this purpose.

Not applicable to Maestro API C and C++ applications. Only for IEC applications.

Motion Mode N/A N/A

Source N/A

Function Parameters

Enable

[INPUT] enabled. Has Boolean value

Result

[OUTPUT] Result code. Has Boolean value

Remarks

None

Scope

N/A

Figure 18-2 describes the function for Elmo_RetainSave as applied within the IEC 61131 programming.

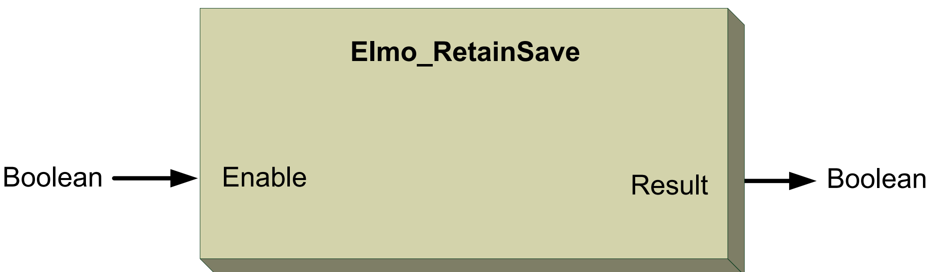


Figure 18-2: Elmo_RetainSave function

18.1.5 MMC_SetImmediateExec

Function blocks are collected in the Maestro queue with their execution bit turned OFF. As each function block is executed, it is immediately turned ON. The IEC program may set the Intermediate flag to insert several function blocks to the queue (invocation of these FBs), and then set the Intermediate flag to ON, for the Maestro to execute them all.

Not applicable to Maestro API C and C++ applications. Only for IEC applications.

Motion Mode N/A N/A

Source GMAS Programming(IEC 61331 Program)\ElmoGenAxis

Function Parameters

N/A

Remarks

None

Scope

N/A

Figure 18-3 describes the function for MMC_SetImmediateExec as applied within the IEC 61131 programming.

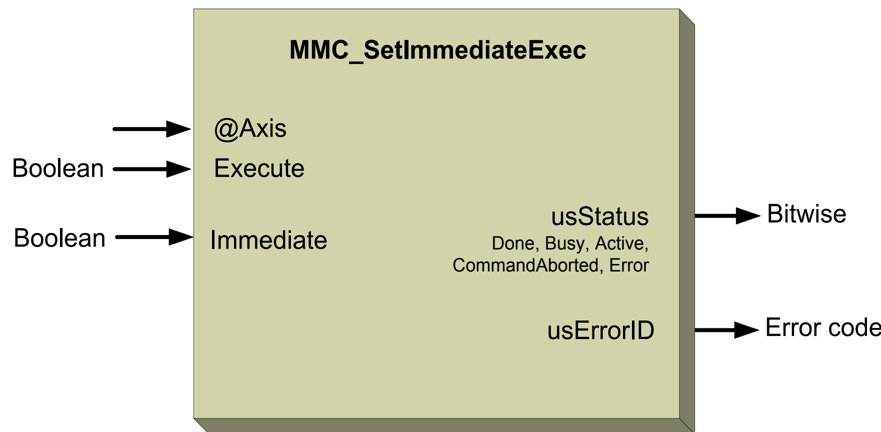


Figure 18-3: MMC_SetImmediateExec function

Chapter 19: Appendix

19.1 Axis Parameters (Explanations)

The axis parameters define the MMC_PARAMETER_LIST_ENUM eParameterNumber values of the axis status. Refer to the section **5.3.2 Parameters Tables** for the integer parameter definitions.

Table 2: Boolean Global Parameters table

Index	Parameter description	Explanation
29	CYCLE_TIME	GMAS Cycle Time [usec]
30	RES_ID	Resource File Identifier
49	CONNECTION_TYPE	GMAS Communication Type: 1 - Ethercat, 2 - CAN
53	SUPPORT_BLENDED_FB_PENDING	Support blended FB insertion during transition between execute bit 1 to 0 (pending).
54	FB_CNT_BEFORE_ACTIVE_FB_ON_END_VELOCITIES_RECALCULATION	Number of FB's before active FB to calculate, when performing EndVelocities Recalculation
55	ESTIMATED_TIME_TOBE_ACTIVE_FB_ON_END_VELOCITIES_RECALCULATION	Execution time limit before FB became active, when performing EndVelocities Recalculation.[ms]
80	IS_FATAL_ERROR	Indicates whether GMAS is in fatal error
81	LAST_SYSTEM_ERROR	The last system error number
93	ErCr_AUTOLOAD	Bitmask variable that defines which error correction tables will be autoloaded
111	GMAS_INIT_STATE	Indicates GMAS initialization state. 0 - init not finished, 1 - init finished, 2 - initializing

Table 3: Boolean Axis Parameters table

Index	Parameter description	Explanation
1	AXIS_MODE	Profile mode of the Axis: 0 - NC. 1 - Distributed. 2 - Virtual.
2	OP_MODE	Operation mode of the Axis: (-1) - OPM402_NO (1) - OPM402_PROFILE_POSITION_MODE (2) - OPM402_VELOCITY_MODE (3) - OPM402_PROFILE_VELOCITY_MODE (4) - OPM402_TORQUE_PROFILE_MODE (6) - OPM402_HOMING_MODE (7) - OPM402_INTERPOLATED_POSITION_MODE (8) - OPM402_CYCLIC_SYNC_POSITION_MODE (9) - OPM402_CYCLIC_SYNC_VELOCITY_MODE (10)- OPM402_CYCLIC_SYNC_TORQUE_MODE

Index	Parameter description	Explanation
3	AXIS_STATE	Refer to section 5.4 for details of these parameters.
5	DRIVE_ID	Drive ID
11	ACT_POS	Actual Position (0x6064)
12	AIM_POS	Last Commanded Position (0x607A)
14	ACT_VEL	Actual velocity of drive (like NC_REC_ACTUAL_POS)
17	IN_CONST_VEL	Indication whether the drive in constant velocity state or not. 1 - In constant velocity. 0-Not in constant velocity.
21	IN_AC	Indication whether the drive in accelerating state or not. 1-In accelerating. 0-Not in accelerating.
25	IN_DC	Indication whether the drive in deceleration state or not. 1-In deceleration 0-Not in deceleration
33	COMM_I_EV_USR_1	Value of UI[] vector within specific index as defined in PDO3\4 Maestro - DRIVE UI Parameter #1 - Main (0x2F00)
34	COMM_I_EV_USR_1_AUX	Value of UI[] vector within specific index of ["USR_1_PARAM" + 1] Maestro - DRIVE UI Parameter #1 - Aux (0x2F00)
37	COMM_I_EV_USR_2	Value of UI[] vector within specific index as defined in PDO3\4 Maestro - DRIVE UI Parameter #2 - Main (0x2F00)
38	COMM_I_EV_USR_2_AUX	Value of UI[] vector within specific index of ["USR_2_PARAM" + 1] Maestro - DRIVE UI Parameter #2 - Aux (0x2F00)
41	COMM_I_EV_USR_3	Value of UI[] vector within specific index as defined in PDO3\4 Maestro - DRIVE UI Parameter #3 - Main (0x2F00)
42	COMM_I_EV_USR_3_AUX	Value of UI[] vector within specific index of ["USR_3_PARAM" + 1] Maestro - DRIVE UI Parameter #1 - Aux (0x2F00)
45	COMM_I_EV_USR_4	Value of UI[] vector within specific index as defined in PDO3\4 Maestro - DRIVE UI Parameter #4 - Main (0x2F00)

Index	Parameter description	Explanation
46	COMM_I_EV_USR_4_AUX	Value of UI[] vector within specific index of ["USR_4_PARAM" + 1] Maestro - DRIVE UI Parameter #4 - Aux (0x2F00)
52	MOTOR_ON_CMD_MAX_TIMEOUT_MS	Motor On Timeout [ms]. Note: Loaded at GMAS Powerup.
56	MAX_CURRENT_PARAM	Maximum Allowed Axis Current. (0x6073)
69	DRIVE_TRACKING_ERROR	Tracking error, read from drive (0x60F4)
71	END_MOTION_REASON	Represent the reason of last motion ending for a specific axis or group
72	AXIS_ERROR_MASK	Mask of the axis error
73	LAST_DRIVE_EMERGENCY	Last emergency code that received from the drive
74	AXIS_ASYNC_ERROR_CODE	Asynchronous axis error code
75	FB_DEPTH	Current number of function block in the axis\group FB queue
76	ANALOG_INPUT	Actual analogue input in internal units. (0x2205)
77	EMCY_SET_ERR	Determine whether to enter DS401 device to error state when emergency occurs
78	ETHERCAT_DRV_OUTPUT	Digital output of EtherCAT drive. (0x60FE)
79	DIGITAL_INPUT_LOGIC	The logic of the digital inputs: DI' = (DI) XOR (DIGITAL_INPUT_LOGIC)
82	LAST_NODE_INIT_ERROR	The last node error number
83	LAST_SDO_ABORT_RAW_DATA	The raw data that was returned from node after a SDO aborted
85	FAST_REFERENCE	Drive Fast reference two MSB indicate axis ref, two LSB indicate what variable to use
87	DIGITAL_INPUT_PARAM	Digital input parameter (0x60FD)
88	CAN_DRV_OUTPUT	Digital output of CAN drive. (0x60FE)
89	DS402_CONTROL_WORD	DS402 control word. (0x6040)
90	DS402_STATUS_WORD	DS402 status word. (0x6041)
91	STATUS_REGISTER	Status register of the axis\group
102	MODULO_AXIS	Rotation axis flag, 1=rotation, 0=linear
105	MODULO_ACTUAL_CYCLE	Number of modulo cycles on actual position
106	MODULO_TARGET_CYCLE	Number of modulo cycles on target position
107	AUXILIARY_POSITION	auxiliary position (0x20A0)
108	UU_ENUM_POS	Type of Position User Units

Index	Parameter description	Explanation
109	UU_ENUM_VEL	Type of Velocity User Units
110	FAST_REFERENCE_MODE	Fast Reference Mode. If = 0 - Position is sent (Actual/Target/S - Please see FAST_REFERENCE variable). If = 1, FAST_REFERENCE is sent as-is

Table 4: Boolean Group Parameters table

Index	Parameter description	Explanation
4	GROUP_ID	Group ID
28	SPATIAL_OPTION	ACS Spacial Option: 0 - SQRT Length (default), 1 - MAX Length
92	MCS_LIMIT_REGISTER	Status of the MCS limits in group

Table 5: Real Axis Parameters table

Index	Parameter description	Explanation
10	SET_POS	Last Set Point
13	SET_VEL	Last Set Velocity
31	SW_LIM_HIGH	Software High Position Limit [units]
32	SW_LIM_LOW	Software Low Position Limit [units]
35	COMM_F_EV_USR_1	Value of UF[] vector within specific index as defined in PDO3\4 Maestro - DRIVE UF Parameter #1 - Main (0x2F01)
36	COMM_F_EV_USR_1_AUX	Value of UF[] vector within specific index of ["USR_1_PARAM" + 1] Maestro - DRIVE UF Parameter #2 - Aux (0x2F01)
39	COMM_F_EV_USR_2	Value of UF[] vector within specific index as defined in PDO3\4 Maestro - DRIVE UF Parameter #2 - Main (0x2F01)
40	COMM_F_EV_USR_2_AUX	Value of UF[] vector within specific index of ["USR_2_PARAM" + 1] Maestro - DRIVE UF Parameter #2 - Aux (0x2F01)
43	COMM_F_EV_USR_3	Value of UF[] vector within specific index as defined in PDO3\4 Maestro - DRIVE UF Parameter #3 - Main (0x2F01)
44	COMM_F_EV_USR_3_AUX	Value of UF[] vector within specific index of ["USR_3_PARAM" + 1] Maestro - DRIVE UF Parameter #3 - Aux (0x2F01)
47	COMM_F_EV_USR_4	Value of UF[] vector within specific index as defined in PDO3\4

Index	Parameter description	Explanation
		Maestro - DRIVE UF Parameter #4 - Main (0x2F01)
48	COMM_F_EV_USR_4_AUX	Value of UF[] vector within specific index of ["USR_4_PARAM" + 1] Maestro - DRIVE UF Parameter #4 - Aux (0x2F01)
50	ERROR_CORRECTION_PARAM	Error Correction for current Axis Position
63	TARGET_RADIUS	Target Radius where the axis get 'Target Reached' state |actual position - target position|
64	TARGET_TIME	Minimal time inside the target radius before 'Target Reached' bit is set [msec]
67	MAX_TRACKING_ERROR_POSITION	Maximum allowed tracking error
68	MAX_TRACKING_ERROR_TIME	Maximum allowed time inside tracking error window
70	GMAS_TRACKING_ERROR	Tracking error, created by Maestro
86	SET_ACDC_PARAM	Last Used AC\DC Command [units/sec^2]
94	MAX_DESIRED_TORQUE_PARAM	Maximum desired torque value in [mA]
95	MAX_TORQUE_VELOCITY_PARAM	Maximum torque velocity value [mA/sec]
96	MAX_TORQUE_ACCELERATION_PARAM	torque acceleration value [mA/sec^2]
97	USER_UNITS_POSITION	Position User Units
98	USER_UNITS_KINEMATICS	Velocity/AC/DC/Jerk User Units
99	POSITION_OFFSET	Position offset between encoder and user units [UU]
100	TARGET_POS_UU	Target position in user units [UU]
101	ACTUAL_POS_UU	Actual position in user units [UU]
103	MODULO_LOW	Modulo, low bound in user units [UU]
104	MODULO_HIGH	Modulo, high bound in user units [UU]

Table 6: Real Group Parameters table

Index	Parameter description	Explanation
15	MAX_VEL	Maximum Allowed Velocity
18	SET_AC	Last Used Acceleration Command [units/sec^2]
19	MAX_AC	Maximum Allowed Acceleration [units/sec^2]
22	SET_DC	Last Used Deceleration Command
23	MAX_DC	Maximum Allowed Deceleration [units/sec^2]
26	MAX_JERK	Maximum Allowed Jerk [units/sec^3]
51	S_Factor_For_Polynomial_Transition	The length of the transition curve on polynomial transition modes. The units are - rational to ideal arc.
57	LIMIT_STOP_DECELERATION	On Limit Axis/Group Stop Deceleration
58	LIMIT_STOP_JERK	On Limit Axis/Group Stop Jerk

Index	Parameter description	Explanation
59	SET_VECTOR_VEL	Last Vector Velocity
60	MCS_SW_LIMIT_LOW_POS_ARRAY	MCS Software Low Position Limit Array [units]
61	MCS_SW_LIMIT_HIGH_POS_ARRAY	MCS Software High Position Limit Array [units]
62	MCS_S_DIRECTION	Path output [units] (S kinematic direction)
65	PROFILE_TIME	Profile time of last motion
66	OVERALL_MOTION_TIME	Overall motion time of last motion that settled inside the target radius and time
84	SPEED_OVERRIDE	Velocity factor that define the actual maximum velocity of the function block

Inspiring Motion
Since 1988
For a list of Elmo's branches, and your local area office, refer to the Elmo site www.elmomc.com
Elmo
Motion Control