Software

--Table of Contents--

Object	pg
Adc	2
Axis	22
Capture	58
Command	80
Compare	100
Control	124
Event	171
EventMask	185
EventMgr	209
Filter	247
Flash	288
Global*	308
Idn	323
IdnList	363

Object	pg
Message*	382
Motion	388
Motor	470
Node	550
Notify	586
Object*	611
Path	631
Platform	655
Recorder	664
Sequence	696
Sercos	738
Service	814
Trace*	818

* - Modules

Copyright © 2002
Motion Engineering

Adc Objects

Introduction

An **Adc** object manages a single analog-to-digital converter on an XMP motion controller. The XMP controller supports 8 channels (0 to 7) of 16 bit analog-to-digital conversions.

To configure the voltage range for an analog input, use the method **mpiAdcConfigSet**(MPIAdc adc, MPIAdcConfig *config, void *external), where *adc* is a handle to an MPIAdc object, and *config* is a pointer to a MPIAdcConfig structure. The MPIAdcConfig structure contains a “range” element, which specifies the analog input’s voltage range. The XMP controller supports voltage ranges of 10.0V, 5.0V, 2.5V, or 1.25V volt ranges (all ranges are bipolar). Set the MPIAdcConfig range to one of the following values:

Bipolar Voltage Range (+/- V)
10.0
5.0
2.5
1.25

To use the ADC, first create an ADC object using **mpiAdcCreate**(MPIControl control, long number), where *control* is a handle to an MPIControl object and *number* is the analog-to-digital converter channel number.

The number of ADCs on the controller can be configured using the method **mpiControlConfigSet**(MPIControl control, MPIControlConfig *config, void *external). The MPIControlConfig structure contains an element called adcCount, which is used to specify the number of analog-to-digital converter channels. The default setting is for eight (8) axes.

After the ADC has been configured, to read a channel use the method **mpiAdc Input**(MPIAdc adc, unsigned long *input), where adc is a handle to an MPIAdc object, and input is a pointer to the digitally converted value.

Methods

Create, Delete, Validate Methods

mpiAdc**Create**	Create Adc object
mpiAdc**Delete**	Delete Adc object
mpiAdc**Validate**	Validate Adc object

Configuration and Information Methods

mpiAdc**ConfigGet**	Get Adc configuration
mpiAdc**ConfigSet**	Set Adc configuration
mpiAdc**FlashConfigGet**	Get Adc flash config
mpiAdc**FlashConfigSet**	Set Adc flash config

Action Methods

mpiAdc**Input**	Read input level

Memory Methods

mpiAdc**Memory**	Set address to Adc memory
mpiAdc**MemoryGet**	Copy Adc memory to application memory
mpiAdc**MemorySet**	Copy application memory to Adc memory

Relational Methods

mpiAdc**Control**	Return handle of Control object associated with Adc
mpiAdc**MotorMapGet**	Get the object map and write it into the structure pointed to by map
mpiAdc**Number**	Get index of Adc

Data Types

MPIAdc**Config** / MEIAdc**Config**
MPIAdc**Message**
MEIAdc**Mux**

Copyright © 2002
Motion Engineering

mpiAdcCreate

Declaration

```
const MPIAdc mpiAdcCreate(MPIControl control,
                          long       number)
```

Required Header

stdmpi.h

Description

AdcCreate creates an Adc object associated with the analog-to-digital converter identified by ***number*** located on motion controller ***control***. AdcCreate is the equivalent of a C++ constructor.

control	a handle to the motion controller.
number	the adc number

Return Values

MPIHandleVOID	if the object could not be created
handle	to an Adc object

See Also

mpiAdcDelete | mpiAdcValidate

mpiAdcDelete

Declaration `long mpiAdcDelete(MPIAdc adc)`

Required Header adc.h

Description **AdcDelete** deletes an Adc object and invalidates its handle (***adc***). *AdcDelete* is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *AdcDelete* successfully deletes the Adc object and invalidates its handle

See Also mpiAdcCreate | mpiAdcValidate

mpiAdcValidate

Declaration long **mpiAdcValidate**(MPIAdc **adc**)

Required Header stdmpi.h

Description **AdcValidate** validates the Adc object and its handle (***adc***).

Return Values	
MPIMessageOK	if Adc is a handle to a valid object.

See Also mpiAdcCreate | mpiAdcDelete

mpiAdcConfigGet

Declaration

```
long mpiAdcConfigGet(MPIAdc        adc,
                     MPIAdcConfig  *config,
                     void          *external)
```

Required Header

adc.h

Description

AdcConfigGet gets the configuration of an Adc object (***adc***) and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type ***MEIAdcConfig{}*** or is NULL.

Return Values

MPIMessageOK	if *AdcConfigGet* successfully gets the configuration of an Adc object and writes it to the structure

See Also

mpiAdcConfigSet | MEIAdcConfig

mpiAdcConfigSet

Declaration

```
long mpiAdcConfigSet(MPIAdc       adc,
                     MPIAdcConfig *config,
                     void         *external)
```

Required Header

stdmpi.h

Description

AdcConfigSet sets the configuration of an Adc object (***adc***) using the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type ***MEIAdcConfig{}*** or is NULL.

Return Values

MPIMessageOK	if *AdcConfigSet* successfully sets the configuration of the Adc object using data from the structure

See Also

mpiAdcConfigGet | MEIAdcConfig

mpiAdcFlashConfigGet

Declaration

```
long mpiAdcFlashConfigGet(MPIAdc        adc,
                          void          *flash,
                          MPIAdcConfig  *config,
                          void          *external
```

Required Header

stdmpi.h

Description

AdcFlashConfigGet gets the flash configuration of an Adc object (***adc***) and writes it into the structure pointed to by ***config***, and also writes it in the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

XMP Only

external either points to a structure of type **MEIAdcConfig{}** or is NULL. **flash** is either an MEIFlash handle or MPIHandleVOID. If **flash** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *AdcFlashConfigGet* successfully gets the Adc's flash configuration and writes it into the structure(s)

See Also

mpiAdcFlashConfigSet | MEIAdcConfig | MEIFlash

mpiAdcFlashConfigSet

Declaration

```
long mpiAdcFlashConfigSet(MPIAdc          adc,
                          void            *flash,
                          MPIAdcConfig    *config,
                          void            *external)
```

Required Header

stdmpi.h

Description

AdcFlashConfigSet sets the flash configuration of an Adc object (***adc***) using the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

XMP Only

external either points to a structure of type **MEIAdcConfig{}** or is NULL. **flash** is either an MEIFlash handle or MPIHandleVOID. If **flash** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *AdcFlashConfigSet* successfully sets the Adc's flash configuration using data from the structure(s)

See Also

MEIFlash | mpiAdcFlashConfigGet

mpiAdcInput

Declaration

```
long mpiAdcInput(MPIAdc          adc,
                 unsigned long   *input)
```

Required Header

stdmpi.h

Description

AdcInput gets the value of an Adc's (***adc***) input and writes it into the location pointed to by ***input***.

Return Values

MPIMessageOK	if *AdcInput* successfully gets the value of the Adc's input and writes it to the location

See Also

mpiAdcMemory

Declaration

```
long mpiAdcMemory(MPIAdc     adc,
                  void       **memory)
```

Required Header

stdmpi.h

Description

AdcMemory writes an address [used to access an Adc's memory (***adc***)] to the contents of ***memory***. This address, or an address calculated from it, can be passed as the src argument to mpiAdcMemoryGet(...) and the ***dst*** argument to mpiAdcMemorySet(...).

Return Values

MPIMessageOK	if *AdcMemory* successfully writes the Adc's address to the contents of ***memory***

See Also

mpiAdcMemoryGet | mpiAdcMemorySet

mpiAdcMemoryGet

Declaration

```
long mpiAdcMemoryGet(MPIAdc     adc,
                     void       *dst,
                     void       *src,
                     long       count)
```

Required Header

stdmpi.h

Description

AdcMemoryGet copies ***count*** bytes of an Adc's memory (***adc***, starting at address ***src***) and to application memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *AdcMemoryGet* successfully copies Adc memory to application memory

See Also

mpiAdcMemorySet | mpiAdcMemory

mpiAdcMemorySet

Declaration

```
long mpiAdcMemorySet(MPIAdc    adc,
                     void      *dst,
                     void      *src,
                     long      count)
```

Required Header

stdmpi.h

Description

AdcMemorySet copies ***count*** bytes of application memory (starting at address ***src***) to an Adc's memory (***adc***, starting at address ***dst***).

Return Values

MPIMessageOK	if *AdcMemorySet* successfully copies application memory to Adc memory

See Also

mpiAdcMemoryGet | mpiAdcMemory

mpiAdcControl

Declaration

```
const MPIControl mpiAdcControl(MPIAdc    adc)
```

Required Header

stdmpi.h

Description

AdcControl returns a handle to the motion controller (Control) with which an Adc (***adc***) is associated.

adc	a handle to the Adc object

Return Values

MPIHandleVOID	if **adc** is invalid
MPIControl	handle to a Control object

See Also

mpiAdcCreate | mpiControlCreate

mpiAdcMotorMapGet

Declaration

```
long mpiAdcMotorMapGet(MPIAdc          adc,
                       MPIObjectMap    *map)
```

Required Header

stdmpi.h

Description

AdcMotorMapGet gets the object map [of the Motors associated with an Adc (***adc***)] and writes it into the structure pointed to by ***map***.

Return Values

MPIMessageOK	if *AdcMotorMapGet* successfully gets the object map of the Motors associated with an Adc and writes it into the structure

See Also

mpiAdcNumber

Declaration

```
long mpiAdcNumber(MPIAdc    adc,
                  long      *number)
```

Required Header

stdmpi.h

Description

AdcNumber writes the index of an Adc object (***adc***, on the motion controller that the Adc object is associated with) to the contents of ***number***.

Return Values

MPIMessageOK	if *AdcNumber* successfully writes the Adc's index to the contents of ***number***

See Also

MPIAdcConfig / MEIAdcConfig

MPIAdcConfig

```
typedef struct MPIAdcConfig {
    double      range;  /* +/- Voltage */
} MPIAdcConfig;
```

Description

range	The voltage range that the ADC can be configured for. All values are measured in volts. Valid values are 2.5, 5.0, and 10.0

MEIAdcConfig

```
typedef struct MEIAdcConfig {
    MEIAdcMux   mux;
} MEIAdcConfig;
```

Description

mux	is used to configure what signal an MPIAdc object will look at.

See Also

MPIAdc

MPIAdcMessage

MPIAdcMessage

```
typedef enum {
                MPIAdcMessageADC_INVALID,
        } MPIAdcMessage;
```

Description

MPIAdcMessageADC_INVALID	
Meaning	The MPIAdc handle passed to an MPIAdc method is invalid.
Possible Causes	Either the handle was never initialized or the mpiAdcCreate method failed.
Recommendations	Use mpiAdcValidate after mpiAdcCreate to see if the returned handle is valid.

See Also

mpiAdcValidate | mpiAdcCreate

MEIAdcMux

MEIAdcMux

```
typedef enum {
   MEIAdcMuxDAC_CAL,
   MEIAdcMuxPLUS10V,
   MEIAdcMux0V,
   MEIAdcMuxMINUS10V,
   MEIAdcMuxANALOG_IN_0,
   MEIAdcMuxANALOG_IN_1,
   MEIAdcMuxANALOG_IN_2,
   MEIAdcMuxANALOG_IN_3,
   MEIAdcMuxANALOG_IN_4,
   MEIAdcMuxANALOG_IN_5,
   MEIAdcMuxANALOG_IN_6,
   MEIAdcMuxANALOG_IN_7,
} MEIAdcMux;
```

Description

AdcMux is an enumeration used by MEIAdcConfig.mux. It configures what signal a particular MPIAdc object will look at. The following table lists what signal each MEIAdcMux enumeration stands for:

MEIAdcMuxDAC_CAL	Used to calibrate an ADC object from a DAC object. Used by MEI for calibration and testing.
MEIAdcMuxPLUS10V	MEIAdcMuxPLUS10V +10V signal. Used by MEI for calibration and testing.
MEIAdcMux0V	0V signal. Used by MEI for calibration and testing.
MEIAdcMuxMINUS10V	-10V signal. Used by MEI for calibration and testing.
MEIAdcMuxANALOG_IN_0	Analog input 0
MEIAdcMuxANALOG_IN_1	Analog input 1
MEIAdcMuxANALOG_IN_2	Analog input 2
MEIAdcMuxANALOG_IN_3	Analog input 3
MEIAdcMuxANALOG_IN_4	Analog input 4
MEIAdcMuxANALOG_IN_5	Analog input 5
MEIAdcMuxANALOG_IN_6	Analog input 6
MEIAdcMuxANALOG_IN_7	Analog input 7

Note

MEIAdcMuxDAC_CAL	These are all specific to XMP Revision 1 boards. The ADC input lines will be floating if these values are used to configure an MPIAdc object.
MEIAdcMuxPLUS10V	
MEIAdcMux0V	
MEIAdcMuxMINUS10V	

See Also

MEIAdcConfig

Axis Objects

Introduction

An **Axis** object manages a single physical axis on a motion controller. It represents a reference line in a coordinate system. The controller calculates an axis's command position every sample based on the motion commanded by the Motion Supervisor. The Axis object contains command, actual, and error position data, plus status.

An Axis can have one or more Filters associated with it and each Filter can have one or more Motors associated with it. The Filter and Motor objects ensure the Axis command path is followed and that the control signals get to the correct motor. Complex mechanical systems with two (or more) motors can be mapped to a single axis of motion, abstracting the details of the physical hardware and making motion software much easier to develop.

For simple systems, there is a one to one relationship between the Axis, Filter and Motor objects.

Methods

Create, Delete, Validate Methods

mpiAxis**Create**	Create Axis object
mpiAxis**Delete**	Delete Axis object
mpiAxis**Validate**	Validate Axis object

Configuration and Information Methods

mpiAxis**ActualPositionGet**	Get actual position
mpiAxis**ActualPositionSet**	Set actual position
mpiAxis**ActualVelocity**	Set actual velocity
mpiAxis**ConfigGet**	Get Axis configuration
mpiAxis**ConfigSet**	Set Axis configuration
mpiAxis**CommandPositionGet**	Get command position
mpiAxis**CommandPositionSet**	Set command position
mpiAxis**FlashConfigGet**	Get Axis flash config
mpiAxis**FlashConfigSet**	Set Axis flash config
mpiAxis**OriginGet**	Get Axis origin
mpiAxis**OriginSet**	Set Axis origin
mpiAxis**PositionError**	Get position error of an Axis
mpiAxis**Status**	Get Axis status
mpiAxis**Trajectory**	Get Axis trajectory

Event Methods

mpiAxis**EventNotifyGet**	Get event mask
mpiAxis**EventNotifySet**	Set event mask
mpiAxis**EventReset**	

Memory Methods

mpiAxis**Memory**	Set Axis memory address
mpiAxis**MemoryGet**	Copy bytes of Axis memory to application memory
mpiAxis**MemorySet**	Copy bytes of application memory to Axis memory

Relational Methods

mpiAxis**Control**	Return handle of Control associated with Axis
mpiAxis**FilterMapGet**	Get object map of Filters
mpiAxis**FilterMapSet**	Set object map of Filters
mpiAxis**MotorMapGet**	Get object map of Motors
mpiAxis**Number**	Get index of Axis

Data Types

MPIAxis**Config** / MEIAxis**Config**

MPIAxis**InPosition**

MPIAxis**Message**

Copyright © 2002
Motion Engineering

mpiAxisCreate

Declaration

```
const MPIAxis mpiAxisCreate(MPIControl  control,
                            long       *number)
```

Required Header

stdmpi.h

Description

AxisCreate creates an axis object associated with the axis identified by ***number*** located on motion controller ***control***. AxisCreate is the equivalent of a C++ constructor.

control	a handle to Axis object.
number	the number specifies which Axis object is being created. The number corresponds to an Axis object in XMP memory.

Remarks

An **Axis** represents a physical axis in space such as X, Y, Z, Theta, or other axes. An Axis may be comprised of one or more motors, such as with a gantry system.

Return Values

handle	to an Axis object
MPIHandleVOID	if the object could not be created

See Also

mpiAxisDelete | mpiAxisValidate

mpiAxisDelete

Declaration

```
long mpiAxisDelete(MPIAxis  axis)
```

Required Header

stdmpi.h

Description

AxisDelete deletes an Axis object and invalidates its handle (**axis**).
AxisDelete is the equivalent of a C++ destructor.

axis	the Axis handle to be deleted

Remarks

All objects that are created in an application should be deleted in reverse order at the end of the code.

Return Values

MPIMessageOK	if *AxisDelete* successfully deletes an Axis object and invalidates its handle

See Also

mpiAxisCreate | mpiAxisValidate

mpiAxisValidate

Declaration

```
long mpiAxisValidate(MPIAxis   axis)
```

Required Header

stdmpi.h

Description

AxisValidate validates the Axis object and its handle (***axis***). AxisValidate should be called immediately after an object is created.

axis	a handle to the Axis object to be validated

Return Values

MPIMessageOK	if Axis is a handle to a valid object.

See Also

mpiAxisCreate | mpiAxisDelete

mpiAxisActualPositionGet

Declaration

```
long mpiAxisActualPositionGet(MPIAxis  axis,
                              double  *actual)
```

Required Header

stdmpi.h

Description

axis	a handle to an Axis object
***actual**	a pointer to the Axis actual position returned by the method.

Return Values

MPIMessageOK	if *AxisActualPositionGet* successfully gets and writes the value of the actual postion of ***axis*** to the location.

See Also

mpiAxisActualPositionSet | Using the Origin Variable

mpiAxisActualPositionSet

Declaration

```
long mpiAxisActualPositionSet(MPIAxis  axis,
                              double   actual)
```

Required Header

stdmpi.h

Description

AxisActualPositionSet sets the value of the actual position of an Axis (***axis***) to ***actual***.

axis	a handle to the Axis object
actual	value to which the Axis actual position will be set

Return Values

MPIMessageOK	if *AxisActualPositionSet* successfully sets the value of the actual postion of an Axis to ***actual***

See Also

AxisCommandPositionSet | Using the Origin Variable

mpiAxisActualVelocity

Declaration

```
long mpiAxisActualVelocity(MPIAxis    axis,
                           double    *actual)
```

Required Header

stdmpi.h

Description

AxisActualVelocity reads the value of the actual velocity on an Axis (***axis***) and writes it in the location pointed to by ***actual***.

Return Values

MPIMessageOK	if *AxisActualVelocity* succesfully gets and writes the value of the actual velocity of ***axis*** to the location.

See Also

mpiAxisConfigGet

Declaration

```
long mpiAxisConfigGet(MPIAxis       axis,
                      MPIAxisConfig *config,
                      void          *external)
```

Required Header

stdmpi.h

Description

AxisConfigGet gets the configuration of an Axis (**axis**) and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

axis	a handle to the Axis object
***config**	pointer to the MPIAxisConfig structure
***external**	pointer to an external. See remarks below

Remarks

For XMP controllers, ***external*** either points to a structure of type **MEIAxisConfig{}** or is NULL.

Return Values

MPIMessageOK	if *AxisConfigGet* successfully gets the Axis configuration and writes it into the structure(s)

See Also

MPIAxisConfig | mpiAxisConfigSet | MEIAxisConfig

mpiAxisConfigSet

Declaration

```
long mpiAxisConfigSet(MPIAxis       axis,
                      MPIAxisConfig *config,
                      void          *external)
```

Required Header

stdmpi.h

Description

AxisConfigSet sets the configuration of an Axis (***axis***) using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

axis	a handle to the Axis object
***config**	pointer to an MPIAxisConfig structure.
***external**	pointer to an external. See remarks below.

XMP Only ***external*** either points to a structure of type **MEIAxisConfig{}** or is NULL.

Return Values

MPIMessageOK	if *AxisConfigSet* successfully sets the Axis configuration

See Also

mpiAxisConfigGet | MEIAdcConfig

mpiAxisCommandPositionGet

Declaration

```
long mpiAxisCommandPositionGet(MPIAxis axis,
                               double  *command)
```

Required Header

stdmpi.h

Description

AxisCommandPositionGet gets the value of the command position of an Axis (**axis**) and puts it in the location pointed to by ***command***.

axis	a handle to the Axis object
***command**	a pointer to the Axis command position returned by the method

Return Values

MPIMessageOK	if *AxisCommandPositionGet* successfully gets the value of the command position of ***axis*** and puts it in the location

See Also

mpiAxisCommandPositionSet

mpiAxisCommandPositionSet

Declaration

```
long mpiAxisCommandPositionSet(MPIAxis  axis,
                               double   command)
```

Required Header

stdmpi.h

Description

AxisCommandPositionSet sets the value of the command position of an Axis (**axis**) from ***command***.

axis	a handle to the Axis object
command	value to which the Actual command position will be set

Remarks

Setting the Axis Command Position may cause the axis to jump. See the discussion of the Axis Origin before using the AxisActualPositionSet and AxisCommandPositionSet methods.

Return Values

MPIMessageOK	if *AxisCommandPositionSet* successfully sets the value of the command position of ***axis*** from ***command***

See Also

mpiAxisFlashConfigGet

Declaration

```
long mpiAxisFlashConfigGet(MPIAxis       axis,
                           void          *flash,
                           MPIAxisConfig *config,
                           void          *external)
```

Required Header

stdmpi.h

Description

AxisFlashConfigGet gets the flash configuration for an Axis (***axis***) and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***sexternal*** (if ***external*** is not NULL).

The Axis flash configuration information in ***external*** is in addition to the Axis flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in external is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

axis	a handle to the Axis object
***flash**	
***config**	pointer to an MPIAxisConfig structure
***external**	pointer to an external. See remarks below.

Remarks

For XMP controllers, ***external*** either points to a structure of type **MEIAxisConfig{}** or is NULL. ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *AxisEventNotifySet* successfully requests host notification of the event(s) that are specified by ***eventMask*** and generated by ***motion***

See Also

MEIFlash | mpiAxisFlashConfigSet |MEIAxisConfig

mpiAxisFlashConfigSet

Declaration

```
long mpiAxisFlashConfigSet(MPIAxis          axis,
                           void             *flash,
                           MPIAxisConfig    *config,
                           void             *external)
```

Required Header

stdmpi.h

Description

AxisFlashConfigSet sets the flash configuration for for an Axis (***axis***) using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Axis flash configuration information in ***external*** is *in addition* to the Axis flash configuration information in ***config***, i.e., the flash configuration information in config and in external is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

axis	a handle to the Axis object
***flash**	
***config**	pointer to an MPIAxisConfig structure
***external**	pointer to an external. See remarks below.

XMP Only

external either points to a structure of type **MEIAxisConfig{}** or is NULL. ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *AxisFlashConfigSet* successfully sets the Axis flash configuration using data from the structure(s)

See Also

MEIFlash | mpiAxisFlashConfigGet | MEIAxisConfig

mpiAxisOriginGet

Declaration

```
long mpiAxisOriginGet(MPIAxis  axis,
                      double   *origin)
```

Required Header

stdmpi.h

Description

AxisOriginGet gets the value of the origin of an Axis (***axis***) and writes it into the location pointed to by ***origin***.

axis	a handle to the Axis object.
***origin**	pointer to the Origin value returned by the method

Return Values

MPIMessageOK	if *AxisOriginGet* successfully gets the value of the origin of the Axis and writes it to the location

See Also

mpiAxisOriginSet | Using the Origin Variable

mpiAxisOriginSet

Declaration

```
long mpiAxisOriginSet(MPIAxis  axis,
                      double   origin)
```

Required Header

stdmpi.h

Description

AxisOriginSet sets the value of the origin of an Axis (***axis***) to ***origin***.

axis	a handle to the Axis object
origin	Value to which the Axis Origin will be set

Return Values

MPIMessageOK	if *AxisOriginSet* successfully sets the origin of an Axis to ***origin***

See Also

mpiAxisOriginGet | Using the Origin Variable

mpiAxisPositionError

Declaration

```
long mpiAxisPositionError(MPIAxis  axis,
                          double  *error)
```

Required Header

stdmpi.h

Description

AxisPositionError gets the value of the position error of an Axis (***axis***) and puts it in the location pointed to by ***error***. The position error is equal to (command position - actual position).

axis	a handle to the Axis object
***error**	a pointer to the Axis position error returned by the method

Return Values

MPIMessageOK	if *AxisPositionError* successfully gets and writes the value of the position error into ****error***

See Also

mpiAxisCommandPositionGet | mpiAxisActualPositionGet

mpiAxisStatus

Declaration

```
long mpiAxisStatus(MPIAxis    axis,
                   MPIStatus  *status,
                   void       *external)
```

Required Header

stdmpi.h

Description

AxisStatus gets the status of an Axis (axis) and writes it into the structure pointed to by ***status***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Axis flash configuration information in ***external*** is in addition to the Axis flash configuration information in ***config***, i.e., the flash configuration information in config and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

axis	a handle to the Axis object
***status**	pointer to MPIStatus structure
***external**	pointer to an external.

XMP Only ***external*** either points to a structure of type MEIStatus{} or is NULL.

Return Values

MPIMessageOK	if *AxisStatus* successfully gets the Axis status and writes it into the structure(s)

See Also

MEIAxisConfig

mpiAxisTrajectory

Declaration

```
long mpiAxisTrajectory(MPIAxis       axis,
                       MPITrajectory *trajectory)
```

Required Header

stdmpi.h

Description

AxisTrajectory reads the default trajectory of ***axis*** and writes it into the structure pointed to by ***trajectory***.

axis	a handle to the Axis object.
***trajectory**	pointer to the MPITrajectory structure

Remarks

The default MPITrajectory structure can be used by the mpiMotionStart(...) and mpiMotionModify() methods.

Return Values

MPIMessageOK	if *AxisTrajectory* successfully gets the Axis trajectory and writes it into the structure

See Also

mpiMotionStart | mpiMotionModify | MPITrajectory

mpiAxisEventNotifyGet

Declaration

```
long mpiAxisEventNotifyGet(MPIAxis        axis,
                           MPIEventMask   *eventMask,
                           void           *external)
```

Required Header

stdmpi.h

Description

AxisEventNotifyGet writes the event mask (that specifies the event type(s) for which host notification has been requested) to the location pointed to by ***eventMask***, and also writes it into the implementation-specific location pointed to by ***external*** (if ***external*** is not NULL).

The event notification information in ***external*** is in addition to the event notification information in ***eventMask***, i.e, the event notification information in ***eventMask*** and in ***external*** is not the same information. Note that ***eventMask*** or ***external*** can be NULL (but not both NULL).

axis	a handle to the Axis object
***eventMask**	pointer to an MPIEventMask
***external**	pointer to an external. See remarks below

XMP Only

external either points to a structure of type **MEIEventNotifyData{}** or is NULL.

The **MEIEventNotifyData{}** structure is an array of firmware addresses, whose contents are placed into the **MEIEventStatusInfo{}** structure (of all events generated by this object).

Return Values

MPIMessageOK	if *AxisEventNotifyGet* successfully writes the event mask to the location(s)

See Also

MEIEventNotifyData | MEIEventStatusInfo | mpiAxisEventNotifySet

mpiAxisEventNotifySet

Declaration

```
long mpiAxisEventNotifySet(MPIAxis        axis,
                           MPIEventMask   eventMask,
                           void           *external)
```

Required Header

stdmpi.h

Description

AxisEventNotifySet requests host notification of the event(s) that are generated by ***axis*** and specified by ***eventMask***, and also specified by the implementation-specific location pointed to by ***external*** (if ***external*** is not NULL).

The event notification information in ***external*** is in addition to the event notification information in ***eventMask***, i.e, the event notification information in ***eventMask*** and in ***external*** is not the same information. Note that ***eventMask*** or ***external*** can be NULL (but not both NULL).

axis	a handle to the Axis object
eventMask	pointer to an MPIEventMask
***external**	pointer to an external

XMP Only

external either points to a structure of type **MEIEventNotifyData{}** or is NULL.

The **MEIEventNotifyData{}** structure is an array of firmware addresses, whose contents are placed into the **MEIEventStatusInfo{}** structure (of all events generated by this object).

To...	Then...
enable host notification of all events	configure ***eventmask*** with mpiEventMaskALL(eventMask)
disable host notification of all events	configure ***eventmask*** with mpiEventMaskCLEAR(eventMask)

Return Values

MPIMessageOK	if *AxisEventNotifySet* successfully requests host notification of the event(s) that are specified by ***eventMask*** and generated by ***motion***

See Also

MEIEventNotifyData | MEIEventStatusInfo | MPIEventMask | MPIEventType | mpiEventMaskALL | mpiEventMaskCLEAR | mpiAxisEventNotifyGet | MEIEventNotifyData

mpiAxisEventReset

Declaration

```
long mpiAxisEventReset(MPIAxis      axis,
                       MPIEventMask eventMask)
```

Required Header

stdmpi.h

Description

To...	Then...
enable host notification of all events	configure ***eventmask*** with mpiEventMaskALL(eventMask)
disable host notification of all events	configure ***eventmask*** with mpiEventMaskCLEAR(eventMask)

Return Values

MPIMessageOK	if *AxisEventNotifySet* successfully requests host notification of the event(s) that are specified by ***eventMask*** and generated by ***motion***

Remarks

Event notification is enabled for event types specified in ***eventMask***, a bit mask generated by the logical OR of the MPIEventMask bits associated with the desired MPIEventType values. Configuration of the eventMask should be done with Event macros. Event notification is disabled for event types that are not specified in ***eventMask***.

The mask of event types generated by a Motion object consists of bits from MPIEventMaskMOTION and MPIEventMaskAXIS.

See Also

MEIEventNotifyData | MEIEventStatusInfo | MPIEventType | mpiEventMaskALL
mpiEventMaskCLEAR | MPIEventMaskMOTION | MPIEventMaskAXIS | MPIEventMask

mpiAxisMemory

Declaration

```
long mpiAxisMemory(MPIAxis axis,
                   void    **memory)
```

Required Header

stdmpi.h

Description

AxisMemory writes an address (that is used to access Axis memory) to the contents of ***memory***. This address (or an address calculated from it) is passed as the ***src*** argument to mpiAxisMemoryGet(...) and as the ***dst*** argument to mpiAxisMemorySet(...).

axis	a handle to the Axis object

Return Values

MPIMessageOK	if *AxisMemory* successfully writes the Axis memory address to the contents of ***memory***

See Also

mpiAxisMemoryGet | mpiAxisMemorySet

mpiAxisMemoryGet

Declaration

```
long mpiAxisMemoryGet(MPIAxis axis,
                      void    *dst,
                      void    *src,
                      long    count)
```

Required Header

stdmpi.h

Description

AxisMemoryGet copies ***count*** bytes of Axis (***axis***) memory (starting at address ***src***) to application memory (starting at address ***dst***).

axis	a handle to the Axis object
***dst**	pointer to the destination location to where the memory will be written
***src**	pointer to the source location of memory being read
count	size of memory to be read

Return Values

MPIMessageOK	if *AxisMemory* successfully writes the Axis memory address to the contents of memory

See Also

mpiAxisMemory | mpiAxisMemorySet

mpiAxisMemorySet

Declaration

```
long mpiAxisMemorySet(MPIAxis axis,
                      void    *dst,
                      void    *src,
                      long    count)
```

Required Header

stdmpi.h

Description

AxisMemorySet copies ***count*** bytes of application memory (starting at address ***src***) to Axis (***axis***) memory (starting at address ***dst***).

axis	a handle to the Axis object
***dst**	pointer to the destination location to where the memory will be written
***src**	pointer to the source location of memory being read
***count**	size of memory to be written

Return Values

MPIMessageOK	if *AxisMemorySet* successfully copies ***count*** bytes of application memory to Axis memory

See Also

mpiAxisMemory | mpiAxisMemoryGet

mpiAxisControl

Declaration

```
const MPIControl mpiAxisControl(MPIAxis  axis)
```

Required Header

stdmpi.h

Description

AxisControl returns a handle to the motion controller (Control) with which an Axis (***axis***) is associated.

axis	a handle to the Axis object

Return Values

MPIHandleVOID	if **axis** is invalid

See Also

mpiAxisFilterMapGet

Declaration

```
long mpiAxisFilterMapGet(MPIAxis      axis,
                         MPIObjectMap *map)
```

Required Header

stdmpi.h

Description

AxisFilterMapGet gets the object map of the Filters [associated with an Axis (***axis***)] and writes it into the structure pointed to by ***map***.

axis	a handle to the Axis object
***map**	a pointer to an ObjectMap of Filters mapped to the axis

Remarks

MPIObjectMap is a *long* that maps the Filters in controller memory to each bit. E.g. A map value of 1 would indicate Filter 0 is mapped the Axis. A value of 6 would indicate that Filters 2 and 3 are mapped to the Axis.

Return Values

MPIMessageOK	if *AxisFilterMapGet* successfully gets and writes the object map to the structure

See Also

mpiAxisFilterMapSet

mpiAxisFilterMapSet

Declaration

```
long mpiAxisFilterMapSet(MPIAxis      axis,
                         MPIObjectMap map)
```

Required Header

stdmpi.h

Description

AxisFilterMapSet sets the Filters [associated with an Axis (***axis***)] using data from the object map specified by ***map***.

axis	a handle to the Axis object
map	a list of Filters to be mapped to the axis

Remarks

MPIObjectMap is a *long* that maps the Filters in controller memory to each bit. E.g. A map value of 1 will map Filter 0 to the Axis. A value of 6 will map both Filters 2 and 3 to the Axis.

Return Values

MPIMessageOK	if *AxisFilterMapSet* successfully sets the Filters using data from the object map

See Also

mpiAxisFilterMapGet | MPIObjectMap

mpiAxisMotorMapGet

Declaration

```
long mpiAxisMotorMapGet(MPIAxis       axis,
                        MPIObjectMap *map)
```

Required Header

stdmpi.h

Description

AxisMotorMapGet gets the object map [of the Motors associated with an Axis (***axis***)] and writes it into the structure pointed to by ***map***.

axis	a handle to the Axis object.
***map**	a pointer to an ObjectMap of Motors mapped to the axis

Remarks

MPIObjectMap is a *long* that maps the Motors in controller memory to each bit. E.g. A **map** value of 1 would indicate Motor 0 is mapped the Axis. A value of 6 would indicate that Motors 2 and 3 are mapped to the Axis.

Remember that Motors are mapped to Axes through the Filter object. To configure the Axis/Motor map, the application will need to set the AxisFilterMap and FilterMotorMap.

Return Values

MPIMessageOK	if *AxisMotorMapGet* successfully gets the object map and writes it into the structure

See Also

mpiAxisNumber

Declaration

```
long mpiAxisNumber(MPIAxis  axis,
                   long     *number)
```

Required Header

stdmpi.h

Description

AxisNumber writes the index of an Axis (***axis***, on the motion controller that the Axis is associated with) to the contents of ***number***.

axis	a handle to the Axis object
***number**	pointer to the number

Return Values

MPIMessageOK	if *AxisNumber* successfully writes the index of Axis to the contents of ***number***

See Also

MPIAxisConfig / MEIAxisConfig

MPIAxisConfig

```
typedef struct MPIAxisConfig {
    MPIAxisInPosition   inPosition;
    MPIObjectMap        filterMap;
}MPIAxisConfig;
```

Description

inPosition	See MPIAxisInPosition.
filterMap	bitmap indicating which Filter objects are mapped to the Axis. See MPIObject for more details.

MEIAxisConfig

```
typedef struct MEIAxisConfig {
    MEIXmpAPosInput     APos [MEIXmpAxisAPosInputs];
    MEIXmpAxisFilter    Filter;
    MEIXmpAxisGear      Gear;
}MEIAxisConfig;
```

Description

APos - an array of structures that set Actual position inputs. The structure has two elements:

- **Ptr** - Pointer to Actual position input register. Default value is corresponding encoder input.
- **Coeff** - Value of Actual position input register

Filter

- Input
- Output
- Delta
- Delay
- Timer
- Pointer

Gear - Coefficients for gearing off a position input

- Ptr - pointer to gear master
- Ratio.A - numerator of multiplier
- Ratio.B - denominator of multiplier
- Ratio.Old -
- Ratio.Remainder -
- Position - final geared position

See Also mpiAxisConfigGet | mpiAxisConfigSet | MPIAxisInPosition | MPIObject

MPIAxisInPosition

MPIAxisInPosition

```
typedef struct MPIAxisInPosition {
    struct {
        float   positionFine;
        long    positionCoarse;
        float   velocity;
    } tolerance;
        float   settlingTime;   /* seconds */
        long    settleOnStop;
        long    settleOnEstop;
} MPIAxisInPosition;
```

Description

tolerance	Includes the following 3 elements that determine settling criteria for an axis.
positionFine	Value, in counts, from the move target position at which the DSP sets the IN_FINE_POSITION flag. This parameter is used as part of the Axis settling criteria to determine Motion Done.
positionCoarse	Value, in counts, from a move target position at which the DSP sets the IN_COARSE_POSITION flag.
velocity	Value, in counts/second, from the final move velocity at which the DSP set the AT_VELOCITY flag. Only used with Velocity type moves.
settlingTime	Duration that PositionFine, PositionCoarse, or Velocity must be TRUE before flag is set.
settleOnStop	If TRUE, DSP will use settling criteria when Stop Event occurs.
settleOnEstop	If TRUE, DSP will use settling criteria when E-Stop Event occurs.

See Also

Special Note on Configuration of IN_POSITION and Done Events after STOP or E_STOP Events

MPIAxisMessage

MPIAxisMessage

```
typedef enum {
    MPIAxisMessageAXIS_INVALID,
 MPIAxisMessageCOMMAND_NOT_SET,
}MPIAxisMessage;
```

Description

AxisMessage is an enumeration of Axis error messages that can be returned by the MPI library.

MPIAxisMessageAXIS_INVALID	Indicates Axis handle is invalid.

See Also

Configuration of IN_POSITION and DONE Events after STOP or E-STOP Events

Two fields, **settleOnStop** and **settleOnEstop** are incorporated into the MPIAxisInPosition{} structure. These fields control the generation and use of IN_FINE_POSITION, and DONE status bits and events. A value of FALSE in these fields causes the IN_FINE_POSITION to be held false after STOP (or E-STOP) events and DONE to be based solely on command velocity (i.e. DONE is true as soon as the command velocity reaches 0). A value of TRUE in these fields causes IN_FINE_POSITION and DONE to be calculated in the same manner as that for normal motion, except that the position where the command velocity reaches zero is used for a target rather than the original Target Position.

The following table shows the generation of these status bits with settleOnStop (settleOnEstop) = FALSE (the default value):

Motion Status	**After S-curve or Trapezoidal Move**	**During Velocity Move**	**After STOP (E-STOP)**	**After ABORT**
IN_FINE_POSITION	Based on target distance (see note 3)	FALSE	FALSE	FALSE
IN_COARSE_POSITION	Based on target distance	FALSE	FALSE	FALSE
AT_TARGET	TRUE when command = target	FALSE	FALSE	FALSE
DONE	TRUE if both TC and IN_FINE_POSITION are true	FALSE	TRUE when command velocity = 0	TRUE

The following table shows the generation of these status bits with settleOnStop (settleOnEstop) = TRUE:

Motion Status	**After S-curve or Trapezoidal Move**	**During Velocity Move**	**After STOP (E-STOP)**	**After ABORT**
IN_FINE_POSITION	Based on target distance (see note 3)	FALSE	Based on position error (see note 1)	FALSE
IN_COARSE_POSITION	Based on target distance	FALSE	FALSE	FALSE
AT_TARGET	TRUE when command = target	FALSE	FALSE	FALSE
DONE	TRUE if both TC and IN_FINE_POSITION are true	FALSE	Same as IN_FINE_ POSITION	TRUE

NOTE 1: IN_FINE_POSITION is based on four criteria:

- The trajectory has completed (see note 2).
- |command position – actual position| < fine position tolerance.
- Target velocity – actual velocity| < velocity tolerance (the default setting for velocity tolerance so large that this criteria is ignored).
- The above 3 criteria have been satisfied for the duration specified by the settling time parameter.

NOTE 2: The reference to “TC” above refers to TRAJECTORY_COMPLETE, an internal status that is set when all of the current motion segments (frames) have completed.

NOTE 3: The criteria used for calculation of IN_FINE_POSITION after s-curve or trapezoidal motion has changed to the following: (This is the same as the MPI-1 criteria.)

- The trajectory has completed (see note 2).
- |target position – actual position| < fine position tolerance.
- |command velocity – actual velocity| < velocity tolerance (the default setting for velocity tolerance so large that this criteria is ignored).
- The above 3 criteria have been satisfied for a duration specified by the settling time parameter.

Return to MPIAxisInPosition

Copyright © 2002
Motion Engineering

Capture Object

Introduction

A **Capture** object manages a single capture in an XMP motion controller. A capture is a hardware latch of a motor position triggered by a motor input. The XMP controller supports ten (10) Capture objects per motion block.

The default mapping for Capture objects on the first motion block is as follows:

Capture Number	Encoder
0	0 Motor (0)
1	Auxiliary
2	1 Motor (1)
3	Auxiliary
4	2 Motor (2)
5	Auxiliary
6	3 Motor (3)
7	Auxiliary
8	Auxiliary
9	Auxiliary

Methods

Create, Delete, Validate Methods

mpiCapture**Create**	Create Capture object
mpiCapture**Delete**	Delete Capture object
mpiCapture**Validate**	Validate Capture object

Configuration and Information Methods

mpiCapture**ConfigGet**	Get Capture configuration
mpiCapture**ConfigSet**	Set Capture configuration
mpiCapture**FlashConfigGet**	Get flash configuration for Capture
mpiCapture**FlashConfigSet**	Set flash configuration for Capture
mpiCapture**Status**	Get status of Capture

Action Methods

mpiCapture**Arm**	Arm capture object

Memory Methods

mpiCapture**Memory**	Set address to Capture memory
mpiCapture**MemoryGet**	Copy Capture memory to application memory

mpiCapture**MemorySet**	Copy application memory to Capture memory

Relational Methods

mpiCapture**Control**	Return handle of Control object associated with Adc
mpiCapture**Number**	Get index of Capture (for Control list)

Data Types

MPICapture**Config** / MEICapture**Config**
MPICapture**Latch**
MPICapture**Message**
MPICapture**SIMConfig**
MPICapture**State**
MPICapture**Status**

Constants

MPICapture**LatchCountMAX**

Copyright © 2002
Motion Engineering

mpiCaptureCreate

Declaration

```
const MPICapture mpiCaptureCreate(MPIControl control,
                                  long       number)
```

Required Header stdmpi.h

Description

CaptureCreate creates a Capture object associated with a number (***number***), that is located on a motion controller (***control***).

CaptureCreate is the equivalent of a C++ constructor. Each motion block supports 10 capture registers. The default configuration is two capture registers per motor, while the last two (8,9) on each motion block are reserved for the Auxiliary Encoder (not supported). The capture registers are default mapped as follows: 0 and 1 for Motor0; 2 and 3 for Motor1; 10 and 11 for Motor4; etc. The first Capture for each motor uses the default (primary) encoder input for position capture. The second uses the AUX encoder input.

Return Values

handle	to a Capture object
MPIHandleVOID	if the object could not be created

See Also mpiCaptureDelete | mpiCaptureValidate

mpiCaptureDelete

Declaration `long mpiCaptureDelete(MPICapture capture)`

Required Header stdmpi.h

Description

CaptureDelete deletes a Capture object and invalidates its handle (***capture***). *CaptureDelete* is the equivalent of a C++ destructor.

Return Values	
MPIMessageOK	if *CaptureDelete* successfully deletes the Capture object and invalidates its handle

See Also mpiCaptureCreate | mpiCaptureValidate

mpiCaptureValidate

Declaration `long mpiCaptureValidate(MPICapture capture)`

Required Header stdmpi.h

Description **CaptureValidate** validates the Capture object and its handle (***capture***).

Return Values	
MPIMessageOK	if Capture is a handle to a valid object.

See Also mpiCaptureCreate | mpiCaptureDelete

mpiCaptureConfigGet

Declaration

```
long mpiCaptureConfigGet(MPICapture       capture,
                         MPICaptureConfig *config,
                         void             *external)
```

Required Header

stdmpi.h

Description

CaptureConfigGet gets a Capture object's (***capture***) configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The a Capture object's configuration information in ***external*** is *in addition* to the Capture object's configuration information in ***config***, i.e, the Capture object's configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEICaptureConfig{}** or is NULL.

Return Values

MPIMessageOK	if CaptureConfigGet successfully writes the Capture object's configuration to the structure(s)

See Also

mpiCaptureConfigSet | MEICaptureConfig

mpiCaptureConfigSet

Declaration

```
long mpiCaptureConfigSet(MPICapture        capture,
                         MPICaptureConfig  *config,
                         void              *external)
```

Required Header

stdmpi.h

Description

CaptureConfigSet sets a Capture object's (***capture***) configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Capture object's configuration information in ***external*** is *in addition* to the Capture object's configuration information in ***config***, i.e, the Capture object's configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type **MEICaptureConfig{}** or is NULL.

Return Values

MPIMessageOK	if *CaptureConfigSet* successfully sets the Capture object's configuration using data from the structure(s)

See Also

mpiCaptureConfigGet | MEICaptureConfig

mpiCaptureFlashConfigGet

Declaration

```
long mpiCaptureFlashConfigGet(MPICapture          capture,
                              void                *flash,
                              MPICaptureConfig    *config,
                              void                *external)
```

Required Header

stdmpi.h

Description

CaptureFlashConfigGet gets a Capture object's (***capture***) flash configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Capture object's flash configuration information in ***external*** is in addition to the Capture object's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type **MEICaptureConfig{}** or is NULL.

Return Values

MPIMessageOK	if *CaptureFlashConfigGet* successfully writes the Capture object's flash configuration to the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiCaptureFlashConfigSet | MEICaptureConfig

mpiCaptureFlashConfigSet

Declaration

```
long mpiCaptureFlashConfigSet(MPICapture        capture,
                              void              *flash,
                              MPICaptureConfig  *config,
                              void              *external)
```

Required Header

stdmpi.h

Description

CaptureFlashConfigSet sets a Capture object's (***capture***) flash configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Capture object's flash configuration information in ***external*** is in addition to the Capture object's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type **MEICaptureConfig{}** or is NULL.

Return Values

MPIMessageOK	if *CaptureFlashConfigSet* successfully sets the Capture object's flash configuration using data from the structure(s) flash is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiCaptureFlashConfigGet | MEICaptureConfig

mpiCaptureStatus

Declaration

```
long mpiCaptureStatus(MPICapture          capture,
                      MPICaptureStatus    *status,
                      void                *external)
```

Required Header

stdmpi.h

Description

CaptureStatus writes a Capture object's (***capture***) status into the structure pointed to by ***status***, and also into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Capture object's status information in ***external*** is in addition to the Capture object's ***status*** information in status, i.e, the ***status*** configuration information in status and in ***external*** is not the same information. Note that ***status*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type **MEICaptureStatus{}** or is NULL.

Return Values

MPIMessageOK	if *CaptureStatus* successfully writes the status of a Capture object to the structure(s)

See Also

mpiCaptureArm

Declaration

```
long mpiCaptureArm(MPICapture  capture,
                   long        arm)
```

Required Header

stdmpi.h

Description

CaptureArm arms or disarms ***capture***.

Value of "arm"	*Action of mpiCaptureArm*
FALSE	Disarms ***capture*** and sets the state of ***capture*** to MPICaptureStateIDLE
TRUE	Arms ***capture*** and sets the state of ***capture*** to MPICaptureStateARMED

Return Values

MPIMessageOK	if the Capture object is successfully armed or disarmed

See Also

MPICaptureState

mpiCaptureMemory

Declaration

```
long mpiCaptureMemory(MPICapture  capture,
                      void        **memory)
```

Required Header

stdmpi.h

Description

CaptureMemory writes an address [which is used to access a Capture object's (***capture***) memory] to the contents of ***memory***. This address, or an address calculated from it, can be passed as the src parameter to mpiCaptureMemoryGet(...) and as the ***dst*** parameter to mpiCaptureMemorySet(...).

Return Values	
MPIMessageOK	if *CaptureMemory* successfully writes the Capture object's memory address to the contents of ***memory***

See Also

mpiCaptureMemoryGet | mpiCaptureMemorySet

mpiCaptureMemoryGet

Declaration

```
long mpiCaptureMemoryGet(MPICapture  capture,
                         void        *dst,
                         void        *src,
                         long        count)
```

Required Header stdmpi.h

Description **CaptureMemoryGet** copies ***count*** bytes of a Capture object's (***capture***) memory (starting at address ***src***) and writes them into application memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *CaptureMemoryGet* successfully copies data from Capture memory to application memory

See Also mpiCaptureMemory | mpiCaptureMemorySet

mpiCaptureMemorySet

Declaration

```
long mpiCaptureMemorySet(MPICapture capture,
                         void       *dst,
                         void       *src,
                         long       count)
```

Required Header stdmpi.h

Description **CaptureMemorySet** copies count bytes of application memory (starting at address ***src***) and writes them into a Capture object's (***capture***) memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *CaptureMemorySet* successfully copies count bytes of application memory to Capture memory

See Also mpiCaptureMemory | mpiCaptureMemoryGet

mpiCaptureControl

Declaration

```
const MPIControl mpiCaptureControl(MPICapture  capture)
```

Required Header

stdmpi.h

Description

CaptureControl returns a handle to the motion controller (Control object) that a Capture object (***capture***) is associated with.

Return Values	
handle	to a Control object that a Capture object is associated with
MPIHandleVOID	if the Capture object is invalid

See Also

mpiCaptureNumber

Declaration

```
long mpiCaptureNumber(MPICapture capture,
                      long       *number)
```

Required Header

stdmpi.h

Description

CaptureNumber writes the index of a Capture object (***capture***, on the motion controller that ***capture*** is associated with) to the contents of ***number***.

Return Values

MPIMessageOK	if *CaptureNumber* successfully writes the index of a Capture object to the contents of **number**

See Also

MPICaptureConfig / MEICaptureConfig

MPICaptureConfig

```
typedef struct MPICaptureConfig {
    MPIIoTrigger     trigger;
    long             latchCount;
    MPICaptureLatch  latch[MPICaptureLatchCountMAX];
} MPICaptureConfig;
```

Description

trigger	Type, source, mask, and pattern are used to select the capture bit and input state upon which to trigger a capture. For more information about setting the trigger please see MPIIoTrigger.
latchCount	The number of valid latches in latch[] array.
latch	Array containing capture latch register number to motor number mapping for each capture object. Latch registers are numbered sequentially across all Capture objects. This mapping is currently fixed. Performing a CaptureConfigSet will return the error message, MPIMessagePARAM_INVALID if modified by the application program.

MEICaptureConfig

```
typedef struct MEICaptureConfig {
    MEICaptureSIMConfig  SIM;
} MEICaptureConfig;
```

Description

SIM	Structure that contains the Sinusoidal Interpolation Module (SIM) capture configuration. See the SIM4 hardware application note 206 for more information.

See Also

MPIIoTrigger

MPICaptureLatch

MPICaptureLatch

```
typedef struct MPICaptureLatch {
    long        number;
    long        motorNumber;
} MPICaptureLatch;
```

Description

CaptureLatch is an enumeration that is explained in MPICatpureConfig.

See Also

MPICaptureMessage

MPICaptureMessage

```
typedef enum {

    MPICaptureMessageCAPTURE_INVALID,
} MPICaptureMessage;
```

Description

MPICaptureMessageCAPTURE_INVALID	capture handle is invalid

See Also

MEICaptureSIMConfig

MEICaptureSIMConfig

```
typedef struct MEICaptureSIMConfig {
    long        enable;
} MEICaptureSIMConfig;
```

Description

CaptureSIMConfig contains the Sinusoidal Interpolation Mode (SIM) capture configuration.

enable	TRUE for SIM capture or FALSE for non-interpolated capture.

Remarks

Sinusoidal Interpolation Mode (SIM) mode requires the specialized SIM4 mezzanine board for the XMP.

See Also

Please refer to the SIM4 hardware application note 206 for further information.

MPICaptureState

MPICaptureState

```
typedef enum {
    MPICaptureStateINVALID,
    MPICaptureStateIDLE,
    MPICaptureStateARMED,
    MPICaptureStateCAPTURED,
} MPICaptureState;
```

Description

MPICaptureStateIDLE	Not armed or captured
MPICaptureStateARMED	Looking for capture I/O trigger
MPICaptureStateCAPTURED	I/O has triggered a capture.

See Also

MPICaptureStatus

MPICaptureLatchCountMAX

MPIObjectCaptureLatchCountMAX

```
#define MPICaptureLatchCountMAX (16)
        /* Maximum latches/capture */
```

Description **CaptureLatchCountMax** is an enumeration that is explained in MPICatpureConfig.

See Also

Command Objects

Introduction

The **Command** object specifies one of a variety of program Sequence commands. These include motion, conditional branch, computational, and time delay commands.

Methods

Create, Delete, Validate Methods

mpiCommand**Create**	Create Command object
mpiCommand**Delete**	Delete Command object
mpiCommand**Validate**	Validate Command object

Configuration and Informational Methods

mpiCommand**Label**	Get pointer to Command label
mpiCommand**Params**	Get Command parameters
mpiCommand**Type**	Return Command type

Other Methods

mpiCommand**AxisListGet**	Get the axisCount and axisList from a Command object.

Data Types

MPICommand**Address**
MPICommand**Constant**
MPICommand**Expr**
MPICommand**Message**
MPICommand**Motion**
MPICommand**Operator**
MPICommand**Params**
MPICommand**Type**

Copyright © 2002
Motion Engineering

mpiCommandCreate

Declaration

```
const MPICommand mpiCommandCreate(MPICommandType   type,
                                  MPICommandParams *params,
                                  const char       *label)
```

Required Header

stdmpi.h

Description

CommandCreate creates a Command object. The command type is specified by ***type***. The type-specific parameters are specified by ***params***. If ***label*** is not Null (i.e., something meaningful), then branch commands can call this Command (by using the ***label***). *CommandCreate* is the equivalent of a C++ constructor.

Return Values

handle	to a Command object
MPIHandleVOID	if the object could not be created

See Also

mpiCommandDelete | mpiCommandValidate

mpiCommandDelete

Declaration

```
long mpiCommandDelete(MPICommand  command)
```

Required Header

stdmpi.h

Description

CommandDelete deletes a Command object and invalidates its handle (***command***). CommandDelete is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *CommandDelete* successfully deletes the Command object and invalidates its handle

See Also

mpiCommandCreate | mpiCommandValidate

mpiCommandValidate

Declaration

```
long mpiCommandValidate(MPICommand  command)
```

Required Header

stdmpi.h

Description

CommandValidate validates the Command object and its handle (***command***).

Return Values

MPIMessageOK	if the Command object and its handle are valid

See Also

mpiCommandCreate | mpiCommandValidate

mpiCommandLabel

Declaration

```
long mpiCommandLabel(MPICommand  command,
                     char        **label)
```

Required Header

stdmpi.h

Description

CommandLabel gets the string from a Command and puts it in the location pointed to by label.

command	a handle to the Command object
****label**	a pointer to a string returned by the method

Return Values

pointer	to a Command's (***command***) label (that is in the location pointed to by ***label***)
MPIMessageOK	if *CommandLabel* successfully returns a pointer to the Command's label that is in the location pointed to by *****label***

See Also

mpiCommandCreate

mpiCommandParams

Declaration

```
long mpiCommandParams (MPICommand        command,
                       MPICommandParams *params)
```

Required Header

stdmpi.h

Description

CommandParams gets the parameters from a Command and puts it in the location pointed to by params.

command	a handle to the Command object
***params**	a pointer to a MPICommandParams structure returned by the method

Return Values

Command (*command*) parameters	in the structure pointed to by ***params***
MPIMessageOK	if *CommandParams* successfully gets and writes the command parameters into ***params***

See Also

mpiCommandCreate | MPICommandParams

mpiCommandType

Declaration

```
long mpiCommandType(MPICommand      command,
                    MPICommandType  *type)
```

Required Header

stdmpi.h

Description

CommandType gets the type from a Command and puts it in the location pointed to by type.

command	a handle to the Command object
***type**	a pointer to a MPICommandType returned by the method

Return Values

Command (*command*) parameters	in the location pointed to by ***type***
MPIMessageOK	if *CommandType* successfully gets and writes the command type into ****type***

See Also

mpiCommandCreate | MPICommandType

meiCommandAxisListGet

Declaration

```
long meiCommandAxisListGet(MPICommand  command,
                           long        *axisCount
                           MPIAxis     *axisList)
```

Required Header

stdmpi.h

Description

CommandAxisListGet reads number of axes and the list of axes associated with a motion type Command object (***command***) and writes them into the long pointed to by ***axisCount*** and the array of axis objects pointed to by ***axisList***.

command	a handle to the Command object
***axisCount**	a pointer to a long, representing the number of axes returned by the method
***axisList**	a pointer to an array of axis objects returned by the method

Return Values

MPIMessageOK	if *CommandAxisListGet* successfully gets the ***axisCount*** and *axisList* from a Command object.

See Also

MPICommand | MPIAxis | MPIMotion

MPICommandAddress

MPICommandAddress

```
typedef union {
    long    *l;
    float   *f;
} MPICommandAddress;
```

Description

CommandAddress defines a generic pointer that can specify either a *long* or a *float* pointer.

***l**	is used to access the long pointer of MPICommandAddress.
***f**	is used to access the float pointer of MPICommandAddress.

See Also

MPICommandConstant

MPICommandConstant

MPICommandConstant

```
typedef union {
    long    l;
    float   f;
} MPICommandConstant;
```

Description

CommandConstant defines a generic variable that can specify either a *long* or *float* value.

l	is used to access the long value of MPICommandConstant.
f	is used to access the float value of MPICommandConstant.

See Also

MPICommandAddress

MPICommandExpr

MPICommandExpr

```
typedef struct MPICommandExpr {
    MPICommandOperator  oper;
    MPICommandAddress   address;
        union {
            MPICommandConstant  value;  /* ['address'] 'oper' ['value'] */
            MPICommandAddress   ref;    /* ['address'] 'oper' [*'ref']  */
        } by;
} MPICommandExpr;
```

Description

CommandExpr is a structure that represents an expression for an MPICommand object.

The expression is evaluated as either:	
	*address **oper** value
	*address **oper** *ref
depending on the command type.	

See Also

MPICommand | MPICommandParams | MPICommandType

MPICommandMessage

MPICommandMessage

```
typedef enum {
   MPICommandMessageCOMMAND_INVALID,
   MPICommandMessageTYPE_INVALID,
   MPICommandMessagePARAM_INVALID,
} MPICommandMessage;
```

Description

MPICommandMessageCOMMAND_INVALID	is currently not supported and is reserved for future use.
MPICommandMessageTYPE_INVALID	The MPICommandType passed to the mpiCommandCreate method is invalid.

See Also

MPICommandType

MPICommandMotion

MPICommandMotion

```
typedef enum {
    MPICommandMotionINVALID,

    MPICommandMotionABORT,
    MPICommandMotionE_STOP,
    MPICommandMotionE_STOP_ABORT,
    MPICommandMotionMODIFY,
    MPICommandMotionRESET,
    MPICommandMotionRESUME,
    MPICommandMotionSTART,
    MPICommandMotionSTOP,
} MPICommandMotion;
```

Description

CommandMotion specifies what type of motion action a motion MPICommand object will perform. Please refer to MPIAction for more information on particular actions.

MPICommandMotionABORT	Commands an ABORT action.
MPICommandMotionE_STOP	Commands an E-STOP action.
MPICommandMotionE_STOP_ABORT	Commands an E-STOP_ABORT action.
MPICommandMotionMODIFY	Modifies a particular motion profile. This is currently not supported and is reserved for future use.
MPICommandMotionRESET	Commands an Reset action.
MPICommandMotionRESUME	Commands an Resume action.
MPICommandMotionSTART	Starts a motion profile.
MPICommandMotionSTOP	Commands an STOP action.

See Also

MPIAction | MPICommand | MPICommandParams

MPICommandOperator

MPICommandOperator

```
typedef enum {
   MPICommandOperatorINVALID,

   /* Arithmetic operators */
   MPICommandOperatorADD,
   MPICommandOperatorSUBTRACT,
   MPICommandOperatorMULTIPLY,
   MPICommandOperatorDIVIDE,

   MPICommandOperatorAND,
   MPICommandOperatorOR,
   MPICommandOperatorXOR,

   /* Logical operators */
   MPICommandOperatorALWAYS,

   MPICommandOperatorEQUAL,
   MPICommandOperatorNOT_EQUAL,

   MPICommandOperatorGREATER_OR_EQUAL,
   MPICommandOperatorGREATER,


   MPICommandOperatorLESS_OR_EQUAL,
   MPICommandOperatorLESS,

   MPICommandOperatorBIT_CLEAR,
   MPICommandOperatorBIT_SET,
} MPICommandOperator;
```

Description

The following are operators used by the MPICommand and MPICompare objects.

Arithmetic Operators	
MPICommandOperatorADD	Performs an addition. Equivalent to the C operator +.
MPICommandOperatorSUBTRACT	Performs a subtraction. Equivalent to the C operator -.
MPICommandOperatorMULTIPLY	Performs a multiplication. Equivalent to the C operator *.
MPICommandOperatorDIVIDE	Performs a divison. Equivalent to the C operator /.
MPICommandOperatorAND	Performs a logical AND. Equivalent to the C operator &.
MPICommandOperatorOR	Performs a logical OR. Equivalent to the C operator |.

MPICommandOperatorXOR	Performs a logical XOR. Equivalent to the C operator ^.

Logical Operators	
MPICommandOperatorALWAYS	Always evaluates TRUE. Equivalent in C to (1) or TRUE.
MPICommandOperatorEQUAL	Performs an equality comparison. Equivalent to the C operator ==.
MPICommandOperatorGREATER_OR_EQUAL	Performs an inequality comparison. Equivalent to the C operator !=.
MPICommandOperatorGREATER_OR_EQUAL	Performs a greater than or equal to comparison. Equivalent to the C operator >=.
MPICommandOperatorGREATER	Performs a greater than comparison. Equivalent to the C operator >.
MPICommandOperatorLESS_OR_EQUAL	Performs a less than or equal to comparison. Equivalent to the C operator <=.
MPICommandOperatorLESS	Performs a less than comparison. Equivalent to the C operator <.
MPICommandOperatorBIT_CLEAR	Clears specified bits. Equivalent in C to the statement: variable &= ~(bits)
MPICommandOperatorBIT_SET	Sets specified bits. Equivalent in C to the statement: variable |= (bits)

See Also

MPICommand | MPICommandExpr | MPICommandParams | MPIComparePosition | MPICompareParams

MPICommandParams

MPICommandParams

```
typedef union {
    struct {    /* *'dst' = 'value' */
        MPICommandAddress       dst;
        MPICommandConstant      value;
        MPIControl              control;        /* Ignored by Sequence */
    } assign;

    struct {    /* branch to 'label' on 'expr' */
        char            *label;         /* NULL => stop sequence */
        MPICommandExpr  expr;           /* expr.oper => MPICommandOperatorLogical */
        MPIControl      control;        /* Ignored by Sequence */
    } branch;

    struct {    /* branch to 'label' on MPIEventMask('handle') 'oper' 'mask' */
        char                    *label;         /* NULL => stop sequence */
        MPIHandle               handle;         /* [MPIMotor|MPIMotion|...] */
        MPICommandOperator      oper;           /* EQUAL/NOT_EQUAL/BIT_CLEAR/BIT_SET
*/
        MPIEventMask            mask;           /* MPIEventMask('handle') 'oper'
'mask' */
    } branchEvent;

    struct {    /* branch to 'label' on Io.input 'oper' 'mask' */
        char                    *label;         /* NULL => stop sequence */
        MPIIoType               type;           /* MOTOR,   USER */
        MPIIoSource             source;         /* MPIMotor     index */
        MPICommandOperator      oper;           /* EQUAL/NOT_EQUAL/BIT_CLEAR/BIT_SET
*/
        long                    mask;           /* [motor|user]Io.input 'oper' 'mask'
*/
    } branchIO;

    struct {    /* *'dst' = 'expr' */
        MPICommandAddress       dst;
        MPICommandExpr          expr;           /* expr.oper =>
MPICommandOperatorArithmetic */
        MPIControl              control;        /* Ignored by Sequence */
    } compute;

    struct {    /* Io.output = Io.output 'oper' 'mask' */
    MPIIoType                   type;           /* MOTOR,   USER */
        MPIIoSource             source;         /* MPIMotor     index */
        MPICommandOperator      oper;           /* AND/OR/XOR */
        long    mask;
    } computeIO;

    struct {    /* memcpy(dst, src, count) */
        void            *dst;
        void            *src;
        long            count;
        MPIControlcontrol;      /* Ignored by Sequence */
    } copy;
```

```
    float       delay;  /* seconds */

    struct {
        long            value;  /* MPIEventStatus.type    = MPIEventTypeEXTERNAL */
                                /*                .source  = MPISequence/MPIProgram */
                                /*                .info[0] = value */
        MPIEventMgr     eventMgr;       /* Ignored by Sequence */
    } event;

    struct {    /* mpiMotion[Abort|EStop|Reset|Resume|Start|Stop](motion[, type,
params]) */
        MPICommandMotion        motionCommand;
        MPIMotion               motion;
        MPIMotionType           type;   /* MPICommandMotionSTART */
        MPIMotionParams         params; /* MPICommandMotionSTART */
    } motion;

    struct {    /* wait until 'expr' */
        MPICommandExpr          expr;   /* expr.oper => MPICommandOperatorLogical */
        MPIControl              control;/* Ignored by Sequence */
    } wait;

    struct {    /* wait until MPIEventMask('handle') 'oper' 'mask' */
        MPIHandle               handle; /* [MPIMotor|MPIMotion|...] */
        MPICommandOperator      oper;   /* EQUAL/NOT_EQUAL/BIT_CLEAR/BIT_SET */
        MPIEventMask            mask;   /* MPIEventMask('handle') 'oper' 'mask' */
    } waitEvent;

    struct {    /* wait until Io.input 'oper' 'mask' */
        MPIIoType               type;   /* MOTOR,   USER */
        MPIIoSource             source; /* MPIMotor     index */
        MPICommandOperator      oper;   /* EQUAL/NOT_EQUAL/BIT_CLEAR/BIT_SET */
        long                    mask;   /* [motor|user]Io.input 'oper' 'mask' */
    } waitIO;
} MPICommandParams;
```

Description

assign	Assign a value to a particular controller address: ***dst = value** assign.control is currently not supported and is reserved for future use.
branch	Branch to a particular command (similar to a *goto* statement) if a particular comparison evaluates to TRUE: branch to **label** on **expr** If *label* = NULL, then no more commands will be executed if the comparison evaluates to TRUE. branch.control is currently not supported and is reserved for future use.
branchEvent	Branch to a particular command (similar to a *goto* statement) if a particular event occurs or has occured: branch to ***label*** on **MPIEventMask**(*handle*) **oper** *mask* If *label* = NULL, then no more commands will be executed if a particular event occurs or has occured.
branchIO	Branch to a particular command (similar to a goto statement) if a particular i/o state matches a specified condition: branch to *label* on *Io.input* **oper** *mask* If *label* = NULL, then no more commands will be executed if a particular i/o state matches a specified condition.
compute	perform some computation and place the result at some controller address: ***dst = expr** compute.control is currently not supported and is reserved for future use.
computeIO	Performs a computation on a set of i/o bits: *Io.output* = *Io.output* **oper** *mask*

copy	Copies controller memory from one place to another: **memcpy**(*dst, src, count*); Remember: ***count*** represents the number of **bytes** copied, NOT the number of controller words. event.control is currently not supported and is reserved for future use.
delay	Delays execution of the next command *delay* seconds.
event	Generates an event: MPIEventStatus.type = MPIEventTypeEXTERNAL MPIEventStatus.source = MPISequence MPIEventStatus.info[0] = value event.eventMgr is currently not supported and is reserved for future use.
motion	Commands a motion action (See MPICommandMotion): **mpiMotionStart** (*motion*, *type*, *params*]); or **mpiMotionAction**(*motion*, MPIAction[ABORT | E_STOP | E_STOP_ABORT | RESET | RESUME | STOP]);
wait	Delays execution of the next command until a particular comparison evaluates to TRUE: wait until *expr* wait.control is currently not supported and is reserved for future use.
waitEvent	Delays execution of the next command until a particular event occurs: wait until **MPIEventMask**(*handle*) **oper** *mask*
waitIO	Delays execution of the next command until a particular i/o state matches a specified condition: wait until *Io.input* **oper** *mask*

See Also

MPICommand | MPICommandType | mpiCommandCreate | mpiCommandParams

MPICommandType

MPICommandType

```
typedef enum {
    MPICommandTypeASSIGN,
    MPICommandTypeASSIGN_FLOAT,

    MPICommandTypeBRANCH,
    MPICommandTypeBRANCH_REF,
    MPICommandTypeBRANCH_FLOAT,
    MPICommandTypeBRANCH_FLOAT_REF,
    MPICommandTypeBRANCH_EVENT,
    MPICommandTypeBRANCH_IO,

    MPICommandTypeCOMPUTE,
    MPICommandTypeCOMPUTE_REF,
    MPICommandTypeCOMPUTE_FLOAT,
    MPICommandTypeCOMPUTE_FLOAT_REF,
    MPICommandTypeCOMPUTE_IO,

    MPICommandTypeCOPY,

    MPICommandTypeDELAY,

    MPICommandTypeEVENT,

    MPICommandTypeMOTION,

    MPICommandTypeWAIT,
    MPICommandTypeWAIT_REF,
    MPICommandTypeWAIT_FLOAT,
    MPICommandTypeWAIT_FLOAT_REF,
    MPICommandTypeWAIT_EVENT,
    MPICommandTypeWAIT_IO,
} MPICommandType;
```

Description

MPICommandTypeASSIGN	
MPICommandTypeASSIGN_FLOAT	These commands assign a value to a particular controller address. MPICommandTypeASSIGN assigns a long value while MPICommandTypeASSIGN_FLOAT assigns a float value.
MPICommandTypeBRANCH	These commands branch to a particular command (similar to a goto statement) if a particular comparison evaluates to TRUE. MPICommandTypeBRANCH compares a controller address to a specified constant long value. MPICommandTypeBRANCH_REF Compares a controller address to a long value at a specified controller address.

MPICommandTypeBRANCH_FLOAT	Compares a controller address to a specified constant float value.
MPICommandTypeBRANCH_FLOAT_REF	Compares a controller address to a float value at a specified controller address.
MPICommandTypeBRANCH_EVENT	Branch to a particular command (similar to a goto statement) if a particular event occurs or has occured.
MPICommandTypeBRANCH_IO	Branch to a particular command (similar to a goto statement) if a particular i/o state matches a specified condition.
MPICommandTypeCOMPUTE	These commands perform some computation and place the result at some controller address.. MPICommandTypeCOMPUTE performs a computation of some controller address and a constant long value.
MPICommandTypeCOMPUTE_REF	Performs a computation of some controller address and a long value at a specified controller address.
MPICommandTypeCOMPUTE_FLOAT	Performs a computation of some controller address and a constant float value.
MPICommandTypeCOMPUTE_FLOAT_REF	Performs a computation of some controller address and a float value at a specified controller address.
MPICommandTypeCOMPUTE_IO	Performs a computation on a set of i/o bits.
MPICommandTypeCOPY	Copies controller memory from one place to another.
MPICommandTypeDELAY	Delays execution of the next command.
MPICommandTypeEVENT	Generate an event.
MPICommandTypeMOTION	Commands a motion action. See MPICommandMotion.
MPICommandTypeWAIT	These delays execution of the next command until a particular comparison evaluates to TRUE. MPICommandTypeWAIT compares a controller address to a specified constant long value. MPICommandTypeWAIT_REF Compares a controller address to a long value at a specified controller address.
MPICommandTypeWAIT_FLOAT	Compares a controller address to a specified constant float value.
MPICommandTypeWAIT_FLOAT_REF	Compares a controller address to a float value at a specified controller address.
MPICommandTypeWAIT_EVENT	Delays execution of the next command until a particular event occurs.
MPICommandTypeWAIT_IO	Delays execution of the next command until a particular i/o state matches a specified condition.

See Also

MPICommand | MPICommandMotion | MPICommandParams | mpiCommandCreate | mpiCommandType

Compare Objects

Introduction

A **Compare** object manages a compare engine in the motion controller hardware. The compare engine compares the actual position feedback from a motor to a preloaded position value. When the actual position feedback exceeds the value, the compare engine sets an output bit to the specified state. Since the compare occurs in hardware, the latency is minimal.

Typically, the MPI is used to configure the compare engine, load the compare value, and arm the compare. The condition of the compare engine can be determined by reading its status.

Methods

Create, Delete, Validate Methods

mpiCompare**Create**	Create Compare object
mpiCompare**Delete**	Delete Compare object
mpiCompare**Validate**	Validate Compare object

Configuration and Information Methods

mpiCompare**ConfigGet**	Get Compare configuration
mpiCompare**ConfigSet**	Set Compare configuration
mpiCompare**FlashConfigGet**	Get flash configuration for Compare
mpiCompare**FlashConfigSet**	Set flash configuration for Compare
mpiCompare**Status**	Get Compare's status

Memory Methods

mpiCompare**Memory**	Set Compare memory address
mpiCompare**MemoryGet**	Copy bytes of Compare memory to application memory
mpiCompare**MemorySet**	Copy bytes of application memory to Compare memory

Action Methods

mpiCompare**Arm**	Arm Compare object
mpiCompare**Load**	

Relational Methods

mpiCompare**Control**	Return handle of Control that is assoiciated with Compare
mpiCompare**Number**	Get index of Compare (for Control list)

Data Types

MPICompare**Config** / MEICompare**Config**
MPICompare**Message**
MPICompare**Params**
MPICompare**Position**
MPICompare**State**
MPICompare**Status**

Constants

MPICompare**PositionCountMAX**

Copyright © 2002
Motion Engineering

mpiCompareCreate

Declaration

```
const MPICompare mpiCompareCreate(MPIControl control,
                                  long       number)
```

Required Header

stdmpi.h

Description

CompareCreate creates a Compare object associated with a number (***number***), that is located on a motion controller (***control***).

CompareCreate is the equivalent of a C++ constructor. Each motion block supports 10 compare registers. The default configuration is two compare registers per motor, while the last two (8,9) on each motion block are reserved for the Auxiliary Encoder (not supported). The compare registers are default mapped as follows: 0 and 1 for Motor0; 2 and 3 for Motor1; 10 and 11 for Motor4; etc. The first Compare for each motor uses the default (primary) encoder input for position compare. The second uses the AUX encoder input.

Return Values

handle	to a Compare object
MPIHandleVOID	if the object could not be created

See Also

mpiCompareCreate | mpiCommandValidate

mpiCompareDelete

Declaration `long mpiCompareDelete(MPICompare compare)`

Required Header stdmpi.h

Description **CompareDelete** deletes a Compare object and invalidates its handle (***compare***). *CompareDelete* is the equivalent of a C++ destructor.

Return Values	
MPIMessageOK	if *CompareDelete* successfully deletes the Compare object and invalidates its handle

See Also mpiCommandCreate | mpiCommandValidate

mpiCompareValidate

Declaration `long mpiCompareValidate(MPICompare compare)`

Required Header compare.h

Description **CompareValidate** validates the Compare object and its handle (***compare***).

Return Values	
MPIMessageOK	if Compare is a handle to a valid object.

See Also mpiCompareCreate | mpiCompareDelete

mpiCompareConfigGet

Declaration

```
long mpiCompareConfigGet(MPICompare        compare,
                         MPICompareConfig *config,
                         void             *external)
```

Required Header stdmpi.h

Description

CompareConfigGet gets a Compare object's (***compare***) configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The a Compare object's configuration information in ***external*** is in addition to the Compare object's configuration information in ***config***, i.e, the Compare object's configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEICompareConfig{}** or is NULL.

Return Values

MPIMessageOK	if *CompareConfigGet* successfully writes the Compare object's configuration to the structure(s)

See Also mpiCompareConfigSet | MEICompareConfig

mpiCompareConfigSet

Declaration

```
long mpiCompareConfigSet(MPICompare        compare,
                         MPICompareConfig *config,
                         void             *external)
```

Required Header

stdmpi.h

Description

CompareConfigSet sets a Compare object's (***compare***) configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Compare object's configuration information in ***external*** is in addition to the Compare object's configuration information in ***config***, i.e, the Compare object's configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type **MEICompareConfig{}** or is NULL.

Return Values

MPIMessageOK	if *CompareConfigSet* successfully sets the Compare object's configuration using data from the structure(s)

See Also

mpiCompareConfigGet | MEICompareConfig

mpiCompareFlashConfigGet

Declaration

```
long mpiCompareFlashConfigGet(MPICompare         compare,
                              void               *flash,
                              MPICompareConfig   *config,
                              void               *external)
```

Required Header

stdmpi.h

Description

CompareFlashConfigGet gets a Compare object's (***compare***) flash configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Compare object's flash configuration information in ***external*** is in addition to the Compare object's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEICompareConfig{}** or is NULL.

Return Values

MPIMessageOK	if *CompareFlashConfigGet* successfully writes the Compare object's flash configuration to the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiCompareFlashConfigSet | MEICompareConfig

mpiCompareFlashConfigSet

Declaration

```
long mpiCompareFlashConfigSet(MPICompare        compare,
                              void              *flash,
                              MPICompareConfig  *config,
                              void              *external)
```

Required Header

stdmpi.h

Description

CompareFlashConfigSet sets a Compare object's (***compare***) flash configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Compare object's flash configuration information in ***external*** is in addition to the Compare object's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only — ***external*** either points to a structure of type **MEICompareConfig{}** or is NULL.

Return Values

MPIMessageOK	if *CompareFlashConfigSet* successfully sets the Compare object's flash configuration using data from the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiCompareFlashConfigGet | MEICompareConfig

mpiCompareStatus

Declaration

```
long mpiCompareStatus(MPICompare          compare,
                      MPICompareStatus    *status,
                      void                *external)
```

Required Header

stdmpi.h

Description

CompareStatus writes a Compare object's (***compare***) status into the structure pointed to by ***status***, and also into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Compare object's status information in ***external*** is in addition to the Compare object's status information in ***status***, i.e, the status configuration information in ***status*** and in ***external*** is not the same information. Note that ***status*** or ***external*** can be NULL (but not both NULL).

XMP Only — ***external*** either points to a structure of type **MEICompareStatus{}** or is NULL.

Return Values

MPIMessageOK	if *CompareStatus* successfully writes the status of a Compare object to the structure(s)

See Also

mpiCompareMemory

Declaration

```
long mpiCompareMemory(MPICompare   compare,
                      void         **memory)
```

Required Header

stdmpi.h

Description

CompareMemory writes an address [which is used to access a Compare object's (***compare***) memory] to the contents of ***memory***. This address, or an address calculated from it, can be passed as the src parameter to mpiCompareMemoryGet(...) and as the dst parameter to mpiCompareMemorySet(...).

Return Values

MPIMessageOK	if *CompareMemory* successfully writes the Compare object's memory address to the contents of memory

See Also

mpiCompareMemoryGet | mpiCompareMemorySet

mpiCompareMemoryGet

Declaration

```
long mpiCompareMemoryGet(MPICompare  compare,
                         void        *dst,
                         void        *src,
                         long        count)
```

Required Header stdmpi.h

Description **CompareMemoryGet** copies ***count*** bytes of a Compare object's (***compare***) memory (starting at address ***src***) and writes them into application memory (starting at address ***dst***).

Return Values	
MPIMessageOK	if *CompareMemoryGet* successfully copies data from Compare memory to application memory

See Also mpiCompareMemorySet | mpiCompareMemory

mpiCompareMemorySet

Declaration

```
long mpiCompareMemorySet(MPICompare  compare,
                         void        *dst,
                         void        *src,
                         long        count)
```

Required Header stdmpi.h

Description **CompareMemorySet** copies ***count*** bytes of application memory (starting at address ***src***) and writes them into a Compare object's (***compare***) memory (starting at address ***dst***).

Return Values	
MPIMessageOK	if *CompareMemoryGet* successfully copies data from application memory to Compare memory

See Also mpiCompareMemoryGet | mpiCompareMemory

mpiCompareArm

Declaration

```
long mpiCompareArm(MPICompare compare,
                   long       arm)
```

Required Header stdmpi.h

Description **CompareArm** sets the MPICompareState to MPICompareStateIDLE (arm = FALSE), or MPICompareStateARMED (arm = TRUE).

Return Values

MPIMessageOK	if the Compare object is successfully armed or disarmed.

See Also MPICompareState

mpiCompareLoad

Declaration

```
long mpiCompareLoad(MPICompare       compare,
                    MPICompareParams *params,
                    void             *external)
```

Required Header

stdmpi.h

Description

CompareLoad loads a Compare object's (***compare***) parameters using data from the structure pointed to by ***params***, and also using data from the implementation-specific structure.

Return Values

MPIMessageOK	if *CompareLoad* successfully loads the Compare object's parameters using the data from the structure.

See Also

mpiCompareControl

Declaration `const MPIControl mpiCompareControl(MPICompare compare)`

Required Header stdmpi.h

Description **CompareControl** returns a handle to the motion controller (Control object) that a Compare object (***compare***) is associated with.

Return Values

handle	to a Control object that a Compare object is associated with
MPIHandleVOID	if the Compare object is invalid

See Also

mpiCompareNumber

Declaration

```
long mpiCompareNumber(MPICompare   compare,
                      long         *number)
```

Required Header

stdmpi.h

Description

CompareNumber writes the index of a Compare object (***compare***, on the motion controller that ***compare*** is associated with) to the contents of ***number***.

Return Values

MPIMessageOK	if *CompareNumber* successfully writes the index of a Compare object to the contents of ***number***

See Also

MPICompareConfig / MEICompareConfig

MPICompareConfig

```
typedef struct MPICompareConfig {
    MPIIoTrigger    trigger;    /* which output to use... */
} MPICompareConfig;
```

Description

trigger	type, source, mask, and pattern used to select the state of the compare output bit upon reaching the compare position. For more information about setting the trigger please see MPIIoTrigger.

MEICompareConfig

```
typedef struct MEICompareConfig {
    long                    continuous;
    MEICompareDivByNConfig  divByN;
} MEICompareConfig;
```

Description

Event compare mode (default) uses a handshake to ensure hardware/software synchronization. A single rising edge and single falling edge on the compare output is guaranteed. This mode is useful when re-arming compare objects is required for multiple compare position. There is system overhead to re-arm compare events.

Continuous compare mode constantly compares the compare position register to the position counter. The compare output is toggled based on compare logic, without software system notification and without the need to re-arm. This mode is useful when a single compare position is required for a valid compare output whenever the position is past some limit. If the position feedback during a move is not monotonic at the compare value (jitters back and forth), the compare output will change state each time the position crosses the compare value.

See Also

MPIIoTrigger | mpiCompareConfigGet | mpiCompareConfigSet

MPICompareMessage

MPICompareMessage

```
typedef enum {
    MPICompareMessageCOMPARE_INVALID,
} MPICompareMessage;
```

Description

MPICompareMessageCOMPARE_INVALID	compare handle is invalid

See Also

MPICompareParams

MPICompareParams

```
typedef struct MPICompareParams {
    long                position;
    long                outputState;
    MPICommandOperator  commandOperator;
} MPICompareParams;
```

Description

position	actual position at which to toggle output bit.
outputState	state of output bit upon reaching the compare position. Set to TRUE (on) or FALSE (off).
commandOperator	logical operator for compare position.

Remarks

Valid values for commandOperator are MPICommandOperatorLESS_OR_EQUAL and MPICommandOperatorGREATER. Based on the commandOperator, the specified outputState of the bit will be set on one side or the other of the compare position.

The commandOperator logic is usually set depending on the direction of travel toward the compare position.

See Also

MPIComparePosition

MPIComparePosition

```
typedef struct MPIComparePosition {
    long                number;
    long                motorNumber;
    long                position;
    MPICommandOperator  commandOperator;
} MPIComparePosition;
```

Description - This structure is currently not supported and is reserved for future use.

number	capture number
motorNumber	number of the motor whose position to compare
position	compare position on which to set the output bit
commandOperator	logical operator for the compare position

See Also

MPICompareState

MPICompareState

```
typedef enum {
    MPICompareStateINVALID,

    MPICompareStateIDLE,
    MPICompareStateARMED,
    MPICompareStateCOMPARED,
} MPICompareState;
```

Description

MPICompareStateIDLE	Not armed or compared
MPICompareStateARMED	Looking for compare position(s)
MPICompareStateCOMPARED	Compare position found

See Also

MPICompareStatus

MPICompareStatus

MPICompareStatus

```
#define MPIComparePositionCountMAX (10)  /* Maximum latches/compare */

typedef enum {
    MPICompareStateINVALID,

    MPICompare StateIDLE,
    MPICompare StateARMED,
    MPICompare StateCOMPARED,
} MPICompareState;

typedef struct MPICompareStatus {
    MPICompareState     state;
    double              position[MPIComparePositionCountMAX];
} MPICompareStatus;
```

Description

State- contains the current state of the compare register:

Idle	Not armed or compared
Armed	Looking for compare position(s)
Compared	Compare position found

Position- array containing the controller's compared position value(s).

See Also

MPICompareState

MPIComparePositionCountMAX

Declaration

```
#define MPIComparePositionCountMAX (10)
/* Maximum latches/compare */
```

Required Header stdmpi.h

Description See MPICompareStatus for a description.

See Also

Control Objects

Introduction

A **Control** object manages a motion controller device. The device is typically a single board residing in a PC or an embedded system. A control object can read and write device memory through one of a variety of methods: I/O port, memory mapped or device driver.

For the case where the application and the motion controller device exist on two physically separate platforms connected by a LAN or serial line, the application creates a client control object which communicates via remote procedure calls with a server.

Unlike the methods of all other objects in the MPI, Control object methods are not thread-safe.

Are you using TCP/IP and Sockets? If yes, click here.

Methods

Create, Delete, Validate Methods

mpiControl**Create**	Create Control object
mpiControl**Delete**	Delete Control object
mpiControl**Validate**	Validate Control object

Configuration and Information Methods

mpiControl**Address**	Get original address of Control object (when it was created)
mpiControl**ConfigGet**	Get Control config
mpiControl**ConfigSet**	Set Control config
meiControl**ExtMemAvail**	
mpiControl**FlashConfigGet**	Get Control flash config
mpiControl**FlashConfigSet**	Set Control flash config
meiControl**GateGet**	Get the closed state (TRUE or FALSE)
meiControl**GateSet**	Set the closed state (TRUE or FALSE)
meiControl**SampleCounter**	
meiControl**SamplestoSeconds**	Converts samples to seconds
meiControl**SecondstoSamples**	Converts seconds to samples
mpiControl**Type**	Get type of Control object (used to create Command object)
meiControl**VersionGet**	Read the version of XMP firmware
meiControl**VersionSet**	Write the version of XMP firmware

Memory Methods

mpiControl**Memory**	Get address of Control memory
mpiControl**MemoryAlloc**	Allocate bytes of firmware memory
mpiControl**MemoryCount**	Get number of bytes available in firmware
mpiControl**MemoryFree**	Free bytes of firmware memory
mpiControl**MemoryGet**	Copy count bytes of Control memory to application memory
mpiControl**MemorySet**	Copy count bytes of application memory to Control memory

Action Methods

mpiControl**CycleWait**	Wait for Control to execute count cycles
mpiControl**Init**	Initialize Control object
mpiControl**InterruptEnable**	Enable interrupts to Control object
mpiControl**InterruptWait**	Wait for controller interrupt
mpiControl**InterruptWake**	Wake all threads waiting for controller interrupt
mpiControl**Reset**	Reset controller hardware
meiControl**SampleWait**	Specify how many samples the host waits for, while the XMP executes

Relational Methods

meiControl**Platform**

Data Types

MPIControl**Address**
MPIControl**Config** / MEIControl**Config**
MPIControl**Io**
MEIControl**Input**
MPIControl**Message** / MEIControl**Message**
MEIControl**Output**
MPIControl**Type**
MEIControl**Version**

Copyright © 2002
Motion Engineering

mpiControlCreate

Declaration `const MPIControl mpiControlCreate(MPIControlType type, MPIControlAddress *address)`

Required Header stdmpi.h

Description **ControlCreate** creates a Control object of the specified ***type*** and type-specific ***address***. ControlCreate is the equivalent of a C++ constructor.

The type parameter determines the form of the address parameter:

If the "type" parameter is	*Then the form of the "address" parameter is*
MPIControlTypeDEFAULT	implementation-specific
MPIControlTypeMAPPED	MPIControlAddress.mapped
MPIControlTypeIOPORT	MPIControlAddress.ioPort
MPIControlTypeDEVICE	MPIControlAddress.device
MPIControlTypeCLIENT	MPIControlAddress.client

Note: This constructor does not reset or initialize the motion control device.

About MPIControlTypes:

If **MPIControlType** is	And **MPIControlAddress** is	Then the **Board Number** is	And the **"address" parameter** to be used is
DEFAULT1	Null address	0 address.number	default address parameter default address parameter
DEVICE2	Null address	0 address.number	default device driver address.type.device (if address.type.device is Null, then default device driver)
A valid ***type***	address	address.number	N/A (if address parameter is not defined or Null, then an error is generated)

1. If the ***type*** is DEFAULT, then the address structure (if supplied) is referenced **only for the board number**. Note that even if the default ***type*** is DEVICE, the default device driver will be used and *address.type.device* will not be used.
2. If the ***type*** is explicitly DEVICE, and the ***address*** is provided, then address.number will be used. If *address.type.device* is NULL, then the default device driver will be used. If *address.type.device* is not NULL, then the specified driver (DEVICE) will be used.

Sample Code

In general, if the caller specifies an explicit type (i.e., not DEFAULT), then the caller must completely fill out the address.type structure.

A simple case that will work for almost anyone who wants to use board #0:

```
mpiControlCreate(MPIControlTypeDEFAULT, NULL);
```

A simple case where board #1 is desired is:

```
{
   MPIControlAddress address;
   address.number = 1;
   mpiControlCreate(MPIControlTypeDEFAULT, &address);
}
```

Since the default MPIControlType = MPIControlTypeDEVICE, the *address* may be on the stack with garbage for the device driver name. This isn't a problem, however, because the board number is the only field in *address* that will be used when the caller specifies the DEFAULT MPIControlType.

Return Values

handle	to a Control object
MPIHandleVOID	if the object could not be created

See Also

MPIControl | MPIControlAddress | MPIControlType | mpiControlValidate
mpiControlInit | mpiControlDelete

mpiControlDelete

Declaration `long mpiControlDelete(MPIControl control)`

Required Header stdmpi.h

Description

ControlDelete deletes a control object and invalidates its handle. *ControlDelete* is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *ControlDelete* successfully deletes a Control object and invalidates its handle

See Also mpiControlCreate | mpiControlValidate

mpiControlValidate

Declaration `long mpiControlValidate(MPIControl control)`

Required Header stdmpi.h

Description **ControlValidate** validates the control object and its handle.

Return Values	
MPIMessageOK	if Control is a handle to a valid object.

See Also mpiControlCreate | mpiControlDelete

mpiControlAddress

Declaration

```
long mpiControlAddress(MPIControl        control,
                       MPIControlAddress *address)
```

Required Header stdmpi.h

Description When a Control object (***control***) is created, an address is used. **ControlAddress** writes this address to the contents of ***address***.

Return Values

MPIMessageOK	if *ControlAddress* successfully writes the address (used when ***control*** was created) to the contents of ***address***

See Also

mpiControlConfigGet

Declaration

```
long mpiControlConfigGet(MPIControl        control,
                         MPIControlConfig  *config,
                         void              *external)
```

Required Header

stdmpi.h

Description

ControlConfigGet gets the configuration of a Control object (***control***) and writes it into the structure pointed to by **config**, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in config, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEIControlConfig{}** or is NULL.

Return Values

MPIMessageOK	if *ControlConfigGet* successfully gets the ***control*** configuration and writes it in the structure(s)

See Also

mpiControlConfigSet | MEIControlConfig |
Special Note on Dynamic Allocation of External Memory Buffers.

mpiControlConfigSet

Declaration

```
long mpiControlConfigSet(MPIControl        control,
                         MPIControlConfig  *config,
                         void              *external)
```

Required Header

stdmpi.h

Description

ControlConfigSet sets (writes) the Control object's (***control***) configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in config, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEIControlConfig{}** or is NULL.

Return Values

MPIMessageOK	if *ControlConfigSet* successfully writes the Control object's configuration using data from the structure(s)

See Also

mpiControlConfigGet | MEIControlConfig |
Special Note on Dynamic Allocation of External Memory Buffers.

meiControlExtMemAvail

Declaration

```
long meiControlExtMemAvail(MPIControl control,
                           long       *size)
```

Required Header

stdmei.h

Description

ControlExtMemAvail gets the amount of external memory available on an XMP-Series controller. It puts the number of words (16 bit) in the location pointed to by size.

control	a handle to the Control object
***size**	a pointer to the available memory words returned by the method

Return Values

MPIMessageOK	if *ControlExtMemAvail* successfully gets and writes the available external memory words into ***size***

See Also

MPIControlConfig

mpiControlFlashConfigGet

Declaration

```
long mpiControlFlashConfigGet(MPIControl        control,
                              void              *flash,
                              MPIControlConfig  *config,
                              void              *external)
```

Required Header stdmpi.h

Description

ControlFlashConfigGet gets the flash configuration of a Control object (control) and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Control's flash configuration information in ***external*** is in addition to the Control's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only

external either points to a structure of type **MEIControlConfig{}** or is NULL. ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *ControlFlashConfigGet* successfully gets the Control's flash configuration and writes it into the structure(s)

See Also MEIFlash | mpiControlFlashConfigSet | | MEIControlConfig

mpiControlFlashConfigSet

Declaration

```
long mpiControlFlashConfigSet(MPIControl        control,
                              void              *flash,
                              MPIControlConfig  *config,
                              void              *external)
```

Required Header

stdmpi.h

Description

ControlFlashConfigSet sets (writes) the flash configuration of a Control object (***control***), using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Control's flash configuration information in ***external*** is in addition to the Control's flash configuration information in config, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only

external either points to a structure of type **MEIControlConfig{}** or is NULL. flash is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *ControlFlashConfigSet* successfully sets (writes) the Control's flash configuration using data from the structure(s)

See Also

MEIFlash | mpiControlFlashConfigGet | | MEIControlConfig

meiControlGateGet

Declaration

```
long meiControlGateGet(MPIControl control,
                       long       gate,
                       long       *closed)
```

Required Header

stdmei.h

Description

ControlGateGet gets the closed state (TRUE or FALSE) from the specified control gate (0 to 31).

Return Values

MPIMessageOK	if *ControlGateGet* successfully gets (reads) the state from the control gate and puts it into closed.

See Also

meiControlGateSet

meiControlGateSet

Declaration

```
long meiControlGateSet(MPIControl control,
                       long       gate,
                       long       closed)
```

Required Header

stdmei.h

Description

ControlGateSet sets the closed state (TRUE or FALSE) for the specified control gate (0 to 31).

Return Values

MPIMessageOK	if *ControlGateSet* successfully sets (writes) the closed state into the control gate.

See Also

meiControlGateGet

meiControlSampleCounter

Declaration

```
long meiControlSampleCounter(MPIControl  control,
                             long        *sampleCounter)
```

Required Header stdmei.h

Description

ControlSampleCounter writes the number of servo cycles (samples) that have occured since the last sample counter reset/rollover, to the ***sampleCounter*** . When the user resets the controller, the sample counter will also be reset. Since the sample counter is a long, if the sample counter is 2147483647 it will roll over on the next servo cycle to -2147483648.

Return Values

MPIMessageOK	if the sample counter could be read

See Also meiControlSecondstoSamples | meiControlSamplestoSeconds | meiControlSampleWait

meiControlSamplesToSeconds

Declaration

```
long meiControlSamplesToSeconds(MPIControl control,
                                long       samples,
                                float      *seconds)
```

Required Header stdmei.h

Description **ControlSamplesToSeconds** writes to seconds the number of seconds it takes to process samples number of ***samples*** (at the current sample rate). Use this function to convert samples to ***seconds***.

Return Values

MPIMessageOK	if *ControlSampleToSeconds* successfully converts the samples to seconds.

See Also meiControlSecondstoSamples | meiControlSampleCounter

meiControlSecondstoSamples

Declaration

```
long meiControlSecondsToSamples(MPIControl control,
                                float      seconds,
                                long       *samples)
```

Required Header stdmei.h

Description **ControlSecondsToSamples** writes to samples the number of servo cycles that will take place in seconds number of ***seconds*** (at the current sample rate). Use this function to convert seconds to ***samples***.

Return Values

MPIMessageOK	if *ControlSecondsToSamples* successfully converts the seconds to samples.

See Also meiControlSamplestoSeconds | meiControlSampleCounter | meiControlSampleWait

mpiControlType

Declaration

```
long mpiControlType(MPIControl     control,
                    MPIControlType *type)
```

Required Header

stdmpi.h

Description

When a Control object (***control***) is created, a type is used. **ControlType** writes this type to the contents of ***type***.

Return Values

MPIMessageOK	if *ControlType* successfully gets the type (used when ***control*** was created) to the contents of ***type***

See Also

meiControlVersionGet

Declaration

```
long meiControlVersionGet(MPIControl        control,
                          MEIControlVersion *version)
```

Required Header stdmei.h

Description **ControlVersionGet** writes the the version numbers of the XMP firmware, hardware, and the MPI library to the structure pointed to by ***version***.

Return Values	
MPIMessageOK	if *ControlVersionGet* successfully writes the the version numbers of the XMP firmware, hardware, and the MPI library to the location

See Also meiControlVersionSet

meiControlVersionSet

Declaration

```
long meiControlVersionSet(MPIControl        control,
                          MPIControlVersion *version)
```

Required Header

stdmei.h

Description

ControlVersionSet sets the version numbers of the XMP firmware, hardware, and the MPI library using data from the structure pointed to by version.

Normally, the MPI library is compatible only with the XMP firmware for which the library is specifically built; i.e., only when

version -> mpi.firmware.version == version -> xmp.firmware.version

However, there are times when it is desirable to have the MPI library ignore incompatible firmware and continue to operate. As an example, the flash utility instructs the MPI library to ignore firmware incompatibility when new firmware is being loaded. Of course, this new firmware should also be compatible with the MPI library. In such cases, the version -> xmp.firmware structure will be copied into ***control***.

Return Values

MPIMessageOK	if *ControlVersionSet* successfully sets the version numbers of the XMP firmware, hardware, and the MPI library using data from the structure

See Also

meiControlVersionGet

mpiControlMemory

Declaration

```
long mpiControlMemory(MPIControl control,
                      void       **memory,
                      void       **external)
```

Required Header

stdmpi.h

Description

ControlMemory sets (writes) an address (used to access a Control object's memory) to the contents of ***memory***.

If ***external*** is not NULL, the contents of ***external*** are set to an implementation-specific address that typically points to a different section or type of Control memory other than ***memory*** (e.g., to external or off-chip memory). These addresses (or addresses calculated from them) are passed as the src argument to mpiControlMemoryGet(...) and the dst argument to mpiControlMemorySet(...).

Return Values

MPIMessageOK	if *ControlMemory* successfully writes the address(es) (used to access Control memory, and optionally to access another section of Control memory) to the contents of ***memory*** (and to ***external***, if ***external*** is not Null)

See Also

mpiControlMemoryGet | mpiControlMemorySet | mpiControlMemoryAlloc | mpiControlMemoryCount | mpiControlMemoryFree

mpiControlMemoryAlloc

Declaration

```
long mpiControlMemoryAlloc(MPIControl           control,
                           MPIControlMemoryType type,
                           long                 count,
                           void                 **memory)
```

Required Header stdmpi.h

Description **ControlMemoryAlloc** allocates ***count*** bytes of firmware memory [of type ***type*** on a Control object (***control***)] and writes the host address (of the allocated firmware memory) to the location pointed to by ***memory***.

Return Values

MPIMessageOK	if *ControlMemoryAlloc* successfully allocates firmware memory and writes the host address of that firmware memory to *memory*

See Also mpiControlMemoryGet | mpiControlMemorySet | mpiControlMemory | mpiControlMemoryCount | mpiControlMemoryFree

mpiControlMemoryCount

Declaration

```
long mpiControlMemoryCount(MPIControl           control,
                           MPIControlMemoryType type,
                           long                 *count)
```

Required Header control.h

Description **ControlMemoryCount** writes the number of bytes of firmware memory [on a Control object (***control***, of type ***type***) that are available to be allocated] to the location pointed to by ***count***.

Return Values

MPIMessageOK	if *ControlMemoryCount* successfully writes the number of bytes of firmware memory (that are available to be allocated) to *count*.

See Also

mpiControlMemoryFree

Declaration

```
long mpiControlMemoryFree(MPIControl           control,
                          MPIControlMemoryType type,
                          long                 count,
                          void                 *memory)
```

Required Header stdmpi.h

Description **ControlMemoryFree** frees ***count*** bytes of firmware memory on a Control object (***control***, of type ***type***) starting at host address ***memory***.

Return Values	
MPIMessageOK	if *ControlMemoryAlloc* successfully frees ***count*** bytes of firmware memory on a Control object

See Also mpiControlMemoryGet | mpiControlMemorySet | mpiControlMemoryAlloc | mpiControlMemoryCount | mpiControlMemory

mpiControlMemoryGet

Declaration

```
long mpiControlMemoryGet(MPIControl control,
                         void       *dst,
                         void       *src,
                         long       count)
```

Required Header

stdmpi.h

Description

ControlMemoryGet gets ***count*** bytes of ***control*** memory (starting at address ***src***) and puts (writes) them in application memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *ControlMemoryGet* successfully gets ***count*** bytes of ***control*** memory and puts (writes) them in application memory

See Also

mpiControlMemorySet | mpiControlMemory | mpiControlMemoryAlloc | mpiControlMemoryCount | mpiControlMemoryFree

mpiControlMemorySet

Declaration

```
long mpiControlMemorySet(MPIControl control,
                         void       *dst,
                         void       *src,
                         long       count)
```

Required Header

stdmpi.h

Description

ControlMemorySet sets (writes) ***count*** bytes of application memory (starting at address ***src***) to ***control*** memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *ControlMemorySet* successfully sets (writes) count bytes of application memory to control memory

See Also

mpiControlMemoryGet | mpiControlMemory | mpiControlMemoryAlloc | mpiControlMemoryCount | mpiControlMemoryFree

meiControlCycleWait

Declaration

```
long meiControlCycleWait(MPIControl control,
                         long       count)
```

Required Header

stdmei.h

Description

ControlCycleWait waits for the XMP motion controller (***control***) to execute for count background cycles. The host will continuously verify that the XMP firmware is operational, and the host will give up its time slice as it waits (for the controller to execute the background cycles).

Return Values

MPIMessageOK	after the motion controller successfully executes for ***count*** cycles

See Also

mpiControlInit

Declaration `long mpiControlInit(MPIControl control)`

Required Header stdmpi.h

Description **ControlInit** initializes the motion control device ***control***. ControlInit must be called at least once after a control object has been created and before any other **mpiControl** methods are called [with the exception of **mpiControlDelete(...)**].

Return Values

MPIMessageOK	if *ControlInit* successfully initializes the motion control device control

See Also mpiControlDelete

mpiControlInterruptEnable

Declaration

```
long mpiControlInterruptEnable(MPIControl control,
                               long       enable)
```

Required Header stdmpi.h

Description

If “enable” is **TRUE**, then **ControlInterruptEnable** **enables** interrupts from the motion controller.

If "enable" is **FALSE**, then **ControlInterruptEnable** **disables** interrupts from the motion controller.

Return Values	
MPIMessageOK	if *ControlInterruptEnable* successfully enables (or disables) interrupts from the motion controller

See Also mpiControlInteruptWait | mpiControlInteruptWake

mpiControlInterruptWait

Declaration

```
long mpiControlInterruptWait(MPIControl  control,
                             long        *interrupted,
                             MPIWait     timeout)
```

Required Header

stdmpi.h

Description

ControlInterruptWait waits for an interrupt from the motion controller if interrupts are enabled.

After the ControlInterruptWait method returns, if the location pointed to by ***interrupted*** contains **TRUE**, then an interrupt has occurred.

After the ControlInterruptWait method returns, if the location pointed to by ***interrupted*** contains **FALSE**, then no interrupt has occurred, and the return of ControlInterruptWait was caused either by a call to **mpiControlInterruptWake(...)** or by a timeout that has occurred.

If timeout is **MPIWaitFOREVER (-1)**, then *ControlInterruptWait* will wait forever for an interrupt
If timeout is **MPIWaitPOLL (0)**, then *ControlInterruptWait* will return immediately
If timeout is **a value (not 0 or -1)**, then *ControlInterruptWait* will wait for an interrupt for ***timeout*** milliseconds

Return Values

MPIMessageOK	if *ControlInterruptWait* waits for an interrupt from the motion controller

See Also

mpiControlInterruptWake | mpiControlInteruptEnable

mpiControlInterruptWake

Declaration `long mpiControlInterruptWake(MPIControl control)`

Required Header stdmpi.h

Description **ControlInterruptWake** wakes all threads waiting for an interrupt from the motion controller ***control*** [as a result of a call to mpiControlInterruptWait(...)]. The waking thread(s) will return from the call with no interrupt indicated.

Return Values

MPIMessageOK	if *ControlInterruptWake* successfully wakes all threads waiting for an interrupt from the motion controller

See Also mpiControlInterruptWait | mpiControlInteruptEnable

mpiControlReset

Declaration `long mpiControlReset(MPIControl control)`

Required Header stdmpi.h

Description **ControlReset** resets the motion controller (***control***) board.

Return Values	
MPIMessageOK	if *ControlReset* successfully resets the motion controller board

See Also

meiControlSampleWait

Declaration

```
long meiControlSampleWait(MPIControl control,
                          long       count)
```

Required Header stdmpi.h

Description **ControlSampleWait** waits for ***count*** samples while the XMP motion controller (associated with ***control***) executes. While the host waits, the host gives up its time slice and continuously verifies that the XMP firmware is operational.

Return Values	
MPIMessageOK	if *ControlSampleWait* successfully waits for count samples while the XMP motion controller executes

See Also meiControlSamplestoSeconds | meiControlSecondstoSamples | meiControlSampleCounter |

meiControlPlatform

Declaration

`MEIPlatform meiControlPlatform(MPIControl control)`

Required Header

stdmei.h

Description

ControlPlatform returns a handle to the Platform object with which the control is associated.

control	a handle to the Control object

Return Values

MPIPlatform	handle to a Platform object
MPIHandleVOID	if ***control*** is invalid

See Also

mpiControlCreate

MPIControlAddress

MPIControlAddress

```
typedef struct MPIControlAddress {
    long        number; /* controller number */

    union {
        void              *mapped;      /* memory address */
        unsigned long     ioPort;       /* I/O port number */
        char              *device;      /* device driver name */
        struct {
            char                 *name;  /* image file name */
            MPIControlFileType   type;  /* image file type */
        } file;
        struct {
            char        *server;        /* IP address: host.domain.com */
            long        port;           /* socket number */
        } client;
    } type;
} MPIControlAddress;
```

Description

ControlAddress is a structure that specifies the location of the controller that to be accessed when mpiControlCreate() is called. Please refer to the documentation for mpiControlCreate() to see how to use this structure.

number	The controller number in the computer
type	A union that holds information about controllers on non-local computers.

See Also

MPIControl | MPIControlType | mpiControlCreate

MPIControlConfig / MEIControlConfig

MPIControlConfig

```
typedef struct MPIControlConfig {
    long        adcCount;
    long        axisCount;
    long        captureCount;
    long        compareCount;
    long        cmdDacCount;
    long        auxDacCount;
    long        filterCount;
    long        motionCount;
    long        motorCount;
    long        recordCount;
    long        sequenceCount;
    long        sercosCount;
    long        userVersion;
    long        sampleRate;

} MPIControlConfig;
```

Description

adcCount	represents the number of ADC(analog to digital converter) objects configured for the controller.
axisCount	represents the number of axis objects configured for the controller.
captureCount	represents the number of capture objects to be configured for the controller. A capture object manages a single capture in an XMP motion controller. A capture is a hardware latch of a motor position triggered by a motor input. The XMP controller supports ten (10) capture objects per motion block. The default configuration is two capture registers per motor, while the last two (8,9) on each motion block are reserved for the auxiliary encoder (not supported).
compareCount	represents the number of compare objects to be configured for the controller. The XMP controller supports ten (10) compare objects per motion block. The default configuration is two compare registers per motor, while the last two (8,9) on each motion block are reserved for the auxiliary encoder (not supported).
cmdDacCount	represents the number of DAC(digital to analog converter) objects to be configured for the controller. There is one DAC per motor and one auxiliary DAC per motor.
auxDacCount	represents the number of Auxiliary DAC objects to be configured for the controller. There is one DAC per motor and one auxiliary DAC per motor.
filterCount	represents the number of filter objects to be configured for the controller.
motionCount	represents the number of motion supervisor objects to be configured for the controller.
motorCount	represents the number of motor objects to be configured for the controller.
recordCount	represents the number of recorder objects to be configured for the controller. This element allows the application to change the size of the recorder object's data buffer using the mpiControlConfigGet/Set(...) methods. A larger data buffer size can improve the performance of Motion Scope running on a slow host or running in Client/Server mode over a congested network.

sequenceCount	represents the number of sequence objects to be configured for the controller.
sercosCount	represents the number of sercos objects to be configured for the controller.
userVersion	allows the user to mark a firmware image with a user-defineable version number.
sampleRate	represents the number of servo cycles the controller will be configured for. The default value is 2000Hz. This means that one servo cycle takes 0.5milliseconds.

Description

ControlConfig is the structure that contains the controller configuration information.

MEIControlConfig

```
typedef struct MEIControlConfig {
    long                preFilterCount;
    long                compensatorCount;
    MEIXmpPreFilter     PreFilter[MEIXmpMAX_PreFilters];
    MEIXmpCompensator   Compensator[MEIXmpMAX_Compensators];
    long                CompensationTable[MEIXmpCompTableSize];
    MEIXmpUserBuffer    UserBuffer;
} MEIControlConfig;
```

Description

preFilterCount	This value defines the number of enabled pre-filters.
compensatorCount	This value defines the number of enabled compensators.
PreFilter	This array defines the configuration for each pre-filter.
Compensator	This array defines the configuration for each compensator.
CompensationTable	This array defines the compensation values for the compensators.
UserBuffer	This structure defines the controller's user buffer. This is used for custom features that require a controller data buffer.

See Also

mpiControlConfigGet | mpiControlConfigSet | meiControlExtMemAvail | Special Note on Dynamic Allocation of External Memory Buffers.

MPIControlIo

MPIControlIo

```
typedef struct MPIControlIo {
        unsigned long   input[MPIControlIoWords];
        unsigned long   output[MPIControlIoWords];;
} MPIControlIo;
```

Description

ControlIo is used to hold the status of all the controller i/o lines on a board. This does not include any CAN i/o lines.

INPUT	all user i/o inputs are stored in bits of input[0]. User i/o input 0 corresponds to bit 0. Control i/o input 1 corresponds to bit 1.
OUTPUT	all user i/o outputs are stored in bits of output[0]. User i/o output 0 corresponds to bit 0. Control i/o output 1 corresponds to bit 1.

See Also

MEIControlOutput | MEIControlInput | mpiControlIoGet | mpiControlIoSet

MEIControlInput

MEIControlInput

```
typedef enum {
    MEIControlInputUSER_0   = MEIXmpControlIOMaskUSER0_IN,
    MEIControlInputUSER_1   = MEIXmpControlIOMaskUSER1_IN,
    MEIControlInputUSER_2   = MEIXmpControlIOMaskUSER2_IN,
    MEIControlInputUSER_3   = MEIXmpControlIOMaskUSER3_IN,
    MEIControlInputUSER_4   = MEIXmpControlIOMaskUSER4_IN,
    MEIControlInputUSER_5   = MEIXmpControlIOMaskUSER5_IN,
} MEIControlInput;
```

Description **ControlInput** contains bit mask definitions for generic MPIControlIo input words.

See Also MEIControlOutput | MPIControlIo | mpiControlIoGet | mpiControlIoSet

MPIControlMessage / MEIControlMessage

MPIControlMessage

```
typedef enum {
    MPIControlMessageLIBRARY_VERSION,   /* Keep as first control message */
    MPIControlMessageADDRESS_INVALID,
    MPIControlMessageCONTROL_INVALID,
    MPIControlMessageTYPE_INVALID,
    MPIControlMessageINTERRUPTS_DISABLED,
    MPIControlMessageEXTERNAL_MEMORY_OVERFLOW,
    MPIControlMessageADC_COUNT_INVALID,
    MPIControlMessageAXIS_COUNT_INVALID,
    MPIControlMessageCAPTURE_COUNT_INVALID,
    MPIControlMessageCOMPARE_COUNT_INVALID,
    MPIControlMessageCMDDAC_COUNT_INVALID,
    MPIControlMessageAUXDAC_COUNT_INVALID,
    MPIControlMessageFILTER_COUNT_INVALID,
    MPIControlMessageMOTION_COUNT_INVALID,
    MPIControlMessageMOTOR_COUNT_INVALID,
} MPIControlMessage;
```

Description

MPIControlMessageADDRESS_INVALID	Not used.
MPIControlMessageCONTROL_INVALID	Not used.
MPIControlMessageTYPE_INVALID	An invalid control type has been specified.
MPIControlMessageINTERRUPTS_DISABLED	Use of interrupt requested, when interrupts are disabled.

MEIControlMessage

```
typedef enum {
    MEIControlMessageFIRMWARE_INVALID = MEIControlMessageLAST,
    MEIControlMessageFIRMWARE_VERSION,
} MEIControlMessage;
```

Description

MEIControlMessageFIRMWARE_INVALID	This message code occurs when the firmware executing in the controller is not valid. This could be caused by incompatible firmware code, corrupted code, or a hardware problem.
MEIControlMessageFIRMWARE_VERSION	This message code occurs when the firmware version is not compatible with the host library version.

See Also

MEIControlOutput

MEIControlOutput

```
typedef enum {
    MEIControlOutputUSER_0  = MEIXmpControlIOMaskUSER0_OUT,
    MEIControlOutputUSER_1  = MEIXmpControlIOMaskUSER1_OUT,
    MEIControlOutputUSER_2  = MEIXmpControlIOMaskUSER2_OUT,
    MEIControlOutputUSER_3  = MEIXmpControlIOMaskUSER3_OUT,
    MEIControlOutputUSER_4  = MEIXmpControlIOMaskUSER4_OUT,
    MEIControlOutputUSER_5  = MEIXmpControlIOMaskUSER5_OUT,
} MEIControlOutput;
```

Description **ControlOutput** contains bit mask definitions for generic MPIControlIo output words.

See Also MEIControlOutput | MPIControlIo | mpiControlIoGet | mpiControlIoSet

MPIControlType

MPIControlType

```
typedef enum {
    MPIControlTypeDEFAULT,
    MPIControlTypeMAPPED,
    MPIControlTypeIOPORT,
    MPIControlTypeDEVICE,
    MPIControlTypeCLIENT,
    MPIControlTypeFILE,
} MPIControlType;
```

Description

ControlType is an enumeration that specifies the type of controller that needs to be accessed when mpiControlCreate() is called. Please refer to the documentation for mpiControlCreate() to see how to use this enumeration.

See Also

MPIControl | mpiControlCreate | mpiControlType

MEIControlVersion

MEIControlVersion

```
typedef struct MEIControlVersion {
    struct {    /* control.c */
        long    version;        /* MEIControlVersionMPI (YYYYMMDD) */

        struct {        /* xmp.h */
            long        version;        /* MEIXmpVERSION */
            long        option;         /* MEIXmpOPTION */
        } firmware;
    } mpi;

    struct {
        long    version;        /* hardware version */


        struct {        /* MEIXmpData.SystemData{} */
            long        version;        /* MEIXmpVERSION_EXTRACT(SoftwareID) */
            char        revision;       /* ('A' - 1) +
MEIXmpREVISION_EXTRACT(SoftwareID) */
            long        subRevision;    /* MEIXmpSUB_REV_EXTRACT(Option) */
            long        developmentId;  /* MEIXmpDEVELOPMENT_ID_EXTRACT(Option) */
            long        option;         /* MEIXmpOPTION_EXTRACT(Option) */
            long      userVersion;
        } firmware;

        struct  {
            long        FPGA[MEIXmpFPGAsPerBlock];
        } motionBlock[MEIXmpMaxMotionBlocks];

        struct {
            struct      {
                long    version;
                long    option;
            } busInterface;
        } board[MEIXmpMaxBoards];
    } xmp;
} MEIControlVersion;
```

Description

ControlVersion is a structure that specifies the version information for the MPI and the controller's firmware, FPGAs, and the bus interface.

mpi	A structure that contains the version information of the MPI
mpi.version	A string representing the version of the MPI. The version of the MPI is broken down by date, branch, and revision (MPIVersion.branch.revision). For ex: 20021220.1.2 means MPI version 20021220, branch 1, revision 2.
mpi.firmware	The firmware version information that the current version of the MPI will work with. A new field has been added to the XMP's firmware to identify and differentiate between intermediate branch software revisions. The branch value is represented as a hex number between 0x00000000 and 0xFFFFFFFF. Each digit represents an instance of a branch (0x1 to 0xF). A single digit represents a single branch from a specific version, two digits represent a branch of a branch, three digits represent a branch of a branch of a branch, etc.
xmp	A structure that contains the version information of the XMP controller
xmp.firmware	The XMP's firmware version information.

xmp.motionBlock[]	An array of structures that contain version information about the motion blocks on the XMP.
xmp.board	An array of structures that contain version information about the XMP controller boards.

See Also

MPIControl

Dynamic Allocation of External Memory Buffers

In previous versions, the XMP external memory was statically allocated at firmware compile time.

In version 20010119 and later, specific buffers of the XMP external memory are dynamically allocated. The dynamic allocation feature allows an application to efficiently use the XMP controller's on-board memory and allows for future expansion. The dynamically allocated buffers currently include the Frame Buffer, Record Buffer, and SERCOS buffer. Each of these buffers sizes are recalculated during a call to mpiControlConfigSet(...) if there is a change in any of the associated ControlConfig values.

The **Frame Buffer** is used for motion on each axis. The Frame Buffer is directly associated with the number of EnabledAxes in the MPIControlConfig structure. The Frame Buffer will be allocated to the minimum size required to support the number of enabled axes. The default number of EnabledAxes is eight (8).

The **Record Buffer** is used for the on-board data recorder. The Record Buffer is directly associated with the number of EnabledRecord in the MPIControlConfig structure. The Record Buffer will be allocated to the minimum size required to support the number of enabled records. The default number of EnabledRecords is 3064. Each record is the size of one memory word.

The **Sercos Buffer** is used for motion on each SERCOS ring network. The Sercos Buffer is directly associated with the number of EnabledSercos in the MPIControlConfig structure. The Sercos Buffer will be allocated to the minimum size required to support the number of enabled Sercos rings. The default number of EnabledSercosRings, for a non-sercos controller is zero (0).

The meiControlExtMemAvail(...) method has been added to discover how much memory is available on your controller.

```
MPI_DEF1 long MPI_DEF2
     meiControlExtMemAvail(MPIControl  control,
                           long        *size)
```

The meiControlExtMemAvail(...) method will return the number of memory words available. Since each record size is one memory word, the size returned from the above function can be used to increase the Record Buffer to maximum size possible. This greatly improves client/server operation of Motion Scope and any application used for data collection.

WARNING! Due to the nature of dynamic allocation and the clearing of external memory buffers mpiControlConfigSet(...) should ONLY be called at motion application initialization time and NOT during motion.

Return to Control Objects page

Copyright © 2002
Motion Engineering

TCP/IP and Sockets for Control Objects

The MPI implements network functionality as client/server. The xmp\util\server.c program implements a basic server. You just create a Control object of type MPIControlTypeCLIENT and specify the server's host in the MPIControlAddress{}.client{} structure.

You can try “MPI networking” on a single machine by starting up the server program in a DOS window, and then running a sample application in another DOS window. Note that you can specify the host name and port of the server as command line arguments to all sample applications and utilities.

The way the MPI client/server works internally is that low-level mpiControlMemory and mpiControlInterrupt methods are intercepted just before they read/write XMP memory. The methods are packaged up as remote procedure calls and sent to the server for execution. The server sends the results back to the client.

There are 2 channels of communication - one channel to wait for interrupts, and another channel to do everything else. All MPI methods that communicate with the XMP do so by calling (eventually) the low level mpiControlMemory methods, so no application code needs to be changed other than the initial call to mpiControlCreate(...). This is all implemented on WinNT using WinSock.

Note that it would be possible to implement the client/server scenario above using an RS-232 line rather than TCP/IP WinSock. The MPI’s client/server protocol only requires a reliable transport mechanism (WinSock, RS-232) between a client and server.

Return to Control Objects page

Copyright © 2002
Motion Engineering

Event Objects

Introduction

An **Event** object contains information about an asynchronous event. Typically, events are generated by the controller, but in some special cases it is possible to generate events from the host computer.

The Event object is retrieved through the EventMgr, via the Notify object. The Event object contains data about the type of event, its source, and other information. The user Event fields can be configured to collect data at the time when the event occurs in the controller.

Methods

Configuration and Information Methods

mpiEvent**StatusGet**	Get Event status
mpiEvent**StatusSet**	Set Event status

Data Types

MPIEvent**Message**
MEIEvent**NotifyData**
MPIEvent**Status**
MEIEvent**StatusInfo**
MPIEvent**Type** / MEIEvent**Type**

Constants

MPIEvent**StatusINFO_COUNT_MAX**	defines the size of the MPIEventStatus.info[] array.

Copyright © 2002
Motion Engineering

mpiEventStatusGet

Declaration

```
long mpiEventStatusGet(MPIEvent       event,
                       MPIEventStatus *status)
```

Required Header event.h

Description **EventStatusGet** gets the status of an Event object (***event***) and writes it into the structure pointed to by ***status***. Event status includes the event type, type-specific codes and the event source.

Return Values	
MPIMessageOK	if *EventStatusGet* successfully gets the status of an Event object and writes it into the structure

See Also mpiEventStatusSet | meiEventStatusInfo

mpiEventStatusSet

Declaration

```
long mpiEventStatusSet(MPIEvent       event,
                       MPIEventStatus *status)
```

Required Header

event.h

Description

EventStatusSet sets (writes) the status of ***event*** using data from the structure pointed to by ***status***. Event status includes the event type, type-specific codes and the event source.

Return Values

MPIMessageOK	if *EventStatusSet* successfully sets (writes) the status of event using data from the structure

See Also

mpiEventStatusGet | meiEventStatusInfo

MPIEventMessage

MPIEventMessage

```
typedef enum {
    MPIEventMessageEVENT_INVALID,
} MPIEventMessage;
```

Description

MPIEventMessageEVENT_INVALID	
Meaning	The MPIEvent handle passed to an MPIEvent method is invalid.
Possible Causes	Either the handle was never initialized or the mpiEventCreate method failed.
Recommendations	MPIEvent objects are used internally. **Do not create your own.** If you have not and you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

See Also

MEIEventNotifyData

MPIEventNotifyData

```
typedef struct MEIEventNotifyData {
    void        *address[MEIXmpSignalUserData];
} MEIEventNotifyData;
```

Description

The ***address*** of an **EventNotifyData** structure is passed as the third (void *external) argument to mpi'Object'EventNotify[GS]et(). The address array contains host-based XMP addresses, the contents of which are returned in MEIEventStatusInfo{}.data.

See Also

MEIEventStatusInfo

MPIEventStatus

MPIEventStatus

```
typedef struct MPIEventStatus {
    MPIEventType        type;
    void                *source;

    long                info[MPIEventStatusINFO_COUNT_MAX];
} MPIEventStatus;
```

Description

EventStatus holds information about a particular event that was generated by the XMP.

type	identifies the type of event that was generated.
***source**	identifies what the source of the event was. source will either be a handle to an MPI object or a host pointer. Use mpiObjectModuleId() to identify what source points to.

See Also

mpiObjectModuleId | MPIEventType | MPIEventMgr | MPINotify | MEIEventStatusInfo | MPIEventStatusINFO_COUNT_MAX

MEIEventStatusInfo

MEIEventStatusInfo

```
typedef struct MEIEventStatusInfo {
    union {
        MPIHandle    handle;    /* generic */
        MPIAxis      axis;      /* MEIEventTypeAXIS_FIRST ... MEIEventTypeAXIS_LAST -
1 */
        long         number;    /* MPIEventTypeMOTION
                    MPIEventTypeMOTOR_FIRST ... MPIEventTypeMOTOR_LAST - 1
                    MEIEventTypeMOTOR_FIRST ... MEIEventTypeMOTOR_LAST - 1 */
        long         value;     /* MPIEventTypeEXTERNAL */
            } type;

            MEIXmpSignalID signalID;

            /* Contents of addresses specified by MEIEventNotifyData{} */
            union {
                  long sampleCounter;
                  struct {
                      long sampleCounter;
                  } motion;
                  struct {
                       long sampleCounter;
                       long actualPosition;
                  } axis;
                  struct {
                       long sampleCounter;
                       long encoderPosition;
                  } motor;
                  long word[MEIXmpSignalUserData];
             } data;
} MEIEventStatusInfo;
```

Description

EventStatusInfo is an information structure that tells the XMP what the data in MPIEventStatus.info holds.

type	A union that specifies the object handle, motion number, or external ID value that generated the event
type.handle	A generic object handle. Used by MPIRecorder and MPIMotor events
type.axis	An axis object handle. Used by MPIAxis events
type.number	The motion number of the MPIMotion object that generated the event
type.value	An ID value used to identify what external source or MPISequence event was generated
signalID	Specifies what type of object actually generated the event
data	A union that contains extra data about the event that was generated
data.sampleCounter	The value of the sampleCounter when the event was generated
data.motion	A union that contains extra data about the motion event that was generated
data.motion.sampleCounter	The value of the sampleCounter when the motion event was generated
data.axis	A union that contains extra data about the axis event that was generated
data.axis.sampleCounter	The value of the sampleCounter when the axis.event was generated
data.axis.actualPosition	The value of the axis' actual position when the event was generated
data.motor	A union that contains extra data about the motor event that was generated

data.motor.sampleCounter	The value of the sampleCounter when the motor event was generated
data.motor.encoderPosition	The value of the motor's ecoder position when the event was generated
data.word[]	The extra data about the event that was generated formatted as an array of long values

Sample Code

```
MPINotify      notify
MPIEventStatus eventStatus;

. . .

/* Wait for event */
returnValue =
   mpiNotifyEventWait(notify,
                      &eventStatus,
                       MPIWaitFOREVER);
msgCHECK(returnValue);

if (eventStatus.type == MPIEventTypeMOTION_DONE) {
   MEIEventStatusInfo *info;

   info = (MEIEventStatusInfo *)eventStatus.info;

   . . .
}
```

See Also

MPIEventStatus | MPIAxis

MPIEventType / MEIEventType

MPIEventType

```
typedef enum {
    MPIEventTypeINVALID,

    MPIEventTypeNONE,           /*  0 */

    /* Motor events */
    MPIEventTypeAMP_FAULT,      /*  1 */
    MPIEventTypeHOME,           /*  2 */
    MPIEventTypeLIMIT_ERROR,    /*  3 */
    MPIEventTypeLIMIT_HW_NEG,   /*  4 */
    MPIEventTypeLIMIT_HW_POS,   /*  5 */
    MPIEventTypeLIMIT_SW_NEG,   /*  6 */
    MPIEventTypeLIMIT_SW_POS,   /*  7 */
    MPIEventTypeENCODER_FAULT,  /*  8 */

    /* Motion events */
    MPIEventTypeMOTION_DONE,            /*  9 */
    MPIEventTypeMOTION_AT_VELOCITY,     /* 10 */

    /* Recorder events */
    MPIEventTypeRECORDER_FULL,  /* 11 */
    MPIEventTypeRECORDER_DONE,  /* 12 */

    /* External events */
    MPIEventTypeEXTERNAL,       /* 13 */
} MPIEventType;
```

Description

EventType is used by the MPIEventMask macros to help generate event masks.

MPIEventTypeNONE	This event type indicates no event was generated.
MPIEventTypeAMP_FAULT	This event type indicates an Amp Fault event was generated from a Motor object.
MPIEventTypeHOME	This event type indicates a Home event was generated from a Motor object.
MPIEventTypeLIMIT_ERROR	This event type indicates a position Error Limit was generated from a Motor object.
MPIEventTypeLIMIT_HW_NEG	This event type indicates a Negative Hardware Limit event was generated from a Motor object.
MPIEventTypeLIMIT_HW_POS	This event type indicates a Positive Hardware Limit event was generated from a Motor object.
MPIEventTypeLIMIT_SW_NEG	This event type indicates a Negative Software Limit event was generated from a Motor object.
MPIEventTypeLIMIT_SW_POS	This event type indicates a Positive Software Limit event was generated from a Motor object.
MPIEventTypeENCODER_FAULT	This event type indicates an Encoder Fault event was generated from a Motor object.

MPIEventTypeMOTION_DONE	This event type indicates a Motion Done event was generated from a Motion Supervisor object.
MPIEventTypeMOTION_AT_VELOCITY	This event type indicates an At Velocity event was generated from a Motion Supervisor object.
MPIEventTypeRECORDER_FULL	This event type indicates a Recorder Full event was generated from a Recorder object.
MPIEventTypeRECORDER_DONE	This event type indicates a Recorder Done event was generated from a Recorder object.
MPIEventTypeEXTERNAL	This event type indicates an External event was generated from an external source.

MEIEventType

```
typedef enum {
        /* Motor events */
        MEIEventTypeLIMIT_USER0 = MPIEventTypeLAST,     /* 14 */
        MEIEventTypeLIMIT_USER1,        /* 15 */
        MEIEventTypeLIMIT_USER2,        /* 16 */
        MEIEventTypeLIMIT_USER3,        /* 17 */
        MEIEventTypeLIMIT_USER4,        /* 18 */
        MEIEventTypeLIMIT_USER5,        /* 19 */
        MEIEventTypeLIMIT_USER6,        /* 20 */
        MEIEventTypeLIMIT_USER7,        /* 21 */
        MEIEventTypeLIMIT_TORQUE,       /* 22 */

        /* Motion events */
        MEIEventTypeMOTION_OUT_OF_FRAMES,       /* 23 */
        MEIEventTypeMOTION_RESERVED0,           /* 24 */

        /* Axis events */
        MEIEventTypeIN_POSITION_COARSE,         /* 25 */
        MEIEventTypeIN_POSITION_FINE,           /* 26 */
        MEIEventTypeAT_TARGET,                  /* 27 */
        MEIEventTypeFRAME,                      /* 28 */
        MEIEventTypeAXIS_RESERVED0,             /* 29 */
        MEIEventTypeAXIS_RESERVED1,             /* 30 */
        MEIEventTypePOWER_FAILURE,              /* 31 */

} MEIEventType;
```

Description

EventType is used by the MPIEventMask macros to help generate event masks.

MEIEventTypeLIMIT_USER0	This event type indicates a User Limit event was generated from a Motor object.
MEIEventTypeLIMIT_USER1	This event type indicates a User Limit event was generated from a Motor object.
MEIEventTypeLIMIT_USER2	This event type indicates a User Limit event was generated from a Motor object.
MEIEventTypeLIMIT_USER3	This event type indicates a User Limit event was generated from a Motor object.
MEIEventTypeLIMIT_USER4	This event type indicates a User Limit event was generated from a Motor object.
MEIEventTypeLIMIT_USER5	This event type indicates a User Limit event was generated from a Motor object.
MEIEventTypeLIMIT_USER6	This event type indicates a User Limit event was generated from a Motor object.
MEIEventTypeLIMIT_USER7	This event type indicates a User Limit event was generated from a Motor object.
MEIEventTypeLIMIT_TORQUE	This event type indicates a Torque event was generated from a Motor object.
MEIEventTypeMOTION_OUT_OF_FRAMES	This event type indicates a Motion Done event was generated from a Motion Supervisor object.
MEIEventTypeMOTION_RESERVED0	This event type indicates a Reserved Motion event was generated from a Motion Supervisor object. This event type is reserved for future use or custom motion events.
MEIEventTypeIN_POSITION_COARSE	This event type indicates an In Coarse Position event was generated from an Axis object.
MEIEventTypeIN_POSITION_FINE	This event type indicates an In Fine Position event was generated from an Axis object.
MEIEventTypeAT_TARGET	Rserved Frame Event.
MEIEventTypeFRAME	This is currently not supported and is reserved for future use.
MEIEventTypeAXIS_RESERVED0	This event type indicates a Reserved Axis event was generated from an Axis object. This event type is reserved for future use or custom axis events.
MEIEventTypeAXIS_RESERVED1	This event type indicates a Reserved Axis event was generated from an Axis object. This event type is reserved for future use or custom axis events.
MEIEventTypePOWER_FAILURE	This event type indicates a Power Failure event was generated from the controller.

See Also

MPIEventMask | MPIEventMgr | MPINotify | MPIEventStatus
Special Note on the use of MPIEventTypeENCODER_FAULT

MPIEventStatusINFO_COUNT_MAX

MPIEventStatusINFO_COUNT_MAX

```
#define MPIEventStatusINFO_COUNT_MAX    (16)
```

Description **EventStatusINFO_COUNT_MAX** defines the size of the MPIEventStatus.info[] array.

See Also MPIEventStatus | MPIEventMgr | MPINotify

Special Note: *Use of MPIEventTypeENCODER_FAULT*

This event type is used to detect three types of encoder faults:

- **Broken wire errors**
- **Illegal state errors**
- **Absolute encoder initialization errors**
  - Timeout errors
  - Protocol errors

Broken wire errors are detected for either incremental or absolute encoders whenever both differential inputs of any encoder receiver (A, B, or Index) are at the same voltage level (i. e., whenever one or both inputs is disconnected from the encoders differential transmitter). The EncoderTermination configuration of the encoder input must be TRUE for correct detection of broken wires.

Illegal state errors occur whenever transitions are seen on both A and B phases of an encoder input at the same time (e.g. noise spikes).

There are two types of **absolute encoder initialization errors**: Timeout errors and Protocol errors. **Timeout errors** occur when an absolute encoder does not transmit absolute encoder data within the timeout period starting at the transition of the interrogation line (SEN line). **Protocol errors** are detected when serial absolute data is sent during the timeout, but the data cannot be interpreted by the XMP. Both error types result in an ENCODER_FAULT event.

Return to MPIEventType

Copyright © 2002
Motion Engineering

EventMask Objects

Introduction

The **EventMask** determines which event types will be sent from the controller to the Notify object. By setting bits in the EventMask, specific events can be sent to the Notify object when they occur or by clearing specific bits, the events can be ignored. By default, no Events will occur until the EventMask is set.

The Event, EventMgr, and Notify objects are used with the EventMask to handle controller events.

Data Types

MPIEventMask	An array of unsigned longs

Macros

mpiEventMask**ALL** / meiEventMask**ALL**	Set all events within the event mask.
mpiEventMask**AND_ASSIGN**	Assign dst all events masked by both *src* and *dst*.
mpiEventMask**ASSIGN**	Assign the value of event mask *src* to the event mask *dst*.
mpiEventMask**AXIS** / meiEventMask**AXIS**	Set all MPIAxis events within the event mask.
mpiEventMask**BIT**	Set mask to only handle events of type *type*.
mpiEventMask**BITS_IN_ELEMENT**	Define the number of bits in each data element of MPIEventMask.
mpiEventMask**BitGET**	Reports if a mask is set to handle events of type type.
mpiEventMask**BitSET**	Sets mask to handle events of type *type*.
mpiEventMask**BIT_POSITION_MASK**	Returns an element's bit-mask for the specified event type.
mpiEventMask**CLEAR**	Set mask to handle no events.
mpiEventMask**COMPLEMENT**	Change the value of every bit within the event mask.
mpiEventMask**EXTERNAL**	Set external events within the event mask.
mpiEventMask**GET**	Reports if a mask is set to handle events of type *type*.
mpiEventMask**IS_EQUAL**	Tests the equality of two event masks.
mpiEventMask**MOTION** / meiEventMask**MOTION**	Set all MPIMotion events within the event mask.
mpiEventMask**MOTOR** / meiEventMask**MOTOR**	Set all MPIMotor events within the event mask.
mpiEventMask**OR_ASSIGN**	Add all events masked by src to the event mask dst.
mpiEventMask**RECORDER**	Sets all MPIRecorder events within the event mask.
mpiEventMask**SET**	Set mask to handle events of type *type*.
mpiEventMask**SET_ALL**	Set mask to handle all events whose type is enumerated less than type.

Constants

MPIEventMask**ELEMENTS**	Define the number of data elements in a MPIEventMask
MPIEventMask**ELEMENT_TYPE**	Define what the data type MPIEventMask is comprised of.

Copyright © 2002
Motion Engineering

MPIEventMask

MPIEventMask

```
#define MPIEventMaskELEMENTS    (2)
typedef unsigned long MPIEventMaskELEMENT_TYPE;
typedef MPIEventMaskELEMENT_TYPE MPIEventMask[MPIEventMaskELEMENTS];
```

Description

EventMask is an array of unsigned longs, with a length defined by MPIEventMaskELEMENTS. Each bit in the array represents a mask for a particular event. Be sure to always use the mpiEventMask…(…) macros to set or clear the event masks.

See Also

MPIEventType | mpiEventMaskBIT

mpiEventMaskALL / meiEventMaskALL

mpiEventMaskALL

Declaration

```
#define mpiEventMaskALL(mask),mpiEventMaskSET_ALL((mask)
                                    MPIEventTypeLAST)
```

Required Header stdmpi.h

Description **EventMaskALL** sets all events within the event mask.

meiEventMaskALL

Declaration

```
#define meiEventMaskALL(mask)
 mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER0, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER1, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER2, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER3, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER4, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER5, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER6, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER7, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeMOTION_OUT_OF_FRAMES, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeMOTION_RESERVED0, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeIN_POSITION_COARSE, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeIN_POSITION_FINE, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeAT_TARGET, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeFRAME, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeAXIS_RESERVED0, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeAXIS_RESERVED1, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypePOWER_FAILURE, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_TORQUE, 1)
```

Required Header stdmei.h

Description **EventMaskALL** sets all MEI events within the event mask

See Also MPIEventType

mpiEventMaskAND_ASSIGN

Declaration

```
#define mpiEventMaskAND_ASSIGN(dst,src)((dst)[0] = ((dst)[0] & (src)[0])), \
```

Required Header

stdmpi.h

Description

EventMaskOR_ASSIGN assigns ***dst*** all events masked by both ***src*** and ***dst***.

See Also

MPIEventMask

mpiEventMaskASSIGN

Declaration `#define mpiEventMaskASSIGN(dst,src)((dst)[0] = (src)[0]), \`

Required Header stdmpi.h

Description **EventMaskASSIGN** assigns the value of event mask ***src*** to the event mask ***dst***.

See Also MPIEventMask

mpiEventMaskAXIS / meiEventMaskAXIS

mpiEventMaskAXIS

Declaration `#define mpiEventMaskAXIS(mask)(mask)`

Required Header stdmpi.h

Description **EventMaskAXIS** sets all MPIAxis events within the event mask.

meiEventMaskAXIS

Declaration

```
#define meiEventMaskAXIS(mask)
        mpiEventMaskBitSET((mask), MEIEventTypeIN_POSITION_COARSE, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeIN_POSITION_FINE, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeAT_TARGET, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeFRAME, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeAXIS_RESERVED0, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeAXIS_RESERVED1, 1)
```

Required Header stdmei.h

Description **EventMaskAXIS** sets all MEIAxis events within the event mask.

See Also MPIEventMask

mpiEventMaskBIT

Declaration

```
#define mpiEventMaskBIT(mask,type)(((mask)[0] = 0); ((mask)[1] = 0); \
((mask)[mpiEventMaskWORD(type)]= mpiEventMaskBIT_POSITION_MASK(type)))
```

Required Header

stdmpi.h

Description

EventMaskBIT sets mask to only handle events of type ***type***.

See Also

MPIEventMask | mpiEventMaskBIT | mpiEventMaskBitGET | mpiEventMaskBitSET | mpiEventMaskBITS_IN_ELEMENT | mpiEventMaskBIT_POSITION_MASK

mpiEventMaskBITS_IN_ELEMENT

mpiEventMaskBITS_IN_ELEMENT

```
#define mpiEventMaskBITS_IN_ELEMENT ((unsigned long)
(sizeof(MPIEventMaskELEMENT_TYPE) * 8))
```

Description **EventMaskBITS_IN_ELEMENT** defines the number of bits in each data element of MPIEventMask.

See Also MPIEventMask | mpiEventMaskBIT | mpiEventMaskBitGET | mpiEventMaskBitSET | mpiEventMaskBIT_POSITION_MASK

mpiEventMaskBitGET

Declaration

```
#define mpiEventMaskBitGET(mask,bit)(((mask)[mpiEventMaskWORD(bit)] &
        mpiEventMaskBIT_POSITION_MASK(bit)) ? 1 : 0)
```

Required Header

stdmpi.h

Description

EventMaskAXIS sets all MPIAxis events within the event mask.

Returns

TRUE if mask is set to handle events of type ***type***. Otherwise, it returns FALSE.

See Also

MPIEventMask | mpiEventMaskBIT | mpiEventMaskBitSET | mpiEventMaskBIT_POSITION_MASK | mpiEventMaskBITS_IN_ELEMENT

mpiEventMaskBitSET

Declaration

```
#define mpiEventMaskBitSET(mask,bit,value)      (((value) == 0) ? \
        ((mask)[mpiEventMaskWORD(bit)] &= ~mpiEventMaskBIT_POSITION_MASK(bit))
              : \
        ((mask)[mpiEventMaskWORD(bit)] |= mpiEventMaskBIT_POSITION_MASK(bit))))
```

Required Header

stdmpi.h

Description

EventMaskBitSET sets mask to handle events of type ***type***.

See Also

MPIEventMask | mpiEventMaskBitGET | mpiEventMaskBIT | mpiEventMaskBIT_POSITION_MASK | mpiEventMaskBITS_IN_ELEMENT

mpiEventMaskBIT_POSITION_MASK

Declaration

```
#define mpiEventMaskBIT_POSITION_MASK (type)
                    ((MPIEventMaskELEMENT_TYPE)(0x1 <<
                      mpiEventMaskBIT_POSITION(type)))
```

Required Header stdmpi.h

Description

type	the event type (MPIEventType / MEIEventType)

Returns an element's bit-mask for the specified event type.

See Also MPIEventMask | mpiEventMaskBIT | mpiEventMaskBitGET | mpiEventMaskBitSET | mpiEventMaskBITS_IN_ELEMENT

mpiEventMaskCLEAR

Declaration

```
#define mpiEventMaskCLEAR(mask)((mask)[0] = 0), \ ((mask)[1] = 0)
```

Required Header

stdmpi.h

Description

EventMaskCLEAR sets mask to handle no events.

See Also

MPIEventMask

mpiEventMaskCOMPLEMENT

Declaration

```
#define mpiEventMaskCOMPLEMENT(mask)((mask)[0] = ~(mask)[0]),
                                \ ((mask)[1] = ~(mask)[1])
```

Required Header

stdmpi.h

Description

EventMaskCOMPLEMENT() changes the value of every bit within the event mask. If the ***mask*** is origininally set to handle an event, then ***mask*** will be set to not handle the event after the call to mpiEventMaskCOMPLEMENT(). Likewise, If the ***mask*** is origininally set to not handle an event, then mask will be set to handle the event after the call to mpiEventMaskCOMPLEMENT().

See Also

MPIEventMask

mpiEventMaskEXTERNAL

Declaration

```
#define mpiEventMaskEXTERNAL(mask)mpiEventMaskBitSET((mask),
                                    MPIEventTypeEXTERNAL, 1)
```

Required Header stdmpi.h

Description **EventMaskEXTERNAL** sets external events within the event mask.

See Also MPIEventMask

mpiEventMaskGET

Declaration

```
#define mpiEventMaskGET(mask,type)
mpiEventMaskBitGET(mask,type)
```

Required Header

stdmpi.h

Description

Returns	TRUE if mask is set to handle events of type ***type***. Otherwise, it returns FALSE.

See Also

MPIEventMask | mpiEventMaskSET

mpiEventMaskIS_EQUAL

Declaration

```
#define mpiEventMaskIS_EQUAL(mask1,mask2)
        (((mask1)[0] == (mask2)[0]) &&
        ((mask1)[1] == (mask2)[1])))
```

Required Header stdmpi.h

Description **EventMaskIS_EQUAL** tests the equality of two event masks.

Returns TRUE if mask1 is the same as mask2. Otherwise it returns FALSE.

See Also MPIEventMask

mpiEventMaskMOTION / meiEventMaskMOTION

mpiEventMaskMOTION

Declaration

```
#define mpiEventMaskMOTION(mask)
        mpiEventMaskBitSET((mask), MPIEventTypeMOTION_DONE, 1), \
        mpiEventMaskBitSET((mask), MPIEventTypeMOTION_AT_VELOCITY, 1)
```

Required Header

stdmpi.h

Description

EventMaskMOTION sets all MPIMotion events within the event mask.

meiEventMaskMOTION

Declaration

```
#define meiEventMaskMOTION(mask)        mpiEventMaskBitSET((mask),
        MEIEventTypeMOTION_OUT_OF_FRAMES, 1), \
        mpiEventMaskBitSET((mask), MEIEventTypeMOTION_RESERVED0, 1)
```

Required Header

stdmei.h

Description

EventMaskMOTION sets all MEIMotion events within the event mask.

See Also

MPIEventMask

mpiEventMaskMOTOR / meiEventMaskMOTOR

mpiEventMaskMOTOR

Declaration

```
#define mpiEventMaskMOTOR(mask)
                mpiEventMaskBitSET((mask), MPIEventTypeAMP_FAULT,1), \
                mpiEventMaskBitSET((mask), MPIEventTypeHOME, 1), \
                mpiEventMaskBitSET((mask), MPIEventTypeLIMIT_ERROR, 1), \
                mpiEventMaskBitSET((mask), MPIEventTypeLIMIT_HW_NEG, 1), \
                mpiEventMaskBitSET((mask), MPIEventTypeLIMIT_HW_POS, 1), \
                mpiEventMaskBitSET((mask), MPIEventTypeLIMIT_SW_NEG, 1), \
                mpiEventMaskBitSET((mask), MPIEventTypeLIMIT_SW_POS, 1), \
                mpiEventMaskBitSET((mask), MPIEventTypeENCODER_FAULT, 1)
```

Required Header stdmpi.h

Description **EventMaskMOTOR** sets all MPIMotor events within the event mask.

meiEventMaskMOTOR

Declaration

```
#define meiEventMaskMOTOR(mask)
                mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER0, 1), \
                mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER1, 1), \
                mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER2, 1), \
                mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER3, 1), \
                mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER4, 1), \
                mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER5, 1), \
                mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER6, 1), \
                mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_USER7, 1), \
                mpiEventMaskBitSET((mask), MEIEventTypePOWER_FAILURE, 1), \
                mpiEventMaskBitSET((mask), MEIEventTypeLIMIT_TORQUE, 1)
```

Required Header stdmei.h

Description **EventMaskMOTOR** sets all MEIMotor events within the event mask.

See Also MPIEventMask

mpiEventMaskOR_ASSIGN

Declaration

```
#define mpiEventMaskOR_ASSIGN(dst,src)  ((dst)[0] = ((dst)[0] | (src)[0])), \
                                        ((dst)[1] = ((dst)[1] | (src)[1]))
```

Required Header

stdmpi.h

Description

EventMaskOR_ASSIGN adds all events masked by ***src*** to the event mask ***dst***.

See Also

MPIEventMask

mpiEventMaskRECORDER

Declaration

```
#define mpiEventMaskRECORDER(mask)      mpiEventMaskBitSET((mask),
        MPIEventTypeRECORDER_FULL, 1), \
        mpiEventMaskBitSET((mask), MPIEventTypeRECORDER_DONE, 1)
```

Required Header

stdmpi.h

Description

EventMaskRECORDER sets all MPIRecorder events within the event ***mask***.

See Also

MPIEventMask

mpiEventMaskSET

Declaration

```
#define mpiEventMaskSET(mask,type)
        mpiEventMaskBitSET(mask,type,1)
```

Required Header stdmpi.h

Description **mpiEventMaskSET** sets mask to handle events of type ***type***.

See Also MPIEventMask | mpiEventMaskGET

mpiEventMaskSET_ALL

Declaration

```
#define mpiEventMaskSET_ALL(mask,type)  ((mpiEventMaskWORD(type) == 0) ? \
                        ((mask)[0] |= mpiEventMaskBIT_POSITION_MASK(type) - 1) : \
                        ((mask)[0] |= 0xFFFFFFFF)), \
                        ((mpiEventMaskWORD(type) == 0) ? \
                        ((mask)[1] |= 0x0) : \
                        ((mask)[1] |= mpiEventMaskBIT_POSITION_MASK(type) - 1))
```

Required Header

stdmpi.h

Description

EventMaskSET_ALL sets ***mask*** to handle all events whose type is enumerated less than ***type***.

See Also

MPIEventMask

MPIEventMaskELEMENTS

MPIEventMaskELEMENTS

```
#define MPIEventMaskELEMENTS    (2)
```

Description

EventMaskELEMENTS defines the number of data elements in a MPIEventMask.

See Also

MPIEventMask

MPIEventMaskELEMENT_TYPE

MPIEventMaskELEMENT_TYPE

```
typedef unsigned long MPIEventMaskELEMENT_TYPE
```

Description

EventMaskELEMENT_TYPE defines what the data type MPIEventMask is comprised of.

See Also

MPIEventMask

EventMgr Objects

Introduction

An **EventMgr**

- Obtains asynchronous events from the Control object(s) that the EventMgr is associated with
- Generates Event objects for enabled event sources
- Awakens any threads that are waiting for events.

Methods

Create, Delete, Validate Methods

mpiEventMgr**Create**	Create EventMgr object
mpiEventMgr**Delete**	Delete EventMgr object
mpiEventMgr**Validate**	Validate EventMgr object

Configuration and Information Methods

mpiEventMgr**ConfigGet**	Get EventMgr config
mpiEventMgr**ConfigSet**	Set EventMgr config
mpiEventMgr**Event**	Request event notification for all Notify objects on EventMgr's list
meiEventMgr**ServiceConfigGet**	Get processes that EventMgr will service
meiEventMgr**ServiceConfigSet**	Set processes that EventMgr will service

Action Methods

mpiEventMgr**Flush**	Flush pending EventMgr events
mpiEventMgr**Service**	Get list of all pending asynchronous events

Relational Methods

List Methods- for Control Objects

mpiEventMgr**Control**	Return handle of indexth Control object in list
mpiEventMgr**ControlAppend**	Append Control's handle to list
mpiEventMgr**ControlCount**	Count the number of Control objects associated with EventMgr (in list)
mpiEventMgr**ControlFirst**	Return handle to first Control object in list
mpiEventMgr**ControlIndex**	Return the index of a Control object in list
mpiEventMgr**ControlInsert**	Insert Control handle into list
mpiEventMgr**ControlLast**	Get handle to last Control object in list
mpiEventMgr**ControlListGet**	Get list of Control objects associated with EventMgr
mpiEventMgr**ControlListSet**	Create a list of Control objects associated with EventMgr
mpiEventMgr**ControlNext**	Get handle to next Control object in list
mpiEventMgr**ControlPrevious**	Get handle to previous Control object in list
mpiEventMgr**ControlRemove**	Remove a Control object's handle from list

List Methods- for Notify Objects

mpiEventMgr**Notify**	Return handle to a Notify object associated with EventMgr

mpiEventMgr**NotifyAppend**	Append Notify object to list
mpiEventMgr**NotifyCount**	Return number of Notify objects in list
mpiEventMgr**NotifyFirst**	Get first Notify object in list
mpiEventMgr**NotifyIndex**	Get index value for a Notify object in list
mpiEventMgr**NotifyInsert**	Place a Notify object after another Notify object in list
mpiEventMgr**NotifyLast**	Get handle to the Notify object that is last on the list
mpiEventMgr**NotifyListGet**	Get a list of Notify objects
mpiEventMgr**NotifyListSet**	Create a list of Notify objects
mpiEventMgr**NotifyNext**	Get the Notify object just after notify in list
mpiEventMgr**NotifyPrevious**	Get the Notify object just before notify in list
mpiEventMgr**NotifyRemove**	Remove a Notify object from list

Data Types

MPIEventMgr**Message**
MEIEventMgr**ServiceConfig**

Copyright © 2002
Motion Engineering

mpiEventMgrCreate

Declaration

```
const MPIEventMgr mpiEventMgrCreate(MPIControl control)
```

Required Header

stdmpi.h

Description

EventMgrCreate creates an EventMgr object, with ***control*** as the initial element in the list of Control objects from which the EventMgr obtains asynchronous events (***control*** may be MPIHandleVOID).

EventMgrCreate is the equivalent of a C++ constructor.

Return Values

handle	to an EventMgr object
MPIHandleVOID	if the object could not be created

See Also

mpiEventMgrDelete | mpiEventMgrValidate

mpiEventMgrDelete

Declaration

```
long mpiEventMgrDelete(MPIEventMgr eventMgr)
```

Required Header

stdmpi.h

Description

EventMgrDelete deletes an EventMgr object and invalidates its handle (***eventMgr***). EventMgrDelete is the equivalent of a C++ destructor.

Deleting an EventMgr object does not delete any of the Control objects that supply the EventMgr with asynchronous events. However, deleting an EventMgr object will delete any unreceived events for that EventMgr.

Return Values

MPIMessageOK	if *EventMgrDelete* successfully deletes an EventMgr object and invalidates its handle

See Also

mpiEventMgrCreate | mpiEventMgrValidate

mpiEventMgrValidate

Declaration `long mpiEventMgrValidate(MPIEventMgr eventMgr)`

Required Header stdmpi.h

Description **EventMgrValidate** validates an EventMgr object and its handle (***eventMgr***).

Return Values	
MPIMessageOK	if EventMgr is a handle to a valid object.

See Also mpiEventMgrCreate | mpiEventMgrDelete

mpiEventMgrConfigGet

Declaration

```
long mpiEventMgrConfigGet(MPIEventMgr       eventMgr,
                          MPIEventMgrConfig *config,
                          void              *external)
```

Required Header

stdmpi.h

Description

EventMgrConfigGet gets the configuration of an ***EventMgr*** object (eventMgr) and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only

external either points to a structure of type **MEIEventMgrConfig{}** or is NULL.

Return Values

MPIMessageOK	if *EventMgrConfigGet* successfully gets the EventMgr's configuration and writes it into the structure(s)

See Also

mpiEventMgrConfigSet

mpiEventMgrConfigSet

Declaration

```
long mpiEventMgrConfigSet(MPIEventMgr       eventMgr,
                          MPIEventMgrConfig *config,
                          void              *external)
```

Required Header

stdmpi.h

Description

EventMgrConfigSet sets (writes) the ***eventMgr*** configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only — ***external*** either points to a structure of type **MEIEventMgrConfig{}** or is NULL.

Return Values

MPIMessageOK	if *EventMgrConfigSet* successfully sets (writes) the EventMgr's configuration using data from the structure(s)

See Also

mpiEventMgrConfigGet

mpiEventMgrEvent

Declaration

```
long mpiEventMgrEvent(MPIEventMgr    eventMgr,
                      MPIEventStatus *status)
```

Required Header

stdmpi.h

Description

EventMgrEvent requests that the EventMgr (***eventMgr***) call mpiNotifyEvent(notify, status) for all Notify objects on that Event Manager's list. Each Notify object represents a thread that has requested event notification. The Notify object will determine whether it accepts the event notification or not.

EventMgrEvent enables your application to use the Event Manager to distribute *user-created events* in the same way that events generated by the motion controller are distributed.

Return Values

MPIMessageOK	if *EventMgrEvent* successfully requests that the EventMgr call mpiNotify Event(notify, status) for all Notify objects on that Event Manager's list

See Also

meiEventMgrServiceConfigGet

Declaration

```
long meiEventMgrServiceConfigGet(MPIEventMgr              eventMgr,
                                 MEIEventMgrServiceConfig *config)
```

Required Header

stdmei.h

Description

EventMgrServiceConfigGet gets the configuration of an ***EventMgr*** object (eventMgr) and writes it into the structure pointed to by ***config***.

Return Values

MPIMessageOK	if *EventMgrServiceConfigGet* successfully gets the EventMgr's configuration and writes it into the structure(s).

See Also

meiEventMgrServiceConfigSet | mpiEventMgrService

meiEventMgrServiceConfigSet

Declaration

```
long meiEventMgrServiceConfigSet(MPIEventMgr              eventMgr,
                                 MEIEventMgrServiceConfig *config)
```

Required Header

stdmei.h

Description

EventMgrServiceConfigSet sets (writes) the flash configuration for an EventMgr object (***eventMgr***) using data from the structure pointed to by ***config***.

Return Values

MPIMessageOK	if *EventMgrServiceConfigSet* successfully sets (writes) the EventMgr's flash configuration using data from the structure(s)

See Also

meiEventMgrServiceConfigGet | mpiEventMgrService

mpiEventMgrFlush

Declaration `long mpiEventMgrFlush(MPIEventMgr eventMgr)`

Required Header stdmpi.h

Description **EventMgrFlush** flushes any pending events from an EventMgr (***eventMgr***).

Return Values	
MPIMessageOK	if *EventMgrFlush* successfully flushes any pending EventMgr events

See Also

mpiEventMgrService

Declaration

```
long mpiEventMgrService(MPIEventMgr eventMgr,
                        MPIControl  control)
```

Required Header

stdmpi.h

Description

EventMgrService obtains all pending asynchronous events from the motion controller (***control***).

Events generated by sources for which no prior call to ***mpiEventMgrEnable(...)*** has been made are discarded. Events generated by enabled sources are returned in First-In/First-Out (FIFO) order each time a thread calls **mpiEventMgrWait(...)**.

Typically, after a motion controller generates a hardware interrupt, the Interrupt Service Routine (ISR) is invoked, and the ISR in turn invokes **mpiEventMgrService(...)** directly or indirectly.

If "control" is	*Then*
MPIHandleVOID	events will be obtained from all motion controllers on the EventMgr's (***eventMgr***) control list
valid	***control*** must also be present on the EventMgr's (***eventMgr***) control list

Return Values

MPIMessageOK	if *EventMgrService* successfully obtains all pending asynchronous events from the motion controller

See Also

meiEventMgrServiceConfigGet | meiEventMgrServiceConfigSet

mpiEventMgrControl

Declaration

```
MPIControl mpiEventMgrControl(MPIEventMgr eventMgr,
                              long        index)
```

Required Header

stdmpi.h

Description

Returns the element at the position on the list indicated by "index".

eventMgr	a handle to the EventMgr object.
index	a position in the list.

Return Values

handle	to the ***index***th motion controller (Control) associated with an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if ***index*** is less than 0 if ***index*** is greater than or equal to **mpiEventMgrCount(eventMgr)**
MPIMessageARG_INVALID	***index*** is a negative number.
MEIListMessageELEMENT_NOT_FOUND	***index*** is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	***eventMgr*** is an invalid handle.

See Also

mpiEventMgrControlAppend

Declaration

```
long mpiEventMgrControlAppend(MPIEventMgr eventMgr,
                              MPIControl  control)
```

Required Header

stdmpi.h

Description

EventMgrControlAppend appends ***control*** to the list of motion controllers associated with an EventMgr (***eventMgr***). Add "control" to the end of the list.

eventMgr	a handle to the EventMgr object.
control	a handle to a Control object.

Return Values

MPIMessageOK	if *EventMgrControlAppend* successfully appends control to the list of motion controllers associated with an EventMgr
MPIMessageHANDLE_INVALID	Either ***eventMgr*** or ***control*** is an invalid handle.
MPIMessageOBJECT_ON_LIST	***control*** is already on the list.
MPIMessageNO_MEMORY	Not enough memory was available.

See Also

mpiEventMgrControlCount

Declaration

```
long mpiEventMgrControlCount(MPIEventMgr eventMgr)
```

Required Header

stdmpi.h

Description

Returns the number of elements on the list.

eventMgr	a handle to the EventMgr object.

Return Values

number	of motion controllers associated with an EventMgr (***eventMgr***)
-1	if ***eventMgr*** is an invalid handle
0	if ***eventMgr*** has no associated motion controllers

See Also

mpiEventMgrControlFirst

Declaration

```
MPIControl mpiEventMgrControlFirst(MPIEventMgr eventMgr)
```

Required Header

stdmpi.h

Description

EventMgrControlFirst returns the first element in the list. This function can be used in conjuntion with mpiEventMgrControlNext() in order to iterate through the list. MPIHandleVOID is returned if the list is empty.

eventMgr	a handle to the EventMgr object.

Return Values

handle	to the first motion controller (Control) associated with an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if ***eventMgr*** has no associated motion controllers
MPIMessageHANDLE_INVALID	if ***eventMgr*** is an invalid handle.

See Also

mpiEventMgrControlLast

mpiEventMgrControlIndex

Declaration

```
long mpiEventMgrControlIndex(MPIEventMgr eventMgr,
                             MPIControl  control)
```

Required Header

stdmpi.h

Description

EventMgrControlIndex returns the position of "control" on the list.

eventMgr	a handle to the EventMgr object.
control	a handle to a Control object.

Return Values

index	of ***control*** in the list of motion controllers associated with an EventMgr (***eventMgr***)
-1	if ***eventMgr*** is invalid if ***control*** was not found in the list of motion controllers

See Also

mpiEventMgrControlInsert

Declaration

```
long mpiEventMgrControlInsert(MPIEventMgr eventMgr,
                              MPIControl  control,
                              MPIControl  insert)
```

Required Header

stdmpi.h

Description

EventMgrControlInsert inserts a Control object (***insert***) after the Control object (***control***) in the list of motion controllers associated with ***eventMgr***.

Return Values

MPIMessageOK	if *EventMgrControlInsert* successfully inserts the Control object (***insert***) after another Control object (***control***) in the list of motion controllers

See Also

mpiEventMgrControlLast

Declaration

```
MPIControl mpiEventMgrControlLast(MPIEventMgr eventMgr)
```

Required Header

stdmpi.h

Description

EventMgrControlLast returns the last element in the list.
This function can be used in conjuntion with mpiEventMgrControlPrevious() in order to iterate through the list backwards.

eventMgr	a handle to the EventMgr object.

Return Values

handle	to the last motion controller (Control) associated with an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if ***eventMgr*** has no associated motion controllers
MPIMessageHANDLE_INVALID	if ***eventMgr*** is an invalid handle.

See Also

mpiEventMgrControlFirst

mpiEventMgrControlListGet

Declaration

```
long mpiEventMgrControlListGet(MPIEventMgr eventMgr,
                               long        *controlCount,
                               MPIControl  *controlList)
```

Required Header

stdmpi.h

Description

EventMgrControlListGet returns the list of Control objects associated with an EventMgr object (***eventMgr***).

EventMgrControlListGet also writes the number of Control handles (in the list) to the location pointed to by ***controlCount***, and writes an array (of ***controlCount*** Control handles) to the location pointed to by ***controlList***.

Return Values

MPIMessageOK	if *EventMgrControlListGet* successfully returns the list of Control objects associated with an EventMgr object

See Also

mpiEventMgrControlListSet

mpiEventMgrControlListSet

Declaration

```
long mpiEventMgrControlListSet(MPIEventMgr eventMgr,
                               long        controlCount,
                               MPIControl  *controlList)
```

Required Header

stdmpi.h

Description

EventMgrControlListSet creates a list of ***controlCount*** Control objects using the Control handles specified by ***controlList***. Any existing motion controller list is completely replaced.

The ***controlList*** parameter is the address of an array of ***controlCount*** Control handles, or is NULL (if ***controlCount*** is equal to zero).

A motion controller list can also be created incrementally (i.e., one Control at a time) by using the append and/or insert methods described in this section. The initial Control of a control list may be specified using the ***control*** parameter of **mpiEventMgrCreate(...)**. The list methods in this section can be used to examine and manipulate a motion controller list regardless of how the list was created.

Return Values

MPIMessageOK	if *EventMgrControlListSet* successfully creates a list of Control objects using the Control handles

See Also

mpiEventMgrControlListGet

mpiEventMgrControlNext

Declaration

```
MPIControl mpiEventMgrControlNext(MPIEventMgr  eventMgr,
                                  MPIControl   control)
```

Required Header

stdmpi.h

Description

EventMgrControlNext returns the next element following "control" on the list. MPIHandleVOID is returned if "control" is the last element on the list, or if "control" is not in the list at all. This function can be used in conjuntion with mpiEventMgrControlFirst() in order to iterate through the list.

eventMgr	a handle to the EventMgr object.
control	a handle to a Control object.

Return Values

handle	to the motion controller following ***control*** in the list of motion controllers associated with an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if ***control*** is the last motion controller
MPIMessageHANDLE_INVALID	Either ***eventMgr*** or ***control*** is an invalid handle.

See Also

mpiEventMgrControlPrevious

mpiEventMgrControlPrevious

Declaration

```
MPIControl  mpiEventMgrControlPrevious(MPIEventMgr eventMgr,
                                       MPIControl  control)
```

Required Header

stdmpi.h

Description

EventMgrControlPrevious returns the previous element prior to "control" on the list. MPIHandleVOID is returned if "control" is the first element on the list, or if "control" is not in the list at all. This function can be used in conjuntion with mpiEventMgrControlLast() in order to iterate through the list backwards.

eventMgr	a handle to the EventMgr object.
control	a handle to a Control object.

Return Values

handle	to the motion controller preceding ***control*** in the list of motion controllers associated with an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if ***control*** is the first motion controller
MPIMessageHANDLE_INVALID	Either ***eventMgr*** or ***control*** is an invalid handle.

See Also

mpiEventMgrControlNext

mpiEventMgrControlRemove

Declaration

```
long mpiEventMgrControlRemove(MPIEventMgr eventMgr,
                              MPIControl  control)
```

Required Header stdmpi.h

Description **EventMgrControlRemove** removes a Control object (***control***) from the list of Control objects associated with ***eventMgr***.

Return Values	
MPIMessageOK	if *EventMgrControlRemove* successfully removes a Control object from the list of Control objects associated with an EventMgr (***eventMgr***)

See Also mpiEventMgrControl

mpiEventMgrNotify

Declaration

```
MPINotify mpiEventMgrNotify(MPIEventMgr eventMgr,
                            long        index)
```

Required Header

stdmpi.h

Description

EventMgrNotify returns the element at the position on the list indicated by "index".

eventMgr	a handle to the EventMgr object.
index	a position in the list.

Return Values

handle	to the ***index***th Notify object associated with an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if ***index*** is less than 0 if ***index*** is greater than or equal to **mpiEventMgrCount(eventMgr)**
MPIMessageARG_INVALID	***index*** is a negative number.
MEIListMessageELEMENT_NOT_FOUND	***index*** is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	***eventMgr*** is an invalid handle.

See Also

mpiEventMgrNotifyAppend

Declaration

```
long mpiEventMgrNotifyAppend(MPIEventMgr eventMgr,
                             MPINotify   notify)
```

Required Header

stdmpi.h

Description

EventMgrNotifyAppend appends a Notify object (***notify***) to the list of Notify objects maintained by an EventMgr object (***eventMgr***).

eventMgr	a handle to the EventMgr object.
notify	a handle to a Notify object.

Return Values

MPIMessageOK	if *EventMgrNotifyAppend* successfully appends a Notify object to the list of Notify objects maintained by an EventMgr object
MPIMessageHANDLE_INVALID	Either ***eventMgr*** or ***notify*** is an invalid handle.
MPIMessageOBJECT_ON_LIST	if ***notify*** is already on the list.
MPIMessageNO_MEMORY	if not enough memory was available.

See Also

mpiEventMgrNotifyCount

Declaration

```
long mpiEventMgrNotifyCount(MPIEventMgr eventMgr)
```

Required Header

stdmpi.h

Description

EventMgrNotifyCount returns the number of elements on the list.

eventMgr	a handle to the EventMgr object.

Return Values

number	of Notify objects in the list (of Notify objects) maintained by an EventMgr (***eventMgr***)
-1	if ***eventMgr*** is invalid
0	if the list (of Notify objects) is empty

See Also

mpiEventMgrNotifyFirst

Declaration

```
MPINotify mpiEventMgrNotifyFirst(MPIEventMgr eventMgr)
```

Required Header

stdmpi.h

Description

EventMgrNotifyFirst returns the first element in the list. This function can be used in conjuntion with mpiEventMgrNotifyNext() in order to iterate through the list.

eventMgr	a handle to the EventMgr object.

Return Values

handle	to the first Notify object in the list (of Notify objects) maintained by an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if the list (of Notify objects) is empty
MPIMessageHANDLE_INVALID	if ***eventMgr*** is an invalid handle.

See Also

mpiEventMgrNotifyLast

mpiEventMgrNotifyIndex

Declaration

```
long mpiEventMgrNotifyIndex(MPIEventMgr eventMgr,
                            MPINotify   notify)
```

Required Header

stdmpi.h

Description

EventMgrNotifyIndex returns the position of "notify" on the list.

eventMgr	a handle to the EventMgr object.
notify	a handle to the EventMgr object.

Return Values

index	of ***notify*** in the list (of Notify objects) maintained by an EventMgr (***eventMgr***)
-1	if eventMgr is invalid if ***notify*** was not found in the list

See Also

mpiEventMgrNotifyInsert

Declaration

```
long mpiEventMgrNotifyInsert(MPIEventMgr eventMgr,
                             MPINotify   notify,
                             MPINotify   insert)
```

Required Header

stdmpi.h

Description

EventMgrNotifyInsert places a Notify object (***insert***) after another Notify object (***notify***) in the list (of Notify objects) maintained by an EventMgr object (***eventMgr***).

Return Values

MPIMessageOK	if *EventMgrNotifyInsert* successfully places a Notify object after another Notify object in the list of Notify objects

See Also

mpiEventMgrNotifyLast

Declaration

```
MPINotify mpiEventMgrNotifyLast(MPIEventMgr eventMgr)
```

Required Header

stdmpi.h

Description

EventMgrNotifyLast returns the last element in the list. This function can be used in conjuntion with mpiEventMgrNotifyPrevious() in order to iterate through the list backwards.

eventMgr	a handle to the EventMgr object.

Return Values

handle	to the last Notify object in the list maintained by an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if the list (of Notify objects) is empty
MPIMessageHANDLE_INVALID	if ***eventMgr*** is an invalid handle.

See Also

mpiEventMgrNotifyFirst

mpiEventMgrNotifyListGet

Declaration

```
long mpiEventMgrNotifyListGet(MPIEventMgr eventMgr,
                              long        *notifyCount,
                              MPINotify   *notifyList)
```

Required Header

stdmpi.h

Description

EventMgrNotifyListGet returns the list of Notify objects for an EventMgr object (***eventMgr***). *EventMgrNotifyListGet* also sets (writes) the number (of Notify objects in the list) to the location pointed to by ***notifyCount***, and sets (writes) an array (of ***notifyCount*** Notify handles) to the contents of the location pointed to by ***notifyList***.

Return Values

MPIMessageOK	if *EventMgrNotifyListGet* successfully returns the list of Notify objects for an EventMgr object

See Also

mpiEventMgrNotifyListSet

mpiEventMgrNotifyListSet

Declaration

```
long mpiEventMgrNotifyListSet(MPIEventMgr  eventMgr,
                              long         notifyCount,
                              MPINotify    *notifyList)
```

Required Header

stdmpi.h

Description

EventMgrNotifyListSet creates a list of Notify objects, where the number of Notify objects is specified by ***notifyCount***, using the Notify handles specified by ***notifyList***. The ***notifyList*** argument is the address of an array of ***notifyCount*** Notify handles, or is NULL if ***notifyCount*** is zero. Any existing notify object list is completely replaced.

You can also create a Notify object list incrementally (i.e., one Notify object is created at a time), by using the *EventMgrAppend* and/or *EventMgrInsert* methods. After creating a list of Notify objects, use the *EventMgrList* methods to examine and manipulate the list, regardless of how the list was created.

Return Values

MPIMessageOK	if *EventMgrNotifyListSet* successfully creates a list of Notify objects using the Notify handles specified by ***notifyList***

See Also

mpiEventMgrNotifyListGet

mpiEventMgrNotifyNext

Declaration

```
MPINotify mpiEventMgrNotifyNext(MPIEventMgr eventMgr,
                                MPINotify   notify)
```

Required Header

stdmpi.h

Description

EventMgrNotifyNext returns the next element following "notify" on the list. This function can be used in conjuntion with mpiEventMgrNotifyFirst() in order to iterate through the list.

eventMgr	a handle to the EventMgr object.
notify	a handle to a Notify object.

Return Values

handle	to the Notify object after another Notify object (***notify***) in the list (of Notify objects) maintained by an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if ***notify*** is the last notify object in the list
MPIMessageHANDLE_INVALID	Either ***eventMgr*** or ***notify*** is an invalid handle.

See Also

mpiEventMgrNotifyPrevious

mpiEventMgrNotifyPrevious

Declaration

```
MPINotify mpiEventMgrNotifyPrevious(MPIEventMgr eventMgr,
                                    MPINotify   notify)
```

Required Header

stdmpi.h

Description

EventMgrNotifyPrevious returns the previous element prior to "notify" on the list. This function can be used in conjuntion with mpiEventMgrNotifyLast() in order to iterate through the list backwards.

eventMgr	a handle to the EventMgr object.
notify	a handle to a Notify object.

Return Values

handle	to the Notify object just before another Notify object (***notify***) in the list (of Notify objects) maintained by an EventMgr (***eventMgr***)
MPIHandleVOID	if ***eventMgr*** is invalid if ***notify*** is the first Notify object in the list
MPIMessageHANDLE_INVALID	either ***eventMgr*** or ***notify*** is an invalid handle

See Also

mpiEventMgrNotifyNext

mpiEventMgrNotifyRemove

Declaration

```
long mpiEventMgrNotifyRemove(MPIEventMgr eventMgr,
                             MPINotify   notify)
```

Required Header stdmpi.h

Description **EventMgrNotifyRemove** removes a Notify object (***notify***) from the list of Notify objects maintained by an EventMgr object (***eventMgr***).

Return Values

MPIMessageOK	if *EventMgrNotifyRemove* successfully removes a Notify object from the list of Notify objects maintained by an EventMgr object

See Also

MPIEventMgrMessage

MPIEventMgrMessage

```
typedef enum {

    MPIEventMgrMessageEVENTMGR_INVALID,
} MPIEventMgrMessage;
```

Description

MPIEventMgrMessageEVENTMGR_INVALID	
Meaning	The MPIEventMgr handle passed to an MPIEventMgr method is invalid.
Possible Causes	Either the handle was never initialized or the mpiEventMgrCreate method failed.
Recommendations	Use mpiEventMgrValidate after mpiEventMgrCreate to see if the returned handle is valid.

Sample Code

```
MPIControl      control;
MPIEventMgr     eventMgr;
long            returnValue;
…

eventMgr =
    mpiEventMgrCreate(control);
returnValue =
    mpiEventMgrValidate(eventMgr);
```

See Also

MPIEventMgr | mpiEventMgrCreate | mpiEventMgrValidate

MEIEventMgrServiceConfig

MEIEventMgrServiceConfig

```
typedef struct MEIEventMgrServiceConfig
            { long allProcesses; /* TRUE => collect events from all processes,
            else EventMgr process only */
} MEIEventMgrServiceConfig;
```

Description

allProcesses	is a boolean value. If allProcesses=TRUE, then the event manager will handle events originating from all processes. If allProcesses=FALSE, then the event manager will only handle events originating from the same process in which the event manager was created.

See Also

Filter Objects

Introduction

A **Filter** object manages a single filter on a controller. It represents the control algorithm used to control a motor in a closed-loop system. The Filter contains an algorithm, a set of coefficients, inputs, and an output. Its primary responsibility is to take the difference between the command and actual positions and then calculate the output based on the control algorithm and coefficients.

For simple systems, there is a one-to-one relationship between the Axis, Filter, and Motor objects.

Methods

Create, Delete, Validate Methods

mpiFilter**Create**	Create Filter object
mpiFilter**Delete**	Delete Filter object
mpiFilter**Validate**	Validate Filter object

Configuration and Information Methods

mpiFilter**ConfigGet**	Get Filter configuration
mpiFilter**ConfigSet**	Set Filter configuration
mpiFilter**FlashConfigGet**	Get flash configuration for Filter
mpiFilter**FlashConfigSet**	Set flash configuration for Filter
mpiFilter**GainGet**	Get gain coefficients
mpiFilter**GainSet**	Set current gain index
mpiFilter**GainIndexGet**	Get current gain index
mpiFilter**GainIndexSet**	Set current gain index
mpiFilter**Status**	Get Filter's status

Memory Methods

mpiFilter**Memory**	Get address to Filter memory
mpiFilter**MemoryGet**	Copy data from Filter memory to application memory
mpiFilter**MemorySet**	Copy data from application memory to Filter memory

Relational Methods

mpiFilter**AxisMapGet**	Get object map of axes associated with Filter
mpiFilter**AxisMapSet**	Set axes associated with Filter
mpiFilter**Control**	Return handle of Control that is assoiciated with Filter
mpiFilter**MotorMapGet**	Get object map of Motors associated with Filter
mpiFilter**MotorMapSet**	Set Motors to be associated with Filter
mpiFilter**Number**	Get index of Filter (for Control list)

Action Methods

mpiFilter**IntergratorReset**	Reset the integrators of filter.

Data Types

MPIFilter**Coeff**
MPIFilter**Config** / MEIFilter**Config**
MPIFilter**Gain**
MEIFilter**GainIndex**
MEIFilter**GainPID**
MEIFilter**GainPIDCoeff**
MEIFilter**GainPIV**
MEIFilter**GainPIVCoeff**
MEIFilter**GainSercos**
MEIFilter**GainSercosCoeff**
MPIFilter**Message**

Constants

MPIFilter**CoeffCOUNT_MAX**
MPIFilter**GainCOUNT_MAX**

Copyright © 2002
Motion Engineering

mpiFilterCreate

Declaration

```
const MPIFilter mpiFilterCreate(MPIControl control,
                                long       number)
```

Required Header stdmpi.h

Description

FilterCreate creates a Filter object associated with a filter (***number***), that is located on a motion controller (***control***).
FilterCreate is the equivalent of a C++ constructor.

Return Values

handle	to an Filter object
MPIHandleVOID	if the Filter object could not be created

See Also mpiFilterDelete | mpiFilterValidate

mpiFilterDelete

Declaration `long mpiFilterDelete(MPIFilter filter)`

Required Header stdmpi.h

Description **FilterDelete** deletes a Filter object and invalidates its handle (***filter***). FilterDelete is the equivalent of a C++ destructor.

Return Values	
MPIMessageOK	if *FilterDelete* successfully deletes a Filter object and invalidates its handle

See Also mpiFilterCreate | mpiFilterValidate

mpiFilterValidate

Declaration `long mpiFilterValidate(MPIFilter filter)`

Required Header stdmpi.h

Description **FilterValidate** validates the Filter object and its handle (***filter***).

Return Values	
MPIMessageOK	if Filter is a handle to a valid object.

See Also mpiFilterCreate | mpiFilterDelete

mpiFilterConfigGet

Declaration

```
long mpiFilterConfigGet(MPIFilter       filter,
                        MPIFilterConfig *config,
                        void            *external)
```

Required Header

stdmpi.h

Description

FilterConfigGet gets a Filter's (***filter***) configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Filter's configuration information in ***external*** is in addition to the Filter's configuration information in ***config***, i.e, the Filter's configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEIFilterConfig{}** or is NULL.

Return Values

MPIMessageOK	if *FilterConfigGet* successfully writes the Filter's configuration to the structure(s)

See Also

mpiFilterConfigSet | MEIFilterConfig

mpiFilterConfigSet

Declaration

```
long mpiFilterConfigSet(MPIFilter       filter,
                        MPIFilterConfig *config,
                        void            *external)
```

Required Header

stdmpi.h

Description

FilterConfigSet sets a Filter's (***filter***) configuration using data from the structure pointed to by ***config***, and from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Filter's configuration information in ***external*** is in addition to the Filter's configuration information in ***config***, i.e, the Filter's configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	external either points to a structure of type MEIFilterConfig{} or is NULL.

Return Values

MPIMessageOK	if *FilterConfigSet* successfully sets the Filter's configuration using data from the structure(s)

See Also

mpiFilterConfigGet | MEIFilterConfig

mpiFilterFlashConfigGet

Declaration

```
long mpiFilterFlashConfigGet(MPIFilter       filter,
                             void            *flash,
                             MPIFilterConfig *config,
                             void            *external)
```

Required Header

stdmpi.h

Description

FilterFlashConfigGet gets a Filter's (***filter***) flash configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Filter's flash configuration information in ***external*** is in addition to the Filter's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEIFilterConfig{}** or is NULL.

Return Values

MPIMessageOK	if *FilterFlashConfigGet* successfully writes the Filter's flash configuration to the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiFilterFlashConfigSet |MEIFilterConfig

mpiFilterFlashConfigSet

Declaration

```
long mpiFilterFlashConfigSet(MPIFilter       filter,
                             void            *flash,
                             MPIFilterConfig *config,
                             void            *external)
```

Required Header

stdmpi.h

Description

FilterFlashConfigSet sets a Filter's (***filter***) flash configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Filter's flash configuration information in ***external*** is in addition to the Filter's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only — ***external*** either points to a structure of type **MEIFilterConfig{}** or is NULL.

Return Values

MPIMessageOK	if *FilterFlashConfigSet* successfully sets the Filter's flash configuration using data from the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiFilterFlashConfigGet | MEIFilterConfig

mpiFilterGainGet

Declaration

```
long mpiFilterGainGet(MPIFilter     filter,
                      long          gainIndex,
                      MPIFilterGain *gain)
```

Required Header stdmpi.h

Description **FilterGainGet** gets the gain coefficients of a Filter (***filter***, for the gain index specified by ***gainIndex***) and writes them into the structure pointed to by ***gain***.

Return Values	
MPIMessageOK	if *FilterGainGet* successfully writes the gain coefficients to the structure

See Also mpiFilterGainSet

mpiFilterGainSet

Declaration

```
long mpiFilterGainSet(MPIFilter      filter,
                      long           gainIndex,
                      MPIFilterGain  *gain)
```

Required Header

stdmpi.h

Description

FilterGainSet sets the gain coefficients of a Filter (***filter***, for the gain index specified by ***gainIndex***) using data from the structure pointed to by ***gain***.

Return Values

MPIMessageOK	if *FilterGainSet* successfully sets the gain coefficients of a Filter using data from the structure

See Also

mpiFilterGainGet

mpiFilterGainIndexGet

Declaration

```
long mpiFilterGainIndexGet(MPIFilter filter,
                           long      *gainIndex)
```

Required Header

stdmpi.h

Description

FilterGainIndexGet gets the current gain index of a Filter (***filter***) and writes it to the location pointed to by ***gainIndex***.

Return Values

MPIMessageOK	if *FilterGainIndexGet* successfully writes the gain index to the location

See Also

mpiFilterGainIndexSet

mpiFilterGainIndexSet

Declaration

```
long mpiFilterGainIndexSet(MPIFilter filter,
                           long      gainIndex)
```

Required Header stdmpi.h

Description **FilterGainIndexSet** sets the current gain index of a Filter (***filter***) to ***gainIndex***.

Return Values	
MPIMessageOK	if *FilterGainIndexSet* successfully sets the current gain index to ***gainIndex***

See Also mpiFilterGainIndexGet

mpiFilterStatus

Declaration

```
long mpiFilterStatus(MPIFilter  filter,
                     MPIStatus  *status,
                     void       *external)
```

Required Header

stdmpi.h

Description

FilterStatus writes a Filter's (***filter***) status into the structure pointed to by ***status***, and also into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Filter's status information in ***external*** is in addition to the Filter's status information in ***status***, i.e, the status configuration information in ***status*** and in ***external*** is not the same information. Note that ***status*** or ***external*** can be NULL (but not both NULL).

XMP Only

external either points to a structure of type **MEIFilterStatus{}** or is NULL.

Return Values

MPIMessageOK	if *FilterStatus* successfully writes the status of a Filter to the structure(s)

See Also

mpiFilterMemory

Declaration

```
long mpiFilterMemory(MPIFilter  filter,
                     void       **memory)
```

Required Header

stdmpi.h

Description

FilterMemory writes an address, which is used to access a Filter's (***filter***) memory to the contents of ***memory***. This address, or an address calculated from it, can be passed as the src parameter to **MPIFilterMemoryGet(...)** and as the ***dst*** parameter to **MPIFilterMemorySet(...)**.

Return Values

MPIMessageOK	if *FilterMemory* successfully writes the Filter's memory address to the contents of ***memory***

See Also

mpiFilterMemoryGet | mpiFilterMemorySet

mpiFilterMemoryGet

Declaration

```
long mpiFilterMemoryGet(MPIFilter filter,
                        void      *dst,
                        void      *src,
                        long      count)
```

Required Header

stdmpi.h

Description

FilterMemoryGet copies ***count*** bytes of a Filter's (***filter***) memory (starting at address src) and writes them into application memory (starting at address ***dst***).

Return Values	
MPIMessageOK	if *FilterMemoryGet* successfully copies data from Filter memory to application memory

See Also

mpiFilterMemorySet | mpiFilterMemory

mpiFilterMemorySet

Declaration

```
long mpiFilterMemorySet(MPIFilter filter,
                        void      *dst,
                        void      *src,
                        long      count)
```

Required Header

stdmpi.h

Description

FilterMemorySet copies ***count*** bytes of application memory (starting at address ***src***) and writes them into a Filter's (***filter***) memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *FilterMemoryGet* successfully copies data from application memory to Filter memory

See Also

mpiFilterMemoryGet | mpiFilterMemory

mpiFilterAxisMapGet

Declaration

```
long mpiFilterAxisMapGet(MPIFilter    filter,
                         MPIObjectMap *map)
```

Required Header

stdmpi.h

Description

FilterAxisMapGet gets the object map of the Axes that are associated with a Filter (***filter***), and writes it into the structure pointed to by ***map***.

Return Values

MPIMessageOK	if *FilterAxisMapGet* successfully writes the object map of Axes to the structure

See Also

mpiFilterAxisMapSet

mpiFilterAxisMapSet

Declaration

```
long mpiFilterAxisMapSet(MPIFilter    filter,
                         MPIObjectMap map)
```

Required Header stdmpi.h

Description **FilterAxisMapSet** sets the Axes associated with a Filter (***filter***), using data from the object map specified by ***map***.

Return Values

MPIMessageOK	if *FilterAxisMapSet* successfully sets the Axes using the object map

See Also mpiFilterAxisMapGet

mpiFilterControl

Declaration `const MPIControl mpiFilterControl(MPIFilter filter)`

Required Header stdmpi.h

Description **FilterControl** returns a handle to the motion controller (Control object) associated with the specified Filter object (***filter***).

Return Values	
handle	to a Control object that a Filter object is associated with
MPIHandleVOID	if the Filter object is invalid

See Also

mpiFilterMotorMapGet

Declaration

```
long mpiFilterMotorMapGet(MPIFilter    filter,
                          MPIObjectMap *map)
```

Required Header stdmpi.h

Description **FilterMotorMapGet** gets the object map of the Motors associated with the Filter (***filter***), and writes it into the structure pointed to by ***map***.

Return Values	
MPIMessageOK	if *FilterMotorMapGet* successfully writes the object map of the Motors to the structure

See Also mpiFilterMotorMapSet

mpiFilterMotorMapSet

Declaration

```
long mpiFilterMotorMapSet(MPIFilter    filter,
                          MPIObjectMap map)
```

Required Header

stdmpi.h

Description

FilterMotorMapSet sets the Motors associated with the Filter (***filter***) using data from the object map specified by ***map***.

Return Values

MPIMessageOK	if *FilterMotorMapGet* successfully sets the Motors using data from the object map

See Also

mpiFilterMotorMapGet

mpiFilterNumber

Declaration

```
long mpiFilterNumber(MPIFilter  filter,
                     long       *number)
```

Required Header

stdmpi.h

Description

For a motion controller that ***filter*** is associated with, **FilterNumber** writes the index of ***filter*** to the contents of ***number***.

Return Values	
MPIMessageOK	if *FilterNumber* successfully writes the index of a Filter to the contents of **number**

See Also

mpiFilterIntegratorReset

Declaration `long mpiFilterIntegratorReset(MPIFilter filter)`

Required Header stdmpi.h

Description **FilterIntegratorReset** resets the integrators of filter.

Return Values	
MPIMessageOK	if *mpiFilterIntegratorReset* successfully clears the integrators of ***filter***.
MPIFilterMessageINVALID_ALGORITHM	if the ***filter's*** current algorithm does not use integrators.

Sample Code

```
/* Enable the amplifier for every motor attached to a motion supervisor */
void motionAmpEnable(MPIMotion  motion)
{
        MPIControl          control;
        MPIAxis             axis;
        MPIMotor            motor;
        MPIFilter           filter;
        MPIObjectMap    map;
        MPIObjectMap    motionMotorMap;
        long                motorIndex;
        long                filterIndex;
        long                returnValue;
        double              position;
        long                enableState;


        /* Get the controller handle */
        control = mpiMotionControl(motion);

        for (axis =  mpiMotionAxisFirst(motion);
                axis != MPIHandleVOID;
                axis =  mpiMotionAxisNext(motion, axis)) {

                /* Get the object map for the motors */
                returnValue = mpiAxisMotorMapGet(axis, &map);
                msgCHECK(returnValue);

                /* Add map to motionMotorMap */
                motionMotorMap |= map;
        }

        /* For every motor ... */
        for (motorIndex = 0; motorIndex < MEIXmpMAX_Motors; motorIndex++) {

                if (mpiObjectMapBitGET(motionMotorMap, motorIndex)) {

                        /* Create motor handle */
                        motor = mpiMotorCreate(control, motorIndex);
                        msgCHECK(mpiMotorValidate(motor));

                        /* Get the state of the amplifier */
                        returnValue = mpiMotorAmpEnableGet(motor, &enableState);
                        msgCHECK(returnValue);
```

```
                        /* If the amplifier is disabled ... */
                        if (enableState == FALSE) {

                            /* For every axis */
                            for (axis =  mpiMotionAxisFirst(motion);
                                 axis != MPIHandleVOID;
                                 axis =  mpiMotionAxisNext(motion, axis)) {

                                /* Get the object map for the motors */
                                returnValue = mpiAxisMotorMapGet(axis, &map);
                                msgCHECK(returnValue);

                                /* If axis is attached to motor ... */
                                if (mpiObjectMapBitGET(map, motorIndex)) {

                                    /* Get the actual position of the axis */
                                    returnValue = mpiAxisActualPositionGet(axis, &position);
                                    msgCHECK(returnValue);

                                    /* Set command position equal to actual position */
                                    returnValue = mpiAxisCommandPositionSet(axis, position);
                                    msgCHECK(returnValue);
                                }
                            }

                            /* Get the object map for the filters */
                            returnValue = mpiMotorFilterMapGet(motor, &map);
                            msgCHECK(returnValue);

                            /* For every filter ... */
                            for (filterIndex = 0;
                                 filterIndex < MEIXmpMAX_Filters;
                                 filterIndex++) {

                                if (mpiObjectMapBitGET(map, filterIndex)) {

                                    /* Create filter handle */
                                    filter = mpiFilterCreate(control, filterIndex);
                                    msgCHECK(mpiFilterValidate(filter));

                                    /* Reset integrator */
                                    returnValue = mpiFilterIntegratorReset(filter);
                                    msgCHECK(returnValue);

                                    /* Delete filter handle */
                                    returnValue = mpiFilterDelete(filter);
                                    msgCHECK(returnValue);
                                }
                            }

                            /* Enable the amplifier */
                            returnValue = mpiMotorAmpEnableSet(motor, TRUE);
                            msgCHECK(returnValue);
                        }

                        /* Delete motor handle */
                        returnValue = mpiMotorDelete(motor);
                        msgCHECK(returnValue);
                    }
                }
            }
```

See Also MPIFilter | MEIFilterConfig | MEIFilterGainPID | MEIFilterGainPIV
mpiAxisActualPositionGet | mpiAxisCommandPositionSet

Troubleshooting / Helpful Hints

If an axis is not in an error state and the filter associated with that axis' motor has a non-zero integration term, then it is very likely that the integrator has built up a substantial integral term. Enabling the motor's amplifier when this has happened could cause the motor to jump with enormous force. Use **mpiFilterIntegratorReset** to reset the integrator before enabling the motor's amplifier to prevent this kind of jump.

Another condition that can cause the motor to jump upon enabling its amplifier is that the command position of the axis is not equal to the actual position of the axis. To prevent this situation, one should use **mpiAxisActualPositionGet** and **mpiAxisCommandPositionSet**. Please refer to this functions for a more in depth discussion.

MPIFilterCoeff

MPIFilterCoeff

```
typedef union {
    float       f;
    long        l;
} MPIFilterCoeff;
```

Description

f	float coefficient
l	long coefficient

See Also

MPIFilterCoeffCOUNT_MAX

MPIFilterConfig / MEIFilterConfig

MPIFilterConfig

```
typedef struct MPIFilterConfig {
    long                gainIndex;
    MPIFilterGain       gain[MPIFilterGainCOUNT_MAX];
    long                sercosGainIdnMap[MPIFilterCoeffCOUNT_MAX];

    MPIObjectMap        axisMap;
    MPIObjectMap        motorMap;
} MPIFilterConfig;
```

Description

gainIndex	gain table index. Gain tables number 0 to MPIFilterGainCOUNT_MAX -1 (MPIFilterGainCOUNT_MAX = 4).
gain	see MPIFilterGain
sercosGainIdnMap	Map of Sercos servo loop gains. MPIFilterCoeffCOUNT_MAX coefficients in the map (MPIFilterCoeffCOUNT_MAX = 20).
axisMap	see MPIObjectMap
motorMap	see MPIObjectMap

MEIFilterConfig

```
typedef struct MEIFilterConfig {
    MEIXmpAlgorithm     Algorithm;

    MEIXmpAxisInput     Axis[MEIXmpFilterAxisInputs];

    long                *VelPositionPtr;
    long                *AuxInput[MEIXmpFilterAuxInputs];

    MEIXmpSwitchType     GainSwitchType;
    float                GainDelay;
    long                 GainWindow;
    MEIXmpSwitchType     PPISwitchType;
    MEIXmpPPIMode        PPIMode;
    float                PPIDelay;
    long                 PPIWindow;
    MEIXmpIntResetConfig ResetIntegratorConfig;
    float                ResetIntegratorDelay;

    MEIXmpPostFilter     PostFilter;
} MEIFilterConfig;
```

Description

Algorithm	This value defines the algorithm that the filter is executing every servo cycle.
Axis[MEIXmpFilterAxisInputs]	This array defines the axis (pointer to the axis) and coefficient for the position input into the filter. The input to the filter is the position error of the axis, which is multiplied by the coefficient defined by the Axis array.
VelPositionPtr	Velocity position pointer to an encoder input for algorithms that require a velocity encoder position input (such as the PIV algorithm).
AuxInput[MEIXmpFilterAuxInputs]	This array is a place holder for additional filter inputs from analog sources. This is currently not supported and is reserved for future use.
GainSwitchType	Value to define the gain table switch type. Not implemented in standard firmware.
GainDelay	Custom Delay not implemented in standard firmware.
GainWindow	Custom Delay not implemented in standard firmware.
PPISwitchType	Value to define the gain switch type for PPI mode. Not implemented in standard firmware.
PPIMode	Value to define the PPI switch mode. Not implemented in standard firmware.
PPIDelay	Custom Delay not implemented in standard firmware.
PPIWindow	Custom Window not implemented in standard firmware.
ResetIntegratorConfig	Value to define the integrator's reset configuration. Not supported in standard firmware.
ResetIntegratorDelay	Value to define the integrator's reset delay. Not supported in standard firmware.
PostFilter	This array defines the configuration for the filter's post filter (the type, the length and values for the post filter coefficients).

See Also

MPIFilterGain

MPIFilterGain

```
typedef struct MPIFilterGain {
    MPIFilterCoeff      coeff[MPIFilterCoeffCOUNT_MAX];
} MPIFilterGain;
```

Description

coeff	see MPIFilterCoeff

See Also

MPIFilterGainCOUNT_MAX

MEIFilterGainIndex

MEIFilterGainIndex

```
typedef enum {  /* NOTE: These *must* match MEIXmpGain{}!!! */
    MEIFilterGainIndexINVALID,

    MEIFilterGainIndexSTOPPING2,  /* MEIXmpGainSTOPPED2 */
    MEIFilterGainIndexSTOPPING1,  /* MEIXmpGainSTOPPED1 */
    MEIFilterGainIndexSETTLING,   /* MEIXmpGainSETTLING */
    MEIFilterGainIndexMOVING,     /* MEIXmpGainMOVING */
    MEIFilterGainIndexLAST,       /* MEIXmpGainLAST */
    MEIFilterGainIndexALL,
    MEIFilterGainIndexDEFAULT   = MEIFilterGainIndexSTOPPING2,
} MEIFilterGainIndex;
```

Description

FilterGainIndex is a constant enum structure defining gain indexes used to select one of the four filter gain tables.

See Also

MEIFilterGainPID

MEIFilterGainPID

```
typedef struct MEIFilterGainPID {
    struct {
    float       proportional;   /* Kp */
    float       integral;       /* Ki */
    float       derivative;     /* Kd */
    } gain;
    struct {
        float   position;       /* Kpff */
        float   velocity;       /* Kvff */
        float   acceleration;   /* Kaff */
        float   friction;       /* Kfff */
    } feedForward;
    struct {
        float   moving;         /* MovingIMax */
        float   rest;           /* RestIMax */
    } integrationMax;
    long        dRate;          /* DRate */
    struct {
        float   limit;          /* OutputLimit */
        float   limitHigh;      /* OutputLimitHigh */
        float   limitLow;       /* OutputLimitLow */
        float   offset;         /* OutputOffset */
    } output;
    struct {
        float   positionFFT;    /* Ka0 */
        float   filterFFT;      /* Ka1 */
        float   velocityFFT;    /* Ka2 */
    } noise;
} MEIFilterGainPID;
```

Description

FilterGainPID is a structure that defines the filter coefficients for the PID filter algorithm.

See Also

High/Low Output Limits section for special instructions regarding MEIFilterGainPID.
MEIFilterGainPIDCoeff

MEIFilterGainPIDCoeff

MEIFilterGainPIDCoeff

```
typedef         enum {
    MEIFilterGainPIDCoeffINVALID = -1,

    MEIFilterGainPIDCoeffGAIN_PROPORTIONAL,     /* Kp */
    MEIFilterGainPIDCoeffGAIN_INTEGRAL,         /* Ki */
    MEIFilterGainPIDCoeffGAIN_DERIVATIVE,       /* Kd */
    MEIFilterGainPIDCoeffFEEDFORWARD_POSITION,  /* Kpff */
    MEIFilterGainPIDCoeffFEEDFORWARD_VELOCITY,  /* Kvff */
    MEIFilterGainPIDCoeffFEEDFORWARD_ACCELERATION,/* Kaff */
    MEIFilterGainPIDCoeffFEEDFORWARD_FRICTION,  /* Kfff */
    MEIFilterGainPIDCoeffINTEGRATIONMAX_MOVING, /* MovingIMax */
    MEIFilterGainPIDCoeffINTEGRATIONMAX_REST,   /* RestIMax */
    MEIFilterGainPIDCoeffDRATE,                 /* DRate */
    MEIFilterGainPIDCoeffOUTPUT_LIMIT,          /* OutputLimit */
    MEIFilterGainPIDCoeffOUTPUT_LIMITHIGH,      /* OutputLimitHigh */
    MEIFilterGainPIDCoeffOUTPUT_LIMITLOW,       /* OutputLimitLow */
    MEIFilterGainPIDCoeffOUTPUT_OFFSET,         /* OutputOffset */
    MEIFilterGainPIDCoeffNOISE_POSITIONFFT,     /* Ka0 */
    MEIFilterGainPIDCoeffNOISE_FILTERFFT,       /* Ka1 */
    MEIFilterGainPIDCoeffNOISE_VELOCITYFFT,     /* Ka2 */

} MEIFilterGainPIDCoeff;
```

Description

FilterGainPIDCoeff is a structure of enums that defines the filter coefficients for the PID filter algorithm.

See Also

MEIFilterGainPID

MEIFilterGainPIV

MEIFilterGainPIV

```
typedef         struct MEIFilterGainPIV {
    struct {
        float   proportional;   /* Kpp */
        float   integral;       /* Kip */
    } gainPosition;
    struct {
        float   proportional;   /* Kpv */
    } gainVelocity1;
    struct {
        float   position;       /* Kpff */
        float   velocity;       /* Kvff */
        float   acceleration;   /* Kaff */
        float   friction;       /* Kfff */
    } feedForward;
    struct {
        float   moving;         /* MovingIMax */
        float   rest;           /* RestIMax */
    } integrationMax;
    struct {
        float   feedback;       /* Kdv */
    } gainVelocity2;
    struct {
        float   limit;          /* OutputLimit */
        float   limitHigh;      /* OutputLimitHigh */
        float   limitLow;       /* OutputLimitLow */
        float   offset;         /* OutputOffset */
    } output;
    struct {
        float   integral;       /* Kiv */
        float   integrationMax; /* VintMax */
    } gainVelocity3;
    struct {
        float   positionFFT;    /* Ka0 */
        float   filterFFT;      /* Ka1 */
    } noise;
} MEIFilterGainPIV;
```

Description

FilterGainPIV is a structure that defines the filter coefficients for the PIV filter algorithm.

See Also

High/Low Output Limits section for special instructions regarding MEIFilterGainPIV.
MEIFilterGainPIVCoeff

MEIFilterGainPIVCoeff

MEIFilterGainPIVCoeff

```
typedef         enum {
    MEIFilterGainPIVCoeffINVALID = -1,

    MEIFilterGainPIVCoeffGAINPOSITION_PROPORTIONAL,     /* Kpp */
    MEIFilterGainPIVCoeffGAINPOSITION_INTEGRAL,         /* Kip */

    MEIFilterGainPIVCoeffGAINVELOCITY_PROPORTIONAL,     /* Kpv */

    MEIFilterGainPIVCoeffFEEDFORWARD_POSITION,          /* Kpff */
    MEIFilterGainPIVCoeffFEEDFORWARD_VELOCITY,          /* Kvff */
    MEIFilterGainPIVCoeffFEEDFORWARD_ACCELERATION,      /* Kaff */
    MEIFilterGainPIVCoeffFEEDFORWARD_FRICTION,          /* Kfff */

    MEIFilterGainPIVCoeffINTEGRATIONMAX_MOVING,         /* MovingIMax */
    MEIFilterGainPIVCoeffINTEGRATIONMAX_REST,           /* RestIMax */

    MEIFilterGainPIVCoeffGAINVELOCITY_FEEDBACK,         /* Kdv */

    MEIFilterGainPIVCoeffOUTPUT_LIMIT,          /* OutputLimit */
    MEIFilterGainPIVCoeffOUTPUT_LIMITHIGH,      /* OutputLimitHigh */
    MEIFilterGainPIVCoeffOUTPUT_LIMITLOW,       /* OutputLimitLow */
    MEIFilterGainPIVCoeffOUTPUT_OFFSET,         /* OutputOffset */

    MEIFilterGainPIVCoeffGAINVELOCITY_INTEGRAL,         /* Kiv */
    MEIFilterGainPIVCoeffGAINVELOCITY_INTEGRATIONMAX,   /* Vintmax */

    MEIFilterGainPIVCoeffNOISE_POSITIONFFT,             /* Ka0 */
    MEIFilterGainPIVCoeffNOISE_FILTERFFT,               /* Ka1 */

} MEIFilterGainPIVCoeff;
```

Description

FilterGainPIVCoeff is a structure of enums that defines the filter coefficients for the PIV filter algorithm.

See Also

High/Low Output Limits section for special instructions regarding MEIFilterGainPIV.
MEIFilterGainPIV

MEIFilterGainSercos

MEIFilterGainSercos

```
typedef struct MEIFilterGainSercos {
    long        Coeff[MEIXmpSercosCoeffMAX];
    struct {
        long    velocity;
        long    acceleration;
        long    friction;
        float   scaler;
    } feedForward;
} MEIFilterGainSercos;
```

Description

FilterGainSercos is a structure that defines the filter coefficients for the SERCOS filter algorithm.

See Also

MEIFilterGainSercosCoeff

MEIFilterGainSercosCoeff

MEIFilterGainSercosCoeff

```
typedef enum {
    MEIFilterGainSercosCoeffINVALID = -1,
    MEIFilterGainSercosCoeff0,
    MEIFilterGainSercosCoeff1,
    MEIFilterGainSercosCoeff2,
    MEIFilterGainSercosCoeff3,
    MEIFilterGainSercosCoeff4,
    MEIFilterGainSercosCoeff5,
    MEIFilterGainSercosCoeff6,
    MEIFilterGainSercosCoeff7,
    MEIFilterGainSercosCoeff8,
    MEIFilterGainSercosCoeff9,
    MEIFilterGainSercosCoeff10,
    MEIFilterGainSercosCoeff11,
    MEIFilterGainSercosCoeff12,
    MEIFilterGainSercosCoeff13,
    MEIFilterGainSercosCoeff14,
    MEIFilterGainSercosCoeff15,
    MEIFilterGainSercosCoeffFEED_FORWARD_VELOCITY,
    MEIFilterGainSercosCoeffFEED_FORWARD_ACCELERATION,
    MEIFilterGainSercosCoeffFEED_FORWARD_FRICTION,
    MEIFilterGainSercosCoeffFEED_FORWARD_SCALER,
    MEIFilterGainSercosCoeffLAST,
    MEIFilterGainSercosCoeffFIRST = MEIFilterGainSercosCoeffINVALID + 1
} MEIFilterGainSercosCoeff;
```

Description

FilterGainSercosCoeff is a structure of enums that defines the filter coefficients for the PID filter algorithm.

See Also

MEIFilterGainSercos

MPIFilterMessage

MPIFilterMessage

```
typedef enum {

    MPIFilterMessageFILTER_INVALID,
    MPIFilterMessageINVALID_ALGORITHM,

} MPIFilterMessage;
```

Description

MPIFilterMessageFILTER_INVALID	This message indicates that an invalid filter number has been specified. This message is returned by the mpiFilterCreate method when the filter number is < 0 or > MEIXmpMAX_Filters
MPIFilterMessageINVALID_ALGORITHM	This message indicates that an invalid filter algorithm has been specified.

See Also

mpiFilterCreate

MPIFilterCoeffCOUNT_MAX

MPIFilterCoeffCOUNT_MAX

```
#define MPIFilterCoeffCOUNT_MAX (20)
```

Description

FilterCoeffCOUNT_MAX is a constant that defines the maximum number of filter coefficients contained in a gain table.

See Also

MPIFilterCoeff

MPIFilterGainCOUNT_MAX

MPIFilterGainCOUNT_MAX

```
#define MPIFilterGainCOUNT_MAX (20)
```

Description

FilterGainCOUNT_MAX is a constant that defines the maximum number of filter gain tables. The first gain table is used by the standard filter types (all filter types except for the user filter type as defined by the structure MEIXmpAlgorithm). Additional gain tables can be used for manual or automatic gain switching. For firmware that implements automatic gain switching, please contact Motion Engineering. Manual gain switching can be accomplished by specifying the gainIndex of the mpiFilterConfig structure using the mpiFilterConfigSet method. Valid gainIndex values range from 0 to MPIFilterGainCOUNT_MAX.

See Also

MPIFilterGain

Special Note: *High / Low Output Limits (MEIFilterGainPID and PIV)*

In the 19990820 release, the MEIFilterGainPID and MEIFilterGainPIV structures were expanded to support High and Low output limits for PID and PIV algorithms. The "High" output limit prevents the filter output from exceeding the "High" value. The "Low" output limit prevents the filter output from falling below the "Low" value. This feature will allow an application to have upper and lower limits which are not centered on zero volts. If the "High" and "Low" values have the same sign, then the output will be limited to either the positive or negative range bounded by "High" and "Low".

The standard Output Limit is still valid. The controller will simultaneously use the standard Output Limit and the High / Low Output Limits to bound the output. The limits, (standard or high or low) that are closest to zero will be used as the boundary for the output.

Return to MEIFilterGainPID or MEIFilterGainPIV

Copyright © 2002
Motion Engineering

Flash Objects

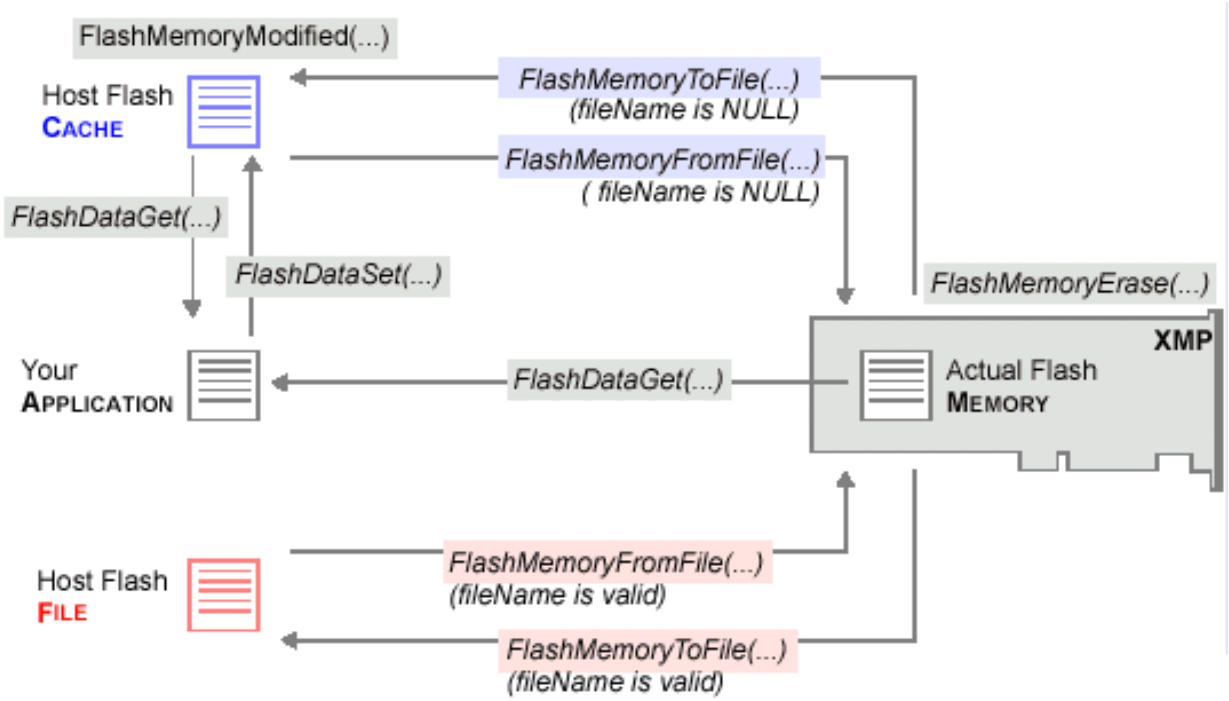
Introduction

A **Flash** object manages nonvolatile flash memory on the XMP motion controller. Because all of flash memory must be written as one contiguous piece (which takes a lot of time), and a host-resident *flash memory cache* is used to provide faster writing performance, and to also reduce the number of write cycles performed.

After your application creates a Flash object (**using meiFlashCreate(...)**), the Flash object then creates a host-resident flash memory cache, initializing it using a copy of actual flash memory [by calling **meiFlashMemoryToFile(fileName=Null)**]. Using the data structures defined in xmp.h and pointers to those structures (returned by the **mpiControlMemory(...)** method), your application can then use the **meiFlashData[Get/Set](...)** methods to access this memory cache. Note that the **meiFlashDataSet(...)** method updates the flash cache, and doesn't update the actual flash memory. To write the flash memory cache to actual flash memory, your application must call **meiFlashMemoryFromFile(fileName=Null)**.

Use the **FlashDataGet/Set(...)** methods to move data between your application and the flash cache. Use the **FlashMemoryTo/FromFile(...)** methods to move data between the flash cache (or file) and the actual flash memory. Typically, your application would:

1. Create a Flash object [using **meiFlashCreate(...)**].
2. Pass the **MEIFlash** handle to the **FlashConfig[Get/Set](...)** methods of the objects to be configured (which in turn call the **meiFlashData[Get/Set](...)** methods).
3. Write the flash cache to actual flash memory [using **meiFlashMemoryFromFile(...)**].
4. Delete the Flash object.



Methods

Create, Delete, Validate Methods

meiFlash**Create**	Create Flash object
meiFlash**Delete**	Delete Flash object
meiFlash**Validate**	Validate Flash object

Configuration and Information Methods

meiFlash**ConfigGet**	Copy flash config from cache to application memory
meiFlash**ConfigSet**	Copy flash config from application memory to cache
meiFlash**DataGet**	Get count bytes of flash data memory and write them in application memory
meiFlash**DataSet**	Set count bytes of flash data memory using application memory

Memory Methods

meiFlash**MemoryErase**	Erase actual flash memory on XMP
meiFlash**MemoryFromFile**	Write actual flash memory using cache or file
meiFlash**MemoryFromFileType**	Write actual flash memory using the binary image contained in filename
meiFlash**MemoryGet**	Copy count bytes of flash memory to application memory
meiFlash**MemoryModified**	Determine if flash cache has been modified
meiFlash**MemorySet**	Copy count bytes of application memory to flash memory
meiFlash**MemoryToFile**	Save actual flash memory to cache or to file
meiFlash**MemoryVerify**	Compare actual flash memory to cache or to file

Relational Methods

meiFlash**Control**	Return handle of Control that is associated with Flash

Data Types

MEIFlash**Files**
MEIFlash**Message**

Copyright © 2002
Motion Engineering

meiFlashCreate

Declaration

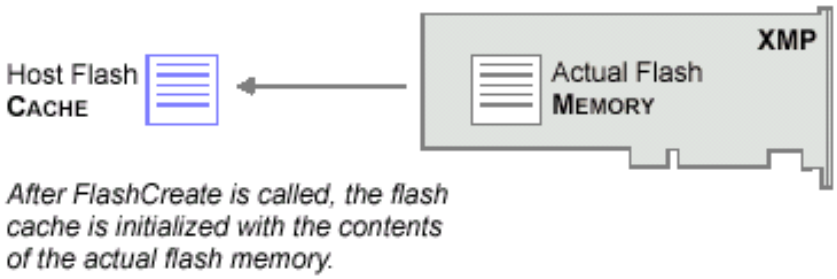
```
const MEIFlash meiFlashCreate(MPIControl control)
```

Required Header

stdmpi.h

Description

FlashCreate creates a Flash object and a host-resident copy of flash memory on motion controller ***control*** (called the flash cache). *FlashCreate* is the equivalent of a C++ constructor.



After FlashCreate is called, the flash cache is initialized with the contents of the actual flash memory.

Return Values

handle	to a Flash object
MPIHandleVOID	if the object could not be created

See Also

meiFlashDelete | meiFlashValidate

meiFlashDelete

Declaration `long meiFlashDelete(MEIFlash flash)`

Required Header stdmei.h

Description **FlashDelete** deletes a Flash object and invalidates its handle (***flash***). *FlashDelete* is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *FlashDelete* successfully deletes a Flash object and invalidates its handle

See Also meiFlashCreate | meiFlashValidate

meiFlashValidate

Declaration `long meiFlashValidate(MEIFlash flash)`

Required Header stdmei.h

Description **FlashValidate** validates the Flash object and its handle (***flash***).

Return Values	
MPIMessageOK	if Flash is a handle to a valid object.

See Also meiFlashCreate | meiFlashDelete

meiFlashConfigGet

Declaration

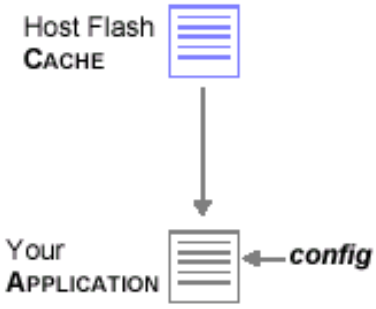
```
long meiFlashConfigGet(MEIFlash       flash,
                       MEIFlashConfig *config)
```

Required Header

stdmei.h

Description

FlashConfigGet gets the flash configuration and writes it into the structure pointed to by ***config***. The flash configuration includes data about the actual flash memory device (its type and size).



Return Values

MPIMessageOK	if *FlashConfigGet* successfully gets the flash configuration and writes it into the structure

See Also

meiFlashConfigSet

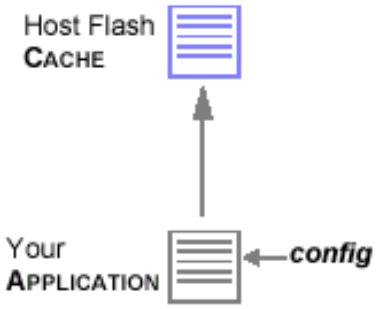
meiFlashConfigSet

Declaration

```
long meiFlashConfigSet(MEIFlash       flash,
                       MEIFlashConfig *config)
```

Required Header stdmei.h

Description **FlashConfigSet** sets (reads) the control configuration from the structure pointed to by ***config***.



Return Values	
MPIMessageOK	if *FlashConfigSet* successfully reads the control configuration from the structure

See Also meiFlashConfigGet

meiFlashDataGet

Declaration

```
long meiFlashDataGet(MEIFlash flash,
                     void     *dst,
                     void     *src,
                     long     count)
```

Required Header

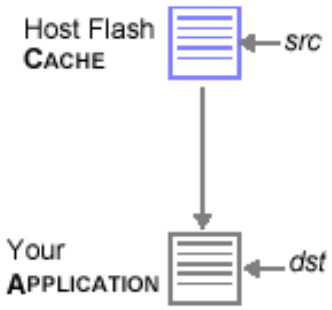
stdmei.h

Description

FlashDataGet gets ***count*** bytes of ***flash*** data memory starting at address ***src*** and puts (writes) them in application memory starting at address ***dst***. The ***src*** pointer must point into the **MEIXmpData{...}** structure defined in *xmp.h* and be based on the firmware address (MEIXmpData *) returned by mpiControlMemory(...).

Your application cannot access Flash memory directly; instead your application will access the host-resident flash memory cache maintained by ***flash***.

meiFlashDataGet(...) reads from the flash cache and is called only by applications and utilities, while **meiFlashMemoryGet(...)** is a low-level method that reads directly from actual flash memory and is called primarily by other flash methods.



Return Values

MPIMessageOK	if *FlashDataGet* successfully gets count bytes of flash data memory writes them to application memory

See Also

mpiControlMemory | meiFlashMemoryGet | meiFlashDataSet

meiFlashDataSet

Declaration

```
long meiFlashDataSet(MEIFlash flash,
                     void     *dst,
                     void     *src,
                     long     count)
```

Required Header

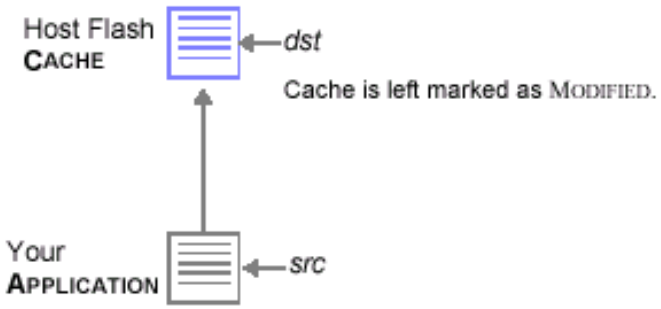
stdmei.h

Description

FlashDataSet sets (writes) ***count*** bytes of ***flash*** data memory (starting at address ***dst***) using application memory (starting at address ***src***). The ***dst*** pointer must point into the **MEIXmpData{...}** structure defined in *xmp.h* and be based on the firmware address (MEIXmpData *) returned by mpiControlMemory(...).

Your application cannot access Flash memory directly; instead your application will access the host-resident flash memory cache maintained by flash.

mpiControlMemory(...) returns an external pointer that points to the MEIXmpBufferData{...} structure. You cannot use this external pointer with the FlashDataSet method to access flash data memory.



Return Values

MPIMessageOK	if *FlashDataSet* successfully writes to the flash cache

See Also

mpiControlMemory | meiFlashDataGet

meiFlashMemoryErase

Declaration

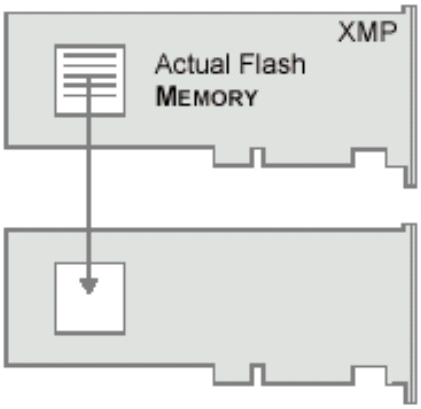
```
long meiFlashMemoryErase(MEIFlash flash,
                         long     erase,
                         long     verify)
```

Required Header

stdmei.h

Description

FlashMemoryErase erases actual flash memory, and also verifies that actual flash memory is erased.



flash	a handle to a Flash object
erase	a value, either zero or non-zero
verify	a value, either zero or non-zero

Return Values

MPIMessageOK	if *FlashMemoryErase* successfully erases the flash memory.

See Also

meiFlashCreate

meiFlashMemoryFromFile

Declaration

```
long meiFlashMemoryFromFile(MEIFlash      flash,
                            MEIFlashFiles *filesIn,
        MEIFlashFiles *filesOut)
```

Required Header stdmei.h

Description

FlashMemoryFromFile writes actual flash memory using the binary image contained in fileName, or using the flash memory cache.

Also, calling the meiFlashMemoryFromFile(...) method resets the flash cache modified indication to FALSE.

Return Values	
MPIMessageOK	if *FlashMemoryFromFile* successfully writes to actual flash memory using the file or using flash cache

See Also meiFlashMemoryFromFileType

meiFlashMemoryFromFileType

Declaration

```
long meiFlashMemoryFromFileType(MEIFlash          flash,
                                const char        *fileName,
                                MEIFlashFileType  fileType)
```

Required Header stdmei.h

Description

FlashMemoryFromFileType writes actual flash memory using the binary image contained in ***fileName***, or using the flash memory cache.

flash	a handle to a Flash object
***filename**	a string
fileType	an enumeration corresponding to the flash file types

Return Values

MPIMessageOK	if *FlashMemoryFromFileType* successfully copies the data from the file to the flash.

See Also meiFlashCreate | meiFlashMemoryFromFile

meiFlashMemoryGet

Declaration

```
long meiFlashMemoryGet(MEIFlash flash,
                       void     *dst,
                       void     *src,
                       long     count)
```

Required Header

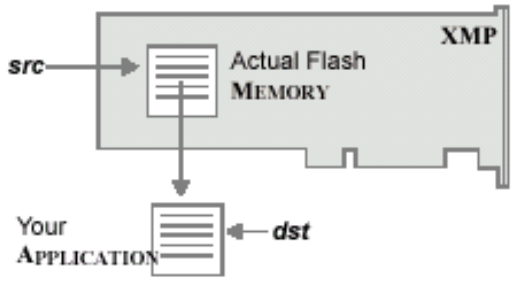
stdmei.h

Description

FlashMemoryGet copies ***count*** bytes of actual flash memory (***flash***, starting at address ***src***) to application memory (starting at address ***dst***).

You should calculate the ***src*** pointer by considering flash memory as a stream of bytes, because FlashMemoryGet will adjust the pointer for the actual type of flash memory.

meiFlashMemoryGet(...) is a low-level method that reads directly from actual flash memory and is called primarily by other flash methods, while **meiFlashDataGet(...)** reads from the flash cache and is called only by applications and utilities.



XMP Only ***flash*** memory is of type *unsigned long* *.

Return Values

MPIMessageOK	if *FlashMemoryGet* successfully copies flash memory to application memory

See Also

meiFlashMemorySet

meiFlashMemoryModified

Declaration

```
long meiFlashMemoryModified(MEIFlash flash,
                            long     *modified)
```

Required Header

stdmei.h

Description

FlashMemoryModified determines if the flash memory cache has been modified. Note that unless meiFlashDataSet(...) has been called previously, the meiFlashMemoryModified(...) method will always return False, regardless of whether the cache has been modified or not modified.

If the "flash cache"	*Then*
has been modified	*FlashMemoryModified* writes True to the location pointed to by modified
has not been modified	*FlashMemoryModified* writes False to the location pointed to by modified

Return Values

MPIMessageOK	if *FlashMemoryModified* successfully determines if the flash cache has been modified

See Also

meiFlashDataSet

meiFlashMemorySet

Declaration

```
long meiFlashMemorySet(MEIFlash flash,
                       void     *dst,
                       void     *src,
                       long     count)
```

Required Header

stdmei.h

Description

FlashMemorySet copies application memory (starting at address ***src***) to ***count*** bytes of flash memory (***flash***, starting at address ***dst***).

You should calculate the ***dst*** pointer by considering flash memory as a stream of bytes, because FlashMemoryGet will adjust the pointer for the actual type of flash memory.

XMP Only ***flash*** memory is of type *unsigned long* *.

Return Values

MPIMessageOK	if *FlashMemorySet* successfully copies application memory to flash memory

See Also

meiFlashMemoryGet

meiFlashMemoryToFile

Declaration

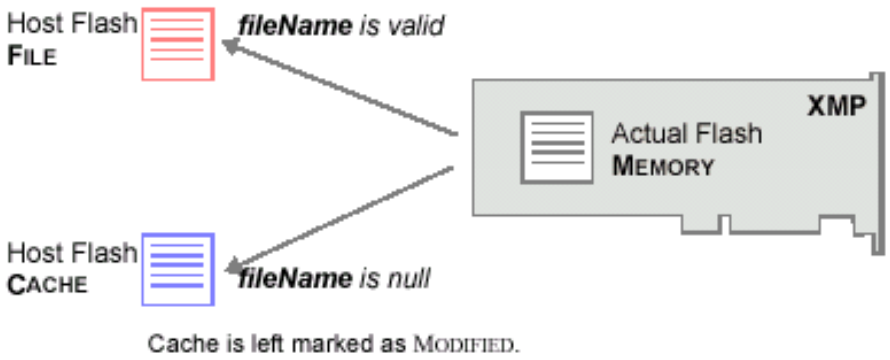
```
long meiFlashMemoryToFile(MEIFlash           flash,
                          const char         *fileName,
                          MEIFlashFileType   fileType)
```

Required Header

stdmei.h

Description

FlashMemoryToFile saves actual ***flash*** memory to a binary image contained in ***fileName***, or to the flash memory cache.



Cache is left marked as MODIFIED.

Return Values

MPIMessageOK	if *FlashMemoryToFile* successfully saves actual flash memory to file or to cache

See Also

meiFlashMemoryVerify

Declaration

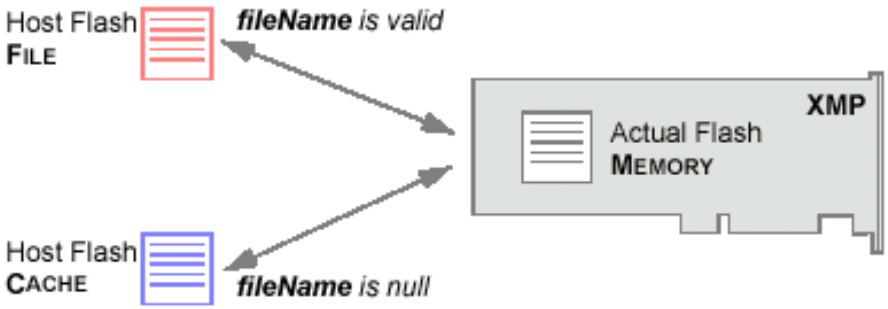
```
long meiFlashMemoryVerify(MEIFlash     flash,
                          const char   *fileName)
```

Required Header

stdmei.h

Description

FlashMemoryVerify compares actual ***flash*** memory to the binary image contained in ***fileName***, or to the flash memory cache (if ***fileName*** is Null). Every word of flash memory (including code and data) is compared with the firmware bin file (or cache if the ***fileName*** is Null). The firmware version number is contained in flash data memory, so even if the only difference is the firmware version number, flash verification will fail. Flash verification will fail if *any* word is different.



Cache is left marked as MODIFIED.

Return Values

MPIMessageOK	if actual flash memory is identical to the binary image in fileName or in cache
MPIMessageNo_Memory	if memory allocation error occurred
MPIMessageFILE_OPEN_ERROR	if the file (***fileName***) was not found
MPIMessageFILE_READ_ERROR	if an error occurred while reading the file (***fileName***)
MPIMessageFILE_CLOSE_ERROR	if an error occurred whileclosing the file (***fileName***)
MPIFlashMessageFLASH_READ_ERROR	if differences were detected during flash validation

See Also

meiFlashControl

Declaration

```
const MPIControl meiFlashControl(MEIFlash flash)
```

Required Header

stdmei.h

Description

FlashControl returns a handle to the motion controller (Control object) that a Flash object (***flash***) is associated with.

Return Values

handle	to a Control object that a Flash object is associated with
MPIHandleVOID	if the Flash object is invalid

See Also

MEIFlashFiles

MEIFlashFiles

```
typedef struct MEIFlashFiles {
    char binFile[MEIFlashFileMaxChars];
    char FPGAFile[MEIXmpFlashMaxFPGAFiles][MEIFlashFileMaxChars];
} MEIFlashFiles;
```

Description

FlashFiles is a configuration structure that specifies what files should be copied to the flash memory of the controller.

binFile	A string buffer that holds the filename of the .bin file to be flashed.
FPGAFile	An array of string buffers that holds the filenames of the FPGA files to be flashed.

See Also

MEIFlash

MEIFlashMessage

MEIFlashMessage

```
typedef enum {

    MEIFlashMessageFLASH_INVALID,
    MEIFlashMessageFLASH_READ_ERROR,
    MEIFlashMessageFLASH_WRITE_ERROR,
    MEIFlashMessagePATH,
} MEIFlashMessage;
```

Description

FlashMessage lists the error messages returned by the MEIFlash module.

MEIFlashMessageFLASH_INVALID	The flash object passed to meiFlash…() methods was invalid.
MEIFlashMessageFLASH_READ_ERROR	The flash object could not successfully read data.
MEIFlashMessageFLASH_WRITE_ERROR	The flash object could not successfully write data.
MEIFlashMessagePATH	The flash file path is too long.

See Also

MEIFlash

Global Objects

Introduction

Data types that are used by more than one module are defined in the mpidef.h header file.The definitions listed in this section are available to all modules, and are defined in mpidef.h.

The object.h header file contains the data type definition for a generic object handle (MPIHandle) and an invalid object handle (MPIHandleVOID). The object.h header file also contains declarations of generic mpiObject functions, the definitions for which are in the object.c source file. These functions take a generic object handle (of type MPIHandle) as their first argument. Any MPI object handle may be treated as an MPIHandle and passed as the first argument to these functions.

Data Types

MPI**Action**

MPI**IoSource**

MPI**IoType**

MPI**IoTrigger**

MPI**ModuleId** / MEI**ModuleId**

MPI**State**

MPI**Status**

MPI**StatusFlag**

MPI**StatusMask**

MPI**Trajectory**

MPI**Wait**

Macros

mpi**StatusMaskBIT**

Copyright © 2002
Motion Engineering

MPIAction

MPIAction

```
typedef enum {
    MPIActionINVALID,

    MPIActionNONE,
    MPIActionSTOP,
    MPIActionE_STOP,
    MPIActionE_STOP_ABORT,
    MPIActionABORT,

    MPIActionDONE,
    MPIActionSTART,
    MPIActionRESUME,
    MPIActionRESET,
} MPIAction;
```

Description

MPIActionNONE	Performs no action. Use with MPIMotorEventConfig to prevent a motor event from performing an action.
MPIActionSTOP	Makes a motion supervisor perform a stop. This action can be commanded with mpiMotionAction(…) or by a motor event on the controller. Please see MPIMotionDecelTime for more information about stop actions.
MPIActionE_STOP	Makes a motion supervisor perform an e-stop. This action can be commanded with mpiMotionAction(…) or by a motor event on the controller. Please see MPIMotionDecelTime for more information about e-stop actions.
MPIActionE_STOP_ABORT	Makes a motion supervisor perform an e-stop and then an abort. This action can be commanded with mpiMotionAction(…) or by a motor event on the controller. Please see MPIMotionDecelTime for more information about e-stop actions.
MPIActionABORT	Makes a motion supervisor perform an abort. This action can be commanded with mpiMotionAction(…) or by a motor event on the controller.
MPIActionDONE	is currently not supported and is reserved for future use.
MPIActionSTART	Intended to force a motion supervisor to start when it is waiting for some event (a delay or hold) before starting. This action is currently not supported.
MPIActionRESUME	Makes a motion supervisor to resume motion after a stop action has occurred. A motion supervisor can only resume a motion after a stop event, not an e-stop event. This action can be commanded with mpiMotionAction(…).

MPIActionRESET	Makes a motion supervisor return to an idle state after an error has occurred or after a stop, e-stop, abort, or e-stop/abort action has occurred. While abort actions and certain errors cause all associated motors to turn off their amp-enable lines, this action does not change the state of any amp-enable lines. One will have to call the method mpiMotorAmpEnableSet(…) to re-enable the amplifiers. This action can be commanded with mpiMotionAction(…).

Remarks

MPIAction enumerations are used to perform some sort of action on an MPI object. Currently, only MPIMotion and MPIMotor use the MPIAction enumerations. One can command an MPIMotion object to perform some action with the mpiMotionAction(…) method, while one can get and set the types of actions that will be performed when certain motor events occur with the MPIMotorEventConfig structure with the mpiMotorEventConfigGet(…) and mpiMotorEventConfigSet(…) methods.

An MPIAction can be generated from the host or the firmware. Below is a table where MPIActions originate (start):

MPIAction	Originating from Host	Originating from XMP Firmware
Start	mpiMotionAction(…) (currently unsupported)	NEVER
Resume	mpiMotionAction(…)	NEVER
Reset	mpiMotionAction(…)	NEVER
Stop	mpiMotionAction(…)	Event
E_Stop	mpiMotionAction(…)	Event
ABORT	mpiMotionAction(…)	Event
DONE	NEVER	NEVER

See Also

mpiMotionAction | MPIMotionDecelTime | MPIMotorEventConfig
mpiMotorEventConfigGet | mpiMotorEventConfigSet | MPIEvent

MPIIoSource

MPIIoSource

```
typedef union {
    MPIHandle   motor;  /* MOTOR */
    long        index;  /* USER */
} MPIIoSource;
```

Description

motor	Handle to a motor object that is the source for the IO.
index	Value of the index for a user input. User IO's are no longer supported by the xmp (user IO's are handled through the motor object).

See Also

MPIIoType

MPIIoType

```
typedef enum {
    MPIIoTypeINVALID,

    MPIIoTypeMOTOR,
    MPIIoTypeUSER,

} MPIIoType;
```

Description

MPIIoTypeMotor	Value specifies the IO type as motor (the IO source is a motor object).
MPIIoTypeUSER	Value specifies the IO type as user (User IO types are currently not supported. User IO is available through the motor objects).

See Also

MPIIoTrigger

MPIIoTrigger

```
typedef struct MPIIoTrigger {
    MPIIoType        type;
    MPIIoSource      source;
    unsigned long    mask;
    unsigned long    pattern;
} MPIIoTrigger;
```

Description

type	see MPIIoType.
source	see MPIIoSource.
mask	Value that specifies the mask to be applied to the IO.
pattern	Value that specifies the pattern to be compared to the masked IO.

See Also

MPIModuleId / MEIModuleId

MPIModuleId

```
typedef enum {
    MPIModuleIdINVALID,

    MPIModuleIdMESSAGE,
    MPIModuleIdADC,
    MPIModuleIdAXIS,
    MPIModuleIdCAPTURE,
    MPIModuleIdCOMMAND,
    MPIModuleIdCOMPARE,
    MPIModuleIdCONTROL,

    MPIModuleIdEVENT,
    MPIModuleIdEVENTMGR,
    MPIModuleIdFILTER,
    MPIModuleIdIDN,
    MPIModuleIdIDNLIST,
    MPIModuleIdMOTION,
    MPIModuleIdMOTOR,
    MPIModuleIdNODE,
    MPIModuleIdNOTIFY,
    MPIModuleIdPATH,
    MPIModuleIdRECORDER,
    MPIModuleIdSEQUENCE,
    MPIModuleIdSERCOS,
    MPIModuleIdSYNQNET,  /* This is a duplicate of MEIModuleIdSYNQNET
                            only for use in calling functions in xmp.c */
    MPIModuleIdBLOCK,    /* This is a duplicate of MEIModuleIdBLOCK only
                            for use in calling functions in xmp.c */

    MPIModuleIdEXTERNAL,

    MPIModuleIdMAX = 0xFF
} MPIModuleId;
```

Description

ModuleId is used to identify what module a particular MPIHandle belongs to. If the handle is an external memory pointer instead of an MPI object handle, MPIModuleIdEXTERNAL will be returned by MPI methods.

MEIModuleId

```
typedef enum {
    MEIModuleIdPLATFORM = MPIModuleIdEXTERNAL,

    MEIModuleIdCAN,

    MEIModuleIdCLIENT,
    MEIModuleIdELEMENT,
    MEIModuleIdFLASH,
    MEIModuleIdLIST,
    MEIModuleIdMAP,
    MEIModuleIdPACKET,
    MEIModuleIdSERVER,
    MEIModuleIdSYNQNET,
    MEIModuleIdBLOCK,
    MEIModuleIdSNDRIVE,
```

```
}MEIModuleId;
```

Description

ModuleId is used to identify what module a particular MPIHandle belongs to. If the handle is an external memory pointer instead of an MPI object handle, MPIModuleIdEXTERNAL will be returned by MPI methods.

See Also

mpiObjectModuleId | mpiObjectValidate

MPIState

MPIState

```
typedef enum {
    MPIStateINVALID,

    MPIStateINIT,
    MPIStateIDLE,
    MPIStateMOVING,
    MPIStateSTOPPING,
    MPIStateSTOPPING_ERROR,
    MPIStateERROR,
} MPIState;
```

Description

State enumerations define basic states the motion is in. MPIState resides in the MPIStatus structure. Currently MPIState is only used with motion module.

MPIStateINIT	The of the motion is performing an initialization.
MPIStateIDLE	The state of motion is idle and waiting to resume motion.
MPIStateMOVING	The state of the motion is moving.
MPIStateSTOPPING	The state of the motion is stopping. This occurs from a Stop event, but not an E_Stop, E_Stop Abort, or Abort events.
MPIStateSTOPPING_ERROR	The state of the motion is performing an emergency stop and/or abort on all axes.
MPIStateERROR	The state of the motion is in error. The error state is generated from an E_Stop or Abort event.

See Also

MPIStatus

MPIStatus

MPIStatus

```
typedef struct MPIStatus {
    MPIState        state;
    MPIAction       action;
    MPIEventMask    eventMask;

    long    settled;
    long    atTarget;

    MPIStatusMask    statusMask;
} MPIStatus;
```

Description

state	Value that indicates the state of an xmp controller's motion supervisor.
action	Value that indicates the action to perform for a motion supervisor.
eventMask	Array that defines the event mask bits. The array is defined as typedef MPIEventMaskELEMENT_TYPE MPIEventMask[MPIEventMaskELEMENTS].
settled	Value that indicates if an axis associated with a motion supervisor has settled (is in fine position).
atTarget	Value that indicates if an axis associated with a motion supervisor has completed its command trajectory (i.e. the command position has reached the targeted end point of the move).

See Also

Note on status information if using a SERCOS controller.
MPIState | MPIStatusFlag | MPIStatusMask

MPIStatusFlag

MPIStatusFlag

```
typedef enum {
    MPIStatusFlagINVALID,

    MPIStatusFlagCOMM_ERROR,

} MPIStatusFlag;
```

Description

See Also

Special Note on status information when using a SERCOS controller.

MPIStatusMask

MPIStatusMask

```
typedef enum {
    MPIStatusMaskNONE   = 0x0,

    MPIStatusMaskCOMM_ERROR     = mpiStatusMaskBIT(MPIStatusFlagCOMM_ERROR),
                                  /* 0x00000001 */

    MPIStatusMaskMOTOR  = MPIStatusMaskCOMM_ERROR,
                                  /* 0x00000001 */

    MPIStatusMaskALL    = mpiStatusMaskBIT(MPIStatusFlagLAST) - 1
                                  /* 0x00000001 */
} MPIStatusMask;
```

Description

MPIStatusMaskCOMM_ERROR	Value for the status mask that indicates a commutation error has occurred.
MPIStatusMaskMOTOR	Value specifies the motor's status mask.
MPIStatusMaskALL	Value specifies the status mask that encompasses all the possible status flags.

See Also

Note on status information if using a SERCOS controller.
MPIStatus | MPIStatusFlag

MPITrajectory

MPITrajectory

```
typedef struct MPITrajectory {
    double    velocity;
    double    acceleration;
    double    deceleration;
    double    jerkPercent;
    double    accelerationJerk;
    double    decelerationJerk;
} MPITrajectory;
```

Description

The **Trajectory** structure is within the MPIMotionVelocity structure which in turn is within the MPIMotionParams structure. The data contained in MPITrajectory are the parameters used in certain motion profiles.

See Also

MPIMotionParams | MPIMotionSCurve | MPIMotionTrapezoidal | MPIMotionType
MPIMotionVelocity | mpiMotionTrajectory

MPIWait

MPIWait

```
typedef enum {
    MPIWaitFOREVER = -1,
    MPIWaitPOLL = 0,
    MPIWaitMSEC
} MPIWait;
```

Description

Wait enumerations define basic wait times for certain MPI methods.

MPIWaitFOREVER	Makes MPI methods wait forever for an event to occur before returning.
MPIWaitPOLL	Makes MPI methods see if a certain event has occurred. If an event has not occurred, then the MPI method will generally return immediately returning the value MPIMessageTIMEOUT.
MPIWaitMSEC	Defines a period of one millisecond. If used alone, this will make MPI methods wait for one millisecond for an event occurs before returning. One can pass an an agument a multiple of MPIWaitMSEC to make MPI methods wait longer periods of time. For example, the following statement will make mpiPlatformKey wait 5 milliseconds for a user keystroke: mpiPlatformKey(5 * MPIWaitMSEC); If an event does not occur within the specified time, MPI methods will generally return the value MPIMessageTIMEOUT.

WARNING

The MPI depends on the ability of the operating system it is running on to be able to activate threads or put threads to sleep for a specified period of time in order for these times to be accurate. Microsoft Windows platforms are not real-time operating systems and are known to be unable to activate threads any quicker than 10 milliseconds. If you encounter a timing problem, it is likely an operating system timing issue.

See Also

mpiControlInterruptWait | mpiNotifyEventWait | mpiObjectTimeoutGet
mpiObjectTimeoutSet | meiPlatformKey

mpiStatusMaskBIT

Declaration `#define mpiStatusMaskBIT(flag) (0x1 << (flag))`

Required Header stdmpi.h

Description **StatusMaskBIT** converts the status flag into the status mask.

See Also MPIStatusFlag | MPIStatusMask

Idn Objects

Introduction

An **Idn** object represents an individual SERCOS IDN. An Idn object is a container for data that is sent to and received from a Node. Methods are provided to get and set the data associated with an Idn object, but these methods do not actually transmit an Idn object to or from a Node.

Methods

Create, Delete, Validate Methods

mpiIdn**Create**	Create Idn object
mpiIdn**Delete**	Delete Idn object
mpiIdn**Validate**	Validate Idn object

Configuration and Information Methods

mpiIdn**Copy**	Copy all element fields of one IDN object to another.
mpiIdn**FieldGet**	Get the value of an IDN (or all fields) from the drive.
mpiIdn**FieldSet**	Set the value of an IDN field. The inverse of mpiIdnFieldGet.
mpiIdn**NumberName**	Fill the buffer with the text form of the IDN number.
mpiIdn**NumberText**	Get the ASCII representation of an Idn objects IDNumber (S-X-XXXX or P-X-XXXX)

Data Types

MPIIdn**Data**
MPIIdn**DataType**
MPIIdn**Element**
MPIIdn**Field**
MPIIdn**Message**
MPIIdn**Number**
MPIIdn**Procedure**
MPIIdn**Value**
MPIIdn**VarLength**

Macros

mpiIdn**DataGET**	Get data type (Element 3) and operation data (Element 7)
mpiIdn**DataSET**	
mpiIdn**ElementGET**	
mpiIdn**FieldATTR**	
mpiIdn**FieldAttrDATA_TYPE**	
mpiIdn**FieldAttrGET**	
mpiIdn**FieldAttrMASK**	
mpiIdn**FieldAttrSET**	
mpiIdn**FieldAttrSHIFT**	
mpiIdn**NUMBER**	
mpiIdn**NumberDATA_BLOCK**	
mpiIdn**NumberGET**	
mpiIdn**NumberPARAMETER_SET**	

mpiIdn**NumberPRODUCT**
mpiIdn**NumberSET**
mpiIdn**NumberSTANDARD**
mpiIdn**NumberTYPE**

Constants

MPIIdn**DATA_SIZE_MAX**

MPIIdn**NAME_SIZE_MAX**

MPIIdn**NUMBER_SIZE_MAX**

MPIIdn**UNIT_SIZE_MAX**

Copyright © 2002
Motion Engineering

mpiIdnCreate

Declaration `const MPIIdn mpiIdnCreate(MPIIdnNumber number)`

Required Header stdmpi.h

Description **IdnCreate** creates an Idn object, where ***number*** argument is the initial SERCOS IDNumber to be assigned to this object.
IdnCreate is the equivalent of a C++ constructor.

Return Values

handle	to an Idn object
MPIHandleVOID	if the object could not be created

See Also mpiIdnDelete | mpiIdnValidate

mpiIdnDelete

Declaration `long mpiIdnDelete(MPIIdn idn)`

Required Header stdmpi.h

Description **IdnDelete** deletes an Idn object and invalidates its handle (***idn***).
IdnDelete is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *IdnDelete* successfully deletes the Idn object and invalidates its handle

See Also mpiIdnCreate | mpiIdnValidate

mpiIdnValidate

Declaration `long mpiIdnValidate(MPIIdn idn)`

Required Header stdmpi.h

Description **IdnValidate** validates the Idn object and its handle (***idn***).

Return Values	
MPIMessageOK	if Idn is a handle to a valid object.

See Also mpiIdnCreate | mpiIdnDelete

mpiIdnCopy

Declaration

```
long mpiIdnCopy(MPIIdn idn,
                MPIIdn dst)
```

Required Header

stdmpi.h

Description

IdnCopy copies all element fields of one IDN object to another.

Return Values

MPIMessageOK	if *IdnCopy* successfully copies all element fields to on IDN object to another

See Also

mpiIdnFieldGet

Declaration

```
long mpiIdnFieldGet(MPIIdn      idn,
                    MPIIdnField field,
                    void        *value)
```

Required Header

stdmpi.h

Description

IdnFieldGet gets the value of an IDN (or all fields) from the drive. The pointer "value" must point to enough memory to hold the value of the requested field. See MPIIdnElement for the type of each field.

Return Values

MPIMessageOK	if successful

See Also

MPIIdnElement | mpiIdnFieldSet | MPIIdnField

mpiIdnFieldSet

Declaration

```
long mpiIdnFieldSet(MPIIdn      idn,
                    MPIIdnField field,
                    void        *value)
```

Required Header

stdmpi.h

Description

IdnFieldSet sets the value of an IDN field. The inverse of mpiIdnFieldGet.

Return Values

MPIMessageOK	if successful

See Also

mpiIdnFieldGet | MPIIdnField

mpiIdnNumberName

Declaration

```
char * mpiIdnNumberName(MPIIdnNumber number,
                        char *buffer)
```

Required Header stdmpi.h

Description Given a pointer to a char buffer of length MPIIdnNUMBER_SIZE_MAX + 1, **IdnNumberName** fills the buffer with the text form of the IDN number.

Return Values

MPIMessageOK	if successful

See Also

mpiIdnNumberText

Declaration

```
long mpiIdnNumberText(MPIIdn      idn,
                      const char **text)
```

Required Header

stdmpi.h

Description

IdnNumberText sets “text” to point to the internal memory storing the text form of the IDN number.

Return Values

MPIMessageOK	if successful

See Also

MPIIdnData

MPIIdnData

```
typedef union {
    unsigned long       binary;
    MPIIdnNumber        idn;
    MPIIdnProcedure     procedure;
    char                *text;
    MPIIdnValue         value;
    MPIIdnVarLength     varLength;
} MPIIdnData;
```

Description

binary	0 or 1
idn	an IDN number
procedure	procedure IDN's data is a word that is a bit mask.
***text**	a char pointer that points to a text string.
value	a long or unsigned long.
valLength	IDN data whose length is not defined, and it's data can be of any number of types.

See Also

MPIIdnDATA_SIZE_MAX | MPIIdnDataType

MPIIdnDataType

MPIIdnDataType

```
typedef enum {
    MPIIdnDataTypeINVALID,
        /*
         * These data types correspond to the
         * SERCOS attribute TYPE field.
         * They should not be changed,
         * though new types may be appended
         */

    MPIIdnDataTypeBINARY,       /* 0x0 */
    MPIIdnDataTypeUNSIGNED,     /* 0x1 */
    MPIIdnDataTypeLONG,         /* 0x2 */
    MPIIdnDataTypeUNSIGNED_HEX, /* 0x4 */
    MPIIdnDataTypeTEXT,         /* 0x5 */
    MPIIdnDataTypeIDN,          /* 0x6 */

} MPIIdnDataType;
```

Description

IdnDataType corresponds to the SERCOS attribute TYPE field. It specifies the type of data that the IDN contains.

See Also

MPIIdnDATA_SIZE_MAX | MPIIdnData

MPIIdnElement

MPIIdnElement

```
typedef struct MPIIdnElement {
    MPIIdnNumber        number;         /* 0    MPIIdnFieldNUMBER       */
    const char          *name;          /* 1    MPIIdnFieldNAME         */
    unsigned long       attributes;     /* 2    MPIIdnFieldATTRIBUTES   */
    const char          *unit;          /* 3    MPIIdnFieldUNIT         */
    MPIIdnValue         minimum;        /* 4    MPIIdnFieldMINIMUM      */
    MPIIdnValue         maximum;        /* 5    MPIIdnFieldMAXIMUM      */
    MPIIdnData          data;           /* 6    MPIIdnFieldDATA         */
} MPIIdnElement;
```

Description

number	The IDN number field.
***name**	A pointer to the IDN name field.
attributes	The IDN attributes field.
***unit**	A pointer to the IDN unit field.
minimum	The IDN minimum field which is the smallest value the IDN data can be (not valid with certain data types).
maximum	The IDN maximum field which is the largest value the IDN data can be (not valid with certain data types).
data	The IDN data.

See Also

mpiIdnElementGET

MPIIdnField

MPIIdnField

```
typedef enum {
    MPIIdnFieldINVALID,

    MPIIdnFieldNUMBER,
    MPIIdnFieldNAME,
    MPIIdnFieldATTRIBUTES,
    MPIIdnFieldUNIT,
    MPIIdnFieldMINIMUM,
    MPIIdnFieldMAXIMUM,
    MPIIdnFieldDATA,

    MPIIdnFieldALL,
} MPIIdnField;
```

Description The **IdnField** enumeration is a list of all the different fields that comprises an IDN.

See Also mpiIdnFieldGet | mpiIdnFieldSet

MPIIdnMessage

MPIIdnMessage

```
typedef enum {

    MPIIdnMessageIDN_INVALID,
    MPIIdnMessageATTRIBUTES_INVALID,
} MPIIdnMessage;
```

Description

The **IdnMessage** is an enumeration that contains possible return values for IDN method calls.

See Also

MPIIdnNumber

MPIIdnNumber

```
typedef enum {
    MPIIdnNumberSTANDARD,
    MPIIdnNumberPRODUCT,

    MPIIdnNumberINVALID,

} MPIIdnNumber;
```

Description

IdnNumber is an IDN number, and also is used to specify whether the IDN is a standard or proprietary IDN.

See Also

MPIIdnNUMBER_SIZE_MAX

MPIIdnProcedure

MPIIdnProcedure

```
typedef enum {
    MPIIdnProcedureSET,
    MPIIdnProcedureENABLED,

    MPIIdnProcedureCANCEL,
    MPIIdnProcedureEXECUTE,

    MPIIdnProcedureRUNNING,
    MPIIdnProcedureERROR,
    MPIIdnProcedureDATA_VALID,
} MPIIdnProcedure;
```

Description

The data portion of a Procedure IDN is a bit map that represents the state of the procedure, this enumeration is a bit mask and can be used to decipher the status of the procedure.

See Also

MPIIdnValue

MPIIdnValue

```
typedef union {
    long                l;
    unsigned long       u;
} MPIIdnValue;
```

Description

IdnValue represents either an unsigned long or long.

See Also

MEIIdnVarLength

MEIIdnVarLength

```
typedef struct MPIIdnVarLength{
    long                count;

    MPIIdnDataType      type;

    union {
        MPIIdnNumber    *idn;
        char            *text;
        MPIIdnValue     *value;
        } as;
} MPIIdnVarLength;
```

Description

count	The size of the data buffer pointed to by **as**.
type	The type of data in the buffer.
as	A union of different pointer types, where the data is located.

See Also

mpiIdnDataGET

Declaration

```
#define mpiIdnDataGET(idn,idnData) \
        mpiIdnFieldGet((idn), MPIIdnFieldDATA, (idnData))
```

Required Header

stdmpi.h

Description

IdnDataGET reads the MPIIdnFieldDATA from the specified ***idn*** and writes it to the location pointed to by ***idnData***.

See Also

mpiIdnDataSET

mpiIdnDataSET

Declaration

```
#define mpiIdnDataSET(idn,idnData) \
        mpiIdnFieldSet((idn), MPIIdnFieldDATA, (idnData))
```

Required Header

stdmpi.h

Description

IdnDataSET writes the MPIIdnFieldDATA to the specified ***idn***, from the location pointed to by ***idnData***.

See Also

mpiIdnDataGET

mpiIdnElementGET

Declaration

```
#define mpiIdnElementGET(idn,idnElement) \
        mpiIdnFieldGet((idn), MPIIdnFieldALL, (idnElement))
```

Required Header

stdmpi.h

Description

IdnElementGET reads the MPIIdnFieldALL from the specified ***idn*** and writes it to the location pointed to by ***idnElement***.

See Also

MPIIdnElement

mpiIdnFieldATTR

Declaration

```
#define mpiIdnFieldATTR(mask,shift)
        (((mask) << 16) | (shift))
```

Required Header

stdmpi.h

Description

IdnFieldATTR ***shifts*** the ***mask*** up by 16 bits and bitwise ORs in the shift value.

See Also

mpiIdnFieldAttrGET | mpiIdnFieldAttrSET

mpiIdnFieldAttrDATA_TYPE

Declaration

```
#define mpiIdnFieldAttrDATA_TYPE(attr)
        ((MPIIdnDataType)mpiIdnFieldAttrGET  MPIIdnField AttrTYPE, (attr)))
```

Required Header

stdmpi.h

Description

IdnFieldAttrDATA_TYPE reads the ***MPIIdnDataType*** from the specified ***attr***.

See Also

mpiIdnFieldAttrGET

Declaration

```
#define mpiIdnFieldAttrGET(attr,word)
        (((word) >> iIdnFieldAttrSHIFT(attr)) &
        mpiIdnFieldAttrMASK(attr))
```

Required Header

stdmpi.h

Description

IdnFieldAttrGET reads the value of the specified ***attr*** (MPIIdnFieldAttr) from the ***word***.

See Also

mpiIdnFieldATTR

mpiIdnFieldAttrMASK

Declaration

```
#define     mpiIdnFieldAttrMASK(attr)
            ((( attr) >> 16) & 0xFFFF)
```

Required Header

stdmpi.h

Description

IdnFieldAttrMASK masks the value of the specified ***attr*** (MPIIdnFieldAttr). Used internally by mpiIdnFieldAttrGET/SET(…).

See Also

mpiIdnFieldAttrGET | mpiIdnFieldAttrSET

mpiIdnFieldAttrSET

Declaration

```
#define mpiIdnFieldAttrSET(attr,bits)
        (((bits) & mpiIdnFieldAttrMASK(attr)) <<
        mpiIdnFieldAttrSHIFT(attr))
```

Required Header

stdmpi.h

Description

IdnFieldAttrSET sets the value of the specified ***attr*** (MPIIdnFieldAttr) to ***bits***.

See Also

mpiIdnFieldAttrGET | mpiIdnFieldATTR

mpiIdnFieldAttrSHIFT

Declaration

```
#define mpiIdnFieldAttrSHIFT(attr)
        ((attr) & 0xFFFF)
```

Required Header

stdmpi.h

Description

IdnFieldAttrSHIFT shifts the value of the specified ***attr*** (MPIIdnFieldAttr). Used internally by mpiIdnFieldAttrGET/SET(…).

See Also

mpiIdnFieldAttrGET | mpiIdnFieldAttrSET

mpiIdnNUMBER

Declaration

```
#define mpiIdnNUMBER(type,param,data) \
((((type) & 0x1) << 15) | (((param) & 0x7) << 12) | ((data) & 0xFFF))
```

Required Header

stdmpi.h

Description

IdnNUMBER converts the idn type, param, and data into a number. Used internally by mpiIdnNumberPRODUCT/STANDARD(…).

See Also

mpiIdnNumberPRODUCT | mpiIdnNumberSTANDARD

mpiIdnNumberDATA_BLOCK

Declaration

```
#define     mpiIdnNumberDATA_BLOCK(number)
            ((number) & 0xFFF)
```

Required Header

stdmpi.h

Description

IdnNumberDATA_BLOCK determines the idn data block from a SERCOS identification ***number***.

See Also

mpiIdnNumberGET

Declaration

```
#define     mpiIdnNumberGET(idn,idnNumber) \
            mpiIdnFieldGet((idn), MPIIdnFieldNUMBER, (idnNumber))
```

Required Header

stdmpi.h

Description

IdnNumberGET gets the SERCOS identification number from the ***idn*** and writes it into the location pointed to by ***idnNumber***.

See Also

mpiIdnNumberSET

mpiIdnNumberPARAMETER_SET

Declaration

```
#define    mpiIdnNumberPARAMETER_SET(number)
          (((number) >> 12) & 0x7)
```

Required Header

stdmpi.h

Description

IdnNumberPARAMETER_SET determines the idn parameters from a SERCOS identification ***number***.

See Also

mpiIdnNumberPRODUCT

Declaration

```
#define  mpiIdnNumberPRODUCT(data)
        mpiIdnNUMBER(MPIIdnNumberPRODUCT, 0, (data))
```

Required Header

stdmpi.h

Description

IdnNumberPRODUCT converts an idn from a product specific value to the SERCOS identification number format.

See Also

mpiIdnNumberSET

Declaration

```
#define     mpiIdnNumberSET(idn,idnNumber) \
            mpiIdnFieldSet((idn), MPIIdnFieldNUMBER, (idnNumber))
```

Required Header

stdmpi.h

Description

IdnNumberSET sets the SERCOS identification number in the ***idn***, to the value from ***idnNumber***.

See Also

mpiIdnNumberGET

mpiIdnNumberSTANDARD

Declaration

```
#define  mpiIdnNumberSTANDARD(data)
         mpiIdnNUMBER(MPIIdnNumberSTANDARD, 0, (data))
```

Required Header stdmpi.h

Description **IdnNumberSTANDARD** converts an idn from a standard value to the SERCOS identification number format.

See Also

mpiIdnNumberTYPE

Declaration

```
#define mpiIdnNumberTYPE(number)
        (((number) >> 15) & 0x1)
```

Required Header

stdmpi.h

Description

IdnNumberTYPE determines the idn type (STANDARD or PRODUCT) from a SERCOS identification ***number***.

See Also

MPIIdnNumber

MPIIdnDATA_SIZE_MAX

MPIIdnDATA_SIZE_MAX

```
#define   MPIIdnDATA_SIZE_MAX   (65532)
```

Description

IdnDATA_SIZE_MAX defines the maximum data size an Idn object supports.

See Also

mpiIdnDataGET | MPIIdnData

MPIIdnNAME_SIZE_MAX

MPIIdnNAME_SIZE_MAX

```
#define MPIIdnNAME_SIZE_MAX     (60)
```

Description

IdnNAME_SIZE_MAX defines the maximum number of characters that can be used to represent the name of an IDN.

See Also

MPIIdnNUMBER_SIZE_MAX

MPIIdnNUMBER_SIZE_MAX

```
#define MPIIdnNUMBER_SIZE_MAX   (8)
```

Description

IdnNUMBER_SIZE_MAX defines the maximum number of characters that can be used in an IDN number string.

See Also

MPIIdnNumber

MPIIdnUNIT_SIZE_MAX

MPIIdnUNIT_SIZE_MAX

```
#define  MPIIdnUNIT_SIZE_MAX    (12)
```

Description

IdnUNIT_SIZE_MAX defines the maximum number of characters that can be used to represent the units used by an IDN.

See Also

IdnList Objects

Introduction

An **IdnList** object manages a list of SERCOS Idn objects.

Methods

Create, Delete, Validate Methods

mpiIdnList**Create**	Create IdnList object
mpiIdnList**Delete**	Delete IdnList object
mpiIdnList**Validate**	Validate IdnList object

Configuration and Information Methods

mpiIdnList**Copy**	Copy the contents of idnList to *dst*.
mpiIdnList**IdnNumber**	Read and write the identification numbers of an IdnList object.

Relational Methods

List Methods- for Idn Objects

mpiIdnList**Append**	Append an Idn to an IdnList
mpiIdnList**Count**	Count the number of Idns in an IdnList
mpiIdnList**First**	Get handle to first Idn in an IdnList
mpiIdnList**Idn**	Return the position of the element on the list indicated by "index."
mpiIdnList**Index**	Get index value to an Idn in an IdnList
mpiIdnList**Insert**	Insert an Idn into an IdnList
mpiIdnList**Last**	Get handle to last Idn in an IdnList
mpiIdnList**Get**	Get array of handles to Idns in an IdnList
mpiIdnList**Set**	Create a list (IdnList) containing idnCount Idn objects
mpiIdnList**Next**	Get handle to the Idn object just after a specified Idn object in an IdnList
mpiIdnList**IdnNumberFind**	
mpiIdnList**Previous**	Get handle to the Idn object preceding a specified Idn object in an IdnList
mpiIdnList**Remove**	Remove an Idn object from an IdnList

Copyright © 2002
Motion Engineering

mpiIdnListCreate

Declaration

```
const MPIIdnList mpiIdnListCreate(MPIIdn idn)
```

Required Header

stdmpi.h

Description

IdnListCreate creates an IdnList object, where the ***idn*** argument specifies the initial element in the list of Idn objects. Note that ***idn*** may be MPIHandleVOID. *IdnListCreate* is the equivalent of a C++ constructor.

Return Values

handle	to an IdnList object
MPIHandleVOID	if the object could not be created

See Also

mpiIdnListDelete | mpiIdnListValidate

mpiIdnListDelete

Declaration `long mpiIdnListDelete(MPIIdnList idnList)`

Required Header stdmpi.h

Description **IdnListDelete** deletes an IdnList object and invalidates its handle (***idnList***). Note that deleting an IdnList object **will also delete all of the Idn objects that it contains**. *IdnListDelete* is the equivalent of a C++ destructor.

Return Values	
MPIMessageOK	if *IdnListDelete* successfully deletes the IdnList object and invalidates its handle

See Also mpiIdnListCreate | mpiIdnListValidate

mpiIdnListValidate

Declaration `long mpiIdnListValidate(MPIIdnList idnList)`

Required Header stdmpi.h

Description **IdnListValidate** validates the IdnList object and its handle (***idnList***).

Return Values	
MPIMessageOK	if IdnList is a handle to a valid object.

See Also mpiIdnListCreate | mpiIdnListDelete

mpiIdnListCopy

Declaration

```
long mpiIdnListCopy(MPIIdnList idnList,
                    MPIIdnList dst)
```

Required Header

stdmpi.h

Description

IdnListCopy copies the contents of ***idnList*** to ***dst***.

Return Values

MPIMessageOK	if *IdnListCopy* successful

See Also

mpiIdnListIdnNumber

Declaration

```
long mpiIdnListIdnNumber(MPIIdnList   idnList,
                         MPIIdnNumber *number)
```

Required Header

stdmpi.h

Description

IdnListIdnNumber reads the identification numbers of an IdnList object (***idnList***) and writes it to an array of MPIIdnNumbers pointed to by ***number***.

idnList	a handle to the IdnList object.
***number**	a pointer to an array of MPIIdnNumbers returned by the method.

Return Values

MPIMessageOK	if *IdnListIdnNumber* successfully gets the list of identification numbers.

See Also

mpiIdnNumberGET

mpiIdnListAppend

Declaration

```
long mpiIdnListAppend(MPIIdnList idnList,
                      MPIIdn     idn)
```

Required Header

stdmpi.h

Description

IdnListAppend appends an Idn (***idn***) to an IdnList (***idnList***).

idnList	a handle to the IdnList object.
idn	a handle to an Idn object.

Return Values

MPIMessageOK	if *IdnListAppend* successfully appends the Idn to the IdnList
MPIMessageHANDLE_INVALID	Either ***idnList*** or ***idn*** is an invalid handle.
MPIMessageNO_MEMORY	Not enough memory was available.

See Also

mpiIdnListCount

Declaration `long mpiIdnListCount(MPIIdnList idnList)`

Required Header stdmpi.h

Description **IdnListCount** returns the number of elements on the list.

idnList	a handle to the IdnList object.

Return Values	
count	the number of elements in idnList
(-1)	if the idnList object is invalid

See Also

mpiIdnListFirst

Declaration

```
MPIIdn mpiIdnListFirst(MPIIdnList idnList)
```

Required Header

stdmpi.h

Description

IdnListFirst return the first element in the list. This function can be used in conjuntion with mpiIdnListIdnNext() in order to iterate through the list.

idnList	a handle to the IdnList object.

Return Values

handle	to the first Idn object of an IdnList (***idnList***)
MPIHandleVOID	if ***idnList*** is invalid if ***idnList*** is empty
MPIMessageHANDLE_INVALID	***idnList*** is an invalid handle.

See Also

mpiIdnListLast

mpiIdnListIdn

Declaration

```
MPIIdn mpiIdnListIdn(MPIIdnList idnList,
                     long       index,)
```

Required Header

stdmpi.h

Description

IdnListIdn returns the position of the element on the list indicated by "index."

idnList	a handle to the IdnList object.
index	a position in the list.

Return Values

MPIMessageARG_INVALID	if ***index*** is a negative number.
MEIListMessageELEMENT_NOT_FOUND	if ***index*** is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	if ***idnList*** is an invalid handle.

See Also

mpiIdnListIndex

Declaration

```
long mpiIdnListIndex(MPIIdnList idnList,
                     MPIIdn     idn)
```

Required Header

stdmpi.h

Description

IdnListIndex returns the position of "idn" on the list.

idnList	a handle to the IdnList object.
idn	a handle to an Idn object.

Return Values

index	of an Idn (***idn***) in an IdnList (***idnList***)
MPIHandleVOID	if ***idnList*** is invalid if ***idnList*** is empty

See Also

mpiIdnListInsert

Declaration

```
long mpiIdnListInsert(MPIIdnList idnList,
                      MPIIdn     idn,
                      MPIIdn     insert)
```

Required Header

stdmpi.h

Description

IdnListInsert inserts an Idn object (***insert***) in an IdnList (***idnList***), just after this Idn object (***idn***).

Return Values

MPIMessageOK	if *IdnListInsert* successfully inserts the Idn into the IdnList, in the slot after the specified Idn (***idn***)

See Also

mpiIdnListLast

Declaration

```
MPIIdn mpiIdnListLast(MPIIdnList idnList)
```

Required Header

stdmpi.h

Description

The last element in the list is returned. This function can be used in conjuntion with mpiIdnListIdnPrevious() in order to iterate through the list backwards.

idnList	a handle to the IdnList object.

Return Values

handle	to the last Idn object of an IdnList (***idnList***)
MPIHandleVOID	if ***idnList*** is invalid if ***idnList*** is empty
MPIMessageHANDLE_INVALID	if ***idnList*** is an invalid handle.

See Also

mpiIdnListFirst

mpiIdnListGet

Declaration

```
long mpiIdnListGet(MPIIdnList idnList,
                   long       *idnCount,
                   MPIIdn     *idnArray)
```

Required Header

stdmpi.h

Description

IdnListGet gets an array (that contains the handles of the Idn objects held by ***idnList***), and writes that array (of ***idnCount*** handles) to the location pointed to by idnArray, and also writes the number of Idn objects (held by ***idnList***) to the location pointed to by ***idnCount***.

Return Values

MPIMessageOK	if *IdnListGet* successfully writes the array of Idn hanldes and the number of Idn objects to the ***idnArray*** and ***idnCount*** locations (respectively)

See Also

mpiIdnListSet

Declaration

```
long mpiIdnListSet(MPIIdnList idnList,
                   long       idnCount,
                   MPIIdn     *idnArray)
```

Required Header

stdmpi.h

Description

IdnListSet creates a list (***idnList***) of ***idnCount*** Idn objects, using the Idn handles specified by ***idnArray***.Any existing list is completely replaced.

The ***idnArray*** argument is the address of an array of ***idnCount*** Idn handles, or is NULL (if ***idnCount*** is equal to zero).

You can also create an IdnList incrementally (one Idn at a time), by using the mpiIdnListAppend(...) and/or mpiIdnListInsert(...) methods. To specify the initial Idn object of an IdnList, use the ***idn*** argument of mpiIdnListCreate(...). You can use any mpiIdnList method to examine and manipulate an IdnList, regardless of how the IdnList was created.

Return Values

MPIMessageOK	if *IdnListSet* successfully creates the IdnList using the handles in ***idnArray***

See Also

mpiIdnListInsert | mpiIdnListAppend | mpiIdnListCreate | mpiIdnListGet

mpiIdnListNext

Declaration

```
MPIIdn mpiIdnListNext(MPIIdnList idnList,
                      MPIIdn     idn)
```

Required Header

stdmpi.h

Description

IdnListNext returns the next element following "idn" on the list. This function can be used in conjuntion with mpiIdnListIdnFirst() in order to iterate through the list.

idnList	a handle to the IdnList object.
idn	a handle to an Idn object.

Return Values

handle	to the Idn object following another Idn object (***idn***) in an IdnList (***idnList***)
MPIHandleVOID	if ***idnList*** is invalid if ***idnList*** is the last Idn object in the IdnList (***idnList***)
MPIMessageHANDLE_INVALID	Either ***idnList*** or ***idn*** is an invalid handle.

See Also

mpiIdnListPrevious

mpiIdnListIdnNumberFind

Declaration

```
long mpiIdnListIdnNumberFind(MPIIdnList   idnList,
                             MPIIdnNumber idnNumber
                             MPIIdn       *idn)
```

Required Header stdmpi.h

Description **IdnListIdnNumberFind** searches the ***idnList*** and writes to ***idn*** the handle to the MPIIdn object whose idn number is ***idnNumber*.**

Return Values

MPIMessageOK	if *IdnListIdnNumberFind* is successful

See Also

mpiIdnListPrevious

Declaration

```
MPIIdn mpiIdnListPrevious(MPIIdnList idnList,
                          MPIIdn     idn)
```

Required Header

stdmpi.h

Description

IdnListPrevious returns the previous element prior to "idn" on the list. This function can be used in conjuntion with mpiIdnListLast() in order to iterate through the list backwards.

idnList	a handle to the IdnList object.
idn	a handle to an Idn object.

Return Values

handle	to the Idn object preceding another Idn object (***idn***) in an IdnList (***idnList***)
MPIHandleVOID	if ***idnList*** is invalid if ***idnList*** is the first Idn object in the IdnList (***idnList***)
MPIMessageHANDLE_INVALID	Either ***idnList*** or ***idn*** is an invalid handle.

See Also

mpiIdnListLast | mpiIdnListNext

mpiIdnListRemove

Declaration

```
long mpiIdnListRemove(MPIIdnList idnList,
                      MPIIdn     idn)
```

Required Header

stdmpi.h

Description

IdnListRemove removes an Idn (***idn***) from an IdnList (***idnList***).

Return Values

MPIMessageOK	if *IdnListRemove* successfully removes the Idn object from the IdnList

See Also

Message Objects

Introduction

The **Message** module manages text strings associated with the error code return values. Each error return value has a unique value which can be decoded using the Message methods and macros.

Methods

Configuration and Information Methods

mpiMessage	
mpiMessage**Function**	Associate message text with a function

Macros

mpiMessage**ID**
mpiMessage**MODULE**
mpiMessage**NUMBER**

Copyright © 2002
Motion Engineering

mpiMessage

Declaration

```
const char *mpiMessage(long  messageId,
                       char  *messageText)
```

Required Header

stdmpi.h

Description

mpiMessage returns the text message associated with ***messageId*** and copies the text to ***messageText*** if ***messageText*** is not NULL.

If "messageText" is	*Then*
NULL	*Message* returns a pointer to message text resident in the library
not NULL	*Message* returns the pointer ***messageText***, which must point to a buffer large enough to hold the message text (that will be copied into it)

Return Values

NULL	if messageId is invalid
Pointer to an empty string (" ")	if the messge utility is not enabled or the message function pointer associated with *messageId* is NULL.
Pointer to a non-zero-length string	if the mpiMessage successsfully retrieves the message associated with *messageId*.

See Also

mpiMessageFunction

mpiMessageFunction

Declaration

```
long mpiMessageFunction(MPIModuleId        moduleId,
                        MPIMessageFunction function)
```

Required Header stdmpi.h

Description

MessageFunction registers ***function*** as the function to be called by **mpiMessage(...)** (in order to obtain the text for a message associated with module ***moduleId***).

MessageFunction is typically called internally by object create methods. Applications generally do not need to call *MessageFunction* directly.

Return Values	
MPIMessageOK	if *MessageFunction* successfully registers ***function*** as the function to be called by **mpiMessage(...)**

See Also mpiMessage

mpiMessageID

Declaration

```
#define mpiMessageID(module number) \
((long)((((module) & MPIModuleIdMAX) << 8) | \ ((number) & 0xFF)))
```

Required Header

stdmpi.h

Description

MessageID converts the message module value and number to a unique message identification value.

See Also

mpiMessage

mpiMessageMODULE

Declaration

```
#define mpiMessageMODULE(messageId) \
(((messageId) & (MPIModuleIdMAX << 8)) >> 8)
```

Required Header

stdmpi.h

Description

MessageMODULE converts the message identification value to the message module value.

See Also

mpiMessage

mpiMessageNUMBER

Declaration

```
#define mpiMessageNUMBER(messageId) ((messageId) & 0xFF)
```

Required Header

stdmpi.h

Description

MessageNUMBER converts the message identification value to the message number.

See Also

mpiMessage

Motion Objects

Introduction

A **Motion** object manages a single axis or group of axes. Its primary function is to provide an interface to command movement on a coordinate system. It also provides status information about all the axes under its control, so motion can be either stopped or resumed in a controlled manner, especially in the event of error recovery. The Motion object is really a host-based object, with a corresponding Motion Supervisor object in the controller. The Motion Supervisor handles the real-time issues associated with axis data and status synchronization.

Some careful consideration should be given to Motion object (Motion Supervisor) to axis mapping. While it's possible to have multiple Motion objects share the same axes, only one Motion Supervisor can command motion to an axis at a time. Motion object to axis maps can be changed dynamically at any time, but Motion Supervisor to axis maps should NOT be changed when the axes are moving.

To learn more about usind MPI Motion Attributes and MEI Motion Attributes, click here.

| Error Messages |

Methods

Create, Delete, Validate Methods

mpiMotion**Create**	Create Motion object
mpiMotion**Delete**	Delete Motion object
mpiMotion**Validate**	Validate Motion object

Configuration and Information Methods

mpiMotion**ConfigGet**	Get Motion configuration
mpiMotion**ConfigSet**	Set Motion configuration
mpiMotion**FlashConfigGet**	Get Motion flash config
mpiMotion**FlashConfigSet**	Set Motion flash config
mpiMotion**ParamsGet**	Get Motion parameters
mpiMotion**ParamsSet**	Set Motion parameters
meiMotion**ParamsValidate**	Validate Motion parameters
mpiMotion**PositionGet**	Get position parameters of all axes associated with Motion
mpiMotion**PositionSet**	Set position parameters of all axes associated with Motion
mpiMotion**Status**	Get Motion status
mpiMotion**Trajectory**	Get trajectories for all Axis associated with Motion

Event Methods

mpiMotion**EventNotifyGet**	Get event mask
mpiMotion**EventNotifySet**	Set event mask
mpiMotion**EventReset**	Reset events specified in event mask

Action Methods

mpiMotion**Action**	Perform an action on a Motion
mpiMotion**Modify**	Modify parameters of Motion while it is executing
mpiMotion**Start**	Start Motion (idle state > moving state)

Memory Methods

mpiMotion**Memory**	Set address to be used to access Motion memory
mpiMotion**MemoryGet**	Get bytes of Motion memory and place it into application memory
mpiMotion**MemorySet**	Put (set) bytes of application memory into Motion memory

Relational Methods

mpiMotion**Control**	Return handle of Control object associated with Motion
mpiMotion**Number**	Get index of Motion
mpiMotion**Axis**	Return handle of axis by index number
mpiMotion**AxisAppend**	Append axis to list
mpiMotion**AxisCount**	Return number of axes in list
mpiMotion**AxisFirst**	Return handle to first axis in list
mpiMotion**AxisIndex**	Return index value of an axis in list
mpiMotion**AxisInsert**	Insert axis into list associated with Motion
mpiMotion**AxisLast**	Return handle to last axis in list
mpiMotion**AxisListGet**	Get list of axes associated with Motion
mpiMotion**AxisListSet**	Create a list of axes associated with Motion
mpiMotion**AxisNext**	Get handle to next axis in list
mpiMotion**AxisPrevious**	Get handle to previous axis in list
mpiMotion**AxisRemove**	Remove handle to axis in list

Data Types

MPIMotion**Attr** / MEIMotion**Attr**

MEIMotion**AttrHold**

MEIMotion**AttrHoldSource**

MEIMotion**AttrHoldType**

MPIMotion**AttrMask** / MEIMotion**AttrMask**

MEIMotion**AttrOutput**

MEIMotion**AttrOutputType**

MPIMotion**Attributes** / MEIMotion**Attributes**

MPIMotion**BESSEL**

MPIMotion**BSPLINE**

MPIMotion**Config** / MEIMotion**Config**

MPIMotion**DecelTime**

MEIMotion**Frame**

MEIMotion**FrameBufferStatus**

MPIMotion**Message** / MEIMotion**Message**
MPIMotion**Params** / MEIMotion**Params**
MPIMotion**Point**
MPIMotion**PT**
MPIMotion**PVT**
MPIMotion**SCurve**
MPIMotion**SPLINE**
MEIMotion**Trace**
MPIMotion**Trapezoidal**
MPIMotion**Type** / MEIMotion**Type**
MPIMotion**Velocity**

Macros

mpiMotion**ATTR**
mpiMotion**AttrMaskBIT**
mpiMotion**TYPE**

Copyright © 2002
Motion Engineering

mpiMotionCreate

Declaration

```
const MPIMotion mpiMotionCreate(MPIControl control,
                                long       number,
                                MPIAxis    axis)
```

Required Header

stdmpi.h

Description

MotionCreate creates a Motion object associated with the motion supervisor identified by ***number*** located on motion controller ***control***. *MotionCreate* is the equivalent of a C++ constructor.

If ***number*** is **-1**, MotionCreate selects the next unused motion supervisor. The ***axis*** parameter specifies the initial element in the list of Axis objects which determine the coordinate system (axis may be MPIHandleVOID).

Return Values

handle	handle to a Motion object
MPIHandleVOID	if the object could not be created

See Also

mpiMotionDelete | mpiMotionValidate

mpiMotionDelete

Declaration `long mpiMotionDelete(MPIMotion motion)`

Required Header stdmpi.h

Description **MotionDelete** deletes a Motion object (***motion***) and invalidates its handle. *MotionDelete* is the equivalent of a C++ destructor.

Deleting a Motion object does not delete any of the Axis objects in the coordinate system.

Return Values

MPIMessageOK	if *MotionDelete* successfully deletes a Motion object and invalidates its handle

See Also mpiMotionCreate | mpiMotionValidate

mpiMotionValidate

Declaration `long mpiMotionValidate(MPIMotion motion)`

Required Header stdmpi.h

Description **MotionValidate** validates the Motion object (***motion***) and its handle. Always call *mpiMotionValidate* after creating a new Motion object.

Return Values

MPIMessageOK	if Motion is a handle to a valid object.

See Also mpiMotionCreate | mpiMotionDelete

mpiMotionConfigGet

Declaration

```
long mpiMotionConfigGet(MPIMotion          motion,
                        MPIMotionConfig    *config,
                        void               *external)
```

Required Header

stdmpi.h

Description

MotionConfigGet gets the configuration of a Motion object (***motion***) and puts (writes) it in the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type **MEIMotionConfig{}** or is NULL.

Return Values

MPIMessageOK	if *MotionConfigGet* successfully gets the configuration of a Motion object and writes it into the structure(s)

See Also

mpiMotionConfigSet | MEIMotionConfig

mpiMotionConfigSet

Declaration

```
long mpiMotionConfigSet(MPIMotion       motion,
                        MPIMotionConfig *config,
                        void            *external)
```

Required Header

stdmpi.h

Description

MotionConfigSet sets (writes) the configuration of a Motion object (***motion***) using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in *addition* to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	external either points to a structure of type **MEIMotionConfig{}** or is NULL.

Return Values

MPIMessageOK	if *MotionConfigSet* successfully writes the configuration of a Motion object using data from the structure(s)

See Also

mpiMotionConfigGet | MEIMotionConfig

mpiMotionFlashConfigGet

Declaration

```
long mpiMotionFlashConfigGet(MPIMotion        motion,
                             void             *flash,
                             MPIMotionConfig  *config,
                             void             *external)
```

Required Header

stdmpi.h

Description

MotionFlashConfigGet gets the flash configuration for a Motion object (motion) and puts (writes) it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Motion's flash configuration information in ***external*** is in addition to the Motion's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL). The implementation-specific ***flash*** argument is used to access flash memory.

XMP Only

external either points to a structure of type **MEIMotionConfig{}** or is **NULL**. ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *MotionFlashConfigGet* successfully gets the Motion's flash configuration and writes it into the structure(s).

See Also

mpiMotionFlashConfigSet

mpiMotionFlashConfigSet

Declaration

```
long mpiMotionFlashConfigSet(MPIMotion        motion,
                             void             *flash,
                             MPIMotionConfig  *config,
                             void             *external)
```

Required Header

stdmpi.h

Description

MotionFlashConfigSet sets (writes) the flash configuration of a Motion object (motion) using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Motion's flash configuration information in ***external*** is in addition to the Motion's flash configuration information in config, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL). The implementation-specific ***flash*** argument is used to access flash memory.

XMP Only

external either points to a structure of type MEIMotionConfig{} or is **NULL**. ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *MotionFlashConfigSet* successfully sets (writes) the Motion's flash configuration using data from the structure(s)

See Also

MEIMotionConfig | MEIFlash | mpiMotionFlashConfigGet

mpiMotionParamsGet

Declaration

```
long mpiMotionParamsGet(MPIMotion       motion,
                        MPIMotionParams *params)
```

Required Header

stdmpi.h

Description

MotionParamsGet reads the parameters of a Motion object (motion) and writes it to the structure pointed to by params. These motion parameters will be used if mpiMotionStart(…) is called with Null motion params.

motion	a handle to the Motion object
***params**	a pointer to the motion parameters structure returned by the method

Return Values

MPIMessageOK	if *MotionParamsGet* successfully returns the parameters associated with a Motion object
motion parameters	that are associated with a Motion object (***motion***). To set these motion parameters, call either mpiMotionParamsSet(...) or mpiMotionStart(...)

See Also

mpiMotionParamsSet | mpiMotionStart | meiMotionParamsValidate

mpiMotionParamsSet

Declaration

```
long mpiMotionParamsSet(MPIMotion       motion,
                        MPIMotionParams *params)
```

Required Header

stdmpi.h

Description

.**MotionParamsSet** sets the parameters of a Motion object (***motion***). These motion parameters will be used if mpiMotionStart(...) is called with a Null motion parameter.

If Motion (***motion***) is active, *MotionParamsSet* will set motion parameters "on-the-fly," i.e., at the first opportunity.

Return Values

MPIMessageOK	if *MotionParamsSet* successfully sets the parameters of a Motion object

See Also

mpiMotionStart | mpiMotionParamsGet

meiMotionParamsValidate

Declaration

```
long meiMotionParamsValidate(MPIMotion        motion,
                             MPIMotionType    type,
                             MPIMotionParams  *params,
                             MEIXmpAction     action,
                             long             *points)
```

Required Header

stdmei.h

Description

MotionParamsValidate validates the type-specific motion parameters pointed to by ***params***, using the coordinate system of ***motion***.

*If "*point" is*	*Then*
not Null	the number of points specified by ***params*** is written to the location (pointed to by ***points***)

Return Values

MPIMessageOK	if *MotionParamsValidate* successfully validates the type-specific motion parameters using the coordinate system of ***motion***

See Also

mpiMotionPositionGet

Declaration

```
long mpiMotionPositionGet(MPIMotion  motion,
                          double     *actual,
                          double     *command)
```

Required Header

stdmpi.h

Description

MotionPositionGet gets the actual and command position values for all axes associated with a Motion (***motion***). The ***actual*** and ***command*** arguments each point to an array with a size equal to the number of axes associated with ***motion***.

Return Values

MPIMessageOK	if *MotionPositionGet* successfully gets the actual and command position values for all axes associated with a Motion object

See Also

mpiMotionPositionSet

mpiMotionPositionSet

Declaration

```
long mpiMotionPositionSet(MPIMotion motion,
                          double    *actual,
                          double    *command)
```

Required Header

stdmpi.h

Description

MotionPositionSet sets the actual and command position values for all axes associated with a Motion (***motion***). The ***actual*** and ***command*** arguments each point to an array with a size equal to the number of axes associated with ***motion***.

Return Values

MPIMessageOK	if *MotionPositionSet* successfully sets the actual and command position values for all axes associated with a Motion object

See Also

mpiMotionPositionGet

mpiMotionStatus

Declaration

```
long mpiMotionStatus(MPIMotion  motion,
                     MPIStatus  *status,
                     void       *external)
```

Required Header

stdmpi.h

Description

MotionStatus gets a Motion's (***motion***) status and writes it to the structure pointed to by ***status***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Motion's status information in ***external*** is in addition to the Motion's status information in ***status***, i.e., the status information in ***status*** and in ***external*** is not the same information. Note that ***status*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEIMotionStatus{}** or is NULL.

Return Values

MPIMessageOK	if *MotionStatus* successfully writes the Motion's status to the structure(s)

See Also

mpiMotionTrajectory

Declaration

```
long mpiMotionTrajectory(MPIMotion      motion,
                         MPITrajectory *trajectory)
```

Required Header

stdmpi.h

Description

MotionTrajectory gets the trajectories for all axes associated with a Motion object (***motion***). The ***trajectory*** argument points to an array of MPITrajectory structures, with a size equal to the number of axes associated with the Motion object (***motion***).

Return Values

MPIMessageOK	if *MotionTrajectory* successfully gets the trajectories for all axes associated with a Motion object

See Also

mpiMotionEventNotifyGet

Declaration

```
long mpiMotionEventNotifyGet(MPIMotion     motion,
                             MPIEventMask *eventmask,
                             void         *external)
```

Required Header

stdmpi.h

Description

MotionEventNotifyGet writes the event mask (that specifies the event type(s) for which host notification has been requested) to the location pointed to by ***eventMask***, and also writes it into the implementation-specific location pointed to by ***external*** (if ***external*** is not NULL).

The event notification information in ***external*** is in addition to the event notification information in ***eventmask***, i.e, the event notification information in ***eventmask*** and in ***external*** is not the same information. Note that ***eventmask*** or ***external*** can be NULL (but not both NULL).

Event notification is enabled for event types specified in ***eventmask***, which is a bit mask generated **by the logical OR** of the MPIEventMask bits associated with the desired MPIEventType values.
Event notification is disabled for event types not specified in ***eventmask***.

XMP Only

external either points to a structure of type **MEIEventNotifyData{}** or is NULL.
The **MEIEventNotifyData{}** structure is an array of firmware addresses, whose contents are placed into the **MEIEventStatusInfo{}** structure (of all events generated by this object).

Return Values

MPIMessageOK	if *MotionEventNotifyGet* successfully writes the event mask to the location(s)

See Also

MPIEventType | MEIEventNotifyData | MEIEventStatusInfo
mpiMotionEventNotifySet

mpiMotionEventNotifySet

Declaration

```
long mpiMotionEventNotifySet(MPIMotion     motion,
                             MPIEventMask  eventmask,
                             void          *external)
```

Required Header

stdmpi.h

Description

MotionEventNotifySet requests host notification of the event(s) that are generated by ***motion*** and specified by ***eventMask***, and also specified by the implementation-specific location pointed to by ***external*** (if ***external*** is not NULL).

The event notification information in ***external*** is in addition to the event notification information in ***eventmask***, i.e, the event notification information in ***eventmask*** and in ***external*** is not the same information. Note that ***eventmask*** or ***external*** can be NULL (but not both NULL).

Event notification is enabled for event types specified in ***eventMask***, a bit mask generated by the bitwise OR of the MPIEventMask bits *associated with* the desired MPIEventType values. Event notification is disabled for event types that are not specified in ***eventMask***.

The mask of event types generated by a Motion object consists of bits from MPIEventMaskMOTION and MPIEventMaskAXIS.

XMP Only

external either points to a structure of type MEIEventNotifyData{} or is NULL. The MEIEventNotifyData{} structure is an array of firmware addresses, whose contents are placed into the MEIEventStatusInfo{} structure (of all events generated by this object).

To...	Then...
enable host notification of all events	set eventmask to MPIEventMaskALL
disable host notification of all events	set eventmask to MPIEventTypeNONE

Return Values

MPIMessageOK	if *MotionEventNotifySet* successfully requests host notification of the event(s) that are specified by ***eventMask*** and generated by ***motion***

See Also

MPIEventType | MEIEventNotifyData | MEIEventStatusInfo
mpiEventMaskMOTION | mpiEventMaskAXIS | mpiMotionEventNotifyGet

mpiMotionEventReset

Declaration

```
long mpiMotionEventReset(MPIMotion     motion,
                         MPIEventMask  eventMask)
```

Required Header

stdmpi.h

Description

MotionEventReset resets the event(s) that are specified in ***eventMask*** and generated by ***motion***.
Your application must call *MotionEventReset* only after one or more latchable events have occurred.

Return Values

MPIMessageOK	if *MotionEventReset* successfully resets the event(s) that are specified in ***eventMask*** and generated by ***motion***

See Also

mpiMotionAction

Declaration

```
long mpiMotionAction(MPIMotion motion,
                     MPIAction action)
```

Required Header

stdmpi.h

Description

MotionAction performs the specified action on the Motion object (motion).

The deceleration stop and Estop times can be set with:

mpiMotionConfigSet(MPIMotion **motion**, MPIMotionConfig ***config**, void ***external**);

where the structure MPIMotionConfig, contains the elements:

MPIMotionConfig.decelTime.stop /* seconds */
MPIMotionConfig.decelTime.eStop /* seconds */

If "action" is	*Then*
MPIActionABORT *or* **MPIActionE_STOP** *or* **MPIActionE_STOP_ABORT**	the Motion will perform an emergency stop and/or abort on all axes, and then change to the error state (MPIStateSTOPPING_ERROR to MPIStateERROR). Before starting another Motion, you must first make a call to mpiMotionAction(motion, MPIActionRESET).
MPIActionABORT	MotionAction will disable the PID control (or other algorithm), set the DAC output to the offset value, and disable the amp enable outputs. After the abort completes, the Motion will be set to the ERROR state .
MPIActionE_STOP	MotionAction will decelerate the axis (or axes) to a stop in the time specified by the “eStop” time. After the Motion stops, the Motion will be set to the ERROR state.
MPIActionE_STOP_ABORT	MotionAction will decelerate the axis (or axes) to a stop in the time specified by the “eStop” time. Next, MotionAction will generate an MPIActionABORT. After the abort completes, the Motion is set to the ERROR state. The E_STOP_ABORT first performs an E_STOP, and then performs an ABORT. The E_STOP decelerates the axis to a stop, then the ABORT disables PID control and disables the amp enable output.
MPIActionRESET	if the Motion (motion) is in the error state (MPIStateERROR), the Motion will change to the idle state (MPIStateIDLE), so that motion can be restarted. If the error state was caused by an Abort action, then the command position will be set equal to the actual position during the Reset.
MPIActionRESUME	if the Motion is in the idle state (MPIStateIDLE), it will change to the moving state (MPIStateMOVING), and the Motion will resume if it was stopped by a prior call to mpiMotionAction(motion, MPIActionSTOP).

MPIActionSTOP	if the Motion is in the moving state (MPIStateMOVING), it will performing a stop on all axes and change to the idle state (MPIStateSTOPPING to MPIStateIDLE). MPIActionSTOP decelerates the axis (or axes) to a stop in the time specified by the “stop” time. After the motion stops, the Motion is set to the ERROR state.

Return Values

MPIMessageOK	if *MotionAction* successfully performs the specified action on the Motion object

See Also

mpiMotionModify

Declaration

```
long mpiMotionModify(MPIMotion       motion,
                     MPIMotionType   type,
                     MPIMotionParams *params)
```

Required Header

stdmpi.h

Description

MotionModify modifies the parameters of a Motion object (***motion***) if motion is in progress (MPIStateMOVING). The types of motion whose parameters can be modified while moving are MPIMotionTypeTRAPEZOIDAL, MPIMotionTypeS_CURVE, MPIMotionTypeVELOCITY, MPIMotionTypePT and MPIMotionTypePVT.

Use the MPIMotionAttrAUTO_START attribute to automatically start a motion profile if the MotionModify call is made too late (i.e., after the previous move has finished).

Return Values

MPIMessageOK	if *MotionModify* successfully modifies the parameters of a Motion object
MPIMotionMessageIDLE	if no Motion was in progress when *MotionModify* was called. In order for *MotionModify* to work, there must be a Motion in progress.
MPIMotionMessageAUTO_START	if *MotionModify* was called when no motion was in progress and the Auto-Start attribute was specified. In this case, the MotionModify is automatically converted into a MotionStart.
MPIMotionMessagePROFILE_ERROR	if the controller cannot generate the motion profile (based on the specified motion parameters and attributes).
For more Returns, click here	

See Also

mpiMotionStart

Declaration

```
long mpiMotionStart(MPIMotion       motion,
                    MPIMotionType   type,
                    MPIMotionParams *params)
```

Required Header

stdmpi.h

Description

MotionStart changes a Motion object (***motion***) from the idle state (MPIStateIDLE) to the moving state (MPIStateMOVING), by initiating a motion of the given ***type*** using the specified parameters (***params***).If ***params*** is Null, then the motion parameters that were set by the most recent call to mpiMotionParamsSet(...) will be used to define the motion.

The coordinate system is defined by the ordered list of Axis object(s) that have been associated with the Motion object (***motion***). There must be at least one Axis in the coordinate system.

Return Values

MPIMessageOK	*MotionStart* successfully changes a Motion object from the idle state to the moving state
MPIMotionMessageMOVING	if *MotionStart* was called when a Motion is in progress
For more Returns, click here	

See Also

mpiMotionParamsSet | MPIState | See diagram on how mpiMotionStart works

mpiMotionMemory

Declaration

```
long mpiMotionMemory(MPIMotion  motion,
                     void       **memory)
```

Required Header

stdmpi.h

Description

MotionMemory sets (writes) the address [that is used to access a Motion's (***motion***) memory] to the contents of ***memory***. This address (or an address calculated from it) is passed as the ***src*** argument to mpiMotionMemoryGet(...), and also as the dst argument to mpiMotionMemorySet(...).

Return Values

MPIMessageOK	if *MotionMemory* successfully writes the address (used to access a Motion's memory) to the contents of memory

See Also

mpiMotionMemoryGet | mpiMotionMemorySet

mpiMotionMemoryGet

Declaration

```
long mpiMotionMemoryGet(MPIMotion motion,
                        void      *dst,
                        void      *src,
                        long      count)
```

Required Header

stdmpi.h

Description

MotionMemoryGet copies ***count*** bytes of a Motion's (***motion***) memory (starting at address ***src***) to application memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *MotionMemoryGet* successfully copies ***count*** bytes of a Motion's memory to application memory

See Also

mpiMotionMemorySet | mpiMotionMemory

mpiMotionMemorySet

Declaration

```
long mpiMotionMemorySet(MPIMotion motion,
                        void      *dst,
                        void      *src,
                        long      count)
```

Required Header

stdmpi.h

Description

MotionMemorySet copies ***count*** bytes of application memory (starting at address src) to a Motion's (***motion***) memory (starting at address dst).

Return Values

MPIMessageOK	if *MotionMemorySet* successfully copies ***count*** bytes of application memory to a Motion's memory

See Also

mpiMotionMemoryGet | mpiMotionMemory

mpiMotionControl

Declaration

```
const MPIControl mpiMotionControl(MPIMotion  motion)
```

Required Header

stdmpi.h

Description

MotionControl returns a handle to the Control object with which the motion is associated.

motion	a handle to the Motion object

Return Values

MPIControl	handle to a Control object
MPIHandleVOID	if ***motion*** is invalid

See Also

mpiMotionCreate | mpiControlCreate

mpiMotionNumber

Declaration

```
long mpiMotionNumber(MPIMotion  motion,
                     long       *number)
```

Required Header

stdmpi.h

Description

MotionNumber writes the index of a Motion object (***motion***, on the motion controller that the Motion object is associated with) to the contents of ***number***.

Return Values

MPIMessageOK	if *MotionNumber* successfully writes the index of a Motion object to the contents of ***number***

See Also

mpiMotionAxis

Declaration

```
const MPIAxis mpiMotionAxis(MPIMotion motion,
                            long      index)
```

Required Header

stdmpi.h

Description

MotionAxis returns the element at the position on the list indicated by ***index***.

motion	a handle to the Motion object.
index	a position in the list.

Return Values

handle	to the ***index***th Axis of a Motion (***motion***)
MPIHandleVOID	if ***motion*** is invalid if ***index*** is less than 0 if ***index*** is greater than or equal to **mpiMotionAxisCount(motion)**
MPIMessageARG_INVALID	***index*** is a negative number.
MEIListMessageELEMENT_NOT_FOUND	***index*** is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	***motion*** is an invalid handle.

See Also

mpiMotionAxisAppend

Declaration

```
long mpiMotionAxisAppend(MPIMotion  motion,
                         MPIAxis    axis)
```

Required Header

stdmpi.h

Description

MotionAxisAppend appends an Axis (***axis***) to a Motion (***motion***).

motion	a handle to the Motion object.
axis	a handle to an Axis object.

Return Values

MPIMessageOK	if *MotionAxisAppend* successfully appends an Axis to a Motion object
MPIMessageHANDLE_INVALID	Either ***motion*** or ***axis*** is an invalid handle.
MPIMessageUNSUPPORTED	The list already contains the maximum number of elements (MEIXmpMAX_COORD_AXES). **-or-** ***motion*** and axis are on different controllers.
MPIMessageOBJECT_NOT_ENABLED	***axis*** is not an enabled axis.
MPIMessageOBJECT_ON_LIST	***axis*** is already on the list.
MPIMessageNO_MEMORY	Not enough memory was available.

See Also

mpiMotionAxisCount

Declaration

```
long mpiMotionAxisCount(MPIMotion motion)
```

Required Header

stdmpi.h

Description

MotionAxisCount returns the number of elements on the list.

motion	a handle to the Motion object.

Return Values

number	of Axes in a Motion (***motion***)
-1	if ***motion*** is invalid
0	if ***motion*** is empty

See Also

mpiMotionAxis | mpiMotionAxisAppend

mpiMotionAxisFirst

Declaration

```
const MPIAxis mpiMotionAxisFirst(MPIMotion  motion)
```

Required Header

stdmpi.h

Description

MotionAxisFirst return the first element in the list. This function can be used in conjuntion with mpiMotionAxisNext() in order to iterate through the list.

motion	a handle to the Motion object.

Return Values

handle	to the first Axis of a Motion (***motion***)
MPIHandleVOID	if ***motion*** is invalid if ***motion*** is empty
MPIMessageHANDLE_INVALID	***motion*** is an invalid handle.

See Also

mpiMotionAxisNext | mpiMotionAxisLast

mpiMotionAxisIndex

Declaration

```
long mpiMotionAxisIndex(MPIMotion motion,
                        MPIAxis   axis)
```

Required Header

stdmpi.h

Description

MotionAxisIndex returns the position of "axis" on the list.

motion	a handle to the Motion object.
axis	a handle to an Axis object.

Return Values

index	of an Axis (***axis***) in a Motion (***motion***)
-1	if ***motion*** is invalid if the Axis (axis) was not found in the Motion (***motion***)

See Also

mpiMotionAxis

mpiMotionAxisInsert

Declaration

```
long mpiMotionAxisInsert(MPIMotion motion,
                         MPIAxis   axis,
                         MPIAxis   insert)
```

Required Header

stdmpi.h

Description

MotionAxisInsert inserts an Axis (***insert***) in a Motion (***motion***), just after the specified Axis (***axis***).

Return Values

MPIMessageOK	if *MotionAxisInsert* successfully inserts an Axis (***insert***) in a Motion (***motion***) following a specified Axis (***axis***)

See Also

mpiMotionAxis

mpiMotionAxisLast

Declaration

```
const MPIAxis mpiMotionAxisLast(MPIMotion motion)
```

Required Header

stdmpi.h

Description

MotionAxisLast returns the last element in the list. This function can be used in conjuntion with mpiMotionAxisPrevious() in order to iterate through the list backwards.

motion	a handle to the Motion object.

Return Values

handle	to the last Axis of a Motion (***motion***)
MPIHandleVOID	if ***motion*** is invalid if ***motion*** is empty
MPIMessageHANDLE_INVALID	Either ***motion*** or ***axis*** is an invalid handle.

See Also

mpiMotionAxisPrevious

mpiMotionAxisListGet

Declaration

```
long mpiMotionAxisListGet(MPIMotion  motion,
                          long       *axisCount,
                          MPIAxis    *axisList)
```

Required Header

stdmpi.h

Description

MotionAxisListGet gets the coordinate system of a Motion object (***motion***).

MotionAxisListGet writes the number of axes (in the coordinate system) to a location (pointed to by ***axisCount***), and also writes an array (of ***axisCount*** Axis handles) to another location (pointed to by ***axisList***).

Return Values

MPIMessageOK	if *MotionAxisListGet* successfully gets the coordinate system of a Motion object

See Also

mpiMotionAxisListSet | mpiMotionAxis

mpiMotionAxisListSet

Declaration

```
long mpiMotionAxisListSet(MPIMotion  motion,
                          long       axisCount,
                          MPIAxis    *axisList)
```

Required Header

stdmpi.h

Description

MotionAxisListSet creates a coordinate system of ***axisCount*** dimensions, using the Axis handles specified by ***axisList***. Any existing coordinate system is completely replaced.

The ***axisList*** parameter is the address of an array of ***axisCount*** Axis handles, or is ***NULL*** (if ***axisCount*** is equal to zero).

A coordinate system may also be created incrementally (i.e. one Axis at a time) by using the append and/or insert methods described in this section. The initial Axis of a coordinate system may be specified using the ***axis*** parameter of mpiMotionCreate(...). The list methods in this section may be used to examine and manipulate a coordinate system (i.e. axis list) regardless of how it was created.

Return Values

MPIMessageOK	if *MotionAxisListSet* successfully creates a coordinate system

See Also

mpiMotionCreate | mpiMotionAxisListGet | mpiMotionAxis

mpiMotionAxisNext

Declaration

```
const MPIAxis mpiMotionAxisNext(MPIMotion motion,
                                MPIAxis   axis)
```

Required Header

stdmpi.h

Description

MotionAxisNext returns the next element following "axis" on the list. This function can be used in conjuntion with mpiMotionAxisFirst() in order to iterate through the list.

motion	a handle to the Motion object.
axis	a handle to an Axis object.

Return Values

handle	to the Axis just after the specified Axis (***axis***) in a Motion (***motion***)
MPIHandleVOID	if ***motion*** is invalid if the specified Axis (***axis***) is the last axis in a Motion (***motion***)
MPIMessageHANDLE_INVALID	Either ***motion*** or ***axis*** is an invalid handle.

See Also

mpiMotionAxisFirst | mpiMotionAxisPrevious

mpiMotionAxisPrevious

Declaration

```
const MPIAxis mpiMotionAxisPrevious(MPIMotion motion,
                                    MPIAxis   axis)
```

Required Header

stdmpi.h

Description

MotionAxisPrevious returns the previous element prior to "axis" on the list. This function can be used in conjuntion with mpiMotionAxisLast() in order to iterate through the list backwards.

motion	a handle to the Motion object.
axis	a handle to an Axis object.

Return Values

handle	to the Axis preceding the specified Axis (***axis***) in a Motion (***motion***)
MPIHandleVOID	if ***motion*** is invalid if the specfied Axis (***axis***) is the first Axis in a Motion (***motion***)
MPIMessageHANDLE_INVALID	Either ***motion*** or ***axis*** is an invalid handle.

See Also

mpiMotionAxisLast | mpiMotionAxisNext

mpiMotionAxisRemove

Declaration

```
long mpiMotionAxisRemove(MPIMotion motion,
                         MPIAxis   axis)
```

Required Header stdmpi.h

Description **MotionAxisRemove** removes an Axis (***axis***) from a Motion (***motion***).

Return Values

MPIMessageOK	if *MotionAxisRemove* successfully removes the Axis from the Motion object

See Also

MPIMotionAttr / MEIMotionAttr

MPIMotionAttr

```
typedef enum {
    MPIMotionAttrINVALID,

    MPIMotionAttrAPPEND,
    MPIMotionAttrAUTO_START,
    MPIMotionAttrDELAY,
    MPIMotionAttrMaskID,
    MPIMotionAttrELEMENT_ID,
    MPIMotionAttrRELATIVE,
    MPIMotionAttrSYNC_END,
    MPIMotionAttrSYNC_START,
    MPIMotionAttrCOUNT,
} MPIMotionAttr;
```

Description

The motion attributes are used to generate the motion attribute masks to enable features with mpiMotionStart(…) and mpiMotionModify(…). Please see MPIMotionAttrMask data type for more information.

MEIMotionAttr

```
typedef enum {

    MEIMotionAttrEVENT,
    MEIMotionAttrFINAL_VEL,
    MEIMotionAttrNO_REVERSAL,
    MEIMotionAttrHOLD,
    MEIMotionAttrOUTPUT,

    MEIMotionAttrCOUNT,
} MEIMotionAttr;
```

Description

The motion attributes are used to generate the motion attribute masks to enable features with mpiMotionStart(…) and mpiMotionModify(…). Please see MPIMotionAttrMask for more information.

See Also MPIMotionAttrMask | mpiMotionStart | mpiMotionModify

MEIMotionAttrHold

MEIMotionAttrHold

```
typedef struct MEIMotionAttrHold {
    MEIMotionAttrHoldType       type;
    MEIMotionAttrHoldSource     source;
    float                       timeout;
} MEIMotionAttrHold;
```

Description

type	This value specifies the motion hold type. Please see MEIMotionAttrHoldType for more information.
source	This value specifies the motion hold conditions. Please see MEIMotionAttrHoldSource for more information.
timeout	This value specifies the motion hold expiration time (seconds). When the time exceeds the timeout value or the hold conditions are met, the motion profile will execute.

See Also

MEIMotionAttrHoldType | MEIMotionAttrHoldSource

MEIMotionAttrHoldSource

MEIMotionAttrHoldSource

```
typedef union {
    long        gate;
    struct {
        long    *input;
        long    mask;
        long    pattern;
    } input;
    struct {
        long    number;
        long    mask;
        long    pattern;
        } motor;
} MEIMotionAttrHoldSource;
```

Description

gate	This value specifies the control gate number when MEIMotionAttrHoldTypeGATE is used. Valid values are between 0 and 31. See meiControlGateGet/Set(…) for more information.
input (input)	This value specifies the input address when MEIMotionAttrHoldTypeINPUT is used.
mask (input)	This value specifies the AND mask when MEIMotionAttrHoldTypeINPUT is used.
pattern (input)	This value specifies the comparison pattern when MEIMotionAttrHoldTypeINPUT is used. The value at input.input is bit-wise ANDed with the input.mask and compared to the input.pattern.
number (motor)	This value specifies the motor number when MEIMotionAttrHoldTypeMOTOR is used.
mask (motor)	This value specifies the AND mask when MEIMotionAttrHoldTypeMOTOR is used.
pattern (motor)	This value specifies the comparison pattern when MEIMotionAttrHoldTypeMOTOR is used. The motor input word is bit-wise ANDed with the motor.mask and compared to the motor.pattern.

See Also

MEIMotionAttrHoldType | meiControlGateGet | meiControlGateSet

MEIMotionAttrHoldType

MEIMotionAttrHoldType

```
typedef enum {
    MEIMotionAttrHoldTypeINVALID,
    MEIMotionAttrHoldTypeNONE,
    MEIMotionAttrHoldTypeGATE,
    MEIMotionAttrHoldTypeINPUT,
    MEIMotionAttrHoldTypeMOTOR,
} MEIMotionAttrHoldType;
```

Description

MEIMotionAttrHoldTypeNONE	This type disables the hold trigger condition.
MEIMotionAttrHoldTypeGATE	This type configures a control gate for the hold trigger condition. See meiControlGateGet/Set(…) for more information.
MEIMotionAttrHoldTypeINPUT	This type configures a memory address for the hold trigger condition.
MEIMotionAttrHoldTypeMOTOR	This type configures a motor's input bit(s) for the hold trigger condition.

Remarks

These types specify the motion profile trigger condition. The hold trigger value is specified with the MEIMotionAttrHoldSource data type.

See Also

MEIMotionAttrHoldSource | MEIMotionAttrHold

MPIMotionAttrMask / MEIMotionAttrMask

MPIMotionAttrMask

```
typedef enum {
    MPIMotionAttrMaskAPPEND,
    MPIMotionAttrMaskAUTO_START,
    MPIMotionAttrMaskDELAY,
    MPIMotionAttrMaskID,
    MPIMotionAttrMaskELEMENT_ID,
    MPIMotionAttrMaskRELATIVE,
    MPIMotionAttrMaskSYNC_END,
    MPIMotionAttrMaskSYNC_START,
    MPIMotionAttrMaskALL,
} MPIMotionAttrMask;
```

Description

MPIMotionAttrMaskAPPEND	This mask enables the motion profile to be added to the end of a previous motion profile, in the controller's memory buffer. The APPENDed profile will begin execution after the previous profile has completed and the settling criteria has been met. The APPEND mask can be used with mpiMotionStart(…) or mpiMotionModify(…).
MPIMotionAttrMaskAUTO_START	This mask converts a mpiMotionModify(…) call to a mpiMotionStart(…) if the modify occurs after the previous motion profile has completed. If the previous profile had completed, then mpiMotionModify(…) will return an error code, MPIMotionMessageAUTO_START.
MPIMotionAttrMaskDELAY	This mask enables a time delay (seconds) before the motion profile begins. Please see MPIMotionAttributes for more information. This mask can be used with mpiMotionStart(…).
MPIMotionAttrMaskID	This mask enables an identification tag to be stored in the motion profile. Please see MPIMotionAttributes for more information. This mask can be used with mpiMotionStart(…) and mpiMotionModify(…).
MPIMotionAttrMaskELEMENT_ID	- This mask enables an identification tag to be stored in the path motion profiles. Please see MPIMotionAttributes for more information.
MPIMotionAttrMaskRELATIVE	This mask changes the profile target position from absolute to relative coordinates.
MPIMotionAttrMaskSYNC_END	This mask synchronizes the motion profiles for multiple axes so they will all end at the same time. Delays are inserted before the shorter profiles. When enabled, each axis will use its own MPITrajectory values.
MPIMotionAttrMaskSYNC_START	This mask synchronizes the motion profiles for multiple axes so they will all start at the same time. Delays are inserted after the shorter profiles. When enabled, each axis will use its own MPITrajectory values.

Remarks

The motion attribute masks are used to enable features with mpiMotionStart(…) and mpiMotionModify(…). The masks are **OR**ed with the MPIMotionType to enable each feature.

MEIMotionAttrMask

```
typedef enum {
    MEIMotionAttrMaskEVENT,
    MEIMotionAttrMaskFINAL_VEL,
    MEIMotionAttrMaskNO_REVERSAL,
    MEIMotionAttrMaskHOLD,
    MEIMotionAttrMaskOUTPUT,
    MEIMotionAttrMaskALL,
} MEIMotionAttrMask;
```

Description

MEIMotionAttrMaskEVENT	This mask allows the user to specify an MPIEventMask during a motion.
MEIMotionAttrMaskFINAL_VEL	This mask allows the user to specify a non-zero target velocity for point to point motion types.
MEIMotionAttrMaskNO_REVERSAL	This mask prevents a motion profile from changing direction.
MEIMotionAttrMaskHOLD	This mask prevents a motion profile from executing until the specified trigger conditions are met.
MEIMotionAttrMaskOUTPUT	This mask allows the user to set or clear bits during a motion.

See Also

mpiMotionStart | mpiMotionModify | MPIMotionType | MPITrajectory | MPIEventMask

MEIMotionAttrOutput

MEIMotionAttrOutput

```
typedef struct MEIMotionAttrOutput {
    MEIMotionAttrOutputType     type;
    union {
        long    *output;
        long    motor;
    } as;
    long        mask;
    long        pattern;
    long        pointIndex;     /* MEIMotionAttrMaskOUTPUT for path motion - point
                        index for turning on output - used with point lists */
} MEIMotionAttrOutput;
```

Description

type	This value specifies the output type to determine the output bits to be set or cleared.
***output**	This value specifies the memory address when MEIMotionAttrOutputTypeOUTPUT is used.
motor	This value specifies the motor number when MEIMotionAttrOutputTypeMOTOR is used.
mask	This value specifies the AND mask when MEIMotionAttrHoldTypeOUTPUT is used.
pattern	This value specifies the comparison pattern when MEIMotionAttrHoldTypeOUTPUT is used. The motor output word or output address is bit-wise ANDed with the mask and compared to the pattern.
pointIndex	This value specifies an index to a point, when multiple point motion is used.

See Also

MEIMotionAttrOutputType

MEIMotionAttrOutputType

MEIMotionAttrOutputType

```
typedef enum {
    MEIMotionAttrOutputTypeINVALID,
    MEIMotionAttrOutputTypeNONE,
    MEIMotionAttrOutputTypeMOTOR,
    MEIMotionAttrOutputTypeOUTPUT,
} MEIMotionAttrOutputType;
```

Description

MEIMotionAttrOutputTypeNONE	This type disables the setting/clearing of output bit(s) during motion.
MEIMotionAttrOutputTypeMOTOR	This type configures a motor's output bit(s) to be set or cleared during motion.
MEIMotionAttrOutputTypeOUTPUT	This type configures bit(s) at a memory address to be set or cleared during motion.

See Also

MEIMotionAttrOutput

MPIMotionAttributes / MEIMotionAttributes

MPIMotionAttributes

```
typedef struct MPIMotionAttributes {
    double      *delay;         /* MPIMotionAttrMaskDELAY */
    long        id;             /* MPIMotionAttrMaskID */
    long        *elementId;     /* MPIMotionAttrMaskELEMENT_ID */
} MPIMotionAttributes;
```

Description

delay	This array defines the delay time (seconds) before a motion profile begins execution.
id	This value defines the identity for a point to point motion.
elementId	This array defines the identity for each element of a path motion.

MEIMotionAttributes

```
typedef struct MEIMotionAttributes {
    MPIEventMask        eventMask;      /* MEIMotionAttrMaskEVENT */
    double              *finalVelocity; /* MEIMotionAttrMaskFINAL_VEL */
    MEIMotionAttrHold   *hold;          /* MEIMotionAttrMaskHOLD */
    long                *outputCount;   /* MEIMotionAttrMaskOUTPUT for path motion
                                        - number of outputs - per axis */
    MEIMotionAttrOutput *output;        /* MEIMotionAttrMaskOUTPUT for path and non
                                        path motion - outputs - per axis */
} MEIMotionAttributes;
```

Description

eventMask	This structure specifies the mask to enable event generation. See MPIEventMask for more information.
***finalVelocity**	This array specifies the target velocity for each axis when MEIMotionAttrMaskFINAL_VEL is used.
***hold**	This array specifies the hold configurations for each axis when MEIMotionAttrMaskHOLD is used.
***outputCount**	This array specifies the number of points per axis, to set/clear an output when MEIMotionAttrMaskOUTPUT is used.
***output**	This structure specifies the output configuration for each axis when MEIMotionAttrMaskOUTPUT is used.

See Also

MPIEventMask | MEIMotionAttrMask

MPIMotionBESSEL

MPIMotionBESSEL

```
typedef struct MPIMotionBESSEL {
    long        pointCount;
    double      *position;
    double      *time;

    MPIMotionPoint  point;
} MPIMotionBESSEL;
```

Description

positionCount	This value specifies the number of points.
***position**	This array stores the positions for the motion profile. There is one position value per point, per axis. The length of the array must be equal to pointCount multiplied by the number of axes. The positions are interleaved in the array by the axis index.
***time**	This array stores the times for the motion profile. There is one time value per point. The time specifies the number of seconds between the specified position, and the next position (point). The length of the time array must be equal to pointCount.
point	This structure contains the point configuration. Please see MPIMotionPoint data type for more information.

See Also

MPIMotionPoint

MPIMotionBSPLINE

MPIMotionBSPLINE

```
typedef struct MPIMotionBSPLINE {
    long                pointCount;
    double              *position;
    double              *time;

    MPIMotionPoint      point;
} MPIMotionBSPLINE;
```

Description

pointCount	This value specifies the number of points.
***position**	This array stores the positions for the motion profile. There is one position value per point, per axis. The length of the array must be equal to pointCount multiplied by the number of axes. The positions are interleaved in the array by the axis index.
***time**	This array stores the times for the motion profile. There is one time value per point. The time specifies the number of seconds between the specified position, and the next position (point). The length of the time array must be equal to pointCount.
point	This structure contains the point configuration. Please see MPIMotionPoint data type for more information.

See Also

MPIMotionPoint

MPIMotionConfig / MEIMotionConfig

MPIMotionConfig

```
typedef struct MPIMotionConfig {
    MPIMotionDecelTime  decelTime;
    float               normalFeedrate;
    float               pauseFeedrate;
}MPIMotionConfig;
```

Description

decelTime	This structure defines the deceleration time for Stop and E-Stop actions. Please see MPIMotionDecelTime data type documentation for more information.
normalFeedrate	This value defines the normal feed speed rate. The default value is 1.0 (100%).
pauseFeedrate	This value defines the feed speed rate for the Stop action. The default value is 0.0.

MEIMotionConfig

```
typedef struct  MEIMotionConfig {
    long        axisCount;
    long        axisNumber[MEIXmpMAX_COORD_AXES];
    double      blendLimit;
} MEIMotionConfig;
```

Description

axisCount	The current number of axes mapped to the Motion Supervisor on the controller.
axisNumber	This array specifies the axis numbers of the current Axis to Motion Supervisor mapping on the controller.
blendLimit	This value specifies the acceleration blending limit criteria. If the change direction is greater than 90 degrees (0 degrees = no change, 180 degrees = about face) the acceleration resulting from the blending can exceed the acceleration limit specified in the motion parameters (180 degrees = acceleration*2.0). The blendLimit allows the user to limit the sharpness of turns to be blended. If cosine (turn angle defined above) is greater than the blendLimit, the motion will be blended. A blend limit value of 0 exclude turns sharper than 90 degrees. 1.0 causes all moves to be blended. -1.0 allows no blending

See Also

MPIMotionDecelTime | mpiMotionModify

MPIMotionDecelTime

MPIMotionDecelTime

```
typedef struct MPIMotionDecelTime {
    float       stop;   /* seconds */
    float       eStop;  /* seconds */
} MPIMotionDecelTime;
```

Description

stop	This value defines the deceleration time (seconds) for a Stop action. The default value is .5 seconds.
eStop	This value defines the deceleration time (seconds) for an E-Stop action. The default value is .05 seconds.

See Also

MEIMotionFrame

MEIMotionFrame

```
typedef struct MEIMotionFrame {
    long                pointCount;
    MEIXmpFrame         *frame;
    MPIMotionPoint      point;
} MEIMotionFrame;
```

Description

pointCount	The value specifies the number of frames.
***frame**	This structure contains the frame data for each frame. See MEIXmpFrame for more information.
point	This structure specifies the points configuration. See MPIMotionPoint for more information.

See Also

MPIMotionPoint

MEIMotionFrameBufferStatus

MEIMotionFrameBufferStatus

```
typedef struct MEIMotionFrameBufferStatus {
    long        size;
    long        frameCount;
} MEIMotionFrameBufferStatus;
```

Description

size	This value specifies the size of the controller's frame buffer.
frameCount	This value specifies the number of frames in the controller's frame buffer.

See Also

MPIMotionMessage / MEIMotionMessage

MPIMotionMessage

```
typedef enum {
    MPIMotionMessageMOTION_INVALID,
    MPIMotionMessageAXIS_NOT_FOUND,
    MPIMotionMessageAXIS_COUNT,
    MPIMotionMessageTYPE_INVALID,
    MPIMotionMessageATTRIBUTE_INVALID,
    MPIMotionMessageNOT_READY,
    MPIMotionMessageIDLE,
    MPIMotionMessageMOVING,
    MPIMotionMessageSTOPPING,
    MPIMotionMessageSTOPPING_ERROR,
    MPIMotionMessageERROR,
    MPIMotionMessageAUTO_START,
    MPIMotionMessagePROFILE_ERROR,
    MPIMotionMessagePATH_ERROR,
    MPIMotionMessageFRAMES_LOW,
    MPIMotionMessageFRAMES_EMPTY,
} MPIMotionMessage;
```

Description

MPIMotionMessageMOTION_INVALID	This message code occurs when the motion supervisor number is not valid.
MPIMotionMessageAXIS_NOT_FOUND	This message code occurs when an axis that is being removed is not a member of the motion supervisor.
MPIMotionMessageAXIS_COUNT	This message code occurs when the axis count exceeds MEIXmpMAX_COORD_AXES.
MPIMotionMessageTYPE_INVALID	This message code occurs when the MPIMotionType is not valid.
MPIMotionMessageATTRIBUTE_INVALID	This message code occurs when the MPIMotionAttrMask is not valid.
MPIMotionMessageNOT_READY	This message code occurs when the controller is in the MPIStateINIT and is not ready to generate motion profiles.
MPIMotionMessageIDLE	This message code occurs when mpiMotionModify(…) or mpiMotionAction(…) is called when the motion supervisor is in the IDLE state.
MPIMotionMessageMOVING	This message code occurs when mpiMotionStart(…) or mpiMotionAction(…) is called when the motion supervisor is in the MOVING state.
MPIMotionMessageSTOPPING	This message code occurs when mpiMotionStart(…), mpiMotionModify(…) or mpiMotionAction(…) is called when the motion supervisor is in the STOPPING state. Motion cannot be started or modified at this time.
MPIMotionMessageSTOPPING_ERROR	This message code occurs when mpiMotionStart(…), mpiMotionModify(…) or mpiMotionAction(…) is called when the motion supervisor is in the STOPPING_ERROR state. Motion cannot be started or modified at this time.
MPIMotionMessageERROR	This message code occurs when mpiMotionStart(…), mpiMotionModify(…) or mpiMotionAction(…) is called when the motion supervisor is in the ERROR state. Motion cannot be started or modified at this time.

MPIMotionMessageAUTO_START	This message code occurs when mpiMotionModify(…) was called when the motion supervisor was in the IDLE state and was automatically converted into an mpiMotionStart(…).
MPIMotionMessagePROFILE_ERROR	This message code occurs when the velocity must be reversed, but mpiMotionModify(…) was called with the NO_REVERSAL attribute.
MPIMotionMessagePATH_ERROR	This message code occurs when the path motion time slice is set to less than one controller sample period.
MPIMotionMessageFRAMES_LOW	XMP internal point list buffer is low.
MPIMotionMessageFRAMES_EMPTY	XMP internal point list ran out of frames.

MEIMotionMessage

```
typedef enum {
    MEIMotionMessageRESERVED0,
    MEIMotionMessageRESERVED1,
    MEIMotionMessageRESERVED2,
    MEIMotionMessageNO_AXES_MAPPED,
} MEIMotionMessage;
```

Description

MEIMotionMessageRESERVED0	Reserved for specialized use.
MEIMotionMessageRESERVED1	Reserved for specialized use.
MEIMotionMessageRESERVED2	Reserved for specialized use.
MEIMotionMessageNO_AXES_MAPPED	Returned from methods that require at least one axes to be associated to motion supervisor object.

See Also

MPIMotionType | MPIMotionAttrMask | mpiMotionModify | mpiMotionAction
mpiMotionStart

MPIMotionParams / MEIMotionParams

MPIMotionParams

```
typedef struct MPIMotionParams {
    MPIMotionJog        jog;

    MPIMotionPT         pt;
    MPIMotionPVT        pvt;
    MPIMotionSPLINE     spline;
    MPIMotionBESSEL     bessel;
    MPIMotionBSPLINE    bspline;

    MPIMotionSCurve        sCurve;
    MPIMotionSCurve        sCurveJerk;
    MPIMotionTrapezoidal   trapezoidal;

    MPIMotionVelocity   velocity;
    MPIMotionVelocity   velocityJerk;

    MPIMotionAttributes attributes;

    void                *external;
} MPIMotionParams;
```

Description

pt	This structure contains the parameters for a PT motion type. Please see MPIMotionPT data type for more information.
pvt	This structure contains the parameters for a PVT motion type. Please see MPIMotionPVT data type for more information.
spline	This structure contains the parameters for a SPLINE motion type. Please see MPIMotionSPLINE data type for more information.
bessel	This structure contains the parameters for a BESSEL motion type. Please see MPIMotionBESSEL data type for more information.
bspline	This structure contains the parameters for a BSPLINE motion type. Please see MPIMotionBSPLINE data type for more information.
sCurve	This structure contains the parameters for a S_CURVE motion type. Please see MPIMotionSCurve data type for more information.
velocity	This structure contains the parameters for a VELOCITY motion type. Please see MPIMotionVelocity data type for more information.
velocityJerk	This structure contains the parameters for a VELOCITY_JERK motion type. Please see MPIMotionVelocity data type for more information.
attributes	This structure contains the parameters for motion attributes. Please see MPIMotionAttributes data type for more information.

***external**	This points to an external structure, containing controller specific parameters. Presently, this only supports MEIMotionAttributes. Please see MEIMotionAttributes data type for more information.

MEIMotionParams

```
typedef         struct MEIMotionParams {
    MEIMotionFrame          frame;
    MPIMotionAttributes     attributes;
    MEIMotionAttributes     attributesMEI;
} MEIMotionParams;
```

Description

frame	This structure contains the frame data and points configuration. See MEIMotionFrame for more information.
attributes	This structure contains the motion attributes data. See MPIMotionAttributes for more information.
attributesMEI	This structure contains the motion attributes data. See MEIMotionAttributes for more information.

See Also

MEIMotionFrame | MPIMotionAttributes | MEIMotionAttributes

MPIMotionPoint

MPIMotionPoint

```
typedef struct MPIMotionPoint {
    long        retain;         /* FALSE => flush points after use */
    long        final;          /* FALSE => more points to come */
    long        emptyCount;     /* # of remaining points to trigger empty event, -1
=> disable */
} MPIMotionPoint;
```

Description

retain	This value specifies whether or not the points should be stored in a buffer after execution. If retain=0, the points will not be stored after execution. If retain=1, the points will be stored in a buffer. This feature is useful for backing up on path.
final	This value specifies if more points will be loaded. If final=1, no more points will be loaded. If final=0, more points can be loaded using mpiMotionModify(…) with the MPIMotionAttrMaskAPPEND attribute mask.
emptyCount	This value specifies the minimum number of points in the buffer. If the number of points in the buffer is below emptyCount, an E_STOP action will occur. When emptyCount is (-1), the buffer low trigger is disabled.

See Also

mpiMotionModify | MPIMotionAttrMaskAPPEND | MPIMotionPT

MPIMotionPT

MPIMotionPT

```
typedef         struct MPIMotionPT {
    long      pointCount;
    double    *position;
    double    *time;
    MPIMotionPoint      point;
} MPIMotionPT;
```

Description

pointCount	This value specifies the number of points.
position	This array stores the positions for the motion profile. There is one position value per point, per axis. The length of the array must be equal to pointCount multiplied by the number of axes. The positions are interleaved in the array by the axis index.
time	This array stores the times for the motion profile. There is one time value per point. The time specifies the number of seconds between the specified position, and the next position (point). The length of the time array must be equal to pointCount.
point	This structure contains the point configuration. Please see MPIMotionPoint data type for more information.

See Also

MPIMotionPoint

MPIMotionPVT

MPIMotionPVT

```
typedef struct MPIMotionPVT {
    long                pointCount;
    double              *position;
    double              *velocity;
    double              *time;
    MPIMotionPoint      point;
} MPIMotionPVT;
```

Description

pointCount	This value specifies the number of points.
position	This array stores the positions for the motion profile. There is one position value per point, per axis. The length of the array must be equal to pointCount multiplied by the number of axes. The positions are interleaved in the array by the axis index.
velocity	This array stores the times for the motion profile. There is one time value per point. The time specifies the number of seconds between the specified position, and the next position (point). The length of the time array must be equal to pointCount.
time	This array stores the times for the motion profile. There is one time value per point. The time specifies the number of seconds between the specified position, and the next position (point). The length of the time array must be equal to pointCount.
point	This structure contains the point configuration. Please see MPIMotionPoint data type for more information.

See Also

MPIMotionPoint

MPIMotionSCurve

MPIMotionSCurve

```
typedef struct MPIMotionSCurve {
    MPITrajectory   *trajectory;
    double          *position;
} MPIMotionSCurve;
```

Description

***trajectory**	This structure specifies the parameters for the motion profile.
***position**	This array specifies the target positions for the axes. There is one position for each axis.

See Also

MPIMotionSPLINE

MPIMotionSPLINE

```
typedef         struct MPIMotionSPLINE {
    long                    pointCount;
    double                  *position;
    double                  *time;

    MPIMotionPoint          point;
} MPIMotionSPLINE;
```

Description

pointCount	This value specifies the number of points.
***position**	This array stores the positions for the motion profile. There is one position value per point, per axis. The length of the array must be equal to pointCount multiplied by the number of axes. The positions are interleaved in the array by the axis index.
***time**	This array stores the times for the motion profile. There is one time value per point. The time specifies the number of seconds between the specified position, and the next position (point). The length of the time array must be equal to pointCount.
point	This structure contains the point configuration. Please see MPIMotionPoint data type for more information.

See Also

MPIMotionPoint

MEIMotionTrace

MEIMotionTrace

```
typedef enum {

    MEIMotionTraceSTATUS,
    MEIMotionTracePARAMS,
} MEIMotionTrace;
```

Description

MEIMotionTraceSTATUS	This trace bit enables motion status tracing for mpiMotion calls.
MEIMotionTracePARAMS	This trace bit enables motion parameters tracing for mpiMotion calls.

See Also

MPIMotionTrapezoidal

MPIMotionTrapezoidal

```
typedef struct MPIMotionTrapezoidal {
    MPITrajectory   *trajectory;
    double          *position;
} MPIMotionTrapezoidal;
```

Description

***trajectory**	This structure specifies the parameters for the motion profile.
***position**	This array specifies the target positions for the axes. There is one position for each axis.

See Also

MPIMotionType / MEIMotionType

MPIMotionType

```
typedef enum {
    MPIMotionTypeINVALID,
    MPIMotionTypeJOG,

    MPIMotionTypePT,
    MPIMotionTypePVT,
    MPIMotionTypeSPLINE,
    MPIMotionTypeBESSEL,
    MPIMotionTypeBSPLINE,
    MPIMotionTypeBSPLINE2,

    MPIMotionTypeS_CURVE,
    MPIMotionTypeTRAPEZOIDAL,
    MPIMotionTypeS_CURVE_JERK,

    MPIMotionTypeVELOCITY,
    MPIMotionTypeVELOCITY_JERK,
    MPIMotionTypeMASK,
} MPIMotionType;
```

Description

MPIMotionType specifies the particular motion profile algorithm to be generated with mpiMotionStart(…) and mpiMotionModify(…).

MPIMotionTypePT	This type fits constant velocity profile segments through a list of position and time points. Please see MPIMotionPT for more information.
MPIMotionTypePVT	This type fits jerk profile segments through a list of position, velocity and time points. Please see MPIMotionPVT for more information.
MPIMotionTypeSPLINE	This type fits a Cubic spline through a specified list of position and time points. Please see MPIMotionSPLINE data type for more information.
MPIMotionTypeBESSEL	This type fits a Bessel spline through a specified list of position and time points. Please see MPIMotionBESSEL data type for more information.
MPIMotionTypeBSPLINE	This type fits a 3rd order B spline through a list of position and time points. Please see MPIMotionBSPLINE data type for more information.
MPIMotionTypeBSPLINE2	This type fits a 2nd order B spline through a list of position and time points. Please see MPIMotionBSPLINE data type for more information.
MPIMotionTypeS_CURVE	This type specifies point to point motion using a S-Curve velocity profile. The profile is specified by acceleration, velocity, deceleration, jerkPercent, and final position. Please see MPIMotionSCurve data type for more information.
MPIMotionTypeS_CURVE_JERK	This type specifies point to point motion using a S-Curve velocity profile. The profile is specified by acceleration, velocity, deceleration, accelerationJerk, decelerationJerk, and final position. Please see MPIMotionSCurve data type for more information.
MPIMotionTypeTRAPEZOIDAL	This type specifies simple point to point motion using a trapezoidal velocity profile. The profile trajectory is specified by acceleration, velocity, deceleration and final position. Please see MPIMotionTrapezoidal data type for more information.
MPIMotionTypeVELOCITY	This type specifies S-Curve acceleration to a constant velocity. The profile trajectory is specified by acceleration, velocity, and jerkPercent. Please see MPIMotionVelocity data type for more information.
MPIMotionTypeVELOCITY_JERK	This type specifies S-Curve acceleration to a constant velocity. The profile trajectory is specified by acceleration, velocity, accelerationJerk and decelerationJerk. Please see MPIMotionVelocity data type for more information.

MEIMotionType

```
typedef enum {
    MEIMotionTypeFRAME,
} MEIMotionType;
```

Description

MEIMotionTypeFRAME	This motion type is used to construct motion profiles at the frame level.

See Also

mpiMotionStart | mpiMotionModify | mpiMotionTYPE

MPIMotionVelocity

MPIMotionVelocity

```
typedef struct MPIMotionVelocity {
    MPITrajectory       *trajectory;
} MPIMotionVelocity;
```

Description

***trajectory**	This structure specifies the parameters for the motion profile, except deceleration is ignored. Please see MPITrajectory data type for more information.

See Also

MPITrajectory

mpiMotionATTR

Declaration

```
#define  mpiMotionATTR(type,attr)
((type) |= mpiMotionAttrMaskBIT(attr))
```

Required Header

stdmpi.h

Description

MotionATTR turns on the specified motion attribute mask bits in the motion type.

See Also

MPIMotionAttr | MPIMotionAttrMask

mpiMotionAttrMaskBIT

Declaration `#define mpiMotionAttrMaskBIT(attr) (0x1 << (attr))`

Required Header stdmpi.h

Description **MotionAttrMaskBIT** converts the motion attribute into the motion attribute mask.

See Also MPIMotionAttr | MPIMotionAttrMask

mpiMotionTYPE

Declaration

```
#define  mpiMotionTYPE(type)    ((type) & MPIMotionTypeMASK)
```

Required Header

stdmpi.h

Description

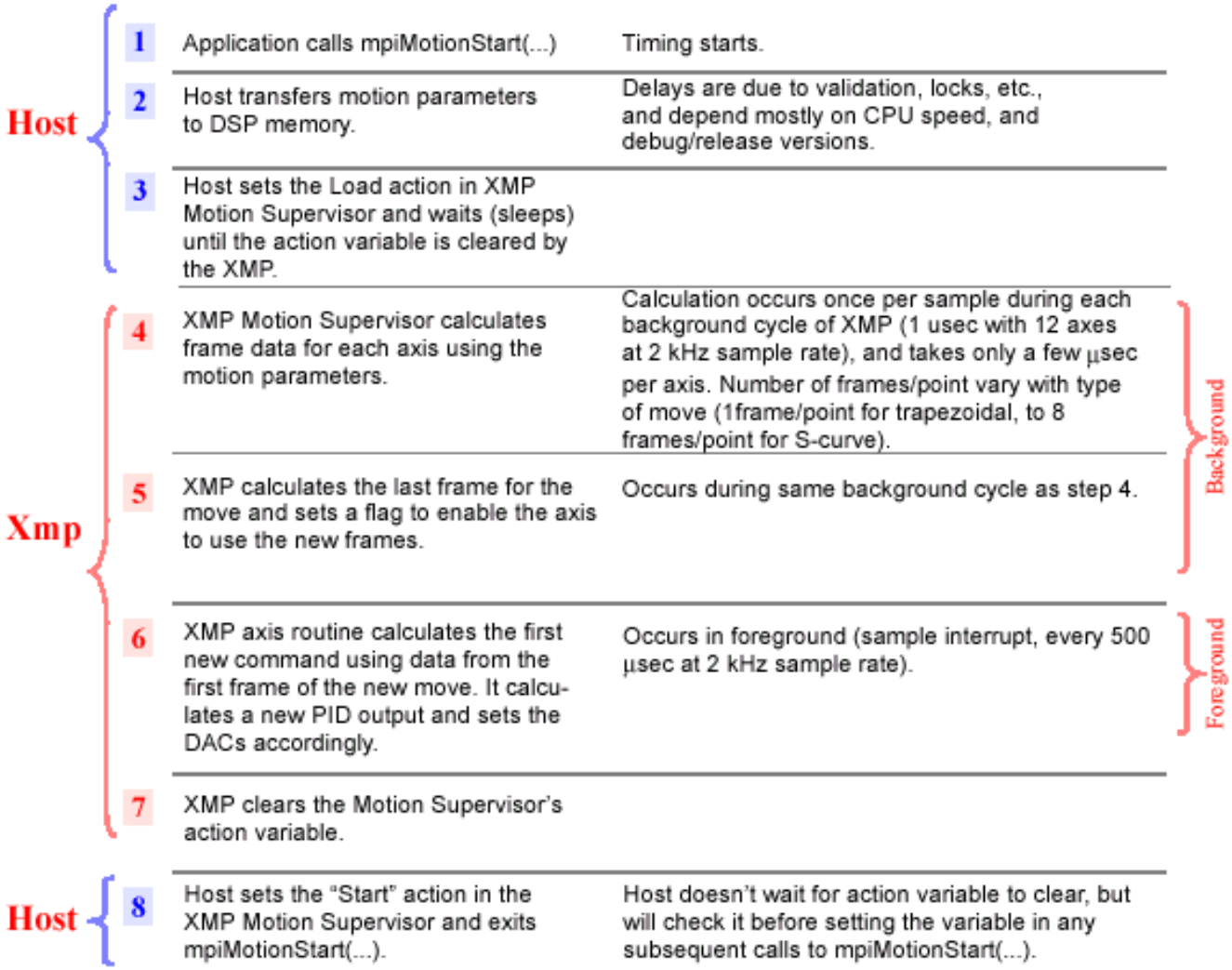
MotionType masks off all other bits in ***type***, leaving the motion type.

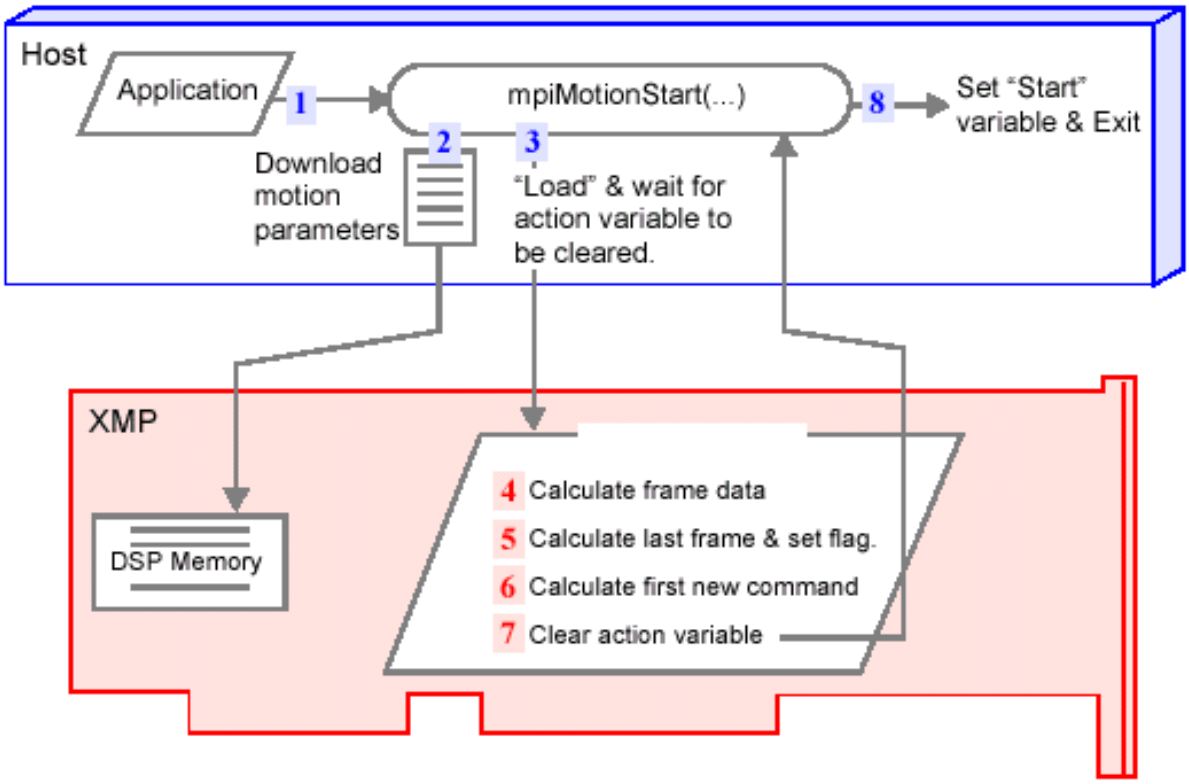
See Also

MPIMotionType

Diagram of mpiMotionStart

The diagram below explains how mpiMotionStart(...) executes.





Return to mpiMotionStart

Copyright © 2002
Motion Engineering

Motion Error Codes and Messages

Code	Message	Tips / Fixes
0x0d01 (3329)	Motion: motion invalid	The Motion parameters are incorrect. Recheck them.
0x0d02 (3330)	Motion: axis not found	The Axis number is invalid.
0x0d03 (3331)	Motion: axis count invalid	You may be commanding moves on Axes that don't exist, or on zero Axes.
0x0d04 (3332)	Motion: type invalid	You may have called a motion type that doesn't exist.
0x0d05 (3333)	Motion: attribute invalid	The value for a motion attribute is wrong.
0x0d06 (3334)	Motion: not ready	The motion controller is still executing the power-up sequence.
0x0d07 (3335)	Motion: MPIStateIDLE	No Motion was in progress when mpiMotionModify(...) was called. In order for mpiMotionModify(...) to work, there must be a motion in progress.
0x0d08 (3336)	Motion: MPIStateMOVING	You may have called mpiMotionStart(...) when a Motion was in progress.
0x0d09 (3337)	Motion: MPIStateSTOPPING	Indicates that a Stop event is in progress.
0x0d0a (3338)	Motion: MPIStateSTOPPING_ERROR	Indicates that an E-Stop event is in progress.
0x0d0b (3339)	Motion: MPIStateERROR	Indicates that an E-Stop or Abort event has occurred. Use mpiMotionAction(MPIActionRESET) to recover.
0x0d0c (3340)	Motion: auto-start	mpiMotionModify(...) was called with the MPIMotionAttrMaskAUTO_START attribute when the motion state was MPIStateIDLE. In this case, the motion profile is automatically started.
0x0d0d (3341)	Motion: profile error	The XMP couldn't generate the next set of points. Check the motion parameters and attributes.

Return to Motion Objects page

Copyright © 2002
Motion Engineering

Motion Attributes

Introduction | MPI Motion Attributes | MEI Motion Attributes

Introduction

This section details the use of various Motion Attributes, which are listed individually below. To use Motion Attributes, OR the attribute mask with the MPIMotionType value.

TOP

MPI Motion Attributes

MPIMotionAttrAPPEND

The MPI has been extended to support mpiMotionStart(...) with the MPIMotionAttrMaskAPPEND attribute. This makes it possible to buffer several point-to-point motion profiles in the controller. Each successful call to mpiMotionStart(...) with the APPEND attribute will generate an MPIEventTypeMOTION_DONE from the controller when the motion is complete.

The MPIMotionAttrMaskID attribute is supported with MPIMotionAttrMaskAPPEND when calling mpiMotionStart(...). The XMP-Series controller firmware has been modified to buffer the ID values inside the axis' frames. Therefore, applications using the motion and axis event user data must be changed. Since the ID is stored in the controller's axis frames, the mpiMotionEventNotifySet(...) and mpiAxisEventNotifySet(...) must explicitly configure an appropriate axis memory location for the firmware to retrieve the ID. The sample programs motionID1.c and motionID2.c demonstrate this feature.

The MEIMotionAttrHOLD attribute is supported with MPIMotionAttrMaskAPPEND when calling mpiMotionStart(...).

Move Types: **PT, PVT, Bessel, Bspline, Bspline2, S-Curve, Trapezoidal, Velocity, Frame**

MPIMotionAttrAUTO_START

The MPI has been extended to support mpiMotionStart(...) with the MPIMotionAttrMaskAUTO_START attribute. When AUTO_START is enabled, calls to mpiMotionModify(…) will automatically start a new motion if the previous motion is done. If AUTO_START is not enabled, and mpiMotionModify(...) is commanded after the initial motion has completed, the method will return an MPIMotionStateIDLE error.

Move Types: **S-Curve, Trapezoidal, Velocity**

MPIMotionAttrDELAY

The MPI has been extended to support mpiMotionStart(...) with the MPIMotionAttrMaskDELAY attribute. This motion attribute will delay the move for a given number of seconds. MPIMotionParams.attributes.delay is a pointer to an array of doubles that assign delay times for each Axis.

Move Types: **S-Curve, Trapezoidal, Velocity**

MPIMotionAttrELEMENT_ID

The MPI has been extended to support mpiMotionStart(...) with the MPIMotionAttrMaskELEMENT_ID attribute. Similar to the MPIMotionAttrMaskID, the ELEMENT_ID allows the application to set an identification value for each element of a path motion. The ID values are long values configured in the MPIMotionParams.attributes.elementId array. Each element in the array will be the ID value for that sequential portion of the motion.

In order to retrieve the ElementID, a pointer to the element ID is placed into the MEIEventNotifyData structure. This will cause the XMP to send the ElementID up to the MEIEventStatusInfo structure when an event occurs that causes an interrupt.

Move Types: PT, PVT, Bessel, Bspline, Bspline2, S-Curve, Trapezoidal, Velocity

MPIMotionAttrMaskID

The MPI has been extended to support mpiMotionStart(...) with the MPIMotionAttrMaskID attribute. The ID attributes allows the application to assign an identification number to each motion. This number can be returned in the MPIEventStatus structure so the application will know which move has ended. This is particularly useful when multiple moves are buffered and the application needs to know which move is executing or has returned an event.

The MPIMotionParams.attributes.id value is a long that identifies the Motion. This ID value is passed to each Axis associated with the MS. In order to retrieve the MoveID, a pointer to the move ID is placed in the MEIEventNotifyData structure. This will cause the XMP to send the MoveID up to the MEIEventStatusInfo structure when an event occurs that causes an interrupt.

Move Types: **PT, PVT, Spline, Bessel, Bspline, Bspline2, S-Curve, Trapezoidal, Velocity, Frame**

MPIMotionAttrRELATIVE

The MPI has been extended to support mpiMotionStart(...) with the MPIMotionAttrMaskRELATIVE attribute. When this mask is ANDed into the attribute mask, all position values will be used as relative motion distances instead of final absolute positions. For example, without the RELATIVE motion attribute, a move beginning at position 1000 with a position parameter value of 2000 will move to position 2000. If the RELATIVE attribute is turned on, the final move position will be 3000, a relative distance of 2000 counts from the starting value of 1000.

Move Types: **PT, PVT, Spline, Bessel, Bspline, Bspline2, S-Curve, Trapezoidal, Velocity**

MPIMotionAttrSYNC_END

The MPI has been extended to support mpiMotionStart(...) with the MPIMotionAttrMaskSYNC_END attribute. SYNC_END is used for motions that include more than one axis. The SYNC_END attribute will generate trajectories for all axes appended to an MS that will end simultaneously. For all but the longest motion profile, wait frames will be added to the beginning of the moves. This will ensure that all axes end simultaneously.

Move Types: **S-Curve, Trapezoidal**

MPIMotionAttrSYNC_START

The MPI has been extended to support mpiMotionStart(...) with the MPIMotionAttrMaskSYNC_START attribute. SYNC_START is used for Motions that include more than one Axis. All Axes will begin simultaneously. For those axes that finish first, a delay frame will be added to the end of the move.

Move Types: **S-Curve, Trapezoidal**

TOP

MEI Motion Attributes

MEIMotionAttrEVENT

The MPI has been extended to support mpiMotionStart(...) with the MEIMotionAttrMaskEVENT attribute.

MEIMotionAttrFINAL_VEL

The MPI has been extended to support mpiMotionStart(...) with the MEIMotionAttrMaskFINAL_VEL attribute.

MEIMotionAttrNO_REVERSAL

The MPI has been extended to support mpiMotionStart(...) with the MEIMotionAttrMaskNO_REVERSAL attribute.

MEIMotionAttrHOLD

The MPI has been extended to support mpiMotionStart(...) with the MEIMotionAttrMaskHOLD attribute. The HOLD attribute prevents execution of a Motion. The HOLD attribute is applied at the beginning of the motion (one HOLD frame) before the execution of the point list. This prevents execution of the Motion until the HOLD frame is disabled.

The HOLD attribute is used to synchronize the start of motion with a host function call, XMP internal variable, or Motor Input state change. More than one Motion Supervisor may be synchronized. The type field of the MEIMotionAttrHold{} structure determines whether the synchronization comes from a host call (meiControlGateSet(), see below), internal variable (Axis Status, Position, etc.) or a Motor Input signal (transceiver, home input, user input, etc.).

Gated Moves: If the value of type is MEIMotionAttrHoldTypeGATE, the motion will be held until a call to mpiControlGateSet() is made with the closed parameter set to FALSE. When a MEIMotionAttrHoldTypeGATE is used, the source.gate field of MEIMotionAttrHold{} must be set to the gate number (0-31). The same gate number must be used for the gate parameter of meiControlGateSet().

Input Hold Moves: If the value of type is MEIMotionAttrHoldTypeINPUT, the motion will be held until then value of the internal Xmp variable specified (pointed to) by source.input bitwise anded with source.mask matches source.pattern.

Motor Input Hold Moves: If the value of type is MEIMotionAttrHoldTypeMOTOR, the motion will be held until the value of the internal dedicated input word (Motor[n].IO. DedicatedIN.IO) for the motor specified by source.motorNumber bitwise anded with source.mask matches source.pattern.

The timeout field of MEIMotionAttrHold{} will cause the motion to start after the specified timeout period (in seconds), even if the other hold criteria have not been satisfied. A value of zero for timeout causes the timeout feature to be disabled and forces the motion to wait for the hold criteria (gate open, or pattern match).

The MPI expects an array of hold attributes specifying separate attributes form each axis of a motion supervisor. All axes holding with the same hold attributes (same gate, same input, mask, and pattern) will start motion in the same sample even if the moves are specified using different motion supervisors.

MEIMotionAttrOUTPUT

The MPI has been extended to support mpiMotionStart(...) with the MPIMotionAttrMaskAPPEND attribute. This motion attribute allows a Motion to change the state of a register in the controller memory. This configures a frame to toggle an output bit as a move begins.

TOP

Return to Motion Objects page

Copyright © 2002
Motion Engineering

Returns for mpiMotionStart and mpiMotionModify

Both mpiMotionStart and mpiMotionModify use motion attributes, which are summarized in the table below.

Return Values - ***for both mpiMotionStart and MotionModify***	
MPIMotionMessageERROR	if *MotionStart* is called when the Motion Supervisor is in the error state.
MPIMotionMessageAXIS_COUNT	if the motion object has no axes.
MPIMotionMessagePROFILE_ERROR	if the Motion profile specified by the parameters cannot be calculated correctly.
MPIMotionMessageATTRIBUTE_INVALID	if the specified motion attribute mask is not valid with the motion type.
MPIMessagePARAM_INVALID	if the specified motion parameters are not valid with the Motion type.
MPIMessageARG_INVALID	if any of the *MotionStart* arguments are not valid.
MPIMessageUNSUPPORTED	if the motion type is not supported by the controller.

Return to mpiMotionStart or mpiMotionModify

Copyright © 2002
Motion Engineering

Motor Objects

Introduction

A **Motor** object manages a single motor on a controller. It represents the physical connections between the motor, drive, and associated I/O. The Motor object contains encoder data, limit switch, home sensor, amp fault and amp enable states, DAC outputs, and other status information.

For simple systems, there is a one-to-one relationship between the Axis, Filter and Motor objects.

For information about using absolute encoders with the MPI click here.

Methods

Create, Delete, Validate Methods

mpiMotor**Create**	Create Motor object
mpiMotor**Delete**	Delete Motor object
mpiMotor**Validate**	Validate Motor object

Configuration and Information Methods

mpiMotor**AmpEnableGet**	Get state of amp enable output
mpiMotor**AmpEnableSet**	Set state of amp enable output
mpiMotor**AxisMapGet**	Get object map of axes
meiMotor**CommutationModeGet**	Gets the commutation mode of a motor.
meiMotor**CommutationModeSet**	Sets the commutation mode of a motor.
mpiMotor**ConfigGet**	Get motor configuration
mpiMotor**ConfigSet**	Set motor configuration
meiMotor**ConfigStepper**	Configures a motor for stepper mode.
meiMotor**DacConfigGet**	Get a Motor's (motor) Dac configuration
meiMotor**DacConfigSet**	Set a Motor's (motor) Dac configuration
mpiMotor**FeedbackGet**	Get feedback position
mpiMotor**FlashConfigGet**	Get flash config of motor
mpiMotor**FlashConfigSet**	Set flash config of motor
mpiMotor**IoGet**	Get dedicated I/O bits
mpiMotor**IoSet**	Set dedicated I/O bits
mpiMotor**Status** / meiMotor**Status**	Get motor status
mpiMotor**Type**	Get Motor type

Event Methods

mpiMotor**EventConfigGet**	Get motor’s event configuration
mpiMotor**EventConfigSet**	Set motor’s event configuration
mpiMotor**EventNotifyGet**	Get motor’s event mask for host notification.
mpiMotor**EventNotifySet**	Set motor’s event mask for host notification.
mpiMotor**EventReset**	Reset events specified in event mask
mpiMotor**EventWait**	Set the contents of the structure pointed to by status.

Memory Methods

mpiMotor**Memory**	Get address of motor memory
mpiMotor**MemoryGet**	Copy motor memory to application memory
mpiMotor**MemorySet**	Copy application memory to motor memory

Action Methods

meiMotor**EncoderInit**	Initializes an absolute encoder.
meiMotor**EncoderReset**	Resets an absolute encoder.

Relational Methods

mpiMotor**Control**	Get handle to associated Control object
mpiMotor**FilterMapGet**	Get object map of associated Filters
mpiMotor**FilterMapSet**	Set the Filters using object map
mpiMotor**Number**	Get index number of motor (in Control list)

Other Methods

meiMotor**CompareListGet**	
mpiMotor**DedicatedInAddrGet**	Get the address of the Dedicated IO for the Motor
meiMotor**DedicatedIoAddrDecode**	
meiMotor**DedicatedOutAddrGet**	
meiMotor**EncoderRatio**	Get encoder ratio from the XMP.
meiMotor**RelatedStepMotorGet**	

Data Types

MPIMotor**Brake**
MPIMotor**BrakeMode**
MPIMotor**Config** / MEIMotor**Config**
MEIMotor**DacConfig**
MEIMotor**DacChannelConfig**
MEIMotor**DacChannelStatus**
MEIMotor**DacStatus**
MPIMotor**EncoderFault**
MPIMotor**EncoderFaultMask**
MPIMotor**EventConfig** / MEIMotor**EventConfig**
MPIMotor**EventTrigger**
MEIMotor**Input**
MPIMotor**Io**
MPIMotor**Message** / MEIMotor**Message**
MEIMotor**Output**
MEIMotor**ResourceNumber**
MEIMotor**Status**
MEIMotor**Stepper**
MEIMotor**Transceiver**

MEIMotor**TransceiverConfig**
MEIMotor**TransceiverExtendedId**
MEIMotor**TransceiverExtendedMask**
MEIMotor**TransceiverId**
MEIMotor**TransceiverMask**
MPIMotor**Type**
MEIMotor**TypeInfo**

Macros

mpiMotor**EncoderFaultMaskBIT**

Copyright © 2002
Motion Engineering

Working with Absolute Encoders

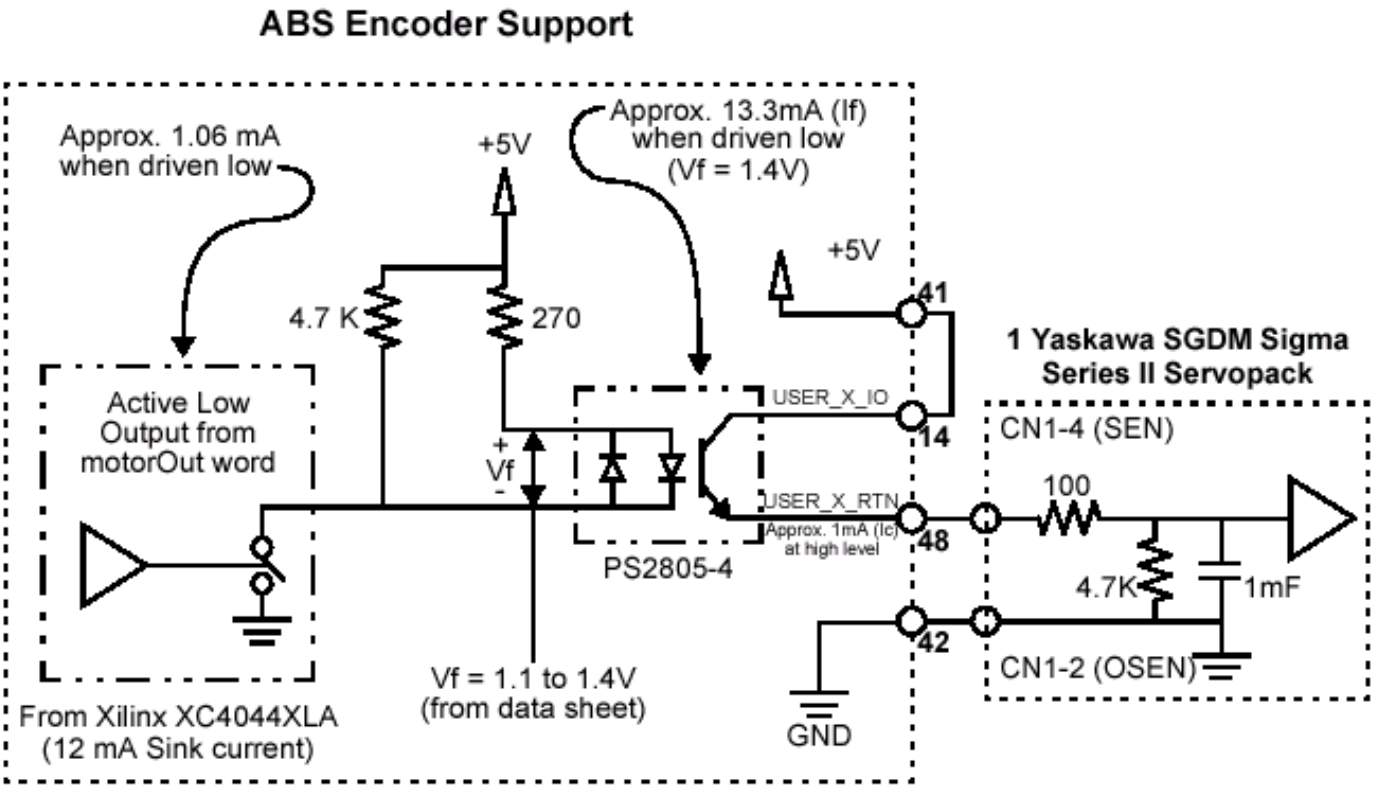
Currently, only absolute motor encoders of Yaskawa SGDM Sigma Series II Servopacks are supported by the MPI. Custom firmware is needed in order to use these absolute encoders. Contact MEI for more information.

The encoders are automatically interrogated at power-up or after a controller reset. Axis Origin and Command Positions are set to correctly reflect the absolute position of the motor with no position error initially. The absolute position is the position within a single revolution.

The encoder interrogation is controlled by a SEN signal (to the drive) which must be connected to a configured XMP Transceiver or User Out signal. There are no restrictions as to which XMP signal is used except that current drive limitations may limit the number of drives connected to the same XMP signal.

The MEIMotorEncoder{} structure has been added to the MEIMotorConfig{} object:

```
typedef struct MEIMotorEncoder {
    MEIXmpEncoderType     type;
    long                  countsPerRev;
} MEIMotorEncoder;

typedef struct MEIMotorConfig {
    MEIMotorEncoder    Encoder[MEIXmpMotorEncoders];
    MEIXmpIO           StatusOutput[MEIXmpMotorStatusOutputs];

    MEIMotorTransceiver     Transceiver[MEIXmpMotorTransceivers];
    long                    UserOutInvert;     /* Opto Polarity */
    MEIMotorStepper         Stepper;
    long                    EncoderTermination;

    /* Commutation is read-only from field Theta to end*/
    MEIXmpCommutationBlock     Commutation;

    MEIXmpLimitData     Limit[MEIXmpLimitLAST];

    MEIMotorFilterInput     FilterInput[MEIXmpMotorFilterInputs];
} MEIMotorConfig;
```



Return to Motor Objects page

Copyright © 2002
Motion Engineering

Special Note: *Using mpiMotorConfigSet with Absolute Encoders*

All absolute encoder configuration through the MPI is made using mpiMotorConfigSet() calls. The below sample code demonstrates the correct way to configure the XMP for Yaskawa absolute encoders.

When using a motor's User Output:

```
returnValue =
        mpiMotorFlashConfigGet(motor,
                               NULL,
                               &motorConfig,
                               &motorConfigMEI);

        motorConfig.encoderPhase = TRUE; /* Reverse */

        /* Config User Output for SEN line */
        motorConfigMEI.UserOutInvert = TRUE;

        motorConfigMEI.Encoder[0].type = MEIXmpEncoderTypeABS_0; /* Yaskawa encoder
*/
        motorConfigMEI.Encoder[0].countsPerRev = 65536;          /* 65536 for 16-bit
                                                        encoders,131072 for 17-bit */

        returnValue =
                mpiMotorFlashConfigSet(motor,
                                       NULL,
                                       &motorConfig,
                                       &motorConfigMEI);
        msgCHECK(returnValue);

        returnValue =
                mpiMotorConfigSet(motor,
                                  &motorConfig,
                                  &motorConfigMEI);
        msgCHECK(returnValue);
```

When using a motor's Transceiver Output:

```
returnValue =
                mpiMotorFlashConfigGet(motor,
                                       NULL,
                                       &motorConfig,
                                       &motorConfigMEI);

        motorConfig.encoderPhase = TRUE; /* Reverse */

        /* Config transceiver for SEN line */
        motorConfigMEI.Transceiver[0].Config = MEIMotorTransceiverConfigOUTPUT;
        motorConfigMEI.Transceiver[0].Invert = TRUE;

        motorConfigMEI.Encoder[0].type = MEIXmpEncoderTypeABS_0; /* Yaskawa encoder
*/
        motorConfigMEI.Encoder[0].countsPerRev = 65536;          /* 65536 for 16-bit
                                                        encoders,  131072 for 17-bit
*/
returnValue =
                mpiMotorFlashConfigSet(motor,
```

```
                                        NULL,
                                        &motorConfig,
                                        &motorConfigMEI);
    msgCHECK(returnValue);

    returnValue =
            mpiMotorConfigSet(motor,
                                           &motorConfig,
                                           &motorConfigMEI);
```

In the above sample code, the steps for configuration are:

1. Choose a transceiver, or User Opto, to be used for the encoders SEN line. The only restriction is that this transceiver must be on the same controller as the absolute encoder (not necessarily the same Motion Block).
2. Get the current motor configuration from flash memory.
3. Configure the encoder phase for the absolute encoder to Reverse.
4. For a User Opto, configure UserOutInvert to be TRUE. When using a transceiver, configure for Output, and Inverted.
5. Configure the encoder type and counts per revolution.
6. Save the current motor configuration from flash memory.

Once configured, the initialization of all axes associated with the motors having absolute encoders is automatic at power up or reset. The SEN line is toggled and the origin and command position are calculated and set from the absolute data sent by the drive.

IMPORTANT NOTE:

The drive must be powered but should not be enabled.

Determining the countPerRev Parameter

The **magnitude** countsPerRev parameter is detemined by the number of encoder counts (after quadrature) for one revolution of the motor. The **sign** of the countsPerRev is derimined by the direction for positive rotation for the motor. For Yaskawa drives this is determined by the drive parameter P000.0. P000.0 = 0 (“Standard Rotation”, factory default setting) will cause the motor to move in a counter-clockwise (CCW) direction for positive increases in encoder counts. For Standard Rotation (Pn000.0 = 0) the countPerRev parameter should be positive.

P000.0 = 1 (“Reverse Rotation”) will cause the motor to move in a clockwise (CCW) direction for positive increases in encoder counts. For Reverse Rotation the countsPerRev parameter should be negative.

For example the following code would be used for a drive congfigured for Standard Rotation where the number of counts for one revolution of the motor shaft is 8,192:

motorConfigMEI.Encoder[0].countsPerRev = 8192;

If the same drive were configured for Reverse Rotation the code would be:

motorConfigMEI.Encoder[0].countsPerRev = -8192;

For both Standard and Reverse Rotation the encoderPhase parameter should be TRUE (encoder reversed).

Return to mpiMotorConfigSet

Copyright © 2002
Motion Engineering

Special Note: *MPIMotorEventConfig and Motor Limit Configuration*

Two fields, directionFlag and duration, are built into the MPIMotorEventConfig{} structure. From motor.h:

```
typedef enum {
    MPIMotorEncoderFaultMaskNONE     = 0x0,

    MPIMotorEncoderFaultMaskBW_DET   =
mpiMotorEncoderFaultMaskBIT(MPIMotorEncoderFaultBW_DET),
    MPIMotorEncoderFaultMaskILL_DET  =
mpiMotorEncoderFaultMaskBIT(MPIMotorEncoderFaultILL_DET),
    MPIMotorEncoderFaultMaskABS_ERR  =
mpiMotorEncoderFaultMaskBIT(MPIMotorEncoderFaultABS_ERR),

    MPIMotorEncoderFaultMaskALL      =
(mpiMotorEncoderFaultMaskBIT(MPIMotorEncoderFaultLAST) - 1)
} MPIMotorEncoderFaultMask;

typedef union {
    long        polarity;       /* 0 => active low, else active high */
    long        position;       /* MPIEventTypeLIMIT_SW_[POS|NEG] */
    float       error;          /* MPIEventTypeLIMIT_ERROR */
    long        mask;           /* MPIEventTypeENCODER_FAULT */
} MPIMotorEventTrigger;

typedef struct MPIMotorEventConfig {
    MPIAction               action;
    MPIMotorEventTrigger    trigger;
    long                    direction;
    float                   duration;     /* seconds */
} MPIMotorEventConfig;
```

The directionFlag field is used to configure MPIEventTypeLIMIT_HW_NEG, MPIEventTypeLIMIT_HW_POS, MPIEventTypeLIMIT_SW_NEG, and MPIEventTypeLIMIT_SW_POS. A value of TRUE for this field will force the command direction for motion to be used by the Xmp controller to qualify these limit events. A value of FALSE for this field will cause the limit event to depend on the state of the limit. If the limit has been exceeded (actual position > software positive limit, actual position < software negative limit, positive or negative hardware overtravel is TRUE) the limit event will be based direction of commanded motion, in exactly the same way as for directionFlag = TRUE. If the limit has not been exceeded the limit event and status will be based solely on the limit input (hardware limits) or actual position (software limits), ignoring the direction of commanded motion. The default status of durationFlag is FALSE (ignore direction). The directionFlag is ignored (returned FALSE from “get” methods) for case MPIEventTypeAMP_FAULT, MPIEventTypeHOME, and MPIEventTypeLIMIT_ERROR.

The duration field may used in the configuration of all MPI Motor Events. A positive value for this field will require the limit condition to exist for duration seconds before an event will occur. This field is useful in overriding noisy limit inputs. For example, an overtravel limit with infrequent short (< 1msec) noise spikes on the limit input will ignore the noise of the limit is configured with a duration of 0.05. A spike whose duration was at least 0.05 seconds (50 milliseconds) would be required before an overtravel event would occur. The

default value for duration is 0.0.

Return to MPIMotorEventConfig

Copyright © 2002
Motion Engineering

mpiMotorCreate

Declaration

```
const MPIMotor mpiMotorCreate(MPIControl control,
                              long       number)
```

Required Header

stdmpi.h

Description

MotorCreate creates a Motor object associated with the motor identified by ***number***, and located on the motion controller (***control***). MotorCreate is the equivalent of a C++ constructor.

Return Values

handle	to a Motor object
MPIHandleVOID	if the Motor object could not be created

See Also

mpiMotorDelete | mpiMotorValidate

mpiMotorDelete

Declaration `long mpiMotorDelete(MPIMotor motor)`

Required Header stdmpi.h

Description **MotorDelete** deletes a Motor object and invalidates its handle (***motor***). *MotorDelete* is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *MotorDelete* successfully deletes a Motor object and invalidates its handle

See Also mpiMotorCreate | mpiMotorValidate

mpiMotorValidate

Declaration `long mpiMotorValidate(MPIMotor motor)`

Required Header stdmpi.h

Description **MotorValidate** validates a Motor object and its handle (***motor***).

Return Values	
MPIMessageOK	if Motor is a handle to a valid object.

See Also mpiMotorCreate | mpiMotorDelete

mpiMotorAmpEnableGet

Declaration

```
long mpiMotorAmpEnableGet(MPIMotor  motor,
                          long      *ampEnable)
```

Required Header

stdmpi.h

Description

MotorAmpEnableGet gets the state of the amp enable output for a Motor (***motor***) and writes it in the location pointed to by ***ampEnable***. Note that the actual state of amp enable output also depends upon the actual wiring and the polarity chosen in the instance of the MPIMotorConfig structure.

If "ampEnable" is	*Then*
FALSE (**0**)	the amp is disabled
TRUE (**1**)	the amp is enabled

Return Values

MPIMessageOK	if *MotorAmpEnableGet* successfully writes the Motor's amp enable output state to the location

See Also

MPIMotorConfig | mpiMotorAmpEnableSet

mpiMotorAmpEnableSet

Declaration

```
long mpiMotorAmpEnableSet(MPIMotor  motor,
                          long      ampEnable)
```

Required Header

stdmpi.h

Description

MotorAmpEnableSet sets the state of the amp enable output for a Motor (***motor***) to ***ampEnable***. Note that the actual state of amp enable output also depends upon the actual wiring and the polarity chosen in the instance of the MPIMotorConfig structure.

If "ampEnable" is	*Then*
FALSE (**0**)	the amp will be disabled
TRUE (**1**)	the amp will be enabled

Return Values

MPIMessageOK	if *MotorAmpEnableset* successfully sets the Motor's amp enable output state to ampEnable

See Also

MPIMotorConfig | mpiMotorAmpEnableGet

mpiMotorAxisMapGet

Declaration

```
long mpiMotorAxisMapGet(MPIMotor      motor,
                        MPIObjectMap  *map)
```

Required Header

stdmpi.h

Description

MotorAxisMapGet gets the object map of the Axes associated with a Motor (***motor***) and writes it into the structure pointed to by ***map***.

Return Values

MPIMessageOK	if *MotorAxisMapGet* successfully writes the Motor's object map of Axes to the structure

See Also

meiMotorCommutationModeGet

Declaration

```
long meiMotorCommutationModeGet(MPIMotor       motor,
                                MEIXmpCommMode *mode)
```

Required Header

stdmei.h

Description

MotorCommutationModeGet gets the commutation mode of a Motor (***motor***) and writes it to the location pointed to by ***mode***.

Return Values

MPIMessageOK	if *MotorCommutationModeGet* successfully gets the commutation mode of a Motor and writes it to the location

See Also

meiMotorCommutationModeSet

meiMotorCommutationModeSet

Declaration

```
long meiMotorCommutationModeSet(MPIMotor       motor,
                                MEIXmpCommMode mode)
```

Required Header stdmei.h

Description **MotorCommutationModeSet** sets the commutation mode of a Motor (***motor***) to ***mode***.

Return Values

MPIMessageOK	if *MotorCommutationModeSet* successfully sets the commutation mode of a Motor to ***mode***

See Also meiMotorCommutationModeGet

mpiMotorConfigGet

Declaration

```
long mpiMotorConfigGet(MPIMotor       motor,
                       MPIMotorConfig *config,
                       void           *external)
```

Required Header

stdmpi.h

Description

MotorConfigGet gets a Motor's (***motor***) configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type MEIMotorConfig{} or is NULL.

Return Values

MPIMessageOK	if *MotorConfigGet* successfully writes the Motor's configuration to the structure(s)

See Also

MEIMotorConfig | mpiMotorConfigSet

mpiMotorConfigSet

Declaration

```
long mpiMotorConfigSet(MPIMotor       motor,
                       MPIMotorConfig *config,
                       void           *external)
```

Required Header

stdmpi.h

Description

MotorConfigSet sets a Motor's (***motor***) configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is *in addition* to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type **MEIMotorConfig{}** or is NULL.

Return Values

MPIMessageOK	if *MotorConfigSet* successfully sets the Motor's configuration using data from the structure(s)

See Also

mpiMotorConfigGet
Special Note: Using mpiMotorConfigSet with Absolute Encoders

meiMotorConfigStepper

Declaration

```
long meiMotorConfigStepper(MPIMotor       motor,
                           MEIMotorConfig *config,
                           long           stepperNumber)
```

Required Header

stdmei.h

Description

MotorConfigStepper modifies the motor configuration structure pointed to by config, to use a step engine (stepperNumber) from another motor. By default, each motor uses its own step engine. Do NOT use more than one motor per step engine. Use the methods mpiMotorConfigGet/Set(…) to read/write the motor configuration from/to the controller.

motor	a handle to the Motion object
***config**	a pointer to the motion frame buffer status structure returned by the method
stepperNumber	index to a step engine

Return Values

MPIMessageOK	if *MotorConfigStepper* successfully modifies the motor configuration structure pointed to by ***config***

See Also

mpiMotorConfigGet | mpiMotorConfigSet

meiMotorDacConfigGet

Declaration

```
long meiMotorDacConfigGet(MPIMotor          motor,
                          MEIMotorDacConfig *dacConfig,
                          MEIFlash          flash);
```

Required Header stdmei.h

Description **MotorDacConfigGet** gets a Motor's (***motor***) Dac configuration and writes it to the structure pointed to by ***dacConfig***. The dac configuration located in ***flash*** is retrieved if the flash structure is not NULL.

Return Values

MPIMessageOK	if *MotorDacConfigGet* successfully writes the Motor's Dac configuration to the config structure

See Also meiMotorDacConfigSet | MEIMotorDacConfig

meiMotorDacConfigSet

Declaration

```
long meiMotorDacConfigSet(MPIMotor          motor,
                          MEIMotorDacConfig *dacConfig,
                          MEIFlash          flash);
```

Required Header stdmei.h

Description **MotorDacConfigSet** configures a Motor's (***motor***) Dac using data from the structure pointed to by ***dacConfig***. The dac configuration located in ***flash*** is set if the flash structure is not NULL.

Return Values

MPIMessageOK	if *MotorDacConfigSet* successfully writes the Motor's Dac configuration to the controller

See Also meiMotorDacConfigGet | MEIMotorDacConfig

mpiMotorFeedbackGet

Declaration

```
long mpiMotorFeedbackGet(MPIMotor motor,
                         double   *feedback)
```

Required Header

stdmpi.h

Description

MotorFeedbackGet gets the feedback position of a Motor (***motor***) and writes it into the location pointed to by ***feedback***.

Return Values

MPIMessageOK	if *MotorFeedbackGet* successfully writes the feedback position into the location

See Also

mpiMotorFeedbackConfigSet

mpiMotorFlashConfigGet

Declaration

```
long mpiMotorFlashConfigGet(MPIMotor          motor,
                            void              *flash,
                            MPIMotorConfig    *config,
                            void              *external)
```

Required Header

stdmpi.h

Description

MotorFlashConfigGet gets a Motor's (***motor***) flash configuration and writes it in the structure pointed to by ***config***, and also writes it in the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Motor's flash configuration information in ***external*** is in addition to the Motor's flash configuration information in ***config***, i.e, the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type MEIMotorConfig{} or is NULL.

Return Values

MPIMessageOK	if *MotorFlashConfigGet* successfully writes the Motor's flash configuration to the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIMotorConfig | MEIFlash | mpiMotorFlashConfigSet

mpiMotorFlashConfigSet

Declaration

```
long mpiMotorFlashConfigSet(MPIMotor        motor,
                            void            *flash,
                            MPIMotorConfig  *config,
                            void            *external)
```

Required Header

stdmpi.h

Description

MotorFlashConfigSet sets a Motor's (***motor***) flash configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Motor's flash configuration information in ***external*** is in addition to the Motor's flash configuration information in ***config***, i.e, the flash configuration information in ***config*** and in external is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type MEIMotorConfig{} or is NULL.

Return Values

MPIMessageOK	if *MotorFlashConfigSet* successfully sets the Motor's flash configuration using data from the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIMotorConfig | MEIFlash | mpiMotorFlashConfigGet

mpiMotorIoGet

Declaration

```
long mpiMotorIoGet(MPIMotor    motor,
                   MPIMotorIo  *io)
```

Required Header

stdmpi.h

Description

MotorIoGet gets a Motor's (***motor***) dedicated I/O bits and writes them into the structure pointed to by ***io***.

Return Values

MPIMessageOK	if *MotorIoGet* successfully writes the Motor's dedicated I/O bits to the structure

See Also

mpiMotorIoSet

mpiMotorIoSet

Declaration

```
long mpiMotorIoSet(MPIMotor    motor,
                   MPIMotorIo  *io)
```

Required Header

stdmpi.h

Description

MotorIoSet sets a Motor's (***motor***) dedicated I/O bits using data from the structure pointed to by ***io***.

Return Values

MPIMessageOK	if *MotorIoSet* successfully sets the Motor's dedicated I/O bits using data from the structure

See Also

mpiMotorIoGet

mpiMotorStatus / meiMotorStatus

mpiMotorStatus

Declaration

```
long mpiMotorStatus(MPIMotor    motor,
                    MPIStatus   *status,
                    void        *external)
```

Required Header stdmpi.h

Description

MotorStatus writes a Motor's (***motor***) status into the structure pointed to by ***status***, and also into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The ***motor's*** status information in ***external*** is in addition to the motor's status information in ***status***, i.e, the status configuration information in ***status*** and in ***external*** is not the same information. Note that ***status*** or ***external*** can be NULL (but not both NULL).

motor	a handle to the Motor object
***status**	a pointer to the motor status structure returned by the method
***external**	external either points to a structure of type MEIMotorStatus{…} or is NULL

Return Values	
MPIMessageOK	if *MotorStatus* successfully gets the status of a Motor object.

See Also MPIStatus | MEIMotorStatus

meiMotorStatus

Declaration

```
long meiMotorStatus(MPIControl  control,
                    long        motorNumber
                    MPIStatus   *status,
                    void        *external)
```

Required Header stdmei.h

Description

MotorStatus gets a Motor's status and writes it to the structure pointed to by ***status***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

control	a handle to the Control object
motorNumber	index to the motor
***status**	a pointer to the motor status structure returned by the method
***external**	external either points to a structure of type MEIMotorStatus{…} or is NULL

Return Values

MPIMessageOK	if *MotorStatus* successfully gets the status of a Motor object.

See Also

MPIStatus | MEIMotorStatus

mpiMotorType

Declaration

```
long mpiMotorType(MPIMotor     motor,
                  MPIMotorType *type)
```

Required Header

stdmpi.h

Description

MotorType writes a Motor's (***motor***) type into the structure pointed to by ***type***.

Return Values

MPIMessageOK	if *MotorType* successfully writes the Motor's type to the structure(s)

See Also

mpiMotorEventConfigGet

Declaration

```
long mpiMotorEventConfigGet(MPIMotor            motor,
                            MPIEventType        eventType,
                            MPIMotorEventConfig *eventConfig,
                            void                *external)
```

Required Header

stdmpi.h

Description

MotorEventConfigGet gets the Motor's (***motor***) configuration for the event specified by ***eventType*** and writes it into the structure pointed to by ***eventConfig***, and also writes it to the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The event configuration information in ***external*** is in addition to the event configuration information in ***eventConfig***, i.e, the event configuration information in ***eventConfig*** and in ***external*** is not the same information. Note: Set ***eventConfig*** or ***external*** to NULL. One must be NULL, the other must be passed a pointer.

XMP Only	***external*** either points to a structure of type MEIMotorEventConfig{} or is NULL.

Return Values

MPIMessageOK	if *MotorEventConfigGet* successfully writes the Motor's configuration for the event to the structure(s)

See Also

MEIMotorEventConfig | mpiMotorEventConfigSet

mpiMotorEventConfigSet

Declaration

```
long mpiMotorEventConfigSet(MPIMotor            motor,
                            MPIEventType        eventType,
                            MPIMotorEventConfig *eventConfig,
                            void                *external)
```

Required Header

stdmpi.h

Description

MotorEventConfigSet

XMP Only	***external*** either points to a structure of type MEIMotorEventConfig{} or is NULL.

Return Values

MPIMessageOK	if *MotorEventConfigGet* successfully writes the Motor's configuration for the event to the structure(s)

See Also

MEIMotorEventConfig | mpiMotorEventConfigGet

mpiMotorEventNotifyGet

Declaration

```
long mpiMotorEventNotifyGet(MPIMotor      motor,
                            MPIEventMask  *eventMask,
                            void          *external)
```

Required Header

stdmpi.h

Description

MotorEventNotifyGet writes the event mask (that specifies the event type(s) for which host notification has been requested) to the location pointed to by ***eventMask***, and also writes it into the implementation-specific location pointed to by ***external*** (if ***external*** is not NULL).

The event notification information in ***external*** is in addition to the event notification information in ***eventmask***, i.e, the event notification information in ***eventmask*** and in ***external*** is not the same information. Note that ***eventmask*** or ***external*** can be NULL (but not both NULL).

Event notification is enabled for event types specified in ***eventmask***, which is a bit mask of MPIEventMask bits associated with the desired MPIEventType values. Event notification is disabled for event types not specified in ***eventmask***. The MPIEventMask bits must be set of cleared using the MPIEventMask macros.

XMP Only

external either points to a structure of type MEIEventNotifyData{} or is NULL. The MEIEventNotifyData{} structure is an array of firmware addresses, whose contents are placed into the MEIEventStatusInfo{} structure (of all events generated by this object).

Return Values

MPIMessageOK	if *MotorEventNotifyGet* successfully writes the event mask to the location(s)

See Also

MPIEventType | MEIEventNotifyData | MEIEventStatusInfo
mpiMotorEventNotifySet

mpiMotorEventNotifySet

Declaration

```
long mpiMotorEventNotifySet(MPIMotor      motor,
                            MPIEventMask  eventMask,
                            void          *external)
```

Required Header

stdmpi.h

Description

MotorEventNotifySet requests host notification of the event(s) that are generated by ***motor*** and specified by ***eventMask***, and also specified by the implementation-specific location pointed to by ***external*** (if ***external*** is not NULL).

The event notification information in ***external*** is in addition to the event notification information in ***eventmask***, i.e, the event notification information in ***eventmask*** and in ***external*** is not the same information. Note that ***eventmask*** or ***external*** can be NULL (but not both NULL).

Event notification is enabled for event types specified in ***eventMask***, a bit mask of MPIEventMask bits associated with the desired MPIEventType values. Event notification is disabled for event types that are not specified in ***eventMask***. The MPIEventMask bits must be set of cleared using the MPIEventMask macros.

The mask of event types generated by a Motor object consists of bits from MPIEventMaskMOTION and MPIEventMaskAXIS.

XMP Only

external either points to a structure of type MEIEventNotifyData{} or is NULL. The MEIEventNotifyData{} structure is an array of firmware addresses, whose contents are placed into the MEIEventStatusInfo{} structure (of all events generated by this object).

To	Then
enable host notification of all events	set ***eventmask*** to MPIEventMaskALL
disable host notification of all events	set ***eventmask*** to MPIEventTypeNONE

Return Values

MPIMessageOK	if *MotorEventNotifySet* successfully requests host notification of the event(s) that are specified by ***eventMask*** and generated by ***motor***

See Also

MPIEventType | MEIEventNotifyData | MEIEventStatusInfo
mpiMotorEventNotifyGet

mpiMotorEventReset

Declaration

```
long mpiMotorEventReset(MPIMotor     motor,
                        MPIEventMask eventMask)
```

Required Header

stdmpi.h

Description

MotorEventReset resets the event(s) that are specified in ***eventMask*** and generated by ***motor***.
Your application must call *MotorEventReset* only after one or more latchable events have occurred.

Return Values

MPIMessageOK	if *MotorEventReset* successfully resets the event(s) that are specified in ***eventMask*** and generated by ***motor***

See Also

mpiMotorEventWait

Declaration

```
long mpiNotifyEventWait(MPINotify      notify,
                        MPIEventStatus *status,
                        MPIWait        timeout)
```

Required Header

stdmpi.h

Description

NotifyEventWait sets the contents of the structure pointed to by ***status***, using the status of the first event in the internal FIFO event queue (maintained by a Notify object (***notify***)), and then removes the first event from the queue. If no event is available in the internal FIFO event queue, NotifyEventWait will wait for ***timeout*** milliseconds.

If "timeout" is	*Then*
MPIWaitPOLL (**0**)	*NotifyEventWait* will not wait for an event to arrive
MPIWaitFOREVER (**-1**)	*NotifyEventWait* will wait forever for an event to arrive

Return Values	
MPIMessageOK	if *NotifyEventWait* successfully sets the contents of the structure pointed to by ***status***, and then removes the first event from the queue
MPIMessageTIMEOUT	if no event is present and the contents of ***status*** are undefined

See Also

mpiMotorMemory

Declaration

```
long mpiMotorMemory(MPIMotor  motor,
                    void      **memory)
```

Required Header stdmpi.h

Description **MotorMemory** writes an address [which can be used to access a Motor's (***motor***) memory] to the contents of ***memory***. This address, or an address calculated from it, can be passed as the ***src*** parameter to MPIMotorMemoryGet(...) and as the ***dst*** parameter to MPIMotorMemorySet(...).

Return Values	
MPIMessageOK	*MotorMemory* successfully writes the Motor's address to the contents of memory

See Also MPIMotorMemoryGet | MPIMotorMemorySet

mpiMotorMemoryGet

Declaration

```
long mpiMotorMemoryGet(MPIMotor motor,
                       void     *dst,
                       void     *src,
                       long     count)
```

Required Header

stdmpi.h

Description

MotorMemoryGet copies ***count*** bytes of a Motor's (***motor***) memory (starting at address ***src***) to application memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *MotorMemoryGet* successfully copies data from Motor memory to application memory

See Also

mpiMotorMemorySet | mpiMotorMemory

mpiMotorMemorySet

Declaration

```
long mpiMotorMemorySet(MPIMotor motor,
                       void     *dst,
                       void     *src,
                       long     count)
```

Required Header

stdmpi.h

Description

MotorMemorySet copies ***count*** bytes of application memory (starting at address ***src***) to a Motor's (***motor***) memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *MotorMemorySet* successfully copies data from application memory to Motor memory

See Also

mpiMotorMemoryGet | mpiMotorMemory

meiMotorEndcoderInit

Declaration

```
long meiMotorEncoderInit(MPIMotor       motor,
                         MEIMotorOutput output,
                         MPIObjectMap   map)
```

Required Header stdmei.h

Description **MotorEncoderInit** inititalizes one or more absolute encoders associated with the specified MPIObjectMap map, using an output signal associated with the motor.

Return Values	
MPIMessageOK	if meiMotorEncoderInit(...) successfully initializes absolute encoder(s).
MEIMotorMessage ABS_ENCODER_FAULT	Returned when a protocol error occurs during the serial data transmission from the drive to controller.
MEIMotorMessage ABS_ENCODER_TIMEOUT	Returned when the controller does not receive serial data from the drive.

See Also MPIObjectMap | meiMotorEncoderInit

meiMotorEncoderReset

Declaration `long meiMotorEncoderReset(MPIMotor motor)`

Required Header stdmei.h

Description **MotorEncoderReset** clears the broken wire and illegal state detection latches for the encoder associated with the specified motor.

Return Values

MPIMessageOK	if *MotorEncoderReset(...)* successfully resets the motor's encoder value.

See Also

mpiMotorControl

Declaration

```
const MPIControl mpiMotorControl(MPIMotor motor)
```

Required Header

stdmpi.h

Description

MotorControl returns a handle to the Control object with which the motor is associated.

motor	a handle to the Motor object

Return Values

MPIControl	handle to a Control object
MPIHandleVOID	if motor is invalid

See Also

mpiMotorCreate | mpiControlCreate

mpiMotorFilterMapGet

Declaration

```
long mpiMotorFilterMapGet(MPIMotor     motor,
                          MPIObjectMap *map)
```

Required Header stdmpi.h

Description **MotorFilterMapGet** gets the object map of the Filters [that are associated with a Motor (***motor***)] and writes it into the structure pointed to by ***map.***

Return Values	
MPIMessageOK	if *MotorFilterMapGet* successfully writes the Filter's object map (associated with a Motor) to the structure

See Also mpiMotorFilterMapSet

mpiMotorFilterMapSet

Declaration

```
long mpiMotorFilterMapSet(MPIMotor     motor,
                          MPIObjectMap map)
```

Required Header stdmpi.h

Description **MotorFilterMapSet** sets the Filters [that are associated with a Motor (***motor***)], using data from the object map specified by ***map***.

Return Values	
MPIMessageOK	if *MotorFilterMapSet* successfully sets the Filters using data from the object map

See Also mpiMotorFilterMapGet

mpiMotorNumber

Declaration

```
long mpiMotorNumber(MPIMotor motor,
                    long     *number)
```

Required Header

stdmpi.h

Description

MotorNumber writes the index of a Motor (***motor***, on the motion controller that ***motor*** is associated with) to the contents of ***number***.

Return Values

MPIMessageOK	if *MotorNumber* successfully writes the Motor's index to the contents of ***number***

See Also

meiMotorCompareListGet

Declaration

```
long meiMotorCompareListGet(MPIMotor motor,
                            long          *compareCount,
                            long          *compareList);
```

Required Header

stdmei.h

Description

MotorCompareListGet sets ***compareCount*** to the number of compares that ***motor*** has. It sets ***compareList*** to the number of the compare object for each compare.

Return Values

MPIMessageOK	if *motor* is a valid MPIMotor object.

See Also

mpiMotorDedicatedInAddrGet

Declaration

```
long mpiMotorDedicatedInAddrGet(MPIMotor motor,
                                long     motorNumber)
```

Required Header stdmpi.h

Description **MotorDedicatedInAddrGet** gets the address of the Dedicated IO for the Motor that is specified by ***motorNumber***.

Return Values

Address	The address of where the Dedicated IO is located in XMP memory.

See Also

meiMotorDedicatedIoAddrDecode

Declaration

```
long meiMotorDedicatedIOAddrDecode(MPIMotor    motor,
                                   long        addr,
                                   long*       motorNumber);
```

Required Header

stdmei.h

Description

MotorDedicatedIoAddrDecode takes an addr and decodes which motor's dedicated IO this ***addr*** points to, and then sets ***motorNumber*** to that motor's motorNumber.

Return Values

MPIMessageOK	if *motor* is a valid handle and *addr* is a valid address (eg. it is located in the motor memory space on the controller).

See Also

meiMotorDedicatedOutAddrGet

Declaration

```
long meiMotorDedicatedOutAddrGet(MPIMotor motor,
                                 long     motorNumber);
```

Required Header

stdmei.h

Description

MotorDedicatedOutAddrGet gets the address of the Dedicated IO for the Motor that is specified by ***motorNumber***.

Return Values

Address	The address of where the Dedicated IO is located in XMP memory.

See Also

meiMotorEncoderRatio

Declaration

```
long meiMotorEncoderRatio(MPIControl           control,
                          long                 motorNumber,
                          long                 encoderNumber,
                          MEIMotorEncoderRatio *ratio)
```

Required Header stdmei.h

Description **MotorEncoderRatio** gets encoder ratio from the XMP.

Return Values	
MPIMessageOK	if *meiMotorEncoderRatio* successfully gets encoder ration from the XMP.
MPIMessageARG_INVALID	if arguments are not valid

See Also

meiMotorRelatedStepMotorGet

Declaration

```
long meiMotorRelatedStepMotorGet(MPIMotor    motor,
                                 long        *motorNumber)
```

Required Header stdmei.h

Description

MotorRelatedStepMotorGet searches for another motor that shares the resources of the motor provided. It then sets ***motorNumber*** to that motor's number. This scenario is only possible among Stepper Motors on a XMP Pulse controller. If there are no related motors found, ***motorNumber*** is set to MEIMotorResourceNumberINVALID.

Return Values

MPIMessageOK	if *motor* is a valid MPIMotor object.

See Also See the Software section of the XMP Pulse App Note 218.

MPIMotorBrake

MPIMotorBrake

```
typedef struct MPIMotorBrake {
    MPIMotorBrakeMode   mode;
    float               enableDelay;
    float               disableDelay;

} MPIMotorBrake;
```

Description

MotorBrake allows the ability to enable and disable a brake when the motor's amp enable output is enabled or disabled.

The reason for this is easiest to imagine on a vertical axis of motion: If you release the brake before enabling servo control on a vertical axis, the axis will not be controlled and will fall under the influence of gravity. Likewise, when setting a brake on a vertical axis, you want to set the brake before turning off the amplifier so that no motion occurs when disabling the servo control.

NOTE: the only output capable of being used for a brake is the USER I/O 0 output of the motor.

mode	Specifies whether a brake is to be tied to the motor's amp enable output.
enableDelay	The amount of time the brake is enabled before servo control is disabled. **NOTE:** the brake is enabled immediately, the delay is for servo control.
disableDelay	The amount of time servo control is turned on before the brake is disabled.

See Also

MPIMotorBrakeMode

MPIMotorBrakeMode

```
typedef enum{
     MPIMotorBrakeModeINVALID = -1,
     MPIMotorBrakeModeNONE,
     MPIMotorBrakeModeDELAY,

} MPIMotorBrakeMode;
```

Description

MotorBrakeMode specifies whether a brake enable is tied to the amp enable output.

MPIMotorBrakeModeNONE	No brake enable output is tied to the amp enable output.
MPIMotorBrakeModeDELAY	Ties a brake enable output to the amp enable output.

See Also

MPIMotorConfig / MEIMotorConfig

MPIMotorConfig

```
typedef struct MPIMotorConfig {
    MPIMotorType    type;

    /* Event configuration, ordered by MPIEventType */
    MPIMotorEventConfig         event[MPIEventTypeMOTOR_LAST];

    long    ampEnablePolarity;  /* FALSE => active lo, else active hi */
    long    encoderPhase;       /* 0 => normal, else reversed */
    long    captureOnChange;    /* 0 => normal, else enabled */

    float           abortDelay;
    float           enableDelay;
    MPIMotorBrake   brake;

    MPIObjectMap    filterMap;

    MPIMotorIo       io;
} MPIMotorConfig;
```

Description

event	Structure to configure various Motor Events. See MPIMotorEventConfig description.
ampEnablePolarity	Configures Amplifier Enable Output polarity. For active low signal = FALSE, for Active High = TRUE
encoderPhase	Configures encoder phasing. Normal (A rising, B rising, A falling, etc.) = 0, Reversed = non-zero.
captureOnChange	Configures Captures objects for this motor to Capture on Rising and Falling transitions.
abortDelay	Sets time value, in seconds, to delay Abort action after Event has occured.
enableDelay	Sets time value, in seconds, to delay Enabling of the amplifier after commanded.
brake	Settings for tying a brake enable output to the amp enable output.
filterMap	Get/Set a map of Filter Objects to which the Motor is mapped. Default mapping is Filter 0 to Motor 0, Fitler 1 to Motor 1, etc. See also MPIObjectMap description in Object section.
io	Structure to read the Motor input and set Motor output values.. See MPIMotorIo description.

MEIMotorConfig

```
typedef struct MEIMotorConfig {
    MEIMotorEncoder      Encoder[MEIXmpMotorEncoders];
    MEIXmpIO             StatusOutput[MEIXmpMotorStatusOutputs];
    MEIMotorTransceiver  Transceiver[MEIXmpMotorTransceivers];
    MEIMotorTransceiver  TransceiverExtended[MEIXmpMotorTransceiversExtended];
    long                 UserOutInvert;    /* Opto Polarity */
    MEIMotorStepper      Stepper;
    long                 EncoderTermination;
    long                 SIM4;
    MEIMotorDacConfig    Dac;
    long                 pulseEnable;     /* 0 => normal, else pulse output */
    long                 pulseWidth;      /* 0.1 to 25.5 microseconds */

    /* Commutation is read-only from field Theta to end*/
    MEIXmpCommutationBlock      Commutation;

    MEIXmpLimitData     Limit[MEIXmpLimitLAST];

    MEIXmpMotorTorqueLimitConfig TorqueLimitConfig;

    long AmpDisableWithLSR;     /* TRUE => XMP disables amp when LSR is active */

    MEIMotorFilterInput FilterInput[MEIXmpMotorFilterInputs];
} MEIMotorConfig;
```

Description

	Encoder	Structure to configure Motor Encoder type and parameters
	Transceiver	Structure to configure Motor Transceivers. See MEIMotorTransceiver description.
	TransceiverExtended	Structure to configure Motor Extended Transceivers. See MEIMotorTransceiver description.
	UserOutInvert	Inverts the User Opto Polarity when wired as an Output.
	Stepper	Structure to configure Motor Stepper parameters. See MEIMotorStepper description.
	EncoderTermination	Enables encoder termination when set to 1. Enables 100 ohm resistor between complementary encoder channels. Set to 0 (zero) to disable.
	SIM4	Enables use of SIM4 scale interpolation module when set to 1. Set to 0 (zero) to disable.
	Dac	Structure that includes Command and Auxiliary DAC configuration for each motor. See MEIMotorDacConfig description.
	pulseEnable	Enables the Divide-by-N output pulse. See MEICompareConfig description.
	pulseWidth	Sets width of Step pulse. Valid values range from a minimum width of 0.1usec to a maximum width of 25.5usec.

See Also mpiMotorConfigGet | mpiMotorConfigSet

MEIMotorDacConfig

MEIMotorDacConfig

```
typedef struct MEIMotorDacConfig {
    MEIXmpDACPhase              Phase;
    MEIMotorDacChannelConfig    Cmd;
    MEIMotorDacChannelConfig    Aux;
} MEIMotorDacConfig;
```

Description

MotorDacConfig is a structure that includes Command and Auxiliary DAC configuration for each motor.

See Also

meiMotorDacConfigGet | meiMotorDacConfigSet

MEIMotorDacChannelConfig

MEIMotorDacChannelConfig

```
typedef struct MEIMotorDacChannelConfig {
    float               Offset; /* volts */
    float               Scale;
    MEIXmpDACInputType  InputType;
    MEIXmpGenericValue  *Input;
} MEIMotorDacChannelConfig;
```

Description

MotorDacChannelConfig is a structure used to configure the DAC settings.

Offset	Set DAC Offset value. Valid values range from -10 Volts to +10 Volts.

See Also

MEIMotorDacChannelStatus

MEIMotorDacChannelStatus

```
typedef struct MEIMotorDacChannelStatus {
    float       level;  /* volts */
} MEIMotorDacChannelStatus;
```

Description

MotorDacChannelStatus is a structure that returns the DAC output value.

level	*level* reflects the DAC output value. Valid values range from -10 Volts to +10 Volts.

See Also

MEIMotorDacStatus

MEIMotorDacStatus

```
typedef struct MEIMotorDacStatus {
    MEIMotorDacChannelStatus    cmd;
    MEIMotorDacChannelStatus    aux;
} MEIMotorDacStatus;
```

Description

MotorDacStatus is a structure that returns the Status for both Command and Auxiliary DACs. It is used to read the ***cmd*** and ***aux*** DAC level (in volts) from the controller.

See Also

MPIMotorEncoderFault

MPIMotorEncoderFault

```
typedef enum {
    MPIMotorEncoderFaultINVALID,
    MPIMotorEncoderFaultBW_DET,
    MPIMotorEncoderFaultILL_DET,
    MPIMotorEncoderFaultABS_ERR,

} MPIMotorEncoderFault;
```

Description

MotorEncoderFault is an enumeration used to get/set Encoder Fault.

MPIMotorEncoderFaultBW_DET	Encoder Broken Wire detection enabled. Error returned if complementary signals are in the same state. (e.g. A+ and A- return 5V.)
MPIMotorEncoderFaultILL_DET	Encoder Illegal State detection enabled.
MPIMotorEncoderFaultABS_ERR	Absolute encoder error detection enabled.

See Also

MPIMotorEncoderFaultMask

MPIMotorEncoderFaultMask

```
typedef enum {
     MPIMotorEncoderFaultMaskNONE,
     MPIMotorEncoderFaultMaskBW_DET,
     MPIMotorEncoderFaultMaskILL_DET,
     MPIMotorEncoderFaultMaskABS_ERR,
     MPIMotorEncoderFaultMaskALL
} MPIMotorEncoderFaultMask;
```

Description

MotorEncoderFaultMask is an enumeration to mask bits from MPIMotorEncoderFault register.

MPIMotorEncoderFaultMaskBW_DET	Mask for Broken Wire detection
MPIMotorEncoderFaultMaskILL_DET	Mask for Illegal State detection
MPIMotorEncoderFaultMaskABS_ERR	Mask for Absolute Encoder Error

See Also

MPIMotorEventConfig / MEIMotorEventConfig

MPIMotorEventConfig

```
typedef struct MPIMotorEventConfig {
    MPIAction             action;
    MPIMotorEventTrigger  trigger;
    long                  direction;
    float                 duration;     /* seconds */
} MPIMotorEventConfig;
```

Description

MotorEventConfig is a structure used to configure Motor Events.

direction	for Hardware and Software limits, enabling "direction" requires the direction of motion to be in direction of Limit.
duration	time that Limit (e.g. Home, Pos. and Neg. Limits, User Limit) must be asserted before Event is generated. Value in seconds.

MEIMotorEventConfig

```
typedef MEIXmpLimitData MEIMotorEventConfig;
typedef struct {
    MEIXmpLimitCondition    Condition[MEIXmpLimitConditions];
    MEIXmpStatus            Status;
    MEIXmpLogic             Logic;
    MEIXmpLimitOutput       Output;
    long                    Count;
    long                    State;
} MEIXmpLimitData;
```

Description

condition	is a structure that configures the conditioanl statements evaluated to generate a Limit Event. Each limit may have up to two conditions (MEIXmpLimitConditions = 2). This structure is described in further detail in App Note 215.
status	an enum that defines what actions the XMP will take when a user limit evaluates TRUE. Always set Status to at least MEIXmpStatusLIMIT to notify the motor object that a limit has occurred. Valid **Status** values are listed in the "Values of Status" table below.
logic	an enum that sets the logic applied between the two condition block outputs, Condition[0] and Condition[1]. Valid **Logic** values are listed in the "Values of Status" table below.
output	is a structure that allows specific action to be taken when the Limit Event is generated. In addition to generatign a Motion Action, a Limit can write to any other valid XMP Firmware register defined in the *OutputPtr with value described by AndMask and OrMask. This structure is described in further detail in App Note 215.
count	For internal use only. The MPI method, mpiMotorEventConfigSet(...) will not write these values.
state	For internal use only. The MPI method, mpiMotorEventConfigSet(...) will not write these values.

Value of Status	Action to be taken
MEIXmpStatusLIMIT	None
MEIXmpStatusLIMIT | MEIXmpStatusPAUSE	Axes attached to the motor will be Paused
MEIXmpStatusLIMIT | MEIXmpStatusSTOP	Axes attached to the motor will be Stopped
MEIXmpStatusLIMIT | MEIXmpStatusABORT	Axes attached to the motor will be Aborted
MEIXmpStatusLIMIT | MEIXmpStatusESTOP	Axes attached to the motor will be E-Stopped
MEIXmpStatusLIMIT | MEIXmpStatusESTOP_ABORT	Axes attached to the motor will be E-Stopped and Aborted

Value of Logic	Evaluates	Motor object notified that a limit has occurred if...
MEIXmpLogicNEVER	Nothing	No event is generated
MEIXmpLogicSINGLE	Condition[0]	Condition[0] = = TRUE
MEIXmpLogicOR	Condition[0], Condition[1]	(Condition[0] || Condition[1]) = = TRUE
MEIXmpLogicAND	Condition[0], Condition[1]	(Condition[0] && Condition[1]) = = TRUE
other **MEIXmpLogic** enums	For internal use only.	

See Also

Special Note: MPIMotorEventConfig and Motor Limit Configuration
See App Note 215 for a more in-depth breakdown of this structure.
mpiMotorEventConfigGet | mpiMotorEventConfigSet

MPIMotorEventTrigger

MPIMotorEventTrigger

```
typedef union {
    long        polarity;       /* 0 => active low, else active high */
    long        position;       /* MPIEventTypeLIMIT_SW_[POS|NEG] */
    float       error;          /* MPIEventTypeLIMIT_ERROR */
    long        mask;           /* MPIEventTypeENCODER_FAULT */
} MPIMotorEventTrigger;
```

Description

polarity	configures the polarity for Motor Event. Active Low = 0, Active High = non-zero.
position	configures position at which Positive and Negative Software limits generate Events.

See Also

MEIMotorInput

MEIMotorInput

```
typedef enum {
    MEIMotorInputSIM4_INDEX,
    MEIMotorInputSIM4_ENCB,
    MEIMotorInputSIM4_ENCA,
    MEIMotorInputXCVR_A,
    MEIMotorInputXCVR_B,
    MEIMotorInputXCVR_C,

    MEIMotorInputBROKEN_WIRE,
    MEIMotorInputILLEGAL_STATE,

    MEIMotorInputOVERTRAVEL_POS,
    MEIMotorInputOVERTRAVEL_NEG,
    MEIMotorInputHOME,
    MEIMotorInputAMP_FAULT,
    MEIMotorInputINDEX,

    MEIMotorInputUSER,
    MEIMotorInputCAPTURE,
} MEIMotorInput;
```

Description

MotorInput is an enumeration of the Motor Input register.

See Also

MPIMotorIo

MPIMotorIo

```
typedef struct MPIMotorIo {
    unsigned long       input;
    unsigned long       output;
} MPIMotorIo;
```

Description

MotorIo is a structure used to read the Motor input and set Motor output values.

input	for XMP users, 'input' value reflects MEIMotorInput. See MEIMotorInput description.
output	for XMP users, 'output' value reflects MEIMotorOutput. See MEIMotorOutput description.

See Also

mpiMotorIoGet | mpiMotorIoSet

MPIMotorMessage / MEIMotorMessage

MPIMotorMessage

```
typedef enum {

    MPIMotorMessageMOTOR_INVALID,

} MPIMotorMessage;
```

Description **MotorMessage** is an enumeration of Motor error messages that can be returned by the MPI library.

MEIMotorMessage

```
typedef enum {
    MEIMotorMessageABS_ENCODER_FAULT,
    MEIMotorMessageABS_ENCODER_TIMEOUT,
    MEIMotorMessageMOTOR_NOT_ENABLED,

} MEIMotorMessage;
```

Description **MotorMessage** is an enumeration of Motor error messages that can be returned by the MPI library.

See Also

MEIMotorOutput

MEIMotorOutput

```
typedef enum {
    MEIMotorOutputXCVR_A,
    MEIMotorOutputXCVR_B,
    MEIMotorOutputXCVR_C,
    MEIMotorOutputBRAKE_ENABLE,
    MEIMotorOutputUSER,
    MEIMotorOutputCOMPARE,
} MEIMotorOutput;
```

Description

MotorOuput lists the Motor Output enumeration values.

See Also

MEIMotorResourceNumber

MEIMotorResourceNumber

```
typedef enum {
    MEIMotorResourceNumberINVALID,
    MEIMotorResourceNumber0,
    MEIMotorResourceNumber1,
    MEIMotorResourceNumber2,
    MEIMotorResourceNumber3,
    MEIMotorResourceNumber4,
    MEIMotorResourceNumber5,
    MEIMotorResourceNumber6,
    MEIMotorResourceNumber7,
    MEIMotorResourceNumber8,
    MEIMotorResourceNumber9,
    MEIMotorResourceNumber10,
    MEIMotorResourceNumber11,
    MEIMotorResourceNumber12,
    MEIMotorResourceNumber13,
    MEIMotorResourceNumber14,
    MEIMotorResourceNumber15,
    MEIMotorResourceNumber16,
    MEIMotorResourceNumber17,
    MEIMotorResourceNumber18,
    MEIMotorResourceNumber19,
    MEIMotorResourceNumber20,
    MEIMotorResourceNumber21,
    MEIMotorResourceNumber22,
    MEIMotorResourceNumber23,
    MEIMotorResourceNumber24,
    MEIMotorResourceNumber25,
    MEIMotorResourceNumber26,
    MEIMotorResourceNumber27,
    MEIMotorResourceNumber28,
    MEIMotorResourceNumber29,
    MEIMotorResourceNumber30,
    MEIMotorResourceNumber31,

}MEIMotorResourceNumber;
```

Description

MotorResourceNumber is an enumeration value used only for XMP-Pulse controllers. All other XMP controllers should have the Step Resource Number set equal to the Motor number. XMP-Pulse users can set the Resource Number to allow sharing of a single motor block's resources between two stepper motors. e.g. Motors 0 and 16 could both be configured to use the output signals from Motor Block 0 by each having their Motor Resource Number set to 0.

See Also

MEIMotorStatus

MEIMotorStatus

```
typedef struct MEIMotorStatus {
    MEIMotorDacStatus   dac;
} MEIMotorStatus;
```

Description

MotorStatus is a structure that returns XMP specific Motor Status registers.

See Also

MEIMotorStepper

MEIMotorStepper

```
typedef struct MEIMotorStepper {
    float       PulseWidth;     /* output pulse width  (sec) */
    long        Loopback;       /* TRUE = count step pulses in encoder reg. */
    MEIMotorResourceNumber      ResourceNumber;
} MEIMotorStepper;
```

Description

MotorStepper is a structure used to configure Stepper Motor parameters.

PulseWidth	sets width of Step pulse. Valid values range from a minimum width of 0.1 usec to maximum width of 25.5 usec.
LoopBack	enables Step Loopback feature. When enabled, step output pulses counted to generate Actual Position. When disabled, external feedback device is required to generate actual position.

See Also

MEIMotorTransceiver

MEIMotorTransceiver

```
typedef struct MEIMotorTransceiver {
        long                            Invert;
                        /* TRUE = invert (not valid for INPUT) */
        MEIMotorTransceiverConfig       Config;
} MEIMotorTransceiver;
```

Description

MotorTransceiver is a structure used to configure Transceiver parameters.

Invert	Inverts polarity of Transceiver Output. Not valid when the Transeiver is configured as an Input.
Config	Enumeration to configure Transceiver usage. See MEIMotorTransceiverConfig description.

See Also

MEIMotorTransceiverConfig

MEIMotorTransceiverConfig

```
typedef enum {
    MEIMotorTransceiverConfigINVALID,
    MEIMotorTransceiverConfigINPUT,                 /* 0 */
    MEIMotorTransceiverConfigOUTPUT,                /* 1 */
    MEIMotorTransceiverConfigSTEP,                  /* 2 */
    MEIMotorTransceiverConfigDIR,                   /* 3 */
    MEIMotorTransceiverConfigCW,                    /* 4 */
    MEIMotorTransceiverConfigCCW,                   /* 5 */
    MEIMotorTransceiverConfigQUAD_A,                /* 6 */
    MEIMotorTransceiverConfigQUAD_B,                /* 7 */
    MEIMotorTransceiverConfigCOMPARE,               /* 8 */
    MEIMotorTransceiverConfigDIAG,                  /* 9 */
    MEIMotorTransceiverConfigNOT_AVAILABLE,

} MEIMotorTransceiverConfig;
```

Description

MotorTranceiverConfig is a structure used to configure various Motor Events.

MEIMotorTransceiverConfigINPUT	Transceiver configured as Input
MEIMotorTransceiverConfigOUTPUT	Transceiver configured as Output
MEIMotorTransceiverConfigSTEP	Transceiver configured as Step pulse output
MEIMotorTransceiverConfigDIR	Transceiver configured as Direction signal output
MEIMotorTransceiverConfigCW	Transceiver configured as Clockwise Step Output
MEIMotorTransceiverConfigCCW	Transceiver configured as Counterclockwise Step Output
MEIMotorTransceiverConfigQUAD_A	Transceiver configured as Quadrature A Output
MEIMotorTransceiverConfigQUAD_B	Transceiver configured as Quadrature B Output
MEIMotorTransceiverConfigCOMPARE	Transceiver configured for use as Compare Output

See Also

MEIMotorTransceiverExtendedId

MEIMotorTransceiverExtendedId

```
typedef enum {
    MEIMotorTransceiverExtendedIdINVALID,

    MEIMotorTransceiverExtendedIdD,
    MEIMotorTransceiverExtendedIdE,
    MEIMotorTransceiverExtendedIdF,

} MEIMotorTransceiverExtendedId;
```

Description

MotorTranceiverExtendedId is an enumeration of Extended Transceiver Identification. Not valid on all XMP controllers.

See Also

MEIMotorTransceiverExtendedMask

MEIMotorTransceiverExtendedMask

```
typedef enum {
    MEIMotorTransceiverExtendedMaskD,
    MEIMotorTransceiverExtendedMaskE,
    MEIMotorTransceiverExtendedMaskF,
} MEIMotorTransceiverExtendedMask;
```

Description

MotorTransceiverExtendedMask is an enumeration of Extended Transceiver I/O masks. Not valid on all XMP controllers.

See Also

MEIMotorTransceiverId

MEIMotorTransceiverId

```
typedef enum {
    MEIMotorTransceiverIdINVALID,

    MEIMotorTransceiverIdA,
    MEIMotorTransceiverIdB,
    MEIMotorTransceiverIdC,

} MEIMotorTransceiverId
```

Description

MotorTraceiverId is an enumeration of Transceiver Identification.

See Also

MEIMotorTransceiverMask

MEIMotorTransceiverMask

```
typedef enum {
    MEIMotorTransceiverMaskA,
    MEIMotorTransceiverMaskB,
    MEIMotorTransceiverMaskC,
} MEIMotorTransceiverMask;
```

Description

MotorTranceiverMask is an enumeration of Extended Transceiver I/O masks. Not valid on all XMP controllers.

See Also

MPIMotorType

MPIMotorType

```
typedef enum {
    MPIMotorTypeINVALID,

    MPIMotorTypeSERVO,
    MPIMotorTypeSTEPPER,
    MPIMotorTypeSERCOS_DRIVE,

} MPIMotorType;
```

Description

MotorType is an enumeration of valid Motor Types.

MPIMotorTypeSERVO	Motor configured as Servo
MPIMotorTypeSTEPPER	Motor configured as Stepper
MPIMotorTypeSERCOS_DRIVE	Motor configured as SERCOS drive

See Also

MEIMotorTypeInfo

MEIMotorTypeInfo

```
typedef union {
    struct {
        long    sercosNumber;
        long    nodeNumber;
    } sercos;
} MEIMotorTypeInfo;
```

Description

MotorTypeInfo is a union with one member (a sercos structure) that contains the sercosNumber and nodeNumber for that motor.

See Also

mpiMotorEncoderFaultMaskBIT

Declaration

```
#define mpiMotorEncoderFaultMaskBIT(fault)      (0x1 << (fault))
```

Required Header

stdmpi.h

Description

MotorEncoderFaultMaskBIT converts the motor encoder fault into the motor encoder fault mask.

See Also

MPIMotorEncoderFault | MPIMotorEncoderFaultMask

Node Objects

Introduction

A **Node** object manages an individual SERCOS node. A SERCOS node may be a drive or I/O module.

Methods

Create, Delete, Validate Methods

mpiNode**Create**	Create Node object
mpiNode**Delete**	Delete Node object
mpiNode**Validate**	Validate Node object

Configuration and Information Methods

mpiNode**ConfigGet**	Get Node config
mpiNode**ConfigSet**	Set Node config
mpiNode**FlashConfigGet**	Get flash configuration for Node
mpiNode**FlashConfigSet**	Set flash configuration for Node
mpiNode**IdnDataGet**	
mpiNode**IdnDataSet**	
mpiNode**IdnFieldGet** / meiNode**IdnFieldGet**	
mpiNode**IdnFieldSet** / meiNode**IdnFieldSet**	
mpiNode**ServiceProcedure**	Execute the procedure in the node described by idn
mpiNode**Status**	Get the status of a Node

Memory Methods

mpiNode**Memory**	Get address of Node memory

Relational Methods

mpiNode**IdnListGet**	Get IdnList associated with a Node
mpiNode**IdnListSet**	Set IdnList associated with a Node
mpiNode**Number**	Get index number of Node
mpiNode**Sercos**	

Other Methods

meiNode**OperationMode**	
meiNode**RealTimeControlWordGet**	
meiNode**RealTimeControlWordSet**	Write the real time control word to a node.
meiNode**RealTimeStatusWordGet**	
meiNode**TelegramType**	

Data Types

MPINode**Config** / MEINode**Config**
MPINode**IdnListType**
MEINode**IdnCyclic**
MEINode**IdnData**
MPINode**Mode**
MEINode**RealTimeBit**
MEINode**RealTimeWord**

Macros

mpiNode**IdnDataGET**
mpiNode**IdnGET**
mpiNode**IdnSET**

Copyright © 2002
Motion Engineering

mpiNodeCreate

Declaration

```
const mpiNodeCreate(MPISercos sercos,
                    long      number)
```

Required Header

stdmpi.h

Description

NodeCreate creates a node object associated with the node identified by ***number*** located on a SERCOS ring, ***sercos***. NodeCreate is the equivalent of a C++ constructor.

Return Values

handle	to a Node object
MPIHandleVOID	if the object could not be created

See Also

mpiNodeValidate | mpiNodeDelete

mpiNodeDelete

Declaration `long mpiNodeDelete(MPINode node)`

Required Header stdmpi.h

Description **NodeDelete** deletes a Node object and invalidates its handle (***node***). *NodeDelete* is the equivalent of a C++ destructor.

Return Values	
MPIMessageOK	if *NodeDelete* successfully deletes the Node object and invalidates its handle

See Also mpiNodeCreate | mpiNodeValidate

mpiNodeValidate

Declaration `long mpiNodeValidate(MPINode node)`

Required Header stdmpi.h

Description **NodeValidate** validates the Node object and its handle (***node***).

Return Values	
MPIMessageOK	if Node is a handle to a valid object.

See Also mpiNodeCreate | mpiNodeDelete

mpiNodeConfigGet

Declaration

```
long mpiNodeConfigGet(MPINode        node,
                      MPINodeConfig  *config,
                      void           *external)
```

Required Header

stdmpi.h

Description

NodeConfigGet gets a Node object's (***node***) configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is *in addition* to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type MEINodeConfig{} or is NULL.

Return Values

MPIMessageOK	if *NodeConfigGet* successfully writes the Node object's configuration to the structure(s)

See Also

MEINodeConfig | mpiNodeConfigSet

mpiNodeConfigSet

Declaration

```
long mpiNodeFlashConfigSet(MPINode        node,
                           void           *flash,
                           MPINodeConfig  *config,
                           void           *external)
```

Required Header

stdmpi.h

Description

NodeFlashConfigSet sets a Node object's (***node***) flash configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL). The Node object's flash configuration information in external is in addition to the Node object's flash configuration information in ***config***, i.e, the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type MEINodeConfig{} or is NULL.

Return Values

MPIMessageOK	if *NodeFlashConfigSet* successfully sets the Node object's flash configuration using data from the structure(s) flash is either an MEIFlash handle or MPIHandleVOID. If flash is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEINodeConfig | MEIFlash | mpiNodeConfigGet

mpiNodeFlashConfigGet

Declaration

```
long mpiNodeFlashConfigGet(MPINode          node,
                           void             *flash,
                           MPINodeConfig    *config,
                           void             *external)
```

Required Header

stdmpi.h

Description

NodeFlashConfigGet gets a Node object's (***node***) flash configuration and writes it in the structure pointed to by ***config***, and also writes it in the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Node object's flash configuration information in ***external*** is in addition to the Node object's flash configuratiosn information in ***config***, i.e, the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type MEINodeConfig{} or is NULL.

Return Values

MPIMessageOK	if *NodeFlashConfigGet* successfully writes the Node object's flash configuration to the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEINodeConfig | MEIFlash | mpiNodeFlashConfigSet

mpiNodeFlashConfigSet

Declaration

```
long mpiNodeFlashConfigSet(MPINode        node,
                           void           *flash,
                           MPINodeConfig  *config,
                           void           *external)
```

Required Header

stdmpi.h

Description

mpiNodeFlashConfigSet sets a Node object's (***node***) flash configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Node object's flash configuration information in ***external*** is in addition to the Node object's flash configuration information in ***config***, i.e, the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type MEINodeConfig{} or is NULL.

Return Values

MPIMessageOK	if *NodeFlashConfigSet* successfully sets the Node object's flash configuration using data from the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEINodeConfig | MEIFlash | mpiNodeFlashConfigGet

meiNodeIdnDataGet / mpiNodeIdnDataGET

meiNodeIdnDataGet

Declaration

```
long meiNodeIdnDataGet(MPIControl       control,
                       MEIMotorTypeInfo *motorInfo,
                       long             count,
                       MEINodeIdnData   *idnData)
```

Required Header stdmei.h

Description **NodeIdnDataGet** reads the idn data and writes it to the structure pointed to by ***idnData***. The data is read from the controller associated with the control object and the nodeNumber specified in the structure pointed to by ***motorInfo***.

control	a handle to a Control object
***motorInfo**	a pointer to a MEIMotorTypeInfo structure
count	number of MEINodeIdnData structures to read
***idnData**	a pointer to an array of MEINodeIdnData structures

Return Values

MPIMessageOK	if *NodeIdnDataGet* successfully reads the idn data.

meiNodeIdnDataGET

Declaration

```
#define  mpiNodeIdnDataGET(node, idn)
         mpiNodeIdnFieldGet((node), (idn), MPIIdnFieldDATA)
```

Required Header stdmpi.h

Description **NodeIdnDataGET** reads the data field from an ***idn*** located on a ***node***, and writes them into the idn object.

See Also meiNodeIdnDataSet

meiNodeIdnDataSet / mpiNodeIdnDataSET

meiNodeIdnDataSet

Declaration

```
long meiNodeIdnDataSet(MPIControl        control,
                       MEIMotorTypeInfo *motorInfo,
                       long              count,
                       MEINodeIdnData    *idnData)
```

Required Header stdmei.h

Description **NodeIdnDataSet** writes the idn data from the structure pointed to by ***idnData***. The data is written to the controller associated with the control object and the nodeNumber specified in the structure pointed to by ***motorInfo***.

control	a handle to a Control object
***motorInfo**	a pointer to a MEIMotorTypeInfo structure
count	number of MEINodeIdnData structures to read
***idnData**	a pointer to an array of MEINodeIdnData structures

Return Values

MPIMessageOK	if *NodeIdnDataSet* successfully writes the idn data.

mpiNodeIdnDataSET

Declaration

```
#define mpiNodeIdnDataSET(node, idn)
mpiNodeIdnFieldSet((node), (idn), MPIIdnFieldDATA)
```

Required Header stdmpi.h

Description **NodeIdnDataSET** writes the data fields from an idn object to the idn located on a node.

See Also meiNodeIdnDataGet | meiNodeIdnDataGET

mpiNodeIdnFieldGet / meiNodeIdnFieldGet

mpiNodeIdnFieldGet

Declaration

```
long mpiNodeIdnFieldGet(MPINode     node,
                        MPIIdn      idn,
                        MPIIdnField field)
```

Required Header stdmpi.h

Description **NodeIdnFieldGet** reads the specified ***field*** from an ***idn*** located on a ***node*** and writes it into the idn object.

node	a handle to the Node object
idn	a handle to an Idn object
field	enumeration corresponding to each element in the structure MPIIdnElement

Return Values	
MPIMessageOK	if *NodeIdnFieldGet* successfully reads the specified field.

meiNodeIdnFieldGet

Declaration

```
long meiNodeIdnFieldGet(MPINode     node,
                        MPIIdn      idn,
                        MPIIdnField field)
```

Required Header stdmei.h

Description **NodeIdnFieldGet** reads the specified ***field*** from the ***node*** into the ***idn***.

Return Values	
MPIMessageOK	if *NodeIdnFieldGet* successfully reads the specified field.

See Also mpiNodeIdnFieldSet

mpiNodeIdnFieldSet / meiNodeIdnFieldSet

mpiNodeIdnFieldSet

Declaration

```
long mpiNodeIdnFieldSet(MPIIdn        idn,
                        MPIIdnField   field,
                        void          *value)
```

Required Header stdmpi.h

Description **NodeIdnFieldSet** writes a ***value*** to a specified ***field*** located in the ***idn*** on a node. The node is determined by the association between the idn object and the node object.

idn	a handle to an Idn object
***value**	a pointer to a void

Return Values	
MPIMessageOK	if *NodeIdnFieldSet* successfully writes the specified field.

See Also mpiNodeIdnFieldGet

meiNodeIdnFieldSet

Declaration

```
long meiNodeIdnFieldSet(MPINode     node,
                        MPIIdn      idn,
                        MPIIdnField field)
```

Required Header stdmei.h

Description **NodeIdnFieldSet** writes the specified ***field*** to the ***node*** from the ***idn*** object.

Return Values	
MPIMessageOK	if *NodeIdnFieldSet* successfully writes the specified field.

See Also mpiNodeIdnFieldGet | meiNodeIdnFieldGet

mpiNodeServiceProcedure

Declaration

```
long mpiNodeServiceProcedure(MPINode                    node,
                             MPISercosProcedureAction   action,
                             MPIIdn                     idn,
                             unsigned long              *status)
```

Required Header stdmpi.h

Description **NodeServiceProcedure** executes the procedure in the ***node*** described by ***idn***. The extent of the execution of the procedure is defined by the action parameter (see the MPISercosProcedureAction structure for the different actions).

Return Values	
MPIMessageOK	if sucessful in executing action
MPISercosMessagePROCEDURE_ERROR	if an error occurs in the drive during execution
MPISercosMessagePROCEDURE_DATA_INVALID	if procedure data is invalid
MPISercosMessagePROCEDURE_TIMEOUT	if a timeout occurs and the procedure has not completed.

See Also

mpiNodeStatus

Declaration

```
long mpiNodeStatus(MPINode       node,
                   MPINodeStatus *status,
                   void          *external)
```

Required Header

stdmpi.h

Description

NodeStatus writes a Node's (***node***) status into the structure pointed to by ***status***, and also into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Node's status information in ***external*** is *in addition* to the Node's status information in status, i.e, the status configuration information in ***status*** and in ***external*** is not the same information. Note that ***status*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type MEINodeStatus{} or is NULL.

Return Values

MPIMessageOK	if *NodeStatus* successfully writes the Node's status to the structure(s)

See Also

mpiNodeMemory

Declaration

```
long mpiNodeMemory(MPINode    node,
                   void       **memory)
```

Required Header

stdmpi.h

Description

NodeMemory writes an address [that is used to access Node (***node***) memory] to the contents of ***memory***.

Return Values

MPIMessageOK	if *NodeMemory* successfully writes the Node memory address to the contents of ***memory***

See Also

mpiNodeIdnListGet

Declaration

```
long mpiNodeIdnListGet(MPINode            node,
                       MPINodeIdnListType type,
                       MPIIdnList         idnList)
```

Required Header stdmpi.h

Description **NodeIdnListGet** gets the Node's (***node***) Idn list specified by ***type*** and writes it to ***idnList***. The existing Idn list at ***idnList*** (if any) will be deleted.

Return Values	
MPIMessageOK	if *NodeIdnListGet* successfully gets and writes the Node's Idn list to ***idnList***

See Also mpiNodeIdnListSet

mpiNodeIdnListSet

Declaration

```
long mpiNodeIdnListSet(MPINode            node,
                       MPINodeIdnListType type,
                       MPIIdnList         idnlist)
```

Required Header stdmpi.h

Description **NodeIdnListSet** sets the Node's (***node***) Idn list (specified by ***type***) using data from ***idnList***. The existing Idn list of the Node (if any) will be deleted.

Return Values	
MPIMessageOK	if *NodeIdnListSet* successfully sets the Node's Idn list using data from ***idnList***

See Also mpiNodeIdnListGet

mpiNodeNumber

Declaration

```
long mpiNodeNumber(MPINode node,
                   long    *number)
```

Required Header

stdmpi.h

Description

NodeNumber writes the index of a Node object (***node***, on the motion controller that the Node object is associated with) to the location pointed to by ***number***.

Return Values

MPIMessageOK	if *NodeNumber* successfully writes the Node object's index to the location

See Also

mpiNodeSercos

Declaration `const MPISercos mpiNodeSercos(MPINode node)`

Required Header stdmpi.h

Description **NodeSercos** returns a handle to Sercos object passed to the Node Object in the mpiNodeCreate() function call.

Return Values

MPIMessageOK	if successful

See Also mpiNodeCreate

meiNodeOperationMode

Declaration

```
long meiNodeOperationMode(MPINode node,
                          long    *operationMode)
```

Required Header

stdmei.h

Description

NodeOperationMode returns in ***operationMode*** the sercos defined operation mode based on the MEI-defined operation mode.

Return Values

MPIMessageOK	if successful

See Also

meiNodeRealTimeControlWordGet

Declaration

```
long meiNodeRealTimeControlWordGet(MPINode node,
                                   long    *realTimeControlWord)
```

Required Header

stdmei.h

Description

NodeRealTimeControlWordGet reads the real time control word from a ***node*** and writes it into the location pointed to by ***realTimeControlWord***.

node	a handle to a Node object
***realTimeControlWord**	a pointer to a long

Return Values

MPIMessageOK	if *NodeRealTimeControlWordGet* successfully reads the real time control word.

See Also

mpiNodeRealTimeControlWordSet

meiNodeRealTimeControlWordSet

Declaration

```
long meiNodeRealTimeControlWordSet(MPINode node,
                                   long    realTimeControlWord)
```

Required Header

stdmei.h

Description

NodeRealTimeControlWordSet writes the real time control word to a node.

node	a handle to a Node object
***realTimeControlWord**	a long value

Return Values

MPIMessageOK	if *NodeRealTimeControlWordSet* successfully writes the real time control word.

See Also

mpiNodeRealTimeControlWordGet

meiNodeRealTimeStatusWordGet

Declaration

```
long meiNodeRealTimeStatusWordGet(MPINode node,
                                  long    *realTimeStatusWord)
```

Required Header

stdmei.h

Description

NodeRealTimeStatusWordGet reads the real time status word from a ***node*** and writes it into the location pointed to by ***realTimeStatusWord***.

node	a handle to a Node object
***realTimeStatuslWord**	a point to a long

Return Values

MPIMessageOK	if *NodeRealTimeStatusWordGet* successfully reads the real time status word.

See Also

meiNodeTelegramType

Declaration

```
long meiNodeTelegramType(MPINode node,
                         long    *telegramType)
```

Required Header

stdmei.h

Description

NodeTelegramType reads the telegram type from a ***node*** and writes it into the location pointed to by ***telegramType***.

node	a handle to a Node object
***telegramType**	a pointer to a long

Return Values

MPIMessageOK	if *NodeTelegramType* successfully reads the telegram type.

See Also

MPINodeConfig / MEINodeConfig

MPINodeConfig

```
typedef struct MPINodeConfig {
    long        address;        /* SERCOS address 1 ..254 */
    long        motorNumber;
    long        filterNumber;
    MPINodeMode mode;
} MPINodeConfig;
```

Description

address	In SERCOS each Node has a unique address on the ring(1 - 254).
motorNumber	The number of the MPIMotor associated with the SERCOS Node (1:1).
filterNumber	The number of the Filter associeated with the SERCOS Node (1:1).
mode	The position control algorithm used by the node.

MEINodeConfig

```
typedef struct MEINodeConfig {
    long                Enabled;
    long                StatusInvertMask;
    MEINodeIdnCyclic    At[MEIXmpNodeCyclicCountMAX];
    MEINodeRealTimeBit  RealTimeStatus[MEIXmpNodeRealTimeBitCountMAX];
    MEINodeRealTimeWord RealTimeStatusWord;
    long                ControlInvertMask;
    MEINodeIdnCyclic    Mdt[MEIXmpNodeCyclicCountMAX];
    MEINodeRealTimeBit  RealTimeControl[MEIXmpNodeRealTimeBitCountMAX];
    long                HostRealTimeControlWord;
    MEINodeRealTimeWord RealTimeControlWord;
} MEINodeConfig;
```

Description

NodeConfig is the configuration data that defines the operation of a Node.

Enabled	this variable reports whether or not the Node is enabled.
StatusInvertMask	a bit mask that is used to identify which status bits are inverted.
At	a list of IDNs that make up the AT data for a Node.
RealTimeStatus	in the RealTimeStatusWord there are 2 bits that can be assigned through IDN's to report certain events.
RealTimeStatusWord	located in the AT, this word is a bit mask that is used to report the status of a node.
ControlInvertMask	a bit mask that is used to identify which control bits are inverted.
Md	a list of IDNs that make up the MDT data that is sent to a Node.
RealTimeControl	in the RealTimeControlWord there are 2 bits that can be assigned through IDNs to trigger certain actions or report status from the controller to the Node.

HostRealTimeControlWord	
RealTimeControlWord	located in the MDT, this word is a bit mask that is used to control the node and report controller status.

See Also

mpiNodeConfigGet | mpiNodeConfigSet

MPINodeIdnListType

MPINodeIdnListType

```
typedef enum {
    MPINodeIdnListTypeINVALID,

    MPINodeIdnListTypePHASE2,
    MPINodeIdnListTypePHASE3,
    MPINodeIdnListTypeAMPLIFIER,
    MPINodeIdnListTypeMASTER_DATA,

} MPINodeIdnListType;
```

Description

MPINodeIdnListTypePHASE2	List of IDN's that must be configured in phase 2
MPINodeIdnListTypePHASE3	List of IDN's that must be configured in phase 3
MPINodeIdnListTypeAMPLIFIER	List of IDN's that are transmitted cyclically in the AT
MPINodeIdnListTypeMASTER_DATA	List of IDN's that are transmitted cyclically in the MDT

See Also

MEINodeIdnCyclic

MEINodeIdnCyclic

```
typedef struct MEINodeIdnCyclic {
    long        IDNumber;
    long        UsageFlags;     /* Gain Scheduling, etc… */
    union {
    long        *Src;           /* Host address */
    long        *Dst;           /* Host address */
    } Data;
} MEINodeIdnCyclic;
```

Description

IDNumber	the number of the IDN
UsageFlags	a bit mask that defines how the IDN is used
Data	a pointer to where the data is located

See Also

MEINodeIdnData

MEINodeIdnData

```
typedef struct MEINodeIdnData {
    MPIIdnNumber        idnNumber;
    MPIIdnData          idnData;
} MEINodeIdnData;
```

Description

idnNumber	The number of an IDN.
idnData	The data of an IDN.

See Also

mpiMotionCreate | mpiControlConfigGet

MPINodeMode

MPINodeMode

```
typedef enum {
    MPINodeModeINVALID,

    MPINodeModeNONE,

    /*   servo_command_feedback         */
    MPINodeModeOPENLOOP_TORQUE,
    MPINodeModeOPENLOOP_VELOCITY,
    MPINodeModeOPENLOOP_POSITION_MOTOR,
    MPINodeModeOPENLOOP_POSITION_EXTERNAL,
    MPINodeModeOPENLOOP_POSITION_DUAL,

    MPINodeModePOSITION_TORQUE_MOTOR,
    MPINodeModePOSITION_TORQUE_EXTERNAL,
    MPINodeModePOSITION_VELOCITY_MOTOR,
    MPINodeModePOSITION_VELOCITY_EXTERNAL,

} MPINodeMode;
```

Description

NodeMode defines the different types of control algorithms that are supported under SERCOS and can be used by a node.

See Also

MEINodeRealTimeBit

MEINodeRealTimeBit

```
typedef struct MEINodeRealTimeBit {
    union {
        long    *Src;   /* Host address */
        long    *Dst;   /* Host address */
    } Data;
    long        BitMask;
} MEINodeRealTimeBit;
```

Description

The SERCOS RealTimeBit can be in the **Status Word** or **Control Word**. This is why Data can be either a ***Src*** or ***Dst***. These pointers point to a 32-bit word. Since we are only looking at a bit, ***BitMask*** is used to mask off all the bits that are not of interest.

See Also

For more information, please refer to a SERCOS Specification Manual.

MEINodeRealTimeWord

MEINodeRealTimeWord

```
typedef struct MEINodeRealTimeWord {
    long                InvertMask;
    MEINodeRealTimeBit  Bit[MEIXmpNodeRealTimeWordBitCountMAX];
} MEINodeRealTimeWord;
```

Description

Each Node has a **NodRealTimeWord** for Control and Status, each of which is made up of 16 (MEIXmpNodeRealTimeWordBitCountMAX) bits. The ***InvertMask*** is used to specify the polarity of each bit.

See Also

For more information, please refer to a SERCOS Specification Manual.

mpiNodeIdnDataGET

Declaration

```
#define     mpiNodeIdnDataGET(node, idn)
            mpiNodeIdnFieldGet((node), (idn), MPIIdnFieldDATA)
```

Required Header

stdmpi.h

Description

NodeIdnDataGET gets all of the fields from an ***idn*** located on a ***node***, and writes them into the idn object.

See Also

mpiNodeIdnGET

Declaration

```
#define     mpiNodeIdnGET(node, idn)
            mpiNodeIdnFieldGet((node), (idn), MPIIdnFieldALL)
```

Required Header

stdmpi.h

Description

NodeIdnGET gets all of the fields from an ***idn*** located on a ***node***, and writes them into the idn object.

See Also

mpiNodeIdnSET

Declaration

```
#define     mpiNodeIdnSET(node, idn)
            mpiNodeIdnFieldSet((node), (idn), MPIIdnFieldDATA)
```

Required Header

stdmpi.h

Description

NodeIdnSET sets all of the fields from an ***idn*** located on a ***node***, and writes them into the idn object.

See Also

Notify Objects

Introduction

A thread uses a **Notify** object to wait for event notification. For each thread intended to wait for events from an object (or objects), your application must create a Notify object. The source of firmware events are Motion, Sequence, and Recorder objects.

When it is desired to wait for event notifications from a single source, that source (i.e., object handle) can be passed as the second argument to mpiNotifyCreate(...). After a Notify object is appended to the EventMgr list of Notify objects, make a call to mpiNotifyEventWait(...) to instruct the Notify object to wait for event notification.

Methods

Create, Delete, Validate Methods

mpiNotify**Create**	Create a Notify object
mpiNotify**Delete**	Delete a Notify object
mpiNotify**Validate**	Validate a Notify object

Event Methods

mpiNotify**Event**	Check to see if events have occurred
mpiNotify**EventFlush**	Flush pending events from event queue
mpiNotify**EventMaskGet**	Get event mask
mpiNotify**EventMaskSet**	Set event mask
mpiNotify**EventWait**	Get next event from queue or wait timeout msec for it to arrive
mpiNotify**EventWake**	Wake up thread waiting for notify object

Relational Methods

List Methods for Event Sources

mpiNotify**Source**	Get the indexth event source in list
mpiNotify**SourceAppend**	Append an event source to list
mpiNotify**SourceCount**	Count number of event sources in list
mpiNotify**SourceFirst**	Get first event source in list
mpiNotify**SourceIndex**	Get index value for event source in list
mpiNotify**SourceInsert**	Place event source after source in list
mpiNotify**SourceLast**	Get last event source in list
mpiNotify**SourceListGet**	Get list of event sources
mpiNotify**SourceListSet**	Create a list of event sources
mpiNotify**SourceNext**	Get next event source after source in list
mpiNotify**SourcePrevious**	Get the event source before source in list
mpiNotify**SourceRemove**	Remove event source from list

Data Types

MPINotify**Message**
MEINotify**Trace**

Copyright © 2002
Motion Engineering

mpiNotifyCreate

Declaration

```
const MPINotify mpiNotifyCreate(MPIEventMask mask,
                                void         *source)
```

Required Header

stdmpi.h

Description

NotifyCreate creates a Notify object that will accept event notifications for the events that are specified in ***mask***. The ***source*** argument specifies the initial element in the list of event sources, from which event notification will be accepted. If ***source*** is NULL, then event notification will be accepted from all event sources. *NotifyCreate* is the equivalent of a C++ constructor.

Return Values

handle	to a Notify object
MPIHandleVOID	if the object could not be created

See Also

mpiNotifyDelete | mpiNotifyValidate

mpiNotifyDelete

Declaration `long mpiNotifyDelete(MPINotify notify)`

Required Header stdmpi.h

Description **NotifyDelete** deletes a Notify object and invalidates its handle (***notify***). *NotifyDelete* is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *NotifyDelete* successfully deletes a Notify object and invalidates its handle

See Also mpiNotifyCreate | mpiNotifyValidate

mpiNotifyValidate

Declaration `long mpiNotifyValidate(MPINotify notify)`

Required Header stdmpi.h

Description **NotifyValidate** validates a Notify object and its handle (***notify***).

Return Values	
MPIMessageOK	if Notify is a handle to a valid object.

See Also mpiNotifyCreate | mpiNotifyDelete

mpiNotifyEvent

Declaration

```
long mpiNotifyEvent(MPINotify      notify,
                    MPIEventStatus *status)
```

Required Header

stdmpi.h

Description

NotifyEvent first checks to see if the type field of ***status*** matches a bit in the event mask maintained by a Notify object (***notify***).

IF
the type field of ***status*** matches a bit in the event mask,

AND
the event source list maintained by the Notify object (***notify***) is empty or the ***source*** field of ***status*** matches a source in the event source list,

THEN
a Notify object (notify) will place the event in a FIFO event queue and signal that an event has been received.

If a thread is waiting for event notification (after having called mpiNotifyEventWait(notify)), the signal will awaken it. Otherwise, the next call to mpiNotifyEventWait(notify) will return immediately with ***status***.

Return Values

MPIMessageOK	if the Notify object (***notify***) successfully places the event in a FIFO event queue and signals that an event has been received

See Also

mpiNotifyEventWait

mpiNotifyEventFlush

Declaration `long mpiNotifyEventFlush(MPINotify notify)`

Required Header stdmpi.h

Description **NotifyEventFlush** flushes any pending events from the internal FIFO event queue maintained by a Notify object (***notify***).

Return Values	
MPIMessageOK	if *NotifyEventFlush* successfully flushes the pending events from the internal FIFO event queue maintained by the Notify object

See Also

mpiNotifyEventMaskGet

Declaration

```
long mpiNotifyEventMaskGet(MPINotify     notify,
                           MPIEventMask *mask)
```

Required Header

stdmpi.h

Description

NotifyEventMaskGet writes an event mask (that specifies the event type(s) for which event notification is accepted by a Notify object (***notify***)) to the location pointed to by ***mask***.

Return Values

MPIMessageOK	if *NotifyEventMaskGet* successfully writes the event mask to the location pointed to by mask

See Also

mpiNotifyEventMaskSet

mpiNotifyEventMaskSet

Declaration

```
long mpiNotifyEventMaskSet(MPINotify     notify,
                           MPIEventMask  mask)
```

Required Header

stdmpi.h

Description

NotifyEventMaskSet sets the event type(s) for which notification will be accepted by a Notify object (***notify***), as specified by ***mask***.

Event notification is accepted for event types specified in ***mask***, a bit mask of MPIEventMask bits (associated with the desired MPIEventType values). The MPIEventMask bits must be set or cleared using the MPIEventMask macros. Event notification is denied for event types not specified in ***mask***.

Return Values

MPIMessageOK	if *NotifyEventMaskSet* successfully sets the event type(s) for which notification will be accepted by a Notify object

See Also

MPIEventType | mpiNotifyEventMaskGet

mpiNotifyEventWait

Declaration

```
long mpiNotifyEventWait(MPINotify      notify,
                        MPIEventStatus *status,
                        MPIWait        timeout)
```

Required Header

stdmpi.h

Description

NotifyEventWait sets the contents of the structure pointed to by status, using the status of the first event in the internal FIFO event queue (maintained by a Notify object (notify)), and then removes the first event from the queue. If no event is available in the internal FIFO event queue, NotifyEventWait will wait for timeout milliseconds.

If "timeout" is	*Then*
MPIWaitPOLL (**0**)	*NotifyEventWait* will not wait for an event to arrive
MPIWaitFOREVER (**-1**)	*NotifyEventWait* will wait forever for an event to arrive

Return Values

MPIMessageOK	if *NotifyEventWait* successfully sets the contents of the structure pointed to by status, and then removes the first event from the queue
MPIMessageTIMEOUT	if no event is present and the contents of status are undefined

See Also

mpiNotifyEventWake

mpiNotifyEventWake

Declaration

```
long mpiNotifyEventWake(MPINotify      notify,
                        MPIEventStatus *status)
```

Required Header

stdmpi.h

Description

NotifyEventWake wakes a thread that is waiting for an event notification from a Notify object (***notify***). The awakened thread will return from its call to mpiNotifyEventWait(notify, status, timeout) with the contents of ***status*** set to the contents of status. If status is NULL, ***status*** will indicate an event of type MPIEventTypeNONE.

NotifyEventWake is different from *NotifyEvent*, because event notification is not accepted based on the event type or source (*NotifyEvent*); instead event notification is always accepted (*NotifyEventWake*).

Return Values

MPIMessageOK	if *NotifyEventWake* successfully wakes a thread that is waiting for an event notification from a Notify object

See Also

mpiNotifyEventWait | MPIEventType

mpiNotifySource

Declaration

```
void* mpiNotifySource(MPINotify notify,
                      long      index)
```

Required Header

stdmpi.h

Description

NotifySource returns the element at the position on the list indicated by ***index***.

notify	a handle to the Notify object.
index	a position in the list.

Return Values

***index*th event source**	in the event source list maintained by a Notify object (***notify***)
NULL	if ***notify*** is invalid if ***index*** is less than 0 if ***index*** is greater than or equal to mpiNotifySourceCount(notify)
MPIMessageARG_INVALID	if ***index*** is a negative number.
MEIListMessageELEMENT_NOT_FOUND	if ***index*** is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	if ***notify*** is an invalid handle.

See Also

mpiNotifySourceAppend

Declaration

```
long mpiNotifySourceAppend(MPINotify notify,
                           void      *source)
```

Required Header

stdmpi.h

Description

NotifySourceAppend appends an event source (***source***) to the list of event sources maintained by a Notify object (***notify***).

notify	a handle to the Notify object.
source	a pointer with an arbitrary (but non-NULL) value.

Return Values

MPIMessageOK	if *NotifySourceAppend* successfully appends source to the list of event ***sources*** maintained by a Notify object
MPIMessageHANDLE_INVALID	Either ***source*** is NULL or ***notify*** is an invalid handle.
MPIMessageNO_MEMORY	Not enough memory was available.

See Also

mpiNotifySourceCount

Declaration

```
long mpiNotifySourceCount(MPINotify notify)
```

Required Header

stdmpi.h

Description

NotifySourceCount returns the number of elements on the list.

notify	a handle to the Notify object.

Return Values

number of event sources	in the event source list maintained by a Notify object (***notify***)
-1	if ***notify*** is invalid
0	if the event source list is empty

See Also

mpiNotifySourceFirst

Declaration

```
void* mpiNotifySourceFirst(MPINotify notify)
```

Required Header

stdmpi.h

Description

NotifySourceFirst returns the first element in the list. This function can be used in conjuntion with mpiNotifySourceNext() in order to iterate through the list.

notify	a handle to the Notify object.

Return Values

first event source	in the event source list maintained by a Notify object (***notify***)
NULL	if ***notify*** is invalid or if the event source list is empty
MPIMessageHANDLE_INVALID	if ***notify*** is an invalid handle.

See Also

mpiNotifySourceNext | mpiNotifySourceLast

mpiNotifySourceIndex

Declaration

```
long mpiNotifySourceIndex(MPINotify notify,
                          void      *source)
```

Required Header

stdmpi.h

Description

NotifySourceIndex returns the position of ***source*** on the list.

notify	a handle to the Notify object.
source	a pointer with an arbitrary (but non-NULL) value.

Return Values

index of *source*	in the event source list maintained by a Notify object (***notify***)
-1	if ***notify*** is invalid if the event source (***source***) was not found in the event source list

See Also

mpiNotifySourceInsert

Declaration

```
long mpiNotifySourceInsert(MPINotify notify,
                           void      *source,
                           void      *insert)
```

Required Header

stdmpi.h

Description

NotifySourceInsert places the event source (pointed to by ***insert***) after a specified event source (***source***), in the list of event sources that are maintained by a Notify object (***notify***).

Return Values

MPIMessageOK	if *NotifySourceInsert* successfully places the event source after the specified event source, in the list of event sources that are maintained by a Notify object

See Also

mpiNotifySourceLast

Declaration

```
void* mpiNotifySourceLast(MPINotify notify)
```

Required Header

stdmpi.h

Description

NotifySourceLast returns the last element in the list. This function can be used in conjuntion with mpiNotifySourcePrevious() in order to iterate through the list backwards.

notify	a handle to the Notify object.

Return Values

last event source	in the list maintained by a Notify object (***notify***)
NULL	if ***notify*** is invalid if the event source list is empty

See Also

mpiNotifySourcePrevious | mpiNotifySourceFirst

mpiNotifySourceListGet

Declaration

```
long mpiNotifySourceListGet(MPINotify notify,
                            long      *sourceCount,
                            void      **sourceList)
```

Required Header

stdmpi.h

Description

NotifySourceListGet returns the event source list for a Notify object (***notify***). *NotifySourceListGet* writes the number of event sources in the event source list to the location (pointed to by ***sourceCount***), and also writes an array of ***sourceCount*** event source pointers to the location (pointed to by ***sourceList***).

Return Values

MPIMessageOK	if *NotifySourceListGet* successfully returns the event source list for a Notify object

See Also

mpiNotifySourceListSet | NotifySourceAppend | NotifySourceInsert | mpiNotifyCreate

mpiNotifySourceListSet

Declaration

```
long mpiNotifySourceListSet(MPINotify notify,
                            long      sourceCount,
                            void      **sourceList)
```

Required Header

stdmpi.h

Description

NotifySourceListSet creates an event source list of length ***sourceCount***, using the source pointers specified by ***sourceList***. The ***sourceList*** argument is the address of an array of ***sourceCount*** event source pointers, or is NULL (if ***sourceCount*** = 0). Any existing event source list is completely replaced after using *NotifySourceListSet*.

You can also create an event source list incrementally (i.e., created one source at a time) by using *NotifySourceAppend/Insert* methods. To specify the first event source of a list, use the ***source*** argument of mpiNotifyCreate(...). Use the *NotifySourceList* methods to examine and manipulate an event source list, regardless of how you created it.

Return Values

MPIMessageOK	if *NotifySourceListSet* successfully creates an *event source* list using the source pointers specified by ***sourceList***

See Also

NotifySourceAppend | NotifySourceInsert | mpiNotifyCreate | mpiNotifySourceListGet

mpiNotifySourceNext

Declaration

```
void * mpiNotifySourceNext(MPINotify  notify,
                           void       *source)
```

Required Header

stdmpi.h

Description

NotifySourceNext returns the next element following "source" on the list. This function can be used in conjuntion with mpiNotifySourceFirst() in order to iterate through the list.

notify	a handle to the Notify object.
source	a pointer with an arbitrary (but non-NULL) value.

Return Values

event source	before the event source (***source***) in the event source list maintained by a Notify object (***notify***)
NULL	if ***notify*** is invalid if the event source (***source***) is the first event source in the event source list
MPIMessageHANDLE_INVALID	Either ***source*** is NULL or ***notify*** is an invalid handle.

See Also

mpiNotifySourcePrevious

mpiNotifySourcePrevious

Declaration

```
void * mpiNotifySourcePrevious(MPINotify notify,
                               void      *source)
```

Required Header

stdmpi.h

Description

NotifySourcePrevious returns the previous element prior to "source" on the list. This function can be used in conjuntion with mpiNotifySourceLast() in order to iterate through the list backwards.

notify	a handle to the Notify object.
source	a pointer with an arbitrary (but non-NULL) value.

Return Values

event source	before the event source (***source***) in the event source list maintained by a Notify object (***notify***)
NULL	if ***notify*** is invalid if the event source (***source***) is the first event source in the event source list
MPIMessageHANDLE_INVALID	Either ***source*** is NULL or ***notify*** is an invalid handle.

See Also

mpiNotifySourceNext

mpiNotifySourceRemove

Declaration

```
long mpiNotifySourceRemove(MPINotify notify,
                           void      *source)
```

Required Header

stdmpi.h

Description

NotifySourceRemove removes an event source (***source***) from the list of event sources maintained by a Notify object (***notify***).

Return Values	
MPIMessageOK	if *NotifySourceRemove* successfully removes the event source (***source***) from the list of event sources maintained by a Notify object

See Also

MPINotifyMessage

MPINotifyMessage

```
typedef enum {

    MPINotifyMessageNOTIFY_INVALID,
    MPINotifyMessageWAIT_IN_PROGRESS,
} MPINotifyMessage;
```

Description

MPINotifyMessageNOTIFY_INVALID	
Meaning	The MPINotify handle passed to an MPINotify method is invalid.
Possible Causes	Either the handle was never initialized or the mpiNotifyCreate method failed.
Recommendations	Use mpiNotifyValidate after mpiNotifyCreate to see if the returned handle is valid.

MPINotifyMessageWAIT_IN_PROGRESS	
Meaning	The MPINotify object is already set to wait for an event in another thread.
Possible Causes	
Recommendations	Have each thread use its own set of MPINotify objects.

Sample Code

```
MPIControl    control;
MPINotify     notify;
long          returnValue;
…

notify =
      mpiNotifyCreate(control);
returnValue =
      mpiNotifyValidate(notify);
```

See Also

MPINotify | mpiNotifyCreate | mpiNotifyValidate

MEINotifyTrace

MEINotifyTrace

```
typedef enum {

    MEINotifyTraceTHREAD,
} MEINotifyTrace;
```

Description

MEINotifyTraceTHREAD	will display trace information when notify objects are set to wait, are finished waiting, and when they are signaled to wake.

See Also

Object Objects

Introduction

Each **Object** shares some common functionality with other objects. The Object module encapsulates the common object methods and macros into a single module. This makes object handling consistent and more efficient.

Methods

Create, Delete, Validate Methods

mpiObject**Validate**

Configuration and Information Methods

mpiObject**ModuleId**

mpiObject**TimeoutGet**

mpiObject**TimeoutSet**

meiObject**TraceGet**

meiObject**TraceSet**

Data Types

MPIObject**Map**

Macros

mpiObjectMap**AND_ASSIGN**

mpiObjectMap**ASSIGN**

mpiObjectMap**BitCountMAX**

mpiObjectMap**BitGET**

mpiObjectMap**BitSET**

mpiObjectMap**CLEAR**

mpiObjectMap**COMPLEMENT**

mpiObjectMap**IS_CLEAR**

mpiObjectMap**IS_EQUAL**

mpiObjectMap**IS_VALID**

mpiObjectMap**MAX**

mpiObjectMap**OR_ASSIGN**

meiObject**TraceGET**

meiObject**TraceSET**

Copyright © 2002
Motion Engineering

mpiObjectValidate

Declaration

```
long mpiObjectValidate(MPIHandle   handle,
                       MPIModuleId moduleId)
```

Required Header

stdmpi.h

Description

The **ObjectValidate** method verifies that the object (handle) is of the type moduleId.

Return Values

MPIMessageOK	if ***handle*** is a handle to a valid object of type ***moduleId***.

See Also

mpiObjectModuleId

Declaration

```
long mpiObjectModuleId(MPIHandle     handle,
                       MPIModuleId   *moduleId)
```

Required Header

stdmpi.h

Description

The **ObjectModuleId** function writes the MPIModuleId of the object (***handle***) to the location pointed to by ***moduleId***.

Return Values	
MPIMessageOK	if *ObjectModuleId* successfully writes the MPIModuleId to the location

See Also

mpiObjectTimeoutGet

Declaration

```
long mpiObjectTimeoutGet(MPIHandle handle,
                         MPIWait   *timeout)
```

Required Header

stdmpi.h

Description

The **ObjectTimeoutGet** method gets the timeout value for the object (***handle***), and writes it to the location pointed to by ***timeout***.

Return Values

MPIMessageOK	if *ObjectTimeoutGet* successfully writes the timeout value to the location

See Also

mpiObjectTimeoutSet

mpiObjectTimeoutSet

Declaration

```
long mpiObjectTimeoutSet(MPIHandle handle,
                         MPIWait   timeout)
```

Required Header

stdmpi.h

Description

ObjectTimeoutSet sets the timeout value for an object (***handle***) to ***timeout***.

The ***timeout*** value is used in a multi-threaded environment when an MPI function must block to wait for a shared resource to become available. The default ***timeout*** value is MPIWaitFOREVER.

If "timeout" is	*Then*
MPIWaitFOREVER	*ObjectTimeoutSet* will wait until the resource becomes available
MPIWaitPOLL	*ObjectTimeoutSet* will not wait for the resource to become available. If the shared resource is not available, a timeout will be considered to have occurred.
a value	*ObjectTimeoutSet* will wait timeout milliseconds for the resource to become available

Return Values

MPIMessageOK	if ObjectTimeoutSet successfully sets the timeout value of the object
MPIMessageTIMEOUT	if a timeout occurs before the shared resource becomes available

See Also

mpiObjectTimeoutGet

meiObjectTraceGet / meiObjectTraceGET

meiObjectTraceGet

Declaration

```
long meiObjectTraceGet(MPIHandle     handle,
                       MEITraceMask *traceMask)
```

Required Header stdmei.h

Description **ObjectTraceGet** gets an Object's trace mask and writes it to the value pointed to by traceMask.

handle	a handle to an object
***traceMask**	a pointer to the value of an object's trace mask

Return Values	
MPIMessageOK	if *ObjectTraceGet* successfully gets the trace mask for an object.

meiObjectTraceGET

Declaration

```
#define meiObjectTraceGET(object)      (((MEIObject)(object))->trace)
```

Required Header stdmei.h

Description **ObjectTraceGET** gets the object's global trace mask.

See Also meiObjectTraceSET | meiObjectTraceSet

meiObjectTraceSet / meiObjectTraceSET

meiObjectTraceSet

Declaration

```
long meiObjectTraceSet(MPIHandle     handle,
                       MEITraceMask  traceMask)
```

Required Header stdmei.h

Description **ObjectTraceSet** sets an Object's trace mask to the value specified by ***traceMask***.

handle	a handle to an object
traceMask	the value of an object's trace mask

Return Values

MPIMessageOK	if *ObjectTraceSet* successfully sets the trace mask for an object.

meiObjectTraceSET

Declaration

```
#define  meiObjectTraceSET(object,mask)
                          (((MEIObject)(object))->trace = (mask))
```

Required Header stdmei.h

Description **ObjectTraceSET** sets the object's global trace mask.

See Also meiObjectTraceGET | meiObjectTraceGet

MPIObjectMap

MPIObjectMap

```
typedef unsigned long   MPIObjectMap;
```

Description

MPIObjectMap	A map of MPI objects. An ObjectMap is a bitmap, where each numbered bit represents the presence or absence of the correspondingly numbered object. Valid maps are Axis/Filter and Filter/Motor. The Axis/Filter map can also be read, but setting this map must be done through the corresponding Filter objects.

See Also

mpiObjectMapAND_ASSIGN

Declaration

```
#define   mpiObjectMapAND_ASSIGN(dst,src)       ((dst) &= (src))
```

Required Header

stdmpi.h

Description

Bitwise ANDs the dst object map with the ***src*** object map and assigns the result to ***dst***.

See Also

mpiObjectMapASSIGN

mpiObjectMapASSIGN

Declaration

```
#define    mpiObjectMapASSIGN(dst,src)   ((dst) = (src))
```

Required Header

stdmpi.h

Description

ObjectMapASSIGN assigns ***src*** object map to the ***dst*** object map.

See Also

mpiObjectMapAND_ASSIGN

mpiObjectMapBitCountMAX

Declaration `#define mpiObjectMapBitCountMAX(map) (sizeof(map) * 8)`

Required Header stdmpi.h

Description **ObjectMapBitCountMAX** calculates the maximum bit count for the specified object ***map***.

See Also

mpiObjectMapBitGET

Declaration

```
#define mpiObjectMapBitGET(map,bit)(((map) &  (0x1 << (bit))) ? 1 : 0)
```

Required Header

stdmpi.h

Description

ObjectMapBitGET gets the bit number's state for the specified object ***map***.

See Also

mpiObjectMapBitSET

mpiObjectMapBitSET

Declaration

```
#define  mpiObjectMapBitSET(map,bit,value) \
                                (((value) == 0) \
                                ? ((map) &= ~(0x1 << (bit))) \
                                : ((map) |=  (0x1 << (bit))))
```

Required Header stdmpi.h

Description **ObjectMapBitSET** sets object map's specified ***bit*** number to the specified ***value***.

See Also mpiObjectMapBitGET

mpiObjectMapCLEAR

Declaration `#define mpiObjectMapCLEAR(map) ((map) = 0x0)`

Required Header stdmpi.h

Description **ObjectMapCLEAR** sets the object ***map*** to zero.

See Also

mpiObjectMapCOMPLEMENT

Declaration `#define mpiObjectMapCOMPLEMENT(map) ((map) = ~(map))`

Required Header stdmpi.h

Description **ObjectMapCOMPLEMENT** performs bitwise complement to the object map and assigns the result to ***map***.

See Also

mpiObjectMapIS_CLEAR

Declaration `#define mpiObjectMapIS_CLEAR(map) ((map) == 0x0)`

Required Header stdmpi.h

Description **ObjectMapIS_CLEAR** compares the map to 0x0. If the map is zero, it returns a non-zero value.

See Also

mpiObjectMapIS_EQUAL

Declaration

```
#define   mpiObjectMapIS_EQUAL(map1,map2)        ((map1) == (map2))
```

Required Header

stdmpi.h

Description

ObjectMapIS_EQUAL compares map1 to map2. If the maps are equal, it returns a non-zero value.

See Also

mpiObjectMapIS_VALID

Declaration

```
#define   mpiObjectMapIS_VALID(map,count) \
                (((count) >= (sizeof(map) * 8)) || \
                (((~((0x1 << (count)) - 1)) & (map)) == 0))
```

Required Header stdmpi.h

Description **ObjectMapIS_VALID** checks if the map is within the count range. If the map is valid, it returns a non-zero value.

See Also

mpiObjectMapMAX

Declaration

```
#define mpiObjectMapMAX(map,count)      ((map) = (0x1 << (count)) - 1)
```

Required Header

stdmpi.h

Description

ObjectMapMAX calculates the maximum object ***map*** with the specified ***count***.

See Also

mpiObjectMapOR_ASSIGN

Declaration `#define mpiObjectMapOR_ASSIGN(dst,src) ((dst) |= (src))`

Required Header stdmpi.h

Description Bitwise ORs the dst object map with the ***src*** object map and assigns the result to ***dst***.

See Also mpiObjectMapASSIGN

Path Objects

Introduction

A **Path** object manages coordinated multi-axis motion profiles. It is used when the motion profiles in an N-Dimensional space are required to follow a specific coordinated trajectory. Motion paths are constructed with high level linear and arc segments and downloaded to the controller. The controller calculates the real-time individual axis profiles.

Generally, Path motion is used when the trajectory through space is more important than the final target position. Several different algorithms can be applied to convert the linear and arc segment path into an interpolated trajectory.

Path trajectory generation is now supported by PT, PVT, SPLINE, BESSEL, BSPLINE, and BSPLINE2 algorithms. Blending of the corners is only available for the 2 bspline algorithms. Blending of a corner is when the path does not hit the corner but goes through a smooth arc.

Methods

Create, Delete, Validate Methods

mpiPath**Create**	Create a Path object
mpiPath**Delete**	Delete a Path object
mpiPath**Validate**	Validate a Path object

Configuration and Information Methods

mpiPath**ParamsGet**

mpiPath**ParamsSet**

Relational Methods

mpiPath**Append**

Action Methods

mpiPath**MotionParamsGenerate**

Data Types

MPIPath**Arc**
MPIPath**ArcCenter**
MPIPath**ArcEndPoint**
MPIPath**Attr**
MPIPath**Direction**
MPIPath**Element**
MPIPath**ElementAttrMask**
MPIPath**ElementAttributes**
MPIPath**ElementType**
MPIPath**Line**
MPIPath**Message**
MPIPath**Params**

MPIPath**Point**

Constants

MPIPathPoint**DIMENSION_MAX**

Copyright © 2002
Motion Engineering

mpiPathCreate

Declaration

```
const MPIPath mpiPathCreate();
```

Required Header

stdmpi.h

Description

PathCreate creates a Path object.

Return Values	
handle	to a Path object
MPIHandleVOID	if the object could not be created

See Also

mpiPathDelete | mpiPathValidate

mpiPathDelete

Declaration

```
long mpiPathDelete(MPIPath path);
```

Required Header stdmpi.h

Description **PathDelete** deletes a Path object and invalidates its handle (***path***). PathDelete is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *PathDelete* successfully deletes the Path object and invalidates its handle

See Also mpiPathCreate | mpiPathValidate

mpiPathValidate

Declaration `long mpiPathValidate(MPIPath path);`

Required Header stdmpi.h

Description **PathValidate** validates the Path object and its handle (***command***)

Return Values	
MPIMessageOK	if the Path object and its handle are valid

See Also mpiPathCreate | mpiPathDelete

mpiPathParamsGet

Declaration

```
long mpiPathParamsGet(MPIPath          path,
                      MPIPathParams    *params,
                      void             *external),
```

Required Header

stdmpi.h

Description

PathParamsGet reads the parameters for a path object and writes them into the structure pointed to by ***params***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL)

path	a handle to a Path object
***params**	a pointer to a MPIPathParams structure
***external**	a pointer to a void or NULL

Return Values

MPIMessageOK	if *PathParamsGet* successfully reads the path parameters.

See Also

mpiPathParamsSet

mpiPathParamsSet

Declaration

```
long mpiPathParamsSet(MPIPath        path,
                      MPIPathParams  *params,
                      void           *external),
```

Required Header

stdmpi.h

Description

PathParamsSet writes the parameters from the structure pointed to by ***params*** into the Path object. Also, it writes the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL) into the Path object.

path	a handle to a Path object
***params**	a pointer to a MPIPathParams structure
***external**	a pointer to a void or NULL

Return Values

MPIMessageOK	if *PathParamsSet* successfully writes the path parameters.

See Also

mpiPathParamsGet

mpiPathAppend

Declaration

```
long mpiPathAppend(MPIPath        path,
                   MPIPathElement *element);
```

Required Header stdmpi.h

Description **PathAppend** adds an array of path elements pointed to by ***element*** to the end of the ***path*** stored in the Path object.

Return Values	
MPIMessageOK	if *PathAppend* successfully adds the elements to the path.

See Also mpiPathCreate | mpiPathMotionParamsGenerate

mpiPathMotionParamsGenerate

Declaration

```
long mpiPathMotionParamsGenerate(MPIPath          path,
                                 MPIMotionParams  *params);
```

Required Header stdmpi.h

Description **PathMotionParamsGenerate**

Return Values

See Also

MPIPathArc

MPIPathArc

```
typedef struct MPIPathArc {
        struct {
                double  start;
                double  included;
        } angle;
        double  radius;
} MPIPathArc;
```

Description

PathArc specifies the parameters for an arc path element. It supports 2 dimensional arcs only. All arcs start at the end position for the last path element added to the path or the present command position if the arc is the first element in the path.

start	This value defines the arc's starting angle. Units are in degrees.
included	This value defines the relative travel angle. Units are in degrees. Positive values specify counter-clockwise motion and negative values specify clockwise motion.
radius	This value defines the distance from the center to the arc edge. Units are in counts.

See Also

MPIPathElement | MPIPathParams | MPIPathArcCenter

MPIPathArcCenter

MPIPathArcCenter

```
typedef struct MPIPathArcCenter {
        MPIPathPoint    center;
        double                  angle;
} MPIPathArcCenter;
```

Description

PathArcCenter specifies the parameters for an arc path element. It supports 2 dimensional arcs only. All arcs start at the end position for the last path element added to the path or the present command position if the arc is the first element in the path.

center	This structure defines the coordinates for the center point of the arc. Please see MPIPathPoint data type documentation for more information.
angle	This value defines the relative travel angle. Units are in degrees. Positive values specify counter-clockwise motion and negative values specify clockwise motion.

See Also

MPIPathElement | MPIPathParams | MPIPathArc

MPIPathArcEndPoint

MPIPathArcEndPoint

```
typedef struct MPIPathArcEndPoint {
        MPIPathPoint            center;
        MPIPathPoint            endPoint;
        MPIPathDirection        direction;
} MPIPathArcEndPoint;
```

Description

PathArcEndPoint specifies the parameters for an arc path element. It supports 2 dimensional arcs only. All arcs start at the end position for the last path element added to the path or the present command position if the arc is the first element in the path.

center	This structure defines the coordinates for the center point of the arc. Please see MPIPathPoint data type documentation for more information.
endPoint	This structure defines the coordinates for the final point of the arc. Please see MPIPathPoint data type documentation for more information.
direction	This value defines the travel direction, counter-clockwise or clockwise. Please see MPIPathDirection data type documentation for more information.

See Also

MPIPathElement | MPIPathParams | MPIPathDirection

MPIPathAttr

MPIPathAttr

```
typedef enum {
        MPIPathElementAttrINVALID,

        MPIPathElementAttrRELATIVE,
        MPIPathElementAttrID,
        MPIPathElementAttrVELOCITY,
        MPIPathElementAttrACCEL,
        MPIPathElementAttrTIMESLICE,
        MPIPathElementAttrCOUNT,
} MPIPathAttr;
```

Description

In **PathAttr**, the path attributes are used to generate the path attribute masks to enable features with mpiPathAppend(…). Please see MPIPathElementAttrMask data type for more information.

See Also

mpiPathAppend

MPIPathDirection

MPIPathDirection

```
typedef enum {
        MPIPathDirectionCW,
        MPIPathDirectionCCW,
} MPIPathDirection;
```

Description

MPIPathDirectionCW	This value defines the clockwise direction.
MPIPathDirectionCCW	This value defines the counter-clockwise direction.

See Also

MPIPathArcEndPoint

MPIPathElement

MPIPathElement

```
typedef struct MPIPathElement {
        MPIPathElementType              type;
        long                            blending;
        union {
                MPIPathArc              arc;
                MPIPathArcCenter        arcCenter;
                MPIPathArcEndPoint      arcEndPoint;
                MPIPathHelix            helix;
                MPIPathIo               io;
                MPIPathLine             line;
        } params;

        MPIPathElementAttributes        attributes;
} MPIPathElement;
```

Description

type	This value defines the type of path element. Please see MPIPathElementType{…} data type documentation for more information.
blending	This value determines whether the corners of the path are rounded or sharp. When set to TRUE, blending is enabled, causing rounded corners. When set to FALSE, blending is disabled, causing sharp corners.
arc	This structure defines the arc's start angle, included angle, and radius. This structure is used when the type is MPIPathElementTypeARC. Please see MPIPathArc data type documentation for more information.
arcCenter	This structure defines the arc's center and angle. This structure is used when the type is MPIPathElementTypeARC_CENTER. Please see MPIPathArcCenter data type documentation for more information.
arcEndPoint	This structure defines the arc's center, end point, and direction. This structure is used when the type is MPIPathElementTypeARC_END_POINT. Please see MPIPathArcEndPoint data type documentation for more information.
helix	This structure is not supported and is reserved for future use.
io	This structure is not supported and is reserved for future use.
line	This structure defines the coordinates for a linear element. This structure is used when the type is MPIPathElementTypeLINE. Please see MPIPathLine data type documentation for more information.
attributes	This structure defines the attributes for a path element. Please see MPIPathElementAttributes data type documentation for more information.

See Also

mpiPathAppend

MPIPathElementAttrMask

MPIPathElementAttrMask

```
typedef enum {
        MPIPathElementAttrMaskRELATIVE,    =
mpiPathElementAttrMaskBIT(MPIPathElementAttrRELATIVE),
        MPIPathElementAttrMaskID,          =
mpiPathElementAttrMaskBIT(MPIPathElementAttrID),
        MPIPathElementAttrMaskVELOCITY,    =
mpiPathElementAttrMaskBIT(MPIPathElementAttrVELOCITY),
        MPIPathElementAttrMaskACCEL,       =
mpiPathElementAttrMaskBIT(MPIPathElementAttrACCEL),
        MPIPathElementAttrMaskTIMESLICE), =
mpiPathElementAttrMaskBIT(MPIPathElementAttrTIMESLICE),

        MPIPathElementAttrMaskALL,
} MPIPathElementAttrMask;
```

Description

In **PathElementAttrMask**, the path attribute masks are used to enable features with mpiPathAppend(…). The masks are ORed with the MPIPathElementType to enable each feature.

MPIPathElementAttrMaskRELATIVE	This mask enables relative coordinates for path motion. This feature is not supported and is reserved for future use.
MPIPathElementAttrMaskID	This mask enables an identification tag to be stored in the path. Each element can have a unique identification. Please see MPIPathElementAttributes{…} data type documentation for more information.
MPIPathElementAttrMaskVELOCITY	This mask enables a path velocity to be specified for each element. Please see MPIPathElementAttributes{…} data type documentation for more information.
MPIPathElementAttrMaskACCEL	This mask enables a path acceleration to be specified for each element. Please see MPIPathElementAttributes{…} data type documentation for more information.

See Also

MPIPathElementType | mpiPathAppend

MPIPathElementAttributes

MPIPathElementAttributes

```
typedef struct MPIPathElementAttributes {
        long    id;                                     /* MPIPathAttrID
*/
        double  velocity;                       /* MPIPathAttrVELOCITY          */
        double  acceleration;           /* MPIPathAttrACCELERATION      */
        double  timeSlice;                      /* MPIPathAttrTIMESLICE         */
} MPIPathElementAttributes;
```

Description

In **PathElementAttributes**, the path attributes define the parameters to be used when specific features are enabled with the path element attribute masks. When using these attributes, be sure to enable the feature with the appropriate MPIPathElementAttrMask{…}.

id	This value defines an identification number to be stored in the path element. During path profile execution, at the start of each element the controller loads the id into the axis' ElementID field. The application can query the controller's axis memory to monitor the path element execution.
velocity	This value defines the velocity for the path element.
acceleration	This value defines the acceleration for the path element.
timeSlice	This value defines the time between interpolation points for the path element. The practical range for the time slice is from 10 msec (.01) to 100 msec (.1). Larger time slice values produce smoother (lower acceleration), less accurate paths. Smaller time slice values produce more accurate (both position and velocity) paths with higher peak accelerations.

See Also

MPIPathElementAttrMask

MPIPathElementType

MPIPathElementType

```
typedef enum {
        MPIPathElementTypeINVALID,

        MPIPathElementTypeARC,  /* only 2D */
        MPIPathElementTypeARC_CENTER,  /* only 2D */
        MPIPathElementTypeARC_END_POINT, /* both 2D and 3D */
        MPIPathElementTypeHELIX, /* not currently supported */
        MPIPathElementTypeIO, /* not currently supported */
        MPIPathElementTypeLINE, /* both 2D and 3D */

        MPIPathElementTypeMASK,
} MPIPathElementType;
```

Description

MPIPathElementTypeARC	This type generates an arc specified by the arc's start angle, included angle, and radius.
MPIPathElementTypeARC_CENTER	This type generates an arc specified by the arc's center and angle.
MPIPathElementTypeARC_END_POINT	This type generates an arc specified by the arc's center, end point, and direction.
MPIPathElementTypeLINE	This type generates a line specified by the position coordinates.

See Also

MPIPathArc | MPIPathLine

MPIPathLine

MPIPathLine

```
typedef struct MPIPathLine {
        MPIPathPoint    point;
} MPIPathLine;
```

Description

PathLine specifies the parameters for a linear path element. It supports up to MPIPathPointDIMENSION_MAX dimensions. All lines start at the end position for the last path element added to the path or the present command position if the line is the first element in the path.

point	This structure defines the end point coordinates for the linear segment.

See Also

MPIPathElement | MPIPathParams | MPIPathPointDIMENSION_MAX

MPIPathMessage

MPIPathMessage

```
typedef enum {

        MPIPathMessagePATH_INVALID,
        MPIPathMessageILLEGAL_DIMENSION,
        MPIPathMessageILLEGAL_ELEMENT,
        MPIPathMessageARC_ILLEGAL_DIMENSION,
        MPIPathMessageHELIX_ILLEGAL_DIMENSION,
        MPIPathMessageILLEGAL_RADIUS,
        MPIPathMessagePATH_TOO_LONG,
        MPIPathMessageILLEGAL_VELOCITY,
        MPIPathMessageILLEGAL_ACCELERATION,
        MPIPathMessageILLEGAL_TIMESLICE,
        MPIPathMessageINVALID_BLENDING,
} MPIPathMessage;
```

Description

MPIPathMessagePATH_INVALID	This message code occurs when the specified path cannot be created due to missing or invalid parameters.
MPIPathMessageILLEGAL_DIMENSION	This message code occurs when the specified dimension is out of range. The dimension must be between 1 and MPIPathPointDIMENSION_MAX or a specific dimension if required for specific path types.
MPIPathMessageILLEGAL_ELEMENT	This message code occurs when the specified path cannot be created due to an unsupported sequence of path elements.
MPIPathMessageARC_ILLEGAL_DIMENSION	This message code occurs when the specified dimension is out of range. An arc element dimension must be 2.
MPIPathMessageHELIX_ILLEGAL_DIMENSION	This message code occurs when the specified dimension is out of range. A helix element dimension must be 2.
MPIPathMessageILLEGAL_RADIUS	This message code occurs when the specified radius is less than or equal to zero.
MPIPathMessagePATH_TOO_LONG	This message code occurs when the specified path is longer than the statically allocated buffer can store.
MPIPathMessageILLEGAL_VELOCITY	This message code occurs when the specified velocity is less than or equal to zero.
MPIPathMessageILLEGAL_ACCELERATION	This message code occurs when the specified acceleration is less than or equal to zero.
MPIPathMessageILLEGAL_TIMESLICE	This message code occurs when the specified time slice is less than or equal to zero.

MPIPathMessageINVALID_BLENDING	This message code occurs when blending is specified but not supported with the particular motion type.

See Also

MPIPathParams

MPIPathParams

```
typedef struct MPIPathParams {
        long            dimension;
        MPIPathPoint    start;
        double          velocity;
        double          acceleration;
        double          deceleration;
        MPIMotionType   interpolation;
        double          timeSlice;
        double          conversion
                    [MPIPathPointDIMENSION_MAX][MPIPathPointDIMENSION_MAX];
} MPIPathParams;
```

Description

dimension	This value defines the number of axes to coordinate. Please see MPIPathPoint data type documentation for more information.
start	This structure defines the initial point for the path.
velocity	This value defines the speed along the path. The units are in counts per second.
acceleration	This value defines the rate of change of speed to reach the velocity along the path. The units are in counts per second * second.
deceleration	This value defines the rate of change of speed to reach zero velocity along the path. The units are in counts per second * second.
interpolation	This value specifies the motion algorithm to generate the path. Please see MPIMotionType data type documentation for more information.

See Also

mpiPathParamsGet | mpiPathParmsSet | mpiPathMotionParamsGenerate | MPIPathPointDIMENSION_MAX

MPIPathPoint

MPIPathPoint

```
typedef struct MPIPathPoint {
        double  position[MPIPathPointDIMENSION_MAX];
} MPIPathPoint;
```

Description

position	This array defines the axis command positions for a path point. There must be one position value for each dimension.

See Also

MPIPathParams | mpiPathParamsGet | mpiPathParmsSet | mpiPathPointDIMENSION_MAX

MPIPathPointDIMENSION_MAX

MPIPathPointDIMENSION_MAX

```
#define MPIPathPointDIMENSION_MAX       (16)
```

Description **PathPointDIMENSION_MAX** defines the maximum dimensions for path objects.

See Also MPIPathParams | MPIPathPoint

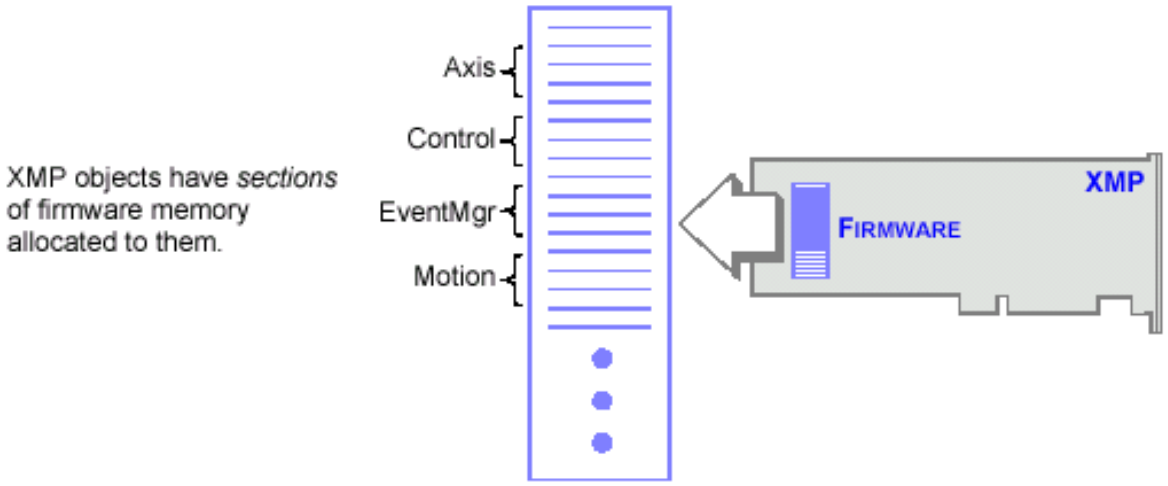
Platform Objects

Introduction

The **Platform** module provides a common interface to platform-specific functionality, such as memory allocation, resource locking, interrupts, signalling, and others.

The Platform object provides low level *platform-specific* functionality and depends upon the combination of the operating system and the C compiler used for development. Although there are circumstances when your application will want to use Platform functions directly, your application won't typically call Platform functions; usually the MPI functions will call Platform functions.

The **meiObjectGive/Take(...)** methods all use the **meiPlatformLockGive/Take(...)** methods. When you take a lock, you take exclusive access to the resource (i.e., the section of XMP firmware memory associated with that Object). When you give a lock, you release (give up) that exclusive access. Think of it as TakeAccessOf and GiveUpAccess.



Methods

meiPlatform**Atol**	Convert a numeric string to a long.
meiPlatform**Key**	Return an input character if an input character is available.
meiPlatform**Trace**	Display printf(...)-style trace information
meiPlatform**TraceEol**	Set the end-of-line (eol) to be used by meiPlatformTrace(...).
meiPlatform**TraceFile**	
meiPlatform**TraceFunction**	

Data Types

MEIPlatform**FileMode**

MEIPlatform**Message**

Copyright © 2002
Motion Engineering

meiPlatformAtol

Declaration `long meiPlatformAtol(const char *ascii)`

Required Header stdmei.h

Description **PlatformAtol** converts a numeric string to a long. This function returns the converted value as a long.

***ascii**	string to be converted

Returns converted the numeric text string ascii to a ***long*** and returned it

See Also

meiPlatformKey

Declaration

```
long meiPlatformKey(MPIWait wait)
```

Required Header

stdmei.h

Description

PlatformKey returns an input character (typically a keystroke) if an input character is available.
If an input charater is not available, *PlatformKey* waits ***wait*** milliseconds for an input character to become available.

If "wait" is	*Then*
MPIWaitFOREVER (-1)	*PlatformKey* will wait for an input character forever
MPIWaitPOLL (0)	*PlatformKey* will return immediately
a value (not -1 or 0)	*PlatformKey* will wait for an input character for ***wait*** milliseconds

Return Values	
-1	if no input character was available
0	*PlatformKey* has read a non-zero character (typically a function key or other non-ASCII value), and **meiPlatformKey(...)** should be called AGAIN immediately to receive that non-zero character
a value (not -1 or 0) (an ASCII character)	(typically a keystroke) if an input character is available

See Also

meiPlatformTrace

Declaration

```
long meiPlatformTrace(const char *format, …)
```

Required Header

stdmei.h

Description

PlatformTrace displays **printf(...)**-style trace information. An *end-of-line* character will be appended to the output, newline by default.

Library modules call **meiTrace#(...)**, a macro which can be conditionally compiled to call **meiPlatformTrace(...)** (by defining the symbol MEI_TRACE when building the library).
Otherwise, calls to **meiTrace#(...)** are removed by the C preprocessor.

Return Values

MPIMessageOK	if *PlatformTrace* successfully executes

See Also

meiPlatformTraceEol

Declaration `char meiPlatformTraceEol(char eol)`

Required Header stdmei.h

Description The **PlatformTraceEol** function sets the *end-of-line* (***eol***) character that will be used by meiPlatformTrace(...).

Returns the previous end-of-line character used by meiPlatformTrace(...)

See Also meiPlatformTrace

meiPlatformTraceFile

Declaration `long meiPlatformTraceFile(const char *fileName)`

Required Header stdmei.h

Description **PlatformTraceFile** redirects trace output to ***fileName***, after first closing any *previously opened* trace file. If no trace file has been explicitly opened, trace output will go to standard output.

Return Values

MPIMessageOK	if *PlatformTraceFile* successfully closes any *previously opened* trace file and redirects trace output to ***fileName***

See Also meiPlatformTrace

meiPlatformTraceFunction

Declaration

```
MEITraceFunction meiPlatformTraceFunction(MEITraceFunction traceFunction)
```

Required Header

stdmei.h

Description

PlatformTraceFunction displays the trace output using ***traceFunction***, and replaces the internal function that was called by meiPlatformTrace(...) to display the trace output. Use *PlatformTraceFunction* to enable your application to take control of the display of trace output.

Return Values

the previous *traceFunction*	if there is a previous function
NULL	if no ***traceFunction*** has been specified (the default trace function is used)

See Also

meiPlatformTrace

MEIPlatformFileMode

MEIPlatformFileMode

```
typedef enum {
    MEIPlatformFileModeREAD,    /* default */
    MEIPlatformFileModeWRITE,
    MEIPlatformFileModeTEXT,    /* default */
    MEIPlatformFileModeBINARY,
} MEIPlatformFileMode;
```

Description

PlatformFileMode is an enumeration that is used as an argument for methods that open files.

See Also

MEIPlatformMessage

MEIPlatformMessage

```
typedef enum {

    MEIPlatformMessagePLATFORM_INVALID,
    MEIPlatformMessageDEVICE_INVALID,
    MEIPlatformMessageDEVICE_ERROR,
    MEIPlatformMessageDEVICE_MAP_ERROR,
} MEIPlatformMessage;
```

Description

MEIPlatformMessageDEVICE_INVALID	Unable to communicate to XMP device driver. (WIN32 Only: If you receive this error, verify that that device driver has been started.)
MEIPlatformMessageDEVICE_ERROR	A low level device driver call has failed. If you receive this error, check for hardware resource conflicts such as base address and IRQ number.
MEIPlatformMessageDEVICE_MAP_ERROR	The device driver is unable to map the XMP memory to host memory system. If you receive this error, check for hardware base address conflict.

See Also

Recorder Objects

Introduction

The **Recorder** object provides a mechanism to collect and buffer data from the controller. Only one Recorder object per controller is supported. After the Recorder is configured and started, the controller copies the data from the specified fields to a local buffer every "*N*" samples. The host can collect the data from the Recorder object using polling or interrupt-based techniques.

| Buffer Size |

Methods

Create, Delete, Validate Methods

mpiRecorder**Create**	Create Recorder object
mpiRecorder**Delete**	Delete Recorder object
mpiRecorder**Validate**	Validate Recorder object

Configuration and Information Methods

mpiRecorder**ConfigGet**	Get Recorder's configuration
mpiRecorder**ConfigSet**	Set Recorder's configuration
mpiRecorder**FlashConfigGet**	Get Recorder's flash configuration
mpiRecorder**FlashConfigSet**	Set Recorder's flash configuration
mpiRecorder**RecordConfig**	Configure type of data record that Recorder will capture
mpiRecorder**Status**	Get status of Recorder

Event Methods

mpiRecorder**EventNotifyGet**	Get event mask of events for which host notification has been requested
mpiRecorder**EventNotifySet**	Set event mask of events for which host notification will be requested
mpiRecorder**EventReset**	Reset the events specified in event mask that are generated by Recorder

Action Methods

mpiRecorder**RecordGet**	Get data records from Recorder
mpiRecorder**Start**	Start recording data records using Recorder
mpiRecorder**Stop**	Stop recording data records using Recorder

Memory Methods

mpiRecorder**Memory**	Get address to Recorder's memory
mpiRecorder**MemoryGet**	Copy data from Recorder memory to application memory
mpiRecorder**MemorySet**	Copy data from application memory to Recorder memory

Relational Methods

mpiRecorder**Control**	Return handle of Control object associated with Recorder

Data Types

MPIRecorder**Config** / MEIRecorder**Config**
MPIRecorder**Message**
MPIRecorder**Record** / MEIRecorder**Record**
MEIRecorder**RecordAxis**
MEIRecorder**RecordFilter**
MPIRecorder**RecordPoint**
MPIRecorder**RecordType** / MEIRecorder**RecordType**
MPIRecorder**Status**
MEIRecorder**Trace**

Constants

MPIRecorder**ADDRESS_COUNT_MAX**

Copyright © 2002
Motion Engineering

mpiRecorderCreate

Declaration `const MPIRecorder mpiRecorderCreate(MPIControl control)`

Required Header stdmpi.h

Description **RecorderCreate** creates a Recorder object associated with a Control object (***control***). *RecorderCreate* is the equivalent of a C++ constructor.

Return Values

handle	to a Recorder object
MPIHandleVOID	if the Recorder object could not be created

See Also mpiRecorderDelete | mpiRecorderValidate

mpiRecorderDelete

Declaration `long mpiRecorderDelete(MPIRecorder recorder)`

Required Header stdmpi.h

Description **RecorderDelete** deletes a Recorder object and invalidates its handle (***recorder***). *RecorderDelete* is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *RecordeDelete* successfully deletes a Recorder object and invalidates its handle

See Also mpiRecorderCreate | mpiRecorderValidate

mpiRecorderValidate

Declaration `long mpiRecorderValidate(MPIRecorder recorder)`

Required Header stdmpi.h

Description **RecorderValidate** validates the Recorder object and its handle (***recorder***).

Return Values

MPIMessageOK	if Recorder is a handle to a valid object.

See Also mpiRecorderCreate | mpiRecorderDelete

mpiRecorderConfigGet

Declaration

```
long mpiRecorderConfigGet(MPIRecorder        recorder,
                          MPIRecorderConfig  *config,
                          void               *external)
```

Required Header

stdmpi.h

Description

RecorderConfigGet gets a Recorder's (***recorder***) configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Recorder's configuration information in ***external*** is in addition to the Recorder's configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only — ***external*** either points to a structure of type **MEIRecorderConfig{}** or is NULL.

Return Values

MPIMessageOK	if *RecorderConfigGet* successfully writes the Recorder's configuration to the structure(s)

See Also

MEIRecorderConfig | mpiRecorderConfigSet

mpiRecorderConfigSet

Declaration

```
long mpiRecorderConfigSet(MPIRecorder        recorder,
                          MPIRecorderConfig  *config,
                          void               *external)
```

Required Header

stdmpi.h

Description

RecorderConfigSet sets a Recorder's (***recorder***) configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Recorder's configuration information in ***external*** is in addition to the Recorder's configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only — ***external*** either points to a structure of type **MEIRecorderConfig{}** or is NULL.

Return Values

MPIMessageOK	if *RecorderConfigSet* successfully sets the Recorder's configuration using data from the structure(s)

See Also

MEIRecorderConfig | mpiRecorderConfigGet

mpiRecorderFlashConfigGet

Declaration

```
long mpiRecorderFlashConfigGet(MPIRecorder        recorder,
                               void               *flash,
                               MPIRecorderConfig  *config,
                               void               *external)
```

Required Header

stdmpi.h

Description

RecorderFlashConfigGet gets the flash configuration (***flash***) for a Recorder (***recorder***) and writes it into the structure pointed to by ***config***, and also writes it in the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Recorder's flash configuration information in ***external*** is *in addition* to the Recorder's flash configuration information in ***config***, i.e, the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only — ***external*** either points to a structure of type **MEIRecorderConfig{}** or is NULL.

Return Values

MPIMessageOK	if *RecorderFlashConfigGet* successfully writes the Recorder's flash configuration to the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIRecorderConfig | MEIFlash | mpiRecorderFlashConfigSet

mpiRecorderFlashConfigSet

Declaration

```
long mpiRecorderFlashConfigSet(MPIRecorder          recorder,
                               void                 *flash,
                               MPIRecorderConfig    *config,
                               void                 *external)
```

Required Header

stdmpi.h

Description

RecorderFlashConfigSet sets the flash configuration (***flash***) for a Recorder (***recorder***) using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Recorder's flash configuration information in ***external*** is in addition to the Recorder's flash configuration information in config, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only

external either points to a structure of type **MEIRecorderConfig{}** or is NULL.

Return Values

MPIMessageOK	if *RecorderFlashConfigSet* successfully sets the Recorder's flash configuration using data from the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIRecorderConfig | MEIFlash | mpiRecorderFlashConfigGet

mpiRecorderRecordConfig

Declaration

```
long mpiRecorderRecordConfig(MPIRecorder           recorder,
                             MPIRecorderRecordType type,
                             long                  count,
                             void                  *handle)
```

Required Header stdmpi.h

Description **RecorderRecordConfig** configures the type (***type***) of record that a Recorder (***recorder***) will capture.

If "type" is	*Then*
MPIRecorderRecordTypePOINT	***count*** data points will be recorded, and ***handle*** points to an array of ***count*** controller addresses
MEIRecorderRecordTypeAXIS	***count*** records of type MPIRecorderRecordAxis{} will be recorded, and ***handle*** points to an array of ***count*** Axis handles
MEIRecorderRecordTypeFILTER	***count*** records of type MPIRecorderRecordFilter{} will be recorded, and ***handle*** points to an array of ***count*** Filter handles

Return Values	
MPIMessageOK	if *RecorderRecordConfig* successfully configures the type of record that the Recorder will capture

See Also MPIRecorderRecordAxis | MPIRecorderRecordFilter

mpiRecorderStatus

Declaration

```
long mpiRecorderStatus(MPIRecorder        recorder,
                       MPIRecorderStatus *status,
                       void              *external)
```

Required Header

stdmpi.h

Description

RecorderStatus gets the status of the Recorder (***recorder***) and writes it into the structure pointed to by ***status***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Recorder's status information in ***external*** is in addition to the Recorder's status information in ***status***, i.e., the status information in ***status*** and in ***external*** is not the same information. Note that ***status*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type MPIRecorderStatus{} or is NULL.

Return Values

MPIMessageOK	if *RecorderStatus* successfully gets the Recorder's status and writes it into the structure(s)

See Also

MPIRecorderStatus

mpiRecorderEventNotifyGet

Declaration

```
long mpiRecorderEventNotifyGet(MPIRecorder  recorder,
                               MPIEventMask *eventMask,
                               void         *external)
```

Required Header

stdmpi.h

Description

RecorderEventNotifyGet writes the event mask into the structure pointed to by **eventMask**, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL). (The event mask specifies the event type(s) generated by a Recorder (***recorder***), for which host notification has been requested.)

The event mask information in ***external*** is in addition to the event mask information in ***eventMask***, i.e, the mask information in ***eventMask*** and in ***external*** is not the same mask information. Note that ***eventMask*** or ***external*** can be NULL (but not both NULL).

XMP Only

external either points to a structure of type MEIEventNotifyData{} or is NULL. An MEIEventNotifyData{} structure is an array of firmware addresses. The contents of these firmware addresses are placed into the MEIEventStatusInfo{} structure (which contains all events generated by this Recorder object).

Return Values

MPIMessageOK	if *RecorderEventNotifyGet* successfully writes the event mask to the structure(s)

See Also

MEIEventNotifyData | MEIEventStatusInfo | mpiRecorderEventNotifySet

mpiRecorderEventNotifySet

Declaration

```
long mpiRecorderEventNotifySet(MPIRecorder   recorder,
                               MPIEventMask  eventMask,
                               void          *external)
```

Required Header

stdmpi.h

Description

RecorderEventNotifySet requests host notification of the event(s) specified by ***eventMask*** and generated by a Recorder (***recorder***), and also generated by the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The events in ***external*** are in addition to the events in ***recorder***, i.e, the events in ***recorder*** and in ***external*** are not necessarily the same events. Note that ***recorder*** or ***external*** can be NULL (but not both NULL).

Event notification is enabled for the event types specified in ***eventMask***. ***eventMask*** is a bit mask generated by the logical OR of the MPIEventMask bits that are associated with the desired MPIEventType values. Event notification is disabled for event types not specified in eventMask.

The mask of event types (generated by a Recorder object) consists of MEIEventMaskRECORDER_FULL and MEIEventMaskRECORDER_DONE.

To	Use "eventMask"
Enable host notification of all Recorder events	MPIEventMaskALL
Disable host notification of all Recorder events	MPIEventTypeNONE

XMP Only

external either points to a structure of type MEIEventNotifyData{} or is NULL. An MEIEventNotifyData{} structure is an array of firmware addresses. The contents of these firmware addresses are placed into the MEIEventStatusInfo{} structure (which contains all events generated by this Recorder object).

Return Values

MPIMessageOK	if *RecorderEventNotifySet* successfully requests host notification of the event(s) as specified by the structure(s)

See Also

MEIEventMaskRECORDER | MEIEventNotifyData | MEIEventStatusInfo
mpiRecorderEventNotifyGet

mpiRecorderEventReset

Declaration

```
long mpiRecorderEventReset(MPIRecorder  recorder,
                           MPIEventMask eventMask)
```

Required Header

stdmpi.h

Description

RecorderEventReset resets the event(s) specified in ***eventMask*** and generated by a Recorder (***recorder***). Your application should call *RecorderEventReset* only after one or more latchable events have occurred.

Return Values

MPIMessageOK	if *RecorderEventReset* successfully resets the event(s) that are specified in ***eventMask*** and generated by a Recorder

See Also

mpiRecorderRecordGet

Declaration

```
long mpiRecorderRecordGet(MPIRecorder        recorder,
                          long               countMax,
                          MPIRecorderRecord  *record,
                          long               *count)
```

Required Header

stdmpi.h

Description

RecorderRecordGet obtains a Recorder's (***recorder***) data records. The record type must have been configured previously, by a prior call to mpiRecorderRecordConfig(...).

RecorderRecordGet gets a maximum of ***countMax*** records and writes them into the location pointed to by ***record*** (the location must be large enough to hold them). RecorderRecordGet also writes the actual number of records that were obtained to the location pointed to by ***count***.

If the recorder data buffer is full and recording is enabled, recording will be temporarily disabled while either all or ***countMax*** records are obtained, whichever is less. Any records not obtained will be lost.

Return Values

MPIMessageOK	if *RecorderRecordGet* successfully gets the data records

See Also

mpiRecorderRecordConfig

mpiRecorderStart

Declaration

```
long mpiRecorderStart(MPIRecorder  recorder,
                      long         count,
                      long         period,
                      long         fullCount)
```

Required Header

stdmpi.h

Description

RecorderStart instructs a Recorder (***recorder***) to begin recording data records. The record type must have been configured previously, by a prior call to mpiRecorderRecordConfig(...).

An event of type MPIEventRECORDER_DONE will be generated when the Recorder has finished collecting data records.

count (for ***count*** > 0)	specifies the total number of data records to record
if ***count*** = -1	then recording is continuous, and uncollected records may be overwritten
period if ***period*** = 0	frequency of recording in milliseconds then data will be recorded as quickly as possible (on every sample)
if ***fullCount*** > 0	then an event of type MPIEventRECORDER_COUNT will be generated when the Recorder's data buffer contains ***fullCount*** data records

Return Values

MPIMessageOK	if *RecorderStart* successfully instructs a Recorder to begin recording data records

See Also

mpiRecorderRecordConfig | mpiRecorderStop

mpiRecorderStop

Declaration `long mpiRecorderStop(MPIRecorder recorder)`

Required Header stdmpi.h

Description **RecorderStop** instructs a Recorder (***recorder***) to stop recording data records.

Return Values	
MPIMessageOK	if *RecorderStop* successfully stops recording data records

See Also mpiRecorderStart

mpiRecorderMemory

Declaration

```
long mpiRecorderMemory(MPIRecorder recorder,
                       void        **memory)
```

Required Header

stdmpi.h

Description

RecorderMemory writes an address to the contents of ***memory***. An address can be used to access a Recorder's (***recorder***) memory. An address calculated from it can be passed as the ***src*** argument to mpiRecorderMemoryGet(...) and as the ***dst*** argument to mpiRecorderMemorySet(...).

Return Values

MPIMessageOK	if *RecorderMemory* successfully writes the Recorder's memory address to the contents of ***memory***

See Also

mpiRecorderMemoryGet | mpiRecorderMemorySet

mpiRecorderMemoryGet

Declaration

```
long mpiRecorderMemoryGet(MPIRecorder recorder,
                          void        *dst,
                          void        *src,
                          long        count)
```

Required Header stdmpi.h

Description **RecorderMemoryGet** copies ***count*** bytes of a Recorder's (***recorder***) memory (starting at address ***src***) to application memory (starting at address ***dst***).

Return Values	
MPIMessageOK	if *RecorderMemoryGet* successfully copies data from Recorder memory to application memory

See Also mpiRecorderMemory | mpiRecorderMemorySet

mpiRecorderMemorySet

Declaration

```
long mpiRecorderMemorySet(MPIRecorder recorder,
                          void        *dst,
                          void        *src,
                          long        count)
```

Required Header

stdmpi.h

Description

RecorderMemorySet copies ***count*** bytes of application memory (starting at address ***src***) to a Recorder's (***recorder***) memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *RecorderMemorySet* successfully copies data from application memory to Recorder memory

See Also

mpiRecorderMemory | mpiRecorderMemoryGet

mpiRecorderControl

Declaration

```
const MPIControl mpiRecorderControl(MPIRecorder recorder)
```

Required Header

stdmpi.h

Description

RecorderControl returns a handle to the motion controller (Control object) that a Recorder (***recorder***) is associated with.

Return Values

handle	to a Control object that a Recorder is associated with
MPIHandleVOID	if the Recorder object is invalid

See Also

MPIRecorderConfig / MEIRecorderConfig

MPIRecorderConfig

```
typedef struct MPIRecorderConfig {
    long        period;         /* collect 1 record every `period` milliseconds */
    long        fullCount;      /* >0 => record count to trigger full buffer */

    long        addressCount;   /* number of data point addresses in address[] */
    void        *address[MPIRecorderADDRESS_COUNT_MAX];
} MPIRecorderConfig;
```

Description

period	The number of milliseconds between successive recordings. A value of zero means the recorder will record data every sample.
fullCount	If fullCount>0, then the recorder will generate a "recorder full" event when this fullCount number or records are in the recorder buffer.
addressCount	The number of XMP memory addresses in address[].
***address[]**	An array of XMP memory addresses the recorder will record.

MEIRecorderConfig

```
typedef MPIEmpty        MEIRecorderConfig;
```

Description

RecorderConfig is currently not supported and is reserved for future use.

See Also

mpiRecorderConfigGet | mpiRecorderConfigSet

MPIRecorderMessage

MPIRecorderMessage

```
typedef enum {

    MPIRecorderMessageRECORDER_INVALID,
    MPIRecorderMessageSTARTED,
    MPIRecorderMessageSTOPPED,
    MPIRecorderMessageNOT_CONFIGURED,
} MPIRecorderMessage;
```

Description

MPIRecorderMessageRECORDER_INVALID	
Meaning	The MPIRecorder handle passed to an MPIRecorder method is invalid.
Possible Causes	Either the handle was never initialized or the MPIRecorderCreate method failed.
Recommendations	Use MPIRecorderValidate after MPIRecorderCreate to see if the returned handle is valid.

MPIRecorderMessageSTARTED	
Meaning	
Possible Causes	An MPIRecorder method that assumes that recorder has not been started has been called while the recorder was recording data.
Recommendations	

MPIRecorderMessageSTOPPED	
Meaning	
Possible Causes	An MPIRecorder method that assumes that recorder has been started has been called while the recorder was not recording data.
Recommendations	

MPIRecorderMessageNOT_CONFIGURED	
Meaning	
Possible Causes	An MPIRecorder method has been called that assumes that the recorder has been configured. mpiRecorderStart is the most common method to return this error code.
Recommendations	

Sample Code

```
MPIControl     control;
MPIRecorder    recorder;
long           returnValue;
…

recorder =
      mpRecorderCreate(control);
returnValue =
      mpiRecorderValidate(recorder);
```

See Also

mpiRecorderCreate | mpiRecorderValidate

MPIRecorderRecord / MEIRecorderRecord

MPIRecorderRecord

```
typedef union {
    MPIRecorderRecordPoint      point[MPIRecorderADDRESS_COUNT_MAX];
} MPIRecorderRecord;
```

Description

point	An array of recorded values corresponding to the XMP addresses stored in MPIRecorderConfig.address[].

MEIRecorderRecord

```
typedef union {
    MEIRecorderRecordAxis       axis[MEIXmpMAX_Axes];
    MEIRecorderRecordFilter     filter[MEIXmpMAX_Filters];
    MPIRecorderRecord           dummy;  /* ensure proper sizing */
} MEIRecorderRecord;
```

Description

axis	An array of MEIRecorderRecordAxis records.
filter	An array of MEIRecorderRecordFilter records.
dummy	A dummy structure that ensures that MEIRecorderRecord has the proper size.

See Also

MEIRecorderRecordAxis

MEIRecorderRecordAxis

```
typedef struct MEIRecorderRecordAxis {
    long    sample;         /* sample number */
    long    command;        /* command position */
    long    actual;         /* actual position */
    float   dac;            /* voltage */
} MEIRecorderRecordAxis;
```

Description

sample	The XMP sample number in which the following values were recorded.
command	The command position of the axis.
actual	The actual position of the axis.
dac	The output of the primary DAC of the motor associated with the axis.

See Also

MEIRecorderRecordFilter

MEIRecorderRecordFilter

```
typedef struct MEIRecorderRecordFilter {
    long        sample;         /* sample number */
    long        command;        /* command position */
    long        actual;         /* actual position */
    float       dac;            /* voltage */
} MEIRecorderRecordFilter;
```

Description

sample	The XMP sample number in which the following values were recorded
command	The command position the filter uses to calculate the filter output.
actual	The actual position (of an axis) the filter uses to calculate the filter output.
dac	The output of the filter that gets sent to a motor's primary DAC.

See Also

MPIRecorderRecordPoint

MPIRecorderRecordPoint

```
typedef long MPIRecorderRecordPoint;
```

Description

MPIRecorderRecordPoint	represents one recorder record. This will correspond to the value of one XMP address.

See Also

MPIRecorderRecordType / MEIRecorderRecordType

MPIRecorderRecordType

```
typedef enum {
    MPIRecorderRecordTypeINVALID,
    MPIRecorderRecordTypePOINT,

} MPIRecorderRecordType;
```

Description

MPIRecorderRecordTypeINVALID	specifies to the data recorder that MPIRecorderRecordPoint records (copies of controller memory locations) are being recorded.
MPIRecorderRecordTypePOINT	an invalid record type.

MEIRecorderRecordType

```
typedef enum {
    MEIRecorderRecordTypeAXIS,
    MEIRecorderRecordTypeFILTER,
} MPIRecorderRecordType;
```

Description

MEIRecorderRecordTypeAXIS	specifies to the data recorder that MEIRecorderRecordAxis records are being recorded.
MEIRecorderRecordTypeFILTER	specifies to the data recorder that MEIRecorderRecordFilter records are being recorded.

Remarks

Predefined types for setting up the type of data an MPIRecorder object will record. This is used by the mpiRecorderRecordConfig() method.

See Also

MPIRecorder | MEIRecorderRecordAxis | MEIRecorderRecordFilter | mpiRecorderRecordConfig

MPIRecorderStatus

MPIRecorderStatus

```
typedef struct MPIRecorderStatus {
    long    enabled;
    long    full;
    long    recordCount;
    long    recordCountMax;
} MPIRecorderStatus;
```

Description

enabled	If the recorder is enabled (recording) then enabled will equal TRUE, otherwise enabled will equal FALSE.
full	If the recorder is full (the number of stored records >= MPIRecorderConfig.fullCount) then full will equal TRUE, otherwise full will equal FALSE.
recordCount	The number of stored records in the recorder.
recordCountMax	The maximum number of records the recorder can store.

See Also

mpiRecorderStatus

MEIRecorderRecordTrace

MEIRecorderRecordTrace

```
typedef enum {

    MEIRecorderTraceRECORD_GET,
    MEIRecorderTraceSTATUS,
    MEIRecorderTraceOVERFLOW,
} MEIRecorderTrace;
```

Description

MEIRecorderTraceRECORD_GET	will display trace information when the data recorder retrieves records.
MEIRecorderTraceSTATUS	will display trace information when the MPI retrieves the data recorder status.
MEIRecorderTraceOVERFLOW	will display trace information when the data recorder overflows.

See Also

MPIRecorderADDRESS_COUNT_MAX

MPIRecorderADDRESS_COUNT_MAX

```
#define MPIRecorderADDRESS_COUNT_MAX    (128)
```

Description

RecorderADDRESS_COUNT_MAX defines the maximum number of addresses the Recorder object supports.

See Also

MPIRecorderConfig

Recorder Buffer Size

The Data Recorder buffer size can be dynamically allocated. The MPIControlConfig{...} structure has a new element, called recordCount. This element allows the application to change the size of the recorder object's data buffer using the mpiControlConfigGet/Set(...) methods. The Record buffer size (the default is 3064 records) is defined within the MEIXmpDefaultEnabled_Records structure (*xmp.h*). Each record is the size of one memory word. Using a larger data buffer size can improve the performance of MotionScope running on a slow host or running in Client/Server mode over a congested network.

A new method, meiControlExtMemAvail(...), has been added which will return the size of external memory available for allocation. This value can be added to the current recordCount to expand the record buffer to the maximum possible size.

For more information, see the Special Note on *Dynamic Allocation of External Memory Buffer*s.

Return to Recorder Object's page

Copyright © 2002
Motion Engineering

Sequence Objects

Introduction

A **Sequence** object manages a set of Commands. The sequence is constructed on the host from a list of commands, then downloaded and executed in the controller. Typically, applications only use Sequences for very small or simple autonomous tasks that require execution in the controller. Due to their embedded execution, debugging can be difficult. It is best to use the host application to execute MPI methods directly for optimum flexibility and performance.

If you are considering using a program Sequencer or Command objects, please contact an MEI Applications Engineer. We recommend that you do **NOT** implement complex Sequences on your own.

Methods

Create, Delete, Validate Methods

mpiSequence**Create**	Create Sequence object
mpiSequence**Delete**	Delete Sequence object
mpiSequence**Validate**	Validate Sequence object

Configuration and Information Methods

mpiSequence**ConfigGet**	Get sequence config
mpiSequence**ConfigSet**	Set sequence config
mpiSequence**FlashConfigGet**	Get sequence flash config
mpiSequence**FlashConfigSet**	Set sequence flash config
mpiSequence**PageSize**	Set pageSize to number of command slots used by sequence
mpiSequence**Status**	Return sequence status

Event Methods

mpiSequence**EventNotifyGet**	Select an event mask for host notification of events
mpiSequence**EventNotifySet**	Enable host notification of sequence events
mpiSequence**EventReset**	Reset sequence events

Action Methods

meiSequence**Compile**	
mpiSequence**Load**	Load sequence commands into firmware
mpiSequence**Resume**	Resume execution of sequence
mpiSequence**Start**	Start execution of sequence
mpiSequence**Step**	Execute count steps of a stopped sequence
mpiSequence**Stop**	Stop sequence

Memory Methods

mpiSequence**Memory**	Set address used to access sequence memory
mpiSequence**MemoryGet**	Get bytes of sequence memory and put into application memory
mpiSequence**MemorySet**	Put (set) bytes of application memory into sequence memory

Relational Methods

mpiSequence**Control**	Get handle to Control
mpiSequence**Number**	Get index number of sequence

List Methods for Event Sources

mpiSequence**Command**	Return handle to indexed command of sequence
mpiSequence**CommandAppend**	Append command to sequence
mpiSequence**CommandCount**	Count the number of commands in sequence
mpiSequence**CommandFirst**	Return handle to first command in sequence
mpiSequence**CommandIndex**	Return the index of a command in sequence
mpiSequence**CommandInsert**	Insert command into sequence
mpiSequence**CommandLast**	Return handle of last command in sequence
mpiSequence**CommandListGet**	Get list of commands in sequence
mpiSequence**CommandListSet**	Set list of commands in sequence
mpiSequence**CommandNext**	Get handle to next command in list
mpiSequence**CommandPrevious**	Get handle to previous command in list
mpiSequence**CommandRemove**	Remove command from list

Data Types

MPISequence**Config** / MEISequence**Config**
MPISequence**Message**
MPISequence**State**
MPISequence**Status**
MEISequence**Trace**

Copyright © 2002
Motion Engineering

mpiSequenceCreate

Declaration

```
const MPISequence mpiSequenceCreate(MPIControl control,
                                    long       number,
                                    long       pageSize)
```

Required Header

stdmpi.h

Description

SequenceCreate creates a Sequence object associated with the program sequencer identified by ***number*** located on motion controller (control). SequenceCreate is the equivalent of a C++ constructor.

If	*Then*
number is **-1**	*SequenceCreate* selects the next unused program sequencer. If this is the first use of the program sequencer, then SequenceCreate will attempt to allocate pageSize firmware command slots.
pageSize is **-1**	*SequenceCreate* will allocate all remaining firmware command slots, which may prevent any more Sequence objects from being created.

Return Values

handle	to a Sequence object
MPIHandleVOID	if the object could not be created

See Also

mpiSequenceDelete | mpiSequenceValidate

mpiSequenceDelete

Declaration `long mpiSequenceDelete(MPISequence sequence)`

Required Header stdmpi.h

Description **SequenceDelete** deletes a Sequence object and invalidates its handle (***sequence***). *SequenceDelete* is the equivalent of a C++ destructor.

All Command objects in a Sequence are deleted when the Sequence object is deleted.

Return Values

MPIMessageOK	if *SequenceDelete* successfully a Sequence object and invalidates its handle

See Also mpiSequenceCreate | mpiSequenceValidate

mpiSequenceValidate

Declaration `long mpiSequenceValidate(MPISequence sequence)`

Required Header stdmpi.h

Description **SequenceValidate** validates the Sequence object and its handle (***sequence***).

Return Values	
MPIMessageOK	if Sequence is a handle to a valid object.

See Also mpiSequenceCreate | mpiSequenceDelete

mpiSequenceConfigGet

Declaration

```
long mpiSequenceConfigGet(MPISequence sequence,
                          MPISequenceConfig *config,
                          void *external)
```

Required Header

stdmpi.h

Description

SequenceConfigGet gets the configuration of a Sequence object (***sequence***) and writes it in the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Sequence's configuration information in ***external*** is in addition to the Sequence's configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type MEISequenceConfig{} or is NULL.

Return Values

MPIMessageOK	if *SequenceConfigGet* successfully gets and writes the configuration of a Sequence object into the structure(s)

See Also

mpiSequenceConfigSet | MEISequenceConfig

mpiSequenceConfigSet

Declaration

```
long mpiSequenceConfigSet(MPISequence sequence,
                          MPISequenceConfig *config,
                          void *external)
```

Required Header

stdmpi.h

Description

SequenceConfigSet sets the configuration of a Sequence (***sequence***) using data from the structure pointed to by ***config***, and also using data from the implementation- specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Sequence's configuration information in ***external*** is in addition to the Sequence's configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type MEISequenceConfig{} or is NULL.

Return Values

MPIMessageOK	if *SequenceConfigSet* successfully sets a Sequence's configuration using data from the structure(s)

See Also

mpiSequenceConfigGet | MEISequenceConfig

mpiSequenceFlashConfigGet

Declaration

```
long mpiSequenceFlashConfigGet(MPISequence          sequence,
                               void                 *flash,
                               MPISequenceConfig    *config,
                               void                 *external)
```

Required Header

stdmpi.h

Description

SequenceFlashConfigGet gets a Sequence's (***sequence***) flash configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Sequence's flash configuration information in ***external*** is in addition to the Sequence's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL). The implementation-specific ***flash*** argument is used to access flash memory.

XMP Only

external either points to a structure of type MEISequenceConfig{} or is NULL. ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *SequenceFlashConfigGet* successfully writes the Sequence's flash configuration to the structure(s)

See Also

mpiSequenceFlashConfigSet

mpiSequenceFlashConfigSet

Declaration

```
long mpiSequenceFlashConfigSet(MPISequence          sequence,
                               void                 *flash,
                               MPISequenceConfig    *config,
                               void                 *external)
```

Required Header

stdmpi.h

Description

SequenceFlashConfigSet sets a Sequence's (***sequence***) flash configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Sequence's flash configuration information in ***external*** is in addition to the Sequence's flash configuration information in ***config***, i.e., the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL). The implementation-specific ***flash*** argument is used to access flash memory.

XMP Only

external either points to a structure of type MEISequenceConfig{} or is NULL. ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if *SequenceFlashConfigSet* successfully sets the Sequence's flash configuration using data from the structure(s)

See Also

MEISequenceConfig | mpiSequenceFlashConfigGet

mpiSequencePageSize

Declaration

```
long mpiSequencePageSize(MPISequence sequence,
                         long        *pageSize)
```

Required Header

stdmpi.h

Description

SequencePageSize writes the *number* of command slots that are available to a Sequence (***sequence***, on its associated motion controller) to the contents of ***pageSize***.

Return Values

MPIMessageOK	if *SequencePageSize* successfully writes the number of command slots (available to the Sequence) to the contents of ***pageSize***

See Also

mpiSequenceStatus

Declaration

```
long mpiSequenceStatus(MPISequence          sequence,
                       MPISequenceStatus    *status,
                       void                 *external)
```

Required Header

stdmpi.h

Description

SequenceStatus returns the status of a Sequence (***sequence***), and writes it into the structure pointed to by ***status***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The status information in ***external*** is *in addition* to the status information in ***status***, i.e, the status information in ***status*** and in ***external*** is not the same information. Note that ***status*** or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type MEISequenceStatus{} or is NULL.

Return Values

MPIMessageOK	if *SequenceStatus* successfully returns the Sequence's status and writes the status to the structure(s)

See Also

MPISequenceStatus

mpiSequenceEventNotifyGet

Declaration

```
long mpiSequenceEventNotifyGet(MPISequence   sequence,
                               MPIEventMask  *eventMask,
                               void          *external)
```

Required Header

stdmpi.h

Description

SequenceEventNotifyGet writes an event mask [that specifies the event types (generated by the Sequence ***sequence***, for which host notification has been requested] to the structure pointed to by ***eventMask***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The event mask information in ***external*** is *in addition* to the event mask information in ***eventMask***, i.e, the event mask information in ***eventMask*** and in ***external*** is not the same information. Note that ***eventMask*** or ***external*** can be NULL (but not both NULL).

XMP Only

external either points to a structure of type **MEIEventMask{}** or is NULL.

Return Values

MPIMessageOK	if *SequenceEventNotifyGet* successfully writes the event mask to the structure(s)

See Also

MEIEventMask | mpiSequenceEventNotifySet

mpiSequenceEventNotifySet

Declaration

```
long mpiSequenceEventNotifySet(MPISequence   sequence,
                               MPIEventMask  eventMask,
                               void          *external)
```

Required Header

stdmpi.h

Description

SequenceEventNotifySet requests host notification of the event(s) specified by ***eventMask*** and generated by a Sequence (***sequence***), and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The event mask information in ***external*** is in addition to the event mask information in ***eventMask***, i.e, the event mask information in ***eventMask*** and in ***external*** is not the same information. Note that ***eventMask*** or external can be NULL (but not both NULL).

The mask of event types generated by a Sequence object consists of MPIEventMaskEXTERNAL. When a Sequence issues a Command of type MPICommandTypeEVENT, an event of type MPIEventTypeEXTERNAL is generated. The only event generated by a Sequence is MPIEventTypeEXTERNAL, which is generated when a Sequence issues a Command of type MPICommandTypeEVENT.

To	Use "eventMask"
Disable host notification of all Sequence events	MPIEventTypeNONE
Enable host notification of all Sequence events	MPIEventMaskALL

XMP Only ***external*** either points to a structure of type MEIEventMask{} or is NULL.

Return Values

MPIMessageOK	if *SequenceEventNotifySet* successfully requests host notification of the events in the event mask(s)

See Also

MPIEventMaskEXTERNAL | MEIEventMask | mpiSequenceEventNotifyGet

mpiSequenceEventReset

Declaration

```
long mpiSequenceEventReset(MPISequence  sequence,
                           MPIEventMask eventMask)
```

Required Header

stdmpi.h

Description

SequenceEventReset resets the event(s) that are specified in ***eventMask*** and generated by a Sequence (***sequence***). Your application should not call SequenceEventReset *until* one or more latchable events have occurred.

Return Values

MPIMessageOK	if *SequenceEventReset* successfully resets the event(s) that are specified in ***eventMask*** and generated by a Sequence object

See Also

meiSequenceCompile

Declaration

```
long meiSequenceCompile(MPISequence sequence)
```

Required Header

stdmpi.h

Description

SequenceCompile "compiles" a ***sequence*** object by reading its list of Command objects and then creating an equivalent list of XMP commands.

Return Values

MPIMessageOK	if *SequenceCompile* successfully reads a Sequence object's list of Command objects and creates an equivalent list of XMP commands

See Also

mpiSequenceLoad

Declaration

```
long mpiSequenceLoad(MPISequence sequence,
                     MPICommand  command,
                     long        start)
```

Required Header

stdmpi.h

Description

SequenceLoad loads the firmware command slots of a Sequence (***sequence***) as necessary, starting with the Command (***command***).

SequenceLoad is intended to be called initially by mpiSequenceStart(...) and called thereafter by mpiEventMgrService(...) (in response to reception of an *internal page fault event notification* from the firmware). Except when you are debugging a sequence via mpiSequenceStep(...), your application should never need to directly call SequenceLoad.

If	*Then*
command is MPIHandleVOID	*SequenceLoad* loads Commands starting with the first Command of the Sequence
start is not FALSE	*SequenceLoad* starts the sequence after the commands are loaded

Return Values

MPIMessageOK	if *SequenceLoad* successfully loads the firmware command slots of a Sequence

See Also

mpiSequenceStart | mpiEventMgrService | mpiSequenceStep

mpiSequenceResume

Declaration `long mpiSequenceResume(MPISequence sequence)`

Required Header stdmpi.h

Description **SequenceResume** resumes a Sequence (***sequence***) from the point where the Sequence has stopped (if execution has been stopped).

Return Values	
MPIMessageOK	if *SequenceResume* successfully resumes a Sequence from the point where the Sequence has stopped

See Also

mpiSequenceStart

Declaration

```
long mpiSequenceStart(MPISequence sequence,
                      MPICommand  command)
```

Required Header

stdmpi.h

Description

SequenceStart begins the execution of a Sequence (***sequence***), starting with the Command (***command***). If ***command*** is MPIHandleVOID, execution starts with the first command of the Sequence.

Return Values

MPIMessageOK	if *SequenceStart* successfully begins the execution of a Command Sequence

See Also

mpiSequenceStop

mpiSequenceStep

Declaration

```
long mpiSequenceStep(MPISequence sequence,
                     long        count)
```

Required Header

stdmpi.h

Description

SequenceStep executes ***count*** steps (Commands) of a stopped Sequence (***sequence***). After executing the Commands, the Sequence will be in the MPISequenceStateSTOPPED state.

Return Values

MPIMessageOK	if *SequenceStep* successfully executes ***count*** steps (Commands) of a stopped Sequence

See Also

mpiSequenceStop

Declaration `long mpiSequenceStop(MPISequence sequence)`

Required Header stdmpi.h

Description **SequenceStop** stops a Sequence (***sequence***), if execution has been started. A stopped Sequence can be resumed from the point where it has stopped.

Return Values	
MPIMessageOK	if *SequenceStop* successfully stops a Sequence (while it is executing)

See Also mpiSequenceStart

mpiSequenceMemory

Declaration

```
long mpiSequenceMemory(MPISequence sequence,
                       void        **memory)
```

Required Header stdmpi.h

Description **SequenceMemory** writes an address [used to access a Sequence's (sequence) memory] to the contents of ***memory***. This address (or an address calculated from it) is passed as the ***src*** argument to mpiSequenceMemoryGet(...) and as the ***dst*** argument to mpiSequenceMemorySet(...).

Return Values	
MPIMessageOK	if *SequenceMemory* successfully writes the address (used to access Sequence memory) to the contents of memory

See Also mpiSequenceMemoryGet | mpiSequenceMemorySet

mpiSequenceMemoryGet

Declaration

```
long mpiSequenceMemoryGet(MPISequence sequence,
                          void        *dst,
                          void        *src,
                          long        count)
```

Required Header

stdmpi.h

Description

SequenceMemoryGet copies ***count*** bytes of a Sequence's (***sequence***) memory (starting at address ***src***) to application memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *SequenceMemoryGet* successfully copies count bytes of Sequence memory to application memory

See Also

mpiSequenceMemorySet | mpiSequenceMemory

mpiSequenceMemorySet

Declaration

```
long mpiSequenceMemorySet(MPISequence sequence,
                          void        *dst,
                          void        *src,
                          long        count)
```

Required Header

stdmpi.h

Description

SequenceMemorySet copies ***count*** bytes of application memory (starting at address ***src***) to a Sequence's (***sequence***) memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *SequenceMemorySet* successfully copies ***count*** bytes of application memory to a Sequence object's memory

See Also

mpiSequenceMemory | mpiSequenceMemoryGet

mpiSequenceControl

Declaration

```
const MPIControl mpiSequenceControl(MPISequence sequence)
```

Required Header

stdmpi.h

Description

SequenceControl returns a handle to the Control object with which the Sequence object is associated.

sequence	a handle to the Sequence object.

Return Values

MPIControl	a handle to the Sequence object
MPIHandleVOID	if ***sequence*** is invalid

See Also

mpiSequenceCreate | mpiControlCreate

mpiSequenceNumber

Declaration

```
long mpiSequenceNumber(MPISequence sequence,
                       long        *number)
```

Required Header

stdmpi.h

Description

SequenceNumber writes the index of a Sequence (***sequence***, on the motion controller that the Sequence object is associated with) to the contents of ***number***.

Return Values	
MPIMessageOK	if *SequenceNumber* successfully writes the Sequence's index to the contents of ***number***

See Also

mpiSequenceCommand

Declaration

```
const MPICommand mpiSequenceCommand(MPISequence sequence,
                                    long        index)
```

Required Header

stdmpi.h

Description

SequenceCommand returns the element at the position on the list indicated by ***index***.

sequence	a handle to the Sequence object.
index	a position in the list.

Return Values

handle	to the ***index***th Command of a Sequence (***sequence***)
MPIHandleVOID	if ***sequence*** is invalid if ***index*** is less than 0 if ***index*** is greater than or equal to **mpiSequenceCount(sequence)**
MPIMessageARG_INVALID	if ***index*** is a negative number.
MEIListMessageELEMENT_NOT_FOUND	if ***index*** is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	if ***sequence*** is an invalid handle.

See Also

mpiSequenceCommandAppend

Declaration

```
long mpiSequenceCommandAppend(MPISequence sequence,
                              MPICommand  command)
```

Required Header

stdmpi.h

Description

SequenceCommandAppend appends a Command (***command***) to a Sequence (***sequence***).

sequence	a handle to the Sequence object.
command	a handle to a Command object.

Return Values

MPIMessageOK	if *SequenceCommandAppend* successfully appends a Command to a Sequence
MPIMessageHANDLE_INVALID	Either ***sequence*** or ***command*** is an invalid handle.
MPIMessageNO_MEMORY	Not enough memory was available.

See Also

mpiSequenceCommandCount

Declaration

```
long mpiSequenceCommandCount(MPISequence sequence)
```

Required Header

stdmpi.h

Description

SequenceCommandCount returns the number of elements on the list.

sequence	a handle to the Sequence object.

Return Values

number of Commands	in a Sequence (***sequence***)
-1	if ***sequence*** is invalid
0	if ***sequence*** is empty

See Also

mpiSequenceCommandFirst

Declaration

```
const MPICommand mpiSequenceCommandFirst(MPISequence sequence)
```

Required Header

stdmpi.h

Description

SequenceCommandFirst returns the first element in the list. This function can be used in conjuntion with mpiSequenceCommandNext() in order to iterate through the list.

sequence	a handle to the Sequence object.

Return Values

handle	to the first Command in a Sequence (***sequence***)
MPIHandleVOID	if ***sequence*** is invalid if ***sequence*** is empty
MPIMessageHANDLE_INVALID	if ***sequence*** is an invalid handle.

See Also

mpiSequenceCommandNext | mpiSequenceCommandLast

mpiSequenceCommandIndex

Declaration

```
long mpiSequenceCommandIndex(MPISequence sequence,
                             MPICommand command)
```

Required Header

stdmpi.h

Description

SequenceCommandIndex returns the position of "command" on the list.

sequence	a handle to the Sequence object.
command	a handle to a Command object.

Return Values

index	of a Command (***command***) in a Sequence (***sequence***)
-1	if ***sequence*** is invalid if the Command (***command***) was not found in the Sequence (***sequence***)

See Also

mpiSequenceCommandInsert

Declaration

```
long mpiSequenceCommandInsert(MPISequence sequence,
                              MPICommand  command,
                              MPICommand  insert)
```

Required Header

stdmpi.h

Description

SequenceCommandInsert inserts a Command (***insert***) in a Sequence (***sequence***) just after the specified Command (***command***).

Return Values

MPIMessageOK	if *SequenceCommandInsert* successfully inserts the Command (***insert***) in a Sequence following the specified Command (***command***)

See Also

mpiSequenceCommandLast

Declaration

```
const MPICommand mpiSequenceCommandLast(MPISequence sequence)
```

Required Header

stdmpi.h

Description

SequenceCommandLast returns the last element in the list. This function can be used in conjuntion with mpiSequenceCommandPrevious() in order to iterate through the list backwards.

sequence	a handle to the Sequence object.

Return Values

handle	to the last Command in a Sequence (***sequence***)
MPIHandleVOID	if ***sequence*** is invalid if ***sequence*** is empty
MPIMessageHANDLE_INVALID	if ***sequence*** is an invalid handle.

See Also

mpiSequenceCommandFirst | mpiSequenceCommandPrevious | mpiSequenceCommandNext

mpiSequenceCommandListGet

Declaration

```
long mpiSequenceCommandListGet(MPISequence  sequence,
                               long         *commandCount,
                               MPICommand   *commandList)
```

Required Header

stdmpi.h

Description

SequenceCommandListGet gets the Commands in a Sequence (***sequence***). *SequenceCommandListGet* writes the number of Commands [in a Sequence (***sequence***)] to the location (pointed to by ***commandCount***), and also writes an array (of ***commandCount*** Command handles) to the location (pointed to by ***commandList***).

Return Values

MPIMessageOK	if *SequenceCommandListGet* successfully gets the list of Commands in a Sequence

See Also

mpiSequenceCommandListSet

mpiSequenceCommandListSet

Declaration

```
long mpiSequenceCommandListSet(MPISequence  sequence,
                               long         commandCount,
                               MPICommand   *commandList)
```

Required Header

stdmpi.h

Description

SequenceCommandListSet creates a Sequence (***sequence***) of ***commandCount*** Commands using the Command handles specified by ***commandList***. Any existing command Sequence is completely replaced.

The ***commandList*** parameter is the address of an array of ***commandCount*** Command handles, or is NULL (if ***commandCount*** is equal to zero).

You can also create a command Sequence incrementally (i.e., one command at a time), by using the Append and/or Insert methods. Use the List methods to examine and manipulate a command Sequence, regardless of how it was created.

Return Values

MPIMessageOK	if *SequenceCommandListGet* successfully creates a Sequence of Commands using the Command handles specified by ***commandList***

See Also

mpiSequenceCommandListGet

mpiSequenceCommandNext

Declaration

```
const MPICommand mpiSequenceCommandNext(MPISequence sequence,
                                        MPICommand  command)
```

Required Header stdmpi.h

Description

SequenceCommandNext returns the next element following "command" on the list. This function can be used in conjuntion with mpiSequenceCommandFirst() in order to iterate through the list.

sequence	a handle to the Sequence object.
command	a handle to a Command object.

Return Values

handle	to the Command following the Command (***command***) in a Sequence (***sequence***)
MPIHandleVOID	if ***sequence*** is invalid if ***command*** is the last command in a Sequence (***sequence***)
MPIMessageHANDLE_INVALID	Either ***sequence*** or ***command*** is an invalid handle.

See Also mpiSequenceCommandFirst | mpiSequenceCommandPrevious

mpiSequenceCommandPrevious

Declaration

```
const MPICommand mpiSequenceCommandPrevious(MPISequence sequence,
                                            MPICommand  command)
```

Required Header

stdmpi.h

Description

SequenceCommandPrevious returns the previous element prior to "command" on the list. This function can be used in conjuntion with mpiSequenceCommandLast() in order to iterate through the list backwards.

sequence	a handle to the Sequence object.
command	a handle to a Command object.

Return Values

handle	to the Command preceding the Command (***command***) in a Sequence (***sequence***)
MPIHandleVOID	if ***sequence*** is invalid if ***command*** is the first command in a Sequence (***sequence***)
MPIMessageHANDLE_INVALID	Either ***sequence*** or ***command*** is an invalid handle.

See Also

mpiSequenceCommandLast | mpiSequenceCommandNext

mpiSequenceCommandRemove

Declaration

```
long mpiSequenceCommandRemove(MPISequence sequence,
                              MPICommand  command)
```

Required Header

stdmpi.h

Description

SequenceCommandRemove removes a Command (***command***) from a Sequence (***sequence***).

Return Values	
MPIMessageOK	if *SequenceCommandRemove* successfully removes the Command from a Sequence

See Also

MPISequenceConfig / MEISequenceConfig

MPISequenceConfig

```
typedef MPIEmpty    MPISequenceConfig;
```

Description **SequenceConfig** is currently not supported and is reserved for future use.

MEISequenceConfig

```
typedef MPIEmpty    MEISequenceConfig;
```

Description **SequenceConfig** is currently not supported and is reserved for future use.

See Also mpiSequenceConfigGet | mpiSequenceConfigSet

MPISequenceMessage

MPISequenceMessage

```
typedef enum {

    MPISequenceMessageSEQUENCE_INVALID,
    MPISequenceMessageCOMMAND_COUNT,
    MPISequenceMessageCOMMAND_NOT_FOUND,
    MPISequenceMessageSTARTED,
    MPISequenceMessageSTOPPED,
} MPISequenceMessage;
```

Description

MPISequenceMessageSEQUENCE_INVALID	An invalid sequence number has been specified or a disabled sequence object has been specified.
MPISequenceMessageCOMMAND_COUNT	The program sequencer's command count is 0.
MPISequenceMessageCOMMAND_NOT_FOUND	The program sequencer command is not a valid command.
MPISequenceMessageSTARTED	The program sequencer has already been started.
MPISequenceMessageSTOPPED	The program sequencer has already been stopped.

See Also

MPISequenceState

MPISequenceState

```
typedef enum {
    MPISequenceStateSTOPPED,
    MPISequenceStateSTARTED,
} MPISequenceState;
```

Description

MPISequenceStateSTOPPED	Means that the XMP's on-board program sequencer state is stopped. The program sequencer is in this state after it is created, and is not running. If the program sequencer has already been started, then a call to the MPI method mpiSequenceStop will stop the sequencer, and the sequencer state will be MPISequenceStateSTOPPED.
MPISequenceStateSTARTED	Means that the XMP's on-board program sequencer state is running. The program sequencer is in this state after it has been created, and successfully started with a call to the MPI method mpiSequenceStart.

See Also

MPISequenceStatus

MPISequenceStatus

```
typedef struct MPISequenceStatus {
    MPICommand          command;
    MPISequenceState    state;
} MPISequenceStatus;
```

Description

MPISequenceStatus is a status structure for MPISequence objects

command	The current command of the MPISequence object
state	The current state of the MPISequence object

See Also

MPISequence | mpiSequenceStatus

MEISequenceTrace

MEISequenceTrace

```
typedef enum {
        MEISequenceTraceLOAD,

} MEISequenceTrace;
```

Description **MPISequenceTrace** sets tracing on for the mpiSequenceLoad() method.

See Also MPISequence | MEITrace | mpiSequenceLoad

Sercos Objects

Introduction

A **Sercos** object manages an individual SERCOS master.

The **SERCOS** (**S**erial **R**eal-**T**ime **C**ommunication **S**ystem) protocol was designed specifically for the motion control industry. The protocol uses a ring topology which allows communication to occur at specific assigned times within a predetermined cycle. Thus, SERCOS is deterministic allowing the Master (control unit) to synchronize command and feedback values to and from all of the Slaves (drives).

The MPI supports the use of SERCOS fiberoptic technology. The XMP-SERCOS-CPCI supports up to 3 SERCOS rings with support for up to 24 drives, in a 2-slot configuration in a CompactPCI form-factor. The MPI includes 4 SERCOS object types: SERCOS, NODE, IDN, and IDNLIST. Utility programs for initializing a ring and reading or writing IDNs are also provided with the software distribution.

Only a limited number of SERCOS drive manufacturers are supported in the MPI. For a list of supported drives, click here. Contact MEI for more information about supported SERCOS drives.

Introduction | Overview | Data Types | Communications
Procedures | Telegrams | Topologies

Methods

Create, Delete, Validate Methods

mpiSercos**Create**	Create Sercos object
mpiSercos**Delete**	Delete Sercos object
mpiSercos**Validate**	Validate Sercos object

Configuration and Information Methods

mpiSercos**ConfigGet**	Get Sercos object's configuration
mpiSercos**ConfigSet**	Set Sercos object's configuration
mpiSercos**Error**	Set contents of error with platform-specific info (about the error)
mpiSercos**FlashConfigGet**	Get flash config of Sercos
mpiSercos**FlashConfigSet**	Set flash config of Sercos
meiSercos**ServiceIdnFieldGet**	
meiSercos**ServiceIdnFieldSet**	
meiSercos**ServiceProcedure**	
mpiSercos**Status** / meiSercos**Status**	Get the status of a Sercos object

Action Methods

mpiSercos**Init**	Initialize a Sercos object to a specified SERCOS phase
mpiSercos**Reset**	Reset a Sercos object to SERCOS Phase 0

Memory Methods

mpiSercos**Memory**	Set Sercos memory address
mpiSercos**MemoryGet**	Copy bytes of Sercos memory to application memory
mpiSercos**MemorySet**	Copy bytes of application memory to Sercos memory

Relational Methods

mpiSercos**Control**	Get handle to Control object that a Sercos object is associated with
mpiSercos**Number**	Get the index of a Sercos object from the Control object that the Sercos object is associated with

List Methods for Sercos Nodes

mpiSercos**Node**	Get handle to the indexth Node in a Sercos object's list
mpiSercos**NodeAppend**	Append a Node object to a Sercos object's list
mpiSercos**NodeCount**	Get the number of Node objects in a Sercos object's list
mpiSercos**NodeFirst**	Get handle to the first Node in a Sercos object's list
mpiSercos**NodeIndex**	Get the index of a Node object in a Sercos object's list
mpiSercos**NodeInsert**	Insert a Node into a Sercos object's list
mpiSercos**NodeLast**	Get handle to the last Node in a Sercos object's list
mpiSercos**NodeListGet**	Get the list of Nodes associated with a Sercos object
mpiSercos**NodeListSet**	Create a list of Nodes associated with a Sercos object
mpiSercos**NodeNext**	Get handle to the next Node in a Sercos object's list
mpiSercos**NodePrevious**	Get handle to the previous Node in a Sercos object's list
mpiSercos**NodeRemove**	Remove a Node from a Sercos object's list

Data Types

MPISercos**Baud**
MPISercos**Error**
MPISercos**ErrorGroup**
MPISercos**ErrorType**
MPISercos**LoopStatus**
MPISercos**Message** / MEISercos**Message**
MPISercos**ProcedureAction**
MEISercos**ServiceContainer**
MPISercos**Status**

Constants

MPISercos**NODE_COUNT_MAX**

Macros

mpiSercos**NodeIdnDataSET**
mpiSercos**NodeIdnGET**

Copyright © 2002
Motion Engineering

mpiSercosCreate

Declaration

```
const MPISercos mpiSercosCreate(MPIControl control,
                                long       number)
```

Required Header

stdmpi.h

Description

SercosCreate creates a Sercos object that is associated with the SERCON 410B chip, which in turn is identified by ***number*** on a motion controller (***control***). *SercosCreate* is the equivalent of a C++ constructor.

Return Values

handle	to a Sercos object (associated with ***number*** and ***control***)
MPIHandleVOID	if the object could not be created

See Also

mpiSercosDelete | mpiSercosValidate

mpiSercosDelete

Declaration

```
long mpiSercosDelete(MPISercos sercos)
```

Required Header

stdmpi.h

Description

SercosDelete deletes a Sercos object and invalidates its handle (***sercos***). *SercosDelete* is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if *SercosDelete* successfully deletes the Sercos object and invalidates its handle

See Also

mpiSercosCreate | mpiSercosValidate

mpiSercosValidate

Declaration `long mpiSercosValidate(MPISercos sercos)`

Required Header stdmpi.h

Description **SercosValidate** validates the Sercos object and its handle (***sercos***).

Return Values	
MPIMessageOK	if Sercos is a handle to a valid object.

See Also mpiSercosCreate | mpiSercosDelete

mpiSercosConfigGet

Declaration

```
long mpiSercosConfigGet(MPISercos       sercos,
                        MPISercosConfig *config,
                        void            *external)
```

Required Header

stdmpi.h

Description

SercosConfigGet gets a Sercos object's (***sercos***) configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is *in addition* to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type MEISercosConfig{} or is NULL.

Return Values

MPIMessageOK	if *SercosConfigGet* successfully writes the Sercos object's configuration to the structure(s)

See Also

mpiSercosConfigSet

mpiSercosConfigSet

Declaration

```
long mpiSercosConfigSet(MPISercos       sercos,
                        MPISercosConfig *config,
                        void            *external)
```

Required Header

stdmpi.h

Description

SercosConfigSet sets a Sercos object's (***sercos***) configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only — ***external*** either points to a structure of type MEISercosConfig{} or is NULL.

Return Values

MPIMessageOK	if *SercosConfigSet* successfully sets the Sercos object's configuration using data from the structure(s)

See Also

mpiSercosConfigGet

mpiSercosError

Declaration

```
long mpiSercosError(MPISercos      sercos,
                    MPISercosError *error)
```

Required Header

stdmpi.h

Description

SercosError reads the error code from a Sercos object and writes it into the location pointed to by ***error***.

sercos	a handle to a Sercos object
***error**	a pointer to a MPISercosError

Return Values

MPIMessageOK	if *mpiSercosError* successfully writes the current ***sercos*** error to ***error***.

See Also

mpiSercosFlashConfigGet

Declaration

```
long mpiSercosFlashConfigGet(MPISercos        sercos,
                             void             *flash,
                             MPISercosConfig  *config,
                             void             *external)
```

Required Header

stdmpi.h

Description

SercosFlashConfigGet gets a Sercos object's (***sercos***) flash configuration and writes it in the structure pointed to by ***config***, and also writes it in the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Sercos object's flash configuration information in ***external*** is in addition to the Sercos object's flash configuration information in ***config***, i.e, the flash configuration information in ***config*** and in ***external*** is not the same information. Note that config or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEISercosConfig{}** or is NULL.

Return Values

MPIMessageOK	if *SercosFlashConfigGet* successfully writes the Sercos object's flash configuration to the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiSercosFlashConfigSet

mpiSercosFlashConfigSet

Declaration

```
long mpiSercosFlashConfigSet(MPISercos        sercos,
                             void             *flash,
                             MPISercosConfig  *config,
                             void             *external)
```

Required Header

stdmpi.h

Description

SercosFlashConfigSet sets a Sercos object's (***sercos***) flash configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Sercos object's flash configuration information in ***external*** is in addition to the Sercos object's flash configuration information in ***config***, i.e, the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only

external either points to a structure of type **MEISercosConfig{}** or is NULL.

Return Values

MPIMessageOK	if *SercosFlashConfigSet* successfully sets the Sercos object's flash configuration using data from the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiSercosFlashConfigGet

meiSercosServiceIdnFieldGet

Declaration

```
long meiSercosServiceIdnFieldGet(MPISercos                  sercos,
                                 MEISercosServiceContainer *service,
                                 MPIIdn                     idn,
                                 MPIIdnField                field)
```

Required Header

stdmei.h

Description

SercosServiceIdnFieldGet uses a service container to read the specified ***field*** from an ***idn***, and write it into the idn object.

sercos	a handle to a Sercos object
***service**	a pointer to a service container
idn	a handle to an Idn object
field	enumeration corresponding to each element in the structure MPIIdnElement

Return Values

MPIMessageOK	if *SercosServiceIdnFieldGet* successfully reads the field from the service container.

See Also

mpiSercosServiceIdnFieldSet | MPIIdnElement

meiSercosServiceIdnFieldSet

Declaration

```
long meiSercosServiceIdnFieldSet(MPISercos                  sercos,
                                 MEISercosServiceContainer *service,
                                 MPIIdn                     idn,
                                 MPIIdnField                field)
```

Required Header

stdmei.h

Description

SercosServiceIdnFieldSet uses a service container to write the specified ***field*** from an idn object to an ***idn***.

sercos	a handle to a Sercos object
***service**	a pointer to a service container
idn	a handle to an Idn object
field	enumeration corresponding to each element in the structure MPIIdnElement

Return Values

MPIMessageOK	if *SercosServiceIdnFieldSet* successfully writes the field to an idn via the service container.

See Also

mpiSercosServiceIdnFieldGet | MPIIdnElement

meiSercosServiceProcedure

Declaration

```
long meiSercosServiceProcedure(MPISercos                  sercos,
                               MEISercosServiceContainer *service,
                               MPISercosProcedureAction   action,
                               MPIIdn                     idn,
                               unsigned long             *status)
```

Required Header

stdmei.h

Description

SercosServiceProcedure uses a service container to send a procedure ***action*** to a sercos ***idn***.

sercos	a handle to a Sercos object
***service**	a pointer to a service container
action	a command used to invoke a procedure
idn	a handle to an Idn object
***status**	a pointer to an unsigned long

Return Values

MPIMessageOK	if *SercosServiceProcedure* successfully sends a procedure action via a service container.

See Also

MPISercosProcedureAction

mpiSercosStatus / meiSercosStatus

mpiSercosStatus

Declaration

```
long mpiSercosStatus(MPISercos        sercos,
                     MPISercosStatus  *status,
                     void             *external)
```

Required Header stdmpi.h

Description **SercosStatus** writes the status of a Sercos object (***sercos***) to the structure pointed to by ***status***.

Return Values	
MPIMessageOK	if *SercosStatus* successfully writes the status of a Sercos object to the structure

meiSercosStatus

Declaration

```
long meiSercosStatus(MPIControl         control,
                     MEIMotorTypeInfo   *motorInfo,
                     MPISercosStatus    *status)
```

Required Header stdmei.h

Description **SercosStatus** reads the status from a ***control*** and writes it into the location pointed to by ***status***. The status is read from the nodeNumber specified in the structure pointed to by ***motorInfo***.

control	a handle to the Sercos object.
***motorInfo**	a pointer to a MEIMotorTypeInfo structure
***status**	a handle to a Node object.

Return Values	
MPIMessageOK	if *SercosStatus* successfully reads the status.

See Also

mpiSercosInit

Declaration

```
long mpiSercosInit(MPISercos sercos,
                   long      phase)
```

Required Header

stdmpi.h

Description

SercosInit advances the initialization of a Sercos object (***sercos***) to a specific phase (***phase***).

If ***phase*** = -1, then the Sercos object is moved to Phase 4 (full initialization). If the current phase number (of the Sercos object) is greater than ***phase***, initialization restarts at Phase 0 and advances to Phase ***phase***.

Return Values

MPIMessageOK	if *SercosInit* successfully initializes the state of a Sercos object (***sercos***) to the Phase ***phase***

See Also

mpiSercosReset

Declaration `long mpiSercosReset(MPISercos sercos)`

Required Header stdmpi.h

Description **SercosReset** resets a Sercos object (***sercos***), thereby setting its initialization phase to 0.

Return Values

MPIMessageOK	if *SercosReset* successfully resets the Sercos object

See Also

mpiSercosMemory

Declaration

```
long mpiSercosMemory(MPISercos sercos,
                     void      **memory)
```

Required Header

stdmpi.h

Description

SercosMemory writes an address [that is used to access Sercos (sercos) memory] to the contents of ***memory***. This address (or an address calculated from it) is passed as the ***src*** argument to mpiSercosMemoryGet(...) and as the ***dst*** argument to mpiSercosMemorySet(...).

Return Values

MPIMessageOK	if *SercosMemory* successfully writes the Sercos memory address to the contents of ***memory***

See Also

mpiSercosMemoryGet | mpiSercosMemorySet

mpiSercosMemoryGet

Declaration

```
long mpiSercosMemoryGet(MPISercos sercos,
                        void      *dst,
                        void      *src,
                        long      count)
```

Required Header

stdmpi.h

Description

SercosMemoryGet copies ***count*** bytes of Sercos (***sercos***) memory (starting at address ***src***) to application memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *SercosMemoryGet* successfully copies count bytes of Sercos memory to application memory

See Also

mpiSercosMemorySet | mpiSercosMemory

mpiSercosMemorySet

Declaration

```
long mpiSercosMemorySet(MPISercos sercos,
                        void      *dst,
                        void      *src,
                        long      count)
```

Required Header

stdmpi.h

Description

SercosMemorySet copies ***count*** bytes of application memory (starting at address ***src***) to Sercos (***sercos***) memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *SercosMemorySet* successfully copies count bytes of application memory to Sercos memory

See Also

mpiSercosMemoryGet | mpiSercosMemory

mpiSercosControl

Declaration `const MPIControl mpiSercosControl(MPISercos sercos)`

Required Header stdmpi.h

Description **SercosControl** returns a handle to the Control object with which the Sercos object is associated.

sercos	a handle to the Sercos object

Return Values

handle	to the Control object that a Sercos object (***sercos***) is associated with
MPIHandleVOID	if ***sercos*** is invalid

See Also mpiSercosCreate | mpiControlCreate

mpiSercosNumber

Declaration

```
long mpiSercosNumber(MPISercos sercos,
                     long      *number)
```

Required Header

stdmpi.h

Description

SercosNumber writes the index of a Sercos object (***sercos***, on the motion controller that the Sercos object is associated with) to the location pointed to by ***number***.

Return Values

MPIMessageOK	if *SercosNumber* successfully writes the Sercos object's index to the location

See Also

mpiSercosNode

Declaration

```
const MPINode  mpiSercosNode(MPISercos sercos,
                             long      index)
```

Required Header

stdmpi.h

Description

SercosNode returns the element at the position on the list indicated by ***index***.

sercos	a handle to the Sercos object.
index	a position in the list.

Return Values

handle	to the ***index***th Node object of a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***index*** is less than 0 if ***index*** is greater than **mpiSercosCount**(***sercos***) if ***index*** is equal to **mpiSercosCount**(***sercos***)
MPIMessageARG_INVALID	if ***index*** is a negative number.
MEIListMessageELEMENT_NOT_FOUND	if ***index*** is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	if ***sercos*** is an invalid handle.

See Also

mpiSercosNodeAppend

Declaration

```
long mpiSercosNodeAppend(MPISercos sercos,
                         MPINode   node)
```

Required Header

stdmpi.h

Description

SercosNodeAppend appends a Node object (***node***) to a Sercos object (***sercos***).

sercos	a handle to the Sercos object.
node	a handle to a Node object.

Return Values

MPIMessageOK	if *SercosNodeAppend* successfully appends the Node object to the Sercos object
MPIMessageHANDLE_INVALID	Either ***sercos*** or ***node*** is an invalid handle.
MPIMessageNO_MEMORY	Not enough memory was available.

See Also

mpiSercosNodeCount

Declaration

```
long mpiSercosNodeCount(MPISercos sercos)
```

Required Header

stdmpi.h

Description

SercosNodeCount returns the number of elements on the list.

sercos	a handle to the Sercos object.

Return Values

number	of Node objects in a Sercos object (***sercos***)
-1	if ***sercos*** is invalid
0	if ***sercos*** is empty

See Also

mpiSercosNodeFirst

Declaration

```
const MPINode mpiSercosNodeFirst(MPISercos sercos)
```

Required Header

stdmpi.h

Description

SercosNodeFirst returns the first element in the list. This function can be used in conjuntion with mpiSercosNodeNext() in order to iterate through the list.

sercos	a handle to the Sercos object.

Return Values

handle	to the first Node object of a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***sercos*** is empty
MPIMessageHANDLE_INVALID	if ***sercos*** is an invalid handle.

See Also

mpiSercosNodeNext | mpiSercosNodeLast

mpiSercosNodeIndex

Declaration

```
long mpiSercosNodeIndex(MPISercos sercos,
                        MPINode node)
```

Required Header

stdmpi.h

Description

SercosNodeIndex returns the position of ***node*** on the list.

sercos	a handle to the Sercos object.
node	a handle to a Node object.

Return Values

index	of a Node object (***node***) in a Sercos object (***sercos***)
-1	if ***sercos*** is invalid if ***node*** was not found in ***sercos***

See Also

mpiSercosNodeInsert

Declaration

```
long mpiSercosNodeInsert(MPISercos sercos,
                         MPINode   node,
                         MPINode   insert)
```

Required Header

stdmpi.h

Description

SercosNodeInsert inserts a Node object (***insert***) in a Sercos object (***sercos***), just after the specified Node object (***node***).

Return Values

MPIMessageOK	if *SercosNodeInsert* successfully inserts the Node object (***insert***) just after the specified Node object (***node***), in the Sercos object (***sercos***)

See Also

mpiSercosNodeLast

Declaration

```
const MPINode mpiSercosNodeLast(MPISercos sercos)
```

Required Header

stdmpi.h

Description

SercosNodeLast returns the last element in the list. This function can be used in conjuntion with mpiSercosNodePrevious() in order to iterate through the list backwards.

sercos	a handle to the Sercos object.

Return Values

handle	to the last Node object (***node***) of a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***sercos*** is empty
MPIMessageHANDLE_INVALID	if ***sercos*** is an invalid handle.

See Also

mpiSercosNodePrevious | mpiSercosNodeFirst

mpiSercosNodeListGet

Declaration

```
long mpiSercosNodeListGet(MPISercos sercos,
                          long      *nodeCount,
                          MPINode   *nodeList)
```

Required Header

stdmpi.h

Description

SercosNodeListGet gets the Nodes in a Sercos object (***sercos***), and writes an array (of handles to those Nodes, size= ***nodeCount***) to the location pointed to by ***nodeList***, and also writes the number of Nodes (in the Sercos object) to the location pointed to by ***nodeCount***.

Return Values

MPIMessageOK	if *SercosNodeListGet* successfully writes the array of Node handles and the number of Nodes to the 2 locations (***nodeList, nodeCount***)

See Also

mpiSercosNodeListSet

mpiSercosNodeListSet

Declaration

```
long mpiSercosNodeListSet(MPISercos sercos,
                          long      nodeCount,
                          MPINode   *nodeList)
```

Required Header

stdmpi.h

Description

Using the Node handles specified by ***nodeList***, **SercosNodeListSet** creates a list of Nodes (length=***nodeCount***). Any existing Node list of the Sercos object (***sercos***) is completely replaced.

The ***nodeList*** argument is the address of an array of ***nodeCount*** Node handles, or is NULL (if ***nodeCount*** is equal to zero).

You can also create a Node list incrementally (one Node at a time), by using the **mpiNodeListAppend(...)** and/or **mpiNodeListInsert(...)** methods. You can use any *mpiNodeList* method to examine and manipulate a Node list sequence, regardless of how the the Node list was created.

Return Values

MPIMessageOK	if *IdnListSet* successfully creates the IdnList using the handles in ***idnArray***

See Also

mpiSercosNodeListGet

mpiSercosNodeNext

Declaration

```
const MPINode  mpiSercosNodeNext(MPISercos sercos,
                                 MPINode   node)
```

Required Header

stdmpi.h

Description

SercosNodeNext returns the next element following "node" on the list. This function can be used in conjuntion with mpiSercosNodeFirst() in order to iterate through the list.

sercos	a handle to the Sercos object.
node	a handle to a Node object.

Return Values

handle	to the Node object that is just after the specified Node object (***node***), in a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***node*** is the last Node in the Sercos object (***sercos***)
MPIMessageHANDLE_INVALID	Either ***sercos*** or ***node*** is an invalid handle.

See Also

mpiSercosNodeFirst | mpiSercosNodePrevious

mpiSercosNodePrevious

Declaration

```
const MPINode  mpiSercosNodePrevious(MPISercos sercos,
                                     MPINode   node)
```

Required Header

stdmpi.h

Description

SercosNodePrevious returns the previous element prior to "node" on the list. This function can be used in conjuntion with mpiSercosNodeLast() in order to iterate through the list backwards.

sercos	a handle to the Sercos object.
node	a handle to a Node object.

Return Values

handle	to the Node object that is just before the specified Node object (node), in a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***node*** is the first Node in the Sercos object (***sercos***)
MPIMessageHANDLE_INVALID	Either ***sercos*** or ***node*** is an invalid handle.

See Also

mpiSercosNodeLast | mpiSercosNodeNext

mpiSercosNodeRemove

Declaration

```
long mpiSercosNodeRemove(MPISercos sercos,
                         MPINode   node)
```

Required Header

stdmpi.h

Description

SercosNodeRemove removes a Node object (***node***) from a Sercos object (***sercos***).

Return Values

MPIMessageOK	if *SercosNodeRemove* successfully removes the Node object from the Sercos object

See Also

MPISercosBaud

MPISercosBaud

```
typedef enum {
    MPISercosBaudINVALID,

    MPISercosBaud2MBIT,
    MPISercosBaud4MBIT,
    MPISercosBaud10MBIT,

} MPISercosBaud;
```

Description

SercosBaud defines the different baud rates that the SERCOS ring can communicate at.

See Also

MPISercosError

MPISercosError

```
typedef enum {
    MPISercosErrorSHIFT,

    MPISercosErrorGENERAL,
    MPISercosErrorNONE,
    MPISercosErrorCHANNEL_NOT_OPEN,
    MPISercosErrorCHANNEL_ACCESS,

    MPISercosErrorIDN,
    MPISercosErrorNO_IDN,
    MPISercosErrorIDN_ACCESS,

    MPISercosErrorNAME,
    MPISercosErrorNO_NAME,
    MPISercosErrorNAME_TOO_SHORT,
    MPISercosErrorNAME_TOO_LONG,
    MPISercosErrorNAME_NO_CHANGE,
    MPISercosErrorNAME_WRITE_PROTECT,

    MPISercosErrorATTR,
    MPISercosErrorATTR_TOO_SHORT,
    MPISercosErrorATTR_TOO_LONG,
    MPISercosErrorATTR_NO_CHANGE,
    MPISercosErrorATTR_WRITE_PROTECT,

    MPISercosErrorUNIT,
    MPISercosErrorNO_UNIT,
    MPISercosErrorUNIT_TOO_SHORT,
    MPISercosErrorUNIT_TOO_LONG,
    MPISercosErrorUNIT_NO_CHANGE,
    MPISercosErrorUNIT_WRITE_PROTECT,

    MPISercosErrorMIN,
    MPISercosErrorNO_MIN,
    MPISercosErrorMIN_TOO_SHORT,
    MPISercosErrorMIN_TOO_LONG,
    MPISercosErrorMIN_NO_CHANGE,
    MPISercosErrorMIN_WRITE_PROTECT,

    MPISercosErrorMAX,
    MPISercosErrorNO_MAX,
    MPISercosErrorMAX_TOO_SHORT,
    MPISercosErrorMAX_TOO_LONG,
    MPISercosErrorMAX_NO_CHANGE,
    MPISercosErrorMAX_WRITE_PROTECT,

    MPISercosErrorOP,
    MPISercosErrorOP_TOO_SHORT,
```

```
        MPISercosErrorOP_TOO_LONG,
        MPISercosErrorOP_NO_CHANGE,
        MPISercosErrorOP_WRITE_PROTECT,
        MPISercosErrorOP_MIN,
        MPISercosErrorOP_MAX,
        MPISercosErrorOP_DATA,
        MPISercosErrorOP_PASSWORD,
    } MPISercosError;
```

Description

The **SercosError** enumeration defines all the different errors that can occur in SERCOS.

See Also

For a more in-depth breakdown of the errors, please refer to a SERCOS Specification Manual.

MPISercosErrorGroup

MPISercosErrorGroup

```
typedef enum {
    MPISercosErrorGroupINVALID,

    MPISercosErrorGroupGENERAL,
    MPISercosErrorGroupIDN,
    MPISercosErrorGroupNAME,
    MPISercosErrorGroupATTR,
    MPISercosErrorGroupUNIT,
    MPISercosErrorGroupMIN,
    MPISercosErrorGroupMAX,
    MPISercosErrorGroupOP,

} MPISercosErrorGroup;
```

Description

When an error occurs, **SercosErrorGroup** defines what part of the IDN field has the error.

See Also

MPISercosErrorType

MPISercosErrorType

```
typedef enum {
    MPISercosErrorTypeNONE,
    MPISercosErrorTypeDOES_NOT_EXIST,
    MPISercosErrorTypeTOO_SHORT,
    MPISercosErrorTypeTOO_LONG,
    MPISercosErrorTypeNO_CHANGE,
    MPISercosErrorTypeWRITE_PROTECT,
    MPISercosErrorTypeTOO_SMALL,
    MPISercosErrorTypeTOO_BIG,
    MPISercosErrorTypeINVALID_DATA,
    MPISercosErrorTypeINVALID_ACCESS,
    MPISercosErrorTypePASSWORD_PROTECT,
} MPISercosErrorType;
```

Description

SercosErrorType enumeration contains basic SERCOS error types that are ORed into the MPISercosErrorGroup enumeration to construct unique MPISercosError codes.

MPISercosErrorTypeNONE	This error type indicates no error occurred.
MPISercosErrorTypeDOES_NOT_EXIST	This error type indicates that a service channel is not open, an idn does not exist or an idn element does not exist.
MPISercosErrorTypeTOO_SHORT	This error type indicates an idn element is too short.
MPISercosErrorTypeTOO_LONG	This error type indicates an idn element is too long.
MPISercosErrorTypeNO_CHANGE	This error type indicates an idn element cannot be changed.
MPISercosErrorTypeWRITE_PROTECT	This error type indicates an idn element is read only.
MPISercosErrorTypeTOO_SMALL	This error type indicates the idn operation data is too small.
MPISercosErrorTypeTOO_BIG	This error type indicates the idn operation data is too big.
MPISercosErrorTypeINVALID_DATA	This error type indicates the idn operation data is not valid.
MPISercosErrorTypeINVALID_ACCESS	This error type indicates a service channel is not accessible or an idn is not accessible.
MPISercosErrorTypePASSWORD_PROTECT	This error type indicates the idn operation data is password protected.

See Also

MPISercosError

MPISercosLoopStatus

MPISercosLoopStatus

```
typedef enum {
     MPISercosLoopStatusOPEN,
} MPISercosLoopStatus;
```

Description

SercosLoopStatus defines the status of a SERCOS loop.

See Also

MPISercosMessage / MEISercosMessage

MPISercosMessage

```
typedef enum {

    MPISercosMessageSERCOS_INVALID,
    MPISercosMessageHANDSHAKE_TIMEOUT,
    MPISercosMessageIDN_FIELD_INVALID,
    MPISercosMessageMST_RECEIVE_ERROR,
    MPISercosMessageNODE_NOT_FOUND,
    MPISercosMessagePROCEDURE_DATA_INVALID,
    MPISercosMessagePROCEDURE_ERROR,
    MPISercosMessagePROTOCOL_ERROR,
    MPISercosMessageRING_NOT_CLOSED,
    MPISercosMessageSAMPLE_RATE_INVALID,
    MPISercosMessageSERVICE_CHANNEL_BUSY,
    MPISercosMessageSERVICE_CHANNEL_ERROR,
    MPISercosMessagePROCEDURE_TIMEOUT,
    MPISercosMessageM_BUSY_TIMEOUT,
} MPISercosMessage;
```

Description

MPISercosMessageSERCOS_INVALID	
Meaning	The specified SERCOS ring is not valid.
Possible Causes	A number passed to MPISercosCreate( ) is larger than the maximum allowable number of Sercos objects.
Recommendations	If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageHANDSHAKE_TIMEOUT	
Meaning	A timeout has occurred during a SERCOS service container operation.
Possible Causes	If there is some sort of communication error, the handshaking will probably not be synchronized, and a timeout will occur.
Recommendations	If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageIDN_FIELD_INVALID	
Meaning	A user is trying to read an IDN field that does not exist.
Possible Causes	This error occurs when the user is trying to read the Min or Max IDN field on an IDN whose data is Variable.
Recommendations	If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageMST_RECEIVE_ERROR	
Meaning	.The wrong data type is being used to access an IDN field.
Possible Causes	This error occurs when the user is trying to read the Min or Max IDN field on an IDN whose data is Variable.
Recommendations	.If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageNODE_NOT_FOUND	
Meaning	The specified Node number is invalid.
Possible Causes	In the phase 2 transition, if a Node in the Sercos ring has a node number that is greater than the MEIXmpSercosNodeCountMAX, this error will occur.
Recommendations	.If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessagePROCEDURE_DATA_INVALID	
Meaning	The wrong data type is being used to access an IDN field.
Possible Causes	This error occurs when the user is trying to read the Min or Max IDN field on an IDN whose data is Variable.
Recommendations	.If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageMST_RECEIVE_ERROR	
Meaning	The wrong data type is being used to access an IDN field.
Possible Causes	This error occurs when the user is trying to read the Min or Max IDN field on an IDN whose data is Variable.
Recommendations	.If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MEISercosMessage

```
typedef enum {
    MEISercosMessage    BUFFER_SIZE_ERROR,
} MEISercosMessage;
```

Description

MEISercosMessageBUFFER_SIZE_ERROR	
Meaning	There is no memory allocated for a given SERCOS object.
Possible Causes	When mpiSercosValidate() is called and the firmware does not have any space allocated for SERCOS.
Recommendations	Make sure the controller has been configured for SERCOS.

See Also

MPISercosProcedureAction

MPISercosProcedureAction

```
typedef enum {
    MPISercosProcedureActionINVALID,

    MPISercosProcedureActionCLEAR_AND_START,
    MPISercosProcedureActionSTART,
    MPISercosProcedureActionCANCEL,
    MPISercosProcedureActionSTATUS,
    MPISercosProcedureActionCLEAR_AND_EXECUTE,
    MPISercosProcedureActionEXECUTE,
    MPISercosProcedureActionNONE,

} MPISercosProcedureAction;
```

Description

SercosProcedureAction lists all of the different commands that can be applied to a SERCOS procedure.

See Also

MEISercosServiceContainer

MEISercosServiceContainer

```
typedef struct MEISercosServiceContainer {
    unsigned long       header[5];

    unsigned long       write[20];
    unsigned long       read[20];
    unsigned long       readOverhang;
} MEISercosServiceContainer;
```

Description

SercosServiceContainer is part of the Sercos message where noncyclic data is requested and sent. It resides in both the AT and MDT.

See Also

For a definition of the individual fields, consult a SERCOS specification manual.

MPISercosStatus

MPISercosStatus

```
typedef struct MPISercosStatus {
    MPISercosLoopStatus  loopStatus;
    long                 phase;
    long                 nodeCount;
} MPISercosStatus;
```

Description

SercosStatus gives the status of the SERCOS ring.

loopStatus	the current state of the ring (ie open or closed)
phase	the phase in which the SERCOS ring is in [0-4]
nodeCount	the number of nodes on the SERCOS ring

See Also

MPISercosNODE_COUNT_MAX

MPISercosNODE_COUNT_MAX

```
#define  MPISercosNODE_COUNT_MAX    (32)
```

Description

SercosNODE_COUNT_MAX defines the maximum number of Nodes that can be on a SERCOS ring.

See Also

mpiSercosNodeIdnDataSET

Declaration

```
#define   mpiSercosNodeIdnDataSET(sercos,node,idn)
          mpiSercosNodeIdnFieldSet((sercos), (node), (idn), MPIIdnFieldDATA)
```

Required Header

stdmpi.h

Description

SercosNodeIdnDataSET writes the data fields from an idn object to an ***idn*** on a sercos ***node***.

See Also

mpiSercosNodeIdnGET

Declaration

```
#define   mpiSercosNodeIdnGET(sercos,node,idn)
          mpiSercosNodeIdnFieldGet((sercos), (node), (idn), MPIIdnFieldALL)
```

Required Header

stdmpi.h

Description

SerocsNodeIdnGET reads all the fields from ***idn*** located on a sercos ***node*** and writes them into the idn object.

See Also

Sercos- Introduction

About SERCOS | Supported Drives

About the MEI SERCOS Controller

The SERCOS/XMP Series controllers from MEI are an extension of the XMP Series motion controllers. XMP Series motion controllers support analog motion control outputs, encoder inputs, and discrete digital I/O. SERCOS/XMP Series motion controllers replace these signals with the SERCOS digital fiber optic network interface. The SERCOS interface only requires two fiber optic connections (one output and one input) to connect to a fiber loop containing up to 8 axes of motion.

Both SERCOS and standard XMP Series controllers share the same basic hardware architecture, onboard firmware, host software and many other features. So, for controller installation procedures, Motion Console application and C programming information, use the XMP's standard documentation.

However, because the SERCOS IDNs are actually implemented in the drive (and not in the controller), there are many SERCOS functions not documented in the XMP documentation, because they are in the drive's documentation (because these functions are associated with the drive and not the controller). Motion Engineering adheres to the specifications set forth by the IEC concerning SERCOS. For more information regarding SERCOS or the SERCOS specification, please contact SERCOS N.A at www.sercos.com.

TOP

Supported Drives / Modules

MEI currently supports drive and I/O modules from a variety of manufacturers. If you desire support for a drive or I/O vendor not listed, please contact MEI.

Manufacturer	Device
Indramat	Servo Drives
Modicon	Servo Drives
Lutze	Digital I/O Modules
Pacific Scientific	Servo Drives
Kollmorgen	Servo Drives
Sanyo Denki	Servo Drives

TOP

Copyright © 2002
Motion Engineering

Sercos- Overview

Summary | Operation Modes | Closed-Loop Tuning | Data Transmission

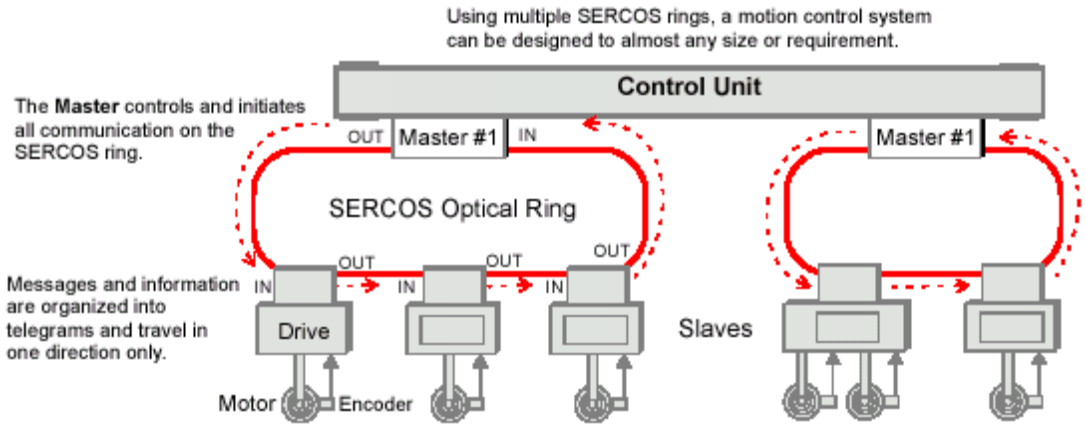
Summary

SERCOS (SErial Real-time COmmunication System) is the international standard for optical communication between motion control units and drive modules. It was developed by the International Electrotechnical Commission (IEC) specifically for motion control and is defined by the IEC 1491 standard. SERCOS supports data rates up to 16 Mbits/sec over a fiber optic ring. Data can be transmitted deterministically in real time based on the loop update rate (cyclic), or at lower bandwidths for less critical operations (non-cyclic). A SERCOS-compatible communication ring must have a single controller (master) and 1 - 254 drive or I/O modules (slaves).

Also, data communication can be performed in either a synchronous or asynchronous manner. The protocol allows the user to configure the communication telegrams to send whatever data is appropriate synchronously. Data that has not been configured to reside in the communication telegrams can be sent or retrieved asynchronously by use of a Service Channel contained within the communication telegrams. Generally, synchronous data is data that is critical to real-time operation (e.g., command and feedback data, status).

SERCOS is a unidirectional serial communications protocol for connecting multiple drives (Slaves) to a motion controller (Master) over a fiber optic ring, in an industrial environment. The control and status information is organized into telegrams, and travels in a serial data stream around the SERCOS ring. All messages are synchronized according to the SERCOS cycle time, the timing of which is configured during initialization by the Master. Each Slave on the ring repeats the telegrams sent to it, sending them to the next device on the ring, and inserting its own telegram into the designated time slot in the serial data stream.

Starting at the output port of the Master, the devices are connected in the ring in a daisy-chain fashion, connecting from the output port of one device to the input port of the next device, and so on, until the ring is closed back at the input port of the Master. Up to 254 drives can be connected on a SERCOS ring, although the systems requirements for update rates and data will usually limit the number of devices to many fewer than that. SERCOS networks can operate at 2, 4, 8 or 16 Mbit/sec. Maximum distances from input port to output port can be 60 meters (plastic fiber) to 250 meters (glass fiber).



TOP

Operation Modes

The SERCOS communication interface supports three main operation modes (Torque, Velocity and Position). The operation mode defines the real-time digital messages sent between the controller and the drive(s).

Mode	To Drive	To Controller
Torque	16 bit Command Torque	32 bit Actual Position
Velocity	32 bit Command Velocity	32 bit Actual Position
Position	32 bit Command Position	32 bit Actual Position

In addition to the main operation modes SERCOS supports several variations. Since the communication interface is determined by the firmware/software in the controller and the firmware/software in the drive, the real-time data is configurable.

Currently the SERCOS/XMP Series firmware/software supports several operation modes. Some operation modes are drive specific while others are drive independent. Motion Engineering is constantly testing and certifying compatibility between our controller and SERCOS-compatible drives.

Most drives support the three main operation modes (Torque, Velocity and Position). Please consult your drive specific documentation regarding supported operation modes and variations.

In all modes, the controller calculates a 32-bit command position every sample. The command position is based on the current command jerk, command acceleration and command velocity.

In Torque mode, the controller sends a 16-bit command torque to the drive. The drive sends a 32-bit actual position back to the controller. Every sample, the controller calculates a new command torque based on the position error and the control algorithm. The controller closes the position and velocity loop and the drive closes the torque loop.

In Velocity mode, the controller sends a 32-bit command velocity to the drive. The drive sends a 32-bit actual position back to the controller. Every sample, the controller calculates a new command velocity based on the position error and the control algorithm. The controller closes the position loop, and the drive closes the velocity and torque loop.

In Position mode, the controller sends a command position to the drive. The drive sends a 32-bit actual position back to the controller. Every sample, the controller calculates a new command position. The drive closes the position, velocity and torque loop.

TOP

Closed-Loop Tuning

A general difference between SERCOS digital drives and conventional analog drives is that digital drives have on-board intelligence and can close position or velocity loops within the drive. In all operation modes, “tuning” requires setting parameters in the drive and the controller. Thus, an understanding of both the controller and drive control algorithm is necessary for successful drive tuning.

The controller tuning parameters can be set using Motion Console (for Windows-based systems). For more information on the controller’s tuning procedures, please consult the Tuning section in the *XMP Motion Controller Hardware Installation* manual.

The drive’s tuning parameters must be set via IDNs, and the values are determined from information supplied by the drive manufacturer. Please consult the drive-specific documentation for more information on the drive’s tuning parameters. While Motion Engineering has considerable experience with the listed drives and can generally offer tuning guidelines for drive parameters, difficult tuning situations may require support from the drive vendor or manufacturer.

In Velocity mode, the controller's CONTROL ALGORITHM output controls the motor's velocity. Therefore, the drive's velocity loop must be tuned and the controller's position loop must be tuned. The tuning procedure is identical to a standard analog output XMP Series controller connected to a velocity-controlled amplifier. Note that the controller's Velocity Feed Forward term is very useful in velocity-controlled systems.

In Position mode, the controller's control algorithm is not used. The drive is responsible for the closed loop control. Therefore, the drive's velocity and position loop must be tuned. The controller's tuning parameters have no effect on the system's response.

TOP

Data Transmission

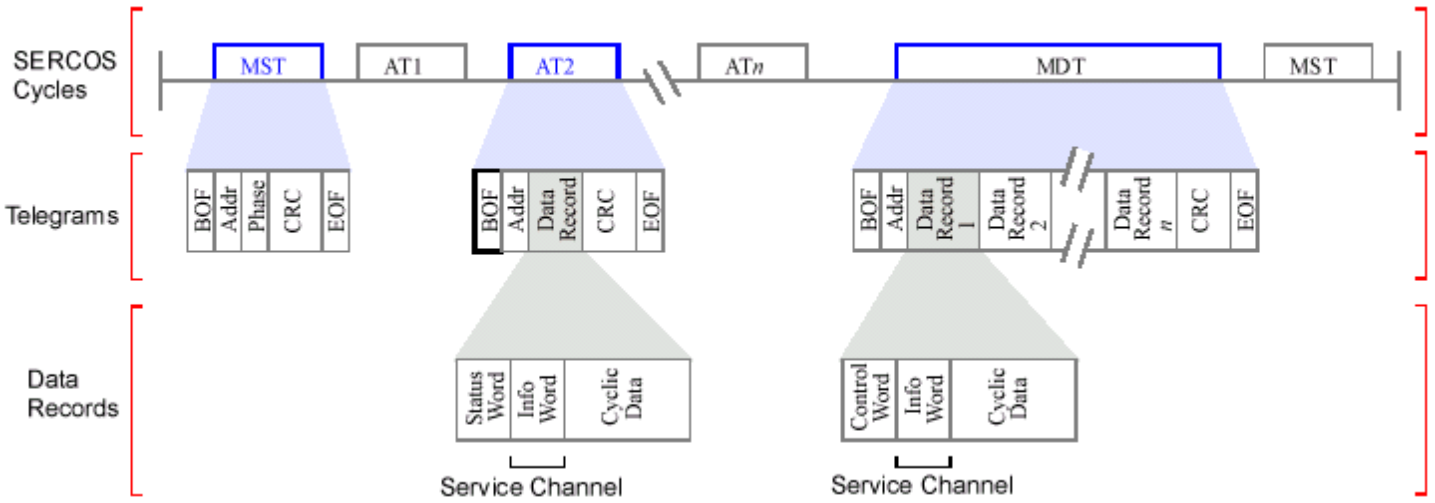
SERCOS supports two types of data transmission, cyclic and non-cyclic.

Cyclic data is the critical real-time synchronized data sent between the Master and the Slaves (drives, I/O modules). In every SERCOS cycle, the Master sends and receives fixed-length messages to the drives and I/O modules. These messages contain the motion control command signal and the feedback response for each drive or the digital I/O commands and responses for each I/O module. The cyclic data is guaranteed to reach each drive and I/O module and return to the Master at a fixed time interval, the SERCOS cycle.

Non-cyclic data is the noncritical asynchronous data. The cyclic fixed-length messages have space (called the Service Channel) reserved for non-cyclic data. In each SERCOS cycle, the Master may transmit two bytes of non-cyclic data through the Service Channel to each Slave. Note that it may require several SERCOS cycles for the Master to complete the transmission of the non-cyclic data to the Slaves. Typically, transmitting non-cyclic data is much slower than cyclic data.

SERCOS cycles are built using telegrams, which in turn contain data records for all of the Slave drives. Cyclic data is transferred in the cyclic data part of the data records. Non-cyclic data is transferred in the Service Channel of data records. Refer to the next figure.

TOP



Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

Copyright © 2002
Motion Engineering

Sercos- Data Types

Summary | Data Block Structure | Data Block Structure of IDNs

- Element 1: IDNumbers
- Element 2: Name of Operation Data
- Element 3: Attributes of Operation Data
- Element 4: Operation Data Unit
- Element 5: Minimum Input Value of Operation Data
- Element 6: Maximum Input Value of Operation Data
- Element 7: Operation Data

Summary

And datas in a slave that can be accessed by the host is represented by a data block. Data blocks are assigned identification numbers (IDNs or “ID numbers”). The data block consists of information about the data; for example, IDNumber, name, attributes, units, min and max values, and the operation data itself.

TOP

Data Block Structure

The SERCOS protocol is designed to handle many different types of data, characterized by two fundamental types: fixed length data and variable length data.

Fixed length data is either 2 bytes or 4 bytes wide and can be used to represent signed or unsigned integers, hexadecimal values, binary codes, IDNumbers (identification numbers) of other Data Blocks, and procedure commands.

The length of **variable length** data depends on what type of data is present, and is defined by the first two words (32 bits), which specify the actual and maximum length of the data. Variable length data can be used to represent character strings, lists of IDNumbers of Data Blocks, lists of signed or unsigned integers (both 2 or 4 bytes wide), lists of hexadecimal values, etc.

All data (fixed and variable length) can be sent or received via the Service Channel. However, **only** fixed length data is allowed to be configured into a communication telegram (MDT, AT). When a communication telegram is configured to send or receive fixed length data, it is only Element 7 data that will be either sent or received.

TOP

Data Block Structure of IDNs

Data is accessed through data blocks referred to as IDNs. An IDN consists of seven elements:

Element 1	IDNumber
Element 2	Name
Element 3	Attribute
Element 4	Unit
Element 5	Minimum Input
Element 6	Maximum Input
Element 7	Operation Date

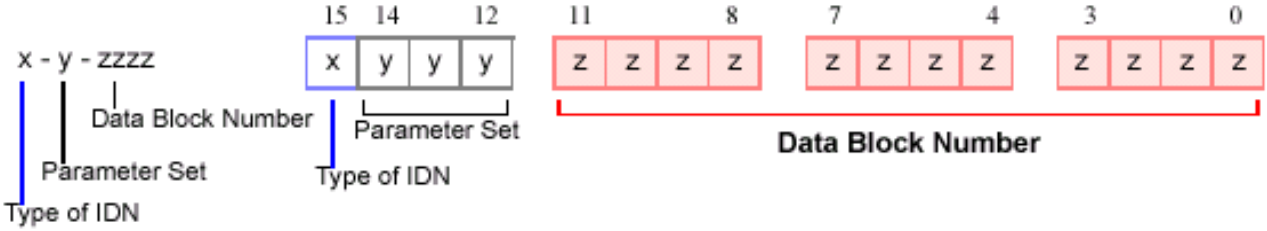
All data exchanged between Master and Slaves has an IDNumber (IDN number) assigned to it. Every IDN has an associated data block which consists of seven elements. The Master can only write Element 7 data; the Master cannot write Elements 1 - 6. Elements 1 - 6 are defined by the drive itself.

Element	Description	Data Type
1	IDNumber (Identification number)	**binary** Expressed as either S-X-XXXX or P-X-XXXX. S denotes IDNs that are defined by the SERCOS Specification. P denotes IDNs that are defined by the manufacturer of the device.
2	Name	**variable length string** Contains the name of the IDN.
3	Attribute	**32 bit binary** Contains information about conversion factors and data representation (signed or unsigned integer, fixed or variable length data, etc.)
4	Unit	**variable length string** Contains a representation of the units for the data.
5	Minimum input value	1 or 2 words
6	Maximum input value	1 or 2 words
7	Operation Data	1 or 2 words, or string

TOP

Element 1: IDNumbers

Each Data Block has a number assigned to it for identification purposes, called the IDNumber. The IDNumber is represented as either S-X-XXXX or P-X-XXXX. S denotes a Data Block that is defined by the SERCOS Specification. P denotes a Data Block that is defined by the manufacturer. The first 'X' identifies the "data set" that the Data Block belongs to. According to the SERCOS Specification, it is possible to switch between data sets.

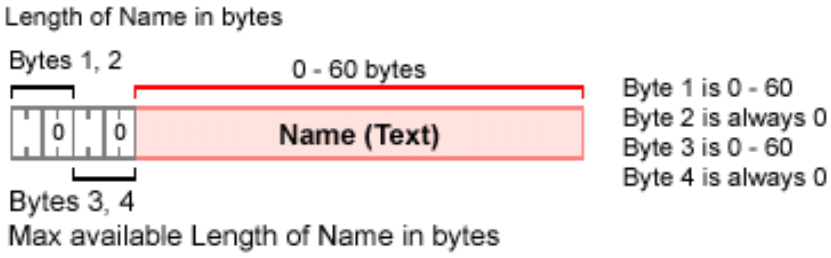


Bits	Name	Values
15	Type of IDN	0 S - Standard data 1 P - Product-specific data as the "S" or "P" part of the IDN notation S-y-zzzz or P-y-zzzz.
14-12	Parameter Set	0 - 7, as the "y" part of the IDN notation S-y-zzzz or P-y-zzzz.
11-0	Data Block Number	0 - 4095, as the "zzzz" part of the IDN notation S-y-zzzz or P-y-zzzz.

TOP

Element 2: Name of Operation Data

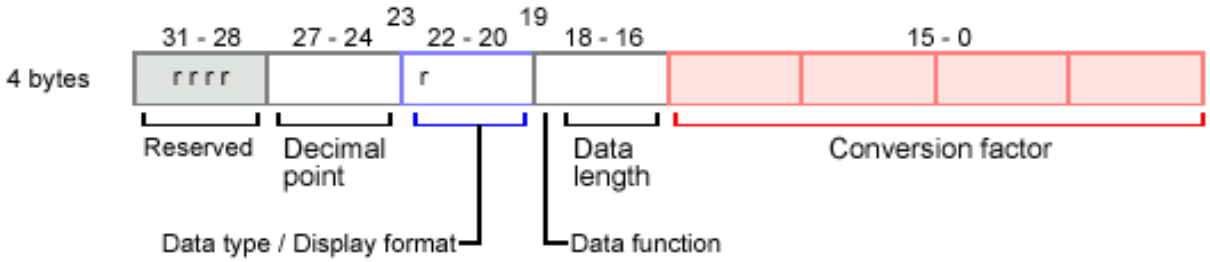
The name of operation data is 64 bytes maximum, with 2 length specifications of 2 bytes each, and a character string of 60 characters maximum. Bytes 1 and 2 contain the number of characters in the text. Bytes 3 and 4 contain the maximum number of characters in the text. Since this element is READ-Only, bytes 3 and 4 will contain the same values as bytes 1 and 2.



TOP

Element 3: Attributes of Operation Data

Every data block has an attribute (4 bytes) which contains all of the information required to display operation data, using universal routines. If data needs to be scaled (to be displayed), then specific scaling parameters are supplied in the attribute.



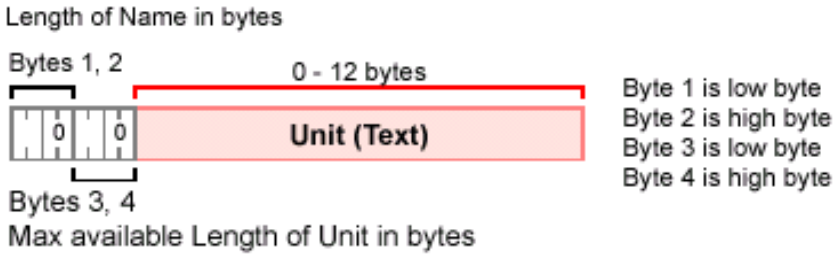
Bits	Name	Values	
31	Reserved		
30	Phase 4 write-protection bit.	1 = write-protected	
29	Phase 3 write-protection bit.	1 = write-protected	
28	Phase 2 write-protection bit.	1 = write-protected	
27-24	Number of places after the decimal point Indicates the position of the decimal point in the data to be displayed. Basically, it's the exponent "x" in 10-x.	000 No places after decimal point 001 1 place after decimal point 010 2 places after decimal point * 111 15 places after decimal point	
23	Reserved		
22-20	Data Type & Display Format Used to convert the operation data, and min/max input values to the correct display format.	**Data Type** 000 binary number 001 unsigned integer 010 integer 011 unsigned integer 100 extended char set 101 unsigned integer 010 Reserved 111 Reserved	**Display Format** binary unsigned decimal signed decimal hexadecimal text IDN number
19	Function of Operation Data Indicates whether this data calls a procedure in a drive	0 Operation data or parameter 1 Procedure command	

18-16	Data Length The data length is required so that the Master is able to complete Service Channel data transfers correctly.	**000 Reserved** 001 Operation data is 2 bytes long 010 Operation data is 4 bytes long **011 Reserved** 100 Variable length with 1-byte data strings 101 Variable length with 2-byte data strings 110 Variable length with 4-byte data strings **111 Reserved**
15-0	Conversion Factor	An unsigned integer used to convert numeric data to display format. Is set to 1 when it is not needed for data display (e.g., binary display or a character string).

TOP

Element 4: Operation Data Unit

The operation data unit is 16 bytes maximum, with 2 length specifications of 2 bytes each, and a character string of 12 characters maximum. Bytes 1 and 2 contain the number of characters in the text. Bytes 3 and 4 contain the maximum number of characters in the text. Since this element is READ-Only, bytes 3 and 4 will contain the same values as bytes 1 and 2.



TOP

Element 5: Minimum Input Value of Operation Data

The minimum input value is the smallest numerical value for operation data that the drive can process. When the Master writes a value to the drive that is less than the minimum value, the drive ignores it and continues to use the previous operation data.

When the operation data is of variable length or a binary number is used, there is no minimum input value of operation data.

TOP

Element 6: Maximum Input Value of Operation Data

The maximum input value is the largest numerical value for operation data that the drive can process. When the Master writes a value to the drive that is more than the maximum value, the drive ignores it and continues to use the previous operation data.

When the operation data is of variable length or a binary number is used, there is no maximum input value of operation data.

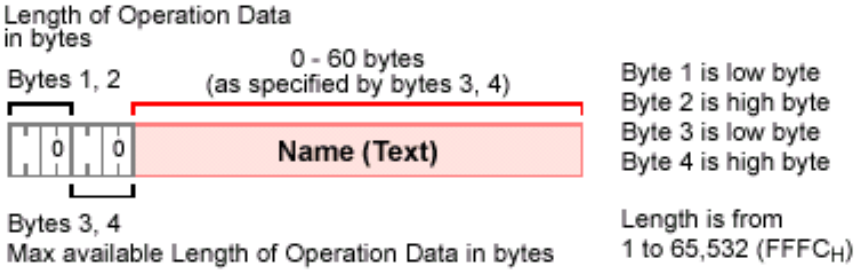
TOP

Element 7: Operation Data

In terms of length, there are 3 types of operation data:

- fixed length with 2 bytes
- fixed length with 4 bytes
- variable length up to 65,532 bytes in 1 byte (char), 2 byte, or 4 byte values

Bytes 1 and 2 contain the number of bytes in the text. Bytes 3 and 4 contain the max number of bytes available in the text. Data in the text may be 1 byte, 2 bytes, or 4 bytes wide.



TOP

Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

NEXT

Copyright © 2002
Motion Engineering

Sercos- Communications

Synchronization | Ring Timing | Initialization

Synchronization

The Master is responsible for sending a synchronization telegram (MST) at the beginning of each communication cycle. All Slaves (drives) will receive the MST and reset their clocks. In this way, all Slaves will run in phase lock with the Master's clock. Because all Slaves are in phase lock with the Master's clock, commands can be made active in all of the Slaves at the same instant. This means that the Master can coordinate motion between all axes without propagation effects distorting the motion profile. Feedback from the Slaves is handled in a similar manner, and is latched in all Slaves at the same instant.

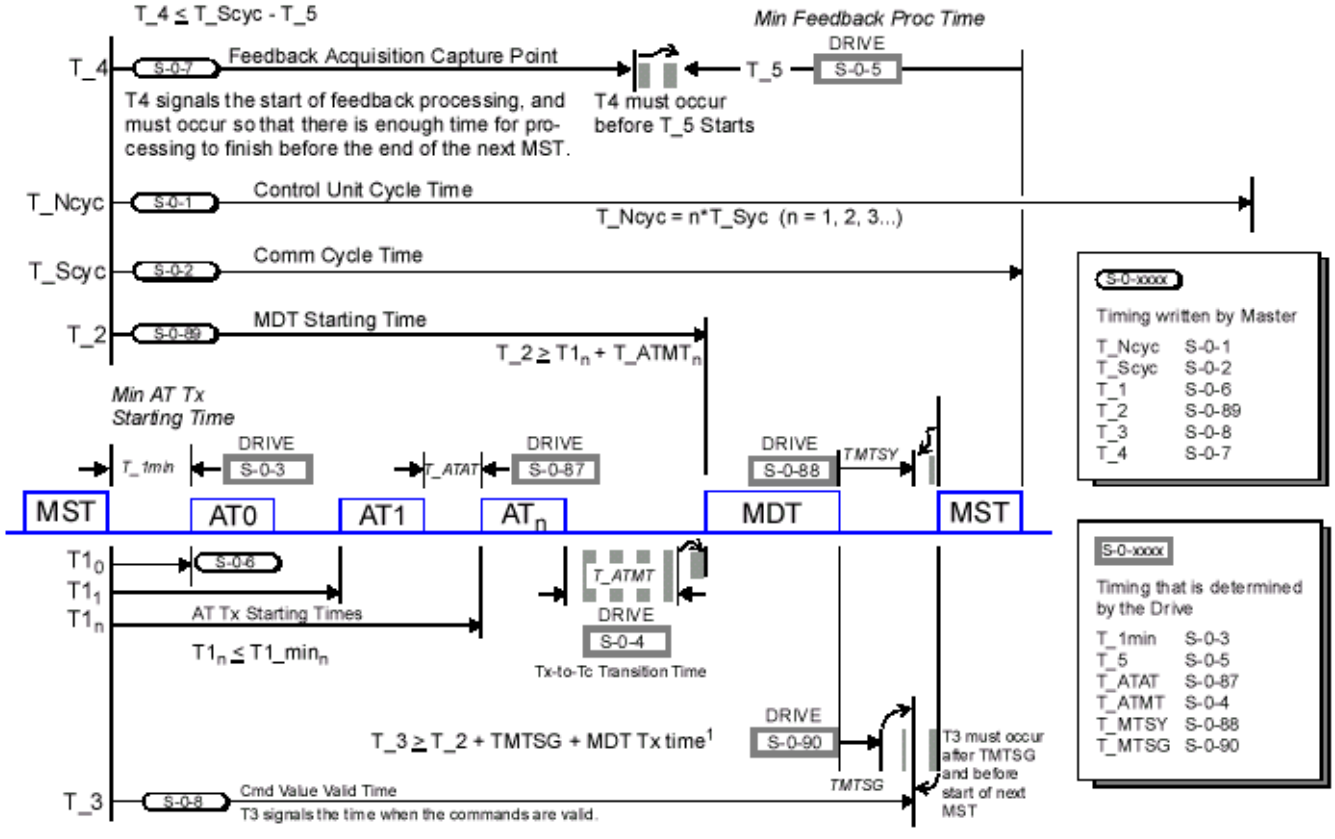
TOP

Ring Timing

Data is sent and received by the Master and Slaves through communication telegrams. The communication telegrams are organized over the SERCOS cycle in the manner shown in the next figure.

The SERCOS ring timing is based on the data to be placed in the telegrams (AT, MDT), and on 6 drive parameters that are determined by the type and features of a drive (or drives), and 6 parameters that are written from the Master, with some of these parameters derived or calculated from the drive's timing parameters. The times at which the MDT and AT are sent (relative to the sending of the MST) are determined by the Master and sent to the Slaves during initialization.

During Phase 2, the Master reads parameters from the drives that determine what and when the drives are able to transmit and receive. Using this information and the desired telegram contents, baud rate and cycle time, the Master determines the timing and telegram parameters for each drive. The Master then writes these parameters to the drives.



[1] *MDT Tx time* depends on the number of slaves and the amount of data sent to the slaves. The Master must calculate the *MDT Tx time* during Phase 2.

For a print-friendly version of the above diagram, click here.

TOP

Initialization

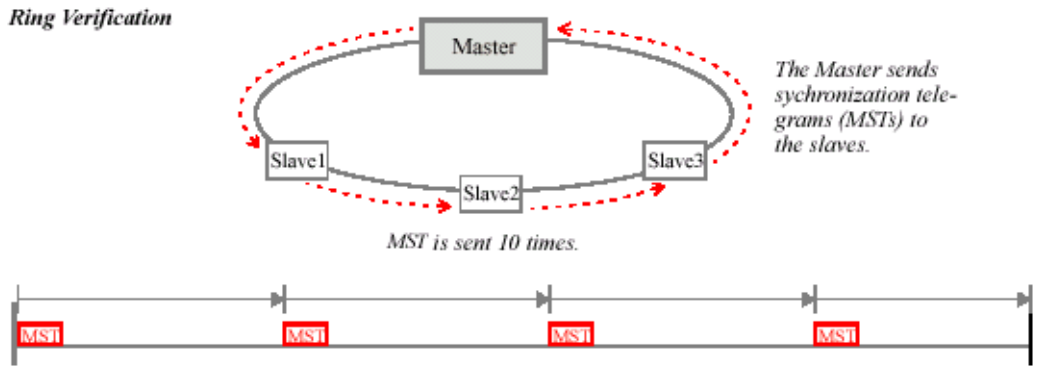
Before true synchronous data transmission can occur, the system must first be initialized. This is done through a series of **communication phases** (or just phases) in which data is first transmitted asynchronously. The data transmitted during these early phases is used to configure the Master and Slaves for synchronous data transmissions in later phases. The SERCOS protocol defines five phases.

Phase	Name	Action
0	Ring Verification	Master verifies ring closure
1	Device Verification	Master verifies devices on ring
2	Telegram Set-Up	Master reads timing data from slaves and sets up telegram timing
3	Device Parameterization	Master continues to configure devices
4	Cyclic Operations	Master commands devices cyclically

On power-up, each drive or I/O module begins an initialization sequence. At this time, each drive and I/O module operates as a repeater, by simply passing received telegrams to the next device on the SERCOS ring.

The Master is only allowed to set the communication phase to the next logical communication Phase **(Phase 0 -> Phase 1 -> Phase 2 -> Phase 3 -> Phase 4)** or directly back to **Phase 0**. If at any time the Master attempts to switch a Slave into a Phase that is not the next logical Phase, then the Slave will immediately return to Phase 0. If at any time the Slave receives two invalid MSTs or MDTs consecutively, the Slave will also switch to Phase 0.

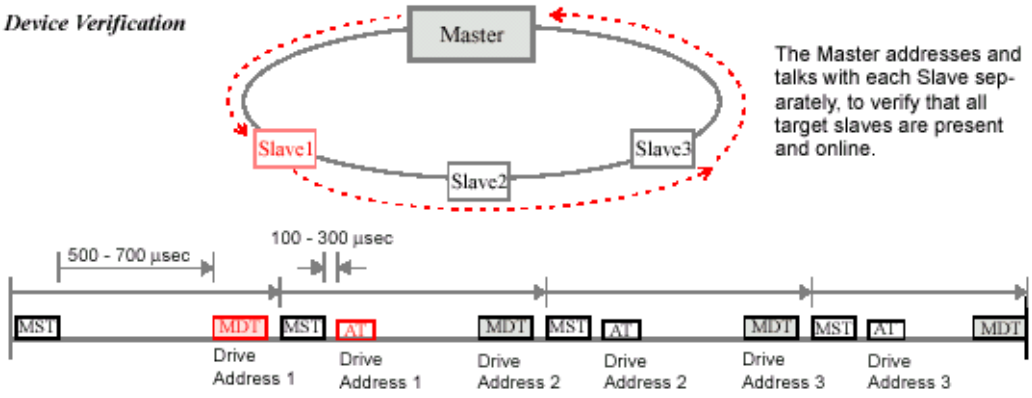
Phase 0:



During **Phase 0** **no data is exchanged** between the Master and the Slaves. All Slaves must be in “repeater” mode. This means that each Slave will retransmit any signal that it receives. In order to verify that the communications ring is intact and capable of sending telegrams, the Master begins sending Master Synchronization Telegrams (MST) through the SERCOS ring.

The Slaves (drives) simply pass the MST to the next drive in the daisy-chained ring, and eventually because of the ring topology, the MST returns to the Master (i.e., Master will receive its own MST). Phase 0 is completed when the Master receives 10 consecutive MSTs. After the tenth consecutive MST, the Master changes the phase information in the MST to 1, which commands all Slaves to switch into Phase 1 operations.

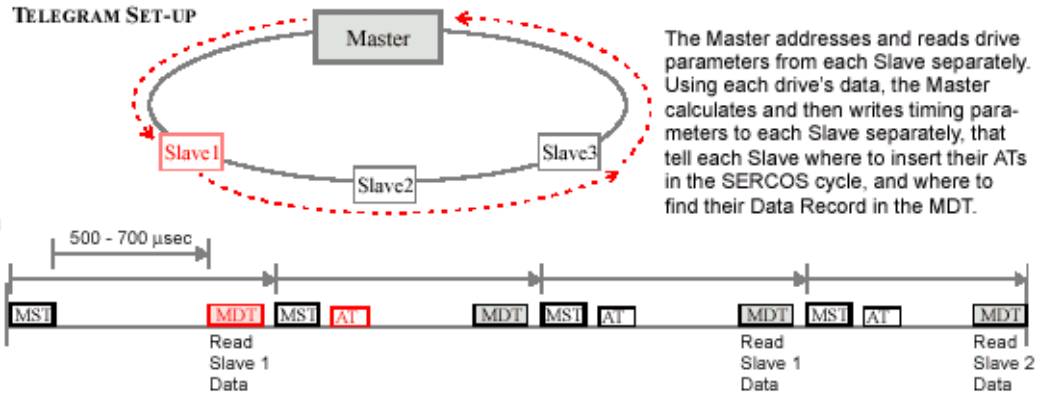
Phase 1:



During **Phase 1**, the Master sends out an MDT with the address of a specific Slave in the system. If present within the system, the Slave with the specified address responds by sending an AT back to the Master. The AT that is sent is rudimentary and is intended solely as a confirmation that the addressed Slave is in the system. The Master then repeats this query for all target Slaves. (Note that not all Slaves in the system will be target Slaves. Target Slaves are chosen by the application before initialization begins.)

When the Master receives an AT from each target Slave (in response to a query), the Master changes the phase information in the MST to 2, which commands all Slaves to switch into Phase 2 operations.

Phase 2:



During **Phase 2**, the Master sets up the configurable data portion and calculates the duration and time slots within the SERCOS cycle for all telegrams to be used in Phases 3 and 4. The Master also determines the slave operation mode for all Slaves. In order to do this, the Master requires certain data from the Slaves. The Master obtains this data by sending an MDT addressed to a specific Slave, that uses the MDT's Service Channel to query the Slave for the required data

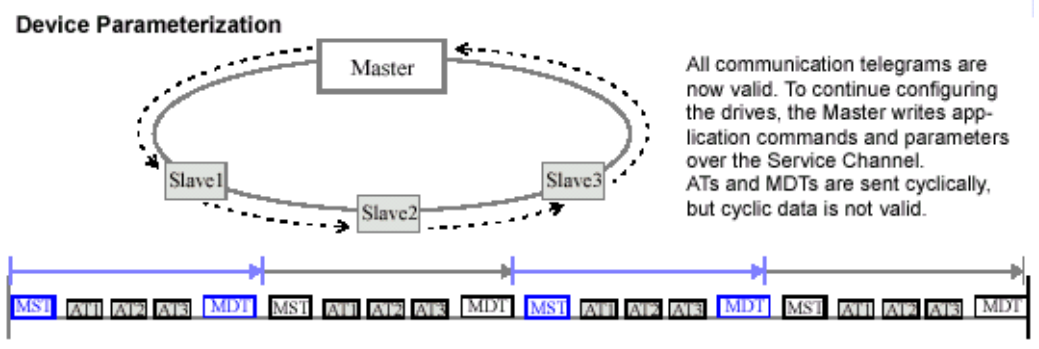
The Slave responds by sending an AT containing the appropriate data, in the AT's Service Channel. Once the Master has determined all parameters, it sends them to each Slave via the MDT's Service Channel. In Phase 2, the Service Channel is active in both MDTs and ATs.

Once all data is transmitted to the Slaves, the Master will initiate the Communication Phase 3 Transition Check S-0-127 procedure for each Slave.

The Communication Phase 3 Transition Check checks the validity of all the data, and if all the data is valid, then the procedure executes successfully. If any data is not valid, the procedure fails and the IDN number of the invalid data is placed in IDN-List of Invalid Operation Data for CP2 S-0-21.

After all Slaves have completed the procedure successfully, the Master changes the phase information in the MST to 3, which commands all Slaves to switch into communication Phase 3.

Phase 3:



In **Phase 3**, the real-time synchronous and asynchronous communication starts, and the Master uses the Service Channel to configure and parameterize (write parameters to) the Slaves. The parameters set in Phase 3 are application-oriented (e.g., conversion factors). During Phase 3, all communication telegram parameters sent in Phase 2 become active. Note that although Configurable Data (in ATs and MDTs) is present in the communication telegrams, some of that Configurable Data may not be valid until Phase 4.

The Master will still send the MST at the beginning of the SERCOS cycle, but will now also send an MDT with a global address at a specific time (all Slaves will receive a global telegram). Each Slave will transmit its AT during its specified time slot. The communication telegrams now contain the Control/Status Word, a Service Channel, and the Configurable Data.

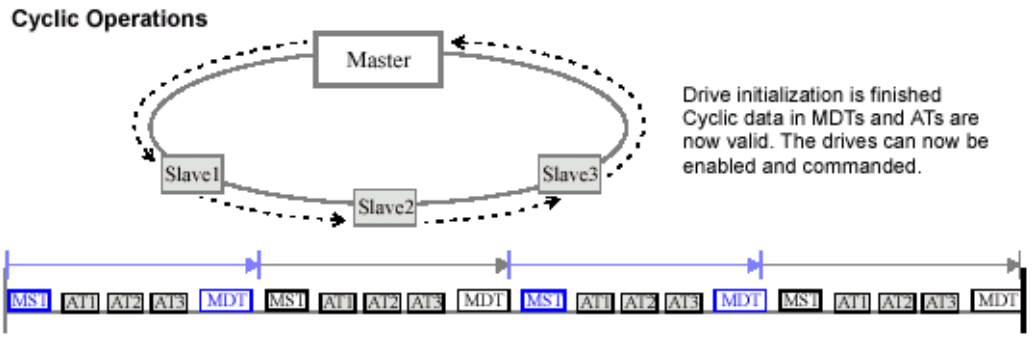
When the Master has finished sending parameters to the Slaves, it will initiate the Communication Phase 4 Transition Check S-0-128 procedure for each Slave.

The Communication Phase 4 Transition Check checks the validity of all the data, and if all the data is valid, then the procedure executes successfully. If any data is not valid, the procedure fails and the IDN number of the invalid data is placed in IDN-List of Invalid Operation Data for CP3 S-0-22.

TOP

If the Communication Phase 4 Transition Check executes successfully and the Phase information in the MST is equal to 4, the Master switches the drives to Phase 4.

Phase 4:



In **Phase 4**, a final verification of error-free drive operation is completed. This completes the drive initialization. The SERCOS communication loop is now operational. During Phase 4, all Control/Status words, Service Channels, and Configurable Data (in ATs and MDTs) are valid for the target Slaves. The Slaves (drives) are ready to follow commands when enabled. Diagnostics (errors, warnings, status) are enabled.

Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

NEXT

Copyright © 2002
Motion Engineering

Sercos- Procedures

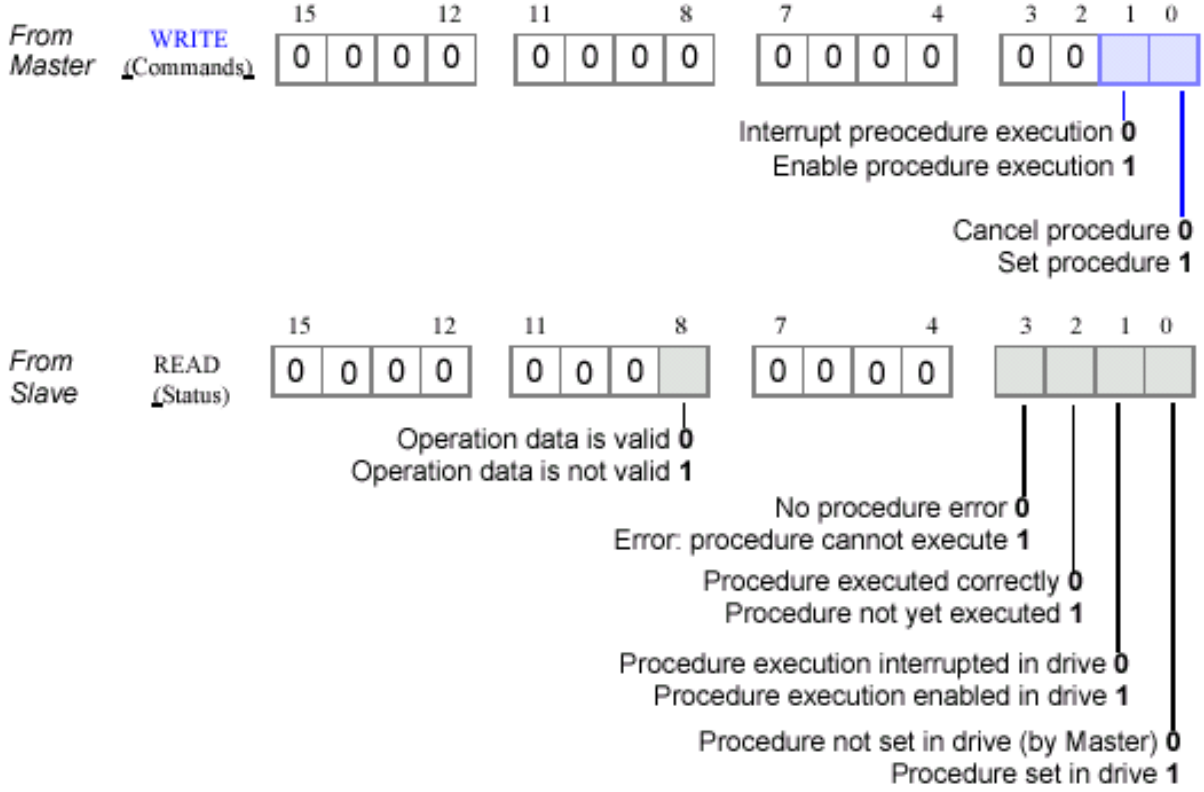
Summary

Many Slaves (drives) come with preprogrammed procedures that the Slave is able to execute without assistance from the Master. Each of these procedures is assigned a Data Block. The Master controls a procedure by reading and writing to Element 7 of the procedure's Data Block. The Master can initiate, interrupt, or cancel a procedure at any time by setting or clearing bits in Element 7 of the procedure's Data Block.

Bits 0 and 1 are responsible for respectively setting and enabling the procedure command. Once set and enabled, the Slave will begin execution of the procedure. To indicate that the procedure is executing, the Slave sets bit 2 of the procedure status word. When the procedure has finished executing, the Slave clears bit 2 of the procedure status word and sets the Procedure Command Change Bit in the Drive Status Word S-0-134. If an error has occurred when executing the procedure, the Slave will set bit 3 of the procedure status word and will also set the Procedure Command Change Bit in the Drive Status Word S-0-135. In either case (successful execution or failure), the Master must cancel the procedure by clearing bits 0 and 1 of the procedure.

In SERCOS, in order for a Master to execute a procedure,

1. The Master writes 0x3 to Element 7 of the desired IDN procedure, which sets and enables the procedure.
2. The Master reads the procedure status word and checks bits 2 & 3 to see if the procedure has finished executing or if there is an error.
3. After reading that the procedure has executed, the Master writes 0x0 to Element 7 of the particular procedure, which cancels the procedure.



TOP

Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

Copyright © 2002
Motion Engineering

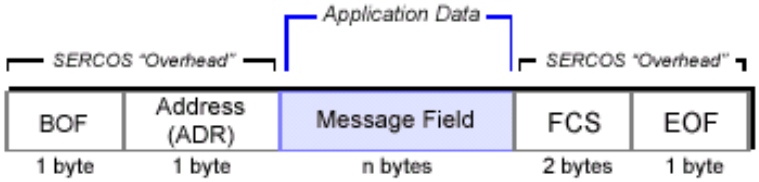
Sercos- Telegrams

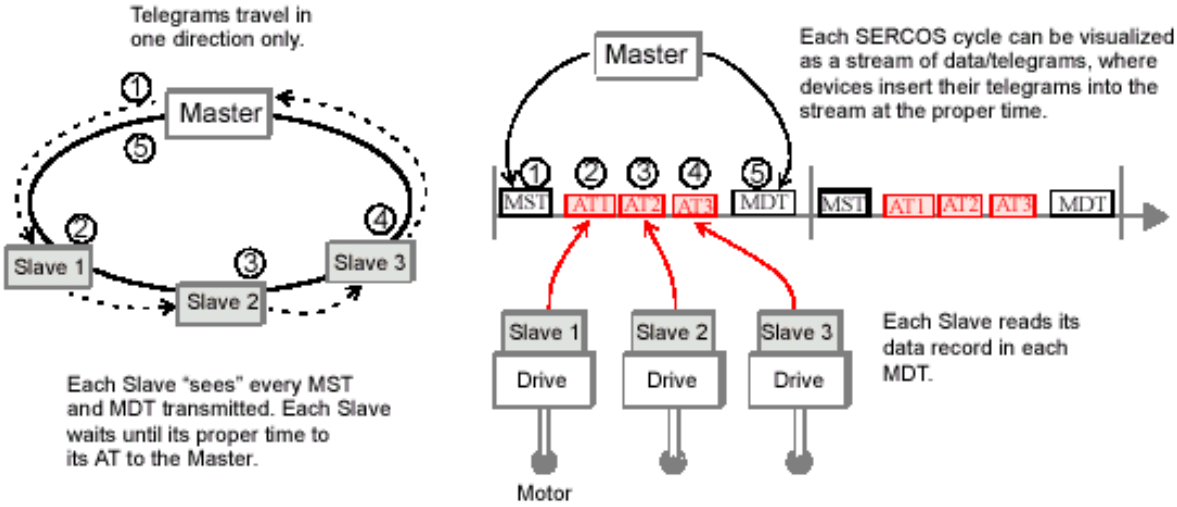
Summary | BOF Delimiter | ADR Target Addresses | Message Field
FCS (Frame CHeck Sequence) | EOF (End of Frame) Delimiter | Data Record
Master Synchronization Telegram | Master Data Telegram | Amplifier Telegram

Summary

The controller (Master) communicates with the drives and I/O modules (Slaves) using telegrams. A telegram is a structure that contains data, error checking and handshaking information. SERCOS supports three types of telegrams: Master Synchronization Telegram (MST), Master Data Telegram (MDT), and Amplifier Telegram (AT).

Each type of telegram contains 5 types of fields: BOF (beginning of frame), ADR (address), message field, FCS (frame check sequence), EOF (end of frame).

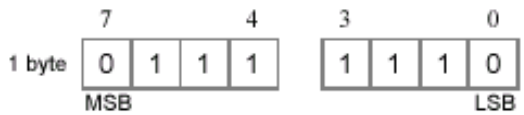




TOP

BOF Delimiter

All telegrams have a BOF (beginning of frame) byte denoting the beginning of the telegram. The BOF is always 0111 1110.



TOP

ADR Target Addresses

All telegrams have an ADR (address) byte, which denotes the address of a drive (Slave). In a telegram from the Master, the address specifies which drive the information is for. In a telegram from a Slave (drive), the address specifies the sourcing drive. The target addresses for the drives are valid if greater than 0 and less than 255. Typically, a drive's address is set using a selector located physically on the drive.

Address 0 is the "no station" address, and is sometimes used to remove a drive from the ring logically, during troubleshooting. During non-cyclic operations (Phases 0 - 2), the Master can only communicate with one drive per cycle. During cyclic operations (Phases 3, 4), the Master can communicate with all drives.

From	**To**	**Telegram**	**Non-Cyclic** (Phases 0,1,2)	**Cyclic** (Phases3,4)
Master	Slave	**MST**	255	255
Master	Slave	**MDT**	$1 \leq ADR \leq 255$	255
Slave	Master	**AT**	$1 \leq ADR \leq 254$	$1 \leq ADR \leq 254$

TOP

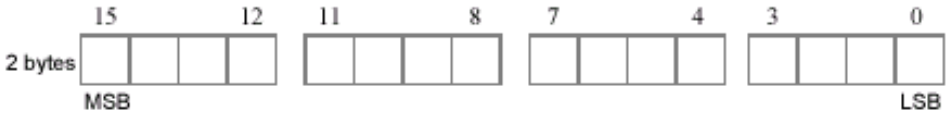
Message Field

Each Telegram (MST, AT, and MDT) contains a message field. The Message Field for the MST consists of one 8-bit word. The lower three bits of the MST contain Phase information. The Message Field for the AT consists of one Data Record. The Message Field for the MDT consists of one Data Record if the MDT's target address is a specific drive. If the MDT's Target address is all drives (255), then the Message Field for the MDT consists of one Data Record per drive in the system.

TOP

FCS (Frame Check Sequence)

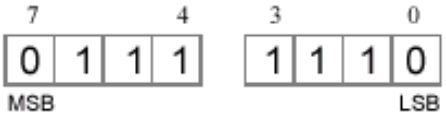
All telegrams have a two-byte FCS number used to check data integrity. The frame check sequence (16 bits) is implemented according to ISO/IEC 3309, 4.5.2.



TOP

EOF (End of Frame) Delimiter

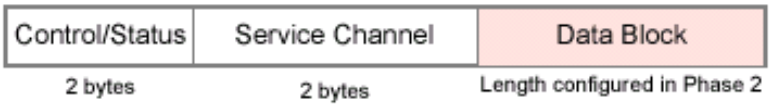
All telegrams have an EOF byte denoting the end of the telegram. The EOF is always 0111 1110.



TOP

Data Record

Data Records are used by both the Amplifier Telegram (AT), and the Master Data Telegram (MDT) to send data. Generally, a Data Record consists of a 16-bit Control/Status word, a 16-bit Service Channel, and a Data Block of 16-bit words.



Amplifier Telegrams contain only one Data Record, because each Slave sends its own Amplifier Telegram.

During Phase 2, the Master sends a Master Data Telegram that is addressed to a specific Slave. Since only one Slave is to receive the MDT, the Phase 2 MDT has only one Data Record. During Phases 3 and 4, the Master sends a Master Data Telegram that is received by all Slaves on the ring. Since all Slaves receive the Master Data Telegram, the Phase 3-4 MDT has one Data Record for each Slave.

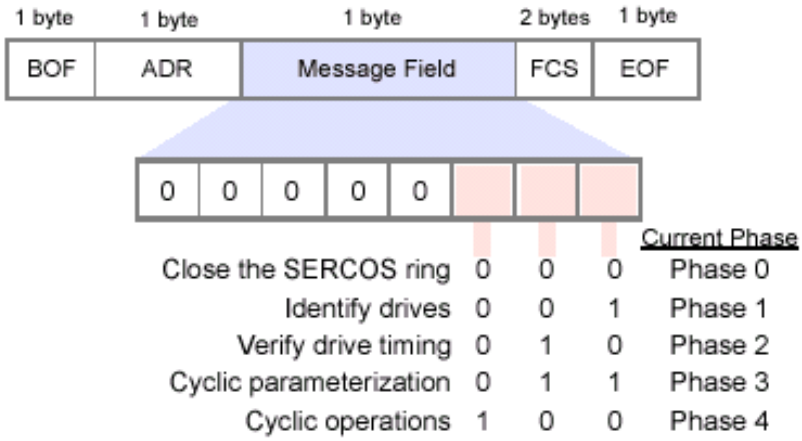
During Phase 2, the Data Block in both the AT and the MDT has a length of 0. This means that all data exchange between Master and Slave must take place using the Service Channel. Because the Service Channel is only 16-bits wide, data exchange can take multiple cycles. Service Channel exchange of data is referred to as **non-cyclic**.

During Phase 2, the Master is responsible for determining the data fields for the Data Blocks in the Phase 3-4 AT and MDT. The size and number of data fields will determine the size of the Data Block. Once defined in Phase 2, the fields and therefore the size of the Data Block is fixed. The data fields in the MDT usually contain command data. The data fields in the AT usually contain feedback data. Because the Data Blocks are configured in Phase 2, there is no overhead using the Data Blocks to send data in Phases 3 and 4. Data exchange using the Data Blocks is referred to as **cyclic**.

TOP

Master Synchronization Telegram

The Master Controller uses master synchronization telegrams (MST) to coordinate its transmission cycle timing with the Slaves. The Master initiates a SERCOS cycle by transmitting an MST to all of the drives and I/O modules on the ring.The MST message field contains one 8-bit word, of which the three lowest bits determine the communication phase of the system. The MST is sent at the beginning of the SERCOS cycle in all communication phases.



Field	Bytes	Description
BOF	1	Beginning of Frame. The BOF marks the start of a telegram.
ADR	1	The target address. In the MST, ADR = 255 (the broadcast address) during Phases 3 and 4.
Current Phase	1	The lower 3 bits designate the SERCOS phase (0 - 4)
FCS	2	Frame check sequence. The FCS field contains circular redundancy check (CRC) information.
EOF	1	End of Frame. The EOF marks the end of the telegram.

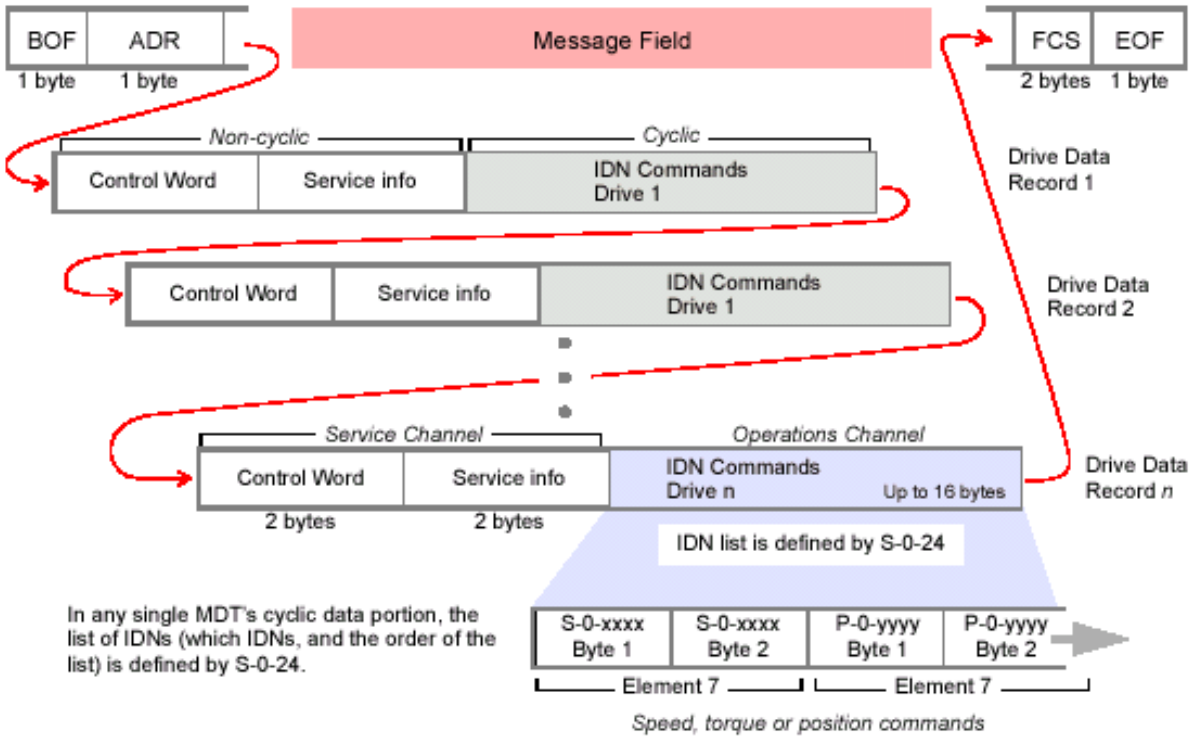
TOP

Master Data Telegram

During Phase 2, the Master must communicate with the Slaves in order to configure them for operations in Phases 3 and 4. In order to send or request data, the Master will send a Master Data Telegram (MDT) to a specific Slave (the Slave is addressed explicitly). Since only one Slave is to receive the MDT, the MDT Message Field contains only one Data Record, where the length of the Data Block (inside the Data Record) is 0, i.e., the data block is empty. During Phase 2, the Master sends data to each Slave via the Service Channel.

During Phase 2, the Master informs the Slave of the byte offset into the MDT at which the Data Record resides for that Slave. The length (in bytes) of the Data Block within each Data Record depends on the data fields configured by the Master during Phase 2. The data fields within the Data Block usually contain command information.

During Phases 3 and 4, once per SERCOS cycle the Master sends a Master Data Telegram (MDT) that has a **global** (broadcast) address. Because the global address is used, all Slaves receive the MDT. Since all Slaves will receive the MDT, the MDT Message Field contains one Data Record for each Slave in the system. During Phases 3 and 4, data is sent to a Slave by using the Data Blocks (cyclic) or the Service Channel (non-cyclic).



Field	Size (bytes)	Description
BOF	1	Beginning of Frame, which is always 0111 1110. The BOF marks the start of a telegram.
ADR	1	Target address. In the MDT, ADR = 255 (the broadcast address).
*Control Word	2	Control word for drive n. Contains operational data.
*Service Info	2	Contains the non-cyclic data for drive n.
*IDN Commands	Variable **	Contains the cyclic data for drive n.
FCS	2	Frame check sequence. Contains circular redundancy check (CRC) info.
EOF	1	End of Frame, which is always 0111 1110. The EOF marks the end of the telegram.

* These fields comprise the data record. There is one data record per drive in the MDT.
** The length if the IDN Commands field is determined during Phase 2

Bit	Name & Value	More Detail
15	0 Drive OFF 1 Drive ON	Bit 15-13=111, the drive should follow command values. When 1->0, the drive removes torque from the motor, and allows the motor to spin down.
14	0 Drive Disable 1 Drive Enable	When 1->0, torque is immediately disabled, independent of bits 15 and 13.
13	0 Drive Halt 1 Drive Restart	
12, 11	**Reserved**	
10	Control Unit Synchronization Bit	

9, 8	Operation Mode 00 primary op mode 01 secondary op mode 1 10 secondary op mode 2 11 secondary op mode 3	Defined byS-0-32 Defined by S-0-33 Defined by S-0-34 Defined by S-0-35
7	Real-time Control Bit 2	S-0-302
6	Real-time Control Bit 1	S-0-300
5, 4, 3	Data Block Element 000 Service channel not active, 001 IDN (number) of the op data 010 Name of operation data 011 Attribute of op data 100 Units of op data 101 Min input value 110 Max input value 111 Operation data	-Close service channel or break a transmission in progress -The service channel is closed for the previous IDN and opened for a new IDN.
2	0 Transmission in progress 1 Last transmission	
1	0 Read service info 1 Write service info	
0	Master Service Transport Handshake	A toggle bit

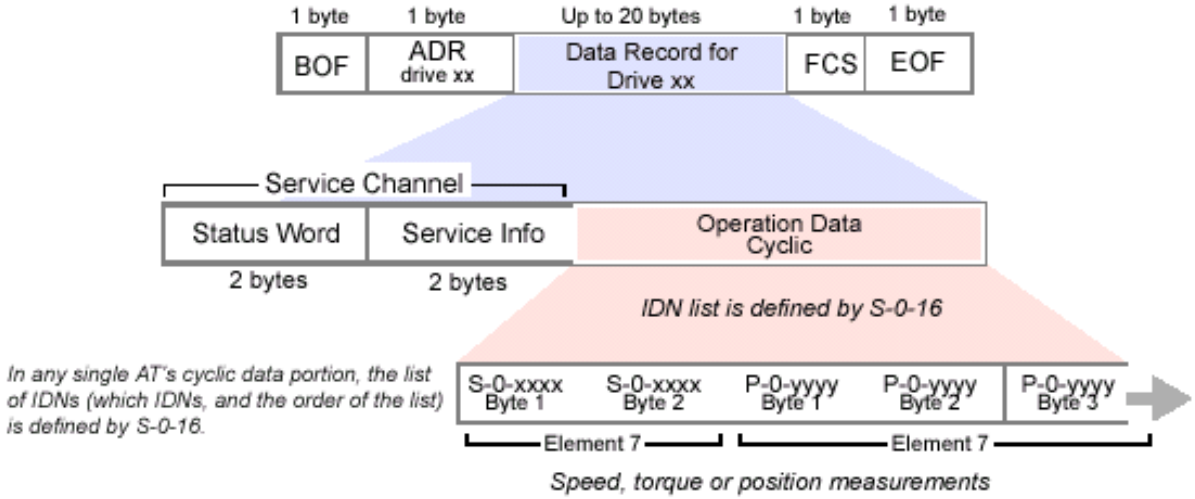
TOP

Amplifier Telegram

During Phase 2, the Slave sends an Amplifier Telegram (AT) only when it receives a Master Data Telegram (MDT) that contains that Slave's address. The AT contains one Data Record where the length of the Data Block is 0, i.e., the data block is empty. During Phase 2, the Master sends data to each Slave via the Service Channel.

During Phases 3 and 4, each Slave sends an Amplifier Telegram (AT) every SERCOS cycle, at the time designated by the Master during Phase 2. The AT Message Field contains one Data Record, where the length of the Data Block (inside that Data Record) is determined by the data fields configured by the Master during Phase 2. The data fields within the Data Block usually contain feedback and status information.

During Phases 3 and 4, data is sent to the Master by using the Data Blocks (cyclic) or the Service Channel (non-cyclic).



Field	Size (bytes)	Description
BOF	1	Beginning of Frame. The BOF marks the start of a telegram.
ADR	1	sends address.
Status	2	Control word for drive n. Contains operational data.
Service Info	2	Contains the non-cyclic data for drive n.
Operation Data	Variable *	Contains the cyclic data for drive n.
FCS	2	Frame check sequence. Contains circular redundancy check (CRC) info.
EOF	1	End of Frame, which is always 0111 1110. The EOF marks the end of the telegram.

* The length of the Operation Data field is determined during Phase 2.

Bit	Name & Value	More Detail
15, 14	Ready to operate 00 drive not ready for power-up 01 drive ready for power-up 10 drive power ready 11 drive ready to operate	
13	Drive Shutdown Error, Class 3 Diags	
12	Change Bit for Class 2 Diags	
11	Change Bit for Class 3 Diags	
9,8	Actual Operation Mode 00 primary op mode 01 secondary op mode 1 10 secondary op mode 2 11 secondary op mode 3	 Defined by S-0-32 Defined by S-0-33 Defined by S-0-34 Defined by S-0-35
7	Real-time Status Bit 2	See S-0-306
6	Real-time Status Bit 1	See S-0-304

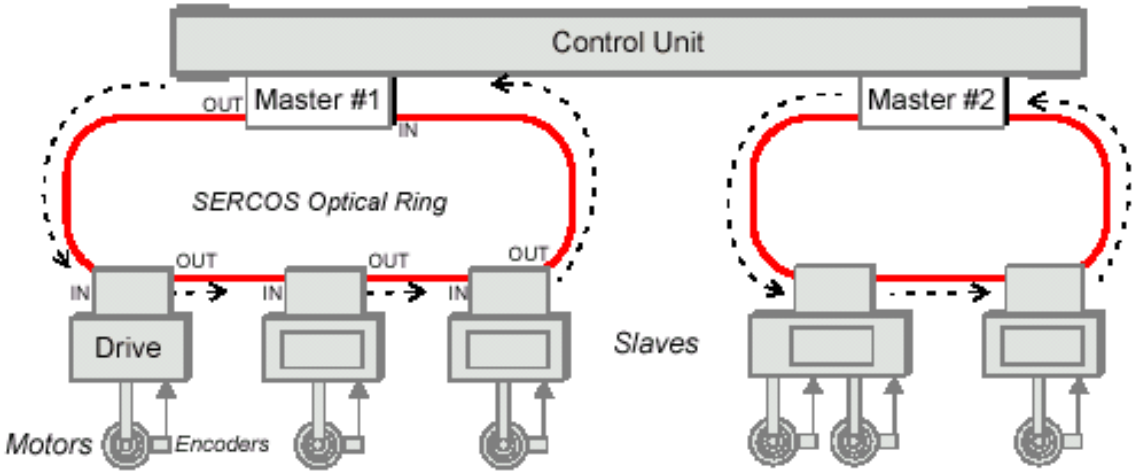
TOP

5	Change Bit Commands 0 No Change in Command Status 1 Changing Command Status	
4, 3	**Reserved**	
2	Error 0 No Error 1 Error in Service Channel	Error message is in drive's Service Channel.
1	Busy 0 Step Finished 1 Step in Progress	
0	Master Service Transport Handshake	A Toggle bit

Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

Copyright © 2002
Motion Engineering

Sercos- Application Topologies



SERCOS

A Sercos object represents a Master on the SERCOS ring.

NODE

A Node object represents a Slave Node on the SERCOS ring, which is typically either a drive or an I/O module.

S-X-XXXX (Standard IDNs)

OR

P-X-XXXX (Manufacturer-specific IDNs)

IDN

An Idn object is simply a container for a SERCOS IDN.

IDNLIST — **IDN 1** **IDN 2** **IDN 3** **IDN 4**

An IdnList object keeps a list of Idns.

SERCOS — **IDN 1** **IDN 2** **IDN 3** **IDN 4**

A Sercos object keeps a list of Node objects.

NODE 3 — **IDNLIST** — **IDN 1** **IDN 2** **IDN 3** **IDN 4**

An IdnList object is always associated with a Node.

Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

Return to Software's Main Menu

Copyright © 2002
Motion Engineering

SERCOS Communication Error Notification

If your motion controller uses SERCOS, you can take advantage of specialized status information. The MPIStatusMask in the MPIStatus structure makes it possible to read the MPIStatusFlag(s):

```
/* MPIStatus */
typedef enum {
        MPIStatusFlagINVALID,
        MPIStatusFlagCOMM_ERROR,
} MPIStatusFlag;

#define mpiStatusMaskBIT(flag)    (0x1 << (flag))

typedef enum {
    MPIStatusMaskNONE         = 0x0,
    MPIStatusMaskCOMM_ERROR   = mpiStatusMaskBIT(MPIStatusFlagCOMM_ERROR),  /* 0x00000001 */
    MPIStatusMaskMOTOR        = MPIStatusMaskCOMM_ERROR,                     /* 0x00000001 */
    MPIStatusMaskALL          = mpiStatusMaskBIT(MPIStatusFlagLAST) - 1      /* 0x00000001 */
} MPIStatusMask;

typedef struct MPIStatus {
    MPIState        state;
    MPIAction       action;
    MPIEventMask    eventMask;

    long     settled;
    long     atTarget;

    MPIStatusMask     statusMask;
} MPIStatus;
```

Presently, the only MPIStatusFlag supported is MPIStatusFlagCOMM_ERROR. This flag represents the communication status. For XMP-Series SERCOS controllers, this flag represents the status of the SERCOS communication. When the SERCOS communication ring fails, the MPIStatusFlagCOMM_ERROR is set. The flag can only be cleared by re-initializating the SERCOS ring with mpiSercosInit(...).

Return to Sercos Objects page

Copyright © 2002
Motion Engineering

Service Objects

Introduction

A **Service** object creates and handles threads that help event managers dispatch events. Typically, one will use a Service Object to create threads that will call **mpiEventMgrService(…)** whenever an XMP interrupt occurs. They are a convenient way to have a program automatically deal with event managers and events. Thread handling is something that is different on every operating system. Service objects may therefore have different behaviors on different operating systems. Programmers that are experienced in multi-threaded application programming will probably want to program their own threads that will call **mpiEventMgrService(…)**.

Methods

Create, Delete, Validate Methods

Service**Create**	Create a Service for EventMgr and the threads necessary for it to run.
Service**Delete**	Stop all threads belonging to the Service and deletes the Service.

Configuration and Information Methods

Service**Enable**	Enable or disables the Service.

Copyright © 2002
Motion Engineering

ServiceCreate

Declaration

```
const Service serviceCreate(MPIEventMgr eventMgr,
                            long        priority,
                            long        sleep)
```

Required Header

service.h

Description

ServiceCreate creates threads for each control associated with ***eventMgr***, flushes ***eventMgr***, and starts threads with priority that call mpiEventMgrService(eventMgr, …) every ***sleep*** milliseconds.

priority is a platform specific variable.

If "priority" is	*Then*
-1	The operating system will choose some default priority for the service's threads.
>0	*ServiceCreate* will attempt to assign the priority to all of the service's threads.

If "sleep" is	*Then*
-1	*ServiceCreate* will attempt to create interrupt driven threads.
0	*ServiceCreate* will create threads that call **mpiEventMgrService(eventMgr,...)** as quickly as possible.
>0	*ServiceCreate* will create threads that attempt to call **mpiEventMgrService(eventMgr,...)** every ***sleep*** milliseconds.

Return Values	
handle	to a Service object
MPIHandleVOID	if the Service could not be created

See Also

mpiEventMgrService | ServiceDelete

ServiceDelete

Declaration `long serviceDelete(Service service)`

Required Header service.h

Description **ServiceDelete** alerts all threads that they should end, waits for all threads to end, and frees the memory allocated to ***service***.

Return Values

MPIMessageOK	if *serviceDelete* successfully deletes a Service object and invalidates its handle

See Also ServiceCreate

ServiceEnable

Declaration

```
long serviceEnable(Service service,
                   long    enabled)
```

Required Header

service.h

Description

ServiceEnable enables or disables all threads belonging to Service.

If "enabled" is	*Then*
FALSE	*serviceEnable* will disable ***service***.
TRUE	*serviceEnable* will enable ***service***.

Return Values

handle	to a Service object
MPIHandleVOID	if the Service could not be created

See Also

Trace Objects

Introduction

Use the **Trace** module to selectively produce trace output on a global and/or per-object basis for your application. You can specify the types of trace output when an application is linked, or dynamically (by using a debugger).

Note: You can also define your own trace function, using meiPlatformTraceFunction(...)
For example, you could define your own function to send traces to a circular memory buffer.

The format of the trace output is determined by **printf(...)**-like trace macros located in MPI library source. The trace macros are of the form **meiTrace#(mask, format, arg ...)**, where ***format*** and the ***args*** determine the trace output, and where **#** indicates the total number of arguments following the ***format*** argument (because macros cannot take variable numbers of arguments).

The placement and content of the **meiTrace(...)** macros in the MPI library source is the responsibility of whomever maintains the library. Because trace can be added as desired, it is often useful to leave trace statements in the library source code rather than remove them, as is similarly done with debug printf(...) statements. It is also useful to define per-object trace output types so that the volume of trace output is set to a manageable level.

The Trace module interface is declared in the **XMP\include\trace.h** header file. In order for your application to use Trace functions, you must build your application with the MEI_TRACE conditional-compile symbol defined.

To install trace, simply install the DLL for either the Debug or DebugSingle configuration. The Debug and DebugSingle configurations of the MPI library are built with the MEI_TRACE compile-time symbol defined.

Note: Debug and DebugSingle are the only MPI library configurations that will produce trace output.

By default, trace output is sent to standard error. However, to send trace output to a file, your application can call the meiTraceFile(char *fileName) function.

To obtain the current global trace mask, call meiTraceGet(...).
To modify the global trace mask, call meiTraceSet(...).

To obtain an object's trace mask, call meiObjectTraceGet(...) (defined in stdmei.h).
To modify an object's trace mask, call meiObjectTraceSet(...).

Trace Masks | Global Trace Outputs | Per-Object Trace Outputs

Methods

Configuration and Information Methods

meiTrace**Eol**	Set the end-of-line character to be used by Trace
meiTrace**File**	Send trace output to a file
meiTrace**Function**	sets function used to display a trace buffer
meiTrace**Get**	Get global trace mask
meiTrace**Set**	Set global trace mask

Data Types

MEITrace**Function**

MEITrace**Mask**

Constants

MEITraceMask**GLOBAL**

Copyright © 2002
Motion Engineering

meiTraceEol

Declaration `char meiTraceEol(char eol)`

Required Header stdmei.h

Description **TraceEol** function simply calls **meiPlatformTraceEol(...)**, which sets the end-of-line character that will be used by meiPlatformTrace(...). By default, meiPlatformTrace(...) will append a newline character ('\n') to the messages that it displays. The **meiPlatformTraceEol(...)** function allows your application to set the default end-of-line character.

Returns the previous end-of-line character used by meiPlatformTrace(...)

See Also meiPlatformTrace | meiPlatformTraceEol

meiTraceFile

Declaration `long meiTraceFile(const char *fileName)`

Required Header stdmei.h

Description **TraceFile** causes trace output to be sent to the file ***fileName***. By default, trace output goes to standard output. Note that if ***fileName*** is Null, trace output still goes to standard output.

Return Values

MPIMessageOK	if *TraceFile* successfully causes trace output to be sent to the file

See Also

meiTraceFunction

Declaration

```
MEITraceFunction meiTraceFunction(MEITraceFunction traceFunction)
```

Required Header

stdmei.h

Description

TraceFunction sets the function used to display a trace buffer.

Front end to meiPlatformTraceFunction(). If traceFunction is NULL (default), then trace functions is fprintf(MEIPlatformTraceSTREAM) (default stdout).

Return Values

handle	to previous Trace function
NULL	otherwise

See Also

meiTraceGet

Declaration `MEITraceMask meiTraceGet(void)`

Required Header stdmei.h

Description **TraceGet** returns the current global trace mask for the application.

Returns	the global trace mask

See Also meiTraceSet

meiTraceSet

Declaration

```
MEITraceMask meiTraceSet(MEITraceMask mask)
```

Required Header

stdmei.h

Description

TraceSet sets the global trace mask to ***mask***.

If "traceMask" is	*Then*
MEITraceALL	all global categories of trace will be enabled
MEITraceNONE	all categories of trace will be disabled

Returns

the value of the previous *global trace mask*

See Also

meiTraceGet

MEITraceFunction

MEITraceFunction

```
typedef long (*MEITraceFunction)(const char *buffer);
```

Description

Definition for a trace function interface. **TraceFunction** can be used to define a custom trace output routine. MEITraceFunction function must take a pointer to a buffer as a parameter and must return a long.

See Also

meiTraceFunction

MEITraceMask

MEITraceMask

```
typedef    unsigned long MEITraceMask;
```

Description **TraceMask** is a bit mask used to enable/disable library trace statement output.

See Also meiTraceGet | meiTraceSet | MEITraceMaskGLOBAL

MEITraceMaskGLOBAL

MEITraceMaskGLOBAL

```
extern MEITraceMask      MEITraceMaskGLOBAL;
```

Description **TraceMaskGlobal** is a non-object specific MPI Trace mask variable used for library wide Trace bits.

See Also MEITraceMask

Trace Masks

Every MPI object contains an MEITraceMask and every process contains a single global MEITraceMask. An MEITraceMask consists of bits, where each bit corresponds to a single trace category. A trace category is a specific type of debug information that you want to be displayed by the MPI library. A trace category can be either global (applying to all MPI objects) or object-specific (applying only to a specific MPI object).

Trace Category	Is
global	declared by the MEITrace{...} enum in trace.h.
object-specific (for MPI objects)	declared in stdmei.h.
object-specific (for MEI objects)	declared in the object header file (for MEI objects). Note that the trace mask bits for object-specific trace categories overlap.

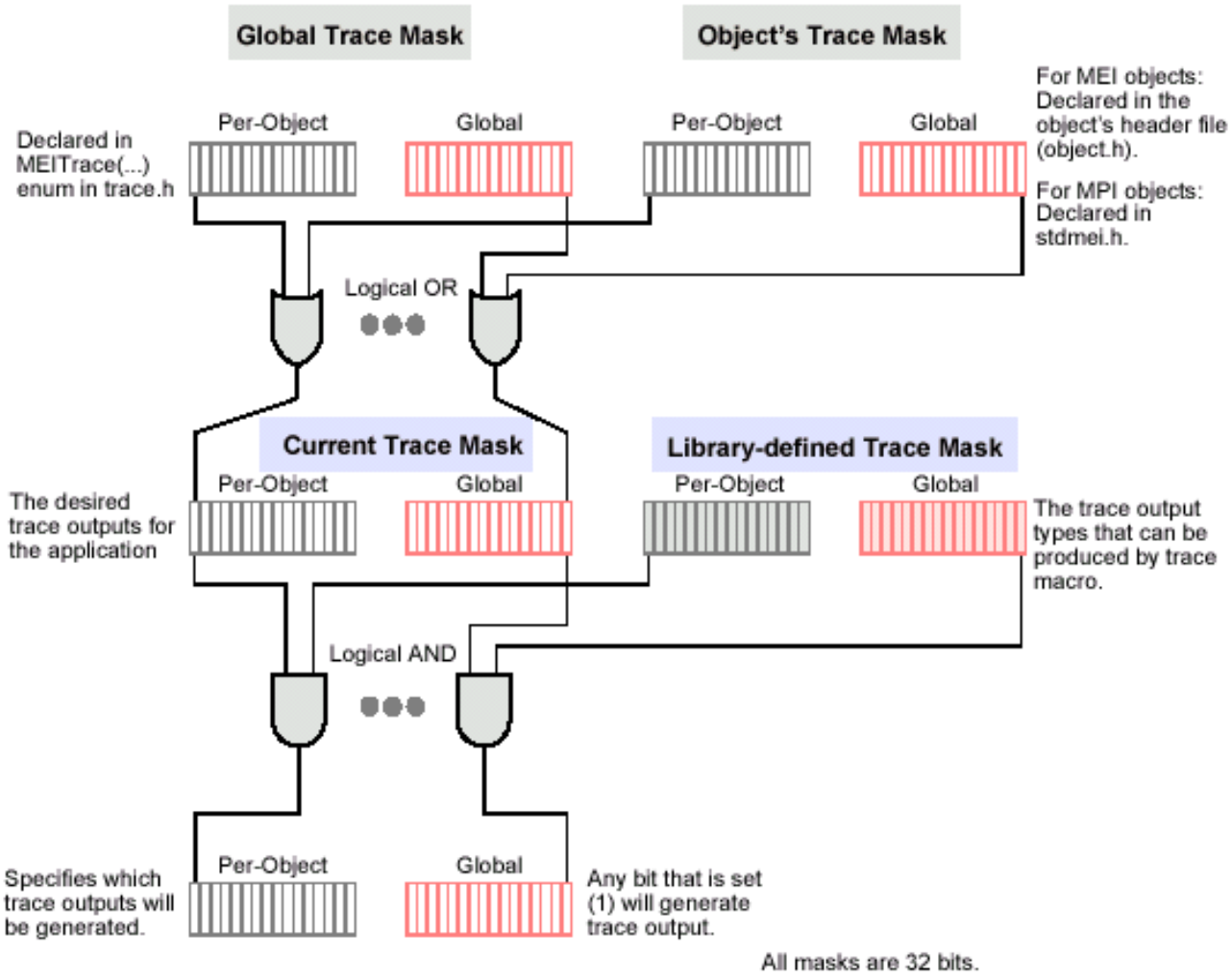
An object will produce trace output for a trace category when the logical **OR** of the **global trace mask** and the **object's trace mask** has the bit set that corresponds to the trace category.

If the global trace mask has all of its bits set, then all objects will display trace output for all trace categories.

If an object's trace mask has **all** of its bits set, then that object will display trace output for all trace categories, but a different object of the same type might produce less or no trace output depending on the setting of its trace mask. The setting of the global and object trace masks is under the control of your application.

The trace mask is derived in 2 steps:

1. The **global trace mask** is logically **OR**ed with the **object trace mask**. This yields the **current trace mask**, representing the desired trace output types as specified by the application.
2. The **current trace mask** (from step1) is logically **AND**ed with a **library-defined trace mask** (that describes the trace output types for which the trace macro should produce output). If the result of the AND is non-zero, trace output will be produced using the ***format*** and the ***args*** [from **meiTrace#(mask, format, arg ...**

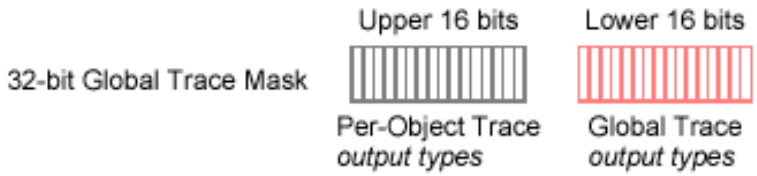


Return to Trace Object's page

Copyright © 2002
Motion Engineering

Global Trace Outputs

There is a global 32-bit trace mask: the low 16 bits are the global trace output types, while the upper 16 bits are the per-object trace output types. Each object has a similar trace mask. The upper 16 bits of the global trace mask are not defined, but can be used to set the per-object output types for all objects. To enable all trace output types for all objects, set the global trace mask to all 1s (i.e., -1).



The **MEITrace{...}** enum (declared in trace.h) specifies the global types of trace output, i.e., the types of trace output that can be produced by any object or module. **The MEITrace{...}** enum defines constants that you use together as a bit mask. You specify the desired trace output as a combination (logical OR) of **MEITrace{...}** constants.

There are 16 possible types of global trace output, with 12 global trace outputs defined.

Output Type	Displays
MEITraceFUNCTION_ENTRY	Function name & calling parameters upon entry to function
MEITraceFUNCTION_RETURN	Function name, calling parameters & return value upon exit from function
MEITraceMEMORY_ALLOC	The Address & byte count when memory is dynamically allocated
MEITraceMEMORY_FREE	The Address & byte count when dynamically allocated memory is freed
MEITraceMEMORY_GET	Source address, destination address, byte count when reading XMP firmware memory
MEITraceMEMORY_SET	Source address, destination address, byte count when writing XMP firmware memory
MEITraceVALIDATE	Results of object validation
MEITraceLOCK_GIVE	When a resource lock is released
MEITraceLOCK_TAKE	When a resource lock is waited for & obtained
MEITraceEVENT	When an XMP event is received
MEITraceALL	All global trace outputs (lower 16 bits)
MEIModuleTraceALL	All per-object trace outputs (upper 16 bits)

Return to Trace Object's page

Copyright © 2002
Motion Engineering

Per-Object Trace Outputs

There are 16 possible types of per-object trace output. Each object can declare up to 16 of its own trace output types. MPI modules declare per-object trace output types in stdmei.h. MEI modules declare per-object trace output types in the module header file.

Output Type	Displays
MEIMotionTraceSTATUS	Status of the Motion Supervisor
MEINotifyTrcaeTHREAD	When a thread goes to sleep or wakes up
MEISequenceTraceLOAD	When a batch of new commands are sent to the XMP Program Sequencer
MEIConfigTracePROGRESS	Displays “.” as it executes (used by config utility)
MEIRecorderTraceRECORD_GET	When the Recorder gets records from the XMP
MEIRecorderTraceSTATUS	The number of data records available in the XMP

Note: The first 5 output types overlap in the mask.

Return to Trace Object's page

Copyright © 2002
Motion Engineering