



# HEIDENHAIN



Operating Instructions

## **MSElibrary Software** v2.3.x

English (en)  
8/2016



## Firmware version

This document describes MSEfirmware.dat v2.3.0 (ID 1090899-05) and MSElibrary v2.3.0 (ID 781598-07) and older.

## Fonts used in these instructions

Items of special interest or concepts that are emphasized to the user are shown in **bold** type.

## Modules

Module	Type	Description	IP	ID	
<b>Required<sup>1)</sup></b>	<b>MSE 1114</b>	Base	4 axes, EnDat22	IP65	747499-01
	<b>MSE 1114</b>	Base	4 axes, EnDat22	IP40	747499-02 <sup>2)</sup>
	<b>MSE 1124</b>	Base	4 axes, TTL	IP65	747511-01
	<b>MSE 1124</b>	Base	4 axes, TTL	IP40	747511-02 <sup>2)</sup>
	<b>MSE 1184</b>	Base	4 axes, 1 Vpp	IP65	747500-01
	<b>MSE 1184</b>	Base	4 axes, 1 Vpp	IP40	747500-02 <sup>2)</sup>
	<b>MSE 1201</b>	Power supply	AC 100 V ... 240 V, power plug	IP40	747501-01
	<b>MSE 1201</b>	Power supply	AC 100 V ... 240 V, Cable 2 m with cable gland	IP65	747501-02
	<b>MSE 1202</b>	Power supply	DC 24 V M8, 3-pin female	IP65	747502-01
	<b>MSE 1202</b>	Power supply	DC 24 V M8, 3-pin female	IP40	747502-02 <sup>2)</sup>
<b>Optional</b>	<b>MSE 1314</b>	Axis	4 axes, EnDat22	IP65	747503-01
	<b>MSE 1314</b>	Axis	4 axes, EnDat22	IP40	747503-02 <sup>2)</sup>
	<b>MSE 1318</b>	Axis	8 axes, EnDat22	IP65	747504-01
	<b>MSE 1318</b>	Axis	8 axes, EnDat22	IP40	747504-02 <sup>2)</sup>
	<b>MSE 1324</b>	Axis	4 axes, TTL	IP65	747512-01
	<b>MSE 1324</b>	Axis	4 axes, TTL	IP40	747512-02 <sup>2)</sup>
	<b>MSE 1328</b>	Axis	8 axes, TTL	IP65	747513-01
	<b>MSE 1328</b>	Axis	8 axes, TTL	IP40	747513-02 <sup>2)</sup>
	<b>MSE 1332</b>	Analog	Analog input	IP65	747509-01
	<b>MSE 1332</b>	Analog	Analog input	IP40	747509-02 <sup>2)</sup>
	<b>MSE 1358</b>	Axis	8 axes, Solartron and Tesa half-bridge transducers	IP65	747514-01
	<b>MSE 1358</b>	Axis	8 axes, Solartron and Tesa half-bridge transducers	IP40	747514-04 <sup>2)</sup>
	<b>MSE 1358</b>	Axis	8 axes, Mahr half-bridge and LVDT transducers	IP65	747514-02
	<b>MSE 1358</b>	Axis	8 axes, Mahr half-bridge and LVDT transducers	IP40	747514-05 <sup>2)</sup>
	<b>MSE 1358</b>	Axis	8 axes, Marposs LVDT transducers	IP65	747514-03
	<b>MSE 1358</b>	Axis	8 axes, Marposs LVDT transducers	IP40	747514-06 <sup>2)</sup>
	<b>MSE 1384</b>	Axis	4 axes, 1 Vpp	IP65	747505-01
	<b>MSE 1384</b>	Axis	4 axes, 1 Vpp	IP40	747505-02 <sup>2)</sup>
	<b>MSE 1388</b>	Axis	8 axes, 1 Vpp	IP65	747506-01
	<b>MSE 1388</b>	Axis	8 axes, 1 Vpp	IP40	747506-02 <sup>2)</sup>
<b>MSE 1401</b>	I/O	4 inputs/4 outputs	IP40	747507-01	
<b>MSE 1401</b>	I/O	4 inputs/4 outputs, M8 connectors	IP65	747507-02	
<b>MSE 1501</b>	Compressed-air	1 channel	IP65	747508-01	
<b>MSE 1501</b>	Compressed-air	1 channel	IP40	747508-02 <sup>2)</sup>	

<sup>1)</sup>One Base module, one Power supply module required.

<sup>2)</sup>Requires MSElibrary v2.2.0 or newer.



# Table of contents

- 1 Configuring the MSE 1000**
- 2 Library software**
- 3 System integrity**
- 4 Diagnostic modes**
- 5 Trigger line**
- 6 Module Networking and Throughput**
- 7 Module descriptions**
- 8 Operating principles**
- 9 C++ examples**
- 10 C examples**
- 11 Visual Basic examples**
- 12 Delphi examples**
- 13 LabVIEW**



## 1 Configuring the MSE 1000 ..... 19

- 1.1 Ethernet cable ..... 20
- 1.2 Initial IP address ..... 20
- 1.3 Changing a modules IP address ..... 20
  - C++ example ..... 21
- 1.4 Changing DHCP ..... 22

## 2 Library software ..... 23

- 2.1 General information ..... 24
- 2.2 Installation instructions ..... 24
- 2.3 Overview ..... 25
  - Modules ..... 25
  - Definitions ..... 25
  - Prerequisites ..... 25
  - Methods and functions ..... 25
  - Classes ..... 26
  - Wrappers ..... 27
  - Module Communication ..... 27
- 2.4 Data Types ..... 28
- 2.5 Enumerations ..... 29
  - MSE\_CHAIN\_CREATION\_STATE ..... 29
  - UOM ..... 29
  - MODULE\_ID ..... 30
  - ENDAT\_ERROR\_RESULT ..... 32
  - ENDAT\_ERRORS ..... 32
  - NUM\_ENDAT\_ERRORS ..... 32
  - ENDAT\_WARNINGS ..... 33
  - NUM\_ENDAT\_WARNINGS ..... 33
  - ENDAT\_DIAG ..... 33
  - MSE\_RESPONSE\_CODE ..... 34
  - LATCH\_OPTIONS ..... 36
  - LATCH\_CHOICE ..... 36
  - ROTARY\_FORMAT ..... 36
  - ADC\_OPTIONS ..... 37
  - COUNT\_REQUEST\_OPTION ..... 37
  - ENCODER\_TYPES\_ENUM ..... 38
  - VPP\_VOLTAGE\_FEEDBACK ..... 38
  - INTEGRITY\_ENUMS ..... 39
  - REFERENCE\_MARK\_ENUM ..... 40
  - REFERENCE\_MARK\_STATE ..... 40
  - COUNTER\_STATUS ..... 41
  - MSE\_XML\_RETURN ..... 42
  - MSE\_XML\_ELEMENTS ..... 43
  - PROGRAMMING\_STATE\_ENUMS ..... 46
  - UdpCmdType ..... 47
  - LVDT\_UOM ..... 49
  - LVDT\_UPDATE\_CHOICES ..... 49
  - LVDT\_OVERSAMPLING\_CHOICES ..... 49
  - ANALOG\_DIAG\_VOLTAGES\_ENUM ..... 50
  - TTL\_INTERPOLATION ..... 50
  - SIGNAL\_TYPE ..... 51
- 2.6 Classes and structures ..... 52
  - ModuleData ..... 52
  - DeviceData ..... 53
  - LeftData ..... 54
  - MSE1000ConnectResponse ..... 54
  - EncoderInfo ..... 55
- 2.7 Return values ..... 56
  - getCode ..... 56
  - getMethod ..... 56
  - getLine ..... 56
  - showRespCode ..... 56

Example: .....	56
2.8 Constants .....	57
NUM_MSE1000_IO_INPUTS .....	57
NUM_MSE1000_IO_OUTPUTS .....	57
DEVICE_NAME_SIZE .....	57
DEVICE_ID_SIZE .....	57
SERIAL_NUMBER_SIZE .....	57
SIZE_IP_ADDRESS .....	58
SIZE_MAC_ADDRESS .....	58
SIZE_BUILD_INFO .....	58
SIZE_SERIAL_NUMBER .....	58
MAX_NUM_MODULES .....	58
MAX_CHANNELS_PER_MODULE .....	58
MSE1000_PORT .....	59
MSE1000_CLIENT_DEFAULT_PORT .....	59
MSE1000_ASYNC_PORT .....	59
NUM_INTEGRITY_RANGES .....	59
NUM_LATCH_TYPES .....	59
COUNTS_PER_LINE .....	59
INTERPOLATION_VALUE .....	59
NUM_LVDT_CHANNELS .....	59
LVDT_EXCITATION_VOLTAGE_MIN_VPP .....	60
LVDT_EXCITATION_VOLTAGE_MAX_VPP .....	60
LVDT_EXCITATION_FREQUENCY_MIN_KHZ .....	60
LVDT_EXCITATION_FREQUENCY_MAX_KHZ .....	60
NUM_MSE1000_ANALOG_CHANNELS .....	60
NUM_MSE1000_ANALOG_VALUES_PER_CHANNEL .....	60
MAX_NUM_ANALOG_AVG_SAMPLES .....	60
2.9 Interface methods .....	61
MseInterface .....	61
addModule .....	61
removeConnections .....	61
createChain .....	62
getChainCreationState .....	62
getNumModules .....	62
getModule .....	62
getDeviceModule .....	63
getEndatModule .....	63
getIoModule .....	63
get1VppModule .....	63
getPneumaticModule .....	64
getAnalogModule .....	64
getLvdtModule .....	64
getTtlModule .....	64
2.10 General methods and functions .....	65
C++ methods .....	65
MseModule .....	65
initializeModule .....	65
initializeFirmware .....	65
getModuleType .....	66
getConfig .....	66
getNumChannels .....	67
getModuleData .....	67
getLeft .....	67
getCounts .....	68
setRotaryFormat .....	68
getRotaryFormat .....	68
setDeviceOffset .....	69
getDeviceOffset .....	69
setRight .....	69
resetMse1000 .....	69
program .....	70



getProgrammingState ..... 70  
 getProgrammingPercentComplete ..... 70  
 showModuleType ..... 70  
 showModuleId ..... 71  
 setIp ..... 71  
 setAsyncPort ..... 71  
 getAsyncPort ..... 72  
 setDhcp ..... 72  
 broadcastOpenConnection ..... 72  
 setBroadcastingNetmask ..... 73  
 restoreFactoryDefaults ..... 73  
 setUdpTimeout ..... 73  
 getUdpTimeout ..... 73  
 setUdpNumRetries ..... 74  
 getUdpNumRetries ..... 74  
 setNetworkDelay ..... 74  
 getNetworkDelay ..... 74  
 setLatch ..... 75  
 getLatch ..... 75  
 getAdcValues ..... 75  
 getIntegrity ..... 76  
 setAsyncMode ..... 76  
 clearAllErrors ..... 77  
 clearIntegrityErrors ..... 77  
 enableDiags ..... 77  
 getLibraryVersion ..... 77  
 C Functions ..... 78  
 MseModuleCreate ..... 78  
 MseModuleDelete ..... 78  
 MseModuleInitialize ..... 78  
 MseModuleGetLibraryVersion ..... 79  
 MseModuleGetModuleType ..... 79  
 MseModuleGetModuleErrorState ..... 79  
 MseModuleGetModuleErrors ..... 80  
 MseModuleGetAdcValues ..... 80  
 MseModuleClearErrors ..... 81  
 MseModuleSetIpAddress ..... 81  
 MseModuleGetIpAddress ..... 81  
 MseModuleGetIpStaticAddress ..... 82  
 MseModuleGetNetmask ..... 82  
 MseModuleGetNetmaskStatic ..... 82  
 MseModuleGetPort ..... 83  
 MseModuleSetAsyncPort ..... 83  
 MseModuleGetAsyncPort ..... 83  
 MseModuleSetUsingDhcp ..... 84  
 MseModuleGetUsingDhcp ..... 84  
 MseModuleGetMacAddress ..... 84  
 MseModuleGetBootloaderVersion ..... 85  
 MseModuleGetFirmwareVersion ..... 85  
 MseModuleGetSerialNumber ..... 85  
 MseModuleReset ..... 86  
 MseModuleShowType ..... 86  
 MseModuleShowId ..... 86  
 MseModuleSetBroadcastingNetmask ..... 87  
 MseModuleSetUdpTimeout ..... 87  
 MseModuleGetUdpTimeout ..... 87  
 MseModuleSetUdpNumRetries ..... 88  
 MseModuleGetUdpNumRetries ..... 88  
 MseModuleSetNetworkDelay ..... 88  
 MseModuleGetNetworkDelay ..... 89  
 MseModuleBroadcast ..... 89  
 MseModuleProgram ..... 90

MseModuleGetProgramState .....	90
MseModuleGetProgramPercentComplete .....	90
MseModuleGetAsyncMsgType .....	91
MseModuleGetAsyncMsgIpAddress .....	91
MseModuleGetAsyncMsgPort .....	91
MseModuleGetAsyncMsgDhcp .....	92
MseModuleGetAsyncMsgMacAddress .....	92
MseModuleGetAsyncMsgNetmask .....	92
MseModuleGetAsyncMsgSerialNumber .....	93
MseModuleGetAsyncMsgChannelStatus .....	93
MseModuleGetAsyncMsgLatch .....	94
MseModuleShowRespCode .....	94
2.11 Device methods .....	95
MseDeviceModule .....	95
getEncoderInfo .....	95
setEncoderInfo .....	95
getCountingDirection .....	96
setErrorCompensation .....	96
getErrorCompensation .....	96
setScaling .....	97
getScaling .....	97
getResolution .....	98
getEncoderType .....	98
getUom .....	98
enableErrorChecking .....	99
getChannelStatus .....	99
clearErrorsAndWarnings .....	99
setLatchDebouncing .....	100
2.12 EnDat methods and functions .....	101
C++ methods .....	101
initializeModule .....	101
getPositions .....	101
getCounts .....	102
getWarnings .....	102
getErrors .....	103
getDiag .....	103
getDeviceData .....	104
getDistinguishableRevolutions .....	104
getEncoderName .....	105
getEncoderId .....	105
getSerialNumber .....	105
setUom .....	106
getChannelPresence .....	106
setEncoderInfo .....	106
C Functions .....	107
MseEndatModuleCreate .....	107
MseEndatModuleDelete .....	107
MseEndatModuleInitialize .....	107
MseEndatModuleGetNumChannels .....	108
MseEndatModuleGetChannelPresence .....	108
MseEndatModuleGetEncoderType .....	108
MseEndatModuleSetUom .....	109
MseEndatModuleGetUom .....	109
MseEndatModuleSetErrorCompensation .....	109
MseEndatModuleGetErrorCompensation .....	110
MseEndatModuleSetScaling .....	110
MseEndatModuleGetScaling .....	110
MseEndatModuleGetCountingDirection .....	111
MseEndatModuleGetDistinguishableRevolutions .....	111
MseEndatModuleGetResolution .....	112
MseEndatModuleGetCounts .....	112
MseEndatModuleGetPositions .....	113

MseEndatModuleSetRotaryFormat	.....	113
MseEndatModuleGetRotaryFormat	.....	114
MseEndatModuleSetDeviceOffset	.....	114
MseEndatModuleGetDeviceOffset	.....	114
MseEndatModuleSetLatch	.....	115
MseEndatModuleGetLatches	.....	115
MseEndatModuleGetModuleErrorState	.....	116
MseEndatModuleGetModuleErrors	.....	116
MseEndatModuleGetChannelErrorState	.....	116
MseEndatModuleGetEndatErrors	.....	117
MseEndatModuleGetChannelWarningState	.....	117
MseEndatModuleGetEndatWarnings	.....	118
MseEndatModuleClearErrors	.....	118
MseEndatModuleGetEncoderName	.....	118
MseEndatModuleGetEncoderId	.....	119
MseEndatModuleGetEncoderSerialNumber	.....	119
MseEndatModuleSetLatchDebouncing	.....	119
MseEndatModuleEnableDiags	.....	120
MseEndatModuleGetDiags	.....	120
MseEndatModuleGetAdcValues	.....	121
MseEndatModuleEnableErrorChecking	.....	121
2.13 1Vpp methods and functions	.....	122
C++ methods	.....	122
initializeModule	.....	122
getDiag	.....	122
enableAnalogDiag	.....	123
getPositions	.....	123
setUom	.....	124
setEncoderType	.....	124
setLineCount	.....	124
getLineCount	.....	125
getSignalPeriod	.....	125
setSignalPeriod	.....	125
setCountingDirection	.....	126
initAbsolutePosition	.....	126
isReferencingComplete	.....	126
acknowledgeAbsolutePosition	.....	127
getReferencingState	.....	127
getSignalType	.....	127
setSignalType	.....	128
detectSignalType	.....	128
C Functions	.....	128
Mse1VppModuleCreate	.....	128
Mse1VppModuleDelete	.....	128
Mse1VppModuleInitialize	.....	129
Mse1VppModuleGetNumChannels	.....	129
Mse1VppModuleSetEncoderType	.....	129
Mse1VppModuleGetEncoderType	.....	130
Mse1VppModuleSetUom	.....	130
Mse1VppModuleGetUom	.....	130
Mse1VppModuleSetErrorCompensation	.....	131
Mse1VppModuleGetErrorCompensation	.....	131
Mse1VppModuleSetScaling	.....	131
Mse1VppModuleGetScaling	.....	132
Mse1VppModuleSetCountingDirection	.....	132
Mse1VppModuleGetCountingDirection	.....	133
Mse1VppModuleGetResolution	.....	133
Mse1VppModuleGetCounts	.....	133
Mse1VppModuleGetPositions	.....	134
Mse1VppModuleSetRotaryFormat	.....	134
Mse1VppModuleGetRotaryFormat	.....	135
Mse1VppModuleSetDeviceOffset	.....	135

Mse1VppModuleGetDeviceOffset	.....	135
Mse1VppModuleSetLatch	.....	136
Mse1VppModuleGetLatches	.....	136
Mse1VppModuleGetModuleErrorState	.....	136
Mse1VppModuleGetChannelErrorState	.....	137
Mse1VppModuleGetChannelStatus	.....	137
Mse1VppModuleClearErrors	.....	138
Mse1VppModuleSetLatchDebouncing	.....	138
Mse1VppModuleEnableDiags	.....	138
Mse1VppModuleGetAdcValues	.....	139
Mse1VppModuleEnableAnalogDiag	.....	139
Mse1VppModuleGetAnalogDiag	.....	139
Mse1VppModuleSetLineCount	.....	140
Mse1VppModuleGetLineCount	.....	140
Mse1VppModuleSetSignalPeriod	.....	140
Mse1VppModuleGetSignalPeriod	.....	141
Mse1VppModuleStartReferencing	.....	141
Mse1VppModuleGetReferencingComplete	.....	141
Mse1VppModuleAcknowledgeAbsolutePosition	.....	142
Mse1VppModuleGetReferencingState	.....	142
Mse1VppModuleEnableErrorChecking	.....	142
Mse1VppGetSignalType	.....	143
Mse1VppSetSignalType	.....	143
Mse1VppDetectSignalType	.....	143
2.14 I/O methods and functions	.....	144
C++ Methods	.....	144
initializeModule	.....	144
setOutputs	.....	144
setOutput	.....	145
getOutputs	.....	145
getInputs	.....	146
getIO	.....	146
C Functions	.....	147
MseIoModuleCreate	.....	147
MseIoModuleDelete	.....	147
MseIoModuleInitialize	.....	147
MseIoModuleGetNumChannels	.....	148
MseIoModuleGetModuleErrorState	.....	148
MseIoModuleGetModuleErrors	.....	148
MseIoModuleGetAdcValues	.....	149
MseIoModuleClearErrors	.....	149
MseIoModuleSetOutputs	.....	149
MseIoModuleSetOutput	.....	150
MseIoModuleGetOutputs	.....	150
MseIoModuleGetInputs	.....	150
MseIoModuleGetIO	.....	151
MseIoModuleGetLatch	.....	151
MseIoModuleClearLatch	.....	151
2.15 Pneumatic methods and functions	.....	152
C++ Methods	.....	152
initializeModule	.....	152
getOutput	.....	152
setOutput	.....	152
C Functions	.....	153
MsePneumaticModuleCreate	.....	153
MsePneumaticModuleDelete	.....	153
MsePneumaticModuleInitialize	.....	153
MsePneumaticModuleGetNumChannels	.....	154
MsePneumaticModuleGetModuleErrorState	.....	154
MsePneumaticModuleGetModuleErrors	.....	155
MsePneumaticModuleGetAdcValues	.....	155
MsePneumaticModuleClearErrors	.....	156

MsePneumaticModuleSetOutput .....	156
MsePneumaticModuleGetOutput .....	156
MsePneumaticModuleGetLatch .....	157
MsePneumaticModuleClearLatch .....	157
2.16 LVDT methods and functions .....	158
C++ Methods .....	158
initializeModule .....	158
getUom .....	158
setUom .....	159
getExcitationValues .....	159
setExcitationVoltage .....	159
setExcitationFrequency .....	160
getVoltage .....	160
getPositions .....	160
setChannelPresence .....	161
getChannelPresence .....	161
getResolution .....	161
setResolution .....	162
setDiagnosticsEnabled .....	162
getFpgaRevision .....	162
setOversampling .....	162
getSensorGain .....	163
setSensorGain .....	163
teachSensorGain .....	163
getTeachSensorGainFinished .....	164
C Functions .....	164
MseLvdModuleCreate .....	164
MseLvdModuleDelete .....	164
MseLvdModuleInitialize .....	165
MseLvdModuleGetNumChannels .....	165
MseLvdModuleSetUom .....	165
MseLvdModuleGetUom .....	166
MseLvdModuleSetResolution .....	166
MseLvdModuleGetResolution .....	166
MseLvdModuleSetScaling .....	167
MseLvdModuleGetScaling .....	167
MseLvdModuleGetLatch .....	167
MseLvdModuleClearLatch .....	168
MseLvdModuleGetModuleErrorState .....	168
MseLvdModuleGetModuleErrors .....	168
MseLvdModuleEnableDiags .....	169
MseLvdModuleGetAdcValues .....	169
MseLvdModuleClearErrors .....	169
MseLvdModuleGetPositions .....	170
MseLvdModuleSetDeviceOffset .....	170
MseLvdModuleGetDeviceOffset .....	171
MseLvdModuleGetExcitationValues .....	171
MseLvdModuleSetExcitationVoltage .....	172
MseLvdModuleSetExcitationFrequency .....	172
MseLvdModuleGetVoltage .....	172
MseLvdModuleSetChannelPresence .....	173
MseLvdModuleGetChannelPresence .....	173
MseLvdSetDiagnosticsEnabled .....	173
MseLvdGetSensorGain .....	174
MseLvdSetSensorGain .....	174
MseLvdTeachSensorGain .....	174
MseLvdGetTeachSensorGainFinished .....	175
MseLvdGetFpgaRevision .....	175
MseLvdModuleSetOversampling .....	175
2.17 Analog methods and functions .....	176
C++ Methods .....	176
initializeModule .....	176

getVoltage .....	176
getCurrent .....	177
getValues .....	177
getScaledValues .....	178
getDiagVoltages .....	178
setNumSamples .....	179
setResolution .....	179
getResolution .....	179
setOffset .....	180
getOffset .....	180
computeResolutionAndOffset .....	180
setScaling .....	181
getScaling .....	181
C Functions .....	182
MseAnalogModuleCreate .....	182
MseAnalogModuleDelete .....	182
MseAnalogModuleInitialize .....	182
MseAnalogModuleGetNumChannels .....	183
MseAnalogModuleGetModuleErrorState .....	183
MseAnalogModuleGetModuleErrors .....	183
MseAnalogModuleGetAdcValues .....	184
MseAnalogModuleClearErrors .....	184
MseAnalogModuleGetLatch .....	184
MseAnalogModuleClearLatch .....	185
MseAnalogModuleGetVoltage .....	185
MseAnalogModuleGetCurrent .....	185
MseAnalogModuleGetValues .....	186
MseAnalogModuleGetScaledValues .....	186
MseAnalogModuleSetDeviceOffset .....	187
MseAnalogModuleGetDeviceOffset .....	187
MseAnalogModuleSetScaling .....	187
MseAnalogModuleGetScaling .....	188
MseAnalogModuleGetDiagVoltages .....	188
MseAnalogModuleSetNumSamples .....	188
MseAnalogModuleSetResolution .....	189
MseAnalogModuleGetResolution .....	189
MseAnalogModuleSetOffset .....	189
MseAnalogModuleGetOffset .....	190
MseAnalogModuleComputeResolutionAndOffset .....	190
2.18 TTL methods and functions .....	191
C++ Methods .....	191
initializeModule .....	191
getPositions .....	191
getCounts .....	192
setEncoderType .....	192
setUom .....	192
setLineCount .....	193
getLineCount .....	193
setSignalPeriod .....	193
getSignalPeriod .....	194
setCountingDirection .....	194
setChannelPresence .....	194
getChannelPresence .....	195
isReferencingComplete .....	195
initReferencing .....	195
acknowledgeReferencing .....	196
getReferencingState .....	196
getFpgaRevision .....	196
C Functions .....	197
MseTtlModuleCreate .....	197
MseTtlModuleDelete .....	197
MseTtlModuleInitialize .....	197

MseTtlModuleGetNumChannels .....	198
MseTtlModuleSetEncoderType .....	198
MseTtlModuleGetEncoderType .....	198
MseTtlModuleSetUom .....	199
MseTtlModuleGetUom .....	199
MseTtlModuleSetErrorCompensation .....	199
MseTtlModuleGetErrorCompensation .....	200
MseTtlModuleSetScaling .....	200
MseTtlModuleGetScaling .....	200
MseTtlModuleSetCountingDirection .....	201
MseTtlModuleGetCountingDirection .....	201
MseTtlModuleSetChannelPresence .....	201
MseTtlModuleGetChannelPresence .....	202
MseTtlModuleGetResolution .....	202
MseTtlModuleSetLineCount .....	202
MseTtlModuleGetLineCount .....	203
MseTtlModuleSetSignalPeriod .....	203
MseTtlModuleGetSignalPeriod .....	203
MseTtlModuleGetCounts .....	204
MseTtlModuleGetPositions .....	204
MseTtlModuleSetRotaryFormat .....	205
MseTtlModuleGetRotaryFormat .....	205
MseTtlModuleSetDeviceOffset .....	205
MseTtlModuleGetDeviceOffset .....	206
MseTtlModuleSetLatch .....	206
MseTtlModuleGetLatches .....	206
MseTtlModuleGetModuleErrorState .....	207
MseTtlModuleGetModuleErrors .....	207
MseTtlModuleEnableDiags .....	207
MseTtlModuleGetAdcValues .....	208
MseTtlModuleClearErrors .....	208
MseTtlModuleIsReferencingComplete .....	208
MseTtlModuleStartReferencing .....	209
MseTtlModuleAcknowledgeReferencing .....	209
MseTtlModuleGetReferencingState .....	209
MseTtlModuleGetFpgaRevision .....	210
MseTtlModuleGetChannelErrorState .....	210
MseTtlModuleEnableErrorChecking .....	210
2.19 Asynchronous methods .....	211
getAsyncMsgType .....	211
decodeConnectMsg .....	211
decodeLatchMsg .....	212
decodeChannelStatusMsg .....	212
2.20 ModuleConfig Base/Reader/Writer .....	213
C++ Methods .....	213
MseConfigBase .....	213
loadXml .....	213
reloadXml .....	213
decodeErrorType .....	213
decodeElementType .....	214
getFilename .....	214
removeSpecificModuleNode .....	214
MseConfigReader .....	214
getElement .....	214
getElement .....	215
getElement .....	215
getAllElements .....	215
validateElements .....	216
getSpecificModule .....	216
getSpecificChannel .....	216
getNumModules .....	217
getNumChannels .....	217

MseConfigWriter .....	217
addElement .....	217
setElement .....	218
writeFile .....	218
writeFile .....	218
C Functions .....	219
MseConfigFileCreate .....	219
MseConfigFileDelete .....	219
MseConfigFileLoadXml .....	219
MseConfigFileReloadXml .....	219
MseConfigFileGetFilename .....	220
MseConfigFileDecodeErrorType .....	220
MseConfigFileDecodeElementType .....	220
MseConfigFileGetElement .....	221
MseConfigFileGetModuleElement .....	221
MseConfigFileGetChannelElement .....	221
MseConfigFileGetAllElements .....	222
MseConfigFileValidateElements .....	222
MseConfigFileGetAllElements .....	222
MseConfigFileGetSpecificModule .....	223
MseConfigFileGetSpecificChannel .....	223
MseConfigFileGetNumModules .....	223
MseConfigFileGetNumChannels .....	224
MseConfigFileSetModuleElement .....	224
MseConfigFileSetChannelElement .....	224
MseConfigFileRemoveModule .....	225
MseConfigFileWriteFile .....	225
MseConfigFileWriteNewFile .....	225

### 3 System integrity ..... 227

3.1 About system Integrity .....	228
3.2 Obtaining IP Address .....	228
3.3 Waiting for Client .....	228
3.4 Duplicate IP .....	228
3.5 Programming Error .....	228
3.6 Ethernet Chip .....	228
3.7 Current .....	229
3.8 24V .....	229
3.9 5V .....	229
3.10 3.3V .....	229
3.11 CPU temperature .....	229
3.12 Non-Volatile Memory Backup Failure .....	229

### 4 Diagnostic modes ..... 231

4.1 About Diagnostic Modes .....	232
4.2 Full .....	232
4.3 Status .....	232
4.4 Minimal .....	232
4.5 None .....	232

### 5 Trigger line ..... 233

5.1 About the Trigger Line .....	234
5.2 Software Latency .....	234
Full Diagnostics .....	234
Status Diagnostics .....	234
Minimal Diagnostics .....	234
5.3 Debouncing Latency .....	235
5.4 Setting a Trigger .....	235
5.5 Determining Which Latches are Set .....	235
5.6 Reading the Latched Data .....	235
5.7 Clearing a Trigger Manually .....	235



## 6 Module Networking and Throughput ..... 237

- 6.1 Overview ..... 238
- 6.2 Effects on Throughput ..... 238
  - Pre-conditions: ..... 238
    - Latching ..... 238
  - Module conditions: ..... 238
  - Post-conditions: ..... 239
    - Operating system context switching ..... 239
    - Post processing of the received data ..... 239
    - Timeout ..... 239
- 6.3 Throughput Test Results ..... 239
  - Setup ..... 239
  - Average Send/Receive Time ..... 240
  - Data Throughput (packets/second) ..... 240
  - Live Data Throughput (packets/second) ..... 240
  - Live Data Throughput (channels/second) ..... 240
  - Oversampling ..... 240
- 6.4 Latching Throughput Test Results ..... 240
  - Setup ..... 240
    - 1 Vpp or EnDat base module ..... 240
    - TTL base module, ..... 240
  - Results ..... 241
    - 1 Vpp and EnDat base modules ..... 241
    - TTL base module ..... 241
- 6.5 Propagation Delay Test Results ..... 241
  - Setup ..... 241
  - Results ..... 241
- 6.6 Example Chain Configurations ..... 241
  - Large Chain ..... 241
  - Small Chain ..... 241
  - Live and Latching with a Base EnDat ..... 241

## 7 Module descriptions ..... 243

- 7.1 Power supply modules ..... 244
- 7.2 EnDat modules ..... 244
- 7.3 1Vpp modules ..... 244
- 7.4 TTL modules ..... 244
- 7.5 LVDT modules ..... 244
- 7.6 Analog module ..... 244
- 7.7 I/O modules ..... 244
- 7.8 Pneumatic module ..... 244

## 8 Operating principles ..... 245

- 8.1 Overview ..... 246
- 8.2 Initialization ..... 246
  - C++ Initialization ..... 246
  - Non-C++ Initialization ..... 246
- 8.3 Configuring the Channels ..... 247
  - 1Vpp ..... 247
  - EnDat ..... 248
  - TTL ..... 248
  - LVDT ..... 249
  - Analog ..... 250
- 8.4 Channel Operations ..... 251
  - 1Vpp ..... 251
  - EnDat ..... 251
  - TTL ..... 251
  - LVDT ..... 252
  - Analog ..... 252
  - I/O ..... 253
  - Pneumatic ..... 253
- 8.5 Latching ..... 254

8.6 Referencing .....	255
8.7 Module Errors and Warnings .....	256
8.8 Channel Errors and Warnings .....	257
EnDat .....	257
1Vpp .....	257
TTL .....	257
8.9 Diagnostics .....	258
1Vpp .....	258
LVDT .....	258
8.10 Asynchronous Communication .....	259

## 9 C++ examples ..... 261

9.1 Overview .....	262
9.2 MseLibraryCppExamples Visual Studio Solution .....	262
Broadcasting Example .....	262
EnDat Example .....	263
TTL Example .....	263
1 Vpp Example .....	263
LVDT Example .....	263
Analog Example .....	264
I/O Example .....	264
Pneumatic Example .....	264
Latching Example .....	264
Referencing Example .....	264
Referencing a 1Vpp Coded Rotary Encoder Example .....	265
Referencing a TTL Coded Encoder Example .....	265
Module Discovery Example .....	265
Module Subscribe Example .....	265
Programming MSEfirmware Example .....	265
ModuleConfig Read and Write Example .....	266
LVDT Teach Gain Example .....	266
9.3 Initializing the module chain .....	267
Creating a Chain via Broadcasting .....	267
Creating a Chain Manually .....	268
9.4 Getting counts .....	269
9.5 Setting the Encoder Information .....	270
9.6 Latching .....	272
9.7 Referencing 1Vpp Linear Encoder .....	274
9.8 Programming Firmware .....	276
9.9 MseConfigReader .....	277

## 10 C examples ..... 279

10.1 Overview .....	280
10.2 MSeLibraryCEXamples Visual Basic Solution .....	280
Module Examples .....	280
Broadcasting Example .....	280
Async Example .....	280
Subscribe Example .....	280
Config File Example .....	280
Setting the IP Address Example .....	280
Programming the Firmware and Bootloader Example .....	280
10.3 Initialize, Configure, and Get Positions from a 1Vpp Module .....	281

## 11 Visual Basic examples ..... 285

11.1 Overview .....	286
11.2 Module Throughput Test .....	286
11.3 Chain Throughput Test .....	286
11.4 Latching Test .....	286
11.5 Voltage Diagnostics .....	287

## 12 Delphi examples ..... 289

12.1 Overview .....	290
---------------------	-----

## 13 LabVIEW ..... 291

13.1 Introduction .....	292
13.2 Installation .....	292
13.3 LabVIEW VI's and corresponding MSELibrary functions .....	293
MSELibrary1VppWrapperVI.lvlib .....	293
MSELibraryAnalogWrapperVI.lvlib .....	294
MSELibraryConfigFileWrapperVIs.lvlib .....	294
MSELibraryEndatWrapperVIs.lvlib .....	295
MSELibraryPneumaticWrapperVIs.lvlib .....	295
MSELibraryIoWrapperVIs.lvlib .....	296
MSELibraryLvdtWrapperVIs.lvlib .....	296
MSELibraryModuleWrapperVIs.lvlib .....	297
MSELibraryTtlWrapperVIs.lvlib .....	298
13.4 MSELabview DAQ utility explanation and operating instructions .....	299



# 1

**Configuring the  
MSE 1000**

## 1.1 Ethernet cable

A MDI-X enabled router, switch, or network interface card (NIC) is required to utilize a straight through Cat5 Ethernet cable. If not using a MDI-X device a crossover cable is required.

## 1.2 Initial IP address

Each module defaults to DHCP. If a DHCP server is not found, the static IP address of 172.31.46.1 is used for each non power supply module. The power supply module defaults to 172.31.46.2. This will cause an IP conflict until the static IP addresses of the modules are changed.

## 1.3 Changing a modules IP address

The IP address of each non power supply module defaults to 172.31.46.1. The port is always 27015. Communication to the MSE will fail if the IP address for each module in the chain has not been changed to a unique value.



Do not plug or unplug modules under power.  
Damage may occur to internal components.

To configure a unique IP address for each module in a chain:

- ▶ Power off the MSE
- ▶ Unplug all modules in the chain
- ▶ Connect the base module to the power supply module
- ▶ Power up the MSE
- ▶ Change the base module to a unique IP address following the example below
- ▶ Power off the MSE
- ▶ Add a new module
- ▶ Power up the base module and the module added in the previous step
- ▶ Change the IP address of the new module following the example below
- ▶ Repeat this process until all modules in the chain have been configured with a unique IP address

## C++ example

Change IP address:

- ▶ Include the headers and instantiate the MseInterface class.

```
#include "MseInterface.h"
MseInterface mse;
```

- ▶ Create the MSE chain by calling createChain() with the IP address and base port to use for the client PC (this is what the MSE modules will use for responses).

```
MseResults retVal = mse.createChain("172.31.46.3",27016,false, "255.255.255.0");
if(RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

- ▶ Get a reference to the desired module to change the IP address

```
MseModule* module = mse.getModule(0);
if(0 == module)
    std::cout << "handle error" << std::endl;
```

- ▶ Change the IP address and netmask

```
retVal = module->setIp("172.31.46.4", "255.255.255.0");
if(RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

- ▶ Reset the module and wait for it to complete

```
module->resetMse1000();
// Sleep for 10 seconds
```

- ▶ Recreate the chain because the UDP connections have changed

```
MseResults retVal = mse.createChain("172.31.46.3",27016,false, "255.255.255.0");
if(RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

## Changing DHCP 1.4 Changing DHCP

The MSE modules are configured for DHCP. If a DHCP server is not found during startup, the modules will default to the IP configured in **Changing a modules IP address**. If DHCP is not needed, the time needed to look for a DHCP server can be eliminated by disabling DHCP. Modules that have DHCP disabled can be re-enabled.



If there is no DHCP server the PC using the library should have its network card connected to the 172.31.46 domain. Do not use an IP address that is the same as one set in the **Changing a modules IP address** section..

To disable DHCP:

- ▶ Include the headers and instantiate the MseInterface class.

```
#include "MseInterface.h"
MseInterface mse;
```

- ▶ Create the MSE chain by calling createChain() with the IP address and base port to use for the client PC (this is what the MSE modules will use for responses).

```
MseResults retVal = mse.createChain("172.31.46.3",27016,false, "255.255.255.0");
if(RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

- ▶ Get a reference to the desired module to change the DHCP usage

```
MseModule* module = mse.getModule(0);
if(0 == module)
    std::cout << "handle error" << std::endl;
```

```
retVal = module->setDhcp(0);
if(RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

```
// Repeat for each module in the chain
```

- ▶ Reset the module and wait for it to complete

```
module->resetMse1000();
// Sleep for 10 seconds
```

- ▶ Recreate the chain after the DHCP settings have changed

```
MseResults retVal = mse.createChain("172.31.46.3",27016,false, "255.255.255.0");
if(RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```



# 2

**Library software**

## 2.1 General information

Functions are provided for accessing MSE modules from a software application. This group of functions is supplied as a DLL for Microsoft Windows systems.

The following operating systems are supported:

- Windows XP
- Windows Vista
- Windows 7

The MSElibrary is compiled for a 32 bit processor and can be used on 64 bit Windows Vista and 64 bit Windows 7 machines.

In addition to the libraries, header files that enable the functions to be integrated into C/C++ programs are supplied. To create a program, the library must be incorporated into the project. This document groups the C++ and C functions in their own sections and refers to the C++ member functions as methods for clarity. Refer to “C++ examples” on page 263 and “Creating a Chain Manually” on page 270.

LabVIEW VI wrappers and an example are provided with MSElibrary. Refer to “LabVIEW” on page 293.

## 2.2 Installation instructions

The MSElibrary has been tested with Microsoft Visual Basic 6.0, Microsoft Visual Basic 2010, LabVIEW 2012, and Delphi XE3 to show compatibility with C applications.

The MSElibrary Installer saves the tools needed to use the library in the MSElibrary directory and the user’s document directory.

Operating system		MSElibrary directory
Windows XP	32 bit	“C:\Program files\HEIDENHAIN”
Windows Vista	32 bit	“C:\Program files\HEIDENHAIN”
	64 bit	“C:\Program files (x86)\HEIDENHAIN”
Windows 7	32 bit	“C:\Program files\HEIDENHAIN”
	64 bit	“C:\Program files (x86)\HEIDENHAIN”

Operating system		Users document directory
Windows XP	32 bit	“C:\Documents and Settings\CurrentUse\My Documents\HEIDENHAIN”
Windows Vista	32 bit, 64 bit	“C:\Users\CurrentUse\My Documents\HEIDENHAIN”
Windows 7	32 bit, 64 bit	“C:\Users\CurrentUse\My Documents\HEIDENHAIN”

“C:\Program Files” and “C:\Program Files (x86)” are write protected folders in Windows 7, requiring the configurable data to be stored in the user’s data directory.

“C:\Program Files\HEIDENHAIN\MSElibrary” or “C:\Program Files (x86)\HEIDENHAIN\MSElibrary” folder contains:

File	Description
MSElibrary.lib	The .lib file required when linking against the library. The MSElibrary.lib file has the stubs needed to call the functions in the dll.
MSElibrary.dll	The .dll file is needed during runtime. This is where the functions linked against are located.
QtCore4.dll	The .dll file needed for using the core Qt methods. Qt is used for the XML reader and writer.
QtXml4.dll	The .dll file needed for using the Qt XML methods.
“\Docs”	The “\Docs” directory contains this file.
“\Headers”	The “\Headers” folder contains the MSElibrary headers for including into source code to utilize the Mselibrary.

“C:\Documents and Settings\CurrentUse\My Documents\HEIDENHAIN\MSElibrary” or “C:\Users\CurrentUse\My Documents\HEIDENHAIN” folder contains:

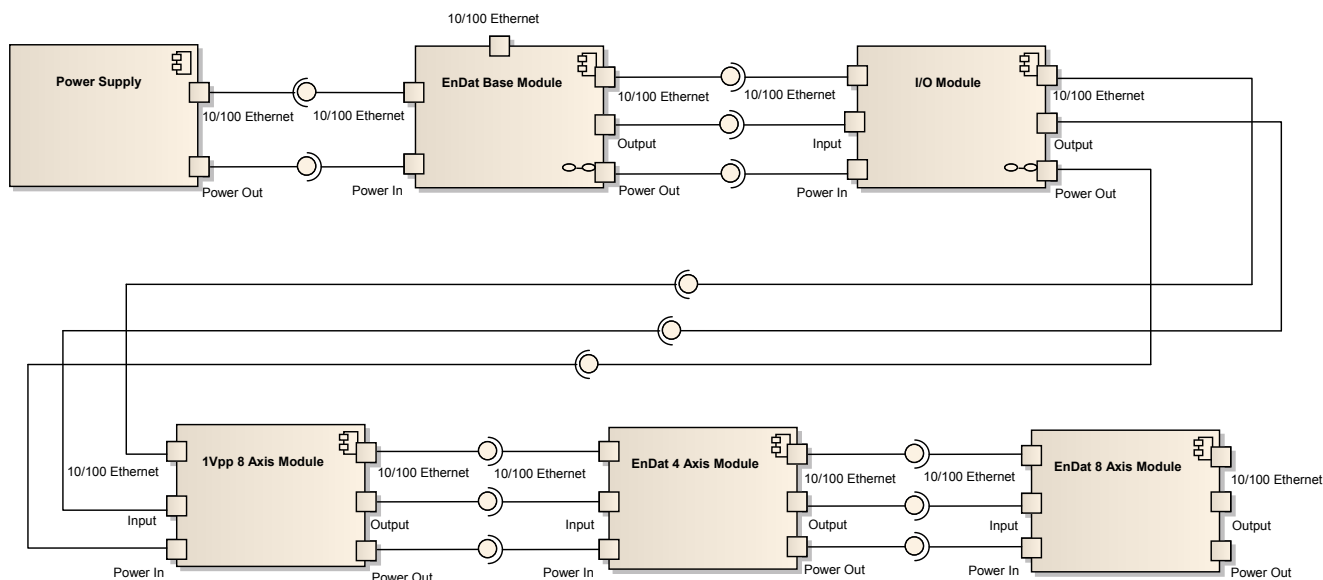
Folder	Description
“\Examples”	The “\Examples” folder contains the “\C++”, “\C”, “\Delphi”, “\LabVIEW”, and “\VisualBasic” subfolders. The “\C++” subfolder contains the C++ MSElibraryCppExample.sln VisualStudio 2010 example solution and source code. The “\C” subfolder contains the C++ MSElibraryCExample.sln VisualStudio 2010 example solution and source code. The MSElibraryCExample.sln utilizes the VisualStudio 2010 C++ compiler but makes C function calls into the MSElibrary.dll to show how the C wrappers are used. The “\Delphi” subfolder contains the Delphi Mse.dproj XE3 example project and source code. The “\LabVIEW” subfolder contains the LabVIEW “LabView MSE 1000 DAQ Utility v100.lvproj” example project and VIs. The “\VisualBasic” subfolder contains the VisualBasic MSEtestbed.sln VisualStudio 2010 example solution and source code.
“\LabVIEW”	The “\LabVIEW” folder contains the LabVIEW VI’s that are needed to interface with the MSElibrary.dll.

## 2.3 Overview

This document is a guide for using the MSElibrary. This document covers the basics needed for initializing the interface to the MSE, setting the encoder parameters, getting counts and positions, programming the firmware, and setting the IP address for the modules.

### Modules

The MSElibrary uses a common design for the base of all the modules. This allows for re-use of most commands as well as the UDP.



Module component diagram

### Definitions

**MSE Chain** Refers to all of the modules in the MSE arrayed sequentially starting from an index of 0.

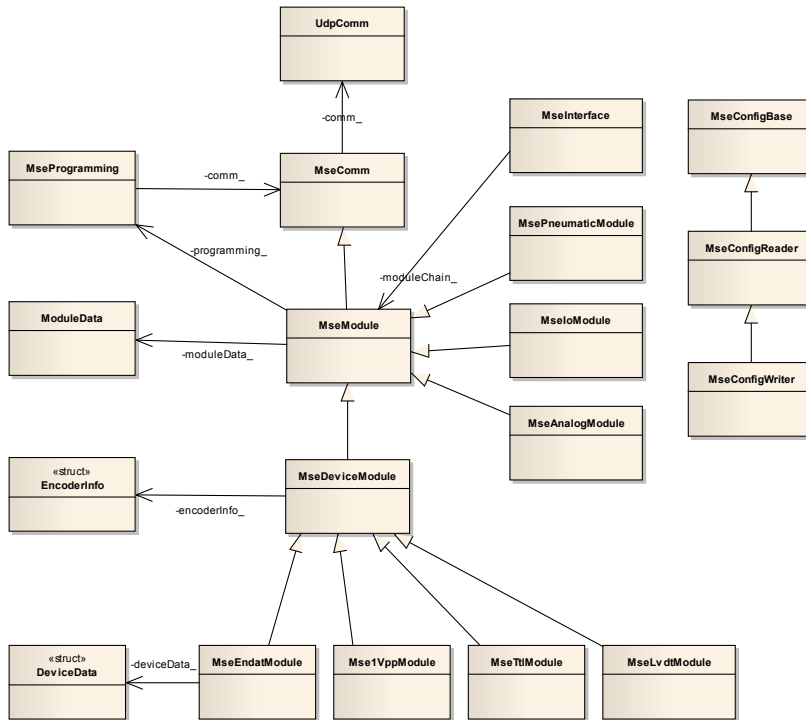
### Prerequisites

The MSElibrary example code requires Microsoft Windows compatible development tools and has been verified with Microsoft Visual Studio 2010.

### Methods and functions

Most of the methods and functions in the MSElibrary (MSElibrary.dll) are made available to the user of the library. This document describes the methods and functions for using the library.

The MSElibrary is written in C++. There are C wrappers for all of the necessary C++ methods in order to expose the calls as C functions. The C wrappers allow programs such as Visual Basic, Delphi, and LabVIEW to utilize the MSElibrary.



Basic class diagram

Class	Description
UdpComm	The UdpComm class is used for the operating system specific UDP code.
MseComm	The MseComm class has the common UDP implementation as well as the commands for setting the IP address and enabling or disabling DHCP for the module.
MseModule	The MseModule class is derived from the MseComm class. The MseModule class has methods for determining the module's type, getting the number of channels and getting the counts. The MseModule class also contains a class used for programming the firmware of a module. The initialization of an MseModule object will connect to a module over UDP and retrieve all of the information from the module and store it.
MseDeviceModule	The MseDeviceModule class is derived from MseModule and has functionality specific to modules with encoders including scaling and retrieving position data.
MseEndatModule	The MseEndatModule class is derived from MseDeviceModule and has EnDat specific initialization and functionality including warning, error, and device.
Mse1VppModule	The Mse1VppModule class is derived from MseDeviceModule and is used in case 1Vpp specific functionality is needed.
MseIoModule	The MseIoModule class is derived from MseModule and has functionality specific to modules with input and output functionality.
MsePneumaticModule	The MsePneumaticModule class is derived from the MseModule and has functionality specific to modules with pneumatic control functionality.
MseAnalogModule	The MseAnalogModule class is derived from MseModule and has functionality specific to modules with analog functionality.
MseLvdtModule	The MseLvdtModule class is derived from MseDeviceModule and has functionality specific to modules with LVDT functionality.
MseTtlModule	The MseTtlModule class is derived from the MseDeviceModule and has functionality specific to modules with TTL functionality.
MseInterface	The MseInterface is the main class used for access of all the MSE features. It provides the ability to create a chain of modules that can be accessed without having to manually instantiate objects and create separate arrays. It is possible to skip using MseInterface and manually instantiate and use any of the exposed library calls of other classes.
MseConfigBase	The MseConfigBase class is a base class used by the MseConfigReader that contains the MSE_XML_RETURN and MSE_XML_ELEMENTS enumerations as well as the methods needed for loading the XML file into memory and decoding the enumerations.

MseConfigReader	The MseConfigReader is derived from the MseConfigBase class. This class is used for reading from the ModuleConfig.xml file that is generated from the MSEsetup application.
MseConfigWriter	The MseConfigWriter is derived from the MseConfigReader class. This class is used for writing to the ModuleConfig.xml file that is generated from the MSEsetup application. The MseConfigWriter is derived from the MseConfigReader class for convenience in order to utilize read functions without having to instantiate two separate classes.

## Wrappers

There are 9 header files that contain the C functions that are used to access the library from C applications. The C functions cannot use inheritance and so must have their own calls into the base classes.

MseModuleWrapper	Provides functions to access the C++ methods that are common to all modules.
MseEndatModuleWrapper	Provides functions to access the C++ methods that are specific to the Endat module.
Mse1VppModuleWrapper	Provides functions to access the C++ methods that are specific to the 1Vpp module.
MseloModuleWrapper	Provides functions to access the C++ methods that are specific to the I/O module.
MsePneumaticModuleWrapper	Provides functions to access the C++ methods that are specific to the pneumatic module.
MseConfigFileWrapper	Provides functions to access the C++ methods that are specific to the configuration file reader and writer.
MseAnalogModuleWrapper	Provides functions to access the C++ methods that are specific to the analog module.
MseTtlModuleWrapper	Provides functions to access the C++ methods that are specific to the TTL module.
MseLvdtModuleWrapper	Provides functions to access the C++ methods that are specific to the LVDT module.

## Module Communication

The modules use UDP for network communication. UDP is non-guaranteed communication. If a message is missed, it can be resent after the timeout period. The timeout period can be modified with the setUdpTimeout method.

The MSElibrary is not threadsafe. This means that multiple threads should not call MSElibrary methods or functions at the same time.

## 2.4 Data Types

The following table shows the size of the data types used in the MSELibrary. Calls into the library must make sure that the parameters and return types match the size of the data type.

Name	Size
Enumeration (like UOM and MODULE_ID)	signed 32 bit integer (4 bytes)
char	signed 8 bit integer (1 byte)
unsigned char	unsigned 8 bit integer (1 byte)
short	signed 16 bit integer (2 bytes)
unsigned short	unsigned 16 bit integer (2 bytes)
int	signed 32 bit integer (4 bytes)
unsigned int	unsigned 32 bit integer (4 bytes)
long	signed 32 bit integer (4 bytes)
unsigned long	unsigned 32 bit integer (4 bytes)
bool	8 bit integer (1 byte)
double	64 bit floating point number (8 bytes)
Mse1VppModulePtr	signed 32 bit integer (4 bytes)
MseAnalogModulePtr	signed 32 bit integer (4 bytes)
MseConfigFilePtr	signed 32 bit integer (4 bytes)
MseEndatModulePtr	signed 32 bit integer (4 bytes)
MseIoModulePtr	signed 32 bit integer (4 bytes)
MseLvdtModulePtr	signed 32 bit integer (4 bytes)
MseModulePtr	signed 32 bit integer (4 bytes)
MsePneumaticModulePtr	signed 32 bit integer (4 bytes)
MseTtlModulePtr	signed 32 bit integer (4 bytes)

## 2.5 Enumerations

### MSE\_CHAIN\_CREATION\_STATE

The MSE\_CHAIN\_CREATION\_STATE enumeration is used to determine the state of the createChain method.

#### Enumeration

```
MSE_CHAIN_CREATION_STATE
{
    MSE_CHAIN_CREATION_IDLE = 0,
    MSE_CHAIN_CREATION_START,
    MSE_CHAIN_CREATION_ORDERING,
    MSE_CHAIN_CREATION_FINISHED,
    MSE_CHAIN_CREATION_FAILED
};
```

#### Parameters

MSE_CHAIN_CREATION_IDLE	The chain creation has not started
MSE_CHAIN_CREATION_START	The broadcast is binding to a socket, sending out the UDP_OPEN broadcast, and receiving responses to the UDP_OPEN command on the bound socket
MSE_CHAIN_CREATION_ORDERING	The responses have been received and the modules are being ordered based on control of the output/input pins
MSE_CHAIN_CREATION_FINISHED	The chain creation has finished successfully
MSE_CHAIN_CREATION_FAILED	The chain creation has failed

### UOM

The UOM enumeration specifies the unit of measurement for the position.

#### Enumeration

```
UOM
{
    UOM_UNDEFINED = 0,
    UOM_RAW_COUNTS,
    UOM_INCHES,
    UOM_MM,
    UOM_DEGREES,
    UOM_COUNT
};
```

#### Parameters

UOM_UNDEFINED = 0	Undefined unit of measurement
UOM_RAW_COUNTS	Raw counts from a device
UOM_INCHES	Inches
UOM_MM	Millimeters
UOM_DEGREES	Decimal degrees
UOM_COUNT	The number of values in the enumeration

**MODULE\_ID**

The MODULE\_ID enumeration specifies the module ID for the module. The module ID can also be found on the module casing and can be retrieved with the showModuleId() method from the MseModule. The showModuleType() method can be used to show the type that is on the module casing.

**Enumeration**

```

MODULE_ID
{
    MODULE_ID_NONE           = 0x000000,
    MODULE_ID_ENDAT_BASE    = 74749901,
    MODULE_ID_ENDAT_BASE_IP40 = 74749902,
    MODULE_ID_1VPP_BASE     = 74750001,
    MODULE_ID_1VPP_BASE_IP40 = 74750002,
    MODULE_ID_PS_120_IP40   = 74750101,
    MODULE_ID_PS_120_IP65   = 74750102,
    MODULE_ID_PS_24         = 74750201,
    MODULE_ID_PS_24_IP40    = 74750202,
    MODULE_ID_ENDAT_4X      = 74750301,
    MODULE_ID_ENDAT_4X_IP40 = 74750302,
    MODULE_ID_ENDAT_8X      = 74750401,
    MODULE_ID_ENDAT_8X_IP40 = 74750402,
    MODULE_ID_1VPP_4X       = 74750501,
    MODULE_ID_1VPP_4X_IP40 = 74750502,
    MODULE_ID_1VPP_8X       = 74750601,
    MODULE_ID_1VPP_8X_IP40 = 74750602,
    MODULE_ID_IO_IP40       = 74750701,
    MODULE_ID_IO_IP65       = 74750702,
    MODULE_ID_PNEUMATIC     = 74750801,
    MODULE_ID_PNEUMATIC_IP40 = 74750802,
    MODULE_ID_ANALOG        = 74750901,
    MODULE_ID_ANALOG_IP40   = 74750902,
    MODULE_ID_TTL_BASE      = 74751101,
    MODULE_ID_TTL_BASE_IP40 = 74751102,
    MODULE_ID_TTL_4X        = 74751201,
    MODULE_ID_TTL_4X_IP40   = 74751202,
    MODULE_ID_TTL_8X        = 74751301,
    MODULE_ID_TTL_8X_IP40   = 74751302,
    MODULE_ID_HBT           = 74751401,
    MODULE_ID_HBT_IP40      = 74751404,
    MODULE_ID_VLDT          = 74751402,
    MODULE_ID_VLDT_IP40     = 74751405,
    MODULE_ID_LVDT          = 74751403,
    MODULE_ID_LVDT_IP40     = 74751406
};

```



## Parameters

MODULE_ID_NONE	= 0x000000	Default case, Only should happen if no firmware is loaded
MODULE_ID_ENDAT_BASE	= 74749901	EnDat module with Ethernet, serial, and 4 channels. IP65 version.
MODULE_ID_ENDAT_BASE_IP40	= 74749902	EnDat module with Ethernet, serial, and 4 channels. IP40 version.
MODULE_ID_1VPP_BASE	= 74750001	1 Vpp module with Ethernet, serial and 4 channels. IP65 version.
MODULE_ID_1VPP_BASE_IP40	= 74750002	1 Vpp module with Ethernet, serial and 4 channels. IP40 version.
MODULE_ID_PS_120_IP40	= 74750101	120/240 Vac Power Supply. IP40 version.
MODULE_ID_PS_120_IP65	= 74750102	120/240 Vac Power Supply. IP65 version.
MODULE_ID_PS_24	= 74750201	24 Vdc Power Supply. IP65 version.
MODULE_ID_PS_24_IP40	= 74750202	24 Vdc Power Supply. IP40 version.
MODULE_ID_ENDAT_4X	= 74750301	EnDat module with 4 channels. IP65 version.
MODULE_ID_ENDAT_4X_IP40	= 74750302	EnDat module with 4 channels. IP40 version.
MODULE_ID_ENDAT_8X	= 74750401	EnDat module with 8 channels. IP65 version.
MODULE_ID_ENDAT_8X_IP40	= 74750402	EnDat module with 8 channels. IP40 version.
MODULE_ID_1VPP_4X	= 74750501	1 Vpp module with 4 channels. IP65 version.
MODULE_ID_1VPP_4X_IP40	= 74750502	1 Vpp module with 4 channels. IP40 version.
MODULE_ID_1VPP_8X	= 74750601	1 Vpp module with 8 channels. IP65 version.
MODULE_ID_1VPP_8X_IP40	= 74750602	1 Vpp module with 8 channels. IP40 version.
MODULE_ID_IO_IP40	= 74750701	I/O module. IP40 version.
MODULE_ID_IO_IP65	= 74750702	I/O module. IP65 version.
MODULE_ID_PNEUMATIC	= 74750801	Pneumatic. IP65 version.
MODULE_ID_PNEUMATIC_IP40	= 74750802	Pneumatic. IP40 version.
MODULE_ID_ANALOG	= 74750901	Analog module. IP65 version.
MODULE_ID_ANALOG_IP40	= 74750902	Analog module. IP40 version.
MODULE_ID_TTL_BASE	= 74751101	TTL module with Ethernet, serial, and 4 channels. IP65 version.
MODULE_ID_TTL_BASE_IP40	= 74751102	TTL module with Ethernet, serial, and 4 channels. IP40 version.
MODULE_ID_TTL_4X	= 74751201	TTL module with 4 channels. IP65 version.
MODULE_ID_TTL_4X_IP40	= 74751202	TTL module with 4 channels. IP40 version.
MODULE_ID_TTL_8X	= 74751301	TTL module with 8 channels. IP65 version.
MODULE_ID_TTL_8X_IP40	= 74751302	TTL module with 8 channels. IP40 version.
MODULE_ID_HBT	= 74751401	LVDT module with 8 channels for Solatron compatible half bridge sensors . IP65 version.
MODULE_ID_HBT_IP40	= 74751404	LVDT module with 8 channels for Solatron compatible half bridge sensors. IP40 version.
MODULE_ID_VLDT	= 74751402	VLDT module with 8 channels for Mahr compatible VLDT (Very Linear Differential Transducer) sensors . IP65 version.
MODULE_ID_VLDT_IP40	= 74751405	VLDT module with 8 channels for Mahr compatible VLDT (Very Linear Differential Transducer) sensors. IP40 version.
MODULE_ID_LVDT	= 74751403	LVDT module with 8 channels for Marposh compatible full bridge sensors. IP65 version.
MODULE_ID_LVDT_IP40	= 74751406	LVDT module with 8 channels for Marposh compatible full bridge sensors. IP40 version.

## ENDAT\_ERROR\_RESULT

The ENDAT\_ERROR\_RESULT enumeration is for the error and warning values specific to the EnDat encoders that can be obtained from the MSE over UDP.

### Enumeration

```

ENDAT_ERROR_RESULT
{
    EET_OK = 0,
    EET_BAD,
    EET_NOT_SUPPORTED,
    EET_COUNT
};

```

### Parameters

EET_OK = 0	There is currently no error or warning
EET_BAD	There is currently an error or warning
EET_NOT_SUPPORTED	The warning or error information is not supported
EET_COUNT	The number of values in the enumeration

## ENDAT\_ERRORS

The ENDAT\_ERRORS enumeration is for the EnDat errors.

### Enumeration

```

ENDAT_ERRORS
{
    ENDAT_ERRORS_LIGHT,
    ENDAT_ERRORS_SIGNALAMP,
    ENDAT_ERRORS_POSVALUE,
    ENDAT_ERRORS_OVERVOLT,
    ENDAT_ERRORS_UNDERVOLT,
    ENDAT_ERRORS_OVERCUR,
    ENDAT_ERRORS_BATTERY
};

```

### Parameters

ENDAT_ERRORS_LIGHT= 0	Light
ENDAT_ERRORS_SIGNALAMP	Signal amplitude
ENDAT_ERRORS_POSVALUE	Position value
ENDAT_ERRORS_OVERVOLT	Over voltage
ENDAT_ERRORS_UNDERVOLT	Under voltage
ENDAT_ERRORS_OVERCUR	Over current
ENDAT_ERRORS_BATTERY	Battery

## NUM\_ENDAT\_ERRORS

The NUM\_ENDAT\_ERRORS enumeration is for the number of ENDAT errors that are enumerated in ENDAT\_ERRORS.

### Enumeration

```

NUM_ENDAT_ERRORS = 7;

```

## ENDAT\_WARNINGS

The ENDAT\_WARNINGS enumeration is for the EnDat warning.

### Enumeration

```
ENDAT_WARNINGS
{
    ENDAT_WARNING_FREQCOLLISION = 0,
    ENDAT_WARNING_TEMPEXCEEDED,
    ENDAT_WARNING_LIGHT_RESERVE,
    ENDAT_WARNING_BATTERYCHARGE,
    ENDAT_WARNING_TRAVERSEREFPT
};
```

### Parameters

ENDAT_WARNING_FREQCOLLISION = 0	Frequency Collision
ENDAT_WARNING_TEMPEXCEEDED	Temperature warning range has been exceeded
ENDAT_WARNING_LIGHTREVERSE	Light source control reserve
ENDAT_WARNING_BATTERYCHARGE	Battery charge
ENDAT_WARNING_TRAVERSEREFPT	Traverse Reference Point

## NUM\_ENDAT\_WARNINGS

The NUM\_ENDAT\_WARNINGS enumeration is the number of ENDAT warnings that are enumerated in ENDAT\_WARNINGS.

### Enumeration

```
NUM_ENDAT_WARNINGS = 5;
```

## ENDAT\_DIAG

The ENDAT\_DIAG enumeration is for the EnDat diagnostic. The getDiag method of the MseEndatModule class fills in the parameter diagVals with an array of unsigned chars. This array can be indexed by this enumeration to access the desired EnDat diagnostic value.

### Enumeration

```
ENDAT_DIAG
{
    ENDAT_DIAG_ABS_TRACK_SUPPORTED,
    ENDAT_DIAG_ABS_TRACK_VALUE,
    ENDAT_DIAG_ABS_TRACK_MIN,
    ENDAT_DIAG_INC_TRACK_SUPPORTED,
    ENDAT_DIAG_INC_TRACK_VALUE,
    ENDAT_DIAG_INC_TRACK_MIN,
    ENDAT_DIAG_POS_VAL_CALC_SUPPORTED,
    ENDAT_DIAG_POS_VAL_CALC_VALUE,
    ENDAT_DIAG_POS_VAL_CALC_MIN,
    ENDAT_DIAG_COUNT
};
```

### Parameters

ENDAT_DIAG_ABS_TRACK_SUPPORTED	Absolute tracking function reserves supported. This value will be either EET_OK or EET_NOT_SUPPORTED which are members of the ENDAT_ERROR_RESULT enumeration.
ENDAT_DIAG_ABS_TRACK_VALUE	Absolute tracking function reserves value
ENDAT_DIAG_ABS_TRACK_MIN	The minimum recorded absolute tracking function reserve value
ENDAT_DIAG_INC_TRACK_SUPPORTED	Incremental tracking function reserves supported. This value will be either EET_OK or EET_NOT_SUPPORTED which are members of the ENDAT_ERROR_RESULT enumeration.
ENDAT_DIAG_INC_TRACK_VALUE	Incremental tracking function reserves value
ENDAT_DIAG_INC_TRACK_MIN	The minimum recorded incremental tracking function reserve value
ENDAT_DIAG_POS_VAL_CALC_SUPPORTED	Position value calculation function reserves supported. This value will be either EET_OK or EET_NOT_SUPPORTED which are members of the ENDAT_ERROR_RESULT enumeration.
ENDAT_DIAG_POS_VAL_CALC_VALUE	Position value calculation function reserves value
ENDAT_DIAG_POS_VAL_CALC_MIN	The minimum recorded position value calculation function reserve value
ENDAT_DIAG_COUNT	The number of values in the enumeration

## MSE\_RESPONSE\_CODE

The MSE\_RESPONSE\_CODE enumeration is for the error and warning values that can be obtained from the MSE over UDP.

### Enumeration

```

MSE_RESPONSE_CODE
{
    RESPONSE_OK = 0,
    RESPONSE_TIMEOUT,
    RESPONSE_TYPE_INVALID,
    RESPONSE_RANGE_MODULE_INVALID,
    RESPONSE_TYPE_MISMATCH,
    RESPONSE_PARAMETER_COUNT_ZERO,
    RESPONSE_PARAMETER_COUNT_MISMATCH,
    RESPONSE_PARAMETER_SEND_SIZE_MISMATCH,
    RESPONSE_PARAMETER_SIZE_MISMATCH,
    RESPONSE_PARAMETER_INVALID_CHANNEL,
    RESPONSE_BROADCAST_SEND_FAILED,
    RESPONSE_BROADCAST_NO_RESPONSE,
    RESPONSE_UDP_SOCKET_BIND_FAILED,
    RESPONSE_UDP_SERVER_CREATION_FAILED,
    RESPONSE_UDP_READ_FLUSH_FAILED,
    RESPONSE_UDP_SEND_FAILED,
    RESPONSE_UDP_READ_FAILED,
    RESPONSE_UDP_RECV_PORT_INVALID,
    RESPONSE_ASYNC_RECV_FAILED,
    RESPONSE_DOWNLOAD_ENDED_OK,
    RESPONSE_COMM_NOT_INITIALIZED,
    RESPONSE_MODULE_NOT_INITIALIZED,
    RESPONSE_IP_ALREADY_USED,
    RESPONSE_MODULE_ALREADY_CREATED,
    RESPONSE_IP_MALFORMED,
    RESPONSE_NETMASK_MALFORMED,
    RESPONSE_IP_MISMATCH,
    RESPONSE_PROGRAM_FILE_PREPARE_FAILED,
    RESPONSE_MODULE_NULL,
    RESPONSE_MODULE_MISMATCH,
    RESPONSE_MODULE_NOT_CONNECTED,
    RESPONSE_MODULE_FIRST_NOT_FOUND,
    RESPONSE_POINTER_PARAMETER_NULL,
    RESPONSE_RANGE_ERROR,
    RESPONSE_FRAM_ERROR,
    RESPONSE_FILE_OPEN_FAILED,
    RESPONSE_FILE_READ_ERROR,
    RESPONSE_MODULE_NOT_IN_BOOTLOADER,
    RESPONSE_ERROR,
    RESPONSE_INVALID_REVISION,
    RESPONSE_INVALID_IMAGE,
    RESPONSE_CHECKSUM_FAILED,
    RESPONSE_FIRMWARE_NOT_LOADED,
    RESPONSE_CANT_PROGRAM_WHILE_DHCP,
    RESPONSE_COUNT
};

```

### Parameters

RESPONSE_OK = 0	Used to represent a good response
RESPONSE_TIMEOUT	A timeout occurred waiting for the response
RESPONSE_TYPE_INVALID	An invalid response type was received
RESPONSE_RANGE_MODULE_INVALID	A module was requested that is out of range
RESPONSE_TYPE_MISMATCH	A response type different from what was expected was received
RESPONSE_PARAMETER_COUNT_ZERO	The number of parameters received is zero but this is not what was expected
RESPONSE_PARAMETER_COUNT_MISMATCH	The number of parameters received does not match what was expected
RESPONSE_PARAMETER_SEND_SIZE_MISMATCH	The command to send was passed in invalid data
RESPONSE_PARAMETER_SIZE_MISMATCH	The size of the data received is not what was expected
RESPONSE_PARAMETER_INVALID_CHANNEL	The channel received is not what was expected

RESPONSE_BROADCAST_SEND_FAILED	The broadcast command could not be sent out the network correctly
RESPONSE_BROADCAST_NO_RESPONSE	Broadcast failed to get a response
RESPONSE_UDP_SOCKET_BIND_FAILED	The binding of the UDP socket failed.
RESPONSE_UDP_SERVER_CREATION_FAILED	The creation of the UDP server failed
RESPONSE_UDP_READ_FLUSH_FAILED	The flush of the UDP read buffer failed
RESPONSE_UDP_SEND_FAILED	The sending of the UDP datagram failed
RESPONSE_UDP_READ_FAILED	The reading of the UDP datagram failed
RESPONSE_UDP_RECV_PORT_INVALID	The UDP port for receiving MSE responses is invalid
RESPONSE_ASYNC_RECV_FAILED	Currently not utilized
RESPONSE_DOWNLOAD_ENDED_OK	Used to distinguish from a RESPONSE_OK, because downloading the program requires multiple segments
RESPONSE_COMM_NOT_INITIALIZED	The communication to the module has not been initialized
RESPONSE_MODULE_NOT_INITIALIZED	The module data has not been initialized
RESPONSE_IP_ALREADY_USED	The IP address is used by another module
RESPONSE_MODULE_ALREADY_CREATED	The module cannot be created twice
RESPONSE_IP_MALFORMED	The IP address is not in the correct format
RESPONSE_NETMASK_MALFORMED	The netmask is not in the correct format
RESPONSE_IP_MISMATCH	The IP address in the return packet does not match the module that sent the command
RESPONSE_PROGRAM_FILE_PREPARE_FAILED	The file preparation for the programming of the module failed
RESPONSE_MODULE_NULL	A NULL pointer was returned when trying to request a module
RESPONSE_MODULE_MISMATCH	The module requested is not the same as the one in the MSE
RESPONSE_MODULE_NOT_CONNECTED	The IS_CONNECT_IN_SET pin used for ordering the modules after a broadcast did not go high when expected
RESPONSE_MODULE_FIRST_NOT_FOUND	The first module could not be identified
RESPONSE_POINTER_PARAMETER_NULL	A NULL pointer was passed into a function
RESPONSE_RANGE_ERROR	A parameter value is out of range
RESPONSE_FRAM_ERROR	The FRAM request returned an error
RESPONSE_FILE_OPEN_FAILED	The file requested could not be opened
RESPONSE_FILE_READ_ERROR	The file requested could not be read
RESPONSE_MODULE_NOT_IN_BOOTLOADER	The module is not in the bootloader
RESPONSE_ERROR	An unknown error
RESPONSE_INVALID_REVISION	The revision of the firmware is invalid
RESPONSE_INVALID_IMAGE	The image selected for programming a module is invalid
RESPONSE_CHECKSUM_FAILED	The checksum has failed
RESPONSE_FIRMWARE_NOT_LOADED	The firmware is not loaded in the module
RESPONSE_CANT_PROGRAM_WHILE_DHCP	The bootloader and firmware cannot be programmed while a module is configured for DHCP
RESPONSE_COUNT	The number of values in the enumeration

## LATCH\_OPTIONS

The LATCH\_OPTIONS enumeration is used to select the latching option.

### Enumeration

```
LATCH_OPTIONS
{
    LATCH_COUNT_SET = 0,
    LATCH_COUNT_RESET
};
```

### Parameters

LATCH_COUNT_SET	The latching is set (activated)
LATCH_COUNT_RESET	The latching is reset (deactivated)

## LATCH\_CHOICE

The LATCH\_CHOICE enumeration is used to select the latch to trigger. There are 5 latch lines. The first 3 are used for software latches. The last 2 are for the footswitches. The footswitch lines can be simulated with a software command. The enumeration LATCH\_CHOICE\_ALL is currently used as a way to clear all latches at one time with the setLatch method.

### Enumeration

```
LATCH_CHOICE
{
    LATCH_CHOICE_SOFTWARE_1 = 0,
    LATCH_CHOICE_SOFTWARE_2,
    LATCH_CHOICE_SOFTWARE_3,
    LATCH_CHOICE_FOOTSWITCH_1,
    LATCH_CHOICE_FOOTSWITCH_2,
    LATCH_CHOICE_ALL,
    LATCH_CHOICE_NONE
};
```

### Parameters

LATCH_CHOICE_SOFTWARE_1	The first trigger line
LATCH_CHOICE_SOFTWARE_2	The second trigger line
LATCH_CHOICE_SOFTWARE_3	The third trigger line
LATCH_CHOICE_FOOTSWITCH_1	The fourth trigger line
LATCH_CHOICE_FOOTSWITCH_2	The fifth trigger line
LATCH_CHOICE_ALL	Used to clear all triggers with the setLatch method
LATCH_CHOICE_NONE	None

## ROTARY\_FORMAT

The ROTARY\_FORMAT enumeration is used to set the type of formatting to apply to an EnDat, 1 Vpp or TTL rotary encoder. The format is applied when calling the getPositions method of the EnDat, 1 Vpp or TTL modules.

### Enumeration

```
ROTARY_FORMAT
{
    ROTARY_FORMAT_360,
    ROTARY_FORMAT_PLUS_MINUS_360,
    ROTARY_FORMAT_PLUS_MINUS_180,
    ROTARY_FORMAT_INFINITE_PLUS_MINUS,
    ROTARY_FORMAT_UNKNOWN
};
```

### Parameters

ROTARY_FORMAT_360	The rotary position is always from 0 to 360
ROTARY_FORMAT_PLUS_MINUS_360	The rotary position is between 0 to 360 when rotating clockwise from the starting 0 position of the encoder. The rotary position is between 0 to -360 when rotating counterclockwise from the starting 0 position of the encoder. The starting 0 position, for relative encoders, is set when the modules are first powered on and when referencing is performed. The 0 position of an absolute encoder is always in the same position.
ROTARY_FORMAT_PLUS_MINUS_180	The rotary position goes from 0 to 180 and then from -180 to 0 when rotating clockwise. The rotary position goes from 0 to -180 and then from 180 to 0 when rotating counterclockwise.

ROTARY_FORMAT_INFINITE_PLUS_MINUS	For 1 Vpp and TTL encoders: The rotary position is between 0 to the largest 64 bit double value when rotating clockwise from the starting 0 position of the encoder. The rotary position is between 0 to smallest 64 bit double value when rotating counterclockwise from the starting 0 position of the encoder. The starting 0 position is set when the modules are first powered on and when referencing is performed. For EnDat encoders: The rotary position is between 0 to a positive 64 bit double value based on the number of revolutions that the encoder can store when rotating clockwise from the 0 position of the encoder. The rotary position is between 0 to a negative 64 bit double value based on the number of revolutions that the encoder can store when rotating counterclockwise from the 0 position of the encoder. The 0 position of an absolute encoder is always in the same position.
ROTARY_FORMAT_UNKNOWN	Unknown

## ADC\_OPTIONS

The ADC\_OPTIONS enumeration is used to select the analog to digital conversion value to read. The getAdcValues method of the MseModuleclass fills in the parameter adcVals with an array of shorts. This array can be indexed by this enumeration to access the desired ADC value.

### Enumeration

```
ADC_OPTIONS
{
    ADC_CH0,
    ADC_CH1,
    ADC_CH2,
    ADC_CH3,
    ADC_TEMP,
    ADC_NUM_CHANNELS
};
```

### Parameters

ADC_CH0	The voltage being read from the ADC on channel 0 in millivolts (reads the 3.3 V supply on the power supply and the 5 V supply on other modules)
ADC_CH1	The voltage being read from the ADC on channel 1 in millivolts (reads the 24 V supply on the power supply and is not used on other modules)
ADC_CH2	The voltage being read from the ADC on channel 2 in millivolts (reads the current draw on the power supply and not used on other modules)
ADC_CH3	The voltage being read from the ADC on channel 3 in millivolts (reads the ground on the power supply and the 3.3 V supply on other modules)
ADC_TEMP	The temperature of the CPU in Celsius * 10. Divide by ten to calculate the degrees in Celsius.
ADC_NUM_CHANNELS	The number of ADC channels available

## COUNT\_REQUEST\_OPTION

The COUNT\_REQUEST\_OPTION enumeration is used to select the type of count value to return.

### Enumeration

```
COUNT_REQUEST_OPTION
{
    COUNT_REQUEST_LATEST,
    COUNT_REQUEST_LATCHED,
};
```

### Parameters

COUNT_REQUEST_LATEST	The latest count value should be returned
COUNT_REQUEST_LATCHED	The last latched count value should be returned

## ENCODER\_TYPES\_ENUM

The ENCODER\_TYPES\_ENUM enumeration is used to store the type of encoder attached to a channel.

### Enumeration

```

ENCODER_TYPES_ENUM
{
    ENCODER_TYPE_NONE,
    ENCODER_TYPE_LINEAR,
    ENCODER_TYPE_GAUGE,
    ENCODER_TYPE_ROTARY
};

```

### Parameters

ENCODER_TYPE_NONE	Enumeration for unknown or no encoder
ENCODER_TYPE_LINEAR	Enumeration for a linear encoder
ENCODER_TYPE_GAUGE	Enumeration for a gauge encoder
ENCODER_TYPE_ROTARY	Enumeration for a rotary encoder

## VPP\_VOLTAGE\_FEEDBACK

The VPP\_VOLTAGE\_FEEDBACK enumeration is used for the voltage feedback for the A and B signals for a specific encoder. The getDiag method of the Mse1VppModule class fills in the parameter diagVals with an array of doubles. This array can be indexed by this enumeration to access the desired voltage value.

### Enumeration

```

VPP_VOLTAGE_FEEDBACK
{
    VPP_VOLTAGE_SIGNAL_A_MV,
    VPP_VOLTAGE_SIGNAL_B_MV,
    VPP_VOLTAGE_NUM
};

```

### Parameters

VPP_VOLTAGE_SIGNAL_A_MV	The last reading for the signal A voltage in millivolts
VPP_VOLTAGE_SIGNAL_B_MV	The last reading for the signal B voltage in millivolts
VPP_VOLTAGE_NUM	The number of values in the enumeration



## INTEGRITY\_ENUMS

The INTEGRITY\_ENUMS enumeration is used for masking the integrity value to determine which warning or error occurred in the module. The getIntegrity method of the MseModule class fills in the parameter integrity with an unsigned long. This value can be masked by this enumeration to determine which warnings or errors have occurred.

### Enumeration

```
INTEGRITY_ENUMS
{
    INTEGRITY_CURRENT_WARNING           = 0x00000001,
    INTEGRITY_CURRENT_ERROR             = 0x00000002,
    INTEGRITY_24V_LOW_ERROR             = 0x00000004,
    INTEGRITY_24V_HIGH_ERROR            = 0x00000008,
    INTEGRITY_24V_LOW_WARNING           = 0x00000010,
    INTEGRITY_24V_HIGH_WARNING          = 0x00000020,
    INTEGRITY_5V_LOW_ERROR              = 0x00000040,
    INTEGRITY_5V_HIGH_ERROR             = 0x00000080,
    INTEGRITY_5V_LOW_WARNING            = 0x00000100,
    INTEGRITY_5V_HIGH_WARNING           = 0x00000200,
    INTEGRITY_TEMPERATURE_LOW_ERROR     = 0x00000400,
    INTEGRITY_TEMPERATURE_HIGH_ERROR    = 0x00000800,
    INTEGRITY_TEMPERATURE_LOW_WARNING   = 0x00001000,
    INTEGRITY_TEMPERATURE_HIGH_WARNING  = 0x00002000,
    INTEGRITY_FRAM_ERROR                = 0x00004000,
    INTEGRITY_FRAM_RECOVERED            = 0x00008000
};
```

### Parameters

INTEGRITY_CURRENT_WARNING	Whether the current has exceeded the warning threshold (only used for the power supply)
INTEGRITY_CURRENT_ERROR	Whether the current has exceeded the error threshold (only used for the power supply)
INTEGRITY_24V_LOW_ERROR	Whether the 24V supply has exceeded the low error threshold (only used for the power supply)
INTEGRITY_24V_HIGH_ERROR	Whether the 24V supply has exceeded the high error threshold (only used for the power supply)
INTEGRITY_24V_LOW_WARNING	Whether the 24V supply has exceeded the low warning threshold (only used for the power supply)
INTEGRITY_24V_HIGH_WARNING	Whether the 24V supply has exceeded the high warning threshold (only used for the power supply)
INTEGRITY_5V_LOW_ERROR	Whether the 5V supply has exceeded the low error threshold (only used for non-power supply modules)
INTEGRITY_5V_HIGH_ERROR	Whether the 5V supply has exceeded the high error threshold (only used for non-power supply modules)
INTEGRITY_5V_LOW_WARNING	Whether the 5V supply has exceeded the low warning threshold (only used for non-power supply modules)
INTEGRITY_5V_HIGH_WARNING	Whether the 5V supply has exceeded the high warning threshold (only used for non-power supply modules)
INTEGRITY_TEMPERATURE_LOW_ERROR	Whether the temperature has exceeded the low error threshold (used for all modules)
INTEGRITY_TEMPERATURE_HIGH_ERROR	Whether the temperature has exceeded the high error threshold (used for all modules)
INTEGRITY_TEMPERATURE_LOW_WARNING	Whether the temperature has exceeded the low warning threshold (used for all modules)
INTEGRITY_TEMPERATURE_HIGH_WARNING	Whether the temperature has exceeded the high warning threshold (used for all modules)
INTEGRITY_FRAM_ERROR	Whether the non-volatile configuration data of the module has failed to be overwritten by the backup data after detection of memory corruption (used for all modules). The module will utilize a default IP address, netmask, and MAC address and will need to be sent back to HEIDENHAIN for reprogramming.
INTEGRITY_FRAM_RECOVERED	Whether the non-volatile configuration data of the module has been overwritten by the backup data because of memory corruption (used for all modules)

## REFERENCE\_MARK\_ENUM

The REFERENCE\_MARK\_ENUM enumeration is used for the type of reference mark used by a 1Vpp or TTL encoder.

### Enumeration

```
REFERENCE_MARK_ENUM
{
    REFERENCE_MARK_NONE,
    REFERENCE_MARK_SINGLE,
    REFERENCE_MARK_CODED_500,
    REFERENCE_MARK_CODED_1000,
    REFERENCE_MARK_CODED_2000,
    REFERENCE_MARK_CODED_5000,
    REFERENCE_MARK_CODED_ANGULAR
};
```

### Parameters

REFERENCE_MARK_NONE	No reference mark
REFERENCE_MARK_SINGLE	A single reference mark
REFERENCE_MARK_CODED_500	Reference marks at a 500 signal period spacing
REFERENCE_MARK_CODED_1000	Reference marks at a 1000 signal period spacing
REFERENCE_MARK_CODED_2000	Reference marks at a 2000 signal period spacing
REFERENCE_MARK_CODED_5000	Reference marks at a 5000 signal period spacing
REFERENCE_MARK_CODED_ANGULAR	Reference marks spaced based on the line count

## REFERENCE\_MARK\_STATE

The REF\_MARK\_STATE enumeration is used to determine the state of the referencing used by the 1Vpp and TTL encoders.

### Enumeration

```
REF_MARK_STATE
{
    REF_MARK_OFF,
    REF_MARK_STARTED,
    REF_MARK_FIND_FIRST,
    REF_MARK_FIND_SECOND,
    REF_MARK_FINISHED
};
```

### Parameters

REF_MARK_OFF	Not referenced
REF_MARK_STARTED	Referencing has started
REF_MARK_FIND_FIRST	Referencing has found the first reference mark
REF_MARK_FIND_SECOND	Referencing has found the second reference mark
REF_MARK_FINISHED	Referencing has finished

## COUNTER\_STATUS

The COUNTER\_STATUS enumeration are used to determine the counter status. The getChannelStatus method of the MseDeviceModule class fills in the parameter channelStatus with an unsigned char. This value can be masked by this enumeration to determine the error status. This method is only used to get the channel error status for a EnDat, 1Vpp, or TTL encoder. EnDat encoders should use the getErrors and getWarnings methods and read the counter status for an added check.

### Enumeration

```
COUNTER_STATUS
{
    COUNTER_STATUS_EDGE_DISTANCE_ERROR           = 0x0001,
    COUNTER_STATUS_AVG_ADDER_OVERFLOW           = 0x0002,
    COUNTER_STATUS_TOO_MANY_AVERAGE_SAMPLES    = 0x0004,
    COUNTER_STATUS_TOUCH_PROBE_OVERFLOW        = 0x0008,
    COUNTER_STATUS_FILTER_SPIKE_DETECTED        = 0x0010,
    COUNTER_STATUS_AMPLITUDE_MIN_ERROR         = 0x0020,
    COUNTER_STATUS_AMPLITUDE_MIN_WARNING       = 0x0040,
    COUNTER_STATUS_AMPLITUDE_MAX_ERROR         = 0x0080
};
```

### Parameters

COUNTER_STATUS_EDGE_DISTANCE_ERROR	This error occurs when the dg00 and dg90 inputs change simultaneously within one system clock cycle resulting in position errors. This error can be reported on EnDat, 1Vpp, and TTL encoders.
COUNTER_STATUS_AVG_ADDER_OVERFLOW	Currently not used
COUNTER_STATUS_TOO_MANY_AVERAGE_SAMPLES	Currently not used
COUNTER_STATUS_TOUCH_PROBE_OVERFLOW	Currently not used
COUNTER_STATUS_FILTER_SPIKE_DETECTED	This error occurs when the filter for the dg00 and dg90 has suppressed input spikes resulting in position errors. This error can be reported on EnDat and 1Vpp encoders.
COUNTER_STATUS_AMPLITUDE_MIN_ERROR	The amplitude of the dg00 and dg90 signals are out of range resulting in position errors. This error can be reported on EnDat and 1Vpp encoders.
COUNTER_STATUS_AMPLITUDE_MIN_WARNING	The amplitude of the dg00 and dg90 signals are close to being out of range. This error can be reported on EnDat and 1Vpp encoders.
COUNTER_STATUS_AMPLITUDE_MAX_ERROR	The amplitude of the dg00 and dg90 signals are out of range resulting in position errors. This error can be reported on EnDat and 1Vpp encoders.

## MSE\_XML\_RETURN

The MSE\_XML\_RETURN enumeration are used for the return values of the MseConfigReader and MseConfigWriter.

### Enumeration

```

MSE_XML_RETURN
{
    MSE_XML_RETURN_OK                = 0,
    MSE_XML_RETURN_INVALID_TAG,
    MSE_XML_RETURN_INVALID_FILE,
    MSE_XML_RETURN_DOM_CREATION_FAILED,
    MSE_XML_RETURN_DOM_NOT_CREATED,
    MSE_XML_RETURN_DOM_NULL,
    MSE_XML_RETURN_DOM_ROOT_ELEMENT_NULL,
    MSE_XML_RETURN_INVALID_MODULE_CONFIG_TAGNAME,
    MSE_XML_RETURN_INVALID_MODULE_TAGNAME,
    MSE_XML_RETURN_INVALID_CHANNEL_TAGNAME,
    MSE_XML_RETURN_INVALID_MODULE,
    MSE_XML_RETURN_INVALID_MODULE_LIST,
    MSE_XML_RETURN_INVALID_CHANNEL,
    MSE_XML_RETURN_INVALID_CHANNEL_LIST,
    MSE_XML_RETURN_TAGNAME_NOT_FOUND,
    MSE_XML_RETURN_DATA_NOT_CHANGED,
    MSE_XML_RETURN_NULL_POINTER,
    MSE_XML_RETURN_NUM
};

```

### Parameters

MSE_XML_RETURN_OK	No error
MSE_XML_RETURN_INVALID_TAG	Invalid tag name
MSE_XML_RETURN_INVALID_FILE	The ModuleConfig file passed in is not valid
MSE_XML_RETURN_DOM_CREATION_FAILED	The DOM parser could not load the file into memory
MSE_XML_RETURN_DOM_NOT_CREATED	The DOM parser was never created
MSE_XML_RETURN_DOM_NULL	The DOM parser is NULL
MSE_XML_RETURN_DOM_ROOT_ELEMENT_NULL	The root element of the DOM parser is NULL
MSE_XML_RETURN_INVALID_MODULE_CONFIG_TAGNAME	The ModuleConfig tag name in the ModuleConfig file is invalid
MSE_XML_RETURN_INVALID_MODULE_TAGNAME	The tag name requested for the module is invalid
MSE_XML_RETURN_INVALID_CHANNEL_TAGNAME	The tag name requested for the channel is invalid
MSE_XML_RETURN_INVALID_MODULE	The module requested is invalid
MSE_XML_RETURN_INVALID_MODULE_LIST	The module list used for finding elements could not be created
MSE_XML_RETURN_INVALID_CHANNEL	The channel requested is invalid
MSE_XML_RETURN_INVALID_CHANNEL_LIST	The channel list used for finding elements could not be created
MSE_XML_RETURN_TAGNAME_NOT_FOUND	The tag name could not be found
MSE_XML_DATA_NOT_CHANGED	The data to set is the same as what is already in the file
MSE_XML_RETURN_NULL_POINTER	A NULL pointer was passed in as a parameter
MSE_XML_RETURN_NUM	The number of values in the enumeration

## MSE\_XML\_ELEMENTS

The MSE\_XML\_ELEMENTS enumeration are used for accessing the XML data loaded into memory.

### Enumeration

```

MSE_XML_ELEMENTS
{
    MSE_XML_ELEMENT_BASE_LABEL           = 0,
    MSE_XML_ELEMENT_SERIAL_NUMBER,
    MSE_XML_ELEMENT_MODEL,
    MSE_XML_ELEMENT_LABEL,
    MSE_XML_ELEMENT_MODEL_ID,
    MSE_XML_ELEMENT_HARDWARE_ID,
    MSE_XML_ELEMENT_BL_VERSION,
    MSE_XML_ELEMENT_FW_VERSION,
    MSE_XML_ELEMENT_USING_DHCP,
    MSE_XML_ELEMENT_NETMASK,
    MSE_XML_ELEMENT_NETMASK_STATIC,
    MSE_XML_ELEMENT_MAC,
    MSE_XML_ELEMENT_IP,
    MSE_XML_ELEMENT_IP_STATIC,
    MSE_XML_ELEMENT_PORT,
    MSE_XML_ELEMENT_STATE,
    MSE_XML_ELEMENT_PRIMARY_EXCITATION_VOLTAGE,
    MSE_XML_ELEMENT_PRIMARY_EXCITATION_FREQUENCY,
    MSE_XML_ELEMENT_LAST_MODULE_ENUM,
    MSE_XML_ELEMENT_CHANNEL_LABEL = 100,
    MSE_XML_ELEMENT_ERROR_MONITORING,
    MSE_XML_ELEMENT_POPULATED,
    MSE_XML_ELEMENT_DEVICE_TYPE,
    MSE_XML_ELEMENT_ROTARY_FORMAT_TYPE,
    MSE_XML_ELEMENT_UOM,
    MSE_XML_ELEMENT_ERROR_COMPENSATION,
    MSE_XML_ELEMENT_SCALE_FACTOR,
    MSE_XML_ELEMENT_MASTERING_ENABLED,
    MSE_XML_ELEMENT_MASTER_DESIRED,
    MSE_XML_ELEMENT_MASTER_OFFSET,
    MSE_XML_ELEMENT_MASTER_UOM,
    MSE_XML_ELEMENT_DISPLAY_RESOLUTION,
    MSE_XML_ELEMENT_RESOLUTION,
    MSE_XML_ELEMENT_OFFSET,
    MSE_XML_ELEMENT_DISTINGUISHABLE_REVOLUTIONS,
    MSE_XML_ELEMENT_ENCODER_NAME,
    MSE_XML_ELEMENT_ENCODER_ID,
    MSE_XML_ELEMENT_ENCODER_SERIAL_NUMBER,
    MSE_XML_ELEMENT_LINE_COUNT,
    MSE_XML_ELEMENT_SIGNAL_PERIOD,
    MSE_XML_ELEMENT_COUNTING_DIRECTION,
    MSE_XML_ELEMENT_REFERENCE_MARK,
    MSE_XML_ELEMENT_UNUSED_1,
    MSE_XML_ELEMENT_INSTRUMENTATION_RANGE_MIN,
    MSE_XML_ELEMENT_INSTRUMENTATION_RANGE_MAX,
    MSE_XML_ELEMENT_ACTUAL_RANGE_MIN,
    MSE_XML_ELEMENT_ACTUAL_RANGE_MAX,
    MSE_XML_ELEMENT_CALIBRATION_TIMESTAMP,
    MSE_XML_ELEMENT_RECALIBRATION_TIMER,
    MSE_XML_ELEMENT_GAIN_CODE,
    MSE_XML_ELEMENT_INTERPOLATION,
    MSE_XML_ELEMENT_SIGNAL_TYPE,
    MSE_XML_ELEMENT_LAST_CHANNEL_ENUM
};

```

## Parameters

MSE_XML_ELEMENT_BASE_LABEL	The base label
MSE_XML_ELEMENT_SERIAL_NUMBER	Used for the Module serial numbers
MSE_XML_ELEMENT_MODEL	The model number of the module
MSE_XML_ELEMENT_LABEL	The customizable label of the module
MSE_XML_ELEMENT_MODEL_ID	The model ID of the module
MSE_XML_ELEMENT_HARDWARE_ID	The hardware ID of the module
MSE_XML_ELEMENT_BL_VERSION	The bootloader version of the module
MSE_XML_ELEMENT_FW_VERSION	The firmware version of the module
MSE_XML_ELEMENT_USING_DHCP	Whether the module is using DHCP
MSE_XML_ELEMENT_NETMASK	The netmask of the module
MSE_XML_ELEMENT_NETMASK_STATIC	The netmask of the module if DHCP cannot be obtained
MSE_XML_ELEMENT_MAC	The MAC address of the module
MSE_XML_ELEMENT_IP	The IP address of the module
MSE_XML_ELEMENT_IP_STATIC	The IP address of the module if DHCP cannot be obtained
MSE_XML_ELEMENT_PORT	The port used by the module
MSE_XML_ELEMENT_STATE	Whether the module is inactive (0), active (1), or bootloader (2)
MSE_XML_ELEMENT_PRIMARY_EXCITATION_VOLTAGE	The primary excitation voltage for an LVDT module
MSE_XML_ELEMENT_PRIMARY_EXCITATION_FREQUENCY	The primary excitation frequency for an LVDT module
MSE_XML_ELEMENT_LAST_MODULE_ENUM	The last module enumeration is for ease of use when indexing this enumerated list
MSE_XML_ELEMENT_CHANNEL_LABEL	The customizable label of the channel
MSE_XML_ELEMENT_ERROR_MONITORING	If error monitoring should be performed in the firmware for a specific channel. Currently on supported for EnDat and 1Vpp.
MSE_XML_ELEMENT_POPULATED	If the channel has an device connected. Currently only used for EnDat and 1Vpp.
MSE_XML_ELEMENT_DEVICE_TYPE	The type of device for a specific channel. The device types are none, Linear, Gauge, and Rotary for EnDat and 1Vpp.
MSE_XML_ELEMENT_ROTARY_FORMAT_TYPE	The type of format to show rotary positioning in for a specific channel
MSE_XML_ELEMENT_UOM	The unit of measurement to display position data in for a specific channel
MSE_XML_ELEMENT_ERROR_COMPENSATION	The multiplier to use for linear error correction for a specific channel
MSE_XML_ELEMENT_SCALE_FACTOR	The scale factor to use for a specific channel
MSE_XML_ELEMENT_MASTERING_ENABLED	Whether mastering is enabled in the MSEsetup for a specific channel
MSE_XML_ELEMENT_MASTER_DESIRED	The desired master position in the MSEsetup for a specific channel
MSE_XML_ELEMENT_MASTER_OFFSET	The computed master offset in the MSEsetup for a specific channel
MSE_XML_ELEMENT_MASTER_UOM	The unit of measurement used when creating the master offset in the MSEsetup for a specific channel
MSE_XML_ELEMENT_DISPLAY_RESOLUTION	The display resolution in the MSEsetup for a specific channel
MSE_XML_ELEMENT_RESOLUTION	The resolution for a specific channel. Used for analog and LVDT devices to scale the raw signal value of the device into the desired units. (currently not supported)
MSE_XML_ELEMENT_OFFSET	The offset to apply for a specific channel. Used for analog modules. (currently not supported)
MSE_XML_ELEMENT_DISTINGUISHABLE_REVOLUTIONS	The number of revolutions for a rotary encoder for a specific channel
MSE_XML_ELEMENT_ENCODER_NAME	The name of an EnDat encoder.
MSE_XML_ELEMENT_ENCODER_ID	The ID of an EnDat encoder.
MSE_XML_ELEMENT_ENCODER_SERIAL_NUMBER	The serial number of an EnDat encoder.
MSE_XML_ELEMENT_LINE_COUNT	The line count for a rotary encoder for a specific channel. Used for 1Vpp encoders.

MSE_XML_ELEMENT_SIGNAL_PERIOD	The signal period for a linear encoder for a specific channel. Used for 1Vpp encoders.
MSE_XML_ELEMENT_COUNTING_DIRECTION	The direction for the encoder for a specific channel
MSE_XML_ELEMENT_REFERENCE_MARK	The reference mark type for the encoder for a specific channel. Used for 1Vpp encoders.
MSE_XML_ELEMENT_UNUSED_1	Unused
MSE_XML_ELEMENT_INSTRUMENTATION_RANGE_MIN	The minimum value that a device can utilize normalized to the UOM
MSE_XML_ELEMENT_INSTRUMENTATION_RANGE_MAX	The maximum value that a device can utilize normalized to the UOM
MSE_XML_ELEMENT_ACTUAL_RANGE_MIN	The minimum value that a device can utilize in raw form (mA or V for Analog module, V for LVDT module)
MSE_XML_ELEMENT_ACTUAL_RANGE_MAX	The maximum value that a device can utilize in raw form (mA or V for Analog module, V for LVDT module)
MSE_XML_ELEMENT_CALIBRATION_TIMESTAMP	The time that the calibration for an Analog or LVDT channel was performed
MSE_XML_ELEMENT_RECALIBRATION_TIMER	The number of hours before a recalibration is needed
MSE_XML_ELEMENT_GAIN_CODE	The gain code used for calibrating an LVDT sensor's output value
MSE_XML_ELEMENT_INTERPOLATION	The interpolation used for a TTL encoder
MSE_XML_ELEMENT_SIGNAL_TYPE	The type of signal that the encoder uses. This will be equal to the enumeration value of SIGNAL_TYPE and can be SIGNAL_TYPE_1VPP or SIGNAL_TYPE_11UAPP and is only used for the 1 Vpp module.
MSE_XML_ELEMENT_LAST_CHANNEL_ENUM	The number of values in the enumeration

## Enumerations **PROGRAMMING\_STATE\_ENUMS**

The PROGRAMMING\_STATE\_ENUMS enumeration is used for the state of the firmware and bootloader updates. The programming state can be queried while the program() method is running by calling the getProgrammingState() method from another thread.

### Enumeration

```
PROGRAMMING_STATE_ENUMS
{
    PROGRAMMING_STATE_IDLE           = 0,
    PROGRAMMING_STATE_INITIALIZING,
    PROGRAMMING_STATE_DOWNLOADING,
    PROGRAMMING_STATE_REBOOTING,
    PROGRAMMING_STATE_FINISHED,
    PROGRAMMING_STATE_FAILED,
    PROGRAMMING_STATE_COUNT
};
```

### Parameters

PROGRAMMING_STATE_IDLE	Programming has not been initiated
PROGRAMMING_STATE_INITIALIZING	Initializing the programming data
PROGRAMMING_STATE_DOWNLOADING	Downloading to the module
PROGRAMMING_STATE_REBOOTING	Rebooting the module
PROGRAMMING_STATE_FINISHED	Finished programming
PROGRAMMING_STATE_FAILED	Failed programming
PROGRAMMING_STATE_COUNT	The number of values in the enumeration



## UdpCmdType

The UdpCmdType enumeration is used internally for communication to a module. A subset of the enumeration is used to determine which asynchronous message was received from a module.

### Enumeration

```
UdpCmdType
{
    UDP_OPEN                = 0,
    UDP_GET_COUNTS,
    UDP_SET_OUTPUT,
    UDP_RESET,
    UDP_CONFIG_IP,
    UDP_CONFIG_DHCP,
    UDP_LOAD_FAIL,
    UDP_GET_CONFIG,
    UDP_GET_LEFT,
    UDP_SET_RIGHT,
    UDP_MOD_TYPE,
    UDP_ENDAT_PARAMS,
    UDP_ENDAT_INFO,
    UDP_CONFIG_PORTS,
    UDP_UNUSED_2,
    UDP_CHANNEL_CONFIG,
    UDP_CHANNEL_PRESENCE,
    UDP_UNKNOWN_CMD,
    UDP_CONNECT,
    UDP_FNET_ERROR,
    UDP_RELOAD_CODE,
    UDP_UNUSED_5,
    UDP_LATCH,
    UDP_ADC,
    UDP_FRAM_DATA,
    UDP_ASYNC,
    UDP_INTEGRITY,
    UDP_ANALOG_DIAGS,
    UDP_INIT_BIN,
    UDP_LOAD_BIN,
    UDP_FINISH_BIN,
    UDP_SET_MODE,
    UDP_G50_DATA,
    UDP_CHANNEL_STATUS,
    UDP_RESTORE_FACTORY,
    UDP_COUNT
};
```

## Parameters

UDP_OPEN	Internal use only
UDP_GET_COUNTS	Internal use only
UDP_SET_OUTPUT	Internal use only
UDP_RESET	Internal use only
UDP_CONFIG_IP	Internal use only
UDP_CONFIG_DHCP	Internal use only
UDP_LOAD_FAIL	Internal use only
UDP_GET_CONFIG	Internal use only
UDP_GET_LEFT	Internal use only
UDP_SET_RIGHT	Internal use only
UDP_MOD_TYPE	Internal use only
UDP_ENDAT_PARAMS	Internal use only
UDP_ENDAT_INFO	Internal use only
UDP_CONFIG_PORTS	Internal use only
UDP_UNUSED_2	Internal use only
UDP_CHANNEL_CONFIG	Internal use only
UDP_CHANNEL_PRESENCE	Internal use only
UDP_UNKNOWN_CMD	Internal use only
UDP_CONNECT	Used by a client to receive asynchronous updates from a module informing of its networking settings
UDP_FNET_ERROR	Used by the firmware to tell the client that an error occurred when reading a message from the socket
UDP_RELOAD_CODE	Internal use only
UDP_UNUSED_5	Internal use only
UDP_LATCH	Used by a client to receive asynchronous updates from a module informing of footswitch presses
UDP_ADC	Internal use only
UDP_FRAM_DATA	Internal use only
UDP_ASYNC	Internal use only
UDP_INTEGRITY	Used by a client to receive asynchronous updates from a module informing of module warnings and errors
UDP_ANALOG_DIAGS	Internal use only
UDP_INIT_BIN	Internal use only
UDP_LOAD_BIN	Internal use only
UDP_FINISH_BIN	Internal use only
UDP_SET_MODE	Internal use only
UDP_G50_DATA	Internal use only
UDP_CHANNEL_STATUS	Used by a client to receive asynchronous updates from a module informing of channel warnings, errors, or referencing completed
UDP_RESTORE_FACTORY	Internal use only
UDP_COUNT	The number of values in the enumeration

## LVDT\_UOM

The LVDT\_UOM enumeration is used for the units of measurement that are allowed for an LVDT sensor. Currently only millimeters is supported.

### Enumeration

```
LVDT_UOM
{
    LVDT_UOM_UNDEFINED = 0,
    LVDT_UOM_INCHES,
    LVDT_UOM_MM
};
```

### Parameters

LVDT_UOM_UNDEFINED	The UOM is undefined
LVDT_UOM_INCHES	Inches
LVDT_UOM_MM	Millimeters

## LVDT\_UPDATE\_CHOICES

The LVDT\_UPDATE\_CHOICES enumeration is used for choosing the desired group of sensors to read in the module. This is useful for diagnostic purposes to isolate specific counts. This enumeration is used by the setDiagnosticsEnabled method.

### Enumeration

```
LVDT_UPDATE_CHOICES
{
    LVDT_UPDATE_GROUP_1 = 0,
    LVDT_UPDATE_GROUP_2,
    LVDT_UPDATE_GROUP_3,
    LVDT_UPDATE_GROUP_4,
    LVDT_UPDATE_EXCITATION_VOLTAGE,
    LVDT_UPDATE_CHOICE_ALL
};
```

### Parameters

LVDT_UPDATE_GROUP_1	The output counts for sensors connected to channel 1 and 2
LVDT_UPDATE_GROUP_2	The output counts for sensors connected to channel 3 and 4
LVDT_UPDATE_GROUP_3	The output counts for sensors connected to channel 5 and 6
LVDT_UPDATE_GROUP_4	The output counts for sensors connected to channel 7 and 8
LVDT_UPDATE_EXCITATION_VOLTAGE	Unused
LVDT_UPDATE_CHOICE_ALL	The output counts for all sensors are read

## LVDT\_OVERSAMPLING\_CHOICES

The LVDT\_OVERSAMPLING\_CHOICES enumeration is used for choosing the desired oversampling of the ADC to use in the module. The ADC is used internally to obtain the count reading of an LVDT sensor. The more oversampling that is performed will take longer to obtain the reading but will also filter out more noise.

### Enumeration

```
LVDT_OVERSAMPLING_CHOICES
{
    LVDT_OVERSAMPLING_MINIMUM = 0x10,
    LVDT_OVERSAMPLING_MEDIUM = 0x18,
    LVDT_OVERSAMPLING_MAXMUM = 0x20
};
```

### Parameters

LVDT_OVERSAMPLING_MINIMUM	Obtains a reading from the ADC approximately once every ms per group of 2 sensors. This will take approximately 4 ms to read from all 4 sensors.
LVDT_OVERSAMPLING_MEDIUM	Obtains a reading from the ADC approximately once every 2 ms per group of 2 sensors. This will take approximately 8 ms to read from all 4 sensors.
LVDT_OVERSAMPLING_MAXIMUM	Obtains a reading from the ADC approximately once every 4 ms per group of 2 sensors. This will take approximately 16 ms to read from all 4 sensors.

## Enumerations **ANALOG\_DIAG\_VOLTAGES\_ENUM**

The ANALOG\_DIAG\_VOLTAGES\_ENUM enumeration is used internally to index into the voltages read from the getDiag-Voltages method.

### Enumeration

```
ANALOG_DIAG_VOLTAGES_ENUM
{
    ANALOG_DIAG_VOLTAGE = 0,
    ANALOG_DIAG_CURRENT,
    ANALOG_DIAG_5V_1,
    ANALOG_DIAG_GROUND,
    ANALOG_DIAG_5V_2,
    ANALOG_DIAG_VREF,
    NUM_ANALOG_DIAG_VOLTAGES
};
```

### Parameters

ANALOG_DIAG_VOLTAGE	The voltages reading from a channel
ANALOG_DIAG_CURRENT	The current reading from a channel
ANALOG_DIAG_5V_1	The 5 volt reading from a channel
ANALOG_DIAG_GROUND	The ground reading from a channel
ANALOG_DIAG_5V_2	The second 5 volt reading from a channel
ANALOG_DIAG_VREF	The reference voltage reading from a channel
NUM_ANALOG_DIAG_VOLTAGES	The number of reading returned for each channel

## TTL\_INTERPOLATION

The TTL\_INTERPOLATION enumeration is used to select the type of interpolation to use for a TTL encoder when the setSignalPeriod or setLineCount methods are called.

### Enumeration

```
TTL_INTERPOLATION
{
    TTL_INTERPOLATION_X1 = 1,
    TTL_INTERPOLATION_X2 = 2,
    TTL_INTERPOLATION_X5 = 5,
    TTL_INTERPOLATION_X10 = 10,
    TTL_INTERPOLATION_X20 = 20,
    TTL_INTERPOLATION_X25 = 25,
    TTL_INTERPOLATION_X50 = 50,
    TTL_INTERPOLATION_X100 = 100,
    TTL_INTERPOLATION_X200 = 200
};
```

### Parameters

TTL_INTERPOLATION_X1	No interpolation is needed
TTL_INTERPOLATION_X2	Uses a 2x multiplier
TTL_INTERPOLATION_X5	Uses a 5x multiplier
TTL_INTERPOLATION_X10	Uses a 10x multiplier
TTL_INTERPOLATION_X20	Uses a 20x multiplier
TTL_INTERPOLATION_X25	Uses a 25x multiplier
TTL_INTERPOLATION_X50	Uses a 50x multiplier
TTL_INTERPOLATION_X100	Uses a 100x multiplier
TTL_INTERPOLATION_X200	Uses a 200x multiplier

## SIGNAL\_TYPE

The SIGNAL\_TYPE enumeration is used to show the type of signal used by an encoder. See the getSignalType, setSignalType, and detectSignalType methods of the 1 Vpp module.

### Enumeration

```
SIGNAL_TYPE
{
    SIGNAL_TYPE_1VPP = 0,
    SIGNAL_TYPE_11UAPP,
    SIGNAL_TYPE_TTL,
    SIGNAL_TYPE_UNKNOWN = 100
};
```

### Parameters

SIGNAL_TYPE_1VPP	The encoder has a 1 Vpp signal
SIGNAL_TYPE_11UAPP	The encoder has a 11 $\mu$ App signal
SIGNAL_TYPE_TTL	The encoder has a TTL signal. Currently not used.
SIGNAL_TYPE_UNKNOWN	The encoder has an unknown signal.

### ModuleData

The ModuleData class holds all of the significant information regarding a specific module. This class can be accessed by the getModuleData method of the MseModule class.

#### Class

```
class ModuleData
{
public:
MODULE_ID                type;
unsigned long            hwId;
unsigned char            hwRev;
unsigned short           numChannels;
bool                     isFirst;
bool                     isInputConnected;
char                     ipAddress[SIZE_IP_ADDRESS];
unsigned short           port;
char                     macAddress[SIZE_MAC_ADDRESS];
char                     bootloaderVersion[SIZE_BUILD_INFO];
char                     firmwareVersion[SIZE_BUILD_INFO];
bool                     isUsingDhcp;
char                     netmask[SIZE_IP_ADDRESS];
char                     ipAddressStatic[SIZE_IP_ADDRESS];
char                     netmaskStatic[SIZE_IP_ADDRESS];
char                     serialNumber[SIZE_SERIAL_NUMBER];
char                     moduleIndex;
};
```

#### Parameters

type	The type of module
hwId	The ID of the PCBA for the module
hwRev	The revision of the PCBA for the module
numChannels	The number of channels in the module
isFirst	Whether the module is the first one in the chain
isInputConnected	Whether there is input connected (used for ordering after a broadcast)
ipAddress	The IP address returned from the module
port	The port used by the module
macAddress	The MAC address used by the module
bootloaderVersion	The bootloader version
firmwareVersion	The firmware version
isUsingDhcp	Whether or not the module is using DHCP
netmask	The netmask of the module that received the UDP_OPEN command
ipAddressStatic	The static IP address of the module that received the UDP_OPEN command
netmaskStatic	The static netmask of the module that received the UDP_OPEN command
serialNumber	The serial number of the module that received the UDP_OPEN command
moduleIndex	The module index character of the module that received the UDP_OPEN command

## DeviceData

The DeviceData structure is used to hold the encoder information returned from an EnDat module. This class can be accessed with the getDeviceData method of the MseEndatModule class.

### Structure

```
struct DeviceData
{
    char                name[DEVICE_NAME_SIZE + 1];
    char                id[DEVICE_ID_SIZE + 1];
    char                serialNum[SERIAL_NUMBER_SIZE + 1];
    bool                isReversed;
    bool                isRotary;
    unsigned long       resolution;
    unsigned long       measurementLength;
    unsigned long       signalPeriod;
    unsigned short      distinguishableRevolutions;
    unsigned char       positionBits;
};
```

### Parameters

name	The name returned from the encoder (enough space is allocated to include the terminating character)
id	The id returned from the encoder (enough space is allocated to include the terminating character)
serialNum	The serial number returned from the encoder (enough space is allocated to include the terminating character)
isReversed	Whether positive numerical count data refers to motion in the negative or positive direction. A value of true refers to negative direction as being positive count increments
isRotary	Whether the encoder is rotary or linear
resolution	The resolution for the encoder (in counts per nanometer for non-rotary and steps per revolution for rotary)
measurementLength	The measurement length of the encoder
signalPeriod	The signal period of the encoder
distinguishableRevolutions	The number of distinguishable revolutions for a multi-turn rotary encoder. Single turn rotary encoders will be 1.
positionBits	The number of bits used for calculating the position. Used internally by the MSElibrary.

## LeftData

The LeftData structure holds all the information returned from the getLeft() command.

### Structure

```

struct LeftData
{
    char                ipAddress[SIZE_IP_ADDRESS];
    bool                isConnectInSet;
    bool                isFirstModule;
    bool                isLastModule;
    bool                isMiddleModule;
};

```

### Parameters

ipAddress	The IP address of the module
isConnectInSet	Whether the module has GPIO input pins set
isFirstModule	Whether the module is the first in the line
isLastModule	Whether the module is the last in the line
isMiddleModule	Whether the module is the middle in the line

## MSE1000ConnectResponse

The MSE1000ConnectResponse structure holds the MSE Connect response information. This structure is filled in when the response is received for a UDP\_OPEN command

### Structure

```

struct MSE1000ConnectResponse
{
    char                ipAddress[SIZE_IP_ADDRESS];
    unsigned short     port;
    bool                isFirstModule;
    bool                isUsingDhcp;
    char                macAddress[SIZE_MAC_ADDRESS];
    char                netmask[SIZE_IP_ADDRESS];
    char                ipAddressStatic[SIZE_IP_ADDRESS];
    char                netmaskStatic[SIZE_IP_ADDRESS];
    char                serialNumber[SIZE_SERIAL_NUMBER];
};

```

### Parameters

ipAddress	The IP address of the module that received the UDP_OPEN command
port	The UDP port of the module that received the UDP_OPEN command
isFirstModule	Whether or not the module is the first one in the MSE module chain
isUsingDhcp	Whether or not the module is using DHCP
macAddress	The MAC address of the module that received the UDP_OPEN command
netmask	The netmask of the module that received the UDP_OPEN command
ipAddressStatic	The static IP address of the module that received the UDP_OPEN command
netmaskStatic	The static netmask of the module that received the UDP_OPEN command
serialNumber	The serial number of the module that received the UDP_OPEN command



## EncoderInfo

The EncoderInfo structure holds the encoder information for a specific device.

### Structure

```
struct EncoderInfo
{
    ENCODER_TYPES_ENUM    encoderType;
    UOM                    uom;
    unsigned long          resolution;
    bool                   countingDirectionPositive;
    double                 errorCompensation;
};
```

### Parameters

encoderType	The type of encoder attached to the channel
uom	The unit of measurement for the device
resolution	Not used (use the getResolution method instead)
countingDirectionPositive	The direction the counts are traversing
errorCompensation	The multiplier used to scale the counts returned from the device for error compensation

## 2.7 Return values

Most methods will return a `MseResults` data type. The `MseResults` contains the following methods for getting the return code and additional information from the `MseResults` data type. Refer to **4.11 MSE\_RESPONSE\_CODE** for response codes.

The following methods are used only for C++ calls. The C wrappers will return the `MSE_RESPONSE_CODE` only.

### getCode

The `getCode` method returns the response code.

#### Method

```
MSE_RESPONSE_CODE getCode();
```

### getMethod

The `getMethod` method returns the method that failed or the first method called.

#### Method

```
char* getMethod();
```

### getLine

The `getLine` method returns the line that failed.

#### Method

```
unsigned long getLine();
```

### showRespCode

The `showRespCode` method is used to return a string representation of the `MSE_RESPONSE_CODE` enumeration.

#### Method

```
char* showRespCode
(
    MSE_RESPONSE_CODE code
);
```

#### Parameters

`code`                    The `MSE_RESPONSE_CODE` to stringify

#### Example:

```
MseResults retVal;
retVal = module->getCounts(counts, module->getNumChannels(), COUNT_REQUEST_LATEST);
if(RESPONSE_OK != retVal.getCode())
{
    std::stringstream ss;
    ss << "Error: " << MseResults::showRespCode(retVal.getCode())
    cout << ss;
}
```

## 2.8 Constants

### NUM\_MSE1000\_IO\_INPUTS

The NUM\_MSE1000\_IO\_INPUTS constant is the number of inputs on the I/O module.

**Constant**

```
static const unsigned int NUM_MSE1000_IO_INPUTS = 4;
```

### NUM\_MSE1000\_IO\_OUTPUTS

The NUM\_MSE1000\_IO\_OUTPUTS constant is the number of outputs on the I/O module.

**Constant**

```
static const unsigned int NUM_MSE1000_IO_OUTPUTS = 4;
```

### DEVICE\_NAME\_SIZE

The DEVICE\_NAME\_SIZE constant is the size of the device name char array returned in the DeviceData structure not including a terminating character.

**Constant**

```
const int DEVICE_NAME_SIZE = 9;
```

### DEVICE\_ID\_SIZE

The DEVICE\_ID\_SIZE constant is the size of the device ID char array returned in the DeviceData structure (not counting a terminating character).

**Constant**

```
const int DEVICE_ID_SIZE = 10;
```

### SERIAL\_NUMBER\_SIZE

The SERIAL\_NUMBER\_SIZE constant is the size of the encoder serial number char array returned in the DeviceData structure not including a terminating character.

**Constant**

```
const int SERIAL_NUMBER_SIZE = 16;
```

## SIZE\_IP\_ADDRESS

The SIZE\_IP\_ADDRESS constant is the size of the IP address string including decimal notation and terminator.

### Constant

```
const short SIZE_IP_ADDRESS = 18;
```

## SIZE\_MAC\_ADDRESS

The SIZE\_MAC\_ADDRESS constant is the size of the MAC address string including colons and terminator.

### Constant

```
const short SIZE_MAC_ADDRESS = 19;
```

## SIZE\_BUILD\_INFO

The SIZE\_BUILD\_INFO constant is the size of the Maximum build info string including terminator.

### Constant

```
const short SIZE_BUILD_INFO = 32;
```

## SIZE\_SERIAL\_NUMBER

The SIZE\_SERIAL\_NUMBER constant is the size of the module serial number (ASCII characters) + 1 byte for the terminator and 2 bytes for the spaces.

### Constant

```
const short SIZE_SERIAL_NUMBER = 11;
```

## MAX\_NUM\_MODULES

The MAX\_NUM\_MODULES constant is the maximum number of modules that can be connected to a MSE.

### Constant

```
const int MAX_NUM_MODULES = 64;
```

## MAX\_CHANNELS\_PER\_MODULE

The MAX\_CHANNELS\_PER\_MODULE constant is the maximum number of channels that can be connected to each module.

### Constant

```
const short MAX_CHANNELS_PER_MODULE = 8;
```

## MSE1000\_PORT

The MSE1000\_PORT constant is the port used by the MSE module for UDP communication.

### Constant

```
const int MSE1000_PORT = 27015;
```

## MSE1000\_CLIENT\_DEFAULT\_PORT

The MSE1000\_CLIENT\_DEFAULT\_PORT constant is the port used by the client PC for UDP communication. The client PC can use a different port if there is another application installed that is using the same port.

### Constant

```
const int PC_PORT = 27016;
```

## MSE1000\_ASYNC\_PORT

The MSE1000\_ASYNC\_PORT const is the port used by the MSE to send asynchronous messages.

### Constant

```
const int MSE1000_ASYNC_PORT = 27300;
```

## NUM\_INTEGRITY\_RANGES

The number of integrity range values that are returned when requesting the integrity value.

### Constant

```
const short NUM_INTEGRITY_RANGES = 14;
```

## NUM\_LATCH\_TYPES

The number of latch types available. The latch types are located in the LATCH\_CHOICE enumeration and consist of the 3 software latches and the two footswitches.

### Constant

```
const unsigned short NUM_LATCH_TYPES = 5;
```

## COUNTS\_PER\_LINE

The number of counts per each line of an analog rotary encoder

### Constant

```
const int COUNTS_PER_LINE = 4;
```

## INTERPOLATION\_VALUE

There are 12 bits used for interpolation of the analog encoders, which equates to a value of 1024

### Constant

```
const int INTERPOLATION_VALUE = 1024;
```

## NUM\_LVDT\_CHANNELS

The NUM\_LVDT\_CHANNELS constant is the maximum number of channels available for an LVDT module.

### Constant

```
const int NUM_LVDT_CHANNELS = 8;
```

### LVDT\_EXCITATION\_VOLTAGE\_MIN\_VPP

The LVDT\_EXCITATION\_VOLTAGE\_MIN\_VPP constant is the minimum voltage allowed for the setExcitationVoltage method.

#### Constant

```
const double LVDT_EXCITATION_VOLTAGE_MIN_VPP = 1.5;
```

### LVDT\_EXCITATION\_VOLTAGE\_MAX\_VPP

The LVDT\_EXCITATION\_VOLTAGE\_MAX\_VPP constant is the maximum voltage allowed for the setExcitationVoltage method.

#### Constant

```
const double LVDT_EXCITATION_VOLTAGE_MAX_VPP = 5.5;
```

### LVDT\_EXCITATION\_FREQUENCY\_MIN\_KHZ

The LVDT\_EXCITATION\_FREQUENCY\_MIN\_KHZ constant is the minimum frequency allowed for the setExcitationFrequency method.

#### Constant

```
const double LVDT_EXCITATION_FREQUENCY_MIN_KHZ = 3.0;
```

### LVDT\_EXCITATION\_FREQUENCY\_MAX\_KHZ

The LVDT\_EXCITATION\_FREQUENCY\_MAX\_KHZ constant is the maximum frequency allowed for the setExcitationFrequency method.

#### Constant

```
const double LVDT_EXCITATION_FREQUENCY_MAX_KHZ = 50.0;
```

### NUM\_MSE1000\_ANALOG\_CHANNELS

The NUM\_MSE1000\_ANALOG\_CHANNELS constant is the maximum number of channels available for an analog module.

#### Constant

```
const int NUM_MSE1000_ANALOG_CHANNELS= 2;
```

### NUM\_MSE1000\_ANALOG\_VALUES\_PER\_CHANNEL

The NUM\_MSE1000\_ANALOG\_VALUES\_PER\_CHANNEL constant is the number of data values returned for each channel of the analog module.

#### Constant

```
const int NUM_MSE1000_ANALOG_VALUES_PER_CHANNEL = 2;
```

### MAX\_NUM\_ANALOG\_AVG\_SAMPLES

The MAX\_NUM\_ANALOG\_AVG\_SAMPLES constant is the maximum number of samples that can be used when computing the average voltage and current in the analog module.

#### Constant

```
const int MAX_NUM_ANALOG_AVG_SAMPLES = 100;
```

## 2.9 Interface methods

The interface methods are available for C++ only. Users of the C wrappers must create their own module chain by creating instances of the modules and initializing them.

### MseInterface

#### Constructor

```
MseInterface();
```

### addModule

The addModule method will create an instance of a new module of the type requested. The module IP must be passed in because each module in the MSE has a different IP address. The new module will be added to the end of the moduleChain\_.

#### Method

```
MseResults addModule
(
    const MODULE_ID    moduleType,
    const char*        moduleIp,
    bool               useAsync
);
```

#### Parameters

moduleType	The type of module to create
moduleIp	The IP address of the module to connect to
useAsync	True if the MSE should send asynchronous messages to the MSE1000_ASYNC_PORT

#### Return value

The return value delivers a status for the method call.

MseResults            A response code representing whether the method succeeded

### removeConnections

The removeConnections method empties the UDP client list used internally by the MSE library.

#### Method

```
void removeConnections();
```

## createChain

The createChain method will perform a broadcast and wait for all of the responses from all of the modules. It will then create an instance of a MseModule for each response and add it to the moduleChain\_ array. The modules will then be re-ordered based on their location in the chain. The modules can then be retrieved with the getModule(), getEndatModule(), getIoModule(), get1VppModule(), or getPneumaticModule() methods. This method may take up to two minutes to complete based on the number of modules in the chain and whether DHCP is enabled or not.

### Method

```
MseResults createChain
(
    const char*      clientIp,
    unsigned short  clientPort,
    bool            useAsync,
    const char*     broadcastNetmask
);
```

### Parameters

clientIp	The IP address for the MSE to respond to for broadcast requests.
clientPort	The port for the MSE to respond to for broadcast requests.
useAsync	True if the MSE should send asynchronous messages to the MSE1000_ASYNC_PORT
broadcastNetmask	The netmask to use to create the broadcast address.

### Return value

MseResults                    A response code representing whether createChain was successful.

## getChainCreationState

The getChainCreationState method is used to get the state of the chain creation initiated by the createChain method. The state represents whether the chain creation is currently idle, broadcasting, ordering the chain, finished, or if it had an error. The error can be determined by the response code of the createChain method.

### Method

```
MSE_CHAIN_CREATION_STATE getChainCreationState();
```

### Return value

The return value delivers a MSE\_CHAIN\_CREATION\_STATE enumeration representing the state of the chain creation.

## getNumModules

The getNumModules method is used to get the number of modules in the chain.

### Method

```
unsigned short getNumModules();
```

### Return value

The return value delivers an unsigned short representing the number of modules in the chain.

## getModule

The getModule method will return the requested module as a base MseModule. It is up to the caller to downcast the module based on the type if needed. This method allows flexibility in that it can be called for any module and the logic can be performed by the client code.

```
MseModule* getModule
(
    const unsigned short  moduleNumber
);
```

### Return value

The return value delivers a pointer to the MseModule. A NULL pointer will be returned if the module is not in the chain.



## getDeviceModule

The `getDeviceModule` method will return the requested module as a base `MseDeviceModule`. This method should be used if the complexity in downcasting from `getModule()` is unnecessary. If the module requested is not of type `MseDeviceModule`, a NULL pointer will be returned.

### Method

```
MseDeviceModule* getDeviceModule
(
    const unsigned short  moduleNumber
);
```

### Return value

The return value delivers a pointer to a `MseDeviceModule`. A NULL pointer will be returned if the module is not in the chain.

## getEndatModule

The `getEndatModule` method will return the requested module as a `MseEndatModule`. This method should be used if the complexity in downcasting from `getModule()` is unnecessary. If the module requested is not of type `MseEndatModule`, a NULL pointer will be returned.

### Method

```
MseEndatModule* getEndatModule
(
    const unsigned short  moduleNumber
);
```

### Return value

The return value delivers a pointer to a `MseEndatModule`. A NULL pointer will be returned if the module is not in the chain or if the module requested is not a `MseEndatModule`.

## getIoModule

The `getIoModule` method will return the requested module as a `MseIoModule`. This method should be used if the complexity in downcasting from `getModule()` is unnecessary. If the module requested is not of type `MseIoModule`, a NULL pointer will be returned.

### Method

```
MseIoModule* getIoModule
(
    const unsigned short  moduleNumber
);
```

### Return value

The return value delivers a pointer to a `MseIoModule`. A NULL pointer will be returned if the module is not in the chain or if the module requested is not a `MseIoModule`.

## get1VppModule

The `get1VppModule` method will return the requested module as a `Mse1VppModule`. This method should be used if the complexity in downcasting from `getModule()` is unnecessary. If the module requested is not of type `Mse1VppModule`, a NULL pointer will be returned.

### Method

```
Mse1VppModule* get1VppModule
(
    const unsigned short  moduleNumber
);
```

### Return value

The return value delivers a pointer to a `Mse1VppModule`. A NULL pointer will be returned if the module is not in the chain or if the module requested is not a `Mse1VppModule`.

## getPneumaticModule

The getPneumaticModule method will return the requested module as a MsePneumaticModule. This method should be used if the complexity in downcasting from getModule() is unnecessary. If the module requested is not of type MsePneumaticModule, a NULL pointer will be returned.

### Method

```
MsePneumaticModule* getPneumaticModule
(
    const unsigned short  moduleNumber
);
```

### Return value

The return value delivers a pointer to a MsePneumaticModule. A NULL pointer will be returned if the module is not in the chain or if the module requested is not a MsePneumaticModule.

## getAnalogModule

The getAnalogModule method will return the requested module as a MseAnalogModule. This method should be used if the complexity in downcasting from getModule() is unnecessary. If the module requested is not of type MseAnalogModule, a NULL pointer will be returned.

### Method

```
MseAnalogModule* getAnalogModule
(
    const unsigned short  moduleNumber
);
```

### Return value

The return value delivers a pointer to a MseAnalogModule. A NULL pointer will be returned if the module is not in the chain or if the module requested is not a MseAnalogModule.

## getLvdtModule

The getLvdtModule method will return the requested module as a MseLvdtModule. This method should be used if the complexity in downcasting from getModule() is unnecessary. If the module requested is not of type MseLvdtModule, a NULL pointer will be returned.

### Method

```
MseLvdtModule* getLvdtModule
(
    const unsigned short  moduleNumber
);
```

### Return value

The return value delivers a pointer to a MseLvdtModule. A NULL pointer will be returned if the module is not in the chain or if the module requested is not a MseLvdtModule.

## getTtlModule

The getTtlModule method will return the requested module as a MseTtlModule. This method should be used if the complexity in downcasting from getModule() is unnecessary. If the module requested is not of type MseTtlModule, a NULL pointer will be returned.

### Method

```
MseTtlModule* getTtlModule
(
    const unsigned short moduleNumber
);
```

### Return value

The return value delivers a pointer to a MseTtlModule. A NULL pointer will be returned if the module is not in the chain or if the module requested is not a MseTtlModule.

## 2.10 General methods and functions

The general methods and functions are provided for common functionality across all module types. The C++ methods and the C functions are separated into two sections for easier lookup.

### C++ methods

#### MseModule

The MseModule constructor instantiates and initializes a ModuleData structure and instantiates a MseProgramming object for programming the modules.

#### Constructor

```
MseModule
    (
        void
    );
```

#### initializeModule

The initializeModule method will configure the UDP messaging and fill in the moduleData\_ structure with all the information known from the module and it's devices. The moduleData\_ is the private object representing the ModuleData class.

#### Method

```
virtual MseResults initializeModule
    (
        const char*      mseIpAddress
        bool              useAsync
    );
```

#### Parameters

mseIpAddress	The MSE Ip address
useAsync	True if the MSE should send asynchronous messages to the MSE1000_ASYNC_PORT

#### Return value

The return value delivers a response code representing whether the initialization information was retrieved correctly.

#### initializeFirmware

The initializeFirmware method will open a connection for use in communicating to a module that is running out of the bootloader. It differs from initializeModule in that it does not try to initialize module data because the information cannot be obtained unless the firmware is loaded.

#### Method

```
virtual MseResults initializeFirmware
    (
        const char*      mseIpAddress
    );
```

#### Parameters

mseIpAddress	The MSE Ip address
--------------	--------------------

#### Return value

The return value delivers a response code representing whether the initialization of the firmware succeeded.

**getModuleType**

The getModuleType method returns the module type information. This information is requested from the MSE and contains the module type and number of axes on the module.

**Method**

```
MseResults getModuleType
(
    MODULE_ID*      moduleType,
    unsigned long*  hwId,
    unsigned char*  hwRev,
    unsigned short* numAxes
);
```

**Parameters**

moduleType	The type of module
hwId	The hardware ID of the PCBA
hwRev	The revision of the PCBA
numAxes	The number of axes

**Return value**

The return value delivers a response code representing whether the getModuleType command was sent.

**getConfig**

The getConfig method gets the configuration of the MSE. The configuration information consists of the IP address, input connection status, bootloader version, and firmware version. The bootloader and firmware versions are also stored in the moduleData\_ structure.

**Method**

```
MseResults getConfig
(
    char*      ipAddress,
    bool*      isInputConnected,
    char*      blVersion,
    char*      fwVersion
);
```

**Parameters**

ipAddress	The IP address of the module
isInputConnected	Whether the input is connected
blVersion	The bootloader version of the module
fwVersion	The firmware version of the module

**Return value**

The return value delivers a response code representing whether the config was read correctly.

**getNumChannels**

The getNumChannels method returns the number of channels in the module.

**Method**

```
unsigned short getNumChannels();
```

**Return value**

The return value delivers an unsigned char representing the number of channels.

**getModuleData**

The getModuleData method returns the ModuleData information. The ModuleData information is filled in when the initializeModule method is called.

**Method**

```
ModuleData getModuleData();
```

**Return value**

ModuleData                      A structure containing the module data information

**getLeft**

The getLeft method is used to get the module location settings. The MSE may have multiple modules connected together and this command can help determine the location of current one.

**Method**

```
MseResults getLeft
(
    LeftData*                      leftData
);
```

**Parameters**

leftData                              The LeftData structure that is filled in with the values returned from the MSE.

**Return value**

The return value delivers a response code representing whether the getLeft command was sent.

**getCounts**

The getCounts method returns the counts of the measurement devices. If the option is set to COUNT\_REQUEST\_LATCHED, the module will clear the latch after the position is read to allow for faster subsequent latching. This method returns just the raw counts from the module.

This method should only be used for the 1Vpp module since the format of the counts is different for each module.

The EnDat module's getCounts method should be used in order to get counts that separate the revolution bits from the position bits, since the raw count value combines these.

The TTL's getCounts method differs since it only utilizes 4 bytes, whereas the 1 Vpp and EnDat utilize 8.

The LVDT uses the getVoltage method which will return the count value in addition to a voltage that represents an approximate voltage of the ADC.

**Method**

```
MseResults getCounts
(
    unsigned long*    counts,
    unsigned short    numChannels,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

counts	The counts returned from the devices
numChannels	The number of channels to read into the counts parameter
option	Whether to retrieve live or latched counts

**Return value**

The return value delivers a response code representing whether the getCounts was retrieved correctly.

**setRotaryFormat**

The setRotaryFormat method is used to set the rotary format that will be applied to the position calculated from the counts in the derived class's getPositions method. This method is used by the EnDat, 1Vpp and TTL modules.

**Method**

```
void setRotaryFormat
(
    const unsigned short& channel,
    const ROTARY_FORMAT& format
);
```

**Parameters**

channel	The channel of the encoder to apply the rotary format to
format	The ROTARY_FORMAT to apply

**getRotaryFormat**

The getRotaryFormat method is used to return the rotary format that will be applied to the position calculated from the counts in the derived class's getPositions method. This method is used by the EnDat, 1Vpp and TTL modules.

**Method**

```
ROTARY_FORMAT getRotaryFormat
(
    const unsigned short& channel
);
```

**Parameters**

channel	The channel of the encoder that the rotary format will be applied to
---------	--

**Return value**

ROTARY_FORMAT	The rotary format that will be applied to the position
---------------	--

**setDeviceOffset**

The setDeviceOffset method is used to set an offset that will be applied to the position calculated from the counts in the derived class's getPositions method. The offset is useful for applying a master position for an encoder. The offset is applied before the rotary formatting and is in the user units set for the channel. This method is used by the EnDat, 1Vpp, TTL, Analog, and LVDT modules.

**Method**

```
void setDeviceOffset
(
    const unsigned short& channel,
    const double& offset
);
```

**Parameters**

channel	The channel of the device to apply the offset to
offset	The offset to apply

**getDeviceOffset**

The getDeviceOffset method is used to return the offset that will be applied to the position calculated from the counts in the derived class's getPositions method. This method is used by the EnDat, 1Vpp, TTL, Analog, and LVDT modules.

**Method**

```
double getDeviceOffset
(
    const unsigned short& channel
);
```

**Parameters**

channel	The channel of the encoder that the offset will be applied to
---------	---

**Return value**

double	The offset that will be applied to the position
--------	---

**setRight**

The setRight method is used to set the input of the next module for use in ordering the module chain and allowing for modules to wait until the communication line is ready during DHCP requests.

**Method**

```
MseResults setRight
(
    const bool setConnectOut
);
```

**Parameters**

setConnectOut	A value of true sets the output pin high, false sets it low. Setting the pin high allows the next module to communicate over the network during DHCP discovery and is useful when determining the ordering of the modules.
---------------	--

**Return value**

The return value delivers a response code representing whether the setRight command was sent.

**resetMse1000**

The resetMse1000 method sends a reset request to the MSE.

**Method**

```
MseResults resetMse1000();
```

**Return value**

The return value delivers a response code representing whether the reset command was sent.

**program**

The program method programs the module with the selected file and then waits for the reboot to finish. The modules must not be in DHCP mode when programming because the IP address must be constant for the entire programming process.

Programming the MSEfirmware with a version > 1.0.2 requires that the MSEbootloader be at least version 1.0.2. The MSE-bootloader and MSEfirmware cannot be programmed to a version prior to version 1.0.3 once they are version 1.0.3 or greater. The versioning incompatibility is due to improvements in configuration data robustness.

**Method**

```
MseResults program
(
    const char* filename,
    bool isBootloader
);
```

**Parameters**

filename                    The file to program into the firmware  
 bool isBootloader        True if the file to program is the bootloader, otherwise false

**Return value**

The return value delivers a response code representing whether the programming completed.

**getProgrammingState**

The getProgrammingState method returns the programming state. This method exists as a convenience for users of this library.

**Method**

```
PROGRAMMING_STATE_ENUMS getProgrammingState();
```

**Return value**

The return value delivers a PROGRAMMING\_STATE\_ENUMS containing the programming state.

**getProgrammingPercentComplete**

The getProgrammingPercentComplete method returns the percent complete of the programming. The percent complete is updated during the PROGRAMMING\_STATE\_DOWNLOADING state.

**Method**

```
double getProgrammingPercentComplete();
```

**Return value**

The return value delivers a double containing the percent complete.

**showModuleType**

The showModuleType method is used to return a string representation of the type based on the MODULE\_ID enumeration.

**Method**

```
static char* showModuleType
(
    MODULE_ID type
);
```

**Parameters**

type                        The MODULE\_ID to stringify

**Return value**

string                      The string type representation of the MODULE\_ID



**showModuleId**

The showModuleId method is used to return a string representation of the ID based on the MODULE\_ID enumeration

**Method**

```
static char* showModuleId
(
    MODULE_ID          type
);
```

**Parameters**

type                      The MODULE\_ID to stringify

**Return value**

string                    The string ID representation of the MODULE\_ID

**setIp**

The setIp method sets the IP address of the module. This method also validates the strings passed in for errors.

**Method**

```
MseResults setIp
(
    const char*        address,
    const char*        netmask
);
```

**Parameters**

address                  The IP address to set the module to  
netmask                  The netmask to set the module to

**Return value**

The return value delivers a response code representing whether the IP was set correctly.

**setAsyncPort**

The setAsyncPort method sets the UDP asynchronous port that the module will send broadcast, error, latching, and referencing commands to.

**Method**

```
MseResults setAsyncPort
(
    const unsigned short* asynchronousPort
);
```

**Parameters**

asynchronousPort      The port for the asynchronous communication. The UDP port must be between 1024 and 49151 (registered ports).

**Return value**

The return value delivers a response code representing whether the command was successful.

**getAsyncPort**

The getAsyncPort method gets the UDP asynchronous port that the module will send broadcast, error, latching, and referencing commands to.

**Method**

```
MseResults getAsyncPort
(
    unsigned short* asynchronousPort
);
```

**Parameters**

asynchronousPort A pointer to an unsigned short that will be filled in with the port for the asynchronous communication

**Return value**

The return value delivers a response code representing whether the command was successful.

**setDhcp**

The setDhcp method sets whether or not to use DHCP.

**Method**

```
MseResults setDhcp
(
    unsigned char choice
);
```

**Parameters**

choice A value of 0 will disable DHCP, 1 will enable it

**Return value**

The return value delivers a response code representing whether the setDhcp was set correctly.

**broadcastOpenConnection**

The broadcastOpenConnection method sends a broadcast message to open a connection to the MSE 1000 module. The modules will all send back responses containing their IP address and other data in the form of a MSE1000ConnectResponse. The total number of responses is also returned. The setBroadcastingNetmask method can be used to set the type of broadcast to perform. The modules will not be ordered.

**Method**

```
MseResults broadcastOpenConnection
(
    const char* clientIpAddress,
    const unsigned short* port,
    MSE1000ConnectResponse* connResponses,
    unsigned short* numResponses
);
```

**Parameters**

- clientIpAddress The IP address of the sender. This is needed in order to bind a socket for the modules to send responses to.
- port The UDP port of the sender needed for the bind
- connResponses The responses to the broadcast. This must be an array of MSE1000ConnectResponses that is large enough to hold MAX\_NUM\_MODULES of responses.
- numResponses The number of modules that responded to the broadcast

**setBroadcastingNetmask**

The setBroadcastingNetmask method sets the netmask used during broadcasting. A netmask of "255.255.255.255" should be used for a limited broadcast. A netmask of "255.255.255.0", "255.255.0.0", or "255.0.0.0" should be used for a directed broadcast. A limited broadcast is limited to a single LAN and is received by all clients connected to that LAN. A directed broadcast will be sent to all clients on a specific subnet.

**Method**

```
void setBroadcastingNetmask
(
    const char*          netmask
);
```

**Parameters**

netmask                      A char\* in the form 255.255.255.0 that will be used for determining the broadcasting address. A value of 255.255.255.255 will be used as the default.

**restoreFactoryDefaults**

The restoreFactoryDefaults method sends a broadcast to all modules that will reset the static IP address to 172.31.46.2 for power supplies and 172.31.46.1 for all other modules. The modules will be set to DHCP addressing and if a DHCP server is not present, each module will change back to static after a 1 minute timeout. The asynchronous port will be set to MSE1000\_ASYNC\_PORT.

**Method**

```
void restoreFactoryDefaults
(
);
```

**setUdpTimeout**

The setUdpTimeout method sets the UDP timeout that is used when waiting for a response from a module. The setUdpTimeout method can be set from 50 ms to 10000 ms. The default is 800ms in order to handle the validation and backup of the FRAM and FLASH memory during programming and setting of the IP address. Very large values are only useful for debugging purposes. Values below the default may not allow enough time for the microcontroller to respond.

**Method**

```
void setUdpTimeout
(
    long                timeoutMs
);
```

**Parameters**

timeoutMs                      The timeout to wait for a UDP response in milliseconds

**getUdpTimeout**

The getUdpTimeout method gets the UDP timeout in milliseconds that is used when waiting for a response from a module.

**Method**

```
long getUdpTimeout
(
);
```

**Return value**

long                              The UDP timeout in milliseconds that is used when waiting for a response from a module

**setUdpNumRetries**

The setUdpNumRetries method sets the number of retries to use if a timeout occurs. The default is 0 and it can be set to as high as 10.

**Method**

```
void setUdpNumRetries
(
    short numRetries
);
```

**Parameters**

numRetries                      The number of retries

**getUdpNumRetries**

The getUdpNumRetries method gets the number of retries to use if a timeout occurs.

**Method**

```
short getUdpNumRetries
(
);
```

**Return value**

short                              The number of retries to use if a timeout occurs

**setNetworkDelay**

The setNetworkDelay method sets the delay in milliseconds to use before each UDP message is sent to the module. A delay less than 17 ms will utilize a busy loop so that an operating system context switch can be avoided, otherwise, a Sleep is used which will give up priority to other processes that may need to run.

**Method**

```
void setNetworkDelay
(
    short networkDelayMs
);
```

**Parameters**

networkDelayMs                      The number of milliseconds to delay. The value can be from 0 to 1000. The default is 0.

**getNetworkDelay**

The getNetworkDelay method sets the delay in milliseconds to use before each UDP message is sent to the module.

**Method**

```
short getNetworkDelay
(
);
```

**Return value**

short                              The network delay in milliseconds to use before each UDP message is sent to the module

**setLatch**

The setLatch method is used to latch or unlatch channel data in the MSE.

**Method**

```
MseResults setLatch
(
    const LATCH_OPTIONS    latchOption,
    const LATCH_CHOICE    latchChoice
);
```

**Parameters**

latchOption	The LATCH_OPTIONS to choose
latchChoice	The enumerated latch number to select for setting or resetting

**getLatch**

The getLatch method is used to get the status of the latches in the MSE 1000. An active latch will inform the client that latched data is available or can be used as a signal. The latch is cleared when the data is read or the setLatch LATCH\_COUNT\_RESET, LATCH\_CHOICE\_ALL method is called.

**Method**

```
MseResults getLatch
(
    unsigned char*    latchState,
    const unsigned short    size
);
```

**Parameters**

latchState	An array that will hold the status of the latch states. Will store a 1 if the corresponding latch is set, otherwise 0.
size	The size of the latchState array passed in. Must be large enough to store all 5 latches.

**getAdcValues**

The getAdcValues method is used to get the ADC (Analog to digital conversion) values from the module. The ADC values are used to get a digital representation of voltages and temperatures used in the module.

**Method**

```
MseResults getAdcValues
(
    short*    adcVals,
    const unsigned short    length
);
```

**Parameters**

adcVals	The address where the ADC values will be returned
length	The length of the adcVals array passed in (must be at least ADC_NUM_CHANNELS shorts to store the entire response)

**getIntegrity**

The getIntegrity method is used to get the system integrity values from the module. The values are returned masked in an unsigned long. The INTEGRITY\_ENUMS enumeration can be used to see what integrity value is currently out of specification. The ranges array stores the warning and error ranges in the following order: Current warning, current error, 24V min error, 24V max error, 24V min warning, 24V max warning, 5V min error, 5V max error, 5V min warning, 5V max warning, temperature min error, temperature max error, temperature min warning, temperature max warning.

**Method**

```
MseResults getIntegrity
(
    unsigned long*    integrity,
    double*          ranges,
    unsigned short   numRanges
);
```

**Parameters**

- integrity                      The address where the integrity masked values values will be stored
- ranges                        A pointer to an array of doubles to store the ranges for the integrity checks
- numRanges                    The size of the ranges array passed in (must hold at least NUM\_INTEGRITY\_RANGES doubles)

**setAsyncMode**

The setAsyncMode method is used to set the asynchronous mode of the module. Setting to asynchronous will allow for logging, footswitch updates, triggering updates, and EnDat encoder warning and error updates. Asynchronous updates will be sent to the MSE1000\_ASYNC\_PORT of the IP address that requested this method.

**Method**

```
MseResults setAsyncMode
(
    bool useAsync
);
```

**Parameters**

- useAsync                      True to enable asynchronous updates, false to disable

**clearAllErrors**

The clearAllErrors method is used to clear the systemIntegrity as well as the encoder warnings and errors. If warnings or errors still persist, they will be immediately set again.

**Method**

```
MseResults clearAllErrors
(
);
```

**clearIntegrityErrors**

The clearIntegrityErrors method is used to clear the module system integrity warnings and errors. If warnings or errors still persist, they will be immediately set again.

**Method**

```
MseResults clearIntegrityErrors
(
);
```

**enableDiags**

The enableDiags method is used to set the diagnostics mode. The diagnostics mode affects the throughput of the data since diagnostics will be performed while the counts are being updated. Diagnostics are performed once every 200 ms as long as they are not set to DIAG\_MODE\_NONE.

**Method**

```
MseResults enableDiags
(
    const DIAG_MODE_OPTIONS choice
);
```

**Parameters**

choice	DIAG_MODE_FULL	Enables function reserves (for Endat), errors and warnings (for EnDat), control register status (EnDat, 1Vpp and TTL), and system integrity (all modules)
	DIAG_MODE_STATUS	Enables errors and warnings (for EnDat), control register status (EnDat, 1Vpp and TTL), and system integrity (all modules)
	DIAG_MODE_MINIMAL	Enables system integrity (all modules)
	DIAG_MODE_NONE	Will not monitor anything

**getLibraryVersion**

The getLibraryVersion method is used to return the version of the library as a string.

**Method**

```
static char* getLibraryVersion
(
);
```

## C Functions

The common C functions can be found in the MseModuleWrapper.h file.

### MseModuleCreate

Creates a MseModule object and returns a pointer to it.

#### Function

```
MseModulePtr MseModuleCreate
(
);
```

#### Return value

The return value delivers a pointer to the MseModule object that was created.

### MseModuleDelete

Deletes the MseModule object that was passed in.

#### Function

```
void MseModuleDelete
(
    MseModulePtr object
);
```

#### Parameters

object                    A pointer to the MseModule object that was created by the MseModuleCreate function

### MseModuleInitialize

Initializes the MseModule object that was passed in.

#### Function

```
MSE_RESPONSE_CODE MseModuleInitialize
(
    MseModulePtr object,
    char* mseIpAddress,
    bool useAsync
);
```

#### Parameters

object                    A pointer to the MseModule object that was created by the MseModuleCreate function

mseIpAddress            The IP address of the module to initialize

useAsync                 Whether to enable asynchronous communication from the module

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseModuleGetLibraryVersion**

Gets the version of the MSElibrary.

**Function**

```
void MseModuleGetLibraryVersion
(
    char* version,
);
```

**Parameters**

version A pointer to the location where the MSElibrary version will be copied to. The location pointed to should be at least 16 bytes to allow for future updates.

**MseModuleGetModuleType**

Gets the module type.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetModuleType
(
    MseModulePtr object,
    MODULE_ID* moduleType
);
```

**Parameters**

object A pointer to the MseModule object that was created by the MseModuleCreate function  
moduleType A pointer to the location where the module type will be copied to

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetModuleErrorState**

Gets the error state of the module. A value of True signifies that there is an error.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetModuleErrorState
(
    MseModulePtr object,
    bool* errorState
);
```

**Parameters**

object A pointer to the MseModule object that was created by the MseModuleCreate function  
errorState A pointer to the location where the error state will be copied to. A subsequent call to MseModuleGetModuleErrors can be made to get the actual errors.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetModuleErrors**

Gets the actual module errors.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetModuleErrors
(
    MseModulePtr      object,
    long*              errors,
    double*            ranges,
    short              size
);
```

**Parameters**

- object            A pointer to the MseModule object that was created by the MseModuleCreate function
- errors           A pointer to the location where the errors will be copied to. The errors is a single long that can be masked with the INTEGRITY\_ENUMS to determine which error has occurred.
- ranges           A pointer to the location where the ranges used to determine an error will be copied to. The ranges is an array that must be large enough to hold NUM\_INTEGRITY\_RANGES.
- size             The size of the ranges array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAdcValues**

Gets the voltage and temperature values for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAdcValues
(
    MseModulePtr      object,
    short*             adcValues,
    short              size
);
```

**Parameters**

- object            A pointer to the MseModule object that was created by the MseModuleCreate function
- adcValues        A pointer to the location where the voltage and temperature values will be copied to. The adcValues is an array that must be large enough to hold ADC\_NUM\_CHANNELS. The array can be indexed using the ADC\_OPTIONS enumeration.
- size             The size of the adcValues array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleClearErrors**

Clears the module errors and warnings.

**Function**

```
MSE_RESPONSE_CODE MseModuleClearErrors
(
    MseModulePtr    object
);
```

**Parameters**

object                    A pointer to the MseModule object that was created by the MseModuleCreate function

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleSetIpAddress**

Sets the static IP address and netmask for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleSetIpAddress
(
    MseModulePtr    object,
    char*           ipAddress,
    char*           netmask
);
```

**Parameters**

object                    A pointer to the MseModule object that was created by the MseModuleCreate function  
ipAddress                 The IP address to set  
netmask                   The netmask to set

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetIpAddress**

Gets the currently used IP address for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetIpAddress
(
    MseModulePtr    object,
    char*           ipAddress
);
```

**Parameters**

object                    A pointer to the MseModule object that was created by the MseModuleCreate function  
ipAddress                 A pointer to the location where the IP address will be stored. Must be large enough to hold SIZE\_IP\_ADDRESS bytes.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetIpStaticAddress**

Gets the static IP address for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetIpStaticAddress
(
    MseModulePtr    object,
    char*           ipAddress
);
```

**Parameters**

- object            A pointer to the MseModule object that was created by the MseModuleCreate function
- ipAddress        A pointer to the location where the static IP address will be stored. Must be large enough to hold SIZE\_IP\_ADDRESS bytes.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetNetmask**

Gets the currently used netmask for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetNetmask
(
    MseModulePtr    object,
    char*           netmask
);
```

**Parameters**

- object            A pointer to the MseModule object that was created by the MseModuleCreate function
- ipAddress        A pointer to the location where the netmask will be stored. Must be large enough to hold SIZE\_IP\_ADDRESS bytes.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetNetmaskStatic**

Gets the static netmask for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetNetmaskStatic
(
    MseModulePtr    object,
    char*           netmask
);
```

**Parameters**

- object            A pointer to the MseModule object that was created by the MseModuleCreate function
- ipAddress        A pointer to the location where the static netmask will be stored. Must be large enough to hold SIZE\_IP\_ADDRESS bytes.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetPort**

Gets the UDP port for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetPort
(
    MseModulePtr    object,
    short*          port
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
ipAddress	A pointer to the location where the UDP port will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleSetAsyncPort**

Sets the UDP asynchronous port that the module will send broadcast, error, latching, and referencing commands to.

**Method**

```
MseResults MseModuleSetAsyncPort
(
    MseModulePtr    object,
    const unsigned short* asynchronousPort
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
asynchronousPort	The port for the asynchronous communication. The UDP port must be between 1024 and 49151 (registered ports).

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncPort**

Gets the UDP asynchronous port that the module will send broadcast, error, latching, and referencing commands to.

**Method**

```
MseResults MseModuleGetAsyncPort
(
    MseModulePtr    object,
    unsigned short* asynchronousPort
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
asynchronousPort	A pointer to an unsigned short that will be filled in with the port for the asynchronous communication

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleSetUsingDhcp**

Sets the DHCP setting for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleSetUsingDhcp
(
    MseModulePtr    object,
    bool            isDhcp
);
```

**Parameters**

- object                    A pointer to the MseModule object that was created by the MseModuleCreate function
- isDhcp                    True to enable DHCP

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetUsingDhcp**

Gets the DHCP setting for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetUsingDhcp
(
    MseModulePtr    object,
    bool*           isDhcp
);
```

**Parameters**

- object                    A pointer to the MseModule object that was created by the MseModuleCreate function
- isDhcp                    A pointer to the location where the DHCP setting will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetMacAddress**

Gets the MAC address for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetMacAddress
(
    MseModulePtr    object,
    char*           macAddress
);
```

**Parameters**

- object                    A pointer to the MseModule object that was created by the MseModuleCreate function
- macAddress                A pointer to the location where the MAC address will be stored. Must be large enough to hold SIZE\_MAC\_ADDRESS bytes.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetBootloaderVersion**

Gets the bootloader version for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetBootloaderVersion
(
    MseModulePtr    object,
    char*           version
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
version	A pointer to the location where the bootloader version will be stored. This must be large enough to hold SIZE_BUILD_INFO bytes

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetFirmwareVersion**

Gets the firmware version for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetFirmwareVersion
(
    MseModulePtr    object,
    char*           version
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
version	A pointer to the location where the firmware version will be stored. This must be large enough to hold SIZE_BUILD_INFO bytes

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetSerialNumber**

Gets the serial number for the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetSerialNumber
(
    MseModulePtr    object,
    char*           serialNumber
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
serialNumber	A pointer to the location where the serial number will be stored. This must be large enough to hold SIZE_SERIAL_NUMBER bytes.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleReset**

Reboots the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleReset
(
    MseModulePtr    object,
);
```

**Parameters**

object                    A pointer to the MseModule object that was created by the MseModuleCreate function

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleShowType**

Shows the textual representation of the module type.

**Function**

```
MSE_RESPONSE_CODE MseModuleShowType
(
    MseModulePtr    object,
    char*           type
);
```

**Parameters**

object                    A pointer to the MseModule object that was created by the MseModuleCreate function  
 type                     A pointer to the location where the type will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleShowId**

Shows the textual representation of the module ID.

**Function**

```
MSE_RESPONSE_CODE MseModuleShowId
(
    MseModulePtr    object,
    char*           id
);
```

**Parameters**

object                    A pointer to the MseModule object that was created by the MseModuleCreate function  
 id                        A pointer to the location where the ID will be stored. This must be large enough to hold 10 bytes.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseModuleSetBroadcastingNetmask**

Set the netmask used for the broadcast. A netmask of "255.255.255.255" should be used for a limited broadcast. A netmask of "255.255.255.0", "255.255.0.0", or "255.0.0.0" should be used for a directed broadcast. A limited broadcast is limited to a single LAN and is received by all clients connected to that LAN. A directed broadcast will be sent to all clients on a specific subnet.

**Function**

```
MSE_RESPONSE_CODE MseModuleSetBroadcastingNetmask
(
    MseModulePtr    object,
    char*           netmask
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
netmask	The netmask to be used for broadcasting. A value of 255.255.255.255 will be used as the default.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleSetUdpTimeout**

Set the UDP timeout used for communication to the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleSetUdpTimeout
(
    MseModulePtr    object,
    long            timeoutMs
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
timeoutMs	The timeout to be used in milliseconds. Can be set from 50 to 10000. The default is 800ms in order to handle the validation and backup of the FRAM and FLASH memory during programming and setting of the IP address. Very large values are only useful for debugging purposes. Values below the default may not allow enough time for the microcontroller to respond.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetUdpTimeout**

Get the UDP timeout used for communication to the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetUdpTimeout
(
    MseModulePtr    object,
    long*           timeoutMs
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
timeoutMs	A pointer to the location where the timeout to be used in milliseconds will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleSetUdpNumRetries**

Set the number of retries to use when communicating to the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleSetUdpNumRetries
(
    MseModulePtr    object,
    short           numRetries
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
numRetries	The number of retries to use

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetUdpNumRetries**

Get the number of retries to use when communicating to the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetUdpNumRetries
(
    MseModulePtr    object,
    short*          numRetries
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
numRetries	A pointer to the location where the number of retries to use will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleSetNetworkDelay**

Set the delay to use between commands to the module. The value can be from 0 ms to 1000 ms and defaults to 0 ms.

**Function**

```
MSE_RESPONSE_CODE MseModuleSetNetworkDelay
(
    MseModulePtr    object,
    short           networkDelayMs
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
networkDelayMs	The network delay to use

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetNetworkDelay**

Get the delay to use between commands to the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetNetworkDelay
(
    MseModulePtr    object,
    short           networkDelayMs
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
networkDelayMs	A pointer to the location where the network delay to use will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleBroadcast**

Sends out a broadcast to obtain the IP addresses, netmasks, and ports of all the modules. The moduleIpAddresses, moduleNetmasks, and modulePorts arrays must be large enough to hold MAX\_NUM\_MODULES. The modules will not be ordered.

**Function**

```
MSE_RESPONSE_CODE MseModuleBroadcast
(
    MseModulePtr    object,
    char*           clientIp,
    short           clientPort,
    short*          numResponses,
    char*           moduleIpAddresses,
    char*           moduleNetmasks,
    short*          modulePorts
);
```

**Parameters**

object	A pointer to the MseModule object that was created by the MseModuleCreate function
clientIp	The IP address of the client
clientPort	The UDP port of the client
numResponses	A pointer to the location where the number of modules found is stored
moduleIpAddresses	A pointer to the location the the IP addresses of the modules are stored. The IP addresses will be delimited by spaces.
moduleNetmasks	A pointer to the location the the netmasks of the modules are stored. The netmasks will be delimited by spaces.
modulePorts	A pointer to the location where the module UDP ports are stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleProgram**

Programs the module with the selected file.

Programming the MSEfirmware with a version > 1.0.2 requires that the MSEbootloader be at least version 1.0.2. The MSE-bootloader and MSEfirmware cannot be programmed to a version prior to version 1.0.3 once they are version 1.0.3 or greater. The versioning incompatibility is due to improvements in configuration data robustness.

**Function**

```
MSE_RESPONSE_CODE MseModuleProgram
(
    MseModulePtr    object,
    char*           filename,
    bool            isBootloader
);
```

**Parameters**

- object                    A pointer to the MseModule object that was created by the MseModuleCreate function
- filename                The file to program into the module
- isBootloader            True if the file being programmed is the MSEbootloader, false if it is the MSEfirmware

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetProgramState**

Gets the programming state.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetProgramState
(
    MseModulePtr    object,
    PROGRAMMING_STATE_ENUMS* programState
);
```

**Parameters**

- object                    A pointer to the MseModule object that was created by the MseModuleCreate function
- programState            The state of the programming

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetProgramPercentComplete**

Gets the programming percent complete. The percent complete is updated during the PROGRAMMING\_STATE\_DOWNLOADING state.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetProgramPercentComplete
(
    MseModulePtr    object,
    double*         percentComplete
);
```

**Parameters**

- object                    A pointer to the MseModule object that was created by the MseModuleCreate function
- percentComplete        The percent complete of the programming

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncMsgType**

Gets the asynchronous message type received from the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAsyncMsgType
(
    char*          msg,
    UdpCmdType*   code
);
```

**Parameters**

msg	The asynchronous message received from the module
code	The location where the message type will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncMsgIpAddress**

Gets the asynchronous IP address received from the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAsyncMsgIpAddress
(
    char*          msg,
    char*          ipAddress
);
```

**Parameters**

msg	The asynchronous message received from the module
ipAddress	The location where the ip address of the module will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncMsgPort**

Gets the asynchronous UDP port received from the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAsyncMsgPort
(
    char*          msg,
    short*         port
);
```

**Parameters**

msg	The asynchronous message received from the module
port	The location where the UDP port of the module will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncMsgDhcp**

Gets the asynchronous DHCP state received from the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAsyncMsgDhcp
(
    char*          msg,
    bool*         isDhcp
);
```

**Parameters**

msg                    The asynchronous message received from the module  
isDhcp                The location where the DHCP state of the module will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncMsgMacAddress**

Gets the asynchronous MAC address received from the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAsyncMsgMacAddress
(
    char*          msg,
    char*         macAddress
);
```

**Parameters**

msg                    The asynchronous message received from the module  
macAddress            The location where the MAC address of the module will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncMsgNetmask**

Gets the asynchronous netmask received from the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAsyncMsgNetmask
(
    char*          msg,
    char*         netmask
);
```

**Parameters**

msg                    The asynchronous message received from the module  
netmask                The location where the netmask of the module will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncMsgSerialNumber**

Gets the asynchronous serial number received from the module.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAsyncMsgSerialNumber
(
    char* msg,
    char* serialNumber
);
```

**Parameters**

msg	The asynchronous message received from the module
serialNumber	The location where the serial number of the module will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncMsgChannelStatus**

Informs the listener that a warning, error, or reference complete has occurred. The listener must then read the channel status or send a reference complete acknowledge in order to tell the module that the message has been received.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAsyncMsgChannelStatus
(
    char* msg,
    short* type,
    short* channel
);
```

**Parameters**

msg	The asynchronous message received from the module
type	The type of channel status. 0 is currently not used, 1 is for warnings and errors, and 2 is for reference complete.
channel	The channel for the 'reference complete' type. This parameter is not used for warnings and errors.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleGetAsyncMsgLatch**

Informs the listener that a latch has occurred.

**Function**

```
MSE_RESPONSE_CODE MseModuleGetAsyncMsgLatch
(
    char* msg,
    char* latchVals
);
```

**Parameters**

- msg                      The asynchronous message received from the module
- latchVals                The value of each of the latches. A value of 0 is not triggered, a value of 1 is triggered.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseModuleShowRespCode**

Gets the string representation of the response code.

**Function**

```
MSE_RESPONSE_CODE MseModuleShowRespCode
(
    char* response,
    MSE_RESPONSE_CODE code
);
```

**Parameters**

- response                The location where the text representation of the MSE\_RESPONSE\_CODE will be stored
- code                    The MSE\_RESPONSE\_CODE to get a string representation of

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



## 2.11 Device methods

Device methods are only available for C++ usage.

### MseDeviceModule

The MseDeviceModule method creates an array of EncoderInfo objects for use in storing encoder information for each channel.

#### Constructor

```
MseDeviceModule();
```

### getEncoderInfo

The getEncoderInfo method returns the EncoderInfo object used to store the encoder information for a specific channel.

#### Method

```
bool getEncoderInfo
(
    EncoderInfo*      encoderInfo,
    const unsigned short channel
);
```

#### Parameters

encoderInfo,	A pointer to a EncoderInfo structure to hold the encoder information requested
channel	The channel to get the encoder data of indexed from 0

#### Return value

The return value delivers true if getEncoderInfo passed and false if the channel requested is an invalid index.

### setEncoderInfo

The setEncoderInfo method sets the encoderInfo\_ object to the desired values.

#### Method

```
bool setEncoderInfo
(
    EncoderInfo*      encoderInfo,
    const unsigned short channel
);
```

#### Parameters

encoderInfo	A pointer to a encoderInfo structure that has the encoder information to set
channel	The channel to set the encoder data to indexed from 0

#### Return value

The return value delivers true if setEncoderInfo passed and false if the channel requested is an invalid index.

## getCountingDirection

Gets the counting direction for a specific channel. True for positive and false for negative. Positive is when the count direction matches the traverse direction.

### Method

```
bool getCountingDirection
(
    bool* isPositive,
    const unsigned short channel
);
```

### Parameters

isPositive      A pointer where the counting direction will be stored

channel          The channel to get the counting direction from indexed from 0

### Return value

The return value delivers False if the channel is out of range or if the pointer is NULL.

## setErrorCompensation

Sets the linear error compensation for a specific channel to the desired multiplier. This value will be applied to the position when getPositions is called.

### Method

```
bool setErrorCompensation
(
    double value,
    const unsigned short channel
);
```

### Parameters

value            The multiplier to use for error compensation of the position value

channel          The channel to set indexed from 0

### Return value

The return value delivers False if the channel is out of range.

## getErrorCompensation

Gets the linear error compensation for a specific channel.

### Method

```
bool getErrorCompensation
(
    double* value,
    const unsigned short channel
);
```

### Parameters

value            The multiplier used for error compensation of the position value

channel          The channel to get indexed from 0

### Return value

The return value delivers False if the channel is out of range or if the pointer is NULL.

## setScaling

Sets the device scaling for a specific channel to the desired multiplier. This value will be applied to the position when `getPosition` is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

### Method

```
bool setScaling
(
    const double      value,
    const unsigned short channel
);
```

### Parameters

value	The multiplier to use for scaling of the position value
channel	The channel to set indexed from 0

### Return value

The return value delivers `False` if the channel is out of range.

## getScaling

Gets the device scaling for a specific channel. This value will be applied to the position when `getPosition` is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

### Method

```
bool getScaling
(
    double*          value,
    const unsigned short channel
);
```

### Parameters

value	The multiplier used for scaling of the position value
channel	The channel to get indexed from 0

### Return value

The return value delivers `False` if the channel is out of range or if the pointer is `NULL`.

## getResolution

Gets the computed encoder resolution for a specific channel. This value is in mm/count for linear encoders and degrees/count for rotary.

### Method

```
bool getResolution
(
    double* resolution,
    const unsigned short channel
);
```

### Parameters

resolution	The resolution of the encoder
channel	The channel to get indexed from 0

### Return value

The return value delivers False if the channel is out of range or if the pointer is NULL.

## getEncoderType

Gets the encoder type for a specific channel.

### Method

```
bool getEncoderType
(
    ENCODER_TYPES_ENUM* type,
    const unsigned short channel
);
```

### Parameters

type	The type of the encoder
channel	The channel to get indexed from 0

### Return value

The return value delivers False if the channel is out of range or if the pointer is NULL.

## getUom

Gets the units of measure for a specific channel.

### Method

```
bool getUom
(
    UOM* uom,
    const unsigned short channel
);
```

### Parameters

uom	The units of measure of the encoder
channel	The channel to get indexed from 0

### Return value

The return value delivers False if the channel is out of range or if the pointer is NULL.

## enableErrorChecking

The enableErrorChecking method sets whether error checking will be done on the specified channel. The error checking refers to the function reserves, errors, and warnings for EnDat modules, and the counter errors for the 1Vpp and TTL modules.

### Method

```
MseResults enableErrorChecking
(
    const bool          choice,
    const unsigned short channel
);
```

### Parameters

choice	True if error checking should be enabled
channel	The channel to enable or disable error checking on indexed from 0

### Return value

The return value delivers a response code representing whether the error checking command was sent correctly.

## getChannelStatus

The getChannelStatus method gets the error status of a 1Vpp or TTL encoder. The COUNTER\_STATUS enumeration has the mask values to compare the status with to determine which error occurred.

### Method

```
MseResults getChannelStatus
(
    const unsigned short channel,
    unsigned char*       channelStatus
);
```

### Parameters

channel	The channel to get the status of indexed from 0
channelStatus	Holds the masked status of the channel. The status can be obtained by masking this value with the COUNTER_STATUS enumeration.

### Return value

The return value delivers a response code representing whether the getChannelStatus command was sent correctly.

## clearErrorsAndWarnings

The clearErrorsAndWarnings method clears errors and warnings. The EnDat modules will clear the errors from the EnDat protocol. The 1Vpp and TTL modules will clear the errors through the counter status register.

### Method

```
MseResults clearErrorsAndWarnings
(
    unsigned short channel
);
```

### Parameters

channel	The channel to clear errors and warnings for indexed from 0
---------	---

### Return value

The return value delivers a response code representing whether the warnings and errors were cleared correctly.

## setLatchDebounce

The setLatchDebounce method sets the number of milliseconds to use for debouncing the hardware latch input. The input defaults to 10ms on bootup of the module and is based on having a footswitch attached. If another device is attached that has a faster or slower debounce time, modifying this value will speed up or slow down latch triggering.

### Method

```
MseResults setLatchDebounce  
(  
    const unsigned char choice,  
    const unsigned short timeMs  
);
```

### Parameters

choice	0 for the first footswitch input, 1 for the second
timeMs	The number of milliseconds to use to debounce the requested input. Can be between 0 - 20ms.

## 2.12 EnDat methods and functions

C++ methods and the C functions are separated into two sections for easier lookup.

### C++ methods

#### Constructor

```
MseEndatModule(void);
```

#### initializeModule

The initializeModule method will fill in the moduleData\_ and deviceData\_ structure with all the information known from the module and it's devices.

#### Method

```
virtual MseResults initializeModule
(
    const char*      mseIpAddress
    bool            useAsync
);
```

#### Parameters

mseIpAddress	The IP address of the module to initialize
useAsync	True if the MSE should send asynchronous messages to the MSE_ASYNC_PORT

#### Return value

The return value delivers a response code representing whether the initialization information was retrieved correctly.

#### getPositions

The getPositions method is used to return the positions and the current revolutions for the attached encoders. Linear encoders just return the positions. The position of a linear encoder is returned in user units. The position of a rotary encoder is returned in degrees. The resulting position is determined by first multiplying the count by the error compensation, then multiplying by the scaling, then computing the position based on the resolution, then converting to the correct UOM, then adding the device offset, and finally formatting the rotary position if necessary. The error compensation is set with the setErrorCompensation method of the base class, it defaults to 1.0. The scaling is set with the setScaling method of the base class, it defaults to 1.0. The resolution is known internally since the EnDat encoders maintain this value. The UOM is set with setUom, it defaults to UOM\_DEGREES for rotary encoders and UOM\_MM for linear encoders. The device offset is set with the setDeviceOffset method of the base class, it defaults to 0.0. The rotary format can be changed with the setRotaryFormat method, it defaults to ROTARY\_FORMAT\_360.

#### Method

```
MSELIB_EXPORT MseResults getPositions
(
    double*          pos,
    long*           currentRevolution,
    const unsigned short& numChannels,
    COUNT_REQUEST_OPTION option
);
```

#### Parameters

pos	The position(s) scaled from counts to the correct unit of measurement
currentRevolution	The current revolution of a rotary encoder
numChannels	The number of channels to read and store into the array passed in with pos and currentRevolution
option	The type of position to return. If COUNT_REQUEST_LATCHED is requested, the latching in the module will be reset after the value is read.

**getCounts**

Gets the counts of all the attached encoders. Rotary encoders will also return the current revolution.

**Method**

```
MSELIB_EXPORT MseResults getCounts
(
    unsigned long*    counts,
    long*            currentRevolution,
    unsigned short    numChannels,
    COUNT_REQUEST_OPTION option
)
```

**Parameters**

- counts            The counts returned from the encoders
- currentRevolution    The current revolution of a rotary encoder
- numChannels        The number of channels to read and store into the array passed in with counts and currentRevolution
- option             The type of count to return. If COUNT\_REQUEST\_LATCHED is requested, the latching in the module for the desired latchChoice will be reset after the value is read.

**Return value**

The return value delivers a response code representing whether the counts were retrieved correctly.

**getWarnings**

The getWarnings method returns the encoder warnings as an array of ENDAT\_ERROR\_RESULT values. The warnings in the array are ordered as indexed in the ENDAT\_WARNINGS enumeration.

**Method**

```
MseResults getWarnings
(
    unsigned short    channel,
    ENDAT_ERROR_RESULT* warnings,
    unsigned char     size
);
```

**Parameters**

- channel            The channel to read the warnings from
- warnings           A pointer to an array of ENDAT\_ERROR\_RESULT values that the warnings returned from the devices will be saved to
- size                The number of ENDAT\_ERROR\_RESULT values in the warnings array passed in

**Return value**

The return value delivers a response code representing whether the warnings were retrieved correctly.



**getErrors**

The getErrors method returns the encoder errors as an array of ENDAT\_ERROR\_RESULT values. The errors are ordered as indexed in the ENDAT\_ERRORS enumeration.

**Method**

```
MseResults getErrors
(
    unsigned short    channel,
    ENDAT_ERROR_RESULT* errors,
    unsigned char     size
);
```

**Parameters**

channel	The channel to read the errors from indexed from 0
errors	A pointer to an array of ENDAT_ERROR_RESULT values that the errors returned from the devices will be saved to
size	The number of ENDAT_ERROR_RESULT values in the errors array passed in

**Return value**

The return value delivers a response code representing whether the errors were retrieved correctly.

**getDiag**

The getDiag method returns the encoder function reserves that are supported and their values for the requested channel. The ENDAT\_DIAG enumeration can be used to index into the diagVals parameter to access the desired value.

**Method**

```
MseResults getDiag
(
    const unsigned char    channel,
    unsigned char*        diagVals,
    unsigned char         arrLength
);
```

**Parameters**

channel	The channel to read the diagnostic from indexed from 0
diagVals	The method reserves supported along with their values from the requested channel
arrLength	The size of the diagVals array passed in (must be >= ENDAT_DIAG_COUNT)

**Return value**

The return value delivers a response code representing whether the diags were retrieved correctly.

**getDeviceData**

The getDeviceData method returns the device data for a specified channel.

**Method**

```
bool getDeviceData
(
    DeviceData* data,
    const unsigned short channelNumber
);
```

**Parameters**

- data                      The DeviceData for the requested channel
- channelNumber          The channel to get the DeviceData of indexed from 0

**Return value**

True if the DeviceData was returned, otherwise false.

**getDistinguishableRevolutions**

Gets the distinguishable revolutions of a rotary encoder for a specified channel.

**Method**

```
bool getDistinguishableRevolutions
(
    short* numRevs,
    const unsigned short channelNum
);
```

**Parameters**

- numRevs                 The distinguishable revolutions for the requested channel
- channelNum             The channel to get indexed from 0

**Return value**

False if the channel is out of range or the pointer is NULL.

**getEncoderName**

Gets the encoder name of an encoder for a specified channel.

**Method**

```
bool getEncoderName
(
    char*          name,
    const unsigned short  channelNum
);
```

**Parameters**

name	The encoder name for the requested channel
channelNum	The channel to get indexed from 0

**Return value**

False if the channel is out of range or the pointer is NULL.

**getEncoderId**

Gets the encoder ID of an encoder for a specified channel.

**Method**

```
bool getEncoderId
(
    char*          id,
    const unsigned short  channelNum
);
```

**Parameters**

id	The encoder ID for the requested channel
channelNum	The channel to get indexed from 0

**Return value**

False if the channel is out of range or the pointer is NULL.

**getSerialNumber**

Gets the serial number of an encoder for a specified channel.

**Method**

```
bool getSerialNumber
(
    char*          serialNumber,
    const unsigned short  channelNum
);
```

**Parameters**

serialNumber	The serial number for the encoder of the requested channel
channelNum	The channel to get indexed from 0

**Return value**

False if the channel is out of range or the pointer is NULL.

**setUom**

Sets the unit of measurement of an encoder for a specified channel. This value will be applied to the position when `getPosition` is called.

**Method**

```
bool setUom
(
    UOM          uom,
    const unsigned short channel
);
```

**Parameters**

`uom`                    The unit of measurement for the encoder of the requested channel  
`channel`                The channel to get indexed from 0

**Return value**

False if the channel is out of range.

**getChannelPresence**

The `getChannelPresence` method returns the encoder connection status for a channel on the module.

**Method**

```
MseResults getChannelPresence
(
    unsigned char*   isConnected,
    unsigned short   channel
);
```

**Parameters**

`isConnected`            True if the channel is populated  
`channel`                The channel to request indexed from 0

**Return value**

The return value delivers a response code representing whether the channel presence were retrieved correctly.

**setEncoderInfo**

The `setEncoderInfo` method sets the encoder information values in the `MseDeviceModule` base class to the desired values for the specified channel. The resolution is not modified because it is already known for EnDat modules.

**Method**

```
bool setEncoderInfo
(
    EncoderInfo*     encoderInfo,
    const unsigned short channel
);
```

**Parameters**

`encoderInfo`            A `EncoderInfo` structure holding the encoder information values for the desired channel of this module  
`channel`                The channel to set the encoder information of indexed from 0

**Return value**

The return value delivers true if the encoder information is set or false if channel is greater than the number of channels in the module.

## C Functions

The EnDat C functions can be found in the MseEndatModuleWrapper.h file.

### MseEndatModuleCreate

Creates a MseEndatModule object and returns a pointer to it.

#### Function

```
MseEndatModulePtr MseEndatModuleCreate
(
);
```

#### Return value

The return value delivers a pointer to the MseEndatModule object that was created.

### MseEndatModuleDelete

Deletes the MseEndatModule object that was passed in.

#### Function

```
void MseEndatModuleDelete
(
    MseEndatModulePtr object
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
--------	--

### MseEndatModuleInitialize

Initializes the MseEndatModule object that was passed in.

#### Function

```
MSE_RESPONSE_CODE MseEndatModuleInitialize
(
    MseEndatModulePtr object,
    char* mseIpAddress,
    bool useAsync
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
mseIpAddress	The IP address of the module to initialize
useAsync	Whether to enable asynchronous communication from the module

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetNumChannels**

Gets the number of channels on the module.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetNumChannels
(
    MseEndatModulePtr    object,
    unsigned short*      numChannels
);
```

**Parameters**

- object                    A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- numChannels             A pointer to the location where the number of channels will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetChannelPresence**

Gets whether there is an encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetChannelPresence
(
    MseEndatModulePtr    object,
    bool*                 isConnected,
    short                 channel
);
```

**Parameters**

- object                    A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- isConnected              A pointer to the location where the connection status is stored
- channel                  The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetEncoderType**

Gets the type of encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetEncoderType
(
    MseEndatModulePtr    object,
    ENCODER_TYPES_ENUM*  type,
    short                 channel
);
```

**Parameters**

- object                    A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- type                      A pointer to the location where the encoder type is stored
- channel                  The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleSetUom**

Sets the unit of measurement of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleSetUom
(
    MseEndatModulePtr    object,
    UOM                  uom,
    short                channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
uom	The unit of measurement of the encoder connected to the channel
channel	The channel to set indexed from 0

**MseEndatModuleGetUom**

Gets the unit of measurement of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetUom
(
    MseEndatModulePtr    object,
    UOM*                 uom,
    short                channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
uom	A pointer to the location where the encoder unit of measurement is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleSetErrorCompensation**

Sets the linear error compensation of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleSetErrorCompensation
(
    MseEndatModulePtr    object,
    double               val,
    short                channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
val	The error compensation to use for the channel
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

### MseEndatModuleGetErrorCompensation

Gets the linear error compensation of the encoder connected to the selected channel.

#### Function

```
MSE_RESPONSE_CODE MseEndatModuleGetErrorCompensation
(
    MseEndatModulePtr    object,
    double*              val,
    short                channel
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
val	A pointer to the location where the error compensation is stored
channel	The channel to get indexed from 0

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

### MseEndatModuleSetScaling

Sets the device scaling for a specific channel to the desired multiplier. This value will be applied to the position when MseEndatGetPositions is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

#### Function

```
MSE_RESPONSE_CODE MseEndatModuleSetScaling
(
    MseEndatModulePtr    object,
    const double          val,
    const unsigned short  channel
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
val	The scaling to use for the channel
channel	The channel to set indexed from 0

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

### MseEndatModuleGetScaling

Gets the device scaling for a specific channel. This value will be applied to the position when MseEndatGetPositions is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

#### Function

```
MSE_RESPONSE_CODE MseEndatModuleGetScaling
(
    MseEndatModulePtr    object,
    double*              val,
    const unsigned short  channel
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
val	A pointer to the location where the scaling is stored
channel	The channel to get indexed from 0

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseEndatModuleGetCountingDirection**

Gets the counting direction of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetCountingDirection
(
    MseEndatModulePtr    object,
    bool*                 isPositive,
    short                 channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
isPositive	A pointer to the location where the counting direction is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetDistinguishableRevolutions**

Gets the distinguishable revolutions of a rotary encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetDistinguishableRevolutions
(
    MseEndatModulePtr    object,
    short*                numRevs,
    short                 channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
numRevs	A pointer to the location where the distinguishable revolutions will be stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

### MseEndatModuleGetResolution

Gets the resolution, in mm/count for linear encoders and degrees/count for rotary encoders, of the encoder connected to the selected channel.

#### Function

```
MSE_RESPONSE_CODE MseEndatModuleGetResolution
(
    MseEndatModulePtr    object,
    double*              resolution,
    short                channel
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
resolution	A pointer to the location where the encoder resolution will be stored
channel	The channel to get indexed from 0

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

### MseEndatModuleGetCounts

Gets the encoder counts for all the channels. The position of a linear encoder, in mm, or a rotary encoder, in degrees, can be obtained by multiplying the counts by the resolution value obtained from MseEndatModuleGetResolution.

#### Function

```
MSE_RESPONSE_CODE MseEndatModuleGetCounts
(
    MseEndatModulePtr    object,
    unsigned long*       counts,
    long*                currentRevolution,
    short                numChannels,
    COUNT_REQUEST_OPTION option
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
counts	A pointer to the location where the counts will be stored. This is an array that must be large enough to store MAX_CHANNELS_PER_MODULE.
currentRevolution	A pointer to the location where the current revolution of a rotary encoder will be stored. This is an array that must be large enough to store MAX_CHANNELS_PER_MODULE.
numChannels	The size of the counts and currentRevolution arrays passed in
option	Whether to get the latest or the latched counts

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetPositions**

Gets the encoder positions for all the channels.

The resulting position is determined by first multiplying the count by the error compensation, then multiplying by the scaling, then computing the position based on the resolution, then converting to the correct UOM, then adding the device offset, and finally formatting the rotary position if necessary.

The error compensation is set with the `MseEndatModuleSetErrorCompensation` method of the base class, it defaults to 1.0.

The scaling is set with the `MseEndatModuleSetScaling` method of the base class, it defaults to 1.0.

The resolution is known internally since the EnDat encoders maintain this value.

The UOM is set with `MseEndatModuleSetUom`, it defaults to `UOM_DEGREES` for rotary encoders and `UOM_MM` for linear encoders.

The device offset is set with the `MseEndatModuleSetDeviceOffset` method of the base class, it defaults to 0.0.

The rotary format can be changed with the `MseEndatModuleSetRotaryFormat` method, it defaults to `ROTARY_FORMAT_360`.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetPositions
(
    MseEndatModulePtr    object,
    double*              pos,
    long*                currentRevolution,
    short                numChannels,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

object	A pointer to the <code>MseEndatModule</code> object that was created by the <code>MseEndatModuleCreate</code> function
pos	A pointer to the location where the positions will be stored. This is an array that must be large enough to store <code>MAX_CHANNELS_PER_MODULE</code> .
currentRevolution	A pointer to the location where the current revolution of a rotary encoder will be stored. This is an array that must be large enough to store <code>MAX_CHANNELS_PER_MODULE</code> .
numChannels	The size of the counts and <code>currentRevolution</code> arrays passed in
option	Whether to get the latest or the latched positions

**Return value**

The return value delivers a `MSE_RESPONSE_CODE` representing whether the function call was successful.

**MseEndatModuleSetRotaryFormat**

The `MseEndatModuleSetRotaryFormat` method is used to set the rotary format that will be applied to the position calculated from the counts in the `MseEndatModuleGetPositions` function.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleSetRotaryFormat
(
    MseEndatModulePtr    object,
    unsigned short        channel,
    ROTARY_FORMAT         format
);
```

**Parameters**

object	A pointer to the <code>MseEndatModule</code> object that was created by the <code>MseEndatModuleCreate</code> function
channel	The channel of the encoder to apply the rotary format to
format	The <code>ROTARY_FORMAT</code> to apply

**Return value**

The return value delivers a `MSE_RESPONSE_CODE` representing whether the function call was successful.

**MseEndatModuleGetRotaryFormat**

The MseEndatModuleGetRotaryFormat method is used to return the rotary format that will be applied to the position calculated from the counts in the MseEndatModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetRotaryFormat
(
    MseEndatModulePtr object,
    unsigned short channel,
    ROTARY_FORMAT* format
);
```

**Parameters**

- object            A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- channel           The channel of the encoder that the rotary format will be applied to
- format            A pointer to the location where the rotary format will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleSetDeviceOffset**

The MseEndatModuleSetDeviceOffset method is used to set the offset that will be applied to the position calculated from the counts in the MseEndatModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleSetDeviceOffset
(
    MseEndatModulePtr object,
    unsigned short channel,
    double offset
);
```

**Parameters**

- object            A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- channel           The channel of the encoder to apply the offset to
- offset            The offset to apply

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetDeviceOffset**

The MseEndatModuleGetDeviceOffset method is used to return the offset that will be applied to the position calculated from the counts in the MseEndatModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetDeviceOffset
(
    MseEndatModulePtr object,
    unsigned short channel,
    double* offset
);
```

**Parameters**

- object            A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- channel           The channel of the encoder that the offset will be applied to
- offset            A pointer to the location where the offset will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleSetLatch**

Sets or clears the desired latch for the module chain.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleSetLatch
(
    MseEndatModulePtr    object,
    LATCH_OPTIONS        option,
    LATCH_CHOICE         latchChoice
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
option	Set or reset the module chain latch. Set is only used on a base module. Reset will clear the latch and must be called on the base module first followed by each additional module.
latchChoice	The type of latch to set. Clearing a latch will clear all latches in the base module. Non-base modules only know about being triggered or not and the base module is used to determine which trigger occurred.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetLatches**

Gets the latches that are active. The base module can differentiate between three software latches and two footswitch latches. All other modules only know if they have been latched or not.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetLatches
(
    MseEndatModulePtr    object,
    char*                latchState,
    short                size
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
latchState	A pointer to the location where the latch state will be stored. This is an array that must be large enough to store NUM_LATCH_TYPES. The non-base modules will only utilize the first byte.
size	The size of the latchState array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetModuleErrorState**

Gets the error state of the module. A value of True signifies that there is an error. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetModuleErrorState
(
    MseEndatModulePtr    object,
    bool*                 errorState
);
```

**Parameters**

- object                    A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- errorState                A pointer to the location where the error state will be copied to. A subsequent call to MseEndatModuleGetModuleErrors can be made to get the actual errors.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetModuleErrors**

Gets the actual module errors. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetModuleErrors
(
    MseEndatModulePtr    object,
    long*                 errors,
    double*               ranges,
    short                 size
);
```

**Parameters**

- object                    A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- errors                    A pointer to the location where the errors will be copied to. The errors is a single long that can be masked with the INTEGRITY\_ENUMS to determine which error has occurred.
- ranges                    A pointer to the location where the ranges used to determine an error will be copied to. The ranges is an array that must be large enough to hold NUM\_INTEGRITY\_RANGES.
- size                      The size of the ranges array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetChannelErrorState**

Gets the encoder error state of a channel. An errorState of 1 signifies an error.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetChannelErrorState
(
    MseEndatModulePtr    object,
    bool*                 errorState,
    short                 channel
);
```

**Parameters**

- object                    A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- errorState                A pointer to the location where the error state will be copied to. A subsequent call to MseEndatModuleGetEndatErrors can be made to get the actual errors.
- channel                    The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetEndatErrors**

Gets the encoder error of a channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetEndatErrors
(
    MseEndatModulePtr    object,
    ENDAT_ERROR_RESULT*  errors,
    short                size,
    short                channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
errors	A pointer to the location where the errors will be stored. The errors is an array used to store the status of each type of error. The array must be large enough to hold NUM_ENDAT_ERRORS.
size	The size of the errors array passed in
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetChannelWarningState**

Gets the encoder warning state of a channel. A warningState of 1 signifies a warning.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetChannelWarningState
(
    MseEndatModulePtr    object,
    bool*                warningState,
    short                channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
warningStat	A pointer to the location where the warning state will be copied to. A subsequent call to MseEndatModuleGetEndatWarning can be made to get the actual warnings.
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

### MseEndatModuleGetEndatWarnings

Gets the encoder warnings of a channel.

#### Function

```
MSE_RESPONSE_CODE MseEndatModuleGetEndatWarnings
(
    MseEndatModulePtr    object,
    ENDAT_ERROR_RESULT*  warnings,
    short                size,
    short                channel
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
warnings	A pointer to the location where the warnings will be stored. The warnings is an array used to store the status of each type of warning. The array must be large enough to hold NUM_ENDAT_WARNINGS.
size	The size of the warnings array passed in
channel	The channel to get indexed from 0

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

### MseEndatModuleClearErrors

Clears the module and channel errors and warnings.

#### Function

```
MSE_RESPONSE_CODE MseEndatModuleClearErrors
(
    MseEndatModulePtr    object
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
--------	--

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

### MseEndatModuleGetEncoderName

Gets the name of the encoder attached to a channel.

#### Function

```
MSE_RESPONSE_CODE MseEndatModuleGetEncoderName
(
    MseEndatModulePtr    object,
    char*                encoderName,
    short                channel
);
```

#### Parameters

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
encoderName	A pointer to the location where the encoder name will be stored
channel	The channel to get indexed from 0

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseEndatModuleGetEncoderId**

Gets the ID of the encoder attached to a channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetEncoderId
(
    MseEndatModulePtr    object,
    char*                 encoderId,
    short                 channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
encoderId	A pointer to the location where the encoder ID will be stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetEncoderSerialNumber**

Gets the ID of the encoder attached to a channel.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetEncoderSerialNumber
(
    MseEndatModulePtr    object,
    char*                 serialNumber,
    short                 channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
serialNumber	A pointer to the location where the encoder serial number will be stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleSetLatchDebouncing**

Sets the latch debouncing used on the footswitch lines in the base module.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleSetLatchDebouncing
(
    MseEndatModulePtr    object,
    char                 choice,
    short                 timeMs
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
choice	0 for the first footswitch input and 1 for the second
timeMs	The number of milliseconds to debounce the input. Can be from 0 - 20 ms.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleEnableDiags**

Sets the diagnostic mode for the channels and module. See the Diagnostics Mode section of this document for more information.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleEnableDiags
(
    MseEndatModulePtr    object,
    DIAG_MODE_OPTIONS    choice
);
```

**Parameters**

- object            A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- choice           The desired level of diagnostics

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetDiags**

Gets the EnDat function reserves for an encoder.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetDiags
(
    MseEndatModulePtr    object,
    char*                 diagVals,
    short                 size,
    short                 channel
);
```

**Parameters**

- object            A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
- diagVals          A pointer to the location where the diagnostic values will be stored. Must be large enough to hold ENDAT\_DIAG\_COUNT. The ENDAT\_DIAG enumeration can be used to index into the diagVals array.
- size              The size of the diagVals array
- channel           The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleGetAdcValues**

Gets the voltage and temperature values for the module.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleGetAdcValues
(
    MseEndatModulePtr    object,
    short*               adcVals,
    short                size1
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
adcVals	A pointer to the location where the voltage and temperature values will be stored. Must be large enough to hold ADC_NUM_CHANNELS. The ADC_OPTIONS enumeration can be used to index into the adcVals array.
size	The size of the adcVals array

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseEndatModuleEnableErrorChecking**

Sets whether checking of the function reserves will be done on the specified channel. The channel defaults to enabled on power up of the module and will be checked as long as the channel is populated and the error checking is enabled. The function reserves can be checked with the MseEndatModuleGetDiags function.

**Function**

```
MSE_RESPONSE_CODE MseEndatModuleEnableErrorChecking
(
    MseEndatModulePtr    object,
    const bool            choice,
    const unsigned short  channel
);
```

**Parameters**

object	A pointer to the MseEndatModule object that was created by the MseEndatModuleCreate function
choice	True to enable error checking, false to disable
channel	The channel to enable or disable error checking on indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful

## 2.13 1Vpp methods and functions

C++ methods and the C functions are separated into two sections for easier lookup.

### C++ methods

#### Constructor

```
Mse1VppModule(void);
```

#### initializeModule

The initializeModule method is used to initialize a 1Vpp module. It calls the MseDeviceModule::initializeModule() and then calls the MseDeviceModule::setEncoderInfo() with default values.

#### Method

```
MseResults initializeModule
(
    const char*      mseIpAddress,
    bool             useAsync
);
```

#### Parameters

mseIpAddress	The IP address of the module to initialize
useAsync	True if the MSE should send asynchronous messages (for notification of footswitch pressed)

#### getDiag

The getDiag method is used to return the diagnostic values of the requested channel. The diagVals is an array of doubles that holds the values enumerated in the VPP\_VOLTAGE\_FEEDBACK enumeration. The ideal diagnostic values for the A and B channels should be between -0.5 V and +0.5 V. The A and B values can be plotted in a graph with A as the X and B as the Y in order to show a Lissajou figure. The Lissajou figure should look like a circle with 1Vpp amplitude but the graph will need to have many samples that occur while the encoder is moving. The enableAnalogDiag method must first be called for the required channel before the diagnostics can be retrieved.

#### Method

```
MseResults getDiag
(
    const unsigned   char channel,
    double*          diagVals,
    unsigned char    arrLength
);
```

#### Parameters

channel	The channel to read the diagnostic from
diagVals	The diagnostic values returned from the module for the requested channel
arrLength	The size of the diagVals array passed in (must be >= VPP_VOLTAGE_NUM)

**enableAnalogDiag**

The enableAnalogDiag method is used to set whether the analog diagnostics for the desired channel should be performed or not. This will start collecting the A and B signals for the desired channel and allow the values to be read with the getDiag command. Only 1 channel can be enabled at a time.

**Method**

```
MseResults enableAnalogDiag
(
    const unsigned char    channel,
    const bool             enable
);
```

**Parameters**

channel	The channel to enable or disable the diagnostics of indexed from 0
enable	True to enable, false to disable

**getPositions**

The getPositions method is used to return the positions of the encoders in user units. If the option is set to COUNT\_REQUEST\_LATCHED, the module will clear the latch after the position is read to allow for faster subsequent latching. The rotary formats for each rotary encoder can be set first with the setRotaryFormat method. The default rotary format is ROTARY\_FORMAT\_360. The resulting position is determined by first multiplying the count by the error compensation, then multiplying by the scaling, then computing the position based on the resolution, then converting to the correct UOM, then adding the device offset, and finally formatting the rotary position if necessary. The error compensation is set with the setErrorCompensation method of the base class, it defaults to 1.0. The scaling is set with the setScaling method of the base class, it defaults to 1.0. The resolution is calculated based on the setLineCount or setSignalPeriod values. The UOM is set with setUom, it defaults to UOM\_RAW\_COUNTS and getPositions will not do any position calculations until set. The device offset is set with the setDeviceOffset method of the base class, it defaults to 0.0. The rotary format can be changed with the setRotaryFormat method, it defaults to ROTARY\_FORMAT\_360.

**Method**

```
MseResults getPositions
(
    double*                pos,
    unsigned short         numChannels,
    COUNT_REQUEST_OPTION   option
);
```

**Parameters**

pos	A pointer to a buffer for storing the positions. The positions will be scaled from counts to the correct unit of measurement. The pointer must point to an array that is large enough to store MAX_CHANNELS_PER_MODULE.
numChannels	The number of channels to read and store into the array passed in with pos
option	The type of position to return

**setUom**

The setUom method sets the unit of measurement to the desired value. This value will be used when computing the position when getPositions() is called.

**Method**

```
bool setUom
(
    UOM          uom,
    const unsigned short channel
);
```

**Parameters**

uom                      The unit of measurement to use for computing the position value  
channel                    The channel to set the UOM of indexed from 0

**Return value**

False if the channel is out of range

**setEncoderType**

The setEncoderType method sets the type of encoder to the desired value. This value will be used when computing the position when getPositions() is called.

**Method**

```
bool setEncoderType
(
    ENCODER_TYPES_ENUM encoderType,
    const unsigned short channel
);
```

**Parameters**

encoderType                The type of encoder to use for computing the position value  
channel                    The channel to set the encoder type of indexed from 0

**Return value**

False if the channel is out of range.

**setLineCount**

The setLineCount method sets the line count used for a rotary encoder to the desired value. This value will be used when computing the position when getPositions() is called.

**Method**

```
bool setLineCount
(
    unsigned long   lineCount,
    const unsigned short channel
);
```

**Parameters**

lineCount                  The line count of the rotary encoder to use for computing the position value  
channel                    The channel to set the line count of indexed from 0

**Return value**

False if the channel is out of range.

**getLineCount**

Gets the line count used for a rotary encoder.

**Method**

```
bool getLineCount
(
    unsigned long*    lineCount,
    const unsigned short channel
);
```

**Parameters**

lineCount	A pointer to the location where the line count of the rotary encoder will be stored
channel	The channel to get indexed from 0

**getSignalPeriod**

Gets the signal period used for a linear encoder.

**Method**

```
bool getSignalPeriod
(
    unsigned short*    signalPeriod,
    const unsigned short channel
)
```

**Parameters**

signalPeriod	A pointer to the location where the signal period of the linear encoder will be stored
channel	The channel to get indexed from 0

**setSignalPeriod**

The setSignalPeriod method sets the signal period used for a linear encoder to the desired value. This value will be used when computing the position when getPositions() is called.

**Method**

```
bool setLineCount
(
    unsigned short    signalPeriod,
    const unsigned short channel
);
```

**Parameters**

signalPeriod	The signal period of the linear encoder to use for computing the position value
channel	The channel to set the signal period of indexed from 0

**Return value**

False if the channel is out of range.

**setCountingDirection**

The setCountingDirection method sets the counting direction to the desired value. This value will be used when computing the position when getPositions() is called.

**Method**

```
bool setCountingDirection
(
    bool           positive,
    const unsigned short channel
);
```

**Parameters**

positive                    True if the encoder should count in the direction of the traversal  
channel                    The channel to set the counting direction of indexed from 0

**Return value**

False if the channel is out of range.

**initAbsolutePosition**

The initAbsolutePosition method will tell the module to start obtaining absolute positions utilizing reference marks. This method will cause all readings to be invalid until the reference mark is crossed.

**Method**

```
MseResults initAbsolutePosition
(
    const unsigned char   channel,
    const REFERENCE_MARK_ENUM refMarkType,
    const unsigned short  value
);
```

**Parameters**

channel                    The channel to start absolute referencing for indexed from 0  
refMarkType                The type of reference mark utilized by the encoder. A value of REFERENCE\_MARK\_NONE will not try to obtain an absolute position  
value                      The signal period for a linear encoder and line count for a rotary encoder

**Return value**

The return value delivers a response code representing whether the initAbsolutePosition command was sent correctly.

**isReferencingComplete**

The isReferencingComplete method will check if referencing is complete for the specified channel.

**Method**

```
MseResults isReferencingComplete
(
    const unsigned char   channel,
    bool*                 isComplete
);
```

**Parameters**

channel                    The channel to check to determine if referencing is complete indexed from 0  
isComplete                True if referencing is complete, otherwise false

**Return value**

The return value delivers a response code representing whether the isReferencingComplete command was sent correctly.



**acknowledgeAbsolutePosition**

The `acknowledgeAbsolutePosition` method will send an acknowledge to the module for the specified channel. The acknowledge tells the module that the asynchronous referencing complete has been received for a specific channel. The module will keep sending asynchronous messages every 5 seconds if the acknowledge is not received. This method only needs to be called if the module is in asynchronous mode.

**Method**

```
MseResults acknowledgeAbsolutePosition
(
    const unsigned char channel
);
```

**Parameters**

channel                      The channel to acknowledge reception of the referencing complete indexed from 0

**Return value**

The return value delivers a response code representing whether the command was successful.

**getReferencingState**

Gets the state of the referencing for the desired channel. This method should be called after `isReferencingComplete` is true or before `acknowledgeAbsolutePosition` is sent in order to check if referencing succeeded. Referencing succeeds if the `refMarkState` is `REF_MARK_FINISHED`.

**Method**

```
MseResults getReferencingState
(
    const unsigned char channel,
    REF_MARK_STATE* refMarkState
);
```

**Parameters**

channel                      The channel to get the referencing state indexed from 0  
refMarkState                The state of the referencing. See the `REF_MARK_STATE` enumeration for more information.

**Return value**

The return value delivers a response code representing whether the command was successful.

**getSignalType**

Gets the signal type of the encoder. The signal type is detected when the module is first powered on. The signal type can also be set with `setSignalType`.

**Method**

```
MseResults getSignalType
(
    const unsigned short channel,
    SIGNAL_TYPE* signalType
)
```

**Parameters**

channel                      The channel to get indexed from 0  
signalType                  The type of signal the encoder uses. This will be 1 Vpp or 11  $\mu$ App.

**Return value**

The return value delivers a response code representing whether the command was successful.

**setSignalType**

Sets the signal type of the encoder. The signal type is detected and set when the module is first powered on. The signal can be set with this method in case of an error in the auto-detection.

**Method**

```
MseResults setSignalType
(
    const unsigned short channel,
    const SIGNAL_TYPE signalType
)
```

**Parameters**

channel                    The channel to set indexed from 0  
 signalType                The type of signal the encoder uses. This can be 1 Vpp or 11  $\mu$ App.

**Return value**

The return value delivers a response code representing whether the command was successful.

**detectSignalType**

Detects the signal type of the encoder. The signal type will be set to the detected value. The signal type is first detected and set when the module is first powered on. The signal type can be set explicitly with setSignalType.

**Method**

```
MseResults detectSignalType
(
    const unsigned short channel,
    SIGNAL_TYPE* signalType
)
```

**Parameters**

channel                    The channel to detect indexed from 0  
 signalType                The type of signal detected. This will be 1 Vpp or 11  $\mu$ App.

**Return value**

The return value delivers a response code representing whether the command was successful.

**C Functions**

The 1Vpp C functions can be found in the Mse1VppModuleWrapper.h file.

**Mse1VppModuleCreate**

Creates a Mse1VppModule object and returns a pointer to it.

**Function**

```
Mse1VppModulePtr Mse1VppModuleCreate
(
);
```

**Return value**

The return value delivers a pointer to the Mse1VppModule object that was created.

**Mse1VppModuleDelete**

Deletes the Mse1VppModule object that was passed in.

**Function**

```
void Mse1VppModuleDelete
(
    Mse1VppModulePtr object
);
```

**Parameters**

object                    A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function

**Mse1VppModuleInitialize**

Initializes the Mse1VppModule object that was passed in.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleInitialize
(
    Mse1VppModulePtr    object,
    char*                mseIpAddress,
    bool                 useAsync
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
mseIpAddress	The IP address of the module to initialize
useAsync	Whether to enable asynchronous communication from the module

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetNumChannels**

Gets the number of channels on the module.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetNumChannels
(
    Mse1VppModulePtr    object,
    unsigned short*     numChannels
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
numChannels	A pointer to the location where the number of channels will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetEncoderType**

Sets the type of encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetEncoderType
(
    Mse1VppModulePtr    object,
    ENCODER_TYPES_ENUM  type,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
type	The type of encoder connected to the channel
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetEncoderType**

Gets the type of encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetEncoderType
(
    Mse1VppModulePtr    object,
    ENCODER_TYPES_ENUM* type,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
type	A pointer to the location where the encoder type is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetUom**

Sets the unit of measurement of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetUom
(
    Mse1VppModulePtr    object,
    UOM                  uom,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
uom	The unit of measurement of the encoder connected to the channel
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetUom**

Gets the unit of measurement of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetUom
(
    Mse1VppModulePtr    object,
    UOM*                 uom,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
uom	A pointer to the location where the encoder unit of measurement is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetErrorCompensation**

Sets the linear error compensation of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetErrorCompensation
(
    Mse1VppModulePtr    object,
    double               val,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
val	The error compensation to apply when requesting a position
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetErrorCompensation**

Gets the linear error compensation of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetErrorCompensation
(
    Mse1VppModulePtr    object,
    double*              val,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
val	A pointer to the location where the error compensation is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetScaling**

Sets the device scaling for a specific channel to the desired multiplier. This value will be applied to the position when Mse1VppGetPositions is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetScaling
(
    Mse1VppModulePtr    object,
    const double         val,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
val	The scaling to use for the channel
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetScaling**

Gets the device scaling for a specific channel. This value will be applied to the position when Mse1VppGetPosition is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetScaling
(
    Mse1VppModulePtr    object,
    double*              val,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
val	A pointer to the location where the scaling is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetCountingDirection**

Sets the counting direction of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetCountingDirection
(
    Mse1VppModulePtr    object,
    bool                 isPositive,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
val	True for positive and false for negative
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetCountingDirection**

Gets the counting direction of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetCountingDirection
(
    Mse1VppModulePtr    object,
    bool*                isPositive,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
val	A pointer to the location where the counting direction is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetResolution**

Gets the resolution of the encoder connected to the selected channel. The resolution is computed internally based on the line count or signal period. The resolution is in mm/count for linear encoders and degrees/count for rotary encoders.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetResolution
(
    Mse1VppModulePtr    object,
    double*              resolution,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
resolution	A pointer to the location where the encoder resolution will be stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetCounts**

Gets the encoder counts for all the channels. The position of a linear encoder, in mm, or a rotary encoder, in degrees, can be obtained by multiplying the counts by the resolution value obtained from Mse1VppModuleGetResolution.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetCounts
(
    Mse1VppModulePtr    object,
    unsigned long*       counts,
    short                numChannels,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
counts	A pointer to the location where the counts will be stored. This is an array that must be large enough to store MAX_CHANNELS_PER_MODULE.
numChannels	The size of the counts array passed in
option	Whether to get the latest or the latched counts

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetPositions**

Gets the encoder positions for all the channels. Refer to the 1Vpp getPositions C++ method on page 123 for more information.

The resulting position is determined by first multiplying the count by the error compensation, then multiplying by the scaling, then computing the position based on the resolution, then converting to the correct UOM, then adding the device offset, and finally formatting the rotary position if necessary.

The error compensation is set with the Mse1VppModuleSetErrorCompensation function, it defaults to 1.0. The scaling is set with the Mse1VppModuleSetScaling function, it defaults to 1.0.

The resolution is calculated based on the Mse1VppModuleSetLineCount or Mse1VppModuleSetSignalPeriod values.

The UOM is set with Mse1VppModuleSetUom, it defaults to UOM\_RAW\_COUNTS and Mse1VppModuleGetPositions will not do any position calculations until set.

The device offset is set with the Mse1VppModuleSetDeviceOffset function, it defaults to 0.0.

The rotary format can be changed with the Mse1VppModuleSetRotaryFormat function, it defaults to ROTARY\_FORMAT\_360.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetPositions
(
    Mse1VppModulePtr    object,
    double*              pos,
    short                numChannels,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
pos	A pointer to the location where the positions will be stored. This is an array that must be large enough to store MAX_CHANNELS_PER_MODULE.
numChannels	The size of the counts array passed in
option	Whether to get the latest or the latched positions

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetRotaryFormat**

The Mse1VppModuleSetRotaryFormat method is used to set the rotary format that will be applied to the position calculated from the counts in the Mse1VppModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetRotaryFormat
(
    Mse1VppModulePtr    object,
    unsigned short       channel,
    ROTARY_FORMAT        format
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel of the encoder to apply the rotary format to
format	The ROTARY_FORMAT to apply

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**Mse1VppModuleGetRotaryFormat**

The Mse1VppModuleGetRotaryFormat method is used to return the rotary format that will be applied to the position calculated from the counts in the Mse1VppModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetRotaryFormat
(
    Mse1VppModulePtr    object,
    unsigned short      channel
    ROTARY_FORMAT*      format
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel of the encoder that the rotary format will be applied to
format	A pointer to the location where the rotary format will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetDeviceOffset**

The Mse1VppModuleSetDeviceOffset method is used to set the offset that will be applied to the position calculated from the counts in the Mse1VppModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetDeviceOffset
(
    Mse1VppModulePtr    object,
    unsigned short      channel,
    double               offset
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel of the encoder to apply the offset to
offset	The offset to apply

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetDeviceOffset**

The Mse1VppModuleGetDeviceOffset method is used to return the offset that will be applied to the position calculated from the counts in the Mse1VppModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetDeviceOffset
(
    Mse1VppModulePtr    object,
    unsigned short      channel
    double*              offset
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	he channel of the encoder that the offset will be applied to
offset	A pointer to the location where the offset will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetLatch**

Sets or clears the desired latch for the module chain.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetLatch
(
    Mse1VppModulePtr    object,
    LATCH_OPTIONS        option,
    LATCH_CHOICE         latchChoice
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
option	Set or reset the module chain latch. Set is only used on a base module. Reset will clear the latch and must be called on the base module first followed by each additional module.
latchChoice	The type of latch to set. Clearing a latch will clear all latches in the base module. Non-base modules only know about being triggered or not and the base module is used to determine which trigger occurred.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetLatches**

Gets the latches that are active. The base module can differentiate between three software latches and two footswitch latches. All other modules only know if they have been latched or not.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetLatches
(
    Mse1VppModulePtr    object,
    char*                latchState,
    short                size
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
latchState	A pointer to the location where the latch state will be stored. This is an array that must be large enough to store NUM_LATCH_TYPES. The non-base modules will only utilize the first byte.
size	The size of the latchState array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetModuleErrorState**

Gets the error state of the module. A value of True signifies that there is an error. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetModuleErrorState
(
    Mse1VppModulePtr    object,
    bool*                errorState
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
errorState	A pointer to the location where the error state will be copied to. A subsequent call to Mse1VppModuleGetModuleErrors can be made to get the actual errors.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetModuleErrors**

Gets the actual module errors. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetModuleErrors.
(
    Mse1VppModulePtr    object,
    long*               errors,
    double*             ranges,
    short               size
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
errors	A pointer to the location where the errors will be copied to. The errors is a single long that can be masked with the INTEGRITY_ENUMS to determine which error has occurred.
ranges	A pointer to the location where the ranges used to determine an error will be copied to. The ranges is an array that must be large enough to hold NUM_INTEGRITY_RANGES.
size	The size of the ranges array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetChannelErrorState**

Gets the encoder error state of a channel. An errorState of 1 signifies an error. The encoder errors are enumerated in the COUNTER\_STATUS.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetChannelErrorState
(
    Mse1VppModulePtr    object,
    bool*               errorState,
    short               channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
errorState	A pointer to the location where the error state will be copied to. A subsequent call to Mse1VppModuleGetChannelStatus can be made to get the actual errors.
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetChannelStatus**

Gets the encoder error status of a channel. The COUNTER\_STATUS can be masked with status to determine the channel status.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetChannelStatus
(
    Mse1VppModulePtr    object,
    char*               status,
    short               channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
status	A pointer to the location where the channel status will be copied to.
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleClearErrors**

Clears the module and channel errors and warnings.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleClearErrors
(
    Mse1VppModulePtr    object
);
```

**Parameters**

object                    A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetLatchDebouncing**

Sets the latch debouncing used on the footswitch lines in the base module.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetLatchDebouncing
(
    Mse1VppModulePtr    object,
    char                 choice,
    short                timeMs
);
```

**Parameters**

object                    A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function

choice                    0 for the first footswitch input and 1 for the second

timeMs                    The number of milliseconds to debounce the input. Can be from 0 - 20 ms.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleEnableDiags**

Sets the diagnostic mode for the channels and module. See the Diagnostics Mode section of this document for more information.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleEnableDiags
(
    Mse1VppModulePtr    object,
    DIAG_MODE_OPTIONS    choice
);
```

**Parameters**

object                    A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function

choice                    The desired level of diagnostics

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetAdcValues**

Gets the voltage and temperature values for the module.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetAdcValues
(
    Mse1VppModulePtr    object,
    short*               adcVals,
    short                size
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
adcVals	A pointer to the location where the voltage and temperature values will be stored. Must be large enough to hold ADC_NUM_CHANNELS. The ADC_OPTIONS enumeration can be used to index into the adcVals array.
size	The size of the adcVals array

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleEnableAnalogDiag**

Sets the diagnostic mode for the analog A and B channels of an encoder. The module can only monitor the diagnostics of 1 channel at a time.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleEnableAnalogDiag
(
    Mse1VppModulePtr    object,
    bool                 enable,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
enable	True to enable monitoring of the diagnostics
channel	The channel to monitor indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetAnalogDiag**

Gets the A and B channel data of an encoder.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetAnalogDiag
(
    Mse1VppModulePtr    object,
    double*              diagVals,
    short                size,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
diagVals	A pointer to the location where the A and B voltage data values will be stored. Must be large enough to hold VPP_VOLTAGE_NUM
size	The size of the diagVals array
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetLineCount**

Sets the line count of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetLineCount
(
    Mse1VppModulePtr    object,
    long                 lineCount,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
lineCount	The line count of a rotary encoder
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetLineCount**

Gets the line count of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetLineCount
(
    Mse1VppModulePtr    object,
    long*                lineCount,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
lineCount	A pointer to the location where the line count is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleSetSignalPeriod**

Sets the signal period of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleSetSignalPeriod
(
    Mse1VppModulePtr    object,
    short                signalPeriod,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
signalPeriod	The signal period of a linear encoder
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetSignalPeriod**

Gets the signal period of the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetSignalPeriod
(
    Mse1VppModulePtr    object,
    short*              signalPeriod,
    short               channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
signalPeriod	A pointer to the location where the signal period is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleStartReferencing**

Starts the referencing for the encoder connected to the selected channel.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleStartReferencing
(
    Mse1VppModulePtr    object,
    short               channel,
    REFERENCE_MARK_ENUM refMarkType,
    unsigned short      value
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel to start referencing on indexed from 0
refMarkType	The type of referencing used by the encoder
value	The signal period or line count used for the referencing

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleGetReferencingComplete**

Gets whether the referencing is complete for the specified encoder.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleGetReferencingComplete
(
    Mse1VppModulePtr    object,
    short               channel,
    bool*               isComplete
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel to get indexed from 0
isComplete	A pointer to the location where the status of the referencing is stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleAcknowledgeAbsolutePosition**

Sends an acknowledge to the module informing it that the client received the asynchronous referencing message.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleAcknowledgeAbsolutePosition
(
    Mse1VppModulePtr    object,
    short                channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel to acknowledge the referencing of indexed from 0

**Mse1VppModuleGetReferencingState**

Gets the state of the referencing for the desired channel. This method should be called after isReferencingComplete is true or before acknowledgeAbsolutePosition is sent in order to check if referencing succeeded. Referencing succeeds if the refMarkState is REF\_MARK\_FINISHED.

**Method**

```
MseResults Mse1VppModuleGetReferencingState
(
    Mse1VppModulePtr    object,
    const unsigned char channel,
    REF_MARK_STATE*     refMarkState
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel to get the referencing state indexed from 0
refMarkState	A pointer to the location where the state of the referencing will be stored. See the REF_MARK_STATE enumeration for more information.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppModuleEnableErrorChecking**

Sets whether error checking will be done on the specified channel. The channel defaults to enabled on power up of the module and will be checked as long as the channel is populated and the error checking is enabled. The channel status can be checked with the Mse1VppModuleGetChannelStatus when error checking is enabled.

**Function**

```
MSE_RESPONSE_CODE Mse1VppModuleEnableErrorChecking
(
    Mse1VppModulePtr    object,
    const bool           choice,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
choice	True to enable error checking, false to disable
channel	The channel to enable or disable error checking on indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**Mse1VppGetSignalType**

Gets the signal type of the encoder. The signal type is detected when the module is first powered on. The signal type can also be set with setSignalType.

**Method**

```
MSE_RESPONSE_CODE Mse1VppGetSignalType
(
    Mse1VppModulePtr    object,
    const unsigned short channel,
    SIGNAL_TYPE*         signalType
)
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel to get indexed from 0
signalType	The type of signal the encoder uses. This will be 1 Vpp or 11 µApp.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppSetSignalType**

Sets the signal type of the encoder. The signal type is detected and set when the module is first powered on. The signal can be set with this function in case of an error in the auto-detection.

**Method**

```
MSE_RESPONSE_CODE Mse1VppSetSignalType
(
    Mse1VppModulePtr    object,
    const unsigned short channel,
    const SIGNAL_TYPE    signalType
)
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel to set indexed from 0
signalType	The type of signal the encoder uses. This can be 1 Vpp or 11 µApp.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**Mse1VppDetectSignalType**

Detects the signal type of the encoder. The signal type will be set to the detected value. The signal type is first detected and set when the module is first powered on. The signal type can be set explicitly with setSignalType.

**Method**

```
MSE_RESPONSE_CODE Mse1VppDetectSignalType
(
    Mse1VppModulePtr    object,
    const unsigned short channel,
    SIGNAL_TYPE*         signalType
)
```

**Parameters**

object	A pointer to the Mse1VppModule object that was created by the Mse1VppModuleCreate function
channel	The channel to detect indexed from 0
signalType	The type of signal detected. This will be 1 Vpp or 11 µApp.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

## 2.14 I/O methods and functions

C++ methods and the C functions are separated into two sections for easier lookup.

### C++ Methods

#### Constructor

```
MseIoModule(void);
```

#### initializeModule

The initializeModule method is used to initialize an I/O module. It calls the MseModule::initializeModule() and then calls setModuleInitialized (true) if the initialization passes. The number of channels is set to NUM\_MSE\_IO\_INPUTS + NUM\_MSE\_IO\_OUTPUTS.

#### Method

```
MseResults initializeModule
(
    const char*      mseIpAddress,
    bool             useAsync
);
```

#### Parameters

mseIpAddress	The IP address of the module to initialize
useAsync	True if the MSE should send asynchronous messages

#### setOutputs

The setOutputs method is used to set the output of the I/O module to the desired values.

#### Method

```
MseResults setOutputs
(
    const unsigned char* output
);
```

#### Parameters

output	A pointer to an unsigned char, each output in the I/O module will be set to the corresponding bit in the char. The char can only set output 1,2,3 and 4 in the module.
--------	--

#### Return value

MseResults	A response code representing whether the setOutputs command was sent
------------	--

**setOutput**

The setOutput method is used to set an individual output of the I/O module to the desired value.

**Method**

```
MseResults setOutput
(
    unsigned char    pin,
    bool             val
);
```

**Parameters**

pin	An unsigned char representing the pin to set to the desired value. Acceptable values are 1 to NUM_MSE_IO_OUTPUTS.
val	A boolean representing whether to set the pin high or low

**Return value**

MseResults	A response code representing whether the setOutput command was sent
------------	---

**getOutputs**

The getOutputs method is used to get the outputs of the I/O module.

**Method**

```
MseResults getOutputs
(
    unsigned char*   outputs
);
```

**Parameters**

outputs	A character where the output settings of the I/O module will be stored. Bit 0 will hold output 1 and bit 3 will hold output 4.
---------	--

**Return value**

MseResults	A response code representing whether the getOutputs command was completed
------------	---

**getInputs**

The getInputs method is used to get the inputs of the I/O module.

**Method**

```
MseResults getInputs
(
    unsigned char*    inputs
);
```

**Parameters**

inputs                      A character where the input settings of the I/O module will be stored. Bit 0 will hold input 1 and bit 3 will hold input 4.

**Return value**

MseResults                      A response code representing whether the getInputs command was completed

**getIO**

The getIO method is used to get the inputs and outputs of the I/O module. If the option is set to COUNT\_REQUEST\_LATCHED, the module will clear the latch after the I/O is read to allow for faster subsequent latching.

**Method**

```
MseResults getIO
(
    unsigned char*    inputs,
    unsigned char*    outputs,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

inputs                      A character where the input settings of the I/O module will be stored. Bit 0 will hold input 1 and bit 3 will hold input 4.

outputs                      A character where the output settings of the I/O module will be stored. Bit 0 will hold output 1 and bit 3 will hold output 4.

option                      The type of position to return

**Return value**

MseResults                      A response code representing whether the getIO command was completed

## C Functions

The I/O C functions can be found in the MseloModuleWrapper.h file.

### MseloModuleCreate

Creates a MseloModule object and returns a pointer to it.

#### Function

```
MseIoModulePtr MseIoModuleCreate
(
);
```

#### Return value

The return value delivers a pointer to the MseloModule object that was created.

### MseloModuleDelete

Deletes the MseloModule object that was passed in.

#### Function

```
void MseIoModuleDelete
(
    MseIoModulePtr    object
);
```

#### Parameters

object                    A pointer to the MseloModule object that was created by the MseloModuleCreate function

### MseloModuleInitialize

Initializes the MseloModule object that was passed in.

#### Function

```
MSE_RESPONSE_CODE MseIoModuleInitialize
(
    MseIoModulePtr    object,
    char*              mseIpAddress,
    bool               useAsync
);
```

#### Parameters

object                    A pointer to the MseloModule object that was created by the MseloModuleCreate function  
mseIpAddress              The IP address of the module to initialize  
useAsync                  Whether to enable asynchronous communication from the module

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleGetNumChannels**

Gets the number of channels on the module.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleGetNumChannels
(
    MseIoModulePtr    object,
    unsigned short*   numChannels
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
numChannels	A pointer to the location where the number of channels will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleGetModuleErrorState**

Gets the error state of the module. A value of True signifies that there is an error. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleGetModuleErrorState
(
    MseIoModulePtr    object,
    bool*             errorState
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
errorState	A pointer to the location where the error state will be copied to. A subsequent call to MseloModuleGetModuleErrors can be made to get the actual errors.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleGetModuleErrors**

Gets the actual module errors. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleGetModuleErrors
(
    MseIoModulePtr    object,
    long*             errors,
    double*           ranges,
    short             size
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
errors	A pointer to the location where the errors will be copied to. The errors is a single long that can be masked with the INTEGRITY_ENUMS to determine which error has occurred.
ranges	A pointer to the location where the ranges used to determine an error will be copied to. The ranges is an array that must be large enough to hold NUM_INTEGRITY_RANGES.
size	The size of the ranges array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleGetAdcValues**

Gets the voltage and temperature values for the module.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleGetAdcValues
(
    MseIoModulePtr    object,
    short*            adcVals,
    short              size
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
adcVals	A pointer to the location where the voltage and temperature values will be stored. Must be large enough to hold ADC_NUM_CHANNELS. The ADC_OPTIONS enumeration can be used to index into the adcVals array.
size	The size of the adcVals array

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleClearErrors**

Clears the module and channel errors and warnings.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleClearErrors
(
    MseIoModulePtr    object
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
--------	--

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleSetOutputs**

Sets the outputs of the module.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleSetOutputs
(
    MseIoModulePtr    object,
    char              output
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
output	The outputs to set masked into a single byte

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleSetOutput**

Sets a single output of the module.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleSetOutput
(
    MseIoModulePtr    object,
    short              pin,
    short              val
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
pin	The output to set. Can be 1 - 4.
val	The value to set the output to. Can be 0 or 1.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleGetOutputs**

Gets the outputs of the module.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleGetOutputs
(
    MseIoModulePtr    object,
    char*              outputs
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
outputs	A pointer to the location where the outputs will be stored. Bit 0 will hold output 1 and bit 3 will hold output 4.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleGetInputs**

Gets the inputs of the module.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleGetInputs
(
    MseIoModulePtr    object,
    char*              inputs
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
inputs	A pointer to the location where the inputs will be stored. Bit 0 will hold input 1 and bit 3 will hold input 4.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseloModuleGetIO**

Gets the inputs and outputs of the module.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleGetInputs
(
    MseIoModulePtr    object,
    char*              inputs,
    char*              outputs,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
inputs	A pointer to the location where the inputs will be stored. Bit 0 will hold input 1 and bit 3 will hold input 4.
outputs	A pointer to the location where the outputs will be stored. Bit 0 will hold output 1 and bit 3 will hold output 4.
option	Whether to get the latest or latched inputs.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleGetLatch**

Gets whether the latch is active.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleGetLatch
(
    MseIoModulePtr    object,
    bool*              isLatched
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
isLatched	A pointer to the location where the latch state will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseloModuleClearLatch**

Clears the latch. The user must make sure that the base module latches are cleared first or else the latch will immediately trigger again.

**Function**

```
MSE_RESPONSE_CODE MseIoModuleClearLatch
(
    MseIoModulePtr    object
);
```

**Parameters**

object	A pointer to the MseloModule object that was created by the MseloModuleCreate function
--------	--

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

## 2.15 Pneumatic methods and functions

C++ methods and the C functions are separated into two sections for easier lookup.

### C++ Methods

#### Constructor

```
MsePneumaticModule(void);
```

#### initializeModule

The initializeModule method is used to initialize a pneumatic module. It calls the MseModule::initializeModule() and then calls setModuleInitialized (true) if the initialization passes. The number of channels is set to 1.

#### Method

```
MseResults initializeModule
(
    const char*      mseIpAddress,
    bool             useAsync
);
```

#### Parameters

mseIpAddress	The IP address of the module to initialize
useAsync	True if the MSE should send asynchronous messages

#### Return value

MseResults	A response code representing whether the method succeeded
------------	---

#### getOutput

The getOutput method is used to get the output of the pneumatic module. If the option is set to COUNT\_REQUEST\_LATCHED, the module will clear the latch after the output is read to allow for faster subsequent latching.

#### Method

```
MseResults getOutput
(
    unsigned char*   output,
    COUNT_REQUEST_OPTION option
);
```

#### Parameters

output	A character where the output setting of the pneumatic module will be stored.
option	The type of output to return.

#### Return value

MseResults	A response code representing whether the getOutput command was completed
------------	--

#### setOutput

The setOutput method is used to set the output of the pneumatic module to the desired value.

#### Method

```
MseResults setOutput
(
    bool             val
);
```

#### Parameters

val	A boolean representing whether to set the output high or low
-----	--

#### Return value

MseResults	A response code representing whether the setOutput command was sent
------------	---

## C Functions

The pneumatic C functions can be found in the MsePneumaticModuleWrapper.h file.

### MsePneumaticModuleCreate

Creates a MsePneumaticModule object and returns a pointer to it.

#### Function

```
MsePneumaticModulePtr MsePneumaticModuleCreate
(
);
```

#### Return value

The return value delivers a pointer to the MsePneumaticModule object that was created.

### MsePneumaticModuleDelete

Deletes the MsePneumaticModule object that was passed in.

#### Function

```
void MsePneumaticModuleDelete
(
    MsePneumaticModulePtr object
);
```

#### Parameters

object	A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function
--------	--

### MsePneumaticModuleInitialize

Initializes the MsePneumaticModule object that was passed in.

#### Function

```
MSE_RESPONSE_CODE MsePneumaticModuleInitialize
(
    MsePneumaticModulePtr object,
    char* mselpAddress,
    bool useAsync
);
```

#### Parameters

object	A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function
mselpAddress	The IP address of the module to initialize
useAsync	Whether to enable asynchronous communication from the module

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MsePneumaticModuleGetNumChannels**

Gets the number of channels on the module.

**Function**

```
MSE_RESPONSE_CODE MsePneumaticModuleGetNumChannels
(
    MsePneumaticModulePtr  object,
    unsigned short*        numChannels
);
```

**Parameters**

object	A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function
numChannels	A pointer to the location where the number of channels will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MsePneumaticModuleGetModuleErrorState**

Gets the error state of the module. A value of True signifies that there is an error. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MsePneumaticModuleGetModuleErrorState
(
    MsePneumaticModulePtr  object,
    bool*                   errorState
);
```

**Parameters**

object	A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function
errorState	A pointer to the location where the error state will be copied to. A subsequent call to MsePneumaticModuleGetModuleErrors can be made to get the actual errors.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MsePneumaticModuleGetModuleErrors**

Gets the actual module errors. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MsePneumaticModuleGetModuleErrors
(
    MsePneumaticModulePtr  object,
    long*                   errors,
    double*                 ranges,
    short                   size
);
```

**Parameters**

object	A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function
errors	A pointer to the location where the errors will be copied to. The errors is a single long that can be masked with the INTEGRITY_ENUMS to determine which error has occurred.
ranges	A pointer to the location where the ranges used to determine an error will be copied to. The ranges is an array that must be large enough to hold NUM_INTEGRITY_RANGES.
size	The size of the ranges array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MsePneumaticModuleGetAdcValues**

Gets the voltage and temperature values for the module.

**Function**

```
MSE_RESPONSE_CODE MsePneumaticModuleGetAdcValues
(
    MsePneumaticModulePtr  object,
    short*                  adcVals,
    short                   size
);
```

**Parameters**

object	A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function
adcVals	A pointer to the location where the voltage and temperature values will be stored. Must be large enough to hold ADC_NUM_CHANNELS. The ADC_OPTIONS enumeration can be used to index into the adcVals array.
size	The size of the adcVals array

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MsePneumaticModuleClearErrors**

Clears the module and channel errors and warnings.

**Function**

```
MSE_RESPONSE_CODE MsePneumaticModuleClearErrors
(
    MsePneumaticModulePtr object
);
```

**Parameters**

object                    A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MsePneumaticModuleSetOutput**

Sets the output of the module.

**Function**

```
MSE_RESPONSE_CODE MsePneumaticModuleSetOutput
(
    MsePneumaticModulePtr object,
    short val
);
```

**Parameters**

object                    A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function

val                        1 to enable and 0 to disable

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MsePneumaticModuleGetOutput**

Gets the output of the module.

**Function**

```
MSE_RESPONSE_CODE MsePneumaticModuleGetOutput
(
    MsePneumaticModulePtr object,
    char outputVal,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

object                    A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function

outputVal                A pointer to the location where the output will be stored

option                    Whether to get the latest or latched output

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MsePneumaticModuleGetLatch**

Gets whether the latch is active.

**Function**

```
MSE_RESPONSE_CODE MsePneumaticModuleGetLatch
(
    MsePneumaticModulePtr object,
    bool* isLatched
);
```

**Parameters**

object	A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function
isLatched	A pointer to the location where the latch state will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MsePneumaticModuleClearLatch**

Clears the latch. The user must make sure that the base module latches are cleared first or else the latch will immediately trigger again.

**Function**

```
MSE_RESPONSE_CODE MsePneumaticModuleClearLatch
(
    MsePneumaticModulePtr object
);
```

**Parameters**

object	A pointer to the MsePneumaticModule object that was created by the MsePneumaticModuleCreate function
--------	--

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

## 2.16 LVDT methods and functions

C++ methods and the C functions are separated into two sections for easier lookup.

### C++ Methods

#### Constructor

```
MseLvdtModule(void);
```

#### initializeModule

The initializeModule method is used to initialize an LVDT module. It calls the MseModule::initializeModule() and then calls setModuleInitialized(true) if the initialization passes. The number of channels is set to 8.

#### Method

```
MseResults initializeModule
(
    const char*      mseIpAddress,
    bool             useAsync
);
```

#### Parameters

mseIpAddress	The IP address of the module to initialize
useAsync	True if the MSE should send asynchronous messages

#### Return value

MseResults	A response code representing whether the method succeeded.
------------	--

#### getUom

The getUom method is used to get the unit of measurement for a specified channel. This method is not used. The setResolution method is used to determine the UOM.

#### Method

```
bool getUom
(
    LVDT_UOM*      uom,
    const unsigned short channel
);
```

#### Parameters

uom	The location the units of measure for the specified channel will be stored
channel	The channel to get indexed from 0

#### Return value

bool	True if the method succeeds, false if the channel is out of range or the pointer is NULL.
------	---



**setUom**

The setUom method is used to set the unit of measurement for a specified channel. This method is not used. The setResolution method is used to determine the UOM.

**Method**

```
bool setUom
(
    LVDT_UOM          uom,
    const unsigned short channel
);
```

**Parameters**

uom                    The units of measure for the specified channel  
channel                The channel to set indexed from 0

**Return value**

bool                    True if the method succeeds, false if the channel is out of range.

**getExcitationValues**

The getExcitationValues method is used to get the excitation voltage and frequency used for the primary windings of the LVDT sensors. The frequency is in kHz.

**Method**

```
MseResults getExcitationValues
(
    double*           voltage,
    double*           frequency
);
```

**Parameters**

voltage                The location where the primary winding excitation voltage will be stored  
frequency              The location where the primary winding excitation frequency will be stored

**Return value**

MseResults            A response code representing whether the method succeeded.

**setExcitationVoltage**

The setExcitationVoltage method is used to set the excitation voltage used for the primary windings of the LVDT sensors.

**Method**

```
MseResults setExcitationVoltage
(
    const double*     voltage
);
```

**Parameters**

voltage                The desired primary winding excitation voltage. The voltage can be from LVDT\_EXCITATION\_VOLTAGE\_MIN\_VPP to LVDT\_EXCITATION\_VOLTAGE\_MAX\_VPP.

**Return value**

MseResults            A response code representing whether the method succeeded.

**setExcitationFrequency**

The setExcitationFrequency method is used to set the excitation frequency used for the primary windings of the LVDT sensors.

**Method**

```
MseResults setExcitationFrequency
(
    const double* frequency
);
```

**Parameters**

frequency The desired primary winding excitation frequency in kHz. The frequency can be from LVDT\_EXCITATION\_FREQUENCY\_MIN\_KHZ to LVDT\_EXCITATION\_FREQUENCY\_MAX\_KHZ.

**Return value**

MseResults A response code representing whether the method succeeded.

**getVoltage**

The getVoltage method is used to get the LVDT sensor count and voltage value that is obtained from the ADC for the desired channel. The count is useful because you can convert this to a position by multiplying it by the resolution of the sensor. The voltage is an approximate value of the reading from the ADC and is informational only.

**Method**

```
MseResults getVoltage
(
    unsigned short channel,
    double* channelVoltage,
    long* counts
);
```

**Parameters**

channel The channel to get the count and voltage of indexed from 0  
 channelVoltage The location where the LVDT sensor output voltage will be stored  
 counts The location where the LVDT sensor output counts will be stored

**Return value**

MseResults A response code representing whether the method succeeded.

**getPositions**

The getPositions method is used to get the position of a specific LVDT sensor in user units. The setResolution method must be called before calling this method.

The resulting position is determined by first multiplying the count \* resolution, then multiplying by the scaling, then adding the device offset.

The scaling is set with the setScaling method of the base class, it defaults to 1.0.

The device offset is set with the setDeviceOffset method of the base class, it defaults to 0.0.

**Method**

```
MseResults getPositions
(
    double* pos,
    unsigned short numChannels,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

pos The location where the position will be stored  
 numChannels The number of doubles in the pos array passed in  
 option Whether to read live or latched positions

**Return value**

MseResults A response code representing whether the method succeeded.

**setChannelPresence**

The setChannelPresence method is used to set whether an LVDT sensor is attached to a specific channel.

**Method**

```
MseResults setChannelPresence
(
    unsigned char    isConnected,
    unsigned char    channel
);
```

**Parameters**

isConnected	Set to 1 if an LVDT is connected to the channel
channel	The channel to set indexed from 0

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**getChannelPresence**

The getChannelPresence method is used to get whether an LVDT sensor is attached to a specific channel.

**Method**

```
MseResults getChannelPresence
(
    unsigned char*    isConnected,
    unsigned char    channel
);
```

**Parameters**

isConnected	1 signifies that an LVDT sensor is connected to the channel
channel	The channel to get indexed from 0

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**getResolution**

The getResolution method is used to get the resolution used when converting an LVDT sensor's count value to a position.

**Method**

```
MseResults getResolution
(
    unsigned short    channelNum,
    double*           resolution
);
```

**Parameters**

channelNum	The channel to get the resolution of indexed from 0
resolution	The location where the resolution will be stored

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**setResolution**

The setResolution method is used to set the resolution used when converting an LVDT sensor's count value to a position.

**Method**

```
MseResults setResolution
(
    unsigned short    channelNum,
    const double      resolution
);
```

**Parameters**

channelNum      The channel to set the resolution of indexed from 0  
 resolution      The resolution to use when converting an LVDT sensor's count value to a position

**Return value**

MseResults      A response code representing whether the method succeeded.

**setDiagnosticsEnabled**

The setDiagnosticsEnabled method is used to set the sensor group to monitor based on the LVDT\_UPDATE\_CHOICES enumeration passed in.

**Method**

```
MseResults setDiagnosticsEnabled
(
    const LVDT_UPDATE_CHOICES choice
);
```

**Parameters**

choice            The desired sensor group to monitor

**Return value**

MseResults      A response code representing whether the method succeeded.

**getFpgaRevision**

The getFpgaRevision method is used to get the revision of the FPGA code used in the module. The revision is in the form 0xMMmm, where MM is the major number and mm is the minor number (e.g. 0x0100 is V1.00).

**Method**

```
MseResults getFpgaRevision
(
    unsigned short*    revision
);
```

**Parameters**

revision          A pointer to the location where the FPGA revision will be stored

**Return value**

MseResults      A response code representing whether the method succeeded.

**setOversampling**

Sets the oversampling used when reading the LVDT sensor count from the ADC. A smaller value will take less time to obtain a reading but will have more noise.

**Method**

```
MseResults setOversampling
(
    const LVDT_OVERSAMPLING_CHOICES& choice
);
```

**Parameters**

choice            One of the LVDT\_OVERSAMPLING\_CHOICES enumerations

**Return value**

MseResults      A response code representing whether the method succeeded.

**getSensorGain**

The getSensorGain method is used to get the gain code used for a given sensor.

**Method**

```
MseResults getSensorGain
(
    const unsigned short channelNum,
    unsigned char* gainCode
);
```

**Parameters**

channelNum	The channel number to get the gain code of indexed from 0
gainCode	A pointer to the location where the gain code for the desired channel will be stored

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**setSensorGain**

The setSensorGain method is used to set the gain code used for a given sensor.

**Method**

```
MseResults setSensorGain
(
    const unsigned short channelNum,
    const unsigned char gainCode
);
```

**Parameters**

channelNum	The channel number to get the gain code of indexed from 0
gainCode	The gain code for the desired channel

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**teachSensorGain**

The teachSensorGain method is used to send a command to the module to adjust the gain until a value is found that allows for the greatest range.

**Method**

```
MseResults teachSensorGain
(
    const unsigned short channelNum
);
```

**Parameters**

channelNum	The channel number to teach the gain code for indexed from 0
------------	--

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**getTeachSensorGainFinished**

The getTeachSensorGainFinished method is used to get whether the teach has finished for a sensor's position gain. The teach is started when the teachSensorGain method is called.

**Method**

```
MseResults getTeachSensorGainFinished
(
    const unsigned short channelNum,
    unsigned short* gainCode
);
```

**Parameters**

- channelNum            The channel number to get the teach completion status for indexed from 0
- gainCode             The resulting gain code from the teach. This will be 0 if the teach has not been completed.

**Return value**

MseResults            A response code representing whether the method succeeded.

**C Functions**

The LVDT module C functions can be found in the MseLvdModuleWrapper.h file.

**MseLvdModuleCreate**

Creates a MseLvdModule object and returns a pointer to it.

**Function**

```
MseLvdModulePtr MseLvdModuleCreate
(
);
```

**Return value**

The return value delivers a pointer to the MseLvdModule object that was created.

**MseLvdModuleDelete**

Deletes the MseLvdModule object that was passed in.

**Function**

```
void MseLvdModuleDelete
(
    MseLvdModulePtr object
);
```

**Parameters**

- object                A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function

**MseLvdtModuleInitialize**

Initializes the MseLvdtModule object that was passed in.

**Function**

```
MSE_RESPONSE_CODE MseLvdtModuleInitialize
(
    MseLvdtModulePtr    object,
    char*                mseIpAddress,
    bool                 useAsync
);
```

**Parameters**

object	A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function
mseIpAddress	The IP address of the module to initialize
useAsync	Whether to enable asynchronous communication from the module

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtModuleGetNumChannels**

Gets the number of channels on the module.

**Function**

```
MSE_RESPONSE_CODE MseLvdtModuleGetNumChannels
(
    MseLvdtModulePtr    object,
    unsigned short*     numChannels
);
```

**Parameters**

object	A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function
numChannels	A pointer to the location where the number of channels will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtModuleSetUom**

Sets the unit of measurement to use for the specified channel when requesting position. This function is not used. The MseLvdtModuleSetResolution function is used to determine the UOM.

**Function**

```
MSE_RESPONSE_CODE MseLvdtModuleSetUom
(
    MseLvdtModulePtr    object,
    LVDT_UOM            uom,
    short                channel
);
```

**Parameters**

object	A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function
uom	The unit of measurement to use when requesting position for the specified channel
channel	The channel to set the unit of measurement of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtModuleGetUom**

Gets the unit of measurement used for the specified channel when requesting position. This function is not used. The MseLvdtModuleSetResolution function is used to determine the UOM.

**Function**

```
MSE_RESPONSE_CODE MseLvdtModuleGetUom
(
    MseLvdtModulePtr    object,
    LVDT_UOM*           uom,
    short                channel
);
```

**Parameters**

object	A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function
uom	A pointer to the location where the unit of measurement will be stored
channel	The channel to get the unit of measurement of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtModuleSetResolution**

Sets the resolution to use for the specified channel when requesting position.

**Function**

```
MSE_RESPONSE_CODE MseLvdtModuleSetResolution
(
    MseLvdtModulePtr    object,
    const double         resolution,
    short                channel
);
```

**Parameters**

object	A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function
resolution	The resolution to use when requesting position for the specified channel
channel	The channel to set the resolution of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtModuleGetResolution**

Gets the resolution used for the specified channel when requesting position.

**Function**

```
MSE_RESPONSE_CODE MseLvdtModuleGetResolution
(
    MseLvdtModulePtr    object,
    double*              resolution,
    short                channel
);
```

**Parameters**

object	A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function
resolution	A pointer to the location where the resolution will be stored
channel	The channel to get the resolution of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseLvdModuleSetScaling**

Sets the device scaling for a specific channel to the desired multiplier. This value will be applied to the position when MseLvdModuleGetPosition is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleSetScaling
(
    MseLvdModulePtr    object,
    const double        val,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
val	The scaling to use for the channel
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleGetScaling**

Gets the device scaling for a specific channel. This value will be applied to the position when MseLvdModuleGetPosition is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleGetScaling
(
    MseLvdModulePtr    object,
    double*            val,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
val	A pointer to the location where the scaling is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleGetLatch**

Gets whether the latch is active.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleGetLatch
(
    MseLvdModulePtr    object,
    bool*              isLatched
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
isLatched	A pointer to the location where the latch state will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtModuleClearLatch**

Clears the latch. The user must make sure that the base module latch is cleared first or the latch will immediately trigger again.

**Function**

```
MSE_RESPONSE_CODE MseLvdtModuleClearLatch
(
    MseLvdtModulePtr    object
);
```

**Parameters**

object                      A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtModuleGetModuleErrorState**

Gets the error state of the module. A value of True signifies that there is an error. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseLvdtModuleGetModuleErrorState
(
    MseLvdtModulePtr    object,
    bool*                errorState
);
```

**Parameters**

object                      A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function

errorState                  A pointer to the location where the error state will be stored. A subsequent call to MseLvdtModuleGetModuleErrors can be made to get the actual errors.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtModuleGetModuleErrors**

Gets the errors specific to the module. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseLvdtModuleGetModuleErrors
(
    MseLvdtModulePtr    object,
    long*                errors,
    double*              ranges,
    short                size
);
```

**Parameters**

object                      A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function

errors                      A pointer to the location where the errors state will stored. The errors is a single long that can be masked with the INTEGRITY\_ENUMS to determine which error has occurred.

ranges                      A pointer to the location where the ranges used to determine an error will be stored. The ranges is an array that must be large enough to hold NUM\_INTEGRITY\_RANGES.

size                         The size of the ranges array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleEnableDiags**

Sets the diagnostic mode for the channels and module. Refer to Diagnostic modes on page 231 for more information.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleEnableDiags
(
    MseLvdModulePtr    object,
    DIAG_MODE_OPTIONS  choice
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
choice	The desired level of diagnostics

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleGetAdcValues**

Gets the voltage and temperature values for the module.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleGetAdcValues
(
    MseLvdModulePtr    object,
    short*              adcVals,
    short               size
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
adcVals	A pointer to the location where the voltage and temperature values will be stored. Must be large enough to hold ADC_NUM_CHANNELS. The ADC_OPTIONS enumeration can be used to index into the adcVals array.
size	The size of the adcVals array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleClearErrors**

Clears the module errors and warnings.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleClearErrors
(
    MseLvdModulePtr    object
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
--------	--

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleGetPosition**

Gets the positions of all of the LVDT sensors in the module.  
 The resulting position is determined by first multiplying the count \* resolution, then multiplying by the scaling, then adding the device offset.  
 The scaling is set with the MseLvdModuleSetScaling function, it defaults to 1.0.  
 The device offset is set with the MseLvdModuleSetDeviceOffset function, it defaults to 0.0.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleGetPosition
(
    MseLvdModulePtr    object,
    double*            pos,
    short              numChannels,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

- object                    A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
- pos                      A pointer to the location where the LVDT positions will be copied to. Must be large enough to hold NUM\_LVDT\_CHANNELS.
- numChannels            The size of the pos array
- option                   Whether to get the latest or the latched positions

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleSetDeviceOffset**

The MseLvdModuleSetDeviceOffset method is used to set the offset that will be applied to the position calculated from the counts in the MseLvdModuleGetPosition function.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleSetDeviceOffset
(
    MseLvdModulePtr    object,
    unsigned short     channel,
    double              offset
);
```

**Parameters**

- object                    A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
- channel                   The channel of the encoder to apply the offset to
- offset                    The offset to apply

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleGetDeviceOffset**

The MseLvdModuleGetDeviceOffset method is used to return the offset that will be applied to the position calculated from the counts in the MseLvdModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleGetDeviceOffset
(
    MseLvdModulePtr    object,
    unsigned short     channel
    double*            offset
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
channel	The channel of the encoder that the offset will be applied to
offset	A pointer to the location where the offset will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleGetExcitationValues**

Gets the primary windings excitation voltage and frequency. The frequency is in kHz.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleGetExcitationValues
(
    MseLvdModulePtr    object,
    double*            voltage,
    double*            frequency
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
voltage	A pointer to the location where the primary winding excitation voltage will be stored
frequency	A pointer to the location where the primary winding excitation frequency will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleSetExcitationVoltage**

Sets the primary windings excitation voltage. The voltage is used for all sensors connected to the module. The data sheet for the sensor should be used to determine the correct voltage.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleSetExcitationVoltage
(
    MseLvdModulePtr    object,
    const double        voltage
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
voltage	The desired primary winding excitation voltage. The voltage can be from LVDT_EXCITATION_VOLTAGE_MIN_VPP to LVDT_EXCITATION_VOLTAGE_MAX_VPP.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleSetExcitationFrequency**

Sets the primary windings excitation frequency. The frequency is used for all sensors connected to the module. The data sheet for the sensor should be used to determine the correct frequency.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleSetExcitationFrequency
(
    MseLvdModulePtr    object,
    const double        frequency
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
frequency	The desired primary winding excitation frequency in kHz. The frequency can be from LVDT_EXCITATION_FREQUENCY_MIN_KHZ to LVDT_EXCITATION_FREQUENCY_MAX_KHZ.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleGetVoltage**

Gets the LVDT sensor count and voltage value that is obtained from the ADC for the desired channel. The count is useful because you can convert this to a position by multiplying it by the resolution of the sensor. The voltage is an approximate value of the reading from the ADC and is informational only.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleGetVoltage
(
    MseLvdModulePtr    object,
    unsigned short      channel,
    double*             channelVoltage,
    long*               counts
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
channel	The channel to get the sensor count and voltage indexed from 0
channelVoltage	The location where the LVDT sensor output voltage will be stored
counts	The location where the LVDT sensor output counts will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleSetChannelPresence**

Sets whether the specified channel is populated.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleSetChannelPresence
(
    MseLvdModulePtr    object,
    unsigned char      isConnected,
    unsigned char      channel
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
isConnected	A value of 1 if an LVDT is connected to the channel
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleGetChannelPresence**

The MseLvdModuleGetChannelPresence method is used to get whether an LVDT sensor is attached to a specific channel.

**Method**

```
MSE_RESPONSE_CODE MseLvdModuleGetChannelPresence
(
    MseLvdModulePtr    object,
    unsigned char*      isConnected,
    unsigned char      channel
);
```

**Parameters**

isConnected	1 signifies that an LVDT sensor is connected to the channel
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdModuleSetDiagnosticsEnabled**

The MseLvdModuleSetDiagnosticsEnabled method is used to set the sensor group to monitor based on the LVDT\_UPDATE\_CHOICES enumeration passed in.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleSetDiagnosticsEnabled
(
    MseLvdModulePtr    object
    const LVDT_UPDATE_CHOICES choice
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
choice	The desired sensor group to monitor

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtGetSensorGain**

The MseLvdtGetSensorGain method is used to get the gain code used for a given sensor.

**Function**

```
MSE_RESPONSE_CODE MseLvdtGetSensorGain
(
    MseLvdtModulePtr    object,
    const unsigned short channelNum,
    unsigned char*       gainCode
);
```

**Parameters**

- object                    A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function
- channelNum                The channel number to get the gain code of indexed from 0
- gainCode                  A pointer to the location where the gain code for the desired channel will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtSetSensorGain**

The MseLvdtSetSensorGain method is used to set the gain code used for a given sensor.

**Function**

```
MSE_RESPONSE_CODE MseLvdtSetSensorGain
(
    MseLvdtModulePtr    object,
    const unsigned short channelNum,
    const unsigned char  gainCode
);
```

**Parameters**

- object                    A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function
- channelNum                The channel number to get the gain code of indexed from 0
- gainCode                  The gain code for the desired channel

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdtTeachSensorGain**

The MseLvdtTeachSensorGain method is used to send a command to the module to adjust the gain until a value is found that allows for the greatest range.

**Function**

```
MSE_RESPONSE_CODE MseLvdtTeachSensorGain
(
    MseLvdtModulePtr    object,
    const unsigned short channelNum
);
```

**Parameters**

- object                    A pointer to the MseLvdtModule object that was created by the MseLvdtModuleCreate function
- channelNum                The channel number to teach the gain code of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseLvdGetTeachSensorGainFinished**

The MseLvdGetTeachSensorGainFinished method is used to get whether the teach has finished for the MseLvdTeachSensorGain function.

**Function**

```
MSE_RESPONSE_CODE MseLvdGetTeachSensorGainFinished
(
    MseLvdModulePtr    object,
    const unsigned short channelNum,
    unsigned short*    gainCode
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
channelNum	The channel number to get the teach completion status of indexed from 0
gainCode	A pointer to the location where the resulting gain code from the teach will be stored. This will be 0 if the teach has not been completed.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseLvdGetFpgaRevision**

The MseLvdGetFpgaRevision method is used to get the revision of the FPGA code used in the module. The revision is in the form 0xMMmm, where MM is the major number and mm is the minor number (e.g. 0x0100 is V1.00).

**Function**

```
MSE_RESPONSE_CODE MseLvdGetFpgaRevision
(
    MseLvdModulePtr    object,
    unsigned short*    revision
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
revision	A pointer to the location where the FPGA revision will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.  
add:

**MseLvdModuleSetOversampling**

Sets the oversampling used when reading the LVDT sensor count from the ADC. A smaller value will take less time to obtain a reading but will have more noise.

**Function**

```
MSE_RESPONSE_CODE MseLvdModuleSetOversampling
(
    MseLvdModulePtr    object,
    const LVDT_OVERSAMPLING_CHOICES choice
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseLvdModuleCreate function
choice	One of the LVDT_OVERSAMPLING_CHOICES enumerations

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

## 2.17 Analog methods and functions

C++ methods and the C functions are separated into two sections for easier lookup.

### C++ Methods

#### Constructor

```
MseAnalogModule(void);
```

#### initializeModule

The initializeModule method is used to initialize an analog module. It calls the MseModule::initializeModule() and then calls setModuleInitialized(true) if the initialization passes. The number of channels is set to NUM\_MSE1000\_ANALOG\_CHANNELS.

#### Method

```
MseResults initializeModule
(
    const char*      mseIpAddress,
    bool             useAsync
);
```

#### Parameters

mseIpAddress	The IP address of the module to initialize
useAsync	True if the MSE should send asynchronous messages

#### Return value

MseResults	A response code representing whether the method succeeded.
------------	--

#### getVoltage

The getVoltage method is used to get the analog voltage for all channels. The voltage is a value between -10 and 10 V.

#### Method

```
MseResults getVoltage
(
    double*          values,
    unsigned short   numValues,
    COUNT_REQUEST_OPTION option
);
```

#### Parameters

values	A pointer to the location where the analog voltage values will be stored. This must be large enough to hold NUM_MSE1000_ANALOG_VALUES_PER_CHANNEL values.
numValues	The number of doubles passed in for the values array
option	Whether to read live or latched positions

#### Return value

MseResults	A response code representing whether the method succeeded.
------------	--

**getCurrent**

The `getCurrent` method is used to get the analog current for all channels. The current is a value between 4 and 20 mA.

**Method**

```
MseResults getCurrent
(
    double*          values,
    unsigned short   numValues,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

values	A pointer to the location where the analog current values will be stored. This must be large enough to hold <code>NUM_MSE1000_ANALOG_VALUES_PER_CHANNEL</code> values.
numValues	The number of doubles passed in for the values array
option	Whether to read live or latched positions

**Return value**

MseResults A response code representing whether the method succeeded.

**getValues**

The `getValues` method is used to get the analog voltage and current for all channels. The voltage is a value between -10 and 10 V. The current is a value between 4 and 20 mA.

**Method**

```
MseResults getValues
(
    double*          values,
    unsigned short   numValues,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

values	A pointer to the location where the analog voltage and current values will be stored. This must be large enough to hold $(\text{NUM\_MSE1000\_ANALOG\_CHANNELS} * \text{NUM\_MSE1000\_ANALOG\_VALUES\_PER\_CHANNEL})$ values. The values returned are in the following order: <code>values[0]</code> = voltage channel 1, <code>values[1]</code> = current channel 1, <code>values[2]</code> = voltage channel 2, <code>values[3]</code> = current channel 2.
numValues	The number of doubles passed in for the values array
option	Whether to read live or latched positions

**Return value**

MseResults A response code representing whether the method succeeded.

**getScaledValues**

The getScaledValues method is used to get the analog voltage and current for all channels. The values for each channel are scaled with the resolution and the offset is subtracted, then the value is multiplied by the scaling, then the device offset is added.

The resolution is set with the setResolution method.

The offset is set with the setOffset method.

The scaling is set with the setScaling method, it defaults to 1.0.

The device offset is set with the setDeviceOffset method in the base class, it defaults to 0.0..

The initial voltage is a value between -10 and 10 V.

The initial current is a value between 4 and 20 mA.

**Method**

```
MseResults getScaledValues
(
    double*          values,
    unsigned short   numValues,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

values	A pointer to the location where the analog voltage and current values will be stored. This must be large enough to hold (NUM_MSE1000_ANALOG_CHANNELS * NUM_MSE1000_ANALOG_VALUES_PER_CHANNEL) values. The values returned are in the following order: values[0] = voltage channel 1, values[1] = current channel 1, values[2] = voltage channel 2, values[3] = current channel 2.
numValues	The number of doubles passed in for the values array
option	Whether to read live or latched positions

**Return value**

MseResults A response code representing whether the method succeeded.

**getDiagVoltages**

The getDiagVoltages method is used to get the supply and output voltages of the analog module ADC. Refer to the ANALOG\_DIAG\_VOLTAGES\_ENUM enumeration on page 50 for information on the data returned.

**Method**

```
MseResults getDiagVoltages
(
    double*          values,
    unsigned short   numValues
);
```

**Parameters**

values	A pointer to the location where the supply and output voltages will be stored. This array must be large enough to hold (NUM_ANALOG_DIAG_VOLTAGES * NUM_MSE1000_ANALOG_CHANNELS).
numValues	The number of doubles passed in for the values array

**Return value**

MseResults A response code representing whether the method succeeded.

**setNumSamples**

The setNumSamples method is used to set the number of samples to use for the averaging function in the analog module. Defaults to MAX\_NUM\_ANALOG\_AVG\_SAMPLES.

**Method**

```
MseResults setNumSamples
(
    const unsigned char    channelNum,
    const unsigned char    numSamples
);
```

**Parameters**

channelNum	The channel number to set the number of samples of indexed from 0
numSamples	The number of samples to use when determining the running average. Can be from 0 to MAX_NUM_ANALOG_AVG_SAMPLES.

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**setResolution**

Sets the resolution to use when converting from the device's signal value in mA or V to a value in user units. Defaults to 1.

**Method**

```
MseResults setResolution
(
    const unsigned short    channelNum,
    const double            resolution
);
```

**Parameters**

channelNum	The channel number to set the resolution of indexed from 0
resolution	The resolution to use

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**getResolution**

Gets the resolution used when converting from the device's signal value in mA or V to a value in user units.

**Method**

```
MseResults getResolution
(
    const unsigned short    channelNum,
    const double*          resolution
);
```

**Parameters**

channelNum	The channel number to get the resolution of indexed from 0
resolution	The resolution used

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**setOffset**

Sets the offset to use after converting from the device's signal value in mA or V to a value in user units. Defaults to 0.

**Method**

```
MseResults setOffset
(
    const unsigned short channelNum,
    const double         offset
);
```

**Parameters**

channelNum	The channel number to set the offset of indexed from 0
offset	The offset to use

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**getOffset**

Gets the offset to use after converting from the device's signal value in mA or V to a value in user units.

**Method**

```
MseResults getOffset
(
    const unsigned short channelNum,
    double*              offset
);
```

**Parameters**

channelNum	The channel number to get the offset of indexed from 0
offset	The offset used

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**computeResolutionAndOffset**

Computes and saves the resolution and offset to use for converting from the device's signal value in mA or V to a value in user units. The computed resolution and offset will be used when the user calls the getScaledValues method.

**Method**

```
MseResults computeResolutionAndOffset
(
    const unsigned short channelNum,
    double*              resolution,
    double*              offset,
    const double         instrumentationMax,
    const double         instrumentationMin,
    const double         signalMax,
    const double         signalMin
);
```

**Parameters**

channelNum	The channel number to compute the resolution and offset for indexed from 0
resolution	A pointer to the location where the computed resolution will be saved
offset	A pointer to the location where the computed offset will be saved
instrumentationMax	The calibration max value of the device in user units
instrumentationMin	The calibration min value of the device in user units
signalMax	The calibration max value of the device in raw voltage or mA
signalMin	The calibration min value of the device in raw voltage or mA

**Return value**

MseResults	A response code representing whether the method succeeded.
------------	--

**setScaling**

Sets the generic device scaling to the desired multiplier. This value will be applied to the position when getScaledValues is called after the resolution and offset are applied.

**Method**

```
bool setScaling
(
    const unsigned short channelNum,
    const double value
);
```

**Parameters**

channelNum	The channel number to set indexed from 0
value	The scaling to use

**Return value**

The return value delivers False if the channel is out of range.

**getScaling**

Gets the generic device scaling of the desired channel. This value will be applied to the position when getScaledValues is called after the resolution and offset are applied.

**Method**

```
MseResults getScaling
(
    const unsigned short channelNum,
    double* value
);
```

**Parameters**

channelNum	The channel number to get indexed from 0
value	The scaling used

**Return value**

The return value delivers False if the channel is out of range or if the pointer is NULL.

## C Functions

The analog module C functions can be found in the MseAnalogModuleWrapper.h file.

### MseAnalogModuleCreate

Creates a MseAnalogModule object and returns a pointer to it.

#### Function

```
MseAnalogModulePtr MseAnalogModuleCreate
(
);
```

#### Return value

The return value delivers a pointer to the MseAnalogModule object that was created.

### MseAnalogModuleDelete

Deletes the MseAnalogModule object that was passed in.

#### Function

```
void MseAnalogModuleDelete
(
    MseAnalogModulePtr object
);
```

#### Parameters

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
--------	--

### MseAnalogModuleInitialize

Initializes the MseAnalogModule object that was passed in.

#### Function

```
MSE_RESPONSE_CODE MseAnalogModuleInitialize
(
    MseAnalogModulePtr object,
    char* mseIpAddress,
    bool useAsync
);
```

#### Parameters

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
mseIpAddress	The IP address of the module to initialize
useAsync	Whether to enable asynchronous communication from the module

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseAnalogModuleGetNumChannels**

Gets the number of channels on the module.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetNumChannels
(
    MseAnalogModulePtr    object,
    unsigned short*       numChannels
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
numChannels	A pointer to the location where the number of channels will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetModuleErrorState**

Gets the error state of the module. A value of True signifies that there is an error. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetModuleErrorState
(
    MseAnalogModulePtr    object,
    bool*                  errorState
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
errorState	A pointer to the location where the error state will be stored. A subsequent call to MseAnalogModuleGetModuleErrors can be made to get the actual errors.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetModuleErrors**

Gets the errors specific to the module. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetModuleErrors
(
    MseAnalogModulePtr    object,
    long*                  errors,
    double*                ranges,
    short                  size
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
errors	A pointer to the location where the errors state will stored. The errors is a single long that can be masked with the INTEGRITY_ENUMS to determine which error has occurred.
ranges	A pointer to the location where the ranges used to determine an error will be stored. The ranges is an array that must be large enough to hold NUM_INTEGRITY_RANGES.
size	The size of the ranges array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetAdcValues**

Gets the voltage and temperature values for the module.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetAdcValues
(
    MseAnalogModulePtr    object,
    short*                 adcVals,
    short                  size
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
adcVals	A pointer to the location where the voltage and temperature values will be stored. Must be large enough to hold ADC_NUM_CHANNELS. The ADC_OPTIONS enumeration can be used to index into the adcVals array.
size	The size of the adcVals array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleClearErrors**

Clears the module errors and warnings.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleClearErrors
(
    MseAnalogModulePtr    object
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
--------	--

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetLatch**

Gets whether the latch is active.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetLatch
(
    MseAnalogModulePtr    object,
    bool*                 isLatched,
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
isLatched	A pointer to the location where the latch state will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleClearLatch**

Clears the latch. The user must make sure that the base module latch is cleared first or the latch will immediately trigger again.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleClearLatch
(
    MseAnalogModulePtr    object
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
--------	--

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetVoltage**

Gets the analog voltage for all channels. The voltage is a value between -10 and 10V.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetVoltage
(
    MseAnalogModulePtr    object,
    double*                values,
    unsigned short         numValues,
    COUNT_REQUEST_OPTION  option
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
values	A pointer to the location where the voltage values will be stored
numValues	The size of the values array. This must be large enough to hold NUM_MSE1000_ANALOG_CHANNELS.
option	Whether to get the latest or the latched positions

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetCurrent**

Gets the analog current for all channels. The current is a value between 4 and 20mA.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetCurrent
(
    MseAnalogModulePtr    object,
    double*                values,
    unsigned short         numValues,
    COUNT_REQUEST_OPTION  option
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
values	A pointer to the location where the current values will be stored
numValues	The size of the values array. This must be large enough to hold NUM_MSE1000_ANALOG_CHANNELS.
option	Whether to get the latest or the latched positions

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetValues**

Gets the analog voltage and current for all channels. The voltage is a value between -10 and 10V. The current is a value between 4 and 20mA.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetValues
(
    MseAnalogModulePtr    object,
    double*                values,
    unsigned short         numValues,
    COUNT_REQUEST_OPTION  option
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
values	A pointer to the location where the analog voltage and current values will be stored. This must be large enough to hold (NUM_MSE1000_ANALOG_CHANNELS * NUM_MSE1000_ANALOG_VALUES_PER_CHANNEL) values. The values returned are in the following order: values[0] = voltage channel 1, values[1] = current channel 1, values[2] = voltage channel 2, values[3] = current channel 2.
numValues	The size of the values array. This must be large enough to hold NUM_MSE1000_ANALOG_CHANNELS.
option	Whether to get the latest or the latched positions

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetScaledValues**

Gets the analog voltage and current for all channels.

The values for each channel are scaled with the resolution and the offset is subtracted, then the value is multiplied by the scaling, then the device offset is added.

The resolution is set with the MseAnalogModuleSetResolution function.

The offset is set with the MseAnalogModuleSetOffset function.

The scaling is set with the MseAnalogModuleSetScaling function, it defaults to 1.0.

The device offset is set with the MseAnalogModuleSetDeviceOffset function, it defaults to 0.0..

The initial voltage is a value between -10 and 10V.

The initial current is a value between 4 and 20mA.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetScaledValues
(
    MseAnalogModulePtr    object,
    double*                values,
    unsigned short         numValues,
    COUNT_REQUEST_OPTION  option
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
values	A pointer to the location where the analog voltage and current values will be stored. This must be large enough to hold (NUM_MSE1000_ANALOG_CHANNELS * NUM_MSE1000_ANALOG_VALUES_PER_CHANNEL) values. The values returned are in the following order: values[0] = voltage channel 1, values[1] = current channel 1, values[2] = voltage channel 2, values[3] = current channel 2.
numValues	The size of the values array. This must be large enough to hold NUM_MSE1000_ANALOG_CHANNELS.
option	Whether to get the latest or the latched positions

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleSetDeviceOffset**

The MseAnalogModuleSetDeviceOffset method is used to set the offset that will be applied to the position calculated from the counts in the MseAnalogModuleGetScaledValues function.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleSetDeviceOffset
(
    MseAnalogModulePtr    object,
    unsigned short        channel,
    double                 offset
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
channel	The channel of the encoder to apply the offset to
offset	The offset to apply

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetDeviceOffset**

The MseAnalogModuleGetDeviceOffset method is used to return the offset that will be applied to the position calculated from the counts in the MseAnalogModuleGetScaledValues function.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetDeviceOffset
(
    MseAnalogModulePtr    object,
    unsigned short        channel,
    double*                offset
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
channel	The channel of the encoder that the offset will be applied to
offset	A pointer to the location where the offset will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleSetScaling**

Sets the generic device scaling to the desired multiplier. This value will be applied to the position when MseAnalogModuleGetScaledValues is called after the resolution and offset are applied.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleSetScaling
(
    MseAnalogModulePtr    object,
    const unsigned short   channelNum,
    const double           value
);
```

**Parameters**

object	A pointer to the MseLvdModule object that was created by the MseAnalogModuleCreate function
channelNum	The channel to set indexed from 0
val	The scaling to use for the channel

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetScaling**

Gets the generic device scaling of the desired channel. This value will be applied to the position when MseAnalogModuleGetScaledValues is called after the resolution and offset are applied.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetScaling
(
    MseAnalogModulePtr    object,
    const unsigned short  channelNum,
    double*                value
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
channelNum	The channel to get indexed from 0
val	A pointer to the location where the scaling is stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetDiagVoltages**

Gets the supply and output voltages of the analog module ADC. Refer to the ANALOG\_DIAG\_VOLTAGES\_ENUM enumeration on page 50 for information on the data returned.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetDiagVoltages
(
    MseAnalogModulePtr    object,
    double*                values,
    unsigned short         numValues
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
values	A pointer to the location where the supply and output voltages will be stored. This array must be large enough to hold (NUM_ANALOG_DIAG_VOLTAGES * NUM_MSE1000_ANALOG_CHANNELS).
numValues	The number of doubles passed in for the values array

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleSetNumSamples**

Sets the number of samples to use for the averaging function in the analog module. Defaults to MAX\_NUM\_ANALOG\_AVG\_SAMPLES.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleSetNumSamples
(
    MseAnalogModulePtr    object,
    const unsigned char    channelNum,
    const unsigned char    numSamples
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
channelNum	The channel number to set the number of samples of indexed from 0
numSamples	The number of samples to use when determining the running average. Can be from 0 to MAX_NUM_ANALOG_AVG_SAMPLES.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleSetResolution**

Sets the resolution to use when converting from the device's signal value in mA or V to a value in user units. Defaults to 1.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleSetResolution
(
    MseAnalogModulePtr    object,
    const unsigned short  channelNum,
    const double           resolution
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
channelNum	The channel number to set the resolution of indexed from 0
resolution	The resolution to use

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetResolution**

Gets the resolution used when converting from the device's signal value in mA or V to a value in user units.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetResolution
(
    MseAnalogModulePtr    object,
    const unsigned short  channelNum,
    const double*         resolution
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
channelNum	The channel number to get the resolution of indexed from 0
resolution	A pointer to the location where the resolution will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleSetOffset**

Gets the resolution used when converting from the device's signal value in mA or V to a value in user units.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleSetOffset
(
    MseAnalogModulePtr    object,
    const unsigned short  channelNum,
    const double           offset
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
channelNum	The channel number to set the offset of indexed from 0
offset	The offset to use

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleGetOffset**

Gets the offset to use after converting from the device's signal value in mA or V to a value in user units.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleGetOffset
(
    MseAnalogModulePtr    object,
    const unsigned short  channelNum,
    const double*         offset
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
channelNum	The channel number to get the offset of indexed from 0
offset	A pointer to the location where the offset will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseAnalogModuleComputeResolutionAndOffset**

Computes and saves the resolution and offset to use for converting from the device's signal value in mA or V to a value in user units. The computed resolution and offset will be used when the user calls the getScaledValues method.

**Function**

```
MSE_RESPONSE_CODE MseAnalogModuleComputeResolutionAndOffset
(
    MseAnalogModulePtr    object,
    const unsigned short  channelNum,
    double*                resolution,
    double*                offset,
    const double          instrumentationMax,
    const double          instrumentationMin,
    const double          signalMax,
    const double          signalMin
);
```

**Parameters**

object	A pointer to the MseAnalogModule object that was created by the MseAnalogModuleCreate function
channelNum	The channel number to compute the resolution and offset for indexed from 0
resolution	A pointer to the location where the computed resolution will be saved
offset	A pointer to the location where the computed offset will be saved
instrumentationMax	The calibration max value of the device in user units
instrumentationMin	The calibration min value of the device in user units
signalMax	The calibration max value of the device in raw voltage or mA
signalMin	The calibration min value of the device in raw voltage or mA

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



## 2.18 TTL methods and functions

C++ methods and the C functions are separated into two sections for easier lookup.

### C++ Methods

#### Constructor

```
MseTtlModule(void);
```

#### initializeModule

The initializeModule method is used to initialize a TTL module. It calls the MseModule::initializeModule() and then calls setModuleInitialized(true) if the initialization passes. The number of channels is set based on the type of TTL module.

#### Method

```
MseResults initializeModule
(
    const char*      mseIpAddress,
    bool             useAsync
);
```

#### Parameters

mseIpAddress	The IP address of the module to initialize
useAsync	True if the MSE should send asynchronous messages

#### Return value

MseResults	A response code representing whether the method succeeded
------------	---

#### getPositions

The getPositions method is used to get the position of a specific TTL encoder in user units. The setEncoderType, setUom, and setLineCount or setSignalPeriod methods must be called before calling this method.

The resulting position is determined by first multiplying the count by the error compensation, then multiplying by the scaling, then computing the position based on the resolution, then converting to the correct UOM, then adding the device offset, and finally formatting the rotary position if necessary.

The error compensation is set with the setErrorCompensation method of the base class, it defaults to 1.0.

The scaling is set with the setScaling method of the base class, it defaults to 1.0.

The resolution is calculated based on the setLineCount or setSignalPeriod values.

The UOM is set with setUom, it defaults to UOM\_RAW\_COUNTS and getPositions will not do any position calculations until set.

The device offset is set with the setDeviceOffset method of the base class, it defaults to 0.0.

The rotary format can be changed with the setRotaryFormat method, it defaults to ROTARY\_FORMAT\_360.

#### Method

```
MseResults getPositions
(
    double*          pos,
    unsigned short   numChannels,
    COUNT_REQUEST_OPTION option
);
```

#### Parameters

pos	A pointer to the location where the position will be stored
numChannels	The number of doubles in the pos array passed in
option	Whether to read live or latched positions

#### Return value

MseResults	A response code representing whether the method succeeded.
------------	--

**getCounts**

Gets the counts of all the attached encoders.

**Method**

```
MSELIB_EXPORT MseResults getCounts
(
    unsigned long*    counts,
    unsigned short   numChannels,
    COUNT_REQUEST_OPTION option
)
```

**Parameters**

counts	The counts returned from the encoders
numChannels	The number of channels to read into the counts array that was passed in
option	The type of count to return

**Return value**

The return value delivers a response code representing whether the counts were retrieved correctly.

**setEncoderType**

Set the type of encoder used on the specified channel.

**Method**

```
MseResults setEncoderType
(
    ENCODER_TYPE_ENUM encoderType,
    const unsigned short channel
);
```

**Parameters**

encoderType	The type of encoder to use for calculating the position value
channel	The channel to set the encoder type of indexed from 0

**Return value**

MseResults	A response code representing whether the method succeeded
------------	---

**setUom**

Set the unit of measurement on the specified channel. This value will be applied to the position when getPositions is called.

**Method**

```
MseResults setUom
(
    UOM uom,
    const unsigned short channel
);
```

**Parameters**

uom	The unit of measurement to use for computing the position value
channel	The channel to set the uom for indexed from 0

**Return value**

MseResults	A response code representing whether the method succeeded
------------	---

**setLineCount**

Set the line count of the encoder for the specified channel.

**Method**

```
MseResults setLineCount
(
    unsigned long    lineCount,
    const unsigned short channel
);
```

**Parameters**

lineCount      The line count of the encoder to use for computing the position value  
channel        The channel to set the line count for indexed from 0

**Return value**

MseResults      A response code representing whether the method succeeded

**getLineCount**

Get the line count of the encoder for the specified channel.

**Method**

```
MseResults getLineCount
(
    unsigned long*    lineCount,
    const unsigned short channel
);
```

**Parameters**

lineCount      A pointer to the location where the line count of the encoder used for computing the position value will be stored  
channel        The channel to get the line count of indexed from 0

**Return value**

MseResults      A response code representing whether the method succeeded

**setSignalPeriod**

Set the signal period of the encoder for the specified channel.

**Method**

```
MseResults setSignalPeriod
(
    unsigned long    signalPeriod,
    const unsigned short channel
);
```

**Parameters**

signalPeriod    The signal period of the encoder to use for computing the position value  
channel        The channel to set the signal period for indexed from 0

**Return value**

MseResults      A response code representing whether the method succeeded

**getSignalPeriod**

Get the signal period of the encoder for the specified channel.

**Method**

```
MseResults getSignalPeriod
(
    unsigned long*    signalPeriod,
    const unsigned short channel
);
```

**Parameters**

signalPeriod	A pointer to the location where the signal period of the encoder used for computing the position value will be stored
channel	The channel to get the signal period of indexed from 0

**Return value**

MseResults	A response code representing whether the method succeeded
------------	---

**setCountingDirection**

Set the counting direction of the encoder for the specified channel.

**Method**

```
MseResults setCountingDirection
(
    const unsigned char    direction,
    const unsigned short    channel
);
```

**Parameters**

direction	The counting direction of the encoder. A value of 0 is used for normal and 1 for inverted.
channel	The channel to set the counting direction for indexed from 0

**Return value**

MseResults	A response code representing whether the method succeeded
------------	---

**setChannelPresence**

Set the connection status of the encoder for the specified channel.

**Method**

```
MseResults setChannelPresence
(
    unsigned char    isConnected,
    unsigned char    channel
);
```

**Parameters**

isConnected	A value of 1 if the channel is populated, otherwise 0
channel	The channel to set the presence for indexed from 0

**Return value**

MseResults	A response code representing whether the method succeeded
------------	---

**getChannelPresence**

Get the connection status of the encoder for the specified channel.

**Method**

```
MseResults getChannelPresence
(
    unsigned char*    isConnected,
    unsigned char     channel
);
```

**Parameters**

isConnected	A pointer to the location where the channel presence will be stored
channel	The channel to get the presence of indexed from 0

**Return value**

MseResults	A response code representing whether the method succeeded
------------	---

**isReferencingComplete**

Gets whether the referencing has been completed for an encoder on the specified channel.

**Method**

```
MseResults isReferencingComplete
(
    const unsigned char    channel,
    bool*                  isComplete
);
```

**Parameters**

channel	The channel to get the referencing state of indexed from 0
isComplete	A pointer to the location where the referencing state value will be stored. A value of True is returned if referencing is complete.

**Return value**

MseResults	A response code representing whether the method succeeded
------------	---

**initReferencing**

Initializes the parameters used for determining absolute position of a specified encoder.

**Method**

```
MseResults initReferencing
(
    const unsigned char    channel,
    const REFERENCE_MARK_ENUM refMarkType,
    const unsigned short   value
);
```

**Parameters**

channel	The channel to initialize the referencing of indexed from 0
refMarkType	The type of referencing used by the encoder
value	This is the signal period of a linear encoder and the line count for a rotary encoder

**Return value**

MseResults	A response code representing whether the method succeeded
------------	---

**acknowledgeReferencing**

Used to send an acknowledge to the module informing it that the asynchronous reference complete message was received. If asynchronous communication is used and this is not sent, the module will keep sending reference complete messages.

**Method**

```
MseResults acknowledgeReferencing
(
    const unsigned char channel
);
```

**Parameters**

channel                    The channel to acknowledge the referencing of indexed from 0

**Return value**

MseResults                A response code representing whether the method succeeded

**getReferencingState**

Gets the state of the referencing for the desired channel. This method should be called after isReferencingComplete is true or before acknowledgeReferencing is sent in order to check if referencing succeeded. Referencing succeeds if the refMarkState is REF\_MARK\_FINISHED.

**Method**

```
MseResults getReferencingState
(
    const unsigned char channel,
    REF_MARK_STATE* refMarkState
);
```

**Parameters**

channel                    The channel to get the referencing state indexed from 0

refMarkState              The state of the referencing. See the REF\_MARK\_STATE enumeration for more information.

**Return value**

The return value delivers a response code representing whether the command was successful.

**getFpgaRevision**

Get the revision of the FPGA code used in the module.

**Method**

```
MseResults getFpgaRevision
(
    unsigned short* revision
);
```

**Parameters**

revision                   A pointer to the location where the FPGA revision will be stored. The revision is in the form 0xMMmm, where MM is the major version and mm is the minor version (e.g. 0x0100 is V1.00).

**Return value**

MseResults                A response code representing whether the method succeeded

## C Functions

The TTL module C functions can be found in the MseTtlModuleWrapper.h file.

### MseTtlModuleCreate

Creates a MseTtlModule object and returns a pointer to it.

#### Function

```
MseTtlModulePtr MseTtlModuleCreate
(
);
```

#### Return value

The return value delivers a pointer to the MseTtlModule object that was created.

### MseTtlModuleDelete

Deletes the MseTtlModule object that was passed in.

#### Function

```
void MseTtlModuleDelete
(
    MseTtlModulePtr    object
);
```

#### Parameters

object                      A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function

### MseTtlModuleInitialize

Initializes the MseTtlModule object that was passed in.

#### Function

```
MSE_RESPONSE_CODE MseTtlModuleInitialize
(
    MseTtlModulePtr    object,
    char*               mseIpAddress,
    bool                useAsync
);
```

#### Parameters

object                      A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function  
mseIpAddress                The IP address of the module to initialize  
useAsync                    Whether to enable asynchronous communication from the module

#### Return value

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetNumChannels**

Gets the number of channels on the module.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetNumChannels
(
    MseTtlModulePtr    object,
    unsigned short*    numChannels
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
numChannels	A pointer to the location where the number of channels will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleSetEncoderType**

Set the type of encoder used on the specified channel.

**Function**

```
MSE_RESPONSE_CODE setEncoderType
(
    MseTtlModulePtr    object,
    ENCODER_TYPE_ENUM  encoderType,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
encoderType	The type of encoder to use for calculating the position value
channel	The channel to set the encoder type of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetEncoderType**

Get the type of encoder used on the specified channel.

**Function**

```
MSE_RESPONSE_CODE getEncoderType
(
    MseTtlModulePtr    object,
    ENCODER_TYPE_ENUM* encoderType,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
encoderType	A pointer to the location where the type of encoder to use for calculating the position value will be stored
channel	The channel to get the encoder type of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseTtlModuleSetUom**

Sets the unit of measurement to use for the specified channel when requesting position.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleSetUom
(
    MseTtlModulePtr    object,
    UOM                uom,
    short              channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
uom	The unit of measurement to use when requesting position for the specified channel
channel	The channel to set the unit of measurement of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetUom**

Gets the unit of measurement used for the specified channel when requesting position.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetUom
(
    MseTtlModulePtr    object,
    UOM*               uom,
    short              channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
uom	A pointer to the location where the unit of measurement will be stored
channel	The channel to get the unit of measurement of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleSetErrorCompensation**

Set the error compensation used on the specified channel.

**Function**

```
MSE_RESPONSE_CODE setErrorCompensation
(
    MseTtlModulePtr    object,
    double             val,
    short              channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
val	The error compensation value to apply when calculating the position
channel	The channel to set the error compensation value of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetErrorCompensation**

Get the error compensation used on the specified channel.

**Function**

```
MSE_RESPONSE_CODE getErrorCompensation
(
    MseTtlModulePtr    object,
    double*            val,
    short               channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
val	A pointer to the location where the error compensation value to apply when calculating the position will be stored
channel	The channel to get the error compensation value of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleSetScaling**

Sets the device scaling for a specific channel to the desired multiplier. This value will be applied to the position when MseTtlGetPositions is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleSetScaling
(
    MseTtlModulePtr    object,
    const double        val,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
val	The scaling to use for the channel
channel	The channel to set indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetScaling**

Gets the device scaling for a specific channel. This value will be applied to the position when MseTtlGetPositions is called. This differs from the error compensation since the scaling can be used for gear ratios and other factors.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetScaling
(
    MseTtlModulePtr    object,
    double*            val,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
val	A pointer to the location where the scaling is stored
channel	The channel to get indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleSetCountingDirection**

Set the counting direction of the encoder for the specified channel.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleSetCountingDirection
(
    MseTtlModulePtr    object,
    const unsigned char direction,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
direction	The counting direction of the encoder. A value of 0 is used for normal and 1 for inverted.
channel	The channel to set the counting direction for indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetCountingDirection**

Get the counting direction of the encoder for the specified channel.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetCountingDirection
(
    MseTtlModulePtr    object,
    const unsigned char* direction,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
direction	A pointer to the location where the counting direction of the encoder will be stored
channel	The channel to get the counting direction for indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleSetChannelPresence**

Set the connection status of the encoder for the specified channel.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleSetChannelPresence
(
    MseTtlModulePtr    object,
    unsigned char       isConnected,
    unsigned char       channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
isConnected	A value of 1 if the channel is populated, otherwise 0
channel	The channel to set the presence for indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetChannelPresence**

Get the connection status of the encoder for the specified channel.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetChannelPresence
(
    MseTtlModulePtr    object,
    unsigned char*     isConnected,
    unsigned char       channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
isConnected	A pointer to the location where the channel presence will be stored
channel	The channel to get the presence of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetResolution**

Get the resolution of the encoder for the specified channel. The resolution is calculated when the MseTtlModuleSetLineCount or MseTtlModuleSetSignalPeriod is called.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetResolution
(
    MseTtlModulePtr    object,
    double*            resolution,
    short               channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
resolution	A pointer to the location where the resolution of the encoder will be stored
channel	The channel to get the resolution for indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleSetLineCount**

Set the line count and interpolation value of the encoder for the specified channel.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleSetLineCount
(
    MseTtlModulePtr    object,
    long               lineCount,
    TTL_INTERPOLATION  interpolation,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
lineCount	The line count of the encoder. This value is used for computing the position value.
interpolation	The interpolation value of the encoder. This value is used for computing the position value.
channel	The channel to set the line count for indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetLineCount**

Get the line count and interpolation value of the encoder for the specified channel.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetLineCount
(
    MseTtlModulePtr    object,
    long*              lineCount,
    TTL_INTERPOLATION* interpolation,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
lineCount	A pointer to the location where the line count of the encoder will be stored. This value is used for computing the position value.
interpolation	A pointer to the location where the interpolation value of the encoder will be stored. This value is used for computing the position value.
channel	The channel to get the line count of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleSetSignalPeriod**

Set the signal period and interpolation value of the encoder for the specified channel.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleSetSignalPeriod
(
    MseTtlModulePtr    object,
    double              signalPeriod,
    TTL_INTERPOLATION* interpolation,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
signalPeriod	The signal period of the encoder. This value is used for computing the position value.
interpolation	The interpolation value of the encoder. This value is used for computing the position value.
channel	The channel to set the signal period for indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetSignalPeriod**

Get the signal period and interpolation value of the encoder for the specified channel.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetSignalPeriod
(
    MseTtlModulePtr    object,
    unsigned long*      signalPeriod,
    TTL_INTERPOLATION* interpolation,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
signalPeriod	A pointer to the location where the signal period of the encoder will be stored. This value is used for computing the position value.
interpolation	A pointer to the location where the interpolation value of the encoder will be stored. This value is used for computing the position value.
channel	The channel to get the signal period of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetCounts**

Get the encoder counts for all channels. The position of a linear encoder, in mm, or a rotary encoder, in degrees, can be obtained by multiplying the counts by the resolution value obtained from `MseTtlModuleGetResolution`.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetCounts
(
    MseTtlModulePtr    object,
    unsigned long*     counts,
    short              numChannels,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

object	A pointer to the <code>MseTtlModule</code> object that was created by the <code>MseTtlModuleCreate</code> function
counts	A pointer to the location where the encoder counts will be stored. This array must be large enough to store <code>MAX_CHANNELS_PER_MODULE</code> .
numChannels	The size of the counts array passed in
option	Whether to get the latest counts or the latched counts

**Return value**

The return value delivers a `MSE_RESPONSE_CODE` representing whether the function call was successful.

**MseTtlModuleGetPositions**

The `getPositions` method is used to get the position of a specific TTL encoder in user units. The `MseTtlModuleSetEncoderType`, `MseTtlModuleSetUom`, and `MseTtlModuleSetLineCount` or `MseTtlModuleSetSignalPeriod` methods must be called before calling this method.

The resulting position is determined by first multiplying the count by the error compensation, then multiplying by the scaling, then computing the position based on the resolution, then converting to the correct UOM, then adding the device offset, and finally formatting the rotary position if necessary.

The error compensation is set with the `MseEndatModuleSetErrorCompensation` function, it defaults to 1.0.

The scaling is set with the `MseEndatModuleSetScaling` function, it defaults to 1.0.

The resolution is calculated based on the `MseTtlModuleSetLineCount` or `MseTtlModuleSetSignalPeriod` values.

The UOM is set with `MseTtlModuleSetUom`, it defaults to `UOM_RAW_COUNTS` and `MseTtlModuleGetPositions` will not do any position calculations until set.

The device offset is set with the `MseTtlModuleSetDeviceOffset` function, it defaults to 0.0.

The rotary format can be changed with the `MseTtlModuleSetRotaryFormat` function, it defaults to `ROTARY_FORMAT_360`.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetPositions
(
    MseTtlModulePtr    object,
    double*            pos,
    unsigned short      numChannels,
    COUNT_REQUEST_OPTION option
);
```

**Parameters**

object	A pointer to the <code>MseTtlModule</code> object that was created by the <code>MseTtlModuleCreate</code> function
pos	The location where the position will be stored
numChannels	The number of doubles in the <code>pos</code> array passed in
option	Whether to read live or latched positions

**Return value**

The return value delivers a `MSE_RESPONSE_CODE` representing whether the function call was successful.

**MseTtlModuleSetRotaryFormat**

The MseTtlModuleSetRotaryFormat method is used to set the rotary format that will be applied to the position calculated from the counts in the MseTtlModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleSetRotaryFormat
(
    MseTtlModulePtr    object,
    unsigned short     channel,
    ROTARY_FORMAT      format
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
channel	The channel of the encoder to apply the rotary format to
format	The ROTARY_FORMAT to apply

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetRotaryFormat**

The MseTtlModuleGetRotaryFormat method is used to return the rotary format that will be applied to the position calculated from the counts in the MseTtlModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetRotaryFormat
(
    MseTtlModulePtr    object,
    unsigned short     channel,
    ROTARY_FORMAT*     format
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
channel	The channel of the encoder that the rotary format will be applied to
format	A pointer to the location where the rotary format will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleSetDeviceOffset**

The MseTtlModuleSetDeviceOffset method is used to set the offset that will be applied to the position calculated from the counts in the MseTtlModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleSetDeviceOffset
(
    MseTtlModulePtr    object,
    unsigned short     channel,
    double             offset
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
channel	The channel of the encoder to apply the offset to
offset	The offset to apply

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetDeviceOffset**

The MseTtlModuleGetDeviceOffset method is used to return the offset that will be applied to the position calculated from the counts in the MseTtlModuleGetPositions function.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetDeviceOffset
(
    MseTtlModulePtr    object,
    unsigned short     channel,
    double*            offset
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
channel	The channel of the encoder that the offset will be applied to
offset	A pointer to the location where the offset will be stored

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleSetLatch**

Set or clear the desired latch for the entire module chain.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleSetLatch
(
    MseTtlModulePtr    object,
    LATCH_OPTIONS       option,
    LATCH_CHOICE        latchChoice
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
option	Set or reset the module chain latch. Set is only used on a base module. The reset functions as a clearing of the latch and must be called on the base module first.
latchChoice	The type of latch to set or clear

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetLatches**

Gets the latches that are active.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetLatches
(
    MseTtlModulePtr    object,
    char*              latchState,
    short               size
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
latchState	A pointer to the location where the latch state(s) will be stored. This is an array that is must be large enough to store NUM_LATCH_TYPES. The non-base modules only utilize the first latch state in the array.
size	The size of the latchState array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.



**MseTtlModuleGetModuleErrorState**

Gets the error state of the module. A value of True signifies that there is an error. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetModuleErrorState
(
    MseTtlModulePtr    object,
    bool*               errorState
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
errorState	A pointer to the location where the error state will be stored. A subsequent call to MseTtlModuleGetModuleErrors can be made to get the actual errors.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetModuleErrors**

Gets the errors specific to the module. Refer to System integrity on page 227 for more information.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetModuleErrors
(
    MseTtlModulePtr    object,
    long*               errors,
    double*             ranges,
    short               size
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
errors	A pointer to the location where the errors state will stored. The errors is a single long that can be masked with the INTEGRITY_ENUMS to determine which error has occurred.
ranges	A pointer to the location where the ranges used to determine an error will be stored. The ranges is an array that must be large enough to hold NUM_INTEGRITY_RANGES.
size	The size of the ranges array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleEnableDiags**

Sets the diagnostic mode for the channels and module. Refer to Diagnostic modes on page 231 for more information.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleEnableDiags
(
    MseTtlModulePtr    object,
    DIAG_MODE_OPTIONS  choice
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
choice	The desired level of diagnostics

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetAdcValues**

Gets the voltage and temperature values for the module.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetAdcValues
(
    MseTtlModulePtr    object,
    short*             adcVals,
    short              size
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
adcVals	A pointer to the location where the voltage and temperature values will be stored. Must be large enough to hold ADC_NUM_CHANNELS. The ADC_OPTIONS enumeration can be used to index into the adcVals array.
size	The size of the adcVals array passed in

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleClearErrors**

Clears the module and channel errors and warnings.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleClearErrors
(
    MseTtlModulePtr    object
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
--------	--

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleIsReferencingComplete**

Gets whether the referencing has been completed for an encoder on the specified channel.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleIsReferencingComplete
(
    MseTtlModulePtr    object,
    const unsigned char channel,
    bool*               isComplete
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
channel	The channel to get the referencing state of indexed from 0
isComplete	A pointer to the location where the referencing state value will be stored. A value of True is returned if referencing is complete.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleStartReferencing**

Initializes the parameters used for determining absolute position of a specified encoder.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleStartReferencing
(
    MseTtlModulePtr      object,
    const unsigned short channel,
    const REFERENCE_MARK_ENUM refMarkType,
    const unsigned short value
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
channel	The channel to initialize the referencing of indexed from 0
refMarkType	The type of referencing used by the encoder
value	This is the signal period of a linear encoder and the line count for a rotary encoder

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleAcknowledgeReferencing**

Used to send an acknowledge to the module informing it that the asynchronous reference complete message was received. If asynchronous communication is used and this is not sent, the module will keep sending reference complete messages.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleAcknowledgeReferencing
(
    MseTtlModulePtr      object,
    const unsigned char   channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
channel	The channel to acknowledge the referencing of indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetReferencingState**

Gets the state of the referencing for the desired channel. This method should be called after isReferencingComplete is true or before acknowledgeAbsolutePosition is sent in order to check if referencing succeeded. Referencing succeeds if the refMarkState is REF\_MARK\_FINISHED.

**Method**

```
MseResults MseTtlModuleGetReferencingState
(
    MseTtlModulePtr      object,
    const unsigned char   channel,
    REF_MARK_STATE*      refMarkState
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
channel	The channel to get the referencing state indexed from 0
refMarkState	A pointer to the location where the state of the referencing will be stored. See the REF_MARK_STATE enumeration for more information.

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetFpgaRevision**

Get the revision of the FPGA code used in the module.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetFpgaRevision
(
    MseTtlModulePtr    object,
    unsigned short*    revision
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
revision	A pointer to the location where the FPGA revision will be stored. The revision is in the form 0xMMmm, where MM is the major version and mm is the minor version (e.g. 0x0100 is V1.00).

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleGetChannelErrorState**

Gets the error state of a channel. An errorState of 1 signifies a COUNTER\_STATUS\_EDGE\_DISTANCE\_ERROR error. Errors can be cleared with the MseTtlModuleClearErrors function

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleGetChannelErrorState
(
    MseTtlModulePtr    object,
    bool*              errorState,
    short               channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
errorState	A pointer to the location where the error state will be copied to
channel	The channel to get

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

**MseTtlModuleEnableErrorChecking**

Sets whether error checking will be done on the specified channel. The channel defaults to enabled on power up of the module and will be checked as long as the channel is populated and the error checking is enabled. The channel status can be checked with the MseTtlModuleGetChannelErrorState when error checking is enabled.

**Function**

```
MSE_RESPONSE_CODE MseTtlModuleEnableErrorChecking
(
    MseTtlModulePtr    object,
    const bool          choice,
    const unsigned short channel
);
```

**Parameters**

object	A pointer to the MseTtlModule object that was created by the MseTtlModuleCreate function
choice	True to enable error checking, false to disable
channel	The channel to enable or disable error checking on indexed from 0

**Return value**

The return value delivers a MSE\_RESPONSE\_CODE representing whether the function call was successful.

## 2.19 Asynchronous methods

The MSE modules utilize Ethernet for communication. The client software that utilizes this library can poll constantly for errors, warnings, latching triggers, etc... or it can create a thread and wait on a socket connected to the MSE\_ASYNC\_PORT. The MSE modules will send UDP\_CONNECT messages once every ten seconds at startup until a connection is obtained via initialization of a module or by disabling the asynchronous communication manually with the setAsyncMode method. The other asynchronous messages are only sent out if the client subscribes using the useAsync parameter during initialization.

### getAsyncMsgType

The getAsyncMsgType method is static and is used to get the type of asynchronous message that was received from the MSE. A subsequent decode request needs to be sent to get the data from the message.

#### Method

```
UdpCmdType getAsyncMsgType
(
    char* msg
);
```

#### Parameters

msg The initial message received from the asynchronous port

#### Return value

UdpCmdType The type of message received

### decodeConnectMsg

The decodeConnectMsg method is static and is used to decode a UDP\_CONNECT asynchronous message from the MSE. The UdpCmdType is validated and the message is then decoded and returned in a MSEConnectResponse structure. Connect messages will stop being sent once initialization is done for the module.

#### Method

```
MseResults decodeConnectMsg
(
    char* msg,
    MSEConnectResponse* resp
);
```

#### Parameters

msg A pointer to the UDP packet that was sent from the module  
resp A pointer to a MSEConnectResponse structure to store the decoded message to

#### Return value

MseResults A response code representing whether the decodeConnectMsg command was completed

## decodeLatchMsg

The decodeLatchMsg method is static and is used to decode a UDP\_LATCH asynchronous message from the MSE . The UdpCmdType is validated and the message is then decoded. The latch states are stored in the array passed in. The module's must read their data with the COUNT\_REQUEST\_OPTION parameter set to COUNT\_REQUEST\_LATCHED, or the latch must be manually cleared in order to acknowledge the latch.

### Method

```
MseResults decodeLatchMsg
(
    char* msg,
    unsigned char* latchVals
);
```

### Parameters

msg	A pointer to the UDP packet that was sent from the module
latchVals	value of 0 if not latched or 1 if latched

### Return value

MseResults A response code representing whether the decodeLatchMsg command was completed

## decodeChannelStatusMsg

The decodeChannelStatusMsg method is static and is used to decode a UDP\_CHANNEL\_STATUS asynchronous message from the MSE . The UdpCmdType is validated and the message is then decoded. The UDP\_CHANNEL\_STATUS informs the listener if warning, error, or reference completion of a channel has occurred.

The getWarnings or getErrors method must be called to acknowledge the asynchronous message if it is a warning or error message for an EnDat module.

A getChannelStatus method must be called to acknowledge the asynchronous message if it is an error message for a TTL or 1Vpp module.

An acknowledgeAbsolutePosition method must be called to acknowledge the asynchronous message if it is a reference complete message for a 1Vpp module.

An acknowledgeReferencing method must be called to acknowledge the asynchronous message if it is a reference complete message for a TTL module.

### Method

```
MseResults decodeChannelStatusMsg
(
    char* msg,
    unsigned short* type,
    unsigned short* axis
);
```

### Parameters

msg	A pointer to the UDP packet that was sent from the module
type	The type of channel status. 1 is for warnings and errors, 2 is for reference complete
axis	The channel for reference complete acknowledgement indexed from 0. This parameter is not used for warnings and errors.

### Return value

MseResults A response code representing whether the decodeChannelStatusMsg command was completed

## 2.20 ModuleConfig Base/Reader/Writer

C++ methods and the C functions are separated into two sections for easier lookup.

### C++ Methods

The MSE module chain can be determined with the `createChain()` method. The MSEsetup application calls the `createChain()` method and stores all of the module and channel information in a file called `ModuleConfig.xml`. The MSEsetup application stores much more information that is used to configure the chain such as the module labels, channel units of measurement, etc... This file can be read and updated via the `MseConfigReader` and `MseConfigWriter` classes.

The `MseConfigBase` class contains methods that are common to the reader and the writer. The `MseConfigReader` is derived from the `MseConfigBase` and has additional methods specific to reading the `ModuleConfig` data. The `MseConfigWriter` is derived from the `MseConfigReader` in order to allow for read and write functionality. The `MseConfigWriter` was originally derived directly from the `MseConfigBase` but caused unnecessary complexity and overhead from needing two instantiated classes for reading and writing.

### MseConfigBase

#### loadXml

The `loadXml` method loads the `ModuleConfig` file passed in into memory using the DOM XML standard. This method must be called before any of the accessor methods can be used.

#### Method

```
MSE_XML_RETURN loadXml
(
    const std::string& filename;
);
```

#### Parameters

filename                                    The filename of the `ModuleConfig` XML file to be loaded into memory

#### Return value

MSE\_XML\_RETURN                    A response code representing whether the `loadXml` command was successful

#### reloadXml

The `reloadXml` method reloads the `ModuleConfig` file that was previously passed in into memory. This method is useful for keeping the DOM data that is in memory synchronized with changes in the XML file.

#### Method

```
MSE_XML_RETURN reloadXml
(
);
```

#### Return value

MSE\_XML\_RETURN                    A response code representing whether the `reloadXml` command was successful

### decodeErrorType

The `decodeErrorType` method returns a string representation of the `MSE_XML_RETURN` enumeration. This method is static and can be called without having to instantiate a `MseConfigBase` object.

#### Method

```
static std::string decodeErrorType
(
    const MSE_XML_RETURN& type;
);
```

#### Parameters

type    The `MSE_XML_RETURN` enumeration to stringify

#### Return value

std::string                                    The string representation of the enumeration

**decodeElementType**

The decodeElementType method returns a string representation of the MSE\_XML\_ELEMENTS enumeration. This method is static and can be called without having to instantiate a MseConfigBase object.

**Method**

```
static std::string decodeElementType
(
    const MSE_XML_ELEMENTS& type;
);
```

**Parameters**

type                      The MSE\_XML\_ELEMENTS enumeration to stringify

**Return value**

std::string                The string representation of the enumeration

**getFilename**

The getFilename method returns the filename as a string. If the filename has not been set, an empty string is returned.

**Method**

```
std::string getFilename
(
);
```

**Return value**

std::string                The filename or an empty string

**removeSpecificModuleNode**

The removeSpecificModuleNode deletes the specified node and all of its children.

**Method**

```
void removeSpecificModuleNode
(
    const unsigned short&        moduleNum
);
```

**Parameters**

moduleNum                 The module to remove indexed from 1

**MseConfigReader**

The loadXml method must be called after the MseConfigReader class is instantiated before any of the accessor methods can be used.

**Constructor**

```
MseConfigReader(void);
```

**getElement**

The getElement method searches the DOM data that is loaded in memory for all instances of the tagname requested. The element data for each match is added to a string with a newline separating each.

**Method**

```
MSE_XML_RETURN getElement
(
    const MSE_XML_ELEMENT& tagname,
    std::string& value
);
```

**Parameters**

tagname                    The XML tag to look for

value                      All of the matching element data is stored in this parameter

**Return value**

MSE\_XML\_RETURN            A response code representing whether the getElement command was successful



**getElement**

The getElement method searches the DOM data that is loaded in memory for a specific element based on the tagname requested. The tagname is searched only in the module requested.

**Method**

```
MSE_XML_RETURN getElement
(
    const MSE_XML_ELEMENT& tagname,
    std::string& value,
    const unsigned short& moduleNum
);
```

**Parameters**

tagname	The XML tag to look for
value	The matching element data is stored in this parameter
moduleNum	The module to search for the tagname indexed from 1

**Return value**

MSE\_XML\_RETURN A response code representing whether the getElement command was successful

**getElement**

The getElement method searches the DOM data that is loaded in memory for a specific element based on the tagname requested. The tagname is searched only in the specific module and channel requested.

**Method**

```
MSE_XML_RETURN getElement
(
    const MSE_XML_ELEMENT& tagname,
    std::string& value,
    const unsigned short& moduleNum,
    const unsigned short& channelNum
);
```

**Parameters**

tagname	The XML tag to look for
value	The matching element data is stored in this parameter
moduleNum	The module to search for the tagname indexed from 1
channelNum	The channel to search for the tagname indexed from 1

**Return value**

MSE\_XML\_RETURN A response code representing whether the getElement command was successful

**getAllElements**

The getAllElements method returns the entire XML file as a string and retains the XML formatting.

**Method**

```
MSE_XML_RETURN getAllElements
(
    std::string& value
);
```

**Parameters**

value	All of the element data is stored in this parameter
-------	---

**Return value**

MSE\_XML\_RETURN A response code representing whether the getAllElements command was successful

**validateElements**

The validateElements method checks if all the module and channel elements exist in the file. If they do not, the file has been invalidated.

**Method**

```
MSE_XML_RETURN validateElements
(
    MSE_XML_ELEMENTS& elementReturn,
    unsigned short& moduleNumReturn,
    unsigned short& channelNumReturn
);
```

**Parameters**

elementReturn	A reference to an enumeration that is filled in with the name of the element that failed validation. This is only used if the return code is MXE_XML_RETURN_TAGNAME_NOT_FOUND.
moduleNumReturn	A reference to an unsigned short that is filled in with the module number of the element that failed validation. This is only used if the return code is MXE_XML_RETURN_TAGNAME_NOT_FOUND.
channelNumReturn	A reference to an unsigned short that is filled in with the channel number of the element that failed validation. This is only used if the return code is MXE_XML_RETURN_TAGNAME_NOT_FOUND.

**Return value**

MSE_XML_RETURN	A response code representing whether the validateElements command was successful
----------------	--

**getSpecificModule**

The getSpecificModule method returns all of the element tag names and corresponding data for a specific module.

**Method**

```
std::string getSpecificModule
(
    const unsigned short& moduleNum
);
```

**Parameters**

moduleNum	The number of the module to request element data of indexed from 1
-----------	--

**Return value**

std::String	A string containing the module element data in csv format separated by new lines. Returns an empty string if there is an error.
-------------	---

**getSpecificChannel**

The getSpecificChannel method returns all of the element tag names and corresponding data for a specific channel.

**Method**

```
std::string getSpecificChannel
(
    const unsigned short& moduleNum,
    const unsigned short& channelNum
);
```

**Parameters**

moduleNum	The number of the module to request element data of indexed from 1
channelNum	The number of the channel to request element data of indexed from 1

**Return value**

std::String	A string containing the channel element data in csv format separated by newlines. Returns an empty string if there is an error.
-------------	---

**getNumModules**

The getNumModules method returns the number of modules in the XML file.

**Method**

```
MSE_XML_RETURN getNumModules
(
    unsigned short& numModules
);
```

**Parameters**

numModules                      The number of the modules in the XML file

**Return value**

MSE\_XML\_RETURN              A response code representing whether the getNumModules command was successful

**getNumChannels**

The getNumChannels method returns the number of channels for a specific module in the XML file.

**Method**

```
MSE_XML_RETURN getNumChannels
(
    const unsigned short& moduleNum,
    unsigned short& numChannels
);
```

**Parameters**

moduleNum                      The module number to request the number of channels indexed from 1  
numChannels                      The number of the channels in the XML file for the specified module

**Return value**

MSE\_XML\_RETURN              A response code representing whether the getNumChannels command was successful

**MseConfigWriter**

The loadXml method must be called after the MseConfigWriter class is instantiated before any of the accessor methods can be used.

**Constructor**

```
MseConfigWriter(void);
```

**setElement**

The setElement method sets the element data for a specific module tagname. This method just sets the data in memory. A subsequent writeFile() must be performed to permanently save the DOM data to the file.

**Method**

```
MSE_XML_RETURN setElement
(
    const MSE_XML_ELEMENT& tagname,
    const std::string& value,
    const unsigned short& moduleNum
);
```

**Parameters**

tagname                      The XML tag to look for  
value                          The value to set the element data to for the tagname  
moduleNum                      The module to use for finding the tagname indexed from 1

**Return value**

MSE\_XML\_RETURN              A response code representing whether the setElement command was successful

**setElement**

The setElement method sets the element data for a specific channel tagname. This method just sets the data in memory. A subsequent writeFile() must be performed to permanently save the DOM data to the file.

**Method**

```
MSE_XML_RETURN setElement
(
    const MSE_XML_ELEMENT& tagname,
    const std::string& value,
    const unsigned short& moduleNum,
    const unsigned short& channelNum
);
```

**Parameters**

tagname	The XML tag to look for
value	The value to set the element data to for the tagname
moduleNum	The module to use for finding the tagname indexed from 1
channelNum	The channel to use for finding the tagname indexed from 1

**Return value**

MSE\_XML\_RETURN A response code representing whether the setElement command was successful

**writeFile**

The writeFile method writes the DOM data to the file loaded from the loadXml method.

**Method**

```
MSE_XML_RETURN writeFile
(
);
```

**Return value**

MSE\_XML\_RETURN A response code representing whether the writeFile command was successful

**writeFile**

The writeFile method writes the DOM data to the desired file.

**Method**

```
MSE_XML_RETURN writeFile
(
    const std::string& filename
);
```

**Parameters**

filename	The filename to write the DOM data to
----------	---------------------------------------

**Return value**

MSE\_XML\_RETURN A response code representing whether the writeFile command was successful

## C Functions

The Configuration file C functions can be found in the MseConfigFileWrapper.h file

### MseConfigFileCreate

Creates a MseConfigWriter object and returns a pointer to it.

#### Function

```
MseConfigFilePtr MseConfigFileCreate
(
);
```

#### Return value

The return value delivers a pointer to the MseConfigWriter object that was created.

### MseConfigFileDelete

Deletes the MseConfigWriter object that was passed in.

#### Function

```
void MseConfigFileDelete
(
    MseConfigFilePtr    object
);
```

#### Parameters

object                    A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function

### MseConfigFileLoadXml

Load the module config XML file into memory.

#### Function

```
MSE_XML_RETURN MseConfigFileLoadXml
(
    MseConfigFilePtr    object,
    char*                filename,
);
```

#### Parameters

object                    A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function  
filename                    The path and filename of the module config XML file

#### Return value

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

### MseConfigFileReloadXml

Reload the module config XML file into memory.

#### Function

```
MSE_XML_RETURN MseConfigFileLoadXml
(
    MseConfigFilePtr    object
);
```

#### Parameters

object                    A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function

#### Return value

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileGetFilename**

Get the filename that is loaded into memory.

**Function**

```
MSE_XML_RETURN MseConfigFileGetFilename
(
    MseConfigFilePtr    object,
    char*                filename,
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
filename	A pointer to the location where the filename will be stored

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileDecodeErrorType**

Decodes a MSE\_XML\_RETURN type as a string.

**Function**

```
MSE_XML_RETURN MseConfigFileDecodeErrorType
(
    MseConfigFilePtr    object,
    char*                decodedErrorType,
    MSE_XML_RETURN      type
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
decodedErrorType	A pointer to the location where the decoded error type will be stored
type	The MSE_XML_RETURN enumeration to decode

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileDecodeElementType**

Decodes a MSE\_XML\_ELEMENTS type as a string.

**Function**

```
MSE_XML_RETURN MseConfigFileDecodeElementType
(
    MseConfigFilePtr    object,
    char*                decodedElementType,
    MSE_XML_ELEMENTS    type
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
decodedElementType	A pointer to the location where the decoded element type will be stored
type	The MSE_XML_ELEMENTS enumeration to decode

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileGetElement**

Get the data for all instances of the desired element.

**Function**

```
MSE_XML_RETURN MseConfigFileGetElement
(
    MseConfigFilePtr    object,
    char*                data,
    MSE_XML_ELEMENTS    tagname
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
data	A pointer to the location where the data will be stored. Each instance is separated by a newline.
tagname	The element to find

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileGetModuleElement**

Get the data for the desired element of a module.

**Function**

```
MSE_XML_RETURN MseConfigFileGetModuleElement
(
    MseConfigFilePtr    object,
    char*                data,
    short               module,
    MSE_XML_ELEMENTS    tagname
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
data	A pointer to the location where the data will be stored
module	The module to look for the element indexed from 1
tagname	The element to find

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileGetChannelElement**

Get the data for the desired element of a channel within a module.

**Function**

```
MSE_XML_RETURN MseConfigFileGetChannelElement
(
    MseConfigFilePtr    object,
    char*                data,
    short               module,
    short               channel,
    MSE_XML_ELEMENTS    tagname
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
data	A pointer to the location where the data will be stored
module	The module to look for the element indexed from 1
channel	The channel to look for the element indexed from 1
tagname	The element to find

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileGetAllElements**

Get the entire XML file.

**Function**

```
MSE_XML_RETURN MseConfigFileGetAllElements
(
    MseConfigFilePtr    object,
    char*                data
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
data	A pointer to the location where the data will be stored

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileValidateElements**

The MseConfigFileValidateElements method checks if all the module and channel elements exist in the file. If they do not, the file has been invalidated.

**Method**

```
MSE_XML_RETURN MseConfigFileValidateElements
(
    MseConfigFilePtr    object,
    MSE_XML_ELEMENTS*   elementReturn,
    unsigned short*     moduleNumReturn,
    unsigned short*     channelNumReturn
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
elementReturn	A pointer to the location where the name of the element that failed validation will be stored. This is only used if the return code is MXE_XML_RETURN_TAGNAME_NOT_FOUND.
moduleNumReturn	A pointer to the location where the module number of the element that failed validation will be stored. This is only used if the return code is MXE_XML_RETURN_TAGNAME_NOT_FOUND.
channelNumReturn	A pointer to the location where the channel number of the element that failed validation will be stored. This is only used if the return code is MXE_XML_RETURN_TAGNAME_NOT_FOUND.

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileGetAllElements**

Get the entire XML file.

**Function**

```
MSE_XML_RETURN MseConfigFileGetAllElements
(
    MseConfigFilePtr    object,
    char*                data
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
data	A pointer to the location where the data will be stored



**MseConfigFileGetSpecificModule**

Get all of the elements and data for the desired module.

**Function**

```
MSE_XML_RETURN MseConfigFileGetSpecificModule
(
    MseConfigFilePtr    object,
    char*               data,
    short               module
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
data	A pointer to the location where the data will be stored. The data for each element is returned as the element name, followed by a comma, followed by the data, followed by a newline.
module	The module to look for elements indexed from 1

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileGetSpecificChannel**

Get all of the elements and data for the desired channel of a module.

**Function**

```
MSE_XML_RETURN MseConfigFileGetSpecificModule
(
    MseConfigFilePtr    object,
    char*               data,
    short               module,
    short               channel
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
data	A pointer to the location where the data will be stored. The data for each element is returned as the element name, followed by a comma, followed by the data, followed by a newline.
module	The module to look for elements indexed from 1
channel	The channel to look for elements indexed from 1

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileGetNumModules**

Get the number of modules in the XML file.

**Function**

```
MSE_XML_RETURN MseConfigFileGetNumModules
(
    MseConfigFilePtr    object,
    short*              numModules
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
numModules	A pointer to the location where the number of modules will be stored

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileGetNumChannels**

Get the number of channels in the XML file.

**Function**

```
MSE_XML_RETURN MseConfigFileGetNumChannels
(
    MseConfigFilePtr    object,
    short               module,
    short*              numChannels
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
module	The module to look for the number of channels indexed from 1
numChannels	A pointer to the location where the number of channels will be stored

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileSetModuleElement**

Set the value for a module element.

**Function**

```
MSE_XML_RETURN MseConfigFileSetModuleElement
(
    MseConfigFilePtr    object,
    char*               data,
    short               module,
    MSE_XML_ELEMENTS    tagname
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
data	The value to set the element to
module	The module to set the element of indexed from 1
tagname	The element to set the value of

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileSetChannelElement**

Set the value for a channel element.

**Function**

```
MSE_XML_RETURN MseConfigFileSetModuleElement
(
    MseConfigFilePtr    object,
    char*               data,
    short               module,
    short               channel,
    MSE_XML_ELEMENTS    tagname
);
```

**Parameters**

object	A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function
data	The value to set the element to
module	The module to set the element of indexed from 1
channel	The channel to set the element of indexed from 1
tagname	The element to set the value of

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileRemoveModule**

Removes the specified module from the XML file.

**Function**

```
MSE_XML_RETURN MseConfigFileRemoveModule
(
    MseConfigFilePtr    object,
    const short         moduleNum
);
```

**Parameters**

object                    A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function  
moduleNum                The module to remove from the XML file indexed from 1

**MseConfigFileWriteFile**

Write the data in memory to the file.

**Function**

```
MSE_XML_RETURN MseConfigFileWriteFile
(
    MseConfigFilePtr    object
);
```

**Parameters**

object                    A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.

**MseConfigFileWriteNewFile**

Write the data in memory to a new file.

**Function**

```
MSE_XML_RETURN MseConfigFileWriteNewFile
(
    MseConfigFilePtr    object,
    char*               filename
);
```

**Parameters**

object                    A pointer to the MseConfigWriter object that was created by the MseConfigFileCreate function  
filename                 The path and filename of the new file to write the XML data to

**Return value**

The return value delivers a MSE\_XML\_RETURN representing whether the function call was successful.



# 3

**System integrity**

## 3.1 About System Integrity

System integrity refers to the monitoring of the Ethernet, programming status, voltages, current, and temperature in the module.

There is a 4 byte value called `systemIntegrity` that is used to store the masked warnings and errors that are active. The `systemIntegrity` variable is only used for the voltages, current, and temperature monitoring. A value of 0 means that there are no values out of range.

The system integrity has warning and error ranges for the voltages, current, and temperature monitoring. If the integrity exceeds a warning or error range, a bit is set in the system integrity. The client code can request the `systemIntegrity` value and check if any of the bits are set.

The power and status LEDs on the module are set based on system integrity errors. The warning limit is only used by clients and does not affect the LEDs.

The `getIntegrity()` method is used to check the `systemIntegrity`.

The `clearIntegrityErrors()` method is used to clear integrity errors.

## 3.2 Obtaining IP Address

The status LED will flash from green to off at a 5 Hz interval when an IP address is being obtained by DHCP or static addressing and then applied to the FNET stack.

The LED will change to a 2Hz interval once the IP address is obtained.

## 3.3 Waiting for Client

The status LED will flash from green to off at a 2 Hz interval when an IP address is obtained and the module is waiting for a connection to a client.

The LED will change to solid green once the connection to a client is obtained.

## 3.4 Duplicate IP

The status LED will flash from green to red at a 2 Hz interval when an IP address is already used.

One of the modules with the identical IP address must be disconnected and the IP address of the module that remains connected must be set to a unique value.

## 3.5 Programming Error

The LED will alternate the power and status LEDs 10 times with a 250ms interval if the MSE firmware has a checksum error after completion.

## 3.6 Ethernet Chip

The LEDs will both flash red 30 times with a 250ms interval if the Ethernet chip has malfunctioned.

## 3.7 Current

The current is only monitored on the power supply modules. The current shows how much current is being drawn by subsequent modules. If the amount of current being drawn is over the error limit, the power LED will stay red until the condition and systemIntegrity value is cleared.

The current sets a warning if > 2.0 amps for the 100-240V power supply.

The current sets a warning if > 2.9 amps for the 24V power supply.

The current sets an error if > 2.1 amps for the 100-240V power supply.

The current sets an error if > 3.0 amps for the 24V power supply.

## 3.8 24V

The power LED will toggle between red and green at a rate of once a second until the error is cleared. The status LED is unaffected.

The 24V sets an error if not between 20 and 28 V.

The 24V sets a warning if not between 21.5 and 26.5V.

## 3.9 5V

The power LED will toggle between red and green at a rate of once every 2 seconds until the error is cleared. The status LED is unaffected.

The 5V sets an error if not between 4.8 and 5.5 V.

The 5V sets a warning if not between 4.9 and 5.3V.

## 3.10 3.3V

The 3.3V is just used for informational purposes and does not affect the LEDs. If the 3.3V is bad, the CPU will stop functioning.

## 3.11 CPU temperature

The power LED will toggle between red and green at a rate of once every 3 seconds until the error is cleared. The status LED is unaffected.

The temperature sets an error if not between -40 and 110 Celsius.

The temperature sets a warning if not between -30 and 100 Celsius.

## 3.12 Non-Volatile Memory Backup Failure

The power LED will toggle between red and green at a rate of once every 10 seconds until the error is cleared. The status LED is unaffected.

This error signifies that the non-volatile configuration data in FRAM memory could not be read and the backup stored in FLASH memory could not be copied into FRAM. If this error occurs, the module memory has become corrupted. Contact HEIDENHAIN Support for instructions.

The configuration data contains the module ID, hardware ID, IP address, netmask, MAC address, firmware checksum, programming state, bootloader versioning information, and DHCP choice.





# 4

**Diagnostic modes**

## 4.1 About Diagnostic Modes

Diagnostics are performed within each module in order to check for warnings and errors regarding voltages, temperature, and encoders.

The diagnostics are checked every 200 ms.

There are 4 diagnostic modes. They are full, status, minimal, and off.

Changing the diagnostic modes has a large benefit for EnDat modules since they can cause a lot of latency when running at 'Full'.

The diagnostic mode is changed with the `enableDiags()` method.

## 4.2 Full

Full diagnostics will perform function reserve checking for EnDat modules.

The 1Vpp and TTL modules will perform the same diagnostics as 'Status'.

The other modules will perform the same diagnostics as 'Minimal'.

The function reserves take 11ms per axis to be performed. The send/receive time for a command to the module will take up to 11 ms if sent while the function reserve is being obtained. It will take 44ms for a 4x EnDat module to have all four axes updated and 88 ms for an 8x EnDat module to update all 8 axes.

The read of the channel's function reserves will be checked once the 200 ms diagnostic timer expires and are broken up so that one channel is read each time through the main loop. This allows for UDP messages to be sent at any time between reads.

## 4.3 Status

The EnDat modules will check for warnings and errors. It takes between 1-2 ms to obtain the error information for a single channel. This means that it will take 4-8 ms for a 4x module and 8-16 ms for an 8x module. If a V2.1 EnDat encoder is being used, this delay cannot be removed. If a V2.2 EnDat encoder is being used, changing the diagnostic mode to minimal will skip checking the warnings and errors and the delay will be 250us per axis.

The errors and warnings for the EnDat module will be read each time the positions are updated (not at the normal 200ms timing of the diagnostic timer). This is because the error and warning information is combined with the read counts command when using the EnDat V2.1 protocol and helps in determining if the count is valid.

The 1Vpp modules will check the counter register to verify amplitude, edge distance errors, and filter spikes. The TTL modules will check the counter register to verify edge distance errors.

The 1Vpp and TTL error checking does not affect the timing since it is performed much faster than the time it takes to request a UDP command.

The read of the channels for both the EnDat and 1Vpp modules are broken up so that one channel is read each time through the main loop. This allows for UDP messages to be sent at any time between reads.

The other modules will perform the same diagnostics as 'Minimal'.

## 4.4 Minimal

Minimal diagnostics will perform checking of the system integrity. The system integrity is described in the 'System Integrity' section.

The system integrity checking does not affect the timing since it is performed much faster than the time it takes to request a UDP command.

## 4.5 None

None of the diagnostics will be performed in this mode.

# 5

**Trigger line**

## About the Trigger Line

### 5.1 About the Trigger Line

A single trigger line is utilized that is electrically connected to all modules. This line is activated by the base module when one of 5 latch requests are received. The base module keeps track of the latch requests until the trigger line is cleared. The trigger line is used for latching data within a small timeframe or for signaling purposes.

The first 3 latch requests are used for software latches. Software latches are received via UDP from an external application. The 4th latch request is used for footswitch 1. The 5th latch request is used for footswitch 2. The footswitch latch requests can be activated by an external footswitch or other device that is attached to the footswitch connector on the base module. The footswitch latch requests can also be activated by a software request if needed.

Once the output line is active, all of the modules will be able to read the line and then latch the data accordingly. The EnDat, 1Vpp, TTL, and LVDT modules will latch the latest count for each axis and store it in buffers. The I/O and pneumatic modules will store the current I/O states. The analog module will store the latest voltage and current.

There is only 1 buffer for storing latched data. The latch must be cleared before the buffer will overwrite the data. The base module must be cleared before all other modules or the latch will immediately trigger again.

The client software must send a read or a 'clear latch' command to clear the latch.

### 5.2 Software Latency

Latency from a software latch is based on the time it takes for the UDP packet to be sent plus the time it takes for the module to store the data.

Most software commands will take less time than it takes for UDP packet transmission which will not add any more overhead other than the UDP transit time.

The EnDat modules level of diagnostics should be changed depending on the throughput desired. The 'Full' diagnostic mode will affect the latch timing dramatically and the 'Status' diagnostic mode will affect the latch timing slightly. The levels are described in the 'Diagnostic Modes' section.

All other modules do not have additional latency.

#### Full Diagnostics

Full diagnostics should not be used if accuracy of the EnDat positions is desired faster than once every 100ms. See the 'Diagnostic Modes' section for more information.

#### Status Diagnostics

Status diagnostics should not be used if accuracy of the EnDat positions is desired faster than once every 8-16ms. See the 'Diagnostic Modes' section for more information.

#### Minimal Diagnostics

Minimal diagnostics should be used if accuracy of the EnDat positions is desired at the maximum rate. The maximum rate is 250us per axis which is 1ms for a 4x module and 2 ms for a 8x module.

## 5.3 Debouncing Latency

The 2 footswitch lines assume that a footswitch is attached to the hardware input lines. The default debouncing time for each line on an EnDat or 1Vpp base module is 10ms. This will add 10ms to the amount of time it takes for the input to be detected. This time does not affect the ability to handle UDP requests since it utilizes a timer instead of holding up the CPU. The debouncing for the TTL module is negligible.

The debouncing time for each footswitch line can be changed to a value between 0 – 20ms. A value of zero will disable debouncing.

Disabling debouncing is useful if a different device is attached to the footswitch input that does not have need for debouncing.

The debouncing latency only occurs when the hardware lines are polled to check for change in state. There is no debouncing if the software simulates a footswitch by setting the 4th or 5th trigger output.

Debouncing can be modified on an EnDat or 1Vpp base module with the `setLatchDebouncing()` method.

## 5.4 Setting a Trigger

The footswitch can be used to set the trigger and the base module will store that latch 4 or 5 occurred.

The `setLatch()` method is used to cause a trigger to be set. Any of the latch requests can cause a latch trigger with this command.

## 5.5 Determining Which Latches are Set

The `getLatch()` method is used to determine which latches are set. The base module will respond with all of the latches that are active. The other modules will just respond with whether or not the latch is currently active.

## 5.6 Reading the Latched Data

The `getCounts()` of the `MseModule`, `getCounts()` and `getPosition()` of the `MseEndatModule` and `MseTtlModule` classes, `getPosition()` of the `Mse1VppModule` and `MseLvdModule` classes, `getVoltage()`, `getCurrent()`, `getValues()`, and `getScaledValues()` of the `MseAnalogModule` class, `getIO` of the `MseloModule` class, and `getOutput` of the `MsePneumaticModule` are used to read the latched data after a trigger is set. Reading the latched data will clear the latch. The base module must be read first or have its latch cleared before reading from other modules.

## 5.7 Clearing a Trigger Manually

The `setLatch()` method is used to reset the desired trigger.

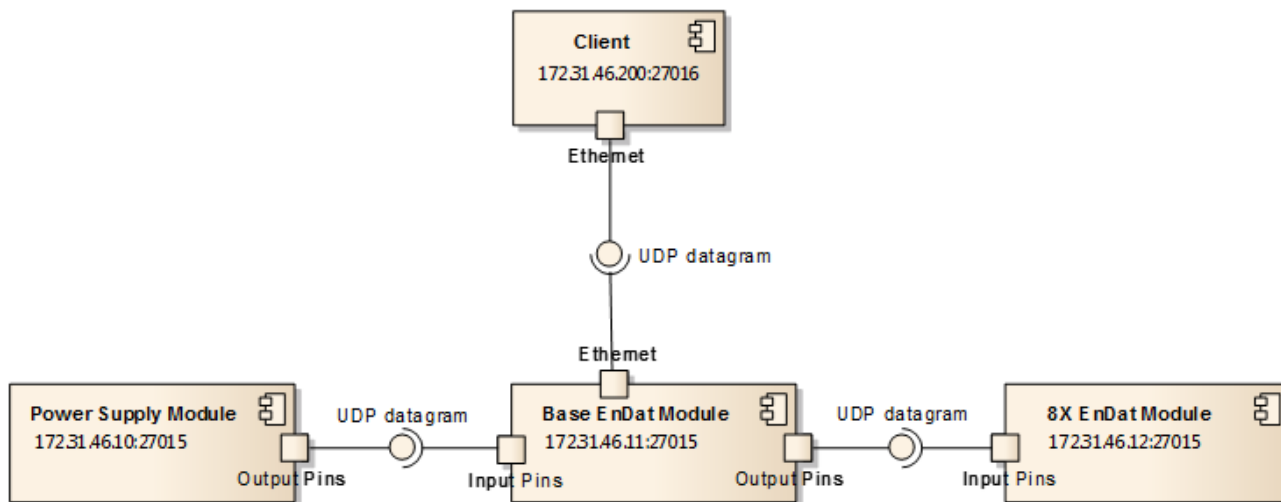


# 6

**Module Networking and  
Throughput**

## 6.1 Overview

The MSE 1000 modules utilize UDP for communication to a client. A client will send a UDP packet to a specific module, over the 10/100 Ethernet connection, and receive a response. The Ethernet connection is located on the base module and the packet is sent to all of the other modules in the chain via the module input and output pins that connect the modules.



The above diagram shows a module chain that consists of a power supply, Base EnDat, and 8X EnDat module. The IP address and port for each module as well as the client is shown.

Each module has an Ethernet chip that will receive a packet as long as the IP address and socket are a match with the values in the packet. A module will perform the request based on the command in the packet and then send a response back to the client. An example would be if the client (IP address 172.31.46.200, port 27016) sends a UDP\_GET\_COUNTS message to the base EnDat module (IP address 172.31.46.11, port 27015). The base EnDat module would copy the count data into a message and then send the message back to the client.

## 6.2 Effects on Throughput

Throughput can be affected by pre-conditions, module conditions, and post-conditions.

### Pre-conditions:

#### Latching

- The software and debouncing latency of the latching affects the throughput. See the Trigger line on page 233 for more information.
- The latch must be cleared before reading the latched values or else the value will immediately latch again. This will not take extra time if the base module reads latched values before reading from the other modules because reading latched values will clear the latch. If you do not need values from the base module then the latch can simply be cleared.

### Module conditions:

- The send/receive transactions take different amounts of time based on the type of module.
- The time it takes to read a count/value from the EnDat, TTL, 1 Vpp, LVDT, I/O, Pneumatic, or Analog module electronics.
- The time it takes to perform diagnostics (see the Diagnostic modes section).
- The location of the module in the chain.



**Post-conditions:**

- The setNetworkDelay method that is supplied by the MSElibrary defaults to 0 ms and is performed after a send/receive transaction has completed.  
This delay can be set to a number of milliseconds specific to a client's needs. An example is setting the delay to 4 ms in order to read only live data for all channels on a single LVDT module with minimal oversampling.

**Operating system context switching**

- You must not give up priority unless throughput speeds less than the system's time slice, or quantum, are acceptable. A typical delay of 16.66 ms will occur if you give up priority after a packet is received.
- A setNetworkDelay of greater than 16 ms will give up priority. A value less than or equal to 16 ms will use a busy loop in order to not give up priority in order to simplify the end user development.

**Post processing of the received data**

- It can take negligible time to store the data to a buffer or container and utilize it later.
- Writing the data to a file or performing complex algorithms can be done as soon as the data arrives, but this will cost time depending on the process used.

**Timeout**

- UDP is not guaranteed. As a result, a sent packet may not get a response. If this occurs, then a delay will be incurred based on the value set with the setUdpTimeout method. The timeout should be set back to the default when done with throughput sensitive procedures since there are other commands that take much longer to complete, such as changing the IP address.
- Retries can be done manually or the setUdpNumRetries method can be used to do them automatically.

## 6.3 Throughput Test Results

**Setup**

- Diagnostics disabled
- No network delay
- Keeping priority
- Negligible post processing
- Populated channels
- Non-base modules are all located as the second module in the chain

Module	Channels	Average Send/ Receive Time (ms)	Data Throughput (packets/second)	Live Data Throughput (packets/second)	Live Data Throughput (channels/ second)
Base EnDat (747499-01)	4	0.917	1082	400	1600
4X EnDat (747503-01)	4	0.965	1028	400	1600
8X EnDat (747504-01)	8	1.224	890	185	1480
Base 1Vpp (747500-01)	4	0.906	1092	639	2556
4X 1Vpp (747505-01)	4	0.919	1076	529	2116
8X 1Vpp (747506-01)	8	1.028	962	527	4216
Base TTL (747511-01)	4	0.863	1144	Same	4576
4XTTL (747512-01)	4	0.845	1170	Same	4680
8XTTL (747513-01)	8	0.906	1092	Same	8736
Analog (747509-01)	2	1.31	756	N/A	N/A
Pneumatic (747508-01)	1	0.753	1310	N/A	N/A
I/O (747507-01)	8	0.751	1313	N/A	N/A

Module	Channels	Average Send/Receive Time	Data Throughput (packets/second)	Live Data Throughput (packets/second)	Live Data Throughput (channels/second)	Oversampling
LVDT (747514-01)	8	1.247	794	180	1440	Minimal
LVDT (747514-01)	8	2.35	423	105	840	Medium
LVDT (747514-01)	8	4.551	219	54	432	Full

### Average Send/Receive Time

- Each module has its count, voltage, etc., read 10,000 times in succession and the average send/receive time is obtained.
- This process is repeated 10 times to show repeatability.
- The results of the throughput test are shown in the table.
- This is just the average, the send/receive time may be faster or slower based on many factors.

### Data Throughput (packets/second)

- This is the number of packets that were able to be sent in a second.
- This may have stale data and only refers to the maximum throughput that can be obtained per second.
- This throughput information is useful if multiple modules are in the chain since the data will not be stale after cycling through all modules.

### Live Data Throughput (packets/second)

- Requesting data very fast, at the rate shown in 'Average Send/Receive Time', may be returning the same data as the previous request(s). This is considered stale data. It takes time for the internal electronics to store the new values of an encoder, sensor, etc. and for the data to then be retrieved.
- The data in the table is the live data which is obtained by storing the results of the packet into a container and post-processing them to determine how many packets a second can be obtained without duplicates. The encoder, sensor, etc. is moved while the readings take place in order to insure that live readings can be obtained.

### Live Data Throughput (channels/second)

- This is equal to (Live Data Throughput (packets/second) \* Channels).

### Oversampling

- Oversampling is utilized exclusively on the LVDT modules in order to filter out noise.
- More oversampling takes a lot of time and is useful when you do not need the throughput.

## 6.4 Latching Throughput Test Results

### Setup

#### 1 Vpp or EnDat base module

- Signal generator attached to the first footswitch of the base module and set to create a TTL square wave at 400 Hz (trigger once every 2.5 ms).
- The debouncing of the footswitch in the base module was set to 0 ms.

#### TTL base module,

- Signal generator attached to the first footswitch of the base module and set to create a TTL square wave at 1 kHz (trigger once every ms).
- A 4x TTL module with a constantly rotating encoder was used to obtain count values.
- The base module latch line is cleared after each read of the count values of the 4x TTL module.
- The 4x TTL module has its count values read 10,000 times in succession and the number of duplicate readings is obtained. This process is repeated 10 times to show repeatability. As soon as there are no duplicates, the delay used between reading one count and the next is considered the maximum throughput for latched data.

## Results

### 1 Vpp and EnDat base modules

Successfully polled the connected TTL module once every 5 ms for new values, using a 5 ms delay after obtaining the latched values.

### TTL base module

Successfully polled the connected TTL module once every 3 ms for new values, using a 3 ms delay after obtaining the latched values.

## 6.5 Propagation Delay Test Results

### Setup

- Diagnostics disabled
- No network delay
- Keeping priority
- Negligible post processing
- Populated channels
- A 4x EnDat module is moved to different positions in the chain
- The module has it's count read 10,000 times in succession and the average send/receive time is obtained
- This process is repeated 10 times to show repeatability

### Results

There is a cumulative delay between approximately 0.01 and 0.05 ms that should be expected for each module in the chain. The cumulative 0.05 ms delay will be used in the examples to take care of worst case scenarios.

#### Example:

If there is a 4x EnDat module connected to the base module, it will take the average send/receive time for a 4x EnDat, 0.965 ms plus approximately 0.05 ms for the propagation delay, for a response to be received. This would be a total of approximately 1.015 ms. It would take 1.065 ms if there is one module between the base module and the 4x EnDat module. It would take 1.115 ms if there are two modules between the base module and the 4x EnDat module.

## 6.6 Example Chain Configurations

- Throughput for an entire chain is dependent on the number of modules connected.
- Throughput for a small chain is bottlenecked by the slowest module.

### Large Chain

If there are 10 non-power supply modules connected to a chain, it will take at least the send/receive time plus the propagation delays in order to read from them all before a new group of readings can be obtained. The 'live data' delay can be ignored since the data will always be live by the time the module is read from again.

### Small Chain

If there is a chain of only 2 non-power supply modules, a base EnDat module and an LVDT module, it will take at least the time it takes for live data to be available on the LVDT, which is approximately 5.55 ms for minimal oversampling, for each reading.

### Live and Latching with a Base EnDat

A chain with a base EnDat module, 4x TTL, and 8x TTL modules would have the following throughput:

- Live throughput would be 0.917 ms (base EnDat) + 0.845 ms (4x TTL) + 0.906 (8x TTL) + 0.05 ms (4x TTL propagation delay) + 0.1 ms (8x TTL propagation delay) = 2.818 ms to read the entire chain.
- Latching throughput would be 0.917 ms (base EnDat) + 0.845 ms (4x TTL) + 0.906 (8x TTL) + 0.05 ms (4x TTL propagation delay) + 0.1 ms (8x TTL propagation delay) = 2.818 ms to read the entire chain + 2.182 ms (since we need a total of 5 ms before the values can be read again based on the latching test results) = 5 ms.

A chain with a base EnDat module, 4x TTL, and 8x TTL, 4x 1Vpp, 8x 1Vpp, and LVDT modules would have the following throughput:

- Live throughput would be 0.917 ms (base EnDat) + 0.845 ms (4x TTL) + 0.906 (8x TTL) + 0.919 (4x 1Vpp) + 1.028 (8x 1Vpp) + 1.247 (LVDT) + 0.05 ms (4x TTL propagation delay) + 0.1 ms (8x TTL propagation delay) + 0.15 ms (4x 1Vpp propagation delay) + 0.2 ms (8x 1Vpp propagation delay) + 0.25 ms (LVDT propagation delay) = 6.612 ms to read the entire chain.
- Latching throughput would be the same since 6.612 ms exceeds the time needed for the next latch to be seen.



# 7

**Module descriptions**

## 7.1 Power supply modules

The power supply modules are used to supply power to each module connected to its right. The current drawn from modules to the right of the power supply is taken from the power supply. Power supplies do not draw current from modules to their left; this allows a new power supply to be placed in the module chain in order to supply power for the next group of modules.

## 7.2 EnDat modules

The EnDat modules can connect up to 4 or up to 8 channels with HEIDENHAIN EnDat encoders. EnDat encoders have non-volatile memory that allow them to store absolute position without the need for referencing as well as encoder information such as resolution, signal period, and more. EnDat encoders are considered gauge, linear, or rotary. Gauge and linear are treated the same in the library and firmware and are enumerated simply to help the user differentiate between the two.

## 7.3 1Vpp modules

The 1Vpp modules can connect up to 4 or up to 8 channels with HEIDENHAIN 1Vpp encoders. HEIDENHAIN 1Vpp encoders do not have non-volatile memory that allow them to store absolute position and therefore need to be manually referenced on each power up of the module if the encoder offers reference marks and the user wants to utilize absolute positioning. The 1Vpp modules can perform referencing for linear and rotary encoders. The encoder information such as device type, signal period, and line count must be configured each time an object is instantiated by the MSElibrary. 1Vpp encoders are considered gauge, linear, or rotary.

HEIDENHAIN 1Vpp modules use the encoders signal period in conjunction with the quadrature signal and internal electronics to compute the resolution of a linear encoder. The line count in conjunction with the quadrature signal and internal electronics is used to compute the resolution of rotary encoders.

## 7.4 TTL modules

The TTL modules can connect up to 4 or up to 8 channels with HEIDENHAIN TTL encoders. HEIDENHAIN TTL encoders do not have non-volatile memory that allow them to store absolute position and therefore need to be manually referenced on each power up of the module if the encoder offers reference marks and the user wants to utilize absolute positioning. The TTL modules can perform referencing for linear encoders only. The encoder information such as device type, signal period, and line count must be configured each time an object is instantiated by the MSElibrary. The interpolation of the TTL encoders must be configured for the TTL encoders. TTL encoders are considered gauge, linear, or rotary.

HEIDENHAIN TTL modules use the encoders signal period in conjunction with the quadrature signal and interpolation value to compute the resolution of a linear encoder. The line count in conjunction with the quadrature signal and interpolation value is used to compute the resolution of rotary encoders.

## 7.5 LVDT modules

The LVDT modules can connect up to 8 channels with Solatron/Tesa compatible, Mahr compatible, or Marposs compatible LVDT sensors depending on the module. The LVDT modules each utilize an excitation frequency and excitation voltage to source the LVDT sensors that are connected. The excitation frequency and voltage are stored in non-volatile memory in the module and therefore only need to be configured each time different sensors are added since the optimal frequency and voltage is dependent on the sensor itself. The gain of each sensor must be configured at least once and is stored in non-volatile memory in the module. The resolution of the sensor is not saved in non-volatile memory and must be configured each time the LVDT module is instantiated. The resolution must be calibrated at least once and should be calibrated as often as required by the user. The LVDT modules have added latency due to the subsequent electronics. Refer to the LVDT\_OVERSAMPLING\_CHOICES enumeration on page 49 for timing information.

## 7.6 Analog module

The Analog modules can connect up to 2 channels with 4-20 mA or +/- 10V analog sensors. The raw current or voltage value of the attached sensor can be retrieved, or the library can be configured to apply a resolution and offset to the value. The resolution and offset need to be configured each time the module is instantiated. The raw value utilizes a running average of the last 100 value and can be modified if this is too much.

## 7.7 I/O modules

The I/O modules can connect up to 4 inputs and 4 outputs. There is no need for software configuration.

## 7.8 Pneumatic module

The pneumatic module has a single output that is used to enable or disable a solenoid. Enabling the solenoid will allow air to pass through the module. There is no need for configuration.

# 8

**Operating principles**

## 8.1 Overview

The MSE 1000 consists of Endat, 1Vpp, TTL, LVDT, Analog, I/O, Pneumatic, and Power Supply modules. Some of the operating principles are common to all modules; others are specific to the type of module.

Examples of the operating principles can be obtained in the example code sections later in the document or in the examples code supplied with the installation of the MSElibrary.

## 8.2 Initialization

The modules communicate over Ethernet using the UDP protocol. Each module has its own IP address and needs to be initialized after it has been instantiated.

Initialization of the modules does the following:

- Creates a UDP socket for communicating with the module
- Sends a UDP\_OPEN command to the module at the specified IP address and receives a MSE1000ConnectResponse response
- Sets the module in asynchronous mode if requested
- Fills in the ModuleData information
- Sets the module communication as initialized. If this has not been completed, calls into the module will fail with a return code of RESPONSE\_COMM\_NOT\_INITIALIZED.

The LVDT module has additional configuration of the excitation voltage and excitation frequency used for all of the sensors. The excitation voltage and frequency is stored in non-volatile memory in the module and needs to be configured on initial use whenever the types of sensors are changed. The setExcitationFrequency and setExcitationVoltage C++ methods, and the MseLvdModuleSetExcitationFrequency and MseLvdModuleSetExcitationVoltage C functions are used for setting the excitation frequency and voltage.

### C++ Initialization

The MseInterface class allows for ease in initializing by offering the createChain and addModule methods. These methods are only available for C++ developers and perform the instantiation of the classes as well as the initialization.

Instantiating an object in C++ refers to creating an instance of the desired class in order to start initializing and/or accessing its methods.

The example “Creating a Chain via Broadcasting” on page 269 shows how to initialize using createChain.

The example “Creating a Chain Manually” on page 270 shows how to initialize using addModule.

All instantiated modules can use the initializeModule C++ method in order to communicate to a module instead of creating a chain.

### Non-C++ Initialization

Non-C++ developers must create their own module objects by instantiating the desired classes and using the initialization methods for each module.

The C calls will wrap the instantiation of the object in the create calls which make the pointer to the object accessible through the return parameter. Refer to “Mse1VppModuleCreate” on page 128. Non-C++ can utilize the C calls to work with all of the functionality in the library.

The C calls for simply instantiating and then initializing are:

Module	Instantiation	Initialization
1Vpp	Mse1VppModuleCreate	Mse1VppModuleInitialize
Analog	MseAnalogModuleCreate	MseAnalogModuleInitialize
EnDat	MseEndatModuleCreate	MseEndatModuleInitialize
I/O	MseIoModuleCreate	MseIoModuleInitialize
LVDT	MseLvdModuleCreate	MseLvdModuleInitialize
Pneumatic	MsePneumaticModuleCreate	MsePneumaticModuleInitialize
TTL	MseTtlModuleCreate	MseTtlModuleInitialize

The example “Creating a Chain via Broadcasting” on page 269 shows how to instantiate and initialize using C functions without a chain.



## 8.3 Configuring the Channels

Most of the modules have the ability to attach devices to one or more channels. The power supply modules do not have any channels. The I/O and Pneumatic modules do not need to be configured through software.

### 1Vpp

The 1Vpp module must set the error compensation, UOM, encoder type, signal period or line count, scaling, device offset, rotary format, and counting direction.

Refer to “1Vpp methods and functions” starting on page 122 for the methods and functions listed below.

C++ methods:

- setErrorCompensation
- setUom
- setEncoderType
- setSignalPeriod
- setLineCount
- setScaling
- setDeviceOffset
- setRotaryFormat
- setCountingDirection
- initAbsolutePosition
- isReferencingComplete
- getReferencingState

C functions:

- Mse1VppModuleSetErrorCompensation
- Mse1VppModuleSetUom
- Mse1VppModuleSetEncoderType
- Mse1VppModuleSetSignalPeriod
- Mse1VppModuleSetLineCount
- Mse1VppModuleSetScaling
- Mse1VppModuleSetDeviceOffset
- Mse1VppModuleSetRotaryFormat
- Mse1VppModuleSetCountingDirection
- Mse1VppModuleStartReferencing
- Mse1VppModuleGetReferencingComplete
- Mse1VppModuleGetReferencingState

Refer to “Setting the Encoder Information” on page 272 for a channel configuration example.

Refer to “Referencing” on page 257 for information on referencing.

Refer to “Referencing 1Vpp Linear Encoder” on page 276 for a referencing example.

The signal period and reference mark type for many of the HEIDENHAIN 1Vpp encoders is shown in the following table:

Encoder	Signal period	Reference marks
ST 128x	20	Single
ST 308x	20	Single
LS 388C	20	1000
LS 688C	20	1000
LS 187 LS 187C	20	Single Coded/1000
LS 487 LS 487C	20	Single Coded/1000
LB 382C	40	Coded/2000
LF 183 LF 183C	4	Single Coded/5000
LF 483 LF 483C	4	Single Coded/5000

## EnDat

The EnDat module only allows for setting the error compensation and UOM. EnDat encoders have non-volatile memory that allow for detection of all other values.

Refer to “EnDat methods and functions” starting on page 101 for the methods and functions listed below.

C++ methods:

- setErrorCompensation
- setUom
- setScaling
- setDeviceOffset
- setRotaryFormat

C functions:

- MseEndatModuleSetErrorCompensation
- MseEndatModuleSetUom
- MseEndatModuleSetScaling
- MseEndatModuleSetDeviceOffset
- MseEndatModuleSetRotaryFormat

Refer to “Setting the Encoder Information” on page 272 for a 1Vpp channel configuration example. The EnDat configuration is performed in the same manner except that it only utilizes the error compensation and UOM.

## TTL

The TTL module must set the error compensation, UOM, encoder type, signal period or line count, interpolation, scaling, device offset, rotary format, and counting direction.

Refer to “TTL methods and functions” starting on page 191 for the methods and functions listed below.

C++ methods:

- setChannelPresence
- setErrorCompensation
- setUom
- setEncoderType
- setSignalPeriod
- setLineCount
- setScaling
- setDeviceOffset
- setRotaryFormat
- setCountingDirection

C functions:

- MseTtlModuleSetChannelPresence
- MseTtlModuleSetErrorCompensation
- MseTtlModuleSetUom
- MseTtlModuleSetEncoderType
- MseTtlModuleSetSignalPeriod
- MseTtlModuleSetLineCount
- MseTtlModuleSetScaling
- MseTtlModuleSetDeviceOffset
- MseTtlModuleSetRotaryFormat
- MseTtlModuleSetCountingDirection

The signal period and reference mark type for many of the HEIDENHAIN TTL encoders is shown in the following table:

Encoder	Resolution	Signal period	Interpol. factor	Reference marks
LS 177/477	1 $\mu\text{m}$	20 $\mu\text{m}$	5-fold	single
	0.5 $\mu\text{m}$	20 $\mu\text{m}$	10-fold	single
	0.25 $\mu\text{m}$	20 $\mu\text{m}$	20-fold	single
LS 177C/477C	1 $\mu\text{m}$	20 $\mu\text{m}$	5-fold	coded/1000
	0.5 $\mu\text{m}$	20 $\mu\text{m}$	10-fold	coded/1000
	0.25 $\mu\text{m}$	20 $\mu\text{m}$	20-fold	coded/1000
LS 328C/628C	5 $\mu\text{m}$	20 $\mu\text{m}$	n/a	coded/1000
LS 378C	1 $\mu\text{m}$	20 $\mu\text{m}$	5-fold	coded/1000
	0.5 $\mu\text{m}$	20 $\mu\text{m}$	10-fold	coded/1000
	0.25 $\mu\text{m}$	20 $\mu\text{m}$	20-fold	coded/1000

## LVDT

The LVDT module must set the channel presence, sensor gain, resolution, device offset, scaling, and oversampling.

Refer to “LVDT methods and functions” starting on page 158 for the methods and functions listed below.

C++ methods:

- setChannelPresence
- setSensorGain
- setResolution
- setScaling
- setDeviceOffset
- setOversampling

C functions:

- MseLvdModuleSetChannelPresence
- MseLvdModuleSetSensorGain
- MseLvdModuleSetResolution
- MseLvdModuleSetScaling
- MseLvdModuleSetDeviceOffset
- MseLvdModuleSetOversampling

The LVDT sensors have the ability to teach the sensor gain. This is useful for getting a gain with 80% usage of the total voltage range of the electronics. Too much gain will cause the sensor to become unreliable. Too little gain will cause lower resolution. Calling the teachSensorGain method will start the teach. Polling the getTeachSensorGainFinished can be used to determine when the gain teach has completed. You should hold the sensor at the minimum or maximum traversal while teaching the gain because it uses this location to determine if the sensor is close to the total voltage range of the ADC.

Refer to “LVDT methods and functions” starting on page 158 for the methods and functions listed below.

C++ methods:

- teachSensorGain
- getTeachSensorGainFinished

C functions:

- MseLvdModuleTeachSensorGain
- MseLvdModuleTeachSensorGainFinished

## Analog

The Analog module must set the resolution, offset, scaling, and device offset.

Refer to “Analog methods and functions” starting on page 176 for the methods and functions listed below.

C++ methods:

- setResolution
- setOffset
- computeResolutionAndOffset
- setScaling
- setDeviceOffset
- setNumSamples

C functions:

- MseAnalogModuleSetResolution
- MseAnalogModuleSetOffset
- MseAnalogModuleComputeResolutionAndOffset
- MseAnalogModuleSetScaling
- MseAnalogModuleSetDeviceOffset
- MseAnalogModuleSetNumSamples

The Analog module utilizes the `getScaledValues` method in order to get a value that is converted from voltage or current to the desired type. The `setResolution`, `setOffset`, and `computeResolution` are used for this purpose. The `setScaling` and `setDeviceOffset` methods are used to scale the resulting value and then apply a device offset. The device offset is used mainly for mastering and is made available to the Analog module through a base class.

The `computeResolutionAndOffset` is a convenient method for creating the resolution and offset that will be applied to the voltage or current in order to map the values from the device to user units. An example is a temperature sensor that will give a reading of -40 degrees Celsius, at the -10 V reading of the sensor, to 100 degrees Celsius, at the 10 V reading of the sensor. The user could compute the resolution and offset manually and then enter them, or could call `computeResolutionAndOffset` as follows:

```
double computedResolution = 0.0;
double computedOffset = 0.0;
computeResolutionAndOffset(0, &computedResolution, &computedOffset, -40, 100, -10, 10);
```

The user could correct for a linear error in the computed values by calling `setScaling` with a multiplier that would scale the values the desired amount.

The user could calibrate the 0 degrees by setting the temperature to be measured to a known 0 degrees and calling `getScaledValues` to see what the reading of the sensor is. The user could then call `setDeviceOffset` with the difference between the reading and 0; if the reading is 2, then the user would call `setDeviceOffset(-2)`.

## 8.4 Channel Operations

The 1Vpp, EnDat, TTL, LVDT, and Analog modules can read information about the devices attached to the channel. The I/O modules can read channel inputs and set outputs. The Pneumatic module can read and set its output used for controlling the solenoid valve.

### 1Vpp

The 1Vpp latest or latched position of all channels can be retrieved as counts or in user units. The counts are a representation of the number of interpolated crossings of the signal periods on the encoder that have been passed. The position in user units is the counts multiplied by the resolution and the error compensation value.

Refer to “1Vpp methods and functions” starting on page 122 for the methods and functions listed below.

C++ methods:

- getCounts
- getPositions
- getResolution

C functions:

- Mse1VppModuleGetCounts
- Mse1VppModuleGetPositions
- Mse1VppModuleGetResolution

Refer to “Setting the Encoder Information” on page 272 for reading the counts and positions from a 1Vpp module.

### EnDat

The EnDat latest or latched position of all channels can be retrieved as counts or in user units. The counts are a representation of the number of interpolated crossings of the signal periods on the encoder that have been passed. The position in user units is the counts multiplied by the resolution and the error compensation value.

The EnDat rotary encoders return the current revolution with the getPositions method. The total number of distinguishable revolutions available to the encoder can be retrieved separately.

Refer to “EnDat methods and functions” starting on page 101 for the methods and functions listed below.

C++ methods:

- getCounts
- getPositions
- getResolution
- getDistinguishableRevolutions

C functions:

- MseEndatModuleGetCounts
- MseEndatModuleGetPositions
- MseEndatModuleGetResolution
- MseEndatModuleGetDistinguishableRevolutions

Refer to “Latching” on page 274 for reading the positions from an EnDat module.

### TTL

The TTL latest or latched position of all channels can be retrieved as counts or in user units. The counts are a representation of the number of interpolated crossings of the signal periods on the encoder that have been passed. The position in user units is the counts multiplied by the resolution and the error compensation value.

Refer to “TTL methods and functions” starting on page 191 for the methods and functions listed below.

C++ methods:

- getCounts
- getPositions
- getResolution.

C functions:

- MseTtlModuleGetCounts
- MseTtlModuleGetPositions
- MseTtlModuleGetResolution

## LVDT

The LVDT latest or latched position of all channels can be retrieved as voltage or in user units. The voltage is a representation of the position of the sensor in relation to its NULL, or center, point. The position is calculated as described in the LVDT module's `getPositions` method.

Refer to "LVDT methods and functions" starting on page 158 for the methods and functions listed below.

C++ methods:

- `getVoltage`
- `getPositions`
- `getResolution`

C functions:

- `MseLvdtModuleGetVoltage`
- `MseLvdtModuleGetPositions`
- `MseLvdtModuleGetResolution`

## Analog

The Analog latest or latched position of all channels can be retrieved as voltage, current or in user units. The position is calculated as described in the Analog module's `getPositions` method, and the Configuring the Channels section on page 249.

Refer to "Analog methods and functions" starting on page 176 for the methods and functions listed below.

C++ methods.

- `getVoltage`
- `getCurrent`
- `getValues`
- `getScaledValues`
- `getOffset`
- `getResolution`
- `getScaling`
- `getDeviceOffset`

C functions:

- `MseAnalogModuleGetVoltage`
- `MseAnalogModuleGetCurrent`
- `MseAnalogModuleGetValues`
- `MseAnalogModuleGetScaledValues`
- `MseAnalogModuleGetOffset`
- `MseAnalogModuleGetResolution`
- `MseAnalogModuleGetScaling`
- `MseAnalogModuleGetDeviceOffset`

## I/O

The I/O latest or latched input and output channel values can be retrieved. The input and output value that are retrieved are each single characters that can be masked to determine which bit is set. Individual latest bit values can be retrieved as well. The outputs can be set individually or from the set bits of a single character.

Refer to “I/O methods and functions” starting on page 144 for the methods and functions listed below.

C++ methods:

- getIO
- getInputs
- getOutputs
- setOutput
- setOutputs

C functions

- MseloModuleGetIO
- MseloModuleGetInputs
- MseloModuleGetOutputs
- MseloModuleSetOutput
- MseloModuleSetOutputs

## Pneumatic

The Pneumatic latest or latched output channel value can be retrieved. The output value that is retrieved is a single character that can be used to determine if the output is set. The output can be set as well.

Refer to “Pneumatic methods and functions” starting on page 152 for the methods and functions listed below.

C++ methods:

- getOutput
- setOutput

C functions:

- MsePneumaticModuleGetOutput
- MsePneumaticModuleSetOutput

## Latching 8.5 Latching

Latching is used to accept a command in the base module that will cause the hardware trigger line to be set for all modules. The setting of the trigger line will allow all of the modules to store data in a very close timeframe which can be used by a client for tolerance or other functionality. The modules each store a single value for each channel no matter how many latch lines are set. The modules will be able to store a new value once the latches are all cleared in the base module followed by the remaining modules. Power supplies do not latch any data and do not need to perform any of the following functionality.

Latching consists of the following process:

- The base module waits for a command in order to activate the trigger line that is described in “Trigger line” on page 233.
- The command for activating the trigger is one of three software latch commands that are sent via the MSElibrary or one of the two footswitch latch requests that are inputs on the base module serial port.
- The command used for activating the software latch is setLatch in C++. The LATCH\_OPTIONS enumeration value of LATCH\_COUNT\_SET must be called. The setLatch command with a value of LATCH\_COUNT\_SET can only be used for activating a latch on the base module.
- The commands used for activating the software latch are MseEndatModuleSetLatch, Mse1VppModuleSetLatch, and MseTtlModuleSetLatch in C.
- Once the trigger line is set, all of the modules will store the most recent count, position, voltage, current, or I/O value into memory.
- The latched position can be read using the COUNT\_REQUEST\_OPTION enumeration value of COUNT\_REQUEST\_LATCHED when getting the counts, position, voltage, current, or I/O values shown in “Channel Operations” on page 253.
- The latch can be cleared on each module by reading the latched position or by calling the setLatch method in C++. The LATCH\_OPTIONS enumeration value of LATCH\_COUNT\_RESET must be called. The clearing of the latch must be done on the base module first or else the modules will immediately latch their values again since the trigger line is still active.
- The commands used for manually clearing the software latch are MseEndatModuleSetLatch, Mse1VppModuleSetLatch, MseTtlModuleSetLatch, MseLvdtModuleClearLatch, MseAnalogModuleClearLatch, MseIoModuleClearLatch, and MsePneumaticModuleClearLatch in C.

The footswitch is available asynchronously or can be polled to determine if it has been set. The asynchronous communication, is described on page 261.

Polling for whether any of the latches are set is done through the getLatch method in C++. The base module will indicate which latch is set. Non-base modules will just indicate whether the trigger line caused a latch to occur.

Polling for whether any of the latches are set is done through the MseEndatModuleGetLatches, Mse1VppModuleGetLatches, MseTtlModuleGetLatches, MseAnalogModuleGetLatch, MseLvdtModuleGetLatch, MseIoModuleGetLatch, and MsePneumaticModuleGetLatch functions in C.



## 8.6 Referencing

Some of the 1Vpp and TTL encoders utilize reference marks to obtain an absolute position on the scale. The encoder must be moved across at least one mark for encoders that have a single reference mark and across at least two for encoders that have coded referencing. The spacing between reference marks is dependent on the encoder and based on the signal period or line count of the encoder as well as the type of reference mark. The referencing process is performed in the module and is started, monitored, verified, and stopped through the MSElibrary. Referencing must be done each time the module is powered on.

The type of referencing for an encoder, interpolation (for TTL encoders), signal period, and line count can be obtained from the encoder manual or from the 1Vpp and TTL tables listed in "Configuring the Channels" on page 249.

Referencing is the following process:

- Configure the encoder and set the referencing type using the functions described in "Configuring the Channels" on page 249.
- The command used to start referencing in the module is `initAbsolutePosition` for a 1Vpp encoder or `initReferencing` for a TTL encoder in C++.
- The command used to start referencing in the module is `Mse1VppModuleStartReferencing` for a 1Vpp encoder or `MseTtlModuleStartReferencing` for a TTL encoder in C.
- The encoder can now be moved so that the reference mark(s) are crossed.
- The command used to poll the module to determine when referencing is complete is `isReferencingComplete` for both 1Vpp and TTL encoders in C++.
- The commands used to poll the module to determine when referencing is complete are `Mse1VppModuleGetReferencingComplete` for 1Vpp and `MseTtlModuleGetReferencingComplete` for TTL encoders in C.
- The command used to determine if referencing was successful is `getReferencingState` for both 1Vpp and TTL encoders in C++.
- The commands used to determine if referencing was successful are `Mse1VppModuleGetReferencingState` for 1Vpp and `MseTtlModuleGetReferencingState` for TTL encoders in C.

When referencing is complete and was verified as successful, the count and position requests will now be absolute.

The referencing complete message is available through the asynchronous communication described later in the document and can be used instead of polling to determine if the referencing is complete.

## Module Errors and Warnings 8.7 Module Errors and Warnings

The modules have errors and warnings described in “System integrity” on page 227.

A subset of the warnings and errors consisting of the INTEGRITY\_ENUMS enumeration can be obtained through library calls. The errors and warnings obtained through the library call can be cleared. If the error or warning is still resident, it will be set again immediately.

The command used to get the integrity errors is `getIntegrity` in C++.

The commands used to get whether there is an integrity error or to get the integrity errors are `Mse1VppModuleGetModuleErrorState`, `Mse1VppModuleGetModuleErrors`, `MseAnalogModuleGetModuleErrorState`, `MseAnalogModuleGetModuleErrors`, `MseEndatModuleGetModuleErrorState`, `MseEndatModuleGetModuleErrors`, `MseloModuleGetModuleErrorState`, `MseloModuleGetModuleErrors`, `MseLvdtModuleGetModuleErrorState`, `MseLvdtModuleGetModuleErrors`, `MsePneumaticModuleGetModuleErrorState`, `MsePneumaticModuleGetModuleErrors`, `MseTtlModuleGetModuleErrorState`, and `MseTtlModuleGetModuleErrors` in C.

The `getIntegrity` method has a field that contains the ranges of each integrity error. This is used in order to retrieve the tolerance values that need to be reached before each integrity error is triggered. The `NUM_INTEGRITY_RANGES` constant is the number of integrity ranges returned. The `INTEGRITY_FRAM_ERROR` and `INTEGRITY_FRAM_RECOVERED` enumerations do not utilize a tolerance and so do not have a range entry.

The command used to get the actual values of the power supply current, power supply 24 V supply, non-power supply 5 V supply, 3.3 V supply, and CPU temperature is `getAdcValues` in C++.

The command used to get the actual values of the power supply current, power supply 24 V supply, non-power supply 5 V supply, 3.3 V supply, and CPU temperature are `Mse1VppModuleGetAdcValues`, `MseAnalogModuleGetAdcValues`, `MseEndatModuleGetAdcValues`, `MseloModuleGetAdcValues`, `MseLvdtModuleGetAdcValues`, `MsePneumaticModuleGetAdcValues`, and `MseTtlModuleGetAdcValues` in C.

The module errors can be cleared with the `clearIntegrityErrors` method in C++.

The module errors and channel errors together can be cleared with the `clearAllErrors` method in C++.

The module errors and channel errors together can be cleared with the `Mse1VppModuleClearErrors`, `MseAnalogModuleClearErrors`, `MseEndatModuleClearErrors`, `MseloModuleClearErrors`, `MseLvdtModuleClearErrors`, `MsePneumaticModuleClearErrors`, and `MseTtlModuleClearErrors` in C.

Module errors are cleared automatically when the MSElibrary initializes a module.

## 8.8 Channel Errors and Warnings

The EnDat encoder has the ability to report warnings and error and to keep track of its operating efficiency with Function Reserves.

The TTL encoder only monitors for a single error.

Errors and warnings are cleared automatically when the MSELibrary initializes a module.

### EnDat

The ENDAT\_ERRORS and ENDAT\_WARNINGS enumerations describe the additional errors and warnings. Refer to “Enumerations” starting on page 29.

The errors and warnings for the EnDat encoders can be read with getErrors and getWarnings C++ method.

The errors and warnings for the EnDat encoders can be read with MseEndatModuleGetChannelErrorState, MseEndatModuleGetEndatErrors, MseEndatModuleGetChannelWarningState, and MseEndatModuleGetEndatWarnings C functions.

The EnDat encoder errors and warnings can be cleared with the clearErrorsAndWarnings C++ method.

The module errors and EnDat encoder errors and warnings can be cleared together with the clearAllErrors method in C++.

The module errors and channel errors together can be cleared with the MseEndatModuleClearErrors function in C.

### 1Vpp

The counter errors are used to determine if the microcontroller that is reading the counts is getting an error. The counter errors are described with the COUNTER\_STATUS enumeration. Refer to “Enumerations” starting on page 29.

The counter errors can be obtained with the getChannelStatus C++ method.

The counter errors can be obtained with the Mse1VppModuleGetChannelStatus C function.

The counter error can be cleared with the clearErrorsAndWarnings C++ method.

The module errors and counter errors can be cleared together with the clearAllErrors method in C++.

The module errors and counter errors together can be cleared with the Mse1VppModuleClearErrors function in C.

### TTL

The TTL module reports only whether there is a COUNTER\_STATUS\_EDGE\_DISTANCE\_ERROR. Refer to “Enumerations” starting on page 29.

The counter error can be obtained with the getChannelStatus C++ method.

The counter errors can be obtained with the MseTtlModuleGetChannelErrorState C function.

The counter error can be cleared with the clearErrorsAndWarnings C++ method.

The module errors and counter errors can be cleared together with the clearAllErrors method in C++.

The module errors and counter errors together can be cleared with the MseTtlModuleClearErrors function in C.

## 8.9 Diagnostics

The common diagnostics are described in “Diagnostic modes” on page 231.

The diagnostics default to DIAG\_MODE\_FULL when the modules power up. The DIAG\_MODE\_OPTIONS enumeration offers additional choices. Refer to “Enumerations” starting on page 29.

The diagnostics mode can be changed with the enableDiags C++ method.

The diagnostics mode can be changed with the Mse1VppModuleEnableDiags, MseAnalogModuleEnableDiags, MseEndatModuleEnableDiags, MseIoModuleEnableDiags, MseLvdtModuleEnableDiags, MsePneumaticModuleEnableDiags and MseTtlModuleEnableDiags C methods.

The EnDat, 1Vpp and LVDT modules each have additional diagnostics that can be monitored.

### EnDat

The EnDat module reports EnDat function reserves. The function reserve report how well the absolute tracking, incremental tracking, and position value calculation are being performed. The type of function reserve available is dependent on the encoder. If a function reserve is not available, it will be reported. If the encoder is normally running below 25% of a supported function reserve, it may be in need of servicing. The function reserves can be disabled using the enableDiags method.

The function reserves can be obtained with the getDiag C++ method.

The function reserves can be obtained with the MseEndatModuleGetDiags C function.

### 1Vpp

The 1Vpp module can return the A and B encoder amplitude values for the user to check that the amplitude is close to 1 Vpp, that they are within 90 degrees phase of each other, and for plotting a Lissajou figure of the signals.

The 1Vpp analog diagnostic is enabled with the enableAnalogDiag C++ method.

The 1Vpp analog diagnostic is enabled with the Mse1VppModuleEnableAnalogDiag C function.

The 1Vpp analog diagnostic values are read with the getDiag C++ method.

The 1Vpp analog diagnostic values are read with the Mse1VppModuleGetAnalogDiag C function.

### LVDT

The LVDT diagnostics allow for a single channel to be monitored rather than all 8. This is useful if a lot of samples need to be gathered of an individual sensor since it will be much faster.

The LVDT diagnostic is enabled with the setDiagnosticsEnabled C++ method.

The LVDT diagnostic is enabled with the MseLvdtModuleSetDiagnosticsEnabled C function.

## 8.10 Asynchronous Communication

Asynchronous communication allows for the client to wait on a socket for messages from the modules.

Asynchronous communication works for:

- Obtaining the IP addresses, netmasks, ports, MAC addresses, DHCP setting, and serial number for module that have been powered on but have not been connected yet.
- Receiving footswitch press notifications
- Receiving module and channel error notifications
- Receiving reference complete notifications

Asynchronous communication is turned on during the initialization of a module based on a parameter passed in by the client. Asynchronous communication can also be enabled or disabled with the `setAsyncMode` C++ method. There are no C functions for enabling asynchronous communication; the modules must be initialized with the required setting.

The client must bind on a UDP socket and wait on the asynchronous port in order to get asynchronous messages. Once the message is received, it can be decoded and/or acted upon.

The module will continue to send the asynchronous message once every 5 seconds until it is acknowledged.

Refer to “Asynchronous methods” on page 211 for more information.

The commands to decode the asynchronous message are:

- The `getAsyncMsgType` C++ method is used to determine the type of message received
- The `MseModuleGetAsyncMsgType` C function is used to determine the type of message received
- The `decodeConnectMsg` C++ method is used to get the information from a UDP\_CONNECT message.
- The `MseModuleGetAsyncMsgIpAddress`, `MseModuleGetAsyncMsgPort`, `MseModuleGetAsyncMsgDhcp`, `MseModuleGetAsyncMsgMacAddress`, `MseModuleGetAsyncMsgNetmask`, and `MseModuleGetAsyncMsgSerialNumber` C functions are used to get the information from a UDP\_CONNECT message.
- The `decodeLatchMsg` C++ method is used to get the information from a UDP\_LATCH message.
- The `MseModuleGetAsyncMsgLatch` C function is used to get the information from a UDP\_LATCH message.
- The `decodeChannelStatusMsg` C++ method is used to get the information from a UDP\_CHANNEL\_STATUS message.
- The `MseModuleGetAsyncMsgChannelStatus` C function is used to get the information from a UDP\_CHANNEL\_STATUS message.
- The UDP\_INTEGRITY message does not need to be decoded. The actual errors should be read after this is received to determine what error occurred.

The UDP\_CONNECT message will stop being sent when an initialize command is sent to the module.

The UDP\_LATCH message will stop being sent when the module’s latch is cleared as described in “Latching” on page 256.

The UDP\_CHANNEL\_STATUS message with a channel status of 1 (warning or error) will stop being sent when the `EnDat`, `1Vpp`, or `TTL` error is read as described in “Channel Errors and Warnings” on page 259.

The UDP\_CHANNEL\_STATUS message with a channel status of 2 (referencing complete) will stop being sent when the `1Vpp` or `TTL` referencing is acknowledged. The acknowledge is sent with the `acknowledgeAbsolutePosition` C++ method for `1Vpp` and `acknowledgeReferencing` C++ method for `TTL`. The acknowledge is sent with the `Mse1VppModuleAcknowledgeAbsolutePosition` C function for `1Vpp` and `MseTtlModuleAcknowledgeReferencing` C function for `TTL`.

The UDP\_INTEGRITY message will stop being sent when the module errors are read as described in “Module Errors and Warnings” on page 258.

The asynchronous port defaults to 27300. The `MSE1000_ASYNC_PORT` constant can be used access the default asynchronous port value. The asynchronous port can be changed for each module with the `setAsyncPort` C++ method. The asynchronous port can be changed for each module with the `MseModuleSetAsyncPort` C function. Changing the asynchronous port will set the value in the module non-volatile memory. All modules can be restored to factory defaults which will set the asynchronous port back to 27300.



# 9

**C++ examples**

## 9.1 Overview

The C++ example section contains basic explanations of the C++ examples that are available in the Visual Studio project packaged with the MSeLibrary as well as some hand picked examples embedded directly in this manual.

## 9.2 MseLibraryCppExamples Visual Studio Solution

The C++ examples are located in the directory described in section 2.2 Installation Instruction.

Open the MSeLibraryCppExamples.sln solution from within Visual Studio 2010 to access the MSeLibraryCppExamples project. The MSeLibraryCppExamples project's main file is MseExamples.cpp.

The C++ example is a command line program that allows for the following command line calls:

Broadcasting, CreateChain, SetIp, Program, Latching, SetLatch, GetLatches, IO, Pneumatic, EnDat, 1Vpp, Analog, LVDT, TTL, Referencing, ReferenceRotary, ReferenceTTL, TeachGain, Discovery, Subscribe, and ReadConfig. The user can run these commands from the command line or by entering the command line arguments in the 'Debugging' screen of the MSeLibrary-CppExamples properties page from inside Visual Studio. The IP address of the workstation is assumed to be 172.31.46.200 for these examples and must be changed to the correct IP address by the user. The IP addresses used to communicate with the modules in these examples are chosen since they match a module chain used to create the tests, the user must change the IP address to match the modules in their system. The C equivalent functions for the C++ library calls can be determined by looking at the 'Library Software' and 'Operating Principles' sections of this document.

The MseExamples.cpp file contains the main() function. The developer can use this file to access all of the examples. For example, if the developer wants to see how to perform a broadcast:

- ▶ Find the "Broadcasting" text in the argv string compare section of the main() function.
- ▶ Open the MseNetworking.cpp file since that is the object that is constructed in the example.
- ▶ Go to the createChain method in the MseNetworking.cpp file since that method is called next. Notice that the createChain method is called with the desired parameters, the return of the method is checked for errors, and the error code of the return can be decoded to get a textual representation.
- ▶ Go to the testChain method in the MseNetworking.cpp file since that method is called next. The module type, module data, counts, number of channels, and input and output values of I/O modules are all read.

### Broadcasting Example

The Broadcasting example will display all of the IP addresses for the modules in the chain. These addresses will be useful for utilizing the other examples.

- ▶ Type "Broadcasting 172.31.46.200" as the parameters to MseExamples.exe, in order to obtain all of the module information plus some data from the channels for all the modules in the chain.

The IP address values from the above command for use in the following examples are:

- 172.31.46.101 Power Supply
- 172.31.46.102 Base EnDat
- 172.31.46.103 4x TTL
- 172.31.46.104 LVDT
- 172.31.46.105 8x EnDat
- 172.31.46.106 4x 1Vpp
- 172.31.46.107 Power Supply
- 172.31.46.108 Analog
- 172.31.46.109 I/O
- 172.31.46.110 Pneumatic



## EnDat Example

- ▶ Type “EnDat 172.31.46.102”, as the parameters to MseExamples.exe, in order to obtain the following:
  - Module information
  - Encoder information for each channel
  - Count values for each channel
  - Position values for each channel
  - Position values after modifying the UOM, error compensation, scaling, and device offset
  - Encoder error information for each channel
  - Encoder warning information for each channel
  - Module integrity information
  - Latch state
  - Module integrity information after clearing errors

Refer to the “EnDat” section of the main() function to find the methods that contain the example code. The EnDat example instantiates a single MseEndatModule and initializes it. The setEncoderInfo call in the MseEndatTesting.cpp file contains example calls for setting up the error compensation, scaling, device offset, and uom. There is no need to configure the channel presence, encoder type, signal period, or line count since the module auto-detects the type of encoder attached and EnDat encoders store information internally.

## TTL Example

- ▶ Type “TTL 172.31.46.103”, as the parameters to MseExamples.exe, in order to obtain the following:
  - Resolution for each channel
  - Position values for each channel

Refer to the “TTL” section of the main() function to find the methods that contain the example code. The 1Vpp example instantiates a single MseTtlModule and initializes it. The getPositions call in the MseTtlTesting.cpp file contains example calls for setting up the channel presence, referencing, error compensation, scaling, device offset, encoder type, signal period, uom, and counting direction.

## 1 Vpp Example

- ▶ Type “1Vpp 172.31.46.106”, as the parameters to MseExamples.exe, in order to obtain the following:
  - Signal type for a specific channel
  - Resolution for a specific channel
  - Position values for a specific channel

Refer to the “1Vpp” section of the main() function to find the methods that contain the example code. The 1Vpp example instantiates a single Mse1VppModule and initializes it. The showLinearEncoderData call in the Mse1VppTesting.cpp file contains example calls for setting up the error compensation, scaling, device offset, encoder type, signal period, uom, and counting direction. There is no need to configure the channel presence since the module auto-detects whether an encoder is attached, and whether it is a 1 Vpp or 11 uApp, when it is first turned on.

## LVDT Example

- ▶ Type “LVDT 172.31.46.104”, as the parameters to MseExamples.exe, in order to obtain the following:
  - Counts for all channels
  - Voltage representation of the counts for all channels
  - Positions for all channels
  - Excitation voltage and frequency of the module

Refer to the “LVDT” section of the main() function to find the methods that contain the example code. The LVDT example instantiates a single MseLvdModule and initializes it. The getPositions call in the MseLvdTesting.cpp file contains example calls for setting up the channel presence, resolution, sensor gain, scaling, device offset, and oversampling.

## Analog Example

- ▶ Type "Analog 172.31.46.108," as the parameters to MseExamples.exe, in order to obtain the following:
  - Voltage and current for both channels
  - Voltage that is manually scaled for the first channel
  - Voltage that is scaled using the computeResolutionAndOffset method for the first channel

Refer to the "Analog" section of the main() function to find the methods that contain the example code. The Analog example instantiates a single MseAnalogModule and initializes it. The getValues call in the MseAnalogTesting.cpp file contains example calls for setting the number of samples used for internal averaging, scaling, resolution, offset, and automatic computation of the resolution and offset.

## I/O Example

- ▶ Type "IO 172.31.46.109 1," as the parameters to MseExamples.exe, in order to obtain the following:
  - The input and output values for all channels

Refer to the "IO" section of the main() function to find the methods that contain the example code. The I/O example instantiates a single MseloModule and initializes it. The setlo and getlo calls in the MseloTesting.cpp file contains example calls for setting I/O output and reading the current state of the input and outputs.

## Pneumatic Example

- ▶ Type "Pneumatic 172.31.46.110 1," as the parameters to MseExamples.exe, in order to obtain the following:
  - The output value

Refer to the "Pneumatic" section of the main() function to find the methods that contain the example code. The Pneumatic example instantiates a single MsePneumaticModule and initializes it. The setOutput and getOutput calls in the MsePneumaticTesting.cpp file contains example calls for setting the output and reading the current state of the output.

## Latching Example

- ▶ Type "SetLatch 172.31.46.200 1," as the parameters to MseExamples.exe, or utilize a footswitch in order to set the latch on the base module.
- ▶ Type "GetLatches 172.31.46.200 1," as the parameters to MseExamples.exe, in order to read the latch states of all modules, latched data of all modules, and clear the latch.
- ▶ Type "Latching 172.31.46.200," as the parameters to MseExamples.exe, in order to set the latch, read the latch states of all modules, latched data of all modules, and clear the latch.

Refer to the "Latching," "SetLatch," and "GetLatches" sections of the main() function to find the methods that contain the example code. The Latching, SetLatch and GetLatches commands will perform a broadcast in order to determine all of the modules in the chain.

## Setting IP Address Example

- ▶ Type "SetIp 172.31.46.200 0 172.31.46.122 255.255.255.0," as the parameters to MseExamples.exe, in order to set the IP address of the first module in the chain, the power supply, to a new IP address and netmask.

Refer to the "SetIp" section of the main() function to find the methods that contain the example code. The SetIp command will perform a broadcast in order to determine all of the modules in the chain, show the current IP settings of the desired module, and set the IP of the desired module. The showModuleInfo and setIp calls in the MseNetworking.cpp file contains example calls for displaying and setting the IP.

## Referencing Example

- ▶ Type "Referencing 172.31.46.106 0," as the parameters to MseExamples.exe, in order to enable single reference mark referencing for a 1Vpp encoder with a signal period of 20.

Refer to the "Referencing" section of the main() function to find the methods that contain the example code. The requestReferencing method in the Mse1VppTesting.cpp file will start the single referencing with a signal period of 20. The waitForReferencing method in the Mse1VppTesting.cpp file contains example calls for waiting until the referencing has been completed.

## Referencing a 1Vpp Coded Rotary Encoder Example

- ▶ Type "Reference1VppCodedRotary 172.31.46.106 0", as the parameters to MseExamples.exe, in order to enable and perform coded reference mark referencing for a ROD 780C 1Vpp encoder with a line count of 36000. See the table of encoders in the "Configuring the Channels" section of this manual for signal period, line count, and distance codings for HEIDENHAIN encoders.

Refer to the "Reference1VppCodedRotary" section of the main() function to find the methods that contain the example code. The requestRotaryReferencing method in the Mse1VppTesting.cpp file will start the angular coded referencing with a line count of 36000. The waitForReferencing method in the Mse1VppTesting.cpp file contains example calls for waiting until the referencing has been completed. The state of the referencing will change and be shown as each reference mark is crossed.

## Referencing a TTL Coded Encoder Example

- ▶ Type "ReferenceTTLcoded 172.31.46.103 0", as the parameters to MseExamples.exe, in order to enable and perform coded reference mark referencing for an LS 378C TTL encoder with a signal period of 20 microns and interpolation of 20X. The type of referencing is REFERENCE\_MARK\_CODED\_1000. See the table of encoders in the "Configuring the Channels" section of this manual for signal period, line count, interpolation, and distance codings for HEIDENHAIN encoders.

Refer to the "ReferenceTTLcoded" section of the main() function to find the methods that contain the example code. The requestReferencing method in the MseTtlTesting.cpp file will start the coded referencing with a signal period of 20 microns. The waitForReferencing method in the MseTtlTesting.cpp file contains example calls for waiting until the referencing has been completed. The state of the referencing will change and be shown as each reference mark is crossed.

## Module Discovery Example

- ▶ Type "Discovery 172.31.46.200", as the parameters to MseExamples.exe, in order to create a thread and wait for asynchronous data from the modules. Power cycle the modules after this command is issued. The modules will each broadcast their connection information to the MSE1000\_ASYNC\_PORT every 10 seconds. The modules will stop broadcasting once connected via module initialization or the setAsyncMode method is called with 0 as the useAsync parameter.

Refer to the "Discovery" section of the main() function to find the methods that contain the example code. The startThread method in the AsyncMessageHandler.cpp file will start the thread, wait on the socket, and decode the message.

## Module Subscribe Example

- ▶ Type "Subscribe 172.31.46.200 172.31.46.102 74749901", as the parameters to MseExamples.exe, in order to create a connection to a module based on the type of module and its IP address. The module will be put into asynchronous mode, and a thread will be created to receive the asynchronous messages. Latching messages, errors and warnings, and referencing complete messages can be obtained asynchronously this way. This is useful if client code does not want to poll for this information. The messages will continue to be sent until acknowledged. The latch is acknowledged by reading the data or clearing the latch. The referencing complete is acknowledged by sending the acknowledgeAbsolutePosition method for 1Vpp or acknowledgeReferencing method for TTL. The getWarnings, getErrors, and getChannelStatus methods will acknowledge the warning and errors.

Refer to the "Subscribe" section of the main() function to find the methods that contain the example code. The startThread method in the AsyncMessageHandler.cpp file will start the thread, wait on the socket, and decode the message.

## Programming MSEfirmware Example

- ▶ Type "Program 172.31.46.110 c:\\Firmware\\MSEfirmware.dat", as the parameters to MseExamples.exe, in order to create a connection to a module based on the IP address and then program the firmware.

Refer to the "Program" section of the main() function to find the methods that contain the example code. The ProgramMse1000 method in the MseProgramTesting.cpp file will initialize the module and program it.

## ModuleConfig Read and Write Example

- ▶ Type "ReadConfig C:\\Users\\Public\\HEIDENHAIN\\MSEsetup\\Config\\ModuleConfig.xml" in order to read the values in the ModuleConfig.xml file and to show how to change the module and channel labels.

Refer to the "ReadConfig" section of the main() function to find the methods that contain the example code. The MseConfigReaderTesting.cpp file contains the examples for reading. The MseConfigWriterTesting.cpp file contains the examples for writing.

## LVDT Teach Gain Example

- ▶ Type "TeachGain 172.31.46.104;" as the parameters to MseExamples.exe, in order to obtain the following:
  - The computed gain value for the first channel

Refer to the "TeachGain" section of the main() function to find the methods that contain the example code. The TeachGain example instantiates a single MseLvdtModule and initializes it. The teachGain method in the MseLvdtTesting.cpp file contains example calls for setting the channel presence, teaching the gain, and checking for and getting the result of the teach.

## 9.3 Initializing the module chain

The initialization of the MSE is performed dynamically via broadcasting or manually by passing the required parameters.

Initialization is achieved by:

- Instantiating the `MseInterface` class.
- Calling the `createChain()` method for broadcasting or `addModule()` method for each module in the chain.

Initialization will create the UDP connection needed for communication to the module as well as gather all the module specific information and device information for `EnDat` modules.

The information returned from the broadcasting can be saved and used by client business logic to store all the needed information so that future broadcasts are not needed.

There is no need to deallocate memory for objects created by the library. The destructors of the classes will delete the dynamically created objects so that they are deallocated when the `MseInterface` goes out of scope.

### Creating a Chain via Broadcasting

The MSE chain is created via broadcasting with the `createChain()` method.

The `createChain()` method will do the following:

- Remove all connections that are currently in the chain.
- Send a broadcast message to the MSE requesting the IP address and port.
  - Each module in the MSE chain will respond to a broadcast message by sending a response with its IP address and port.
  - The information needed to communicate with each module is available once the responses are received but the order of the chain is not currently known.
- Read up to 64 responses from the MSE and stop waiting for responses after a timeout.
- Initialize each device that was found from the broadcast and add them to the MSE chain.
  - Initialization consists of setting up the UDP communication, requesting the module information from the module, and requesting the device information for each channel if the module is an `EnDat` module.
  - The asynchronous communication is not needed unless a separate thread is going to be used to watch for foot switch latches, 1Vpp reference complete, or errors.
- Organize the chain.
  - Utilizes the modules input and output connections, via the `setRight()` and `getLeft()` functions, to determine the order of the chain.
  - Reorganizes the MSE chain to represent the real physical locations of the modules.

#### Example

To initialize and create a chain of modules via broadcasting:

- ▶ Include the `MseInterface.h` header and instantiate the `MseInterface` class.

```
#include "MseInterface.h"
#include <iostream> // for sending results to the console
MseInterface mse;
```

- ▶ Create the MSE chain by calling `createChain()` with the IP address, base port to use for the client PC (this is what the MSE devices will use for responses), asynchronous messages set to false, and the netmask for use by the broadcasting.

```
MseResults retVal = mse.createChain("172.31.46.3",27016,false, "255.255.255.0");
if (RESPONSE_OK != retVal.getCode())
std::cout << "handle error" << std::endl;
```

The MSE chain is created manually with the `addModule()` method. This is the preferred method once all of the module IP addresses are known because it allows for the chain to be created faster, in the correct order from the start, and also allows for asynchronous communication from the modules.

The `addModule()` function will do the following:

- Return an error code of `RESPONSE_MODULE_MISMATCH` if the module requested does not match the module type in the firmware.
- Create a new module of the type requested.
- Initialize the module.
- Add the module to the MSE chain.

### Example

To create a chain of modules manually:

- ▶ Include the `MseInterface.h` header and instantiate the `MseInterface` class.

```
#include "MseInterface.h"
#include <iostream> // for sending results to the console
#include <sstream> // for streaming object data
MseInterface mse;
```

- ▶ Create the MSE chain by calling `addModule()` for each module in the chain.

```
MseResults retVal = mse.addModule(MODULE_ID_ENDAT_BASE, "172.31.46.4", false);
if (RESPONSE_OK != retVal.getCode())
{
    std::stringstream ss;
    ss << "Error: " << MseResults::showRespCode(retVal.getCode());
    cout << ss;
    std::cout << "handle error" << std::endl;
}
retVal = mse.addModule(MODULE_ID_IO_IP40, "172.31.46.5", false);
retVal = mse.addModule(MODULE_ID_1VPP_4X, "172.31.46.6", false);
retVal = mse.addModule(MODULE_ID_ENDAT_4X, "172.31.46.7", false);
retVal = mse.addModule(MODULE_ID_ENDAT_8X, "172.31.46.8", false);
```

## 9.4 Getting counts

This example shows how to get the counts of the measurement devices. The `getCounts` will return the requested number of channels worth of counts (unless the `numChannels` is > the number of channels available in that module).

### Example

To initialize and read count data:

- ▶ Include the `MseInterface.h` header and instantiate the `MseInterface` class.

```
#include "MseInterface.h"
#include <iostream> // for sending results to the console
MseInterface mse;
```

- ▶ Create the MSE chain by calling `createChain()` with the IP address, base port to use for the client PC (this is what the MSE devices will use for responses), asynchronous messages set to false, and the netmask for use by the broadcasting.

```
MseResults retVal = mse.createChain("172.31.46.3",27016,false, "255.255.255.0");
if(RESPONSE_OK != retVal.getCode())
std::cout << "handle error" << std::endl;
```

- ▶ Get a reference to the desired module, request the counts, iterate through the channels and display the count value.

```
MseResults retVal;
// Allocate enough memory for the maximum number of channels
unsigned long counts[MAX_CHANNELS_PER_MODULE];

// Get the reference to the first module (indexed from 0)
MseModule* module = mse.getModule(0);
if(0 == module)
std::cout << "handle error" << std::endl;

// Request counts
retVal = module->getCounts(counts, module->getNumChannels(), COUNT_REQUEST_LATEST);
if(RESPONSE_OK != retVal.getCode())
std::cout << "handle error" << std::endl;

// Display the resulting counts
for(unsigned int i = 0; i < module->getNumChannels(); ++i)
{
    cout << "Channel[" << i << "] = " << counts[i] << "\n";
}
```

## 9.5 Setting the Encoder Information

The encoder information is used for EnDat and 1Vpp modules. The encoder information is set after the module chain is created. The EnDat devices will default to millimeters for linear encoders and degrees for rotary encoders.

The 1Vpp encoders have additional information that needs to be set in order to get a position. 1Vpp rotary encoders require the line count to be set and 1Vpp linear encoders require the signal period.

### Example

To set the encoder info:

- ▶ Include the headers and instantiate the MseInterface class.

```
#include "MseInterface.h"
MseInterface mse;
```

- ▶ Create the MSE chain by calling createChain() with the IP address and base port to use for the client PC (this is what the MSE devices will use for responses).

```
MseResults retVal = mse.createChain("172.31.46.3",27016,false, "255.255.255.0");
if(RESPONSE_OK != retVal.getCode())
std::cout << "handle error" << std::endl;
```

- ▶ Get a pointer to the 1Vpp module that needs to be configured and set the encoder information.

```
Mse1VppModule* module = mse.get1VppModule(0);
if(0 == module)
std::cout << "handle error" << std::endl;
```

```
module->setUom(UOM_INCHES,0);
module->setErrorCompensation(1.0,0);
module->setEncoderType(ENCODER_TYPE_GAUGE,0);
module->setSignalPeriod(20,0);
module->setCountingDirection(true,0);
```

```
module->setUom(UOM_DEGREES,1);
module->setErrorCompensation(1.0,1);
module->setEncoderType(ENCODER_TYPE_ROTARY,1);
module->setLineCount(4096,1);
module->setCountingDirection(true,1);
```

```
module->setUom(UOM_MM,2);
module->setErrorCompensation(1.0,2);
module->setEncoderType(ENCODER_TYPE_LINEAR,2);
module->setSignalPeriod(20,2);
module->setCountingDirection(false,2);
```

```
module->setUom(UOM_DEGREES,3);
module->setErrorCompensation(1.0,3);
module->setEncoderType(ENCODER_TYPE_ROTARY,3);
module->setLineCount(18000,3);
module->setCountingDirection(true,3);
```



- ▶ Read the counts. The counts will be raw values.

```
// Allocate enough memory for the maximum number of channels
unsigned long counts[MAX_CHANNELS_PER_MODULE];

retVal = module->getCounts(counts, module->getNumChannels(), COUNT_REQUEST_LATEST);
if(RESPONSE_OK != retVal.getCode())
std::cout << "handle error" << std::endl;

// Display the resulting counts
for(unsigned int i = 0; i < module->getNumChannels(); ++i)
{
    cout << "Channel[" << i << "] = " << counts[i] << "\n";
}

```

- ▶ Read the position. The position will be based on the uom, line count, signal period, counting direction, and error compensation passed in.

```
double pos[MAX_CHANNELS_PER_MODULE];
retVal = module->getPositions(pos, module->getNumChannels(), COUNT_REQUEST_LATEST);
if(RESPONSE_OK != retVal.getCode())
std::cout << "handle error" << std::endl;

for(unsigned int i = 0; i < module->getNumChannels(); ++i)
{
    cout << "Channel[" << i << "] = " << pos[i] << "\n";
}

```

## Latching 9.6 Latching

Latching is used to capture position and I/O data for all modules for a specific moment in time. Latching occurs by telling the base module to start the latch via the MSELlibrary or from a footswitch attached to the serial port of the base module. Latching is done after the module chain is created.

The MSELlibrary utilizes the `setLatch(LATCH_COUNT_RESET, LATCH_CHOICE_ALL)` method for clearing a latch and the `setLatch(LATCH_COUNT_SET, LATCH_CHOICE_SOFTWARE_1)` method for causing the latch to occur. The `getPosition()` method with a option parameter of `COUNT_REQUEST_LATCHED` is used to get the latched data from a module. The stored latched position in a module will not be updated again unless the latch is cleared.

### Example

To set latching:

- ▶ Include the headers and instantiate the `MseInterface` class.

```
#include "MseInterface.h"
MseInterface mse;
```

- ▶ Create the MSE chain by calling `createChain()` with the IP address and base port to use for the client PC. The MSE devices will use this for responses.

```
MseResults retVal = mse.createChain("172.31.46.3", 27016, false, "255.255.255.0");
if (RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

- ▶ Loop through the chain and clear the latches on all modules.

```
for (unsigned int moduleIndex = 0; moduleIndex < mse.getNumModules(); ++moduleIndex)
{
    MseModule* module = mse.getModule(moduleIndex);
    if (module)
    {
        retVal = module->setLatch(LATCH_COUNT_RESET, LATCH_CHOICE_ALL);
        if (RESPONSE_OK != retVal.getCode())
        {
            std::cout << "Could not clear the latch in the module";
            return false;
        }
    }
}
```

- ▶ Set the latch on the base module.

```
MseModule* baseModule = mse.getModule(1);
if (baseModule)
{
    retVal = baseModule->setLatch(LATCH_COUNT_SET, LATCH_CHOICE_SOFTWARE_1);
    if (RESPONSE_OK != retVal.getCode())
    {
        std::cout << "Could not set the latch in the base module";
        return false;
    }
}
```

- ▶ Loop through the chain and get the latched data from each 1Vpp and EnDat module

```
MODULE_ID moduleType;
unsigned long hwId = 0;
unsigned char hwRev = 0;
unsigned short numAxes;
double pos[MAX_CHANNELS_PER_MODULE];
long currentRevolution[MAX_CHANNELS_PER_MODULE];

for (unsigned int moduleIndex = 0; moduleIndex < mse.getNumModules(); ++moduleIndex)
{
    if (0 == mse.getModule(moduleIndex))
    {
        std::cout << "Module returned NULL" << std::endl;
        continue;
    }
}
```

```

retVal = mse.getModule(moduleIndex)->getModuleType(&moduleType, &hwId, &hwRev,
&numAxes);
if (RESPONSE_OK != retVal.getCode())
{
    std::cout << "Could not get the module type " << std::endl;
    continue;
}
switch(moduleType)
{
    case MODULE_ID_ENDAT_BASE:
    case MODULE_ID_ENDAT_8X:
    case MODULE_ID_ENDAT_4X:
    {
        MseEndatModule* module = mse.getEndatModule(moduleIndex);
        if(0 == module)
        {
            std::cout << "Module " << moduleIndex << " is NULL " << std::endl;
            continue;
        }

        retVal = module->getPositions(pos,currentRevolution,module-
>getNumChannels(),
COUNT_REQUEST_LATCHED);
        if (RESPONSE_OK != retVal.getCode())
        {
            std::cout << "Could not get positions from EnDat module" << std::endl;
            continue;
        }

        for(unsigned int i = 0; i < module->getNumChannels(); ++i)
        {
            std::cout << "Channel[" << i << "] = " << std::setiosflags
(std::ios::fixed)
<< std::setprecision(4) << pos[i] << std::endl;
        }
    }
    break;

    case MODULE_ID_1VPP_BASE:
    case MODULE_ID_1VPP_8X:
    case MODULE_ID_1VPP_4X:
    {
        Mse1VppModule* module = mse.get1VppModule(moduleIndex);
        if(0 == module)
        {
            std::cout << "Module " << moduleIndex << " is NULL " << std::endl;
            continue;
        }

        retVal = module->getPositions(pos,module->getNumChannels(),
COUNT_REQUEST_LATCHED);
        if (RESPONSE_OK != retVal.getCode())
        {
            std::cout << "Could not get positions from 1Vpp module << std::endl;
            continue;
        }

        for(unsigned int i = 0; i < module->getNumChannels(); ++i)
        {
            std::cout << "Channel[" << i << "] = " << std::setiosflags
(std::ios::fixed)
<< std::setprecision(4) << pos[i] << std::endl;
        }
    }
    break;

    default:
    break;
}
}
}

```

## 9.7 Referencing 1Vpp Linear Encoder

The referencing procedure is used for 1Vpp linear and rotary encoders. This example is for a HEIDENHAIN LS 388C linear encoder with a signal period of 20 micrometers and 1000 signal period spacing. Referencing the encoder is the process used to obtain an absolute position on the encoder's scale. The user must initialize the module, set the encoder data, start the referencing, move the encoder across the reference marks, wait for the referencing to complete, and then verify that the referencing was successful. Polling the referencing complete can be replaced with monitoring the asynchronous thread for a UDP\_CHANNEL\_STATUS message followed by decoding the asynchronous message with the `decodeChannelStatusMsg` method and then sending an `acknowledgeAbsolutePosition` method to the module.

### Example

To reference the encoder:

- ▶ Include the headers and instantiate the `MseInterface` class.

```
#include "MseInterface.h"
MseInterface mse;
Create the MSE chain by calling createChain() with the IP address and base port to use
for the client PC (this is what the MSE devices will use for responses).
```

```
MseResults retVal = mse.createChain("172.31.46.3",27016,false, "255.255.255.0");
if(RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

- ▶ Get a pointer to the 1Vpp module and set the encoder data.

```
Mse1VppModule* module = mse.get1VppModule(0);
if(0 == module)
    std::cout << "handle error" << std::endl;
```

```
module->setUom(UOM_MM,0);
module->setErrorCompensation(1.0,0);
module->setEncoderType(ENCODER_TYPE_LINEAR,0);
module->setSignalPeriod(20,0);
module->setCountingDirection(true,0);
```

- ▶ Tell the module to obtain the reference position.

```
module_>initAbsolutePosition(0,REFERENCE_MARK_CODED_1000,20);
```

- ▶ Poll the module until the reference position has been obtained (the user must move the encoder read head until it crosses at least 2 reference marks).

```
bool isReferenced = false;
```

```
while(1)
{
    retVal = module_>isReferencingComplete(0,&isReferenced);
    if(RESPONSE_OK != retVal.getCode())
    {
        std::cout << "Could not check isReferenceComplete: " <<
MseResults::showRespCode(retVal.getCode()) << std::endl;
        return false;
    }

    if(isReferenced)
        break;
}
```

- ▶ Verify that the referencing has been successful.

```
REF_MARK_STATE refState = REF_MARK_OFF;

retVal = module_>getReferencingState(0,&refState);
if(RESPONSE_OK != retVal.getCode())
{
    std::cout << "Could not get the referencing state: " <<
MseResults::showRespCode(retVal.getCode()) << std::endl;
    return false;
}

if(refState == REF_MARK_FINISHED)
    std::cout << "The encoder referencing passed" << std::endl;
else
    std::cout << "The encoder referencing failed" << std::endl;
```

## 9.8 Programming Firmware

The firmware is programmed using the `program()` method in the `MseModule` class. The percent complete of the programming can be obtained with the `getProgrammingPercentComplete()` method. The `program` method blocks until done, so the `getProgrammingPercentComplete()` needs to run in another thread by the client. The percent complete is returned as a double. The programming can be done by using a module in the MSE chain or by IP address in case there is an initialization problem with a module in the chain.

### Example

To program firmware:

- ▶ Include the headers and instantiate the `MseInterface` class.

```
#include "MseInterface.h"
MseInterface mse;
```

- ▶ Create the MSE chain by calling `createChain()` with the IP address and base port to use for the client PC (this is what the MSE devices will use for responses).

```
MseResults retVal = mse.createChain("172.31.46.3", 27016, false, "255.255.255.0");
if (RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

- ▶ Get a reference to the desired module to program.

```
MseModule* module = mse.getModule(0);
if (0 == module)
    std::cout << "handle error" << std::endl;
```

- ▶ Program the module with the new firmware.

```
MseResults retVal;
retVal = module->program("C:\\Program Files\\MSEsetup\\Firmware\\MSEfirmware.dat");
if (RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

To program firmware with IP address:

- ▶ Include the headers.

```
#include "MseInterface.h"
```

- ▶ Create a new module for programming (because the chain status is unknown) and remove it from the UDP server when done (in the destructor of the `MseModule`).

```
MseModule* module = new MseModule();
if (0 == module)
    std::cout << "handle error" << std::endl;
```

- ▶ Initialize the module with the `initializeFirmware` function of the `MseModule` class.

```
MseResults retVal;
retVal = module->initializeFirmware("172.31.46.1");
if (RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

- ▶ Program the module with the new firmware.

```
retVal = module->program("C:\\Program Files\\MSEsetup\\Firmware\\MSEfirmware.dat");
if (RESPONSE_OK != retVal.getCode())
    std::cout << "handle error" << std::endl;
```

## 9.9 MseConfigReader

The MseConfigReader is used to read configuration data that was created by the MSEsetup application. This data can be very useful when utilizing the MSElibrary as it allows a developer to leverage this data instead of having to create their own persistent storage.

The MSEsetup application must successfully perform a broadcast before the following example can be utilized.

The examples in this section utilize C++.

### Example

To read from the ModuleConfig.xml file:

- ▶ Include the header and instantiate the MseConfigReader class.

```
#include "MseConfigReader.h"
#include "mseDeviceModule.h"
MseConfigReader configReader;
```

- ▶ Call loadXml with the filename of the ModuleConfig.xml file:

```
MSE_XML_RETURN retVal = MSE_XML_RETURN_OK;
retVal = configReader.loadXml("C:\\ProgramData\\HEIDENHAIN\\MSEsetup\\config\\
ModuleConfig.xml");
if (retVal != MSE_XML_RETURN_OK)
{
    std::cout << "Error: " << MseConfigbase::decodeErrorType(retVal) << std::endl;
    return;
}
```

- ▶ Call getElement with the desired tag name, module number, and channel number (getElement is an overloaded function that can be called with or without the module number or channel number based on the desired tag to retrieve). The following example retrieves the 1Vpp line count for module 2, channel 2.

```
MSE_XML_RETURN retVal = MSE_XML_RETURN_OK;
std::String tempStr;
int moduleNum = 2;
int channelNum = 2;

retVal = configReader.getElement(MSE_XML_ELEMENT_LINE_COUNT, tempStr, moduleNum,
channelNum);
if (retVal != MSE_XML_RETURN_OK)
{
    std::cout << "Error: " << MseConfigbase::decodeErrorType(retVal) << std::endl;
    return;
}
else
{
    std::cout << MseConfigBase::decodeElementType(MSE_XML_ELEMENT_LINE_COUNT)
<< " = " << tempStr << std::endl;
}
```





# 10

▮ **C examples**

## 10.1 Overview

The C example section contains explanations of the C examples that are available in the Visual Studio project packaged with the MSElibrary as well as an example embedded directly in this manual. Refer to the C++ examples and the OperatingPrinciples section for more information on interacting with the modules and the correct methods and functions to utilize.

## 10.2 MSElibraryCEXamples Visual Basic Solution

The C examples are located in the directory described in the Installation Instruction section on page 24.

Open the MSElibraryCEXamples.sln solution from within Visual Studio 2010 to access the MSElibraryCEXamples project. The MSElibraryCEXamples project's main file is main.cpp.

The main function take command line parameters in order to perform the correct examples.

The MSElibraryCEXamples.exe is a command line program that allows for the following:

- MSElibraryCEXamples.exe [MODULE\_IP\_ADDRESS]
- MSElibraryCEXamples.exe Broadcasting [CLIENT\_IP\_ADDRESS]
- MSElibraryCEXamples.exe Async [CLIENT\_IP\_ADDRESS]
- MSElibraryCEXamples.exe ConfigFile [FILE\_LOCATION]
- MSElibraryCEXamples.exe SetIp [MODULE\_IP\_ADDRESS] [NEW\_IP\_ADDRESS] [NEW\_NETMASK]
- MSElibraryCEXamples.exe Program [CLIENT\_IP\_ADDRESS] [TYPE] [FILE]

### Module Examples

The MSElibraryCEXamples.exe [MODULE\_IP\_ADDRESS] command will connect to the module at the MODULE\_IP\_ADDRESS, determine it's type, and then perform all the available examples for that module.

### Broadcasting Example

The MSElibraryCEXamples.exe Broadcasting [CLIENT\_IP\_ADDRESS] will perform a broadcast and obtain all of the module information for connecting to the modules in the chain.

### Async Example

The MSElibraryCEXamples.exe Async [CLIENT\_IP\_ADDRESS] will create a thread and wait on the asynchronous port for asynchronous messages from the modules. The example will show the asynchronous connect messages only. The warning, error, referencing, and latching messages can be monitored in the subscribe example.

### Subscribe Example

The MSElibraryCEXamples.exe Subscribe [CLIENT\_IP\_ADDRESS] [MODULE\_IP\_ADDRESS] will create a thread and wait on the asynchronous port for asynchronous messages from the specified module. The 1 Vpp and TTL modules will send errors if a channel is considered populated and there are no encoders connected.

### Config File Example

The MSElibraryCEXamples.exe ConfigFile [FILE\_LOCATION] will read information from the ModuleConfig.xml file. Writing to the file is also shown and can be enabled by uncommenting the configFileTesting.write() line of code.

### Setting the IP Address Example

The MSElibraryCEXamples.exe SetIp [MODULE\_IP\_ADDRESS] [NEW\_IP\_ADDRESS] [NEW\_NETMASK] will set the IP address of the desired module to the new IP address and netmask passed in.

### Programming the Firmware and Bootloader Example

The MSElibraryCEXamples.exe Program [CLIENT\_IP\_ADDRESS] [TYPE] [FILE] will program the module with the MSEfirmware.dat or MSEbootloader.dat file passed in. Make sure to never power off the module while programming the bootloader or it will become unusable and will have to be sent back to HEIDENHAIN for reprogramming. Failure to program the MSE-firmware can recover and be programmed again since the module will still have a bootloader loaded. The [TYPE] can be BL for bootloader or FW for firmware.

## 10.3 Initialize, Configure, and Get Positions from a 1Vpp Module

The following example will connect to a 1Vpp module, configure the encoder information for the first channel, and get the encoder position. This example assumes a gauge with a signal period of 20um.

### Example

- ▶ Include the headers and instantiate a Mse1VppModule.

```
#include "MseModuleWrapper.h"
#include "Mse1VppModuleWrapper.h"
#include <iostream>
#include <iomanip>
#include <sstream>

using namespace std;

char ipAddress[16];
MSE_RESPONSE_CODE respCode;
char respStr[256];
unsigned long counts[MAX_CHANNELS_PER_MODULE];
double pos[MAX_CHANNELS_PER_MODULE];
double resolution = 0.0;
short channelNum = 0;
ENCODER_TYPES_ENUM encoderType = ENCODER_TYPE_NONE;
UOM uom = UOM_UNDEFINED;
short signalPeriod = 0;

// The IP address of the module to connect with
strcpy(ipAddress, "172.31.46.17");

// Get a pointer to a Mse1VppModule in order to initialize, configure, and get data
MseModulePtr module = Mse1VppModuleCreate();
if(!module )
    return 0;

// Initialize the module
respCode = Mse1VppModuleInitialize(module ,&ipAddress[0],false);
if(respCode != RESPONSE_OK)
{
    MseModuleShowRespCode(respStr, respCode);
    cout << "Mse1VppModuleInitialize failed: " << respStr << endl;
    Mse1VppModuleDelete(module);
    return;
}

encoderType = ENCODER_TYPE_LINEAR;
respCode = Mse1VppModuleSetEncoderType(module, encoderType, channelNum);
if(respCode != RESPONSE_OK)
{
    Mse1VppModuleDelete(module);
    return;
}
```

```

uom = UOM_MM;
respCode = MselVppModuleSetUom(module, uom, channelNum);
if(respCode != RESPONSE_OK)
{
    MselVppModuleDelete(module);
    return;
}

signalPeriod = 20;
respCode = MselVppModuleSetSignalPeriod(module, signalPeriod, channelNum);
if(respCode != RESPONSE_OK)
{
    MselVppModuleDelete(module);
    return;
}

respCode = MselVppModuleGetResolution(module, &resolution, channelNum);
if(respCode != RESPONSE_OK)
{
    MselVppModuleDelete(module);
    return;
}
cout << "The Resolution for channel " << channelNum << " is " << std::setiosflags
(std::ios::fixed) << std::setprecision(8) << resolution << endl;

respCode = MselVppModuleGetCounts(module, &counts[0], MAX_CHANNELS_PER_MODULE,
COUNT_REQUEST_LATEST);
if(respCode != RESPONSE_OK)
{
    MselVppModuleDelete(module);
    return;
}
cout << "The Counts for channel " << channelNum << " are " << counts[0] << endl;

respCode = MselVppModuleGetPositions(module, &pos[0], MAX_CHANNELS_PER_MODULE,
COUNT_REQUEST_LATEST);
if(respCode != RESPONSE_OK)
{
    MselVppModuleDelete(module);
    return;
}
cout << "The Position for channel " << channelNum << " is " << pos[0] << endl;

// Delete the module when not using it anymore and free up the memory
MselVppModuleDelete(module);

```





# 11

**Visual Basic examples**

## 11.1 Overview

The Visual Basic examples are located in the directory described in section 2.2 Installation Instruction.

Open the MSETestbed.sln solution from within Visual Studio 2010 to access the MSETestbed project. The MSETestbed project's main file is MainForm.vb.

The MSElibrary.dll, QtCore4.dll, and QtXml4.dll should be deleted and re-added to the project if they cannot be found or are incorrect versions.

The following files contain most of the function declarations that are needed for calling into the MSElibrary dll: AnalogModule.vb, CommonModule.vb, ConfigFileModule.vb, EndatModule.vb, IoModule.vb, LvdtModule.vb, OneVppModule.vb, PneumaticModule.vb, and TtlModule.vb. Function declarations that are not provided can be added by the developer based on the already included declarations and the information provided in this manual.

The Visual Basic example relies on an already existing ModuleConfig.xml file to have been created by the MSEsetup application. The MSEsetup application is supplied by HEIDENHAIN as a graphical application to configure the module, store the configuration into a ModuleConfig.xml file, and perform data capturing of the modules. The example will create a module chain once the 'Init' button is pressed and the ModuleConfig.xml file is chosen. The module chain is visible in the Module drop down box. The desired test can then be selected from the 'Tests' dropdown box followed by clicking on 'Open Test'.

## 11.2 Module Throughput Test

The Module Throughput test is used to determine how many channels per second can be captured, the number of packets sent, and the number of dropped packets for an individual module. The results are shown dynamically and can be stored to a file.

The ModuleThroughputForm.vb file has the code used for the test.

The UDP delay, timeout, and number of retries can all be set to limit the number of dropped packets if necessary.

The diagnostic level can be changed in order to show how it affects the EnDat modules throughput.

The 'Auto Control Valve' checkbox will appear if there are any Pneumatic modules in the chain. Clicking on this check box will cause the output of the Pneumatic module(s) to be enabled.

## 11.3 Chain Throughput Test

The Chain Throughput test is used to determine how many channels per second can be captured, the number of packets sent, and the number of dropped packets for an entire chain. The results are shown dynamically and can be stored to a file.

The ChainThroughputForm.vb file has the code used for the test.

The UDP delay and number of retries can be set to limit the number of dropped packets if necessary.

The 'Auto Control Valve' checkbox will appear if there are any Pneumatic modules in the chain. Clicking on this check box will cause the output of the Pneumatic module(s) to be enabled.

## 11.4 Latching Test

The Latching test can be used to show the latch state of all modules.

The LatchingForm.vb file has the code used for the test.

All 5 of the latches can be set through software. The physical footswitch can be tested by clicking on the 'Get latch(es)' button.

The 'Clear Latches' button can be used to clear the latches without reading the positions.

The 'Latched Values' button can be used to read all of the latched values into the screen.

The 'Live Values' button can be used to read all of the latest values into the screen. This is useful for testing latching to make sure that the latched value differs from the live value when a footswitch occurs and the value is read even after the position has changed.

The 'Latch and Read' button will cause a latch to occur and then time how long it takes to read all of the latched values from the modules.

The 'Auto Control Valve' checkbox will appear if there are any Pneumatic modules in the chain. Clicking on this check box will cause the output of the Pneumatic module(s) to be enabled.

The 'External Capture' button is used to capture the latching of the data from the footswitch input. The base module is polled from a thread as fast as possible checking whether the latch has been set. If it has, then the latched data from each module is read and saved into a list. The list is saved to a file called Latching.csv when external capturing is stopped. The file can also be saved to another location and opened using the Save and Open buttons.



## 11.5 Voltage Diagnostics

The voltage diagnostics test is used to read the sensor gain code, read the sensor voltage, set the sensor gain code, get the excitation voltage, get the excitation frequency, set the excitation voltage, and set the excitation frequency.

The LvdDiagnosticForm.vb file has the code used for the test.

The drop-down list box allows you to isolate the desired voltage reading to the selected choice. This utilizes the MseLvd-ModuleSetDiagnosticsEnabled C function. The selection of 'All' in the drop down will not show any reading when 'Get Voltage' is clicked but will instead set the module back to its normal polling routine of reading all populated sensors plus the excitation voltage.

The 'Start' button will allow the voltage value to be read continuously.



# 12

**Delphi examples**

## 12.1 Overview

The Delphi examples are located in the users directory listed in “Installation instructions” on page 24.

- ▶ Open the Mse.dproj project from within Delphi XE3. The MSE project’s main file is Main.pas.

The MSElibrary.dll, QtCore4.dll, and QtXml4.dll should be available to the project if they cannot be found or are incorrect versions.

The following files contain some examples of the function declarations that are needed for calling into the MSElibrary dll:

- Mse1VppModule.pas
- MseEndatModule.pas
- MseModule.pas
- MseConfigFile.pas

Function declarations that are not provided can be added by the developer based on the already included declarations and the information provided in these operating instructions.

The Delphi examples are in the files:

- OneVppExamples.pas
- EndatExamples.pas
- ConfigFileExamples.pas

# 13

LabVIEW

## 13.1 Introduction

The MSElibrary software package contains pre-made LabVIEW VI wrappers to access all of the functionality within the MSElibrary. These wrappers were created with LabVIEW 2012 v120f3 (32-bit) and MSElibrary 2.0.0. An experienced Labview developer can choose to edit or create the VI's. To do this refer to the Labview documentation on integrating third party software into Labview. The included VI's are then used in a sample application to demonstrate using MSElibrary within Labview.

## 13.2 Installation

The MSElibrary installer installs both pre-configured wrapper VI's and an example application, both are based on MSElibrary v2.0.0. The wrapper VI's are located in the HEIDENHAIN\MSElibrary\LabVIEW\Wrappers folder, under which there will be individual folders grouped by library. Each individual library folder will contain the .lvlib file and a VI folder. The example application is located in the HEIDENHAIN\MSElibrary\Examples\LabVIEW folder. This folder contains the LabVIEW project file (.lvproj) and a sub-folder containing the necessary VI's. Refer to "Installation instructions" on page 24 for the location of the \HEIDENHAIN folder on different operating systems.

## 13.3 LabVIEW VI's and corresponding MSELibrary functions

### MSELibrary1VppWrapperVI.lvlib

LabVIEW VI's	MSELibrary Function
Mse1Vpp Module Acknowledge Absolute Position.vi	Mse1VppModuleAcknowledgeAbsolutePosition
Mse1Vpp Module Clear Errors.vi	Mse1VppModuleClearErrors
Mse1Vpp Module Create.vi	Mse1VppModuleCreate
Mse1Vpp Module Delete.vi	Mse1VppModuleDelete
Mse1Vpp Module Enable Analog Diag.vi	Mse1VppModuleEnableAnalogDiag
Mse1Vpp Module Enable Diags.vi	Mse1VppModuleEnableDiags
Mse1Vpp Module Enable Error Checking.vi	Mse1VppModuleEnableErrorChecking
Mse1Vpp Module Get Adc Values.vi	Mse1VppModuleGetAdcValues
Mse1Vpp Module Get Analog Diag.vi	Mse1VppModuleGetAnalogDiag
Mse1Vpp Module Get Channel Error State.vi	Mse1VppModuleGetChannelErrorState
Mse1Vpp Module Get Channel Status.vi	Mse1VppModuleGetChannelStatus
Mse1Vpp Module Get Counting Direction.vi	Mse1VppModuleGetCountingDirection
Mse1Vpp Module Get Counts.vi	Mse1VppModuleGetCounts
Mse1Vpp Module Get Encoder Type.vi	Mse1VppModuleGetEncoderType
Mse1Vpp Module Get Error Compensation.vi	Mse1VppModuleGetErrorCompensation
Mse1Vpp Module Get Latches.vi	Mse1VppModuleGetLatches
Mse1Vpp Module Get Line Count.vi	Mse1VppModuleGetLineCount
Mse1Vpp Module Get Module Error State.vi	Mse1VppModuleGetModuleErrorState
Mse1Vpp Module Get Module Errors.vi	Mse1VppModuleGetModuleErrors
Mse1Vpp Module Get Num Channels.vi	Mse1VppModuleGetNumChannels
Mse1Vpp Module Get Positions.vi	Mse1VppModuleGetPositions
Mse1Vpp Module Get Referencing Complete.vi	Mse1VppModuleGetReferencingComplete
Mse1Vpp Module Get Referencing State.vi	Mse1VppModuleGetReferencingState
Mse1Vpp Module Get Resolution.vi	Mse1VppModuleGetResolution
Mse1Vpp Module Get Signal Period.vi	Mse1VppModuleGetSignalPeriod
Mse1Vpp Module Get Uom.vi	Mse1VppModuleGetUom
Mse1Vpp Module Initialize.vi	Mse1VppModuleInitialize
Mse1Vpp Module Set Counting Direction.vi	Mse1VppModuleSetCountingDirection
Mse1Vpp Module Set Encoder Type.vi	Mse1VppModuleSetEncoderType
Mse1Vpp Module Set Error Compensation.vi	Mse1VppModuleSetErrorCompensation
Mse1Vpp Module Set Latch Debouncing.vi	Mse1VppModuleSetLatchDebouncing
Mse1Vpp Module Set Latch.vi	Mse1VppModuleSetLatch
Mse1Vpp Module Set Line Count.vi	Mse1VppModuleSetLineCount
Mse1Vpp Module Set Signal Period.vi	Mse1VppModuleSetSignalPeriod
Mse1Vpp Module Set Uom.vi	Mse1VppModuleSetUom
Mse1Vpp Module Start Referencing.vi	Mse1VppModuleStartReferencing

## MSElibraryAnalogWrapperVI.lvlib

LabVIEW VI's	MSElibrary Function
Mse Analog Module Clear Errors.vi	MseAnalogModuleClearErrors
Mse Analog Module Clear Latch.vi	MseAnalogModuleClearLatch
Mse Analog Module Compute Resolution And Offset.vi	MseAnalogModuleComputeResolutionAndOffset
Mse Analog Module Create.vi	MseAnalogModuleCreate
Mse Analog Module Delete.vi	MseAnalogModuleDelete
Mse Analog Module Get Adc Values.vi	MseAnalogModuleGetAdcValues
Mse Analog Module Get Current.vi	MseAnalogModuleGetCurrent
Mse Analog Module Get Diag Voltages.vi	MseAnalogModuleGetDiagVoltages
Mse Analog Module Get Latch.vi	MseAnalogModuleGetLatch
Mse Analog Module Get Module Error State.vi	MseAnalogModuleGetModuleErrorState
Mse Analog Module Get Module Errors.vi	MseAnalogModuleGetModuleErrors
Mse Analog Module Get Num Channels.vi	MseAnalogModuleGetNumChannels
Mse Analog Module Get Offset.vi	MseAnalogModuleGetOffset
Mse Analog Module Get Resolution.vi	MseAnalogModuleGetResolution
Mse Analog Module Get Scaled Values.vi	MseAnalogModuleGetScaledValues
Mse Analog Module Get Values.vi	MseAnalogModuleGetValues
Mse Analog Module Get Voltage.vi	MseAnalogModuleGetVoltage
Mse Analog Module Initialize.vi	MseAnalogModuleInitialize
Mse Analog Module Set Num Samples.vi	MseAnalogModuleSetNumSamples
Mse Analog Module Set Offset.vi	MseAnalogModuleSetOffset
Mse Analog Module Set Resolution.vi	MseAnalogModuleSetResolution

## MSElibraryConfigFileWrapperVIs.lvlib

LabVIEW VI's	MSElibrary Function
Mse Config File Create.vi	MseConfigFileCreate
Mse Config File Decode Element Type.vi	MseConfigFileDecodeElementType
Mse Config File Decode Error Type.vi	MseConfigFileDecodeErrorType
Mse Config File Delete.vi	MseConfigFileDelete
Mse Config File Get All Elements.vi	MseConfigFileGetAllElements
Mse Config File Get Channel Element.vi	MseConfigFileGetChannelElement
Mse Config File Get Element.vi	MseConfigFileGetElement
Mse Config File Get Filename.vi	MseConfigFileGetFilename
Mse Config File Get Module Element.vi	MseConfigFileGetModuleElement
Mse Config File Get Num Channels.vi	MseConfigFileGetNumChannels
Mse Config File Get Num Modules.vi	MseConfigFileGetNumModules
Mse Config File Get Specific Channel.vi	MseConfigFileGetSpecificChannel
Mse Config File Get Specific Module.vi	MseConfigFileGetSpecificModule
Mse Config File Load Xml.vi	MseConfigFileLoadXml
Mse Config File Reload Xml.vi	MseConfigFileReloadXml
Mse Config File Remove Module.vi	MseConfigFileRemoveModule
Mse Config File Set Channel Element.vi	MseConfigFileSetChannelElement
Mse Config File Set Module Element.vi	MseConfigFileSetModuleElement
Mse Config File Validate Elements.vi	MseConfigFileValidateElements
Mse Config File Write File.vi	MseConfigFileWriteFile
Mse Config File Write New File.vi	MseConfigFileWriteNewFile



**MSElibraryEndatWrapperVIs.lvlib**

LabVIEW VI's	MSElibrary Function
Mse Endat Module Clear Errors.vi	MseEndatModuleClearErrors
Mse Endat Module Create.vi	MseEndatModuleCreate
Mse Endat Module Delete.vi	MseEndatModuleDelete
Mse Endat Module Enable Diags.vi	MseEndatModuleEnableDiags
Mse Endat Module Enable Error Checking.vi	MseEndatModuleEnableErrorChecking
Mse Endat Module Get Adc Values.vi	MseEndatModuleGetAdcValues
Mse Endat Module Get Channel Error State.vi	MseEndatModuleGetChannelErrorState
Mse Endat Module Get Channel Presence.vi	MseEndatModuleGetChannelPresence
Mse Endat Module Get Channel Status.vi	MseEndatModuleGetChannelStatus
Mse Endat Module Get Channel Warning State.vi	MseEndatModuleGetChannelWarningState
Mse Endat Module Get Counting Direction.vi	MseEndatModuleGetCountingDirection
Mse Endat Module Get Counts.vi	MseEndatModuleGetCounts
Mse Endat Module Get Diags.vi	MseEndatModuleGetDiags
Mse Endat Module Get Distinguishable Revolutions.vi	MseEndatModuleGetDistinguishableRevolutions
Mse Endat Module Get Encoder Id.vi	MseEndatModuleGetEncoderId
Mse Endat Module Get Encoder Name.vi	MseEndatModuleGetEncoderName
Mse Endat Module Get Encoder Serial Number.vi	MseEndatModuleGetEncoderSerialNumber
Mse Endat Module Get Encoder Type.vi	MseEndatModuleGetEncoderType
Mse Endat Module Get Endat Errors.vi	MseEndatModuleGetEndatErrors
Mse Endat Module Get Endat Warnings.vi	MseEndatModuleGetEndatWarnings
Mse Endat Module Get Error Compensation.vi	MseEndatModuleGetErrorCompensation
Mse Endat Module Get Latches.vi	MseEndatModuleGetLatches
Mse Endat Module Get Module Error State.vi	MseEndatModuleGetModuleErrorState
Mse Endat Module Get Module Errors.vi	MseEndatModuleGetModuleErrors
Mse Endat Module Get Num Channels.vi	MseEndatModuleGetNumChannels
Mse Endat Module Get Positions.vi	MseEndatModuleGetPositions
Mse Endat Module Get Resolution.vi	MseEndatModuleGetResolution
Mse Endat Module Get Uom.vi	MseEndatModuleGetUom
Mse Endat Module Initialize.vi	MseEndatModuleInitialize
Mse Endat Module Set Error Compensation.vi	MseEndatModuleSetErrorCompensation
Mse Endat Module Set Latch Debouncing.vi	MseEndatModuleSetLatchDebouncing
Mse Endat Module Set Latch.vi	MseEndatModuleSetLatch

**MSElibraryPneumaticWrapperVIs.lvlib**

LabVIEW VI's	MSElibrary Function
Mse Pneumatic Module Clear Errors.vi	MsePneumaticModuleClearErrors
Mse Pneumatic Module Clear Latch.vi	MsePneumaticModuleClearLatch
Mse Pneumatic Module Create.vi	MsePneumaticModuleCreate
Mse Pneumatic Module Delete.vi	MsePneumaticModuleDelete
Mse Pneumatic Module Get Adc Values.vi	MsePneumaticModuleGetAdcValues
Mse Pneumatic Module Get Latch.vi	MsePneumaticModuleGetLatch
Mse Pneumatic Module Get Module Error State.vi	MsePneumaticModuleGetModuleErrorState
Mse Pneumatic Module Get Module Errors.vi	MsePneumaticModuleGetModuleErrors
Mse Pneumatic Module Get Num Channels.vi	MsePneumaticModuleGetNumChannels
Mse Pneumatic Module Get Output.vi	MsePneumaticModuleGetOutput
Mse Pneumatic Module Initialize.vi	MsePneumaticModuleInitialize
Mse Pneumatic Module Set Output.vi	MsePneumaticModuleSetOutput

## MSElibraryWrapperVIs.lvlib

LabVIEW VI's	MSElibrary Function
Mse Io Module Clear Errors.vi	MseIoModuleClearErrors
Mse Io Module Clear Latch.vi	MseIoModuleClearLatch
Mse Io Module Create.vi	MseIoModuleCreate
Mse Io Module Delete.vi	MseIoModuleDelete
Mse Io Module Get Adc Values.vi	MseIoModuleGetAdcValues
Mse Io Module Get Inputs.vi	MseIoModuleGetInputs
Mse Io Module Get IO.vi	MseIoModuleGetIO
Mse Io Module Get Latch.vi	MseIoModuleGetLatch
Mse Io Module Get Module Error State.vi	MseIoModuleGetModuleErrorState
Mse Io Module Get Module Errors.vi	MseIoModuleGetModuleErrors
Mse Io Module Get Num Channels.vi	MseIoModuleGetNumChannels
Mse Io Module Get Outputs.vi	MseIoModuleGetOutputs
Mse Io Module Initialize.vi	MseIoModuleInitialize
Mse Io Module Set Output.vi	MseIoModuleSetOutput
Mse Io Module Set Outputs.vi	MseIoModuleSetOutputs

## MSElibraryLvdtWrapperVIs.lvlib

LabVIEW VI's	MSElibrary Function
Mse Lvdt Get Fpga Revision.vi	MseLvdtModuleGetFpgaRevision
Mse Lvdt Module Clear Errors.vi	MseLvdtModuleClearErrors
Mse Lvdt Module Clear Latch.vi	MseLvdtModuleClearLatch
Mse Lvdt Module Create.vi	MseLvdtModuleCreate
Mse Lvdt Module Delete.vi	MseLvdtModuleDelete
Mse Lvdt Module Enable Diags.vi	MseLvdtModuleEnableDiags
Mse Lvdt Module Get Adc Values.vi	MseLvdtModuleGetAdcValues
Mse Lvdt Module Get Channel Presence.vi	MseLvdtModuleGetChannelPresence
Mse Lvdt Module Get Excitation Values.vi	MseLvdtModuleGetExcitationValues
Mse Lvdt Module Get Latch.vi	MseLvdtModuleGetLatch
Mse Lvdt Module Get Module Error State.vi	MseLvdtModuleGetModuleErrorState
Mse Lvdt Module Get Module Errors.vi	MseLvdtModuleGetModuleErrors
Mse Lvdt Module Get Num Channels.vi	MseLvdtModuleGetNumChannels
Mse Lvdt Module Get Positions.vi	MseLvdtModuleGetPositions
Mse Lvdt Module Get Resolution.vi	MseLvdtModuleGetResolution
Mse Lvdt Module Get Teach Sensor Gain Finished.vi	MseLvdtModuleGetTeachSensorGainFinished
Mse Lvdt Module Get Sensor Gain.vi	MseLvdtModuleGetSensorGain
Mse Lvdt Module Get Uom.vi	MseLvdtModuleGetUom
Mse Lvdt Module Get Voltage.vi	MseLvdtModuleGetVoltage
Mse Lvdt Module Initialize.vi	MseLvdtModuleInitialize
Mse Lvdt Module Set Channel Presence.vi	MseLvdtModuleSetChannelPresence
Mse Lvdt Module Set Diagnostics Enabled.vi	MseLvdtModuleSetDiagnosticsEnabled
Mse Lvdt Module Set Excitation Frequency.vi	MseLvdtModuleSetExcitationFrequency
Mse Lvdt Module Set Excitation Voltage.vi	MseLvdtModuleSetExcitationVoltage
Mse Lvdt Module Set Resolution.vi	MseLvdtModuleSetResolution
Mse Lvdt Module Set Sensor Gain.vi	MseLvdtModuleSetSensorGain
Mse Lvdt Module Set Uom.vi	MseLvdtModuleSetUom
Mse Lvdt Module Teach Sensor Gain.vi	MseLvdtModuleTeachSensorGain

## MSElibraryModuleWrapperVIs.lvlib

LabVIEW VI's	MSElibrary Function
Mse Module Broadcast.vi	MseModuleBroadcast
Mse Module Clear Errors.vi	MseModuleClearErrors
Mse Module Create.vi	MseModuleCreate
Mse Module Delete.vi	MseModuleDelete
Mse Module Get Adc Values.vi	MseModuleGetAdcValues
Mse Module Get Async Msg Channel Status.vi	MseModuleGetAsyncMsgChannelStatus
Mse Module Get Async Msg Dhcp.vi	MseModuleGetAsyncMsgDhcp
Mse Module Get Async Msg Ip Address.vi	MseModuleGetAsyncMsgIpAddress
Mse Module Get Async Msg Latch.vi	MseModuleGetAsyncMsgLatch
Mse Module Get Async Msg Mac Address.vi	MseModuleGetAsyncMsgMacAddress
Mse Module Get Async Msg Netmask.vi	MseModuleGetAsyncMsgNetmask
Mse Module Get Async Msg Port.vi	MseModuleGetAsyncMsgPort
Mse Module Get Async Msg Serial Number.vi	MseModuleGetAsyncMsgSerialNumber
Mse Module Get Async Msg Type.vi	MseModuleGetAsyncMsgType
Mse Module Get Async Port.vi	MseModuleGetAsycPort
Mse Module Get Bootloader Version.vi	MseModuleGetBootloaderVersion
Mse Module Get Firmware Version.vi	MseModuleGetFirmwareVersion
Mse Module Get Ip Address.vi	MseModuleGetIpAddress
Mse Module Get Ip Static Address.vi	MseModuleGetIpStaticAddress
Mse Module Get Library Version.vi	MseModuleGetLibraryVersion
Mse Module Get Mac Address.vi	MseModuleGetMacAddress
Mse Module Get Module Error State.vi	MseModuleGetModuleErrorState
Mse Module Get Module Errors.vi	MseModuleGetModuleErrors
Mse Module Get Module Type.vi	MseModuleGetModuleType
Mse Module Get Netmask Static.vi	MseModuleGetNetmaskStatic
Mse Module Get Netmask.vi	MseModuleGetNetmask
Mse Module Get Network Delay.vi	MseModuleGetNetworkDelay
Mse Module Get Port.vi	MseModuleGetPort
Mse Module Get Program Percent Complete.vi	MseModuleGetProgramPercentComplete
Mse Module Get Program State.vi	MseModuleGetProgramState
Mse Module Get Serial Number.vi	MseModuleGetSerialNumber
Mse Module Get Udp Num Retries.vi	MseModuleGetUdpNumRetries
Mse Module Get Udp Timeout.vi	MseModuleGetUdpTimeout
Mse Module Get Using Dhcp.vi	MseModuleGetUsingDhcp
Mse Module Initialize.vi	MseModuleInitialize
Mse Module Program.vi	MseModuleProgram
Mse Module Reset.vi	MseModuleReset
Mse Module Set Async Port.vi	MseModuleSetAsyncPort
Mse Module Set Broadcasting Netmask.vi	MseModuleSetBroadcastingNetmask
Mse Module Set Ip Address.vi	MseModuleSetIpAddress
Mse Module Set Network Delay.vi	MseModuleSetNetworkDelay
Mse Module Set Udp Num Retries.vi	MseModuleSetUdpNumRetries
Mse Module Set Udp Timeout.vi	MseModuleSetUdpTimeout
Mse Module Set Using Dhcp.vi	MseModuleSetUsingDhcp
Mse Module Show Id.vi	MseModuleShowId
Mse Module Show Resp Code.vi	MseModuleShowRespCode
Mse Module Show Type.vi	MseModuleShowType

## LabVIEW VI's and corresponding MSElibrary functions

LabVIEW VI's	MSElibrary Function
Mse Ttl Module Acknowledge Referencing.vi	MseTtlModuleAcknowledgeReferencing
Mse Ttl Module Clear Errors.vi	MseTtlModuleClearErrors
Mse Ttl Module Create.vi	MseTtlModuleCreate
Mse Ttl Module Delete.vi	MseTtlModuleDelete
Mse Ttl Module Enable Diags.vi	MseTtlModuleEnableDiags
Mse Ttl Module Enable Error Checking.vi	MseTtlModuleEnableErrorChecking
Mse Ttl Module Get Adc Values.vi	MseTtlModuleGetAdcValues
Mse Ttl Module Get Channel Error State.vi	MseTtlModuleGetChannelErrorState
Mse Ttl Module Get Channel Presence.vi	MseTtlModuleGetChannelPresence
Mse Ttl Module Get Counting Direction.vi	MseTtlModuleGetCountingDirection
Mse Ttl Module Get Counts.vi	MseTtlModuleGetCounts
Mse Ttl Module Get Encoder Type.vi	MseTtlModuleGetEncoderType
Mse Ttl Module Get Error Compensation.vi	MseTtlModuleGetErrorCompensation
Mse Ttl Get Fpga Revision.vi	MseTtlModuleGetFpgaRevision
Mse Ttl Module Get Latches.vi	MseTtlModuleGetLatches
Mse Ttl Module Get Line Count.vi	MseTtlModuleGetLineCount
Mse Ttl Module Get Signal Period.vi	MseTtlModuleGetSignalPeriod
Mse Ttl Module Get Module Error State.vi	MseTtlModuleGetModuleErrorState
Mse Ttl Module Get Module Errors.vi	MseTtlModuleGetModuleErrors
Mse Ttl Module Get Num Channels.vi	MseTtlModuleGetNumChannels
Mse Ttl Module Get Positions.vi	MseTtlModuleGetPositions
Mse Ttl Module Get Referencing Complete.vi	MseTtlModuleGetReferencingComplete
Mse Ttl Module Get Referencing State.vi	MseTtlModuleGetReferencingState
Mse Ttl Module Get Resolution.vi	MseTtlModuleGetResolution
Mse Ttl Module Get Uom.vi	MseTtlModuleGetUom
Mse Ttl Module Initialize.vi	MseTtlModuleInitialize
Mse Ttl Module Set Channel Presence.vi	MseTtlModuleSetChannelPresence
Mse Ttl Module Set Counting Direction.vi	MseTtlModuleSetCountingDirection
Mse Ttl Module Set Encoder Type.vi	MseTtlModuleSetEncoderType
Mse Ttl Module Set Error Compensation.vi	MseTtlModuleSetErrorCompensation
Mse Ttl Module Set Latch.vi	MseTtlModuleSetLatch
Mse Ttl Module Set Line Count.vi	MseTtlModuleSetLineCount
Mse Ttl Module Set Signal Period.vi	MseTtlModuleSetSignalPeriod
Mse Ttl Module Set Uom.vi	MseTtlModuleSetUom
Mse Ttl Module Start Referencing.vi	MseTtlModuleStartReferencing

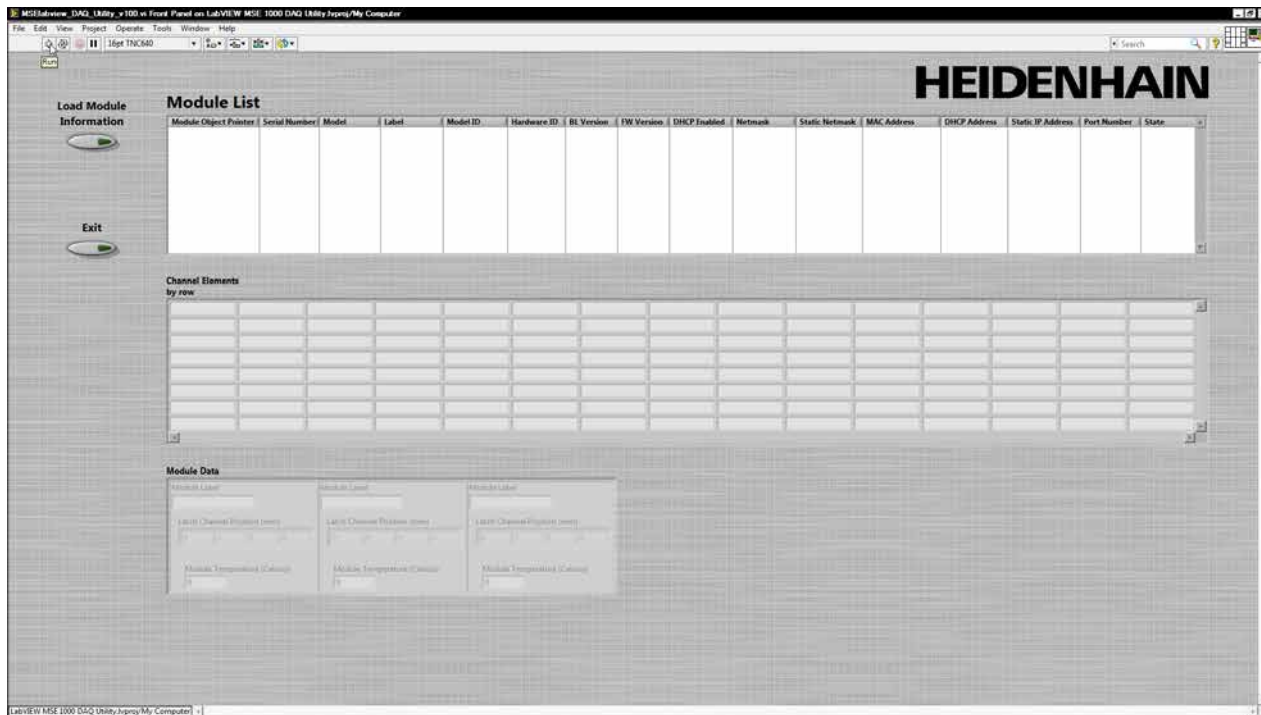
# 13.4 MSElabview DAQ utility explanation and operating instructions

The MSElabview DAQ (Data Acquisition) Utility is a VI that uses sub-VI wrappers to access the MSElibrary.dll. The DAQ utility must be run from within the LabVIEW IDE and the PC it is running on must have an Ethernet connection (direct or via a networking device) to the base module of the MSE1000 system. The utility requires MSE1000 system's initial configuration to be completed using the MSEsetup application, the resulting ModuleConfig.xml file is then accessed from the DAQ utility.

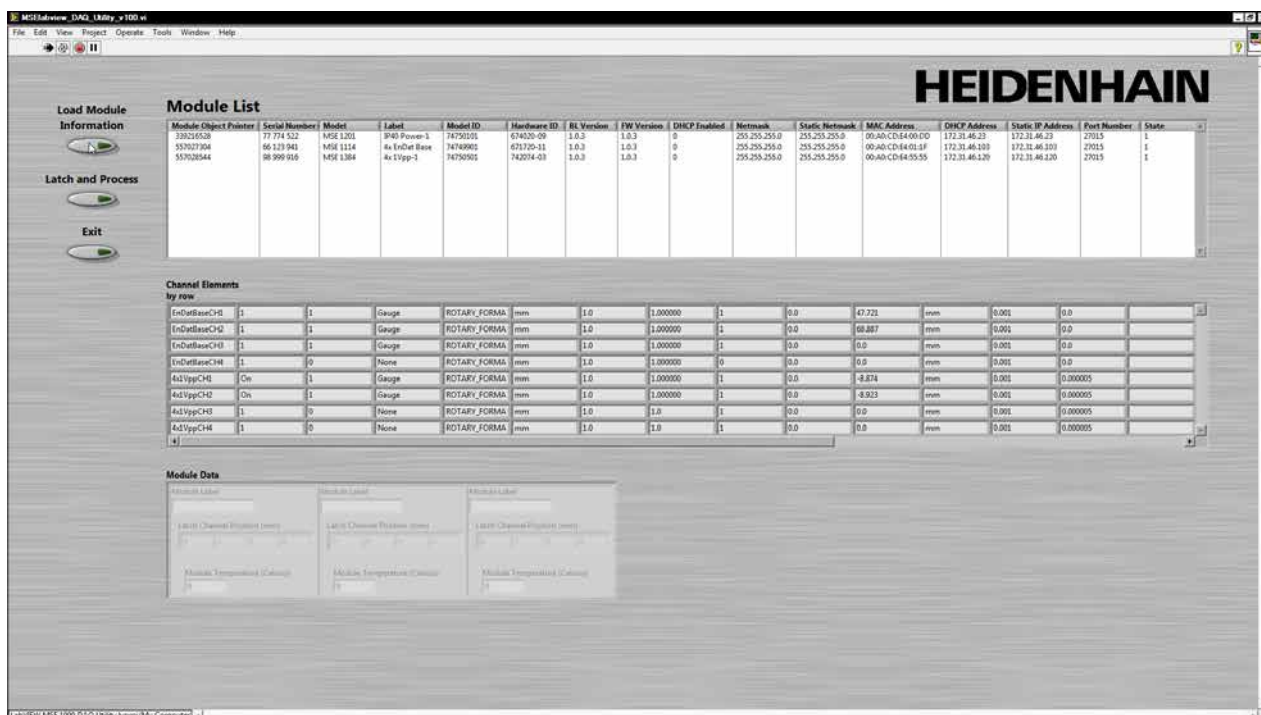
The necessary files are:

- MSElabview\_DAQ\_UTILITY\_v100.vi
- Load Module MCL.vi
- LabVIEW MSE 1000 DAQ Utility.lvproj (for LabVIEW project)
- MSElibrary.dll
- QtCore4.dll
- QtXml4.dll
- .xml file created using MSEsetup

▶ Run MSElabview\_DAQ\_utility\_v100.vi



▶ Click the *Load Module Information* button to load the ModuleConfig.xml file into memory and display it to the screen



- ▶ Click the *Latch and Process* button to latch live data and display it to the screen

Each press of this button will re-latch the live data. This can represent latching data of a part from a manufacturing run and compare it to a "gold master" part for a PASS/FAIL comparison.

A first pass shows a good part...

**HEIDENHAIN**

**Module List**

Module Object Printer	Serial Number	Model	Label	Model ID	Hardware ID	RL Version	FW Version	DHCP Enabled	Netmask	Static Netmask	MAC Address	DHCP Address	Static IP Address	Port Number	State
330216528	77 774 522	MSE 1201	IP40 Power-1	74750101	674020-09	1.0.3	1.0.3	0	255.255.255.0	255.255.255.0	00:AB:CD:E4:00:00	172.31.46.23	172.31.46.23	27015	1
557027304	86 123 941	MSE 1114	As EndCut Base	74749901	671707-11	1.0.3	1.0.3	0	255.255.255.0	255.255.255.0	00:AB:CD:E4:01:1F	172.31.46.100	172.31.46.100	27015	1
55702844	98 999 916	MSE 1384	As 1Vpp-3	74750901	742014-01	1.0.3	1.0.3	0	255.255.255.0	255.255.255.0	00:AB:CD:E4:55:55	172.31.46.120	172.31.46.120	27015	1

**Channel Elements by row**

EndCutBaseCH1	1	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	47.721	mm	0.001	0.0
EndCutBaseCH2	1	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	68.887	mm	0.001	0.0
EndCutBaseCH3	1	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	0.0	mm	0.001	0.0
EndCutBaseCH4	1	0	None	ROTARY_FORMA	mm	1.0	1.000000	0	0.0	0.0	mm	0.001	0.0
As1VppCH1	On	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	-8.874	mm	0.001	0.000005
As1VppCH2	On	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	-8.923	mm	0.001	0.000005
As1VppCH3	1	0	None	ROTARY_FORMA	mm	1.0	1.0	1	0.0	0.0	mm	0.001	0.000005
As1VppCH4	1	0	None	ROTARY_FORMA	mm	1.0	1.0	1	0.0	0.0	mm	0.001	0.000005

**Module Data**

Module Label	Module Label	Module Label
IP40 Power-1	As EndCut Base	As 1Vpp-3
Latch Channel Position (mm): 0	Latch Channel Position (mm): 47.7211 68.8887 0 0	Latch Channel Position (mm): -8.8677 -8.9229 -0.0111 0.02704
Module Temperature (Celsius): 58.2	Module Temperature (Celsius): 45.7	Module Temperature (Celsius): 48.9

**PASS**

a second pass reveals a part that does not meet PASS criteria.

**HEIDENHAIN**

**Module List**

Module Object Printer	Serial Number	Model	Label	Model ID	Hardware ID	RL Version	FW Version	DHCP Enabled	Netmask	Static Netmask	MAC Address	DHCP Address	Static IP Address	Port Number	State
330216528	77 774 522	MSE 1201	IP40 Power-1	74750101	674020-09	1.0.3	1.0.3	0	255.255.255.0	255.255.255.0	00:AB:CD:E4:00:00	172.31.46.23	172.31.46.23	27015	1
557027304	86 123 941	MSE 1114	As EndCut Base	74749901	671707-11	1.0.3	1.0.3	0	255.255.255.0	255.255.255.0	00:AB:CD:E4:01:1F	172.31.46.100	172.31.46.100	27015	1
55702844	98 999 916	MSE 1384	As 1Vpp-3	74750901	742014-01	1.0.3	1.0.3	0	255.255.255.0	255.255.255.0	00:AB:CD:E4:55:55	172.31.46.120	172.31.46.120	27015	1

**Channel Elements by row**

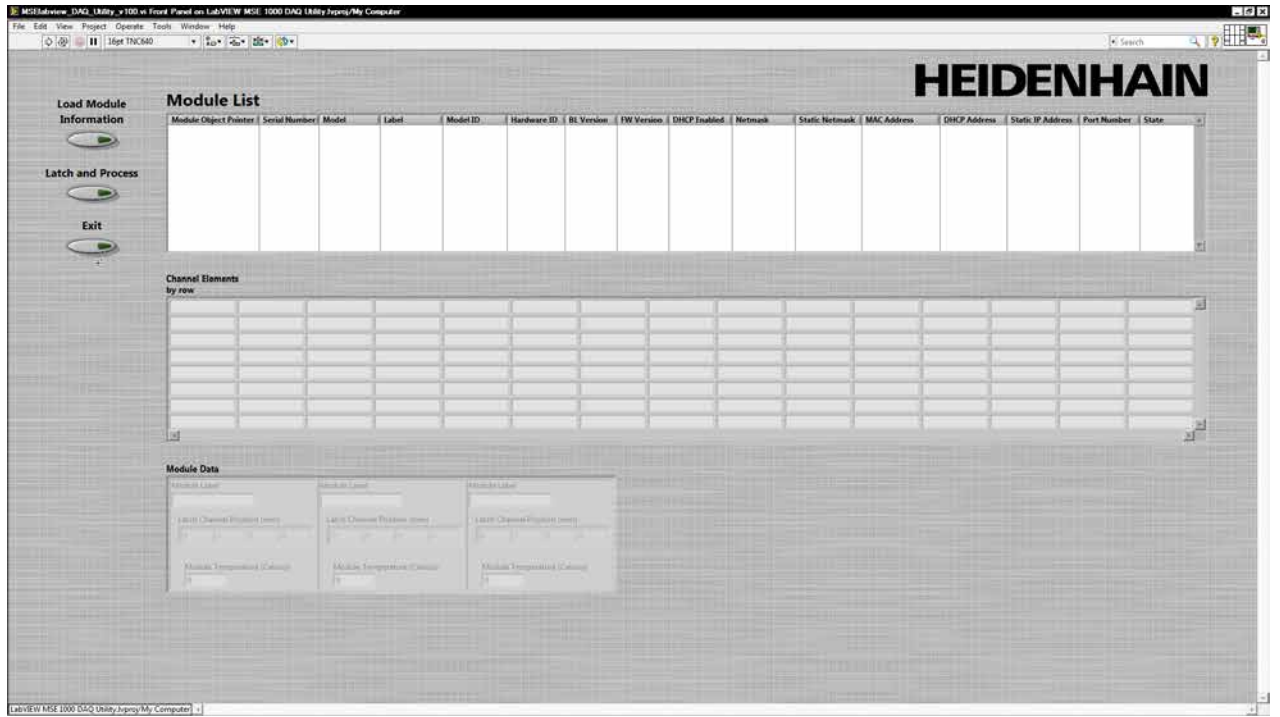
EndCutBaseCH1	1	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	47.721	mm	0.001	0.0
EndCutBaseCH2	1	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	68.887	mm	0.001	0.0
EndCutBaseCH3	1	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	0.0	mm	0.001	0.0
EndCutBaseCH4	1	0	None	ROTARY_FORMA	mm	1.0	1.000000	0	0.0	0.0	mm	0.001	0.0
As1VppCH1	On	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	-8.874	mm	0.001	0.000005
As1VppCH2	On	1	Gauge	ROTARY_FORMA	mm	1.0	1.000000	1	0.0	-8.923	mm	0.001	0.000005
As1VppCH3	1	0	None	ROTARY_FORMA	mm	1.0	1.0	1	0.0	0.0	mm	0.001	0.000005
As1VppCH4	1	0	None	ROTARY_FORMA	mm	1.0	1.0	1	0.0	0.0	mm	0.001	0.000005

**Module Data**

Module Label	Module Label	Module Label
IP40 Power-1	As EndCut Base	As 1Vpp-3
Latch Channel Position (mm): 0	Latch Channel Position (mm): 47.721 68.8887 0 0	Latch Channel Position (mm): 0.00912 -1.9070 -0.3085 0.20687
Module Temperature (Celsius): 58.2	Module Temperature (Celsius): 46.7	Module Temperature (Celsius): 48.5

**FAIL**

- ▶ Once done processing click the *Exit* button







**Symbols**

1Vpp functions ... 122  
1Vpp methods ... 122

**A**

acknowledgeAbsolutePosition ... 127  
acknowledgeReferencing ... 196  
ADC\_OPTIONS ... 37  
addModule ... 61  
ANALOG\_DIAG\_VOLTAGES\_ENUM ... 50  
asynchronous methods ... 211

**B**

broadcastOpenConnection ... 72

**C**

C examples ... 273  
classes ... 26, 52  
clearAllErrors ... 77  
clearErrorsAndWarnings ... 99  
clearIntegrityErrors ... 77  
computeResolutionAndOffset ... 180  
configuring MSE 1000 ... 19  
constants ... 57  
COUNTER\_STATUS ... 41  
COUNT\_REQUEST\_OPTION ... 37  
counts, getting ... 263  
COUNTS\_PER\_LINE ... 59  
createChain ... 62

**D**

debouncing latency ... 235  
decodeChannelStatusMsg ... 212  
decodeConnectMsg ... 211  
decodeElementType ... 214  
decodeErrorType ... 213  
decodeLatchMsg ... 212  
definitions ... 25  
Delphi examples ... 283  
detectSignalType ... 128  
DeviceData ... 53  
DEVICE\_ID\_SIZE ... 57  
device methods ... 95  
DEVICE\_NAME\_SIZE ... 57  
DHCP, changing ... 22  
diagnostic modes ... 232  
  full ... 232  
  minimal ... 232  
  none ... 232  
  status ... 232

**E**

enableAnalogDiag ... 123  
enableDiags ... 77  
enableErrorChecking ... 99  
EncoderInfo ... 55  
Encoder, setting information ... 264  
ENCODER\_TYPES\_ENUM ... 38  
ENDAT\_DIAG ... 33

HEIDENHAIN MSElibrary

ENDAT\_ERROR\_RESULT ... 32  
ENDAT\_ERRORS ... 32  
EnDat functions ... 101  
EnDat methods ... 101  
ENDAT\_WARNINGS ... 33  
enumerations ... 29  
Ethernet cable ... 20

**F**

firmware ... 3  
firmware, programming ... 270  
fonts ... 3  
functions ... 25

**G**

general functions ... 65  
general methods ... 65  
get1VppModule ... 63  
getAdcValues ... 75  
getAllElements ... 215  
getAnalogModule ... 64  
getAsyncMsgType ... 211  
getAsyncPort ... 72  
getChainCreationState ... 62  
getChannelPresence ... 106, 161, 195  
getChannelStatus ... 99  
getCode ... 56  
getConfig ... 66  
getCountingDirection ... 96  
getCounts ... 68, 102, 192  
getCurrent ... 177  
getDeviceData ... 104  
getDeviceModule ... 63  
getDeviceOffset ... 69  
getDiag ... 103, 122  
getDiagVoltages ... 178  
getDistinguishableRevolutions ... 104  
getElement ... 214, 215  
getEncoderId ... 105  
getEncoderInfo ... 95  
getEncoderName ... 105  
getEncoderType ... 98  
getEndatModule ... 63  
getErrorCompensation ... 96  
getErrors ... 103  
getExcitationValues ... 159  
getFilename ... 214  
getFpgaRevision ... 162, 196  
getInputs ... 146  
getIntegrity ... 76  
getIO ... 146  
getIoModule ... 63  
getLatch ... 75  
getLeft ... 67  
getLibraryVersion ... 77  
getLine ... 56  
getLineCount ... 125, 193  
getLvdtModule ... 64  
getMethod ... 56  
getModule ... 62  
getModuleData ... 67  
getModuleType ... 66  
getNetworkDelay ... 74

getNumChannels ... 67, 217  
getNumModules ... 62, 217  
getOffset ... 180  
getOutput ... 152  
getOutputs ... 145  
getPneumaticModule ... 64  
getPositions ... 101, 123, 160, 191  
getProgrammingPercentComplete ... 70  
getProgrammingState ... 70  
getReferencingState ... 127, 196  
getResolution ... 98, 161, 179  
getRotaryFormat ... 68  
getScaledValues ... 178  
getScaling ... 97, 181  
getSensorGain ... 163  
getSerialNumber ... 105  
getSignalPeriod ... 125, 194  
getSignalType ... 127  
getSpecificChannel ... 216  
getSpecificModule ... 216  
getTeachSensorGainFinished ... 164  
getTtlModule ... 64  
getUdpNumRetries ... 74  
getUdpTimeout ... 73  
getUom ... 98, 158  
getValues ... 177  
getVoltage ... 160, 176  
getWarnings ... 102

**I**

initAbsolutePosition ... 126  
initializeFirmware ... 65  
initializeModule ... 65, 101, 122, 144, 152, 158, 176, 191  
initReferencing ... 195  
installation ... 24  
INTEGRITY\_ENUMS ... 39  
interface methods ... 61  
INTERPOLATION\_VALUE ... 59  
I/O functions ... 144  
I/O methods ... 144  
IP address  
  changing ... 20  
  initial ... 20  
isReferencingComplete ... 126, 195

**L**

LabVIEW ... 285  
latch  
  determining which are set ... 235  
  reading data ... 235  
LATCH\_CHOICE ... 36  
latching ... 266  
LATCH\_OPTIONS ... 36  
LeftData ... 54  
library software ... 23  
loadXml ... 213  
LVDT\_EXCITATION\_FREQUENCY\_MAX\_KHZ ... 60  
LVDT\_EXCITATION\_FREQUENCY\_MIN\_KHZ ... 60  
LVDT\_EXCITATION\_VOLTAGE\_

MAX\_VPP ... 60  
 LVDT\_EXCITATION\_VOLTAGE\_  
 MIN\_VPP ... 60  
 LVDT methods and func-  
 tions ... 158  
 LVDT\_UOM ... 49  
 LVDT\_UPDATE\_CHOICES ... 49

## M

MAX\_CHANNELS\_PER\_MODULE ... 58  
 MAX\_NUM\_ANALOG\_AVG\_SAMPLES ... 60  
 MAX\_NUM\_MODULES ... 58  
 methods ... 25  
 module chain  
   creating, broadcasting ... 261  
   creating, manually ... 262  
   initializing ... 261  
 ModuleConfig base ... 213  
 ModuleConfig reader ... 213  
 ModuleConfig writer ... 213  
 ModuleData ... 52  
 MODULE\_ID ... 30  
 modules ... 3, 25  
 Mse1VppDetectSignalType ... 143  
 Mse1VppGetSignalType ... 143  
 Mse1VppModule ... 26  
 Mse1VppModuleAcknowledgeAbsolutePosition ... 142  
 Mse1VppModuleClearErrors ... 138  
 Mse1VppModuleCreate ... 128  
 Mse1VppModuleDelete ... 128  
 Mse1VppModuleEnableAnalogDiag ... 139  
 Mse1VppModuleEnableDiags ... 138  
 Mse1VppModuleEnableErrorChecking ... 142  
 Mse1VppModuleGetAdcValues ... 139  
 Mse1VppModuleGetAnalogDiag ... 139  
 Mse1VppModuleGetChannelErrorState ... 137  
 Mse1VppModuleGetChannelStatus ... 137  
 Mse1VppModuleGetCountingDirection ... 133  
 Mse1VppModuleGetCounts ... 133  
 Mse1VppModuleGetDeviceOffset ... 135  
 Mse1VppModuleGetEncoderType ... 130  
 Mse1VppModuleGetErrorCompensation ... 131  
 Mse1VppModuleGetLatches ... 136  
 Mse1VppModuleGetLineCount ... 140  
 Mse1VppModuleGetModuleErrors ... 137  
 Mse1VppModuleGetModuleErrorState ... 136

Mse1VppModuleGetNumChannels ... 129  
 Mse1VppModuleGetPosition ... 134  
 Mse1VppModuleGetReferencingComplete ... 141  
 Mse1VppModuleGetReferencingState ... 142  
 Mse1VppModuleGetResolution ... 133  
 Mse1VppModuleGetRotaryFormat ... 135  
 Mse1VppModuleGetSignalPeriod ... 141  
 Mse1VppModuleGetUom ... 130  
 Mse1VppModuleInitialize ... 129  
 Mse1VppModuleSetCountingDirection ... 132  
 Mse1VppModuleSetDeviceOffset ... 135  
 Mse1VppModuleSetEncoderType ... 129  
 Mse1VppModuleSetErrorCompensation ... 131  
 Mse1VppModuleSetLatch ... 136  
 Mse1VppModuleSetLatchDebouncing ... 138  
 Mse1VppModuleSetLineCount ... 140  
 Mse1VppModuleSetRotaryFormat ... 134  
 Mse1VppModuleSetSignalPeriod ... 140  
 Mse1VppModuleSetUom ... 130  
 Mse1VppModuleStartReferencing ... 141  
 Mse1VppModuleWrapper ... 27  
 Mse1VppSetSignalType ... 143  
 MSE1000\_ASYNC\_PORT ... 59  
 MSE1000\_CLIENT\_DEFAULT\_PORT ... 59  
 MSE1000ConnectResponse ... 54  
 MSE1000\_PORT ... 59  
 MseAnalogModule ... 26  
 MseAnalogModuleClearErrors ... 184  
 MseAnalogModuleClearLatch ... 185  
 MseAnalogModuleComputeResolutionAndOffset ... 190  
 MseAnalogModuleCreate ... 182  
 MseAnalogModuleDelete ... 182  
 MseAnalogModuleGetAdcValues ... 184  
 MseAnalogModuleGetCurrent ... 185  
 MseAnalogModuleGetDeviceOffset ... 187  
 MseAnalogModuleGetDiagVoltages ... 188  
 MseAnalogModuleGetLatch ... 184  
 MseAnalogModuleGetModuleErrors ... 183  
 MseAnalogModuleGetModuleErrorState ... 183

MseAnalogModuleGetNumChannels ... 183  
 MseAnalogModuleGetOffset ... 190  
 MseAnalogModuleGetResolution ... 189  
 MseAnalogModuleGetScaledValues ... 186  
 MseAnalogModuleGetScaling ... 188  
 MseAnalogModuleGetValues ... 186  
 MseAnalogModuleGetVoltage ... 185  
 MseAnalogModuleInitialize ... 182  
 MseAnalogModuleSetDeviceOffset ... 187  
 MseAnalogModuleSetNumSamples ... 188  
 MseAnalogModuleSetOffset ... 189  
 MseAnalogModuleSetResolution ... 189  
 MseAnalogModuleSetScaling ... 187  
 MseAnalogModuleWrapper ... 27  
 MSE\_CHAIN\_CREATION\_STATE ... 29  
 MseComm ... 26  
 MseConfigBase ... 26, 213  
 MseConfigFileCreate ... 219  
 MseConfigFileDecodeElementType ... 220  
 MseConfigFileDecodeErrorType ... 220  
 MseConfigFileDelete ... 219  
 MseConfigFileGetAllElements ... 222  
 MseConfigFileGetChannelElement ... 221  
 MseConfigFileGetElement ... 221  
 MseConfigFileGetFilename ... 220  
 MseConfigFileGetModuleElement ... 221  
 MseConfigFileGetNumChannels ... 224  
 MseConfigFileGetNumModules ... 223  
 MseConfigFileGetSpecificChannel ... 223  
 MseConfigFileGetSpecificModule ... 223  
 MseConfigFileLoadXml ... 219  
 MseConfigFileReloadXml ... 219  
 MseConfigFileRemoveModule ... 225  
 MseConfigFileSetChannelElement ... 224  
 MseConfigFileSetModuleElement ... 224  
 MseConfigFileValidateElements ... 222  
 MseConfigFileWrapper ... 27  
 MseConfigFileWriteFile ... 225  
 MseConfigReader ... 27, 214, 271

- MseConfigWriter ... 27, 217
- MseDeviceModule ... 26, 95
- MseEndatModule ... 26
- MseEndatModuleClearErrors ... 118
- MseEndatModuleCreate ... 107
- MseEndatModuleDelete ... 107
- MseEndatModuleEnableDiags ... 120
- MseEndatModuleEnableErrorChecking ... 121
- MseEndatModuleGetAdcValues ... 121
- MseEndatModuleGetChannelErrorState ... 116
- MseEndatModuleGetChannelPresence ... 108
- MseEndatModuleGetChannelStatus ... 117
- MseEndatModuleGetChannelWarningState ... 117
- MseEndatModuleGetCountingDirection ... 111
- MseEndatModuleGetCounts ... 112
- MseEndatModuleGetDeviceOffset ... 114
- MseEndatModuleGetDiags ... 120
- MseEndatModuleGetDistinguishableRevolutions ... 111
- MseEndatModuleGetEncoderId ... 119
- MseEndatModuleGetEncoderName ... 118
- MseEndatModuleGetEncoderSerialNumber ... 119
- MseEndatModuleGetEncoderType ... 108
- MseEndatModuleGetEndatErrors ... 117
- MseEndatModuleGetEndatWarnings ... 118
- MseEndatModuleGetErrorCompensation ... 110
- MseEndatModuleGetLatches ... 115
- MseEndatModuleGetModuleErrors ... 116
- MseEndatModuleGetModuleErrorState ... 116
- MseEndatModuleGetNumChannels ... 108
- MseEndatModuleGetPositions ... 113
- MseEndatModuleGetResolution ... 112
- MseEndatModuleGetRotaryFormat ... 114
- MseEndatModuleGetScaling ... 110
- MseEndatModuleGetUom ... 109
- MseEndatModuleInitialize ... 107
- MseEndatModuleSetDeviceOffset ... 114
- MseEndatModuleSetErrorCompensation ... 109
- MseEndatModuleSetLatch ... 115
- MseEndatModuleSetLatchingDe-bouncing ... 119
- MseEndatModuleSetRotaryFormat ... 113
- MseEndatModuleSetScaling ... 110
- MseEndatModuleSetUom ... 109
- MseEndatModuleWrapper ... 27
- MseInterface ... 26, 61
- MseloModule ... 26
- MseloModuleClearErrors ... 149
- MseloModuleClearLatch ... 151
- MseloModuleCreate ... 147
- MseloModuleDelete ... 147
- MseloModuleGetAdcValues ... 149
- MseloModuleGetInputs ... 150
- MseloModuleGetIO ... 151
- MseloModuleGetLatch ... 151
- MseloModuleGetModuleErrors ... 148
- MseloModuleGetModuleErrorState ... 148
- MseloModuleGetNumChannels ... 148
- MseloModuleGetOutputs ... 150
- MseloModuleInitialize ... 147
- MseloModuleSetOutput ... 150
- MseloModuleSetOutputs ... 149
- MseloModuleWrapper ... 27
- MSElabview DAQ utility ... 293
- MSElibrary1VppWrapperVl.lib ... 287
- MSElibraryAnalogWrapperVl.lib ... 288
- MSElibraryConfigFileWrapperVls.lib ... 288
- MSElibraryEndatWrapperVls.lib ... 289
- MSElibraryIoWrapperVls.lib ... 290
- MSElibraryLvdtWrapperVls.lib ... 290
- MSElibraryModuleWrapperVls.lib ... 291
- MSElibraryPneumaticWrapperVls.lib ... 289
- MSElibraryTtlWrapperVls.lib ... 292
- MseLvdtGetFpgaRevision ... 175
- MseLvdtGetSensorGain ... 174
- MseLvdtGetTeachSensorGainFinished ... 175
- MseLvdtModule ... 26
- MseLvdtModuleClearErrors ... 169
- MseLvdtModuleClearLatch ... 168
- MseLvdtModuleCreate ... 164
- MseLvdtModuleDelete ... 164
- MseLvdtModuleEnableDiags ... 169
- MseLvdtModuleGetAdcValues ... 169
- MseLvdtModuleGetChannelPresence ... 173
- MseLvdtModuleGetDeviceOffset ... 171
- MseLvdtModuleGetExcitationValues ... 171
- MseLvdtModuleGetLatch ... 167
- MseLvdtModuleGetModuleErrors ... 168
- MseLvdtModuleGetModuleErrorState ... 168
- MseLvdtModuleGetNumChannels ... 165
- MseLvdtModuleGetPositions ... 170
- MseLvdtModuleGetResolution ... 166
- MseLvdtModuleGetScaling ... 167
- MseLvdtModuleGetUom ... 166
- MseLvdtModuleGetVoltage ... 172
- MseLvdtModuleInitialize ... 165
- MseLvdtModuleSetChannelPresence ... 173
- MseLvdtModuleSetDeviceOffset ... 170
- MseLvdtModuleSetExcitationFrequency ... 172
- MseLvdtModuleSetExcitationVoltage ... 172
- MseLvdtModuleSetOversampling ... 175
- MseLvdtModuleSetResolution ... 166
- MseLvdtModuleSetScaling ... 167
- MseLvdtModuleSetUom ... 165
- MseLvdtModuleWrapper ... 27
- MseLvdtSetDiagnosticsEnabled ... 173
- MseLvdtSetSensorGain ... 174
- MseLvdtTeachSensorGain ... 174
- MseModule ... 26, 65
- MseModuleBroadcast ... 89
- MseModuleClearErrors ... 81
- MseModuleCreate ... 78
- MseModuleDelete ... 78
- MseModuleGetAdcValues ... 80
- MseModuleGetAsyncMsgChannelStatus ... 93
- MseModuleGetAsyncMsgDh-cp ... 92
- MseModuleGetAsyncMsgIpAddress ... 91
- MseModuleGetAsyncMsgSignalLatch ... 94
- MseModuleGetAsyncMsgMacAddress ... 92
- MseModuleGetAsyncMsgNetmask ... 92
- MseModuleGetAsyncMsgSerialPort ... 91
- MseModuleGetAsyncMsgSerialNumber ... 93
- MseModuleGetAsyncMsgSignalType ... 91
- MseModuleGetBootloaderVersion ... 85
- MseModuleGetFirmwareVer-

- tion ... 85
  - MseModuleGetIpAddress ... 81
  - MseModuleGetIpStaticAddress ... 82
  - MseModuleGetLibraryVersion ... 79
  - MseModuleGetMacAddress ... 84
  - MseModuleGetModuleErrors ... 80
  - MseModuleGetModuleErrorState ... 79
  - MseModuleGetModuleType ... 79
  - MseModuleGetNetmask ... 82
  - MseModuleGetNetmaskStatic ... 82
  - MseModuleGetNetworkDelay ... 89
  - MseModuleGetPort ... 83
  - MseModuleGetProgramPercentComplete ... 90
  - MseModuleGetProgramState ... 90
  - MseModuleGetSerialNumber ... 85
  - MseModuleGetUdpNumRetries ... 88
  - MseModuleGetUdpTimeout ... 87
  - MseModuleGetUsingDhcp ... 84
  - MseModuleInitialize ... 78
  - MseModuleProgram ... 90
  - MseModuleReset ... 86
  - MseModuleSetAsyncPort ... 83
  - MseModuleSetBroadcastingNetmask ... 87
  - MseModuleSetIpAddress ... 81
  - MseModuleSetNetworkDelay ... 88
  - MseModuleSetUdpNumRetries ... 88
  - MseModuleSetUdpTimeout ... 87
  - MseModuleSetUsingDhcp ... 84
  - MseModuleShowId ... 86
  - MseModuleShowRespCode ... 94
  - MseModuleShowType ... 86
  - MseModuleWrapper ... 27
  - MsePneumaticModule ... 26
  - MsePneumaticModuleClearErrors ... 156
  - MsePneumaticModuleClearLatch ... 157
  - MsePneumaticModuleCreate ... 153
  - MsePneumaticModuleDelete ... 153
  - MsePneumaticModuleGetAdcValues ... 155
  - MsePneumaticModuleGetLatch ... 157
  - MsePneumaticModuleGetModuleErrors ... 155
  - MsePneumaticModuleGetModuleErrorState ... 154
  - MsePneumaticModuleGetNumChannels ... 154
  - MsePneumaticModuleGetOutput ... 156
  - MsePneumaticModuleInitialize ... 153
  - MsePneumaticModuleSetOutput ... 156
  - MsePneumaticModuleWrapper ... 27
  - MSE\_RESPONSE\_CODE ... 34
  - MseTtlModule ... 26
  - MseTtlModuleAcknowledgeReferencing ... 209
  - MseTtlModuleClearErrors ... 208
  - MseTtlModuleCreate ... 197
  - MseTtlModuleDelete ... 197
  - MseTtlModuleEnableDiags ... 207
  - MseTtlModuleEnableErrorChecking ... 210
  - MseTtlModuleGetAdcValues ... 208
  - MseTtlModuleGetChannelErrorState ... 210
  - MseTtlModuleGetChannelPresence ... 202
  - MseTtlModuleGetCountingDirection ... 201
  - MseTtlModuleGetCounts ... 204
  - MseTtlModuleGetDeviceOffset ... 206
  - MseTtlModuleGetEncoderType ... 198
  - MseTtlModuleGetErrorCompensation ... 200
  - MseTtlModuleGetFpgaRevision ... 210
  - MseTtlModuleGetLatches ... 206
  - MseTtlModuleGetLineCount ... 203
  - MseTtlModuleGetModuleErrors ... 207
  - MseTtlModuleGetModuleErrorState ... 207
  - MseTtlModuleGetNumChannels ... 198
  - MseTtlModuleGetPositions ... 204
  - MseTtlModuleGetReferencingState ... 209
  - MseTtlModuleGetResolution ... 202
  - MseTtlModuleGetRotaryFormat ... 205
  - MseTtlModuleGetScaling ... 200
  - MseTtlModuleGetSignalPeriod ... 203
  - MseTtlModuleGetUom ... 199
  - MseTtlModuleInitialize ... 197
  - MseTtlModuleIsReferencingComplete ... 208
  - MseTtlModuleSetChannelPresence ... 201
  - MseTtlModuleSetCountingDirection ... 201
  - MseTtlModuleSetDeviceOffset ... 205
  - MseTtlModuleSetEncoderType ... 198
  - MseTtlModuleSetErrorCompensation ... 199
  - MseTtlModuleSetLatch ... 206
  - MseTtlModuleSetLineCount ... 202
  - MseTtlModuleSetRotaryFormat ... 205
  - MseTtlModuleSetScaling ... 200
  - MseTtlModuleSetSignalPeriod ... 203
  - MseTtlModuleSetUom ... 199
  - MseTtlModuleStartReferencing ... 209
  - MseTtlModuleWrapper ... 27
  - MSE\_XML\_ELEMENTS ... 43
  - MSE\_XML\_RETURN ... 42
- ## N
- NUM\_ENDAT\_ERRORS ... 32
  - NUM\_ENDAT\_WARNINGS ... 33
  - NUM\_INTEGRITY\_RANGES ... 59
  - NUM\_LATCH\_TYPES ... 59
  - NUM\_LVDT\_CHANNELS ... 59
  - NUM\_MSE1000\_ANALOG\_CHANNELS ... 60
  - NUM\_MSE1000\_ANALOG\_VALUES\_PER\_CHANNEL ... 60
  - NUM\_MSE1000\_IO\_INPUTS ... 57
  - NUM\_MSE1000\_IO\_OUTPUTS ... 57
- ## O
- overview ... 25
- ## P
- pneumatic functions ... 152
  - pneumatic methods ... 152
  - prerequisites ... 25
  - program ... 70
  - PROGRAMMING\_STATE\_ENUMS ... 46
- ## R
- REFERENCE\_MARK\_ENUM ... 40
  - reloadXml ... 213
  - removeConnections ... 61
  - removeSpecificModuleNode ... 214
  - resetMse1000 ... 69
  - restoreFactoryDefaults ... 73
  - return values ... 56
  - ROTARY\_FORMAT ... 36
- ## S
- SERIAL\_NUMBER\_SIZE ... 57
  - setAsyncMode ... 76
  - setAsyncPort ... 71
  - setBroadcastingNetmask ... 73
  - setChannelPresence ... 161, 194
  - setCountingDirection ... 126, 194
  - setDeviceOffset ... 69
  - setDhcp ... 72
  - setDiagnosticsEnabled ... 162

setElement ... 217, 218  
 setEncoderInfo ... 95, 106  
 setEncoderType ... 124, 192  
 setErrorCompensation ... 96  
 setExcitationFrequency ... 160  
 setExcitationVoltage ... 159  
 setIp ... 71  
 setLatch ... 75  
 setLatchDebouncing ... 100  
 setLineCount ... 124, 193  
 setNetworkDelay ... 74  
 setNumSamples ... 179  
 setOffset ... 180  
 setOutput ... 145, 152  
 setOutputs ... 144  
 setOversampling ... 162  
 setResolution ... 162, 179  
 setRight ... 69  
 setRotaryFormat ... 68  
 setScaling ... 97, 181  
 setSensorGain ... 163  
 setSignalPeriod ... 125, 193  
 setSignalType ... 128  
 setUdpNumRetries ... 74  
 setUdpTimeout ... 73  
 setUom ... 106, 124, 159, 192  
 showModuleId ... 71  
 showModuleType ... 70  
 showRespCode ... 56  
 SIGNAL\_TYPE ... 51  
 SIZE\_BUILD\_INFO ... 58  
 SIZE\_IP\_ADDRESS ... 58  
 SIZE\_MAC\_ADDRESS ... 58  
 SIZE\_SERIAL\_NUMBER ... 58  
 software latency ... 234  
 structures ... 52  
 system integrity ... 228  
   3.3V ... 229  
   5V ... 229  
   24V ... 229  
   CPU temperature ... 229  
   current ... 229  
   duplicate IP address ... 228  
   Ethernet chip ... 228  
   obtaining IP address ... 228  
   programming error ... 228  
   waiting for client ... 228

## T

teachSensorGain ... 163  
 trigger  
   clearing ... 235  
   setting ... 235  
 trigger line ... 234  
 TTL\_INTERPOLATION ... 50

## U

UdpCmdType ... 47  
 UdpComm ... 26  
 UOM ... 29

## V

validateElements ... 216  
 Visual Basic examples ... 279

VPP\_VOLTAGE\_FEEDBACK ... 38

## W

wrappers ... 27  
 writeFile ... 218





# HEIDENHAIN


---

## DR. JOHANNES HEIDENHAIN GmbH

Dr.-Johannes-Heidenhain-Straße 5

83301 Traunreut, Germany

☎ +49 8669 31-0

 +49 8669 32-5061

E-mail: [info@heidenhain.de](mailto:info@heidenhain.de)

---

**Technical support**  +49 8669 32-1000

**Measuring systems** ☎ +49 8669 31-3104

E-mail: [service.ms-support@heidenhain.de](mailto:service.ms-support@heidenhain.de)

**TNC support** ☎ +49 8669 31-3101

E-mail: [service.nc-support@heidenhain.de](mailto:service.nc-support@heidenhain.de)

**NC programming** ☎ +49 8669 31-3103

E-mail: [service.nc-pgm@heidenhain.de](mailto:service.nc-pgm@heidenhain.de)

**PLC programming** ☎ +49 8669 31-3102

E-mail: [service.plc@heidenhain.de](mailto:service.plc@heidenhain.de)

**Lathe controls** ☎ +49 8669 31-3105

E-mail: [service.lathe-support@heidenhain.de](mailto:service.lathe-support@heidenhain.de)

---

[www.heidenhain.de](http://www.heidenhain.de)