

Select an MPI/MEI Object

MPI Library: 03.01.00

Last Modified : 27 July 2004

[Axis](#)

[EventMgr](#)

[Platform](#)

[CAN](#)

[Filter](#)

[Probe](#)

[Capture](#)

[Flash](#)

[Recorder](#)

[Command](#)

[Global](#)*

[Sequence](#)

[Compare](#)

[Message](#)*

[Service](#)

[Compensator](#)

[Motion](#)

[SqNode](#)

[Control](#)

[Motor](#)

[SynqNet](#)

[DriveMap](#)

[Notify](#)

[Trace](#)*

[Event](#)

[Object](#)*

[Xmp](#)

[EventMask](#)

[Path](#)

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

<u>mpiAxisCreate</u>	Create Axis object
<u>mpiAxisDelete</u>	Delete Axis object
<u>mpiAxisValidate</u>	Validate Axis object

Configuration and Information Methods

<u>mpiAxisActualPositionGet</u>	Get actual position
<u>mpiAxisActualPositionSet</u>	Set actual position
<u>mpiAxisActualVelocity</u>	Get actual velocity
<u>mpiAxisConfigGet</u>	Get Axis configuration
<u>mpiAxisConfigSet</u>	Set Axis configuration
<u>mpiAxisCommandPositionGet</u>	Get command position
<u>mpiAxisCommandPositionSet</u>	Set command position
<u>mpiAxisFlashConfigGet</u>	Get Axis flash config
<u>mpiAxisFlashConfigSet</u>	Set Axis flash config
<u>mpiAxisOriginGet</u>	Get Axis origin
<u>mpiAxisOriginSet</u>	Set Axis origin
<u>mpiAxisPositionError</u>	Get position error of an Axis
<u>mpiAxisStatus</u>	Get Axis status

[mpiAxisTrajectory](#)

Get Axis trajectory

Event Methods

[mpiAxisEventNotifyGet](#)

Get event mask

[mpiAxisEventNotifySet](#)

Set event mask

[mpiAxisEventReset](#)

Memory Methods

[mpiAxisMemory](#)

Set Axis memory address

[mpiAxisMemoryGet](#)

Copy bytes of Axis memory to application memory

[mpiAxisMemorySet](#)

Copy bytes of application memory to Axis memory

Relational Methods

[mpiAxisControl](#)

Return handle of Control associated with Axis

[mpiAxisFilterMapGet](#)

Get object map of Filters

[mpiAxisFilterMapSet](#)

Set object map of Filters

[mpiAxisMotorMapGet](#)

Get object map of Motors

[mpiAxisNumber](#)

Get index of Axis

Data Types

[MPIAxisConfig](#) / [MEIAxisConfig](#)[MPIAxisInPosition](#)[MPIAxisMaster](#)[MPIAxisMasterType](#)[MPIAxisMessage](#)

mpiAxisCreate

Declaration

MPIAxis [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 <i>AxisDelete</i> 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](#)

Required Header

Description

Return Values

See Also [mpiAxisActualPositionSet](#) | [Using the Origin Variable](#)

mpiAxisActualPositionSet

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

Required Header

Description	AxisActualPositionSet sets the value of the actual position of an Axis (<i>axis</i>) to <i>actual</i> .

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

Return Values

MPIMessageOK	if <i>AxisActualPositionSet</i> successfully sets the value of the actual position of an Axis to <i>actual</i>

See Also [AxisCommandPositionSet](#) | [Using the Origin Variable](#)

Required Header

mpiAxisCommandPositionSet(...) Error Check

- Axis is in a STOPPING, STOPPED, or MOVING state.
- Any motor associated with the axis has the disableAction configuration set to MEIMotorDisableActionCMD_EQ_ACT and the motor's Amp Enable is disabled.
- If the command position read from the controller does not match the requested position.

Remarks

Return Values

MPIMessageOK	if <i>AxisCommandPositionSet</i> successfully sets the value of the command position of <i>axis</i> from <i>command</i>

Axis: unable to set command position

Indicates that the motor associated to this axis is configured to mode MEIMotorDisableActionCMD_EQ_ACT and the motor is disabled. When the motor is disabled and in the MEIMotorDisableActionCMD_EQ_ACT mode, the command position is continually set to the actual position.

To set the command position, enable the motor or take the motor out of the MEIMotorDisableActionCMD_EQ_ACT mode.

Alternatively, setting the origin for the motor can often perform an equivalent result in this situation, as the command position will be set to the actual position the next sample.

See Also

[MEIMotorDisableAction](#) | [AxisActualPositionSet](#) | [AxisCommandPositionSet](#) | [MPIAxisMessage](#)

mpiAxisActualVelocity

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

Required Header

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

Return Values

MPIMessageOK	if <i>AxisActualVelocity</i> succesfully gets and writes the value of the actual velocity of <i>axis</i> 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.

Sample Code

```
/* Change axis encoder scaling.
   limit scale to +/- 2.0 */
void axisScale(MPIAxis axis, float scale)
{
    MPIAxisConfig config;
    MEIAxisConfig xmpConfig;

    mpiAxisConfigGet(axis, &config, &xmpConfig);
    xmpConfig.APos[0].Coeff = (long)(scale * MEIXmpFRACTIONAL_UNITY);
    mpiAxisConfigSet(axis, &config, &xmpConfig);
}
```

Return Values

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

See Also

[MPIAxisConfig](#) | [mpiAxisConfigSet](#) | [MEIAxisConfig](#)

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

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).

The MEIXmpAxisGear firmware feature only supports servo motor types. The axis gear feature does not support step motor types.

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.

Sample Code

```

/* Change axis encoder scaling.
   limit scale to +/- 2.0 */
void axisScale(MPIAxis axis, float scale)
{
    MPIAxisConfig config;
    MEIAxisConfig xmpConfig;

    mpiAxisConfigGet(axis, &config, &xmpConfig);
    xmpConfig.APos[0].Coeff = (long)(scale * MEIXmpFRACTIONAL_UNITY);
    mpiAxisConfigSet(axis, &config, &xmpConfig);
}

```

Return Values

MPIMessageOK	if <i>AxisConfigSet</i> successfully sets the Axis configuration.

See Also

[mpiAxisConfigGet](#) | [MEIAdcConfig](#) | [MEIAxisConfig](#)

mpiAxisCommandPositionGet

Declaration

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

Required Header

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 <i>AxisCommandPositionGet</i> successfully gets the value of the command position of <i>axis</i> and puts it in the location.

See Also [mpiAxisCommandPositionSet](#)

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 <i>AxisEventNotifySet</i> 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 (MPIAxisaxis, 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 <i>AxisFlashConfigSet</i> successfully sets the Axis flash configuration using data from the structure(s)

See Also

[MEIFlash](#) | [mpiAxisFlashConfigGet](#) | [MEIAxisConfig](#)

Required Header

```
stdmpi.h
```

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

MPIMessageOK	if <i>AxisOriginGet</i> successfully gets the value of the origin of the Axis and writes it to the location

file:///D:/pdfs/030100/html/Software-MPI/docs/Axis/Method/orgget1.htm [7/22/2004 1:15:45 PM]

mpiAxisOriginSet

Declaration

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

Required Header

Description	AxisOriginSet sets the value of the origin of an Axis (<i>axis</i>) to <i>origin</i> .

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

Return Values

MPIMessageOK	if <i>AxisOriginSet</i> successfully sets the origin of an Axis to <i>origin</i>

See Also [mpiAxisOriginGet](#) | [Using the Origin Variable](#)

Required Header

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

MPIMessageOK	if <i>AxisPositionError</i> successfully gets and writes the value of the position error into <i>*error</i>

file:///D:/pdfs/030100/html/Software-MPI/docs/Axis/Method/poserr1.htm [7/22/2004 1:15:45 PM]

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).

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

XMP Only

external should always be set to NULL.

Return Values	
MPIMessageOK	if <i>AxisStatus</i> successfully gets the Axis status and writes it into the structure(s)
MPIMessageARG_INVALID	if the <i>status</i> pointer is NULL.

See Also

Required Header

NOTE: deceleration, jerkPercent, accelerationJerk, and decelerationJerk fields of *trajectory* cannot be read from the controller and consequently are set to zero.

Remarks

Sample Code

```
MPITrajectory trajectory;

    mpiAxisTrajectory(axis, &trajectory);

    printf("Velocity %.3f\n"
           "Acceleration %.3f\n",
           trajectory.velocity,
           trajectory.acceleration);
```

Return Values

MPIMessageOK	if <i>AxisTrajectory</i> successfully gets the Axis trajectory and writes it into the structure

See Also [mpiMotionStart](#) | [mpiMotionModify](#) | [MPITrajectory](#)

mpiAxisEventNotifyGet

Declaration

long mpiAxisEventNotifyGet (MPIAxis
MPIEventMask
void
axis,
*eventMask,
*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](#)
[MPIEventMask](#)
voidaxis,
eventMask,
*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

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

XMP Only

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 <i>eventmask</i> with mpiEventMaskALL(eventMask)
disable host notification of all events	configure <i>eventmask</i> with mpiEventMaskCLEAR(eventMask)

Return Values	
MPIMessageOK	if <i>AxisEventNotifySet</i> successfully requests host notification of the event(s) that are specified by <i>eventMask</i> and generated by <i>motion</i>

See Also

[MEIEventNotifyData](#) | [MEIEventStatusInfo](#) | [MPIEventMask](#) | [MPIEventType](#) | [mpiEventMaskALL](#) | [mpiEventMaskCLEAR](#) | [mpiAxisEventNotifyGet](#) | [MEIEventNotifyData](#)

Required Header

```
stdmpi.h
```

Description

To...	Then...
enable host notification of all events	configure <i>eventmask</i> with mpiEventMaskALL(eventMask)
disable host notification of all events	configure <i>eventmask</i> with mpiEventMaskCLEAR(eventMask)

Return Values

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

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 <i>AxisMemory</i> 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 <i>AxisMemorySet</i> successfully copies count bytes of application memory to Axis memory

See Also [mpiAxisMemory](#) | [mpiAxisMemoryGet](#)

mpiAxisControl

Declaration `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

Required Header

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

MPIObjMap 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.

MPIMessageOK	if <i>AxisFilterMapGet</i> successfully gets and writes the object map to the structure

file:///D:/pdfs/030100/html/Software-MPI/docs/Axis/Method/ftmapget1.htm [7/22/2004 1:15:48 PM]

mpiAxisFilterMapSet

```
Declaration      long mpiAxisFilterMapSet(MPIAxis axis,
                               MPIObjectMap filterMap)
```

Required Header

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

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

Remarks

MPIOObjectMap 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 <i>AxisFilterMapSet</i> successfully sets the Filters using data from the object map

See Also [mpiAxisFilterMapGet](#) | [MPIObjectMap](#)

Required Header

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

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.

Return Values

See Also

mpiAxisNumber

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

Required Header

Command	Description
AxisNumber	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 <i>AxisNumber</i> successfully writes the index of <i>Axis</i> to the contents of <i>number</i>

See Also

MPIAxisConfig / MEIAxisConfig

MPIAxisConfig

```
typedef struct MPIAxisConfig {
    MPIAxisInPosition    inPosition;
    MPIAxisMaster         master;
    long                 masterCorrection;
    MPIObjectMap          filterMap;
}MPIAxisConfig;
```

Description

inPosition	See MPIAxisInPosition .
master	This field defines the source of the position and velocities used as the master for cam motion. See Master Position Source .
masterCorrection	Specifies which axis provides the master position correction. A value of -1 stops any stops master corrections from being used. See Camming: Correctional Moves .
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 - Coefficient that multiplies the encoder input. Coeff is a custom unit. The range of Coeff is +/- 2.0 (+/- 2*MEIXmpFRACTIONAL_UNITY).
	For a 1:1 ratio of encoder input to reported encoder input set: Coeff = MEIXmpFRACTIONAL_UNITY.
	For 0.5:1 ratio, set: Coeff = MEIXmpFRACTIONAL_UNITY / 2.
	When the distance between the positive and negative limit configurations exceed 32 bits

(4,294,967,296 counts), both limits are triggered. The distance between the positive and negative software position limits must be less than 32 bits (4,294,967,296 counts).

Filter

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

Gear - Coefficients for gearing off a position input. The MEIXmpAxisGear firmware feature only supports servo motor types. The axis gear feature does not support step motor types.

- **Ptr** - Host pointer to a gear master

Example:

```
MEIXmpData      *firmware;
MEIXmpBufferData *bufferData;
```

```
mpiControlMemory(control,&firmware,&bufferData);
```

```
...
```

```
msgCHECK(mpiAxisConfigGet(axis, &axisConfig, &axisConfigXmp));
axisConfigXmp.Gear.Ptr = &bufferData->PreFilter[0].Output;
msgCHECK(mpiAxisConfigSet(axis, &axisConfig, &axisConfigXmp));
```

- **Ratio.A** - numerator of multiplier
- **Ratio.B** - denominator of multiplier
- **Ratio.Old** -
- **Ratio.Remainder** -
- **Position** - final geared position

Sample Code

```
/* Change axis encoder scaling.
   limit scale to +/- 2.0 */
void axisScale(MPIAxis axis, float scale)
{
    MPIAxisConfig config;
    MEIAxisConfig xmpConfig;

    mpiAxisConfigGet(axis, &config, &xmpConfig);
    xmpConfig.APos[0].Coeff = (long)(scale * MEIXmpFRACTIONAL_UNITY);
    mpiAxisConfigSet(axis, &config, &xmpConfig);
}
```

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
    long     settleOnEstopCmdEqAct;
} MPIAxisInPosition;
```

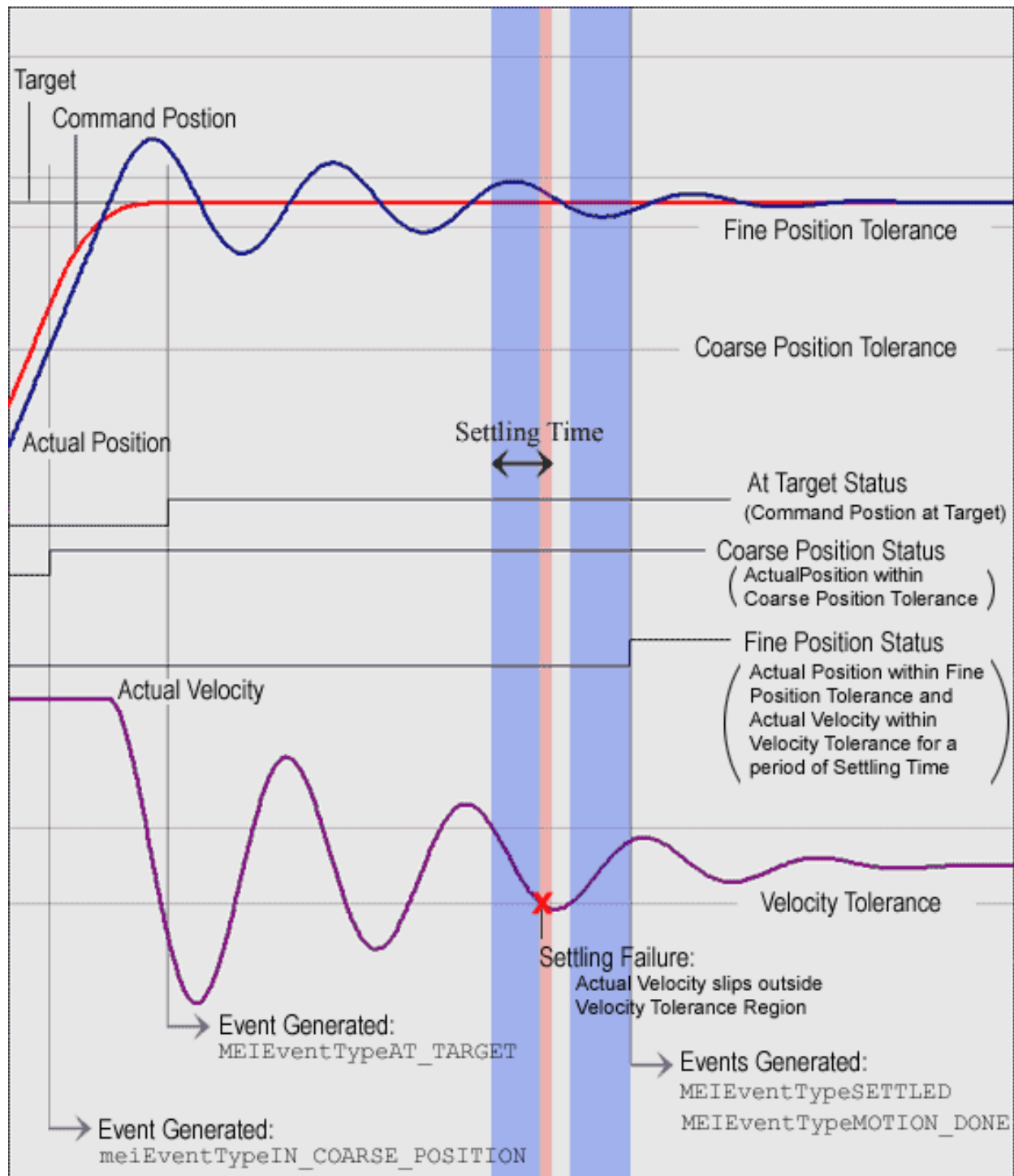
Description

tolerance	Includes the following 3 elements that determine settling tolerances for an axis.
positionFine	Value, in counts, from the move target position at which the controller sets the "in fine position" status flag. This parameter is used as part of the Axis settling criteria to determine when a point-to-point motion is complete and when MPIEventTypeMOTION_DONE and MEIEventTypeSETTLEDevents are generated.
positionCoarse	Value, in counts, from a move target position at which the controller sets the "in coarse position" status flag. This value does not affect the settling time status.
velocity	Value, in counts/second, from the final move velocity at which the controller sets the "at velocity" status flag. This parameter is used as part of the Axis settling criteria to determine when: - a position-based move is complete and an MPIEventTypeMOTION_DONE event is generated - a velocity move is complete and an MPIEventTypeMOTION_AT_VELOCITY event is generated - an axis is settled and an MPIEventTypeSETTLED event is generated
settlingTime	Duration in seconds that an axis must satisfy the positionFine and/or velocity tolerance, before the respective status flag is set.

settleOnStop	If TRUE, the controller will use settle on stop mode. If FALSE, the controller will not use the settle on stop mode. When in settleOnStop mode and a STOP event has occurred, the axis will stay in an MPIStateSTOPPING state until: 1. The settling criteria are satisfied AND 2. The stop duration for the axis' Motion Supervisor has elapsed. 3. This state can be read with <code>mpiAxisStatus(MPIAxis axis, MPIStatus *status, void *external)</code>. The value to look for is (MPIState) <code>status.state</code>. If <code>settleOnStop = FALSE</code>, the axis will stay in an MPIStateSTOPPING state only until the stop duration for the axis' Motion Supervisor has elapsed.
settleOnEstop	If TRUE, the controller will use settle on Estop mode. If FALSE, the controller will not use the settle on Estop mode. When in settleOnEstop mode and a ESTOP event has occurred, the axis will stay in an MPIStateSTOPPING_ERROR state until: 1. The settling criteria are satisfied AND 2. The Estop duration for the axis' Motion Supervisor has elapsed. 3. This state can be read with <code>mpiAxisStatus(MPIAxis axis, MPIStatus *status, void *external)</code>. The value to look for is (MPIState) <code>status.state</code>. If <code>settleOnEstop = FALSE</code>, the axis will stay in an MPIStateSTOPPING_ERROR state only until the Estop duration for the axis' Motion Supervisor has elapsed.
settleOnEstopCmdEqAct	If TRUE, the controller will use settle on EstopCmdEqAct mode. If FALSE, the controller will not use the settle on EstopCmdEqAct mode. ***settleOnEstopCmdEqAct mode is not recommended*** SettleOnEstopCmdEqAct is an alternative to Estop mode. When this mode is enabled, the following things happen: - During normal motion, there is no difference. - During an Estop, Cmd Eq Act action, the command position is set equal to the actual position from the previous servo sample. This can have a damping effect in some systems with some tuning parameters, causing the stage to slow. The behavior of the stage in this mode can be vastly different than in normal

servoing mode. Approach this mode with great caution. The axis will stay in this mode for the amount of time that the Axis' Motion Supervisor Estop time.

- After the Estop time elapses, the axis' motors will disable the amplifiers.



Sample Code

```

/*
    Set the settling time of an axis.  Sample usage:
    returnValue =
        setAxisSettlingTime(axis, 0.05);
*/
long setAxisSettlingTime(MPIAxis axis, double settlingTime)
{
    MPIAxisConfig config;
    long returnValue;

    returnValue =
        mpiAxisConfigGet(axis, &config, NULL);

    if (returnValue == MPIMessageOK)
    {
        config.inPosition.settlingTime = (float) settlingTime;
        returnValue =
            mpiAxisConfigSet(axis, &config, NULL);
    }

    return returnValue;
}

```

See Also [MPIAxisConfig](#) | [MPIAction](#)

Axis Tolerances and How Motion Related Events are Generated
[How Motion Completion Events are Generated](#)

[Special Note](#) on Configuration of IN_POSITION and Done Events after STOP or E_STOP
 Events

MPIAxisMaster

MPIAxisMaster

```
typedef enum {
    MPIAxisMasterType    type;
    long                 number;
    long                 *address;
    long                 encoderFaultMotorNumber;
}MPIAxisMaster;
```

Description

AxisMaster defines the source of the position and velocities used as the master for cam motion. See also [Master Position Source](#).

The *type* field specifies if the number or address fields are used and which object the number field refers to.

MPIMasterType	Number	Address
MPIAxisMasterTypeMOTOR_FEEDBACK_PRIMARY	motor number	Not used
MPIAxisMasterTypeMOTOR_FEEDBACK_SECONDARY	motor number	Not used
MPIAxisMasterTypeAXIS_COMMANDED POSITION	Axis number	Not used
MPIAxisMasterTypeAXIS_ACTUAL POSITION	Axis number	Not used
MPIAxisMasterTypeADDRESS	Not used	Any controller address

type	This field defines the type of master position source is being used.
number	the motor or axis number.
address	The controller address to be used as the master position.
encoderFaultMotorNumber	The number of the motor that is checked for an encoder fault. If this motor detects an encoder fault this axis will abort. A value of -1 disables this encoder fault function. See Master Encoder Faults .

See Also [MPIAxisMasterType](#)

MPIAxisMasterType

MPIAxisMasterType

```
typedef enum {
    MPIAxisMasterTypeMOTOR_FEEDBACK_PRIMARY,
    MPIAxisMasterTypeMOTOR_FEEDBACK_SECONDARY,
    MPIAxisMasterTypeAXIS_CMDANDED_POSITION,
    MPIAxisMasterTypeAXIS_ACTUAL_POSITION,
    MPIAxisMasterTypeADDRESS,
}MPIAxisMasterType;
```

Description [AxisMaster](#) specifies the type of master position source used with cam motions. See also [MPIAxisMaster](#).

Fields	Number	Address
MPIAxisMasterTypeMOTOR_FEEDBACK_PRIMARY	Motor number	Not used
MPIAxisMasterTypeMOTOR_FEEDBACK_SECONDARY	Motor number	Not used
MPIAxisMasterTypeAXIS_CMDANDED_POSITION	Axis number	Not used
MPIAxisMasterTypeAXIS_ACTUAL_POSITION	Axis number	Not used
MPIAxisMasterTypeADDRESS	Not used	Any controller address

See Also [MPIAxisMaster](#)

MPIAxisMessage

MPIAxisMessage

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

Description

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

MPIAxisMessageAXIS_INVALID

The axis number is out of range. This message code is returned by [mpiAxisCreate\(...\)](#) if the axis number is less than zero or greater than or equal to MEIXmpMAX_Axes.

MPIAxisMessageCOMMAND_NOT_SET

The axis command position did not get set. This message code is returned by [mpiAxisCommandPositionSet\(...\)](#) if the controller's command position does not match the specified value. Internally, the `mpiAxisCommandPositionSet(...)` method requests the controller to change the command position, waits for the controller to process the request, and reads back the controller's command position. There are several cases where the controller will calculate a new command position to replace the requested command position. For example, if motion is in progress, stopped, or if the amp enable is disabled (when the motor's disableAction is configured for command equals actual), the controller will calculate a new command position every sample. To prevent this problem, set the command position when the motion is in an IDLE state and the motor's disableAction is configured for no action.

mpiAxisCommandPositionSet(...) Error Check

The `mpiAxisCommandPositionSet(...)` error check has been extended. If the controller is updating the axis's command position when `mpiAxisCommandPositionSet(...)` is called, `MPIAxisMessageCOMMAND_NOT_SET` will be returned. `mpiAxisCommandPositionSet(...)` checks for the following conditions:

- Axis is in a STOPPING, STOPPED, or MOVING state.
- Any motor associated with the axis has the disableAction configuration set to MEIMotorDisableActionCMD_EQ_ACT and the motor's Amp Enable is disabled.
- If the command position read from the controller does not match the requested position.

MPIAxisMessageNOT_MAPPED_TO_MS

An axis is not mapped to the motion supervisor. This message code is returned by [mpiMotionDelete\(...\)](#), [mpiMotionAxisList\(...\)](#), or [mpiMotionAxisRemove\(...\)](#) when an axis is associated with a motion object, but not mapped to a motion supervisor. To correct this problem, map the axes to the motion supervisor in the controller by calling: [mpiMotionAction\(...\)](#) with [MEIActionMAP](#) or [MPIActionRESET](#), [mpiMotionStart\(...\)](#), [mpiMotionModify\(...\)](#), or [mpiMotionEventNotifySet\(...\)](#).

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](#)).
- $|\text{command position} - \text{actual position}| < \text{fine position tolerance}$.
- $|\text{Target velocity} - \text{actual velocity}| < \text{velocity tolerance}$ (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](#)).
- $|\text{target position} - \text{actual position}| < \text{fine position tolerance}$.
- $|\text{command velocity} - \text{actual velocity}| < \text{velocity tolerance}$ (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](#)

CAN Objects

Introduction

The CAN object allow the user easy access to the I/O nodes connected to a controller's CANOpen interface.

If a controller does not support the CANOpen interface, the meiCanValidate function will return MEICanMessageINTERFACE_NOT_FOUND.

The CAN system uses the MEICanConfig and MEICanNodeConfig structures to hold all of the user configurable quantities. These structures are stored in non-volatile flash memory. When the XMP is released from reset (normally soon after the host powers up or after a call to mpiControlReset), the CAN Processor will initialize itself with data from MEICanConfig and MEICanNodeConfig before starting to scanning the network for nodes.

The functions meiCanConfigGet, meiCanConfigSet, meiCanNodeConfigGet and meiCanNodeConfigSet allow the user to modify the current configuration of the CAN Processor, and meiCanFlashConfigGet, meiCanFlashConfigSet, meiCanFlashNodeConfigGet, and meiCanFlashNodeConfigSet functions allow the user to modify the configuration that the CAN system will use after the next reset.

The MEICanVersion structure returns the version information about the CAN system on a controller.

After the CAN processor has finished scanning the network, it will have completed the MEICanNodeInfo structures for each node. The user can call the meiCanNodeInfo function to query this initial configuration for each of the nodes.

[Bit Rate](#) | [Transmission Types](#) | [Bus State](#) | [CAN Hardware](#) |
[Node Health](#) | [Emergency Messages](#) | [Handling Events](#) | [XMP Overview](#) |

Methods

Create, Delete, Validate Methods

meiCanCreate	Create Can object
meiCanDelete	Delete Can object
meiCanValidate	Validate Can object

Configuration and Information Methods

[meiCanConfigGet](#)

Get Can's configuration

[meiCanConfigSet](#)

Set Can's configuration

[meiCanFlashConfigGet](#)

Get Can's flash configuration

[meiCanFlashConfigSet](#)

Set Can's flash configuration

[meiCanStatus](#)

Get status of the CAN controller.

[meiCanVersion](#)

Returns the version information about a controller's CAN system.

[meiCanCommand](#)

Get Can's flash configuration

[meiCanNodeConfigGet](#)

Return a copy of the current configuration

[meiCanNodeConfigSet](#)

Update the current configuration that the specified CAN node is using.

[meiCanNodeFlashConfigGet](#)

Get the flash configuration of the Can node

[meiCanNodeFlashConfigSet](#)

Set the flash configuration of the Can node

[meiCanNodeStatus](#)

Get the instantaneous state of the local CAN interface.

[meiCanNodeInfo](#)

Return the node information after the XMP finishes scanning the network.

I/O Methods

[meiCanNodeAnalogInputGet](#)

Get current analog input

[meiCanNodeAnalogOutputGet](#)

Get current analog output

[meiCanNodeAnalogOutputSet](#)

Set current analog output

[meiCanNodeDigitalInputGet](#)

Get the current state of the digital input bit.

[meiCanNodeDigitalInputsGet](#)

Get the current state of ALL the digital input bits.

[meiCanNodeDigitalOutputGet](#)

Get the current state of the digital output bit.

[meiCanNodeDigitalOutputsGet](#)

Get the current state of all the digital output bits.

[meiCanNodeDigitalOutputSet](#)

changes the state of the digital output bit.

[meiCanNodeDigitalOutputsSet](#)

Changes the current state of all the digital output bits.

Event Methods

[meiCanEventNotifyGet](#)

Get event mask of events for which host notification has been requested

[meiCanEventNotifySet](#)

Set event mask of events for which host notification will be requested

Firmware Methods

[meiCanFirmwareDownload](#)

Downloads firmware to the Can controller

[meiCanFirmwareErase](#)

Erases firmware on the Can controller

[meiCanFirmwareUpload](#)

Uploads firmware from the Can controller

Memory Methods

[meiCanMemory](#)

Get address to Can's memory

[meiCanMemoryGet](#)

Copy data from Can memory to application memory

[meiCanMemorySet](#)

Copy data from application memory to Recorder memory

Data Types

[MEICanBitRate](#)

[MEICanBusState](#)

[MEICanCallback](#)

[MEICanCommand](#)

[MEICanCommandType](#)

[MEICanConfig](#)

[MEICanDigitalIO](#)

[MEICanHealthType](#)

[MEICanMessage](#)

[MEICanNodeConfig](#)

[MEICanNodeInfo](#)

[MEICanNodeStatus](#)

[MEICanNodeType](#)

[MEICanNMTState](#)

[MEICanStatus](#)

[MEICanTransmissionType](#)

[MEICanVersion](#)

mpiCanCreate

Declaration `MEICan meiCanCreate(MPISControl long control, network);`

Required Header `stdmei.h`

Description `CanCreate` creates a CAN object handle that is used subsequently to address the CAN network on this controller. You will need a valid CAN handle to use the MPI's CANOpen functionality.

control	a handle to the controller object that contains the CAN object.
network	the number of the CAN network on the specified controller. For most controllers with a single CAN network interface this will be zero. Network numbers are zero based.

Example Code

The following sample code shows the creation and destruction of a valid CAN handle.

```
MPISControl ControlHandle;
MEICan CANHandle;
long Result;

/* Create, validate and initialise a handle to the controller. */
ControlHandle = mpiControlCreate( MPISControlTypeDEFAULT, NULL );
Result = mpiControlValidate( ControlHandle );
assert( Result == MPIMessageOK );

Result = mpiControlInit( ControlHandle );
assert( Result == MPIMessageOK );

/* Create and validate a handle to the CAN object. */
CANHandle = meiCanCreate( ControlHandle, 0 );
Result = meiCanValidate( CANHandle );
assert( Result == MPIMessageOK );

/* Use the CAN object here */

/* Delete the CAN and Controller objects */
Result = meiCanDelete( CANHandle );
assert( Result == MPIMessageOK );
Result = mpiControlDelete( ControlHandle );
assert( Result == MPIMessageOK );
```

Return Values

handle	Handle to the CAN object created or MPIHandleVOID.
MPIHandleVOID	if the object could not be created

See Also [mpiCanDelete](#) | [mpiCanValidate](#)

meiCanDelete

Declaration long **meiCanDelete**([MEICan](#) can);

Required Header stdmei.h

Description **CanDelete** deletes the specified CAN object.

can	handle to the CAN object to delete.

Example Code

See [meiCanCreate](#) for an example of how to use meiCanDelete.

Return Values

MPIMessageOK	if <i>CanDelete</i> successfully deletes a CAN object and invalidates its handle

See Also [meiCanCreate](#) | [meiCanValidate](#)

Required Header `stdmei.h`

Description	CanValidate validates the specified CAN handle.

Example Code

Return Values

See Also [meiCanNodeInfo](#) | [meiCanNodeStatus](#)

meiCanConfigGet

```
Declaration      long meiCanConfigGet(MEICan      can,
                        MEICanConfig* config);
```

Required Header

Description	CanConfigGet returns a copy of the current configuration of the CAN controller.

can	a handle to the CAN object
config	a pointer to the CAN configuration structure that will be filled in by this function..

Return Values

MPIMessageOK	if <i>CanConfigGet</i> successfully returns the copy of the current configuration of the CAN controller.

See Also [meiCanConfigSet](#)

meiCanFlashConfigGet

Declaration

```
long meiCanFlashConfigGet(MEICan can,  
                           void* flash,  
                           MEICanConfig* config);
```

Required Header

stdmei.h

Description

CanFlashConfigGet returns a copy of the current flash configuration that the CAN controller is using.

can	handle to the CAN object
flash	normally NULL
config	a pointer to the CAN configuration structure that will be filled in by this function.

Return Values

MPIMessageOK	if <i>CanFlashConfigGet</i> successfully returns a copy of the current flash configuration that the CAN controller is using.

See Also [meiCanFlashConfigSet](#)

meiCanFlashConfigSet

```
Declaration      long  meiCanFlashConfigSet(MEICan      can,  
                  void*      flash,  
                  MEICanConfig*  config);
```

Required Header

Description	CanFlashConfigSet updates the current flash configuration that the CAN sun-system is using.

can	handle to the CAN object
flash	normally NULL
config	a pointer to the CAN configuration structure that will be filled in by this function.

Return Values

MPIMessageOK	if <i>CanFlashConfigSet</i> successfully updates the current flash configuration that the CAN sun-system is using.

See Also [meiCanFlashConfigGet](#)

Required Header `stdmei.h`

can	handle to the CAN object
node	the node number of the CANOpen node.
status	a pointer to where this function will put the status.

MPIMessageOK	if <i>CanStatus</i> successfully gets the instantaneous state of the local CAN interface to the CAN network.

file:///D:/pdfs/030100/html/Software-MPI/docs/CAN/Method/sts2.htm [7/22/2004 1:40:45 PM]

meiCanConfigSet

```
Declaration      long meiCanConfigSet(MEICan can,  
                  MEICanConfig* config);
```

Required Header `stdmei.h`

Description	CanConfigSet updates the current configuration of the CAN controller.

can	a handle to the CAN object
config	a pointer to the CAN configuration structure containing the new configuration.

Return Values

MPIMessageOK	if <i>CanConfigSet</i> successfully updates the current configuration of the CAN controller.

See Also [meiCanConfigGet](#)

Required Header `stdmei.h`

can	handle to the CAN object
version	a pointer to where this function will put the version information.

MPIMessageOK	if <i>CanVersion</i> successfully returns the version of the firmware being used by the CAN controller.

file:///D:/pdfs/030100/html/Software-MPI/docs/CAN/Method/ver2.htm [7/22/2004 1:40:46 PM]

Required Header `stdmei.h`

can	a handle to the CAN object
command	a pointer to a structure which contains the details of the command to be issued. On the functions return, it will contain the result of the requested command.

MPIMessageOK	if <i>CanCommand</i> successfully performs the set of specified commands.

file:///D:/pdfs/030100/html/Software-MPI/docs/CAN/Method/cmd2.htm [7/22/2004 1:40:46 PM]

Required Header `stdmei.h`

can	a handle to the CAN object
node	the node number of the CANOpen node
nodeConfig	a pointer to the CAN node configuration structure that will be filled in by this function.

MPIMessageOK	if <i>CanNodeConfigGet</i> successfully returns the copy of the current configuration of the specified CAN node.

file:///D:/pdfs/030100/html/Software-MPI/docs/CAN/Method/ndcfget2.htm [7/22/2004 1:40:47 PM]

Declaration	long meiCanNodeConfigSet (MEICan	can ,
	long	node ,
	MEICanNodeConfig *	nodeConfig);

Required Header

Description	CanNodeConfigSet updates the current configuration that the specified CAN node is using.

can	a handle to the CAN object
node	the node number of the CANOpen node
nodeConfig	a pointer to the CAN node configuration structure containing the new configuration.

Return Values

MPIMessageOK	if <i>CanNodeConfigSet</i> successfully updates the current configuration of the specified CAN node.

See Also [meiCanNodeConfigGet](#) | [meiCanConfigGet](#) | [meiCanConfigSet](#)

Declaration	long meiCanNodeFlashConfigGet (MEICan void* long MEICanNodeConfig *	can, flash, node, nodeConfig);

Description	CanNodeFlashConfigGet returns a copy of the current flash configuration of the CAN controller.

can	a handle to the CAN object
flash	normally NULL
node	the node number of the CANOpen node
nodeConfig	a pointer to the CAN node configuration structure that will be filled in by this function

MPIMessageOK	if <i>CanNodeFlashConfigGet</i> successfully returns the copy of the current flash configuration of the CAN controller.

file:///D:/pdfs/030100/html/Software-MPI/docs/CAN/Method/ndflacfgset2.htm [7/22/2004 1:40:47 PM]

meiCanNodeFlashConfigSet

Declaration

long meiCanNodeFlashConfigSet([MEICan](#) can, void* flash, long node, [MEICanNodeConfig](#)* nodeConfig);

Required Header

stdmei.h

Description

[CanNodeFlashConfigSet](#) updates the current flash configuration for the node.

can	a handle to the CAN object
flash	normally NULL
node	the node number of the CANOpen node
nodeConfig	a pointer to the CAN node configuration structure containing the new configuration.

Return Values	
MPIMessageOK	if <i>CanNodeFlashConfigSet</i> successfully updates the current flash configuration for the node.

See Also

[meiCanNodeFlashConfigGet](#)

Required Header

can	handle to the CAN object
node	the node number of the CANOpen node.
nodeStatus	a pointer to where this function will put the node status.

MPIMessageOK

See Also [meiCanNodeInfo](#) | [meiCanStatus](#)

```
Declaration      long meiCanNodeInfo(MEICan can,  
                  long node,  
                  MEICanNodeInfo* nodeInfo);
```

Required Header `stdmei.h`

Description

CanNodeInfo returns the node information for the specified node on the CAN network that was generated when the XMP finished scanning the network.

can	handle to the CAN object
node	the filename of the CAN controller firmware (*.out file).
nodeInfo	a pointer to where this function will put the node information.

Return Values

MPIMessageOK	if <i>CanNodeInfo</i> successfully returns the node information for the specified node.

See Also [meiCanNodeStatus](#) | [meiCanStatus](#)

meiCanNodeAnalogInputGet

Declaration

long meiCanNodeAnalogInputGet([MEICan](#) can,
long node,
long index,
double* data);

Required Header

stdmei.h

Description

CanNodeAnalogInputGet gets the current analog input from the specified CAN Node. The analog data returned is scaled to between ± 1.0 .

can	handle to the CAN object
node	the node number of the CANOpen node.
index	the index to the analog input on the node.
data	a pointer to where the current analog input is returned.

Return Values	
MPIMessageOK	if <i>CanNodeAnalogInputGet</i> successfully gets the current analog input from the specified CAN Node.

See Also

meiCanNodeAnalogOutputGet

Declaration long [meiCanNodeAnalogOutputGet](#) ([MEICan](#) **can,**
 long **node,**
 long **index,**
 double* **data**);

Required Header stdmei.h

Description [CanNodeAnalogOutputGet](#) gets the current analog output from the specified CAN node and channel. The analog data returned is scaled to between ± 1.0 .

can	handle to the CAN object
node	the node number of the CANOpen node.
index	the index to the analog input on the node.
data	a pointer to where the current analog output is returned.

Return Values	
MPIMessageOK	if <i>CanNodeAnalogOutputGet</i> successfully gets the current analog output from the specified CAN Node and channel.

See Also [meiCanNodeAnalogOutputSet](#)

meiCanNodeAnalogOutputSet

Declaration

long meiCanNodeAnalogOutputSet([MEICan](#) can,
 long node,
 long index,
 double* data);

Required Header

stdmei.h

Description

[CanNodeAnalogOutputSet](#) sets the current analog output for the specified CAN node and channel. The analog data used is assumed to be between ± 1.0 .

can	handle to the CAN object
node	the node number of the CANOpen node.
index	the index to the analog input on the node.
data	the new analog value to be output.

Return Values	
MPIMessageOK	if <i>CanNodeAnalogOutputSet</i> successfully sets the current analog output for the specified CAN Node and channel.

See Also

[meiCanNodeAnalogOutputGet](#)

```
Declaration      long  meiCanNodeDigitalInputGet( MEICan    can,
                                           long      node,
                                           long      bit,
                                           long*     data);
```

Description	CanNodeDigitalInputGet gets the current state of the digital input bit on the specified CAN node.

can	handle to the CAN object
node	the node number of the CANOpen node
bit	Which bit on this node
data	a pointer to where the current digital bit is returned

The following sample code shows how to interrogate the current state of a single digital input bit on a controller. The variable **Bit** will contain either one or zero depending on the electrical signal being applied to the input pin on the CANOpen node. See [meiCanCreate](#) on how to create the CANHandle.

```
long Bit;
long Result;
Result = meiCanNodeDigitalInputGet(CANHandle,
                                     3, /*node*/
                                     0, /*bit*/
                                     &Bit );
```

Return Values

MPIMessageOK	if <i>CanNodeAnalogInputGet</i> successfully gets the current analog input from the specified CAN Node.

See Also [meiCanCreate](#)

Required Header

```
stdmei.h
```

(Not to be confused with `meiCanNodeDigitalInputGet`.)

Return Values

See Also [meiCanNodeDigitalInputGet](#)

```
long meiCanNodeDigitalOutputGet(MEICan can,
                                long node,
                                long bit,
                                long* data);
```

Required Header `stdmei.h`

Description	CanNodeAnalogOutputGet gets the current state of the digital output bit on the specified CAN Node.

(Not to be confused with [meiCanNodeDigitalOutputsGet](#).)

can	handle to the CAN object
node	the node number of the CANOpen node.
bit	which bit on this node.
data	a pointer to where the current digital bit is returned.

MPIMessageOK	if <i>CanNodeDigitalOutputGet</i> successfully gets the current digital output bit on the specified CAN Node.

See Also [meiCanNodeDigitalOutputSet](#) | [meiCanNodeDigitalOutputsGet](#) | [meiCanNodeDigitalOutputsSet](#)

Required Header

```
stdmei.h
```

(Not to be confused with [meiCanNodeDigitalOutputsGet](#).)

Return Values

MPIMessageOK	if <i>CanNodeDigitalOutputsGet</i> successfully gets the current digital output bits on the specified CAN node.

See Also [meiCanNodeDigitalOutputGet](#) | [meiCanNodeDigitalOutputSet](#) | [meiCanNodeDigitalOutputsSet](#)

meiCanNodeDigitalOutputSet

Declaration

long meiCanNodeDigitalOutputSet([MEICan](#) can, long node, long bit, long data);

Required Header

stdmei.h

Description

CanNodeDigitalOutputSet changes the state of the digital output bit on the specified CAN Node.

(Not to be confused with [meiCanNodeDigitalOutputsSet](#).)

can	handle to the CAN object
node	the node number of the CANOpen node.
bit	which bit on this node.
data	the new state of the digital bit.

Return Values	
MPIMessageOK	if <i>CanNodeDigitalOutputSet</i> successfully changes the state of the digital output bit on the specified CAN Node.

See Also

[meiCanNodeDigitalOutputGet](#) | [meiCanNodeDigitalOutputsGet](#) | [meiCanNodeDigitalOutputsSet](#)

Required Header

```
stdmei.h
```

(Not to be confused with [meiCanNodeDigitalOutputSet](#).)

Return Values

MPIMessageOK	if <i>CanNodeDigitalOutputsSet</i> successfully changes the current state of all the digital output bits on the specified CAN node.

See Also [meiCanNodeDigitalOutputGet](#) | [meiCanNodeDigitalOutputSet](#) | [meiCanNodeDigitalOutputsGet](#)

meiCanEventNotifyGet

Declaration

long meiCanEventNotifyGet([MEICan](#)
[MPIEventMask](#)
void

can,
*eventMask,
*external);

Required Header

stdmei.h

Description

[CanEventNotifyGet](#) gets the current CAN event mask.

can	handle to the CAN object.
*eventMask	a pointer to the MPI event mask that will be filled in by this function.
*external	external points to an implementation specific structure. Since there is currently no implementation specific data, NULL should be used.

Return Values

MPIMessageOK

if *EventNotifyGet* successfully gets the current CAN event mask.

See Also

[meiCanNotifySet](#)

meiCanEventNotifySet

Declaration

long meiCanEventNotiySet([MEICan](#)
[MPIEventMask](#)
void
can,
eventMask,
*external);

Required Header

stdmei.h

Description

[CanEventNotifySet](#) updates the current CAN event mask.

can	handle to the CAN object.
eventMask	a pointer to the new MPI event mask that will be filled in by this function.
*external	external points to an implementation specific structure. Since there is currently no implementation specific data, NULL should be used.

Return Values

MPIMessageOK

if *CanEventNotifySet* successfully sets the current CAN event mask.

See Also

[meiCanEventNotifyGet](#)

meiCanFirmwareDownload

Declaration

long meiCanFirmwareDownload([MEICan](#) can,
char* filename,
[MEICanCallback](#) callback);

Required Header

stdmei.h

Description

CanFirmwareDownload allows the user to upgrade the CAN controller's firmware.

This operation will take some time (between 10 and 30 seconds) to perform the download process. Therefore, the callback function is provided to allow the current status of the download operation to be reported to the calling application and to also allow the calling application to abort the download if required. The callback function passes the progress of the download process to the calling application. The calling applications normally returns a 0 unless it wants to abort the upgrade. If the upgrade is aborted, it returns a 1.

can	handle to the CAN object
filename	the filename of the CAN controller firmware (*.out file).
callback	a pointer to the call back function. (Pass an address of zero if you do not have a callback function.)

Return Values

MPIMessageOK if *CanFirmwareDownload* successfully uploads the CAN controller's firmware.

See Also

[meiCanFirmwareErase](#) | [meiCanFirmwareUpload](#)

meiCanFirmwareErase

Declaration long **meiCanFirmwareErase**([MEICan](#) **can**);

Required Header stdmei.h

Description **CanFirmwareErase** allows the user to erase the CAN controllers firmware.

Return Values

MPIMessageOK	if <i>CanFirmwareErase</i> successfully erases the CAN controller's firmware.

See Also [meiCanFirmwareDownload](#) | [meiCanFirmwareUpload](#)

meiCanFirmwareUpload

Declaration

long meiCanFirmwareUpload([MEICan](#)
char*
[MEICanCallback](#)
can,
filename,
callback);

Required Header

stdmei.h

Description

CanFirmwareUpload allows the user to get a copy of the current CAN controller’s firmware.

This operation will take some time (between 10 and 30 seconds) to perform the upload process. Therefore, the callback function is provided to allow the current status of the upload operation to be reported to the calling application and to also allow the calling application to abort the upgrade (if required). The callback function passes the progress of the upgrade process to the calling application. The calling applications normally returns 0 unless it wants to abort the upgrade. If the upgrade is aborted, it returns a 1.

can	handle to the CAN object
filename	the filename of the CAN controller firmware (*.out file).
callback	a pointer to the call back function. (Pass an address of zero if you do not have a callback function.)

Return Values	
MPIMessageOK	if <i>CanFirmwareUpload</i> successfully retrieves a copy of the current CAN controller's firmware.

See Also

[meiCanFirmwareErase](#) | [meiCanFirmwareDownload](#)

meiCanMemory

Declaration

long meiCanMemory([MEICan](#)
void** can,
memory);

Required Header

stdmei.h

Description

CanMemory returns a pointer to the base of the CAN processors DPR. This function is generally not used and is provided for implementing advanced features of the MPI.

can	handle to the CAN object
memory	a pointer to the base of the CAN processors DPR.

Return Values

MPIMessageOK if *CanMemory* successfully returns a pointer to the base of the CAN processors DPR.

See Also

[meiCanMemoryGet](#) | [meiCanMemorySet](#)

meiCanMemoryGet

Declaration

long meiCanMemoryGet([MEICan](#) can,
 void* dst,
 void* src,
 long count);

Required Header

stdmei.h

Description

CanMemoryGet copies the specified number of bytes from controller’s memory to the application's memory. This function is generally not used and is provided for implementing advanced features of the MPI.

can	handle to the CAN object
dst	the base address of the destination
src	the base address of the source
count	the number of bytes to copy

Return Values	
MPIMessageOK	if <i>CanMemoryGet</i> successfully copies the specified number of bytes to the application's memory.

See Also

[meiCanMemory](#) | [meiCanMemorySet](#)

meiCanMemorySet

Declaration

long meiCanMemorySet([MEICan](#) can, void* dst, void* src, long count);

Required Header

stdmei.h

Description

CanMemorySet copies the specified number of bytes from the application's memory to the controller's memory. This function is generally not used and is provided for implementing advanced features of the MPI.

can	handle to the CAN object
dst	the base address of the destination
src	the base address of the source
count	the number of bytes to copy

Return Values	
MPIMessageOK	if <i>CanMemorySet</i> successfully copies the specified number of bytes to the controller's memory.

See Also

[meiCanMemory](#) | [meiCanMemoryGet](#)

MEICanBitRate

MEICanBitRate

```
typedef enum {  
    MEICanBitRate1000K = 0,  
    MEICanBitRate800K,  
    MEICanBitRate500K,  
    MEICanBitRate250K,  
    MEICanBitRate125K,  
    MEICanBitRate50K,  
    MEICanBitRate20K,  
    MEICanBitRate10K  
} MEICanBitRate;
```

Description

CanBitRate enumerates all the valid bit rates that the CANOpen interface can use. These are the recommended bit rates that the CANOpen standard defines.

For more information see the [Bit Rate](#) section.

See Also

MEICanBusState

MEICanBusState

```
typedef enum {  
    MEICanBusStateOFF,  
    MEICanBusStatePASSIVE,  
    MEICanBusStateOPERATIONAL  
} MEICanBusState;
```

Description

CanBusState enumerates the bus states that the controller's CAN interface can take.

To see how the CanBusState is displayed in Motion Console, [click here](#).

See Also Documentation on [CAN Bus State](#)

MEICanCallback

MEICanCallback

```
typedef long (*MEICanCallback)(long percentage);
```

Description

CanCallback is the definition of a call back function used during the firmware download.

See Also

MEICanCommand

MEICanCommand

```
typedef struct MEICanCommand {
    MEICanCommandType type;
    long data[6];
} MEICanCommand;
```

Description **CanCommand** holds the command request and response for an meiCanCommand.

type	The type of CAN command.
data	Data associated with the command.

See Also [meiCanCommand](#)

MEICanCommandType

MEICanCommandType

```
typedef enum {
    MEICanCommandTypeSDO_READ,
    MEICanCommandTypeSDO_WRITE,
    MEICanCommandTypeCLEAR_STATUS_BITS,
    MEICanCommandTypeBUS_START,
    MEICanCommandTypeBUS_STOP,
    MEICanCommandTypeNMT_ENTER_PRE_OPERATIONAL,
    MEICanCommandTypeNMT_START_REMOTE_NODE,
    MEICanCommandTypeNMT_STOP_REMOTE_NODE,
    MEICanCommandTypeNMT_RESET_NODE,
    MEICanCommandTypeNMT_RESET_COMMUNICATION,
} MEICanCommandType;
```

Description

CanCommandType enumerates the different type of commands that can be used with meiCanCommand.

MEICanCommandTypeSDO_READ

This command reads the remote nodes object dictionary using the SDO protocol.

Command data:

data[0] = Node
data[1] = Index
data[2] = SubIndex
data[3] = Length

Returned data:

data[0] = Error code
data[4] = Low Data word
data[5] = High Data word

MEICanCommandTypeSDO_WRITE

This issues the CANOpen NMT command "Enter Pre-Operational" to a node.

Command data:

data[0] = Node number, (0 broadcasts to all nodes)

Returned data:

data[0] = Error code

MEICanCommandTypeNMT_START_REMOTE_NODE

This issues the CANOpen NMT command "Start Remote Node" to a node.

Command data:

data[0] = Node number, (0 broadcasts to all nodes)

Returned data:

data[0] = Error code

MEICanCommandTypeNMT_STOP_REMOTE_NODE

This issues the CANOpen NMT command "Stop Remote Node" to a node.

Command data:

data[0] = Node number, (0 broadcasts to all nodes)

Returned data:

data[0] = Error code

MEICanCommandTypeNMT_RESET_NODE

This issues the CANOpen NMT command "Reset Node" to a node.

Command data:

data[0] = Node number, (0 broadcasts to all nodes)

Returned data:

data[0] = Error code

MEICanCommandTypeNMT_RESET_COMMUNICATION

This issues the CANOpen NMT command "Reset Communication" to a node.

Command data:

data[0] = Node number, (0 broadcasts to all nodes)

Returned data:

data[0] = Error code

See Also [meiCanCommand](#)

This command writes to a remote nodes object dictionary using the SDO protocol.

Command data:

data[0] = Node

data[1] = Index

data[2] = SubIndex

data[3] = Length

data[4] = Low Data word

data[5] = High Data word

Returned data:

data[0] = Error code

MEICanCommandTypeCLEAR_STATUS_BITS

Clear selected MEICanStatusBits.

Command data:

data[0], Bit map of MEICanStatusBits to clear.

Returned data:

data[0] = Error code

MEICanCommandTypeBUS_START

This puts the CAN bus into operational state if it is Bus off.

Command data:

None

Returned data:

data[0] = Error code

MEICanCommandTypeBUS_STOP

This puts the CAN bus into operational state if it is Bus off.

Command data:

None

Returned data:

data[0] = Error code

MEICanCommandTypeNMT_ENTER_PRE_OPERATIONAL

MEICanConfig

MEICanConfig

```
typedef struct MEICanConfig {
    MEICanBitRate      bitRate;
    unsigned long      cyclicPeriod;
    unsigned long      healthPeriod;
    unsigned long      nodeNumber;
    unsigned long      inhibitTime;
} MEICanConfig;
```

Description

CanConfig holds the configuration of the CAN object. The default state for this structure is held in the controller's flash. Use the `meiCanConfigGet/Set` and `meiCanNodeConfigGet/Set` to interrogate and change to what the CAN system is currently using or the default.

bitRate	The bit rate the CAN bus uses. See also CAN Bit Rate .
cyclicPeriod	The period (milliseconds) between sending consecutive SYNC messages. A value of zero will disable the SYNC messages from being produced. See also CAN Transmission Types .
healthPeriod	The period (milliseconds) used for checking the health of nodes. A value of zero will disable the health checking protocol. For nodes that use the node guarding protocol, this is the node guarding period. For nodes that use the heartbeating protocol, this is the heartbeat consumer time (the heartbeat producers are half this period). See also CAN Node Health .
nodeNumber	The node number of the controller on the CAN network. CANOpen requires that the master node has a valid node number to implement the heartbeat protocol. See also CAN Node Numbers .
inhibitTime	The global time used for the node health protocols. See also CAN Transmission Types .

See Also [meiCanConfigGet](#) | [meiCanConfigSet](#) | [meiCanNodeConfigGet](#) | [meiCanNodeConfigSet](#) |

MEICanDigitalIO

MEICanDigitalIO

```
typedef struct MEICanDigitalIO {  
    unsigned long    data[2];  
} MEICanDigitalIO;
```

Description **CanDigitalIO** holds the state of all the digital inputs or outputs on a CANOpen node.

NOTE: the maximum number of inputs or outputs on a single node supports is 64.

data	Data associated with the command.

See Also

MEICanHealthType

MEICanHealthType

```
typedef enum {  
    MEICanHealthTypeNODE_GUARDING,  
    MEICanHealthTypeHEART_BEATING  
} MEICanHealthType;
```

Description

CanHealthType is used to report the health protocol that the XMP is using with each node.

See Also

MEICanMessage

MEICanMessage

```
typedef enum {
    MEICanMessageFIRMWARE_INVALID,
    MEICanMessageFIRMWARE_VERSION,
    MEICanMessageNOT_INITIALIZED,
    MEICanMessageIO_NOT_SUPPORTED,
    MEICanMessageFILE_FORMAT_ERROR,
    MEICanMessageUSER_ABORT,
    MEICanMessageCOMMAND_PROTOCOL,
    MPICanMessageINTERFACE_NOT_FOUND,
    MEICanMessageNODE_DEAD,
    MEICanMessageSDO_TIMEOUT,
    MEICanMessageSDO_ABORT,
    MEICanMessageSDO_PROTOCOL,
    MEICanMessageTX_OVERFLOW,
    MEICanMessageRTR_TX_OVERFLOW,
    MEICanMessageRX_BUFFER_EMPTY,
    MEICanMessageBUS_OFF,
    MEICanMessageSIGNATURE_INVALID,
} MEICanMessage;
```

Description

MEICanMessageFIRMWARE_INVALID

The CAN firmware is not valid. This message code is returned by [meiCanCreate\(...\)](#) if the CAN hardware bootloader detects no firmware has been loaded or the firmware signature is not recognized. To correct this problem, download valid firmware with [meiCanFirmwareDownload\(...\)](#).

MEICanMessageFIRMWARE_VERSION

The CAN firmware version does not match the software version. This message code is returned by [meiCanCreate\(...\)](#), [meiCanFirmwareDownload\(...\)](#), or [meiCanFirmwareUpload\(...\)](#) if the CAN firmware version is not compatible with the MPI library. To correct this problem, download the proper firmware version with [meiCanFirmwareDownload\(...\)](#).

MEICanMessageNOT_INITIALIZED

The CAN firmware did not initialize. This message code is returned by [meiCanCreate\(...\)](#) if the controller did not copy the configuration structure from flash to memory after power-on or controller reset. To correct this problem, verify the controller firmware is correct and the controller hardware is operating properly.

MEICanMessageIO_NOT_SUPPORTED

The CAN node does not support the specified I/O. This message code is returned by CAN methods that read/write to a digital or analog input/output that is out of range. To prevent this problem, specify a supported I/O bit.

MEICanMessageFILE_FORMAT_ERROR

The CAN firmware file format has an error. This message code is returned by [meiCanFirmwareDownload\(...\)](#) if the specified file has an error in its internal headers. This indicates a corrupted file. To correct this problem, use the original CAN firmware file or reinstall the software distribution.

MEICanMessageUSER_ABORT

The CAN firmware loading was aborted. This message code is returned by [meiCanFirmwareDownload\(...\)](#) or [meiCanFirmwareUpload\(...\)](#) when the firmware loading is aborted by the user via the callback function. This message code is returned for application notification. It is not an error.

MEICanMessageCOMMAND_PROTOCOL

The CAN command failed due to a protocol error. This message code is returned by CAN methods that do not get a valid response from a CAN node. To correct this problem, check your CAN nodes for proper operation.

MPICanMessageINTERFACE_NOT_FOUND

The CAN interface is not available. This message code is returned by [meiCanCreate\(...\)](#) if the specified controller does not support a CAN network interface. To correct this problem, use a controller that has a CAN interface.

MEICanMessageNODE_DEAD

The CAN node does not respond. This message code is returned by CAN methods that read/write from a CAN node and the node fails the health check. This message code indicates a node hardware or network connection problem. To correct this problem, verify the node operation and network connections.

MEICanMessageSDO_TIMEOUT

The CAN command failed due to a timeout. This message code is returned by CAN methods that do not get a response from a CAN node within the timeout period. To correct this problem, check your CAN nodes for proper operation.

MEICanMessageSDO_ABORT

The CAN command failed due to a user abort. This message code is returned by CAN methods when an SDO transaction is aborted.

MEICanMessageSDO_PROTOCOL

The CAN command failed due to an SDO protocol error. This message code is returned by CAN methods when an SDO transaction fails because the node did not conform to the CANOpen protocol.

MEICanMessageTX_OVERFLOW

The controller's transmit buffer overflowed. This message code is returned by CAN methods that failed to transmit a message due to an internal memory buffer overflow.

MEICanMessageRTR_TX_OVERFLOW

The controller's transmit buffer overflowed. This message code is returned by CAN methods that failed to transmit a message due to an internal memory buffer overflow.

MEICanMessageRX_BUFFER_EMPTY

The controller's receive buffer is empty. This message code is returned by CAN methods that expected to get a response from a CAN node, but the controller's receive buffer was empty.

MEICanMessageBUS_OFF

The CAN network bus is in the off state. This message code is returned by CAN methods that are not able to use the CAN network because the bus is off. To correct this problem, verify the node operation and network connections.

MEICanMessageSIGNATURE_INVALID

When initialising the CAN system, some tests are performed to make sure that the CAN processor is returning a valid signature value. If an unexpected signature is returned, this error message is returned. A probable cause for this error is that the bootloader is invalid. To correct this problem, you will need to return the controller to MEI to fix the bootloader.

See Also

MEICanNodeConfig

MEICanNodeConfig

```
typedef struct MEICanNodeConfig {  
    MEICanTransmissionType digitalOutTransmissionType;  
    MEICanTransmissionType analogOutTransmissionType;  
    MEICanTransmissionType digitalInTransmissionType;  
    MEICanTransmissionType analogInTransmissionType;  
} MEICanNodeConfig;
```

Description

CanNodeConfig is the configuration of each node on the CAN bus. You can select which type of communication (event or cyclic) is to be used for the different types of IO data that a node supports.

For more information, see the [CAN Transmission Types](#) section.

See Also

MEICanNodeInfo

MEICanNodeInfo

```
typedef struct MEICanNodeInfo {
    MEICanNodeType      type;
    unsigned long        digitalInputCount;
    unsigned long        digitalOutputCount;
    unsigned long        analogInputCount;
    unsigned long        analogOutputCount;
    MEICanHealthType    healthType;
    unsigned long        vendorID;
    unsigned long        productCode;
    unsigned long        versionNumber;
    unsigned long        serialNumber;
} MEICanNodeInfo;
```

Description

CanNodeInfo describes how many of the different types of I/O are on this node.

type	An enumeration indicating the type of node found at startup, or MEICanNodeTypeNONE if no node was found.
digitalInputCount	The number of digital inputs supported by this node. The CANOpen protocol only allows the number of digital inputs to be interrogated in multiples of eight, i.e. if a node has two digital inputs then digitalInputCount will return eight. MEI CANOpen SLICE nodes support an extension to the CANOpen protocol that allows the exact number of digital inputs to be returned in this field.
digitalOutputCount	The number of digital outputs supported by this node. The CANOpen protocol only allows the number of digital outputs to be interrogated in multiples of eight, i.e. if a node has two digital outputs then digitalOutputCount will return eight. MEI CANOpen SLICE nodes support an extension to the CANOpen protocol that allows the exact number of digital outputs to be returned in this field.
analogInputCount	The number of analog inputs supported by this node.
analogOutputCount	The number of analog outputs supported by this node.
healthType	The type of health checking protocol being used with this node. Also see CAN Node Health .
vendorId	This is a number read from the node. Vendor ID numbers are unique numbers allocated to each manufacturer of CANOpen nodes. Not all CANOpen nodes support this feature, in which case, these nodes will return zero for this field. MEI CANOpen nodes always return x014Fh.

productCode	This is a number read from the node. The product code is made up of numbers allocated by each manufacturer to uniquely identify their different types of nodes. Not all CANOpen nodes support this feature, in which case, these nodes will return zero for this field. MEI CANOpen SLICE nodes always return x0204h.
versionNumber	This is a number read from the node. The version number identify the version of code running on this CANOpen node. Not all CANOpen nodes support this feature, in which case, these nodes will return zero for this field. MEI CANOpen nodes do support this field.
serialNumber	This is a number read from the node. The serial number uniquely identifies each CANOpen node. Not all CANOpen nodes support this feature, in which case, these nodes will return zero for this field. MEI CANOpen SLICE nodes do support this field and the number is also on the side label of the Network adapter.

See Also

MEICanNodeStatus

MEICanNodeStatus

```
typedef struct MEICanNodeStatus {
    unsigned long    live;
    MEICanNMTState   nmtState;
} MEICanNodeStatus;
```

Description **CanNodeStatus** holds the current status of a node.

live	Set if the node is alive, clear if the node is dead.
nmtState	The current NMT state that the node is reporting.

See Also Documentation on [CAN Node Health](#).

MEICanNodeType

MEICanNodeType

```
typedef enum {  
    MEICanNodeTypeNONE = 0,  
    MEICanNodeTypeIO   = 401  
} MEICanNodeType;
```

Description

CanNodeType enumerates the different types of nodes that the XMP has detected.

MEICanNodeTypeNONE is returned if no node is found or an unsupported node type is detected.

See Also

MEICanNMTState

MEICanNMTState

```
typedef enum {  
    MEICanNMTStateBOOT_UP,  
    MEICanNMTStateSTOPPED,  
    MEICanNMTStateOPERATIONAL,  
    MEICanNMTStatePRE_OPERATIONAL,  
    MEICanNMTStateUNKNOWN,  
} MEICanNMTSTATE;
```

Description

CanNMTState enumerates the NMT (network management) states of a node on a CANOpen network. The XMP's CAN controller will automatically put all nodes into the Operational state during the initialization of the network.

See Also

MEICanStatus

MEICanStatus

```
typedef struct MEICanStatus {
    MEICanBusState busState;
    long transmitErrorCounter;
    long receiveErrorCounter;
    long messageRate;
    long tick;
    long softwareReceiveOverflow;
    long hardwareReceiveOverflow;
} MEICanStatus;
```

Description CanStatus holds the current status of the XMP’s CAN object.

busState	The current bus state of the XMP’s CAN interface.
transmitErrorCounter	The current value of the transmit error counter.
receiveErrorCounter	The current state of the receive error counter.
messageRate	The number of messages received and transmitted per second.
tick	This is incremented every 1ms by the CAN firmware.
softwareReceiveOverflow	This bit will be set if software receive buffer has overflowed. This bit can be cleared by using the CLEAR_STATUS_BITS command.
hardwareReceiveOverflow	This bit will be set if the CAN interface hardware has detected an overflow. This bit can be cleared by using the CLEAR_STATUS_BITS command.

See Also

MEICanTransmissionType

MEICanTransmissionType

```
typedef enum {  
    MEICanTransmissionTypeCYCLIC = 0,  
    MEICanTransmissionTypeEVENT  = 1,  
} MEICanTransmissionType;
```

Description

CanTransmissionType enumerates the transmission types a node can use.

For more information, see the [CAN Transmission Types](#) section.

See Also

MEICanVersion

MEICanVersion

```
typedef struct MEICanVersion {
    long    bootloaderVersion;
    long    firmwareVersion;
    char    firmwareRevision;
    long    firmwareSubRevision;
} MEICanVersion;
```

Description [CanVersion](#) holds the version information about the XMP’s CAN object.

bootloaderVersion	The version number of the CAN bootloader.
firmwareVersion	The CAN firmware version.
firmwareRevsion	The CAN firmware revision.
firmwareSubRevision	The CAN firmware subrevision.

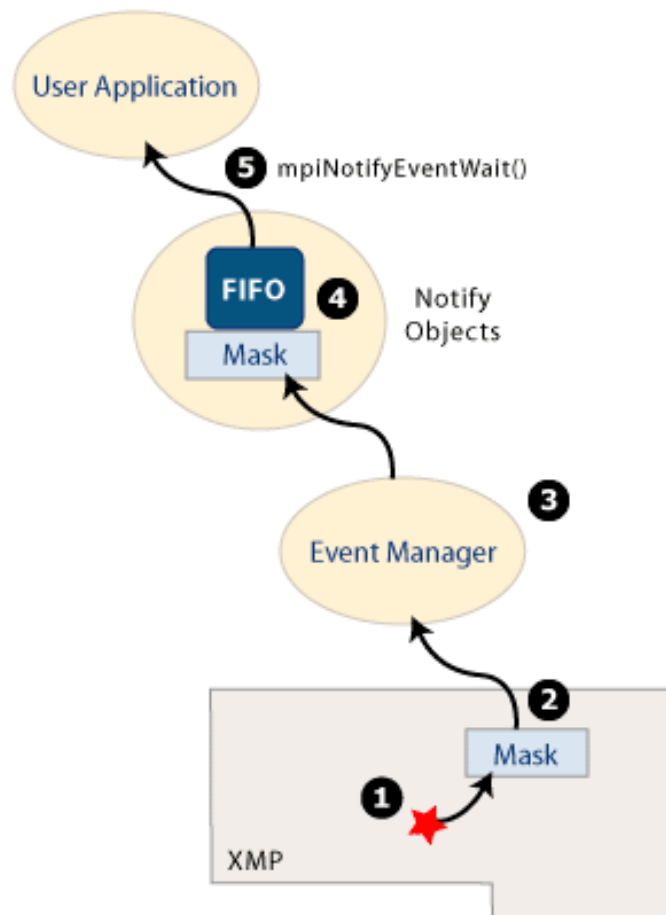
See Also

Handling Events

The CAN interface on the XMP generates many different types of asynchronous events such as:

- a change in the XMP's bus state
- a change in a node's health
- a change in the state of an input node's analog or digital inputs
- an emergency message is transmitted by a node
- a boot message is transmitted by a node
- a lost message is detected by the XMP CAN firmware

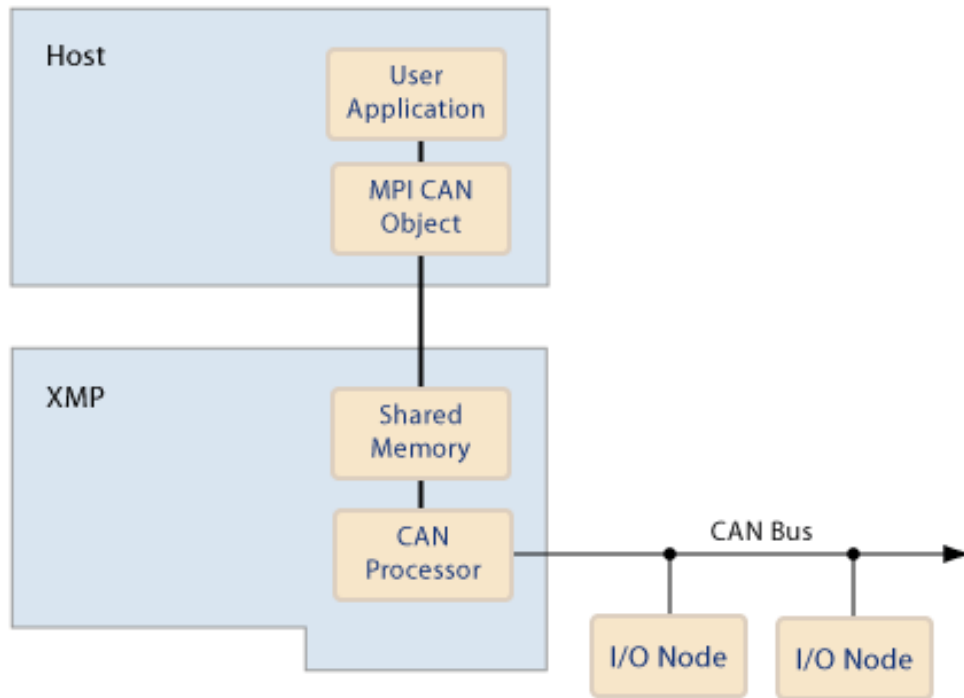
The events above have been appended to the standard MPI event handling scheme in order to provide the user the ability to respond to these events. The diagram below shows an overview of how events are relayed to the user's application.



1. The CANOpen firmware detects one of the CAN events.
2. There is a mask within the XMP firmware that allows only a specified set of events to reach the host. This mask is interrogated and modified with the [meiCanEventNotifyGet](#) and [meiCanEventNotifySet](#) functions.
3. Like all other events in the MPI, the user must install an Event Manager on the host. You will find the serviceCreate and serviceDelete functions from apputils convenient for installing an Event Manager.
4. For each thread that needs to know about CAN events, the user will need to create a notify object, specifying a mask for the required events.
5. The user's application can use the [mpiNotifyEventWait](#) function to either poll or wait for a CAN event to be generated. A valid event returned from mpiNotifyEventWait may also contain extra fields of information relevant to the event produced. (ex: the new bus state or node number).

XMP Overview

In the example below, the XMP uses a dedicated CAN processor to handle the network. This ensures that the motion will not be affected by the CAN network. The XMP operates as a master node on the network with all the I/O nodes being slaves. This arrangement implies that there may only be one XMP on any CAN Network.



The XMP operates as a master node on the network with all the IO nodes being slaves. This arrangement implies that there may only be one XMP on any CAN Network.

CAN Bit Rate

The CANOpen standard defines a set of bit rates that can be supported. Any CANOpen node must support at least one of these bit rates. All the nodes on the CAN network must be operating at the same bit rate. Any of these standard bit rates can be used with the XMP.

Due to the electrical characteristics of a CAN network, the maximum length of a CAN network (and the corresponding drop lengths) is dependent upon the bit rate that is chosen. See the table below.

It is recommended that opto-isolated nodes are used on networks with bus lengths longer than 200m.

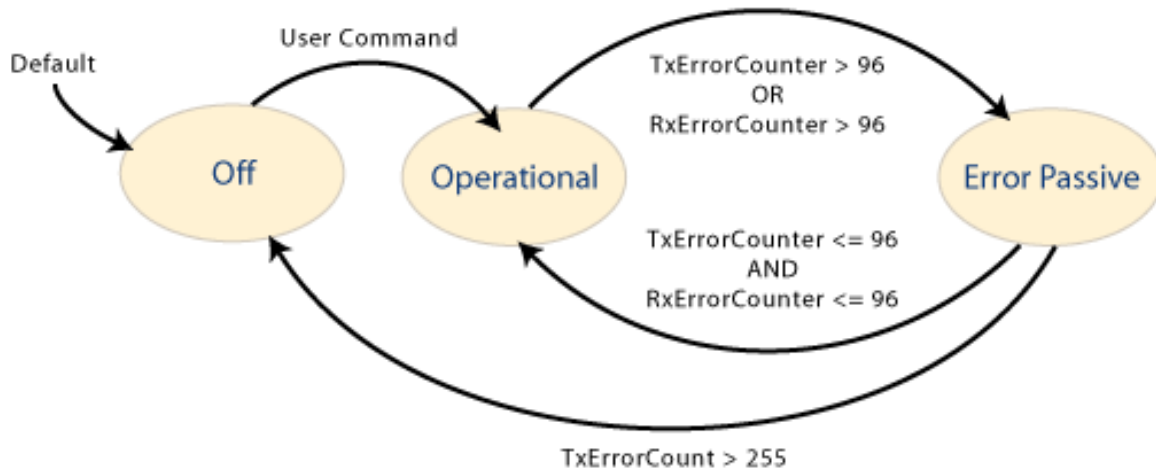
CANOpen Bit Rates

Bit Rate	Max Bus Length (m)	Max Drop Length (m)	Max Cumulative Drop Length (m)
1M	25*	2	10
800k	50*	3	15
500k	100	6	30
250k	250	12	60
125k	500	24	120
50k	1000	60	300
20k	2500	150	750
10k	5000	300	1500

* No opto-isolation

CAN Bus State

All CAN hardware maintains two error counters that are increased when transmit or receive errors are detected, and decreased when successful transmissions or receptions are achieved. In an error free operational system, these counters should be zero. The magnitude of these counters control the following state machine:



When a node is in the **Operational** state it will participate fully with all communications over the network, as the errors increase the CAN hardware will become **Passive** (detecting errors but not generating error messages), before turning **Off** and isolating the node from the network once the TxErrorCount exceeds 255 error messages. This feature allows nodes that are either malfunctioning or not configured correctly to be isolated for the network, thereby allowing the remaining nodes to successfully communicate.

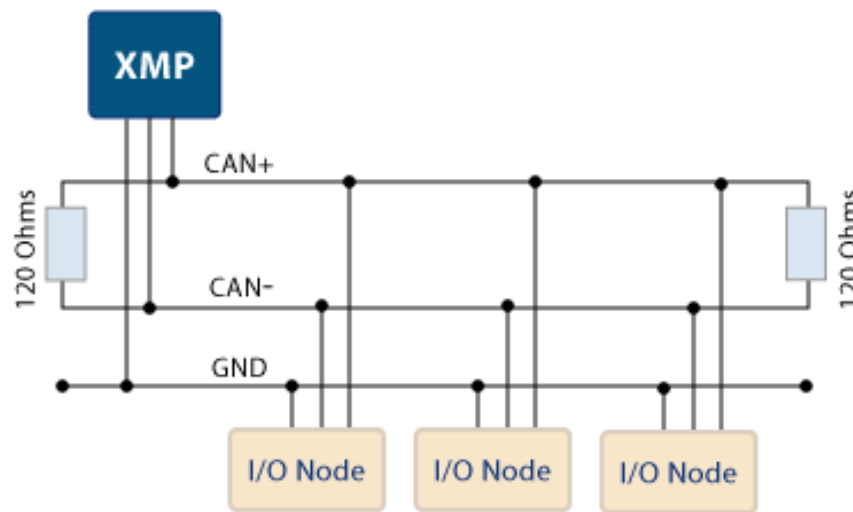
CAN Emergency Messages

Every type of CANOpen node can transmit an emergency message. These messages are designed to report errors and warnings, as well as fatal problems on a node. The contents of these emergency messages are very dependent upon the node manufacturer and node type. To interpret this data, you will need to refer to the node manufacture's data. If an emergency message is generated by a node, the event handling scheme described in the events section below allows the user's application to receive the emergency message data.

CAN Hardware

CANOpen is a serial network that uses a bus topology. The CANOpen bus always contains two signal wires, CAN+ and CAN-, which carry the differential serial data and a ground (GND). It is also common for most CANOpen nodes to provide a shield connection.

Similar to most industrial buses, the signal wires need to be terminated. CANOpen requires a 120ohm resistor at both ends of the main bus. If these resistors are not fitted, the network will not function properly. Some node suppliers build the terminating resistor into the node and provide a jumper or switch to enable it. You will need to check your nodes' datasheets for the inclusion of a terminating resistor. The XMP does not have any terminating resistors.



For pinout information, go to the XMP's [CAN D-9 connector](#) page.

A CANOpen node either has an opto-isolated or non-isolated interface. The use of optoisolation is primarily provided as an EMC countermeasure and is used to cope with potential differences in the ground. These effects are more pronounced for large machines and cable lengths. Therefore, the use of opto-couplers is recommended for bus lengths greater than 200m. The disadvantage of opto-couplers is that they reduce the maximum permissible bus length for a given bit rate.

The XMP CAN interface is available with or without opto-isolation. This option needs to be specified at the time your XMP is ordered.

Most types of nodes require a separate power supply to drive the local logic and the I/O interfaces. For nodes that use opto-isolated interfaces, a separate supply of +7 to 24V needs to be provided to power the interface circuitry. The user must also supply an external 24V to the XMP (CAN_V+) if the opto-isolated interface option is being used.

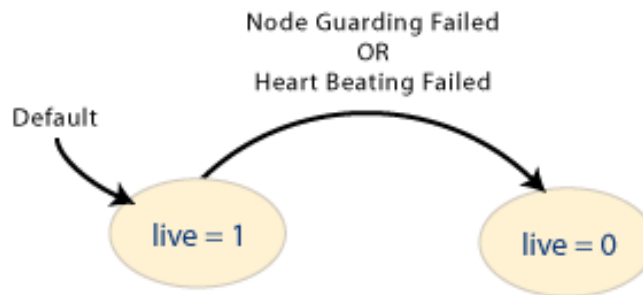
Each node on the network must have a unique node number, in the range of 1 to 127. The node number is commonly set with a bank of DIP switches on each node. If two nodes are given the same node number, network errors are generated and unpredictable problems will be encountered. The node number of the XMP can be changed from the factory default of 1 using the [meiCanConfigSet](#) function.

In order for all nodes to communicate they must all use the same bit rate. Normally the bit rate that a node uses is set by DIP switches. If all of the nodes on a CANOpen network do not use the same bit rate then the whole network or some of the nodes on the network will not work properly. The bit rate of the XMP is set via software [meiCanConfigSet](#). See also [CAN Bit Rate](#).

CAN Node Health

All networks including CAN are vulnerable to faults such as breaks in the bus wiring or loss of power by some of the nodes. CANopen defines two methods for the master node (the XMP in our case) to periodically check the presence of nodes on the network—node guarding and heart beating.

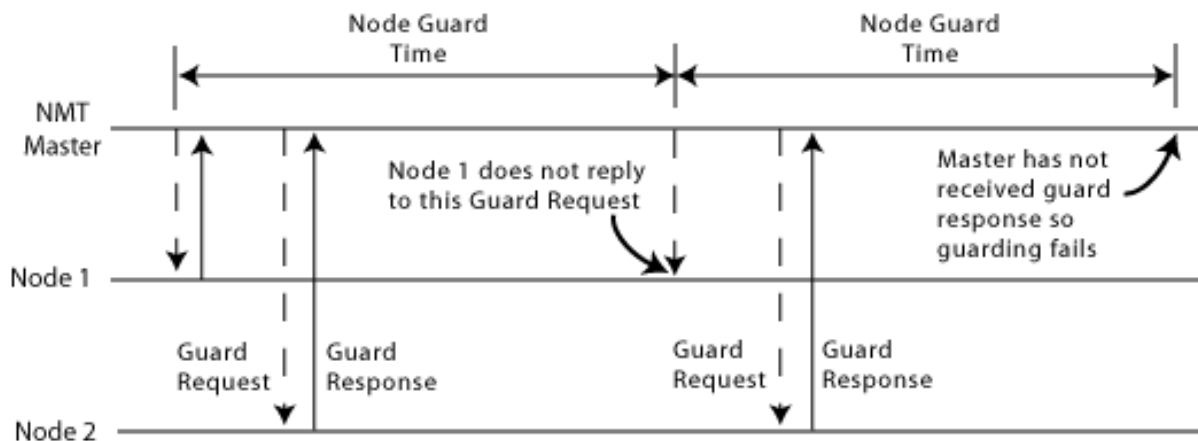
Using these services the XMP can monitor the health of the communications to each of the nodes. The current health of each node is reported in the live field of the [MEICANNodeStatus](#) structure.



It is mandatory for a node to either support the node guarding or heart beating protocols, or to support both. The heartbeat protocol has recently been introduced to CANOpen (in June 1999), and will probably NOT be supported on many nodes, but its adoption is recommended for all new nodes. The XMP's implementation will operate with either protocol and will automatically detect the protocol that each node supports and then use the most appropriate protocol for the CAN network. The healthType field of the [MEICanNodeInfo](#) structure reports the health checking protocol being used with each node.

Node Guarding protocol

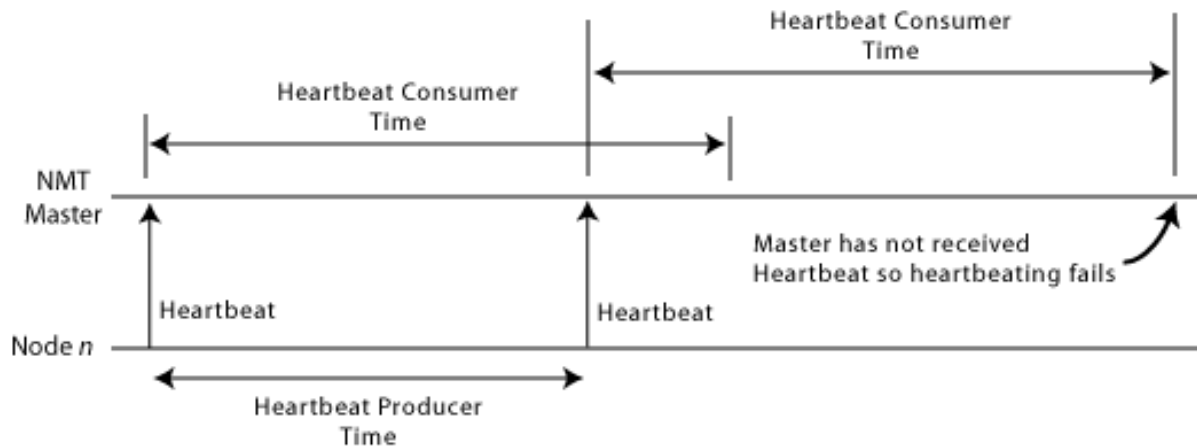
The Node Guarding protocol has the master sending an RTR message to all nodes on the network and checks to see whether a response is received from each of the nodes.



Heart Beating protocol

In the Heart Beating protocol, each node periodically broadcasts a heartbeat message. The period between transmitting the heartbeat messages is half the health period. If the XMP does not receive a message within a specific time window, it generates a heartbeat error for that node.

The advantage of the Heart Beating protocol over the Node Guarding protocol is that the number of messages is reduced in half, thereby freeing up bandwidth for other messages.



Health Period

The healthPeriod field of the [MEICanConfig](#) structure allows the user to specify the Node Guard and Heartbeat times for the health protocols according to the following table. The same period is used for all nodes.

Node Health Times

Protocol Times	Value
Node Guard Time	healthPeriod

Heartbeat Producer Time	healthPeriod / 2
Heartbeat Consumer Time	healthPeriod

For most applications it is recommended that the healthPeriod should be set to ten times the cyclic period.

CAN Node Numbers

Each node on the network must have a unique node number, in the range of 1 to 127. The node number is commonly set with a bank of DIP switches on each node. If two nodes are given the same node number, network errors are generated and unpredictable problems will be encountered. The node number of the XMP can be changed from the factory default of 1 using the [meiCanConfigSet](#) function.

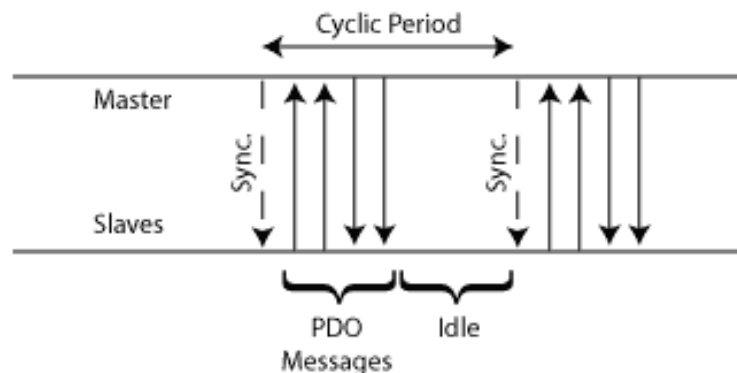
CAN Transmission Types

Introduction

The XMP CANOpen interface uses four messages (serial packets of data on the CAN bus) to pass I/O data between the XMP and an I/O node. Each message contains either the digital input, digital output, analog input, or analog output data. The XMP supports two standard communication methods to transmit I/O data between the XMP and each of the I/O nodes—**cyclic transmission** and **event transmission**. For most applications, cyclic messaging (the default) will be sufficient, but the transmission type fields within the [MEICanNodeConfig](#) structure allow the user to select an alternative transmission type for each of the I/O messages going to and from a node.

Cyclic Transmission

The Cyclic Transmission type, transfers I/O data messages between the XMP and the nodes using a cyclic protocol. The trigger for each cycle is a synchronization message that is transmitted at a regular rate by the XMP. When a node receives the synchronization message, it latches and transmits the current state of its inputs. Immediately after receiving the synchronization message, the master also transmits command messages to all the nodes with their new output states, which will get applied on the next synchronization message. An idle period is also needed to allow time for any non-cyclic messages to be transmitted.



The advantage of this scheme is that it generates a predicable loading of data on the bus. The latency on transmitted data is predictable, but the latency is not the absolute minimum that can be achieved.

Cyclic Period

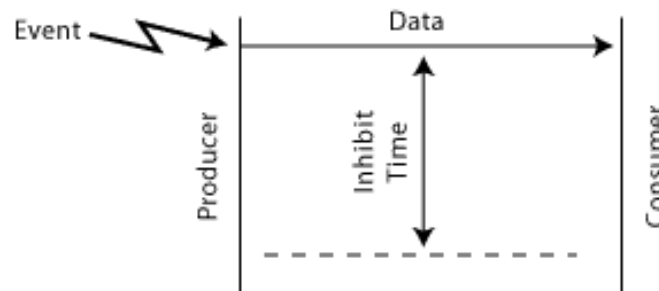
The `cyclicPeriod` field within the [MEICanConfig](#) structure allows the user to specify the period (in milliseconds) that the XMP will use between the successive transmission of synchronization messages. The minimum cyclic period that can be used is dependent upon the chosen bit rate and the number of nodes. Assuming that all the nodes have

inputs and outputs that are analog and digital, the minimum cyclic period that can be used is given in the following table.

CANOpen Cyclic Period				
Bit Rate	< 5 Nodes	< 10 Nodes	< 50 Nodes	< 128 Nodes
1M	3	5	30	60
800k	3	6	30	80
500k	5	10	50	200
250k	10	18	89	300
125k	19	36	200	500
50k	46	90	500	2000
20k	200	300	2000	3000
10k	300	500	3000	6000

Event Transmission

The Event Transmission type, only transmits I/O data messages when an “event” occurs on the source node (either the XMP or the I/O node) to change the I/O data. The event that forces the transmission is either a new state of an input that is detected on an I/O node or a new output state that is commanded on the XMP.



The advantage of this type of messaging is that short reaction times are attainable, but this is accomplished at the expense of variable network traffic, and the possibility of saturating the network. In many cases, the reaction time is not significant in relation to other time delays in the system (ex: the user’s application or delays in task switching).

Inhibit Time

If the source node’s events occur at a very fast rate, the number of messages generated can swamp the network and consequently block out other messages. To prevent an excess of messages, nodes can optionally support inhibit times for their transmit PDOs. This value defines the minimum time between two successive PDO

messages.

The inhibitTime field within the [MEICanConfig](#) structure allows the user to specify the period (in milliseconds) that all nodes on the network will use. A reasonable inhibit time is half a cyclic period.

Capture Object

Introduction

A **Capture** object manages a single position capture logic block. It represents the physical hardware capture logic and data. When configured and armed, the capture logic block can latch a motor's position based on one or more source input triggers.

The Capture object's number, motor input trigger sources, edge, type, feedback source, and capture index are all configurable. There are two capture types: Position and Time based. For the Position type, the position counters are latched in the FPGA and are read directly by the controller. This methodology works well for incremental quadrature encoders. For the Time type, the FPGA latches the clock and the controller reads the clock value and position value for that sample period. The controller interpolates the position value from the previous sample's position, the present sample's position, and the clock data. This methodology works very well for cyclic feedback data that is digitally transmitted from the drive to the FPGA. Many drives have a proprietary serial encoder that decodes the encoder position and sends the position information to the FPGA once per sample. In these cases, time-based capture is more accurate than position-based capture.

For the **Position** type, the motor number for the input sources and the feedback motor number must be the same.

For the **Time** type, the motor number and feedback motor number can be different. This makes it possible to use inputs from one node to capture positions on another node.

When using captures, the controller must have enough enabled captures to process the specified capture number. The controller will process the enabled captures (captureCount) every sample period. Since each capture object is configurable, use the minimum number of captures possible for best controller performance. For example, if you want to use 2 captures for motor 0 and motor 3, set the capture count to 2 and use capture number 0 and 1.

Methods

Create, Delete, Validate Methods

<u>mpiCaptureCreate</u>	Create Capture object
<u>mpiCaptureDelete</u>	Delete Capture object
<u>mpiCaptureValidate</u>	Validate Capture object

Configuration and Information Methods

<u>mpiCaptureConfigGet</u>	Get Capture configuration
<u>mpiCaptureConfigSet</u>	Set Capture configuration
<u>mpiCaptureStatus</u>	Get status of Capture
<u>mpiCaptureConfigReset</u>	

Action Methods

<u>mpiCaptureArm</u>	Arm capture object

Memory Methods

<u>mpiCaptureMemory</u>	Set address to Capture memory
<u>mpiCaptureMemoryGet</u>	Copy Capture memory to application memory
<u>mpiCaptureMemorySet</u>	Copy application memory to Capture memory

Relational Methods

<u>mpiCaptureNumber</u>	Get index of Capture (for Control list)

Data Types

[MPICaptureConfig](#)
[MPICaptureEdge](#)
[MPICaptureMessage](#) / [MEICaptureMessage](#)
[MPICaptureSource](#)
[MPICaptureState](#)
[MPICaptureStatus](#)
[MPICaptureTrigger](#)
[MPICaptureTriggerGlobal](#)
[MPICaptureType](#)

Constants

[MPICaptureNOT_MAPPED](#)

mpiCaptureCreate

Declaration

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

Required Header `stdmpi.h`

Description **CaptureCreate** creates a Capture object. The Capture object is identified by its association with a motor object, the motor's encoder and the encoder's capture number. The maximum number of enabled captures is 16.

CaptureCreate is the equivalent of a C++ constructor.

control	a handle to a Control object
number	An index to the encoder's capture block.

Return Values

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

See Also [mpiCaptureNumber](#)

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 <i>CaptureDelete</i> 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. CaptureValidate should be called immediately after an object is created.

capture	a handle to a capture object

Return Values

MPIMessageOK	if Capture is a handle to a valid object.

See Also [mpiCaptureCreate](#) | [mpiCaptureDelete](#)

mpiCaptureConfigGet

Declaration	long mpiCaptureConfigGet (MPICapture MPICaptureConfig voidcapture, *config, *external)
Required Header	stdmpi.h
Description	CaptureConfigGet gets a Capture object's (<i>capture</i>) configuration and writes it into the structure pointed to by <i>config</i>, and also writes it into the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> is not NULL). The a Capture object's configuration information in <i>external</i> is <i>in addition</i> to the Capture object's configuration information in <i>config</i>, i.e, the Capture object's configuration information in <i>config</i> and in <i>external</i> is not the same information. Note that <i>config</i> or <i>external</i> can be NULL (but not both NULL). If a capture is in an unknown configuration (non-default), use mpiCaptureConfigReset(...) to return the capture to the default configuration before calling mpiCaptureConfigGet(...) and mpiCaptureConfigSet(...). Or if you do not call mpiCaptureConfigReset(...), make sure that all members of the <code>MPICaptureConfig{...}</code> structure are explicitly set before calling mpiCaptureConfigSet(...).

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](#) | [mpiCaptureConfigReset](#)

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).

If a capture is in an unknown configuration (non-default), use `mpiCaptureConfigReset(...)` to return the capture to the default configuration before calling `mpiCaptureConfigGet(...)` and `mpiCaptureConfigSet(...)`. Or if you do not call `mpiCaptureConfigReset(...)`, make sure that all members of the `MPICaptureConfig{...}` structure are explicitly set before calling `mpiCaptureConfigSet(...)`.

XMP Only

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

Return Values	
MPIMessageOK	if <i>CaptureConfigSet</i> successfully sets the Capture object's configuration using data from the structure(s)

See Also

[mpiCaptureConfigGet](#) | [mpiCaptureConfigReset](#)

Required Header

external is reserved for future functionality and should be set to NULL.

XMP Only

external should always be set to NULL.

Return Values

See Also

mpiCaptureConfigReset

Declaration

```
long mpiCaptureConfigReset(MPICapture capture);
```

Required Header

stdmpi.h

Description

[CaptureConfigReset](#) return the capture object to its unmapped state.

A capture object has no assumed resources, and is unmapped under default conditions. When a capture is first created, its `captureMotorNumber` and `feedbackMotorNumber` are unmapped. Once a capture has been configured, the next time that the capture object is created, it will retain the `captureMotorNumber` and `feedbackMotorNumber` that was previously assigned. `mpiCaptureConfigReset(...)` will return the capture object to its unmapped state.

If a capture is in an unknown configuration (non-default), use `mpiCaptureConfigReset(...)` to return the capture to the default configuration before calling `mpiCaptureConfigGet(...)` and `mpiCaptureConfigSet(...)`. Or if you do not call `mpiCaptureConfigReset(...)`, make sure that all members of the `MPICaptureConfig{...}` structure are explicitly set before calling `mpiCaptureConfigSet(...)`.

capture	a handle to a Capture object

See Also

[mpiCaptureConfigGet](#) | [mpiCaptureConfigSet](#) | [MPICaptureConfig](#)

Required Header stdmpi.h

Value of "arm"	Action of <i>mpiCaptureArm</i>
FALSE	Disarms <i>capture</i> and sets the state of <i>capture</i> to MPICaptureStateIDLE
TRUE	Arms <i>capture</i> and sets the state of <i>capture</i> to MPICaptureStateARMED

MPIMessageOK	if the Capture object is successfully armed or disarmed

file:///D:/pdfs/030100/html/Software-MPI/docs/Capture/Method/arm1.htm [7/22/2004 2:26:25 PM]

mpiCaptureMemory

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

Required Header

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 <i>CaptureMemory</i> successfully writes the Capture object's memory address to the contents of <i>memory</i>

See Also [mpiCaptureMemoryGet](#) | [mpiCaptureMemorySet](#)

mpiCaptureMemoryGet

Declaration long **mpiCaptureMemoryGet**([MPICapture](#) capture,
 void *dst,
 void *src,
 long count)

Required Header

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

Return Values

MPIMessageOK	if <i>CaptureMemoryGet</i> 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 <i>CaptureMemorySet</i> successfully copies count bytes of application memory to Capture memory

See Also

[mpiCaptureMemory](#) | [mpiCaptureMemoryGet](#)

mpiCaptureNumber

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

Required Header

Description	Value
CaptureNumber	reads the index of the capture block associated with the capture object and writes it into the contents of a long pointed to by encoder.

capture	a handle to a capture object
*number	pointer to the capture number.

Return Values

MPIMessageOK	if <i>CaptureNumber</i> successfully writes the index of a Capture object to the contents of number

See Also [mpiCaptureCreate](#)

MPICaptureConfig

MPICaptureConfig

```
typedef struct MPICaptureConfig {
    MPICaptureTrigger      source[MPICaptureSourceCOUNT];
                               /* use MPICaptureSource to index */
    MPICaptureEdge        edge;
    MPICaptureTriggerGlobal global;
    MPICaptureType        type;
    long                   captureMotorNumber;
    long                   feedbackMotorNumber; /* the same as
                               captureMotorNumber for POSITION capture */
    MPIMotorEncoder       encoder;
    long                   captureIndex;      /* 0,1,... */
} MPICaptureConfig;
```

Description

source [MPICaptureSourceCOUNT]	An array of capture trigger source inputs. The capture can be configured to trigger from one or more sources. See MPICaptureTrigger and MPICaptureSourceCOUNT .
edge	An enumerated index to the trigger edge type. The capture can be configured to trigger from a variety of logic. See MPICaptureEdge .
global	A structure to configure the global capture, to chain capture block triggering. See MPICaptureTriggerGlobal .
type	Specifies either position-based or time-based capture. Use MPICaptureTypePOSITION for position-based capture and MPICaptureTypeTIME for time-based capture.
captureMotorNumber	The number of the motor whose "source" (MPICaptureTrigger) is used to capture position.
feedbackMotorNumber	The number of the motor whose position is being returned from the capture event. (It must be the same as captureMotorNumber for position capture).
encoder	Specifies the encoder feedback being captured.

captureIndex

A zero-based index that specifies which capture resource on an axis is to be associated with the capture object.

Each axis on a node has a given number of captures associated with it. An axis may have up to 4 capture resources on it. At present, no vendor provides a node with more than one capture resource, therefore, **captureIndex must be set to zero.**

See Also [MPICaptureType](#)

MPICaptureEdge

MPICaptureEdge

```
typedef enum MPICaptureEdge {  
    MPICaptureEdgeNONE,  
    MPICaptureEdgeRISING,  
    MPICaptureEdgeFALLING,  
    MPICaptureEdgeEITHER,  
} MPICaptureEdge;
```

Description

CaptureEdge is an enumeration of input trigger edge logic for a capture.

MPICaptureEdgeRISING	Triggers on a 0 to 1 transition.
MPICaptureEdgeFALLING	Triggers on a 1 to 0 transition.
MPICaptureEdgeEITHER	Triggers on either 0 to 1 or 1 to 0 transitions.

See Also

[MPICaptureTrigger](#)

MPICaptureMessage / MEICaptureMessage

MPICaptureMessage

```
typedef enum {
    MPICaptureMessageMOTOR_INVALID,
    MPICaptureMessageCAPTURE_TYPE_INVALID,
    MPICaptureMessageCAPTURE_INVALID,
    MPICaptureMessageENCODER_INVALID,
} MPICaptureMessage;
```

Description

CaptureMessage is an enumeration of Capture error messages that can be returned by the MPI library.

MEICaptureMessageMOTOR_INVALID

mpiCaptureConfigSet(...) --> config.captureMotorNumber is not valid. It's either greater than maxMotors or == MPICaptureNOT.MAPPED.

MEICaptureMessageCAPTURE_TYPE_INVALID

mpiCaptureConfigSet(...) --> config.Type == MPICaptureNOT.MAPPED.

MPICaptureMessageCAPTURE_INVALID

The capture number is out of range. This message code is returned by mpiCaptureCreate(...) if the capture number is less than zero or greater than or equal to MEIXmpMaxCapturesPerMotor.

MPICaptureMessageENCODER_INVALID

The encoder index is out of range. This message code is returned by mpiCaptureCreate(...) if the encoder index is less than MPIMotorEncoderFIRST or greater than or equal to MPIMotorEncoderLAST.

See Also

[mpiCaptureCreate](#) | [mpiControlConfigSet](#)

MEICaptureMessage

```
typedef enum {
    MEICaptureMessageINVALID_EDGE,
    MEICaptureMessageGLOBAL_CONFIG_ERR,
    MEICaptureMessageGLOBAL_ALREADY_ENABLED,
    MEICaptureMessageCAPTURE_NOT_ENABLED,
    MEICaptureMessageCAPTURE_STATE_INVALID,
    MEICaptureMessageNOT_MAPPED,
    MEICaptureMessageUNSUPPORTED_PRIMARY,
    MEICaptureMessageUNSUPPORTED_SECONDARY,
    MEICaptureMessageSECONDARY_INDEX_INVALID,
} MEICaptureMessage;
```

Description

MEICaptureMessageINVALID_EDGE

The encoder edge trigger type is not valid. This message code is returned by `mpiCaptureConfigSet(...)` if the encoder capture edge type is not a member of the `MPICaptureEdge` enumeration.

MEICaptureMessageGLOBAL_CONFIG_ERR

The global trigger configuration is not valid. This message code is returned by `mpiCaptureConfigSet(...)` if the capture's trigger source is set to global and the capture's global trigger is enabled simultaneously. To correct this problem, either set the capture's trigger source to global or enable the capture's global trigger (not both).

MEICaptureMessage_GLOBAL_ALREADY_ENABLED

The global trigger is already enabled. This message code is returned by `mpiCaptureConfigSet(...)` if a global trigger is already enabled on another capture on the same node. Only one global trigger enable is allowed per node. To prevent this problem, do not enable a second global trigger on a single node.

MEICaptureMessageCAPTURE_NOT_ENABLED

This value is returned by `mpiCatureCreate(...)` when the capture number specified is greater than the number of captures enabled in firmware. See [MPIControlConfig](#).

MEICaptureMessageCAPTURE_STATE_INVALID

This value is returned by `mpiCaptureStatus(...)` when the communication between the controller and the capture logic on the node fails resulting in an invalid capture state. See [MPICaptureState](#).

MEICaptureMessageNOT_MAPPED

The capture object's hardware resource is not available. This message code is returned by [mpiCaptureCreate\(...\)](#) if the node hardware for the specified motor and encoder is not found. During controller and network initialization the nodes and motor count for each node is discovered and mapped to the controller's motor and capture objects. A capture object cannot be created if there is no mapped hardware to support it. To correct this problem, verify that all expected nodes were found. Use [meiSynqNetInfo\(...\)](#) and [meiSqNodeInfo\(...\)](#) to determine the node topology and motor count per node. Check the node hardware power and network connections.

MEICaptureMessageUNSUPPORTED_PRIMARY

The capture hardware does not support the primary encoder. This message code is returned by [mpiCaptureCreate\(...\)](#) if the node hardware's primary encoder does not support the specified capture. To correct this problem, select a different motor, encoder, or capture number.

MEICaptureMessageUNSUPPORTED_SECONDARY

The capture hardware does not support the secondary encoder. This message code is returned by [mpiCaptureCreate\(...\)](#) if the node hardware's secondary encoder does not support the specified capture. To correct this problem, select a different motor, encoder, or capture number.

MEICaptureMessageSECONDARY_INDEX_INVALID

This message is returned from [MPICaptureConfigSet\(...\)](#) when the secondary encoder's index is specified as a trigger source in conjunction with other capture sources.

See Also [mpiCaptureCreate](#)

MPICaptureSource

MPICaptureSource

```
typedef enum MPICaptureSource {
    MPICaptureSourceMOTOR_IO_0,
    MPICaptureSourceMOTOR_IO_1,
    MPICaptureSourceMOTOR_IO_2,
    MPICaptureSourceMOTOR_IO_3,
    MPICaptureSourceMOTOR_IO_4,
    MPICaptureSourceMOTOR_IO_5,
    MPICaptureSourceMOTOR_IO_6,
    MPICaptureSourceMOTOR_IO_7,
    MPICaptureSourceHOME,
    MPICaptureSourceINDEX,
    MPICaptureSourceLIMIT_HW_NEG,
    MPICaptureSourceLIMIT_HW_POS,
    MPICaptureSourceGLOBAL,
    MPICaptureSourceINDEX_SECONDARY,
    MPICaptureSourceCOUNT,
} MPICaptureSource;
```

Description **CaptureSource** is an enumeration of input trigger sources for a capture.

When using one of the MPICaptureSourceMOTOR_IO values in MPICaptureSource, you can determine which MPICaptureSourceMOTOR_IO to use by referencing the appropriate node module. Look in *NodeMotorIoConfig* (replacing *Node* with your node name) in the appropriate node module. Add the appropriate *NodeMotorIoConfig* value to MPICaptureSourceMOTOR_IO_0.

Example: **RMB-10V**

Let's say you are using an MEI RMB-10V and want to find the trigger for XCVR_C.

Look in RMBMotorIoConfig in *mei_rmb.h*. You will find that the appropriate value for XCVR_C is RMBMotorIoConfigXCVR_C. RMBMotorIoConfigXCVR_C is the third value in RMBMotorIoConfig. This means that the value to use in MPICaptureSource is MPICaptureSourceMOTOR_IO_2 (the third MPICaptureSourceMOTOR_IO value).

A better way of making this conversion in your program is to add the MPICaptureSourceMOTOR_IO_0 to the nodeMotorIoConfig value you want to use. In the above example, it would be (MPICaptureSourceMOTOR_IO_0 + RMBMotorIoConfigXCVR_C).

Example: **Trust TA800**

To trigger off of hall A on a Trust TA800 node, you would use (MPICaptureSourceMOTOR_IO_0 + TA800MotorIoConfigHALL_A). Remember that you will need to look in *trust_ta800.h* (the node module) to find TA800MotorIoConfigHALL_A.

MPICaptureSourceMOTOR_IO_0	a capture trigger source is the 0 bit in the motor's configurable I/O.
MPICaptureSourceMOTOR_IO_1	a capture trigger source is the 1 bit in the motor's configurable I/O.
MPICaptureSourceMOTOR_IO_2	a capture trigger source is the 2 bit in the motor's configurable I/O.
MPICaptureSourceMOTOR_IO_3	a capture trigger source is the 3 bit in the motor's configurable I/O.
MPICaptureSourceMOTOR_IO_4	a capture trigger source is the 4 bit in the motor's configurable I/O.
MPICaptureSourceMOTOR_IO_5	a capture trigger source is the 5 bit in the motor's configurable I/O.
MPICaptureSourceMOTOR_IO_6	a capture trigger source is the 6 bit in the motor's configurable I/O.
MPICaptureSourceMOTOR_IO_7	a capture trigger source is the 7 bit in the motor's configurable I/O.
MPICaptureSourceHOME	a capture trigger source is the HOME input in the dedicated I/O input.
MPICaptureSourceINDEX	a capture trigger source is the encoder INDEX input in the dedicated I/O input.
MPICaptureSourceLIMIT_HW_NEG	a capture trigger source is the Hardware Negative Limit input in the dedicated I/O input.
MPICaptureSourceLIMIT_HW_POS	a capture trigger source is the Hardware Positive Limit input in the dedicated IO word. Please see MPIMotorDedicatedIn .
MPICaptureSourceGLOBAL	a capture trigger source is the Global capture signal found on the node. Please see MPICaptureTriggerGlobal .
MPICaptureSourceINDEX_SECONDARY	A a capture trigger source is the index on the secondary encoder. If position based capture is selected with the feedback source being the secondary encoder, this is the only valid capture source.
MPICaptureSourceCOUNT	Total number of possible input sources for a capture.

See Also [MPICaptureTrigger](#) | [MEIMotorIoMask](#)

MPICaptureState

MPICaptureState

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

Description

MPICaptureStateIDLE	Capture is not armed. This is the default state.
MPICaptureStateARMED	Capture is armed, but has not triggered yet.
MPICaptureStateCAPTURED	Capture triggered and position data is valid.
MPICaptureStateCLEAR	Capture is not armed, but has not transitioned to the IDLE state yet. This is an internal transitional state between CAPTURED and IDLE. It occurs when a capture is disarmed.

See Also [MPICaptureStatus](#)

MPICaptureStatus

MPICaptureStatus

```
typedef struct MPICaptureStatus {
    MPICaptureState    state;
    double            latchedValue;
} MPICaptureStatus;
```

Description

state	An enumerated value representing the present state of the capture logic
latchedValue	The captured position value. This value is only valid when the state is CAPTURED.

See Also

[MPICaptureState](#)

MPICaptureTrigger

MPICaptureTrigger

```
typedef struct MPICaptureTrigger {
    long enabled;    /* TRUE/FALSE */
    long invert;     /* TRUE = invert, FALSE = normal */
} MPICaptureTrigger;
```

Description

The **CaptureTrigger** structure specifies the trigger configurations for a capture.

enabled	Enables or disables the trigger. A value of TRUE enables the trigger, FALSE disables the trigger.
invert	Normal or inverted trigger polarity. A value of FALSE indicates normal polarity, TRUE indicates inverted polarity.

See Also

[MPICaptureSource](#)

MPICaptureTriggerGlobal

MPICaptureTriggerGlobal

```
typedef struct MPICaptureTriggerGlobal {
    long    enabled;    /* TRUE/FALSE */
} MPICaptureTriggerGlobal;
```

Description

The **CaptureTriggerGlobal** structure specifies the global input trigger configuration for a capture.

enabled	Enables or disables the global input trigger. A value of TRUE enables the trigger, FALSE disables the trigger.

See Also

[MPICaptureConfig](#)

MPICaptureType

MPICaptureType

```
typedef enum {  
    MPICaptureTypePOSITION,  
    MPICaptureTypeTIME,  
} MPICaptureType;
```

Description

MPICaptureTypePOSITION	An actual position is captured by the Node from its feedback source.
MPICaptureTypeTIME	An internal timer is captured by the node and then a captured position is interpolated by the XMP firmware.

See Also

MPICaptureNOT_MAPPED

Declaration `#define MPICaptureNOT_MAPPED (-1)`

Required Header `stdmpi.h`

Description Capture objects are associated with the controller and are not mapped to any hardware resources under default conditions. MPICaptureNOT_MAPPED will be assigned to:

```
long    captureMotorNumber;  
long    feedbackMotorNumber;
```

when mpiCaptureConfigGet() is called for the first time on a capture object. After a capture object has been used once, the resource mapping will remain in place until it is reassigned.

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.

Information about the different types of commands can be found on [MPICommandType](#) and [MPICommandParams](#).

Methods

Create, Delete, Validate Methods

mpiCommandCreate	Create Command object
mpiCommandDelete	Delete Command object
mpiCommandValidate	Validate Command object

Configuration and Informational Methods

mpiCommandLabel	Get pointer to Command label
mpiCommandParams	Get Command parameters
mpiCommandType	Return Command type

Other Methods

mpiCommandAxisListGet	Get the axisCount and axisList from a Command object.

Data Types

[MPICommandAddress](#)
[MPICommandConstant](#)
[MPICommandExpr](#)
[MPICommandMessage](#)
[MPICommandMotion](#)
[MPICommandOperator](#)
[MPICommandParams](#)

[MPICommandType](#)

See Also

[MPISequence](#)

Required Header

Return Values

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 <i>CommandDelete</i> 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](#)

Required Header stdmpi.h

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

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

file:///D:/pdfs/030100/html/Software-MPI/docs/Command/Method/lbl1.htm [7/22/2004 2:53:15 PM]

Required Header

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

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

file:///D:/pdfs/030100/html/Software-MPI/docs/Command/Method/par1.htm [7/22/2004 2:53:14 PM]

Required Header

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

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

file:///D:/pdfs/030100/html/Software-MPI/docs/Command/Method/ty1.htm [7/22/2004 2:53:13 PM]

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

Required Header

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

MPIMessageOK	if <i>CommandAxisListGet</i> successfully gets the <i>axisCount</i> and <i>axisList</i> 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

Currently not supported and is reserved for future use.

MPICommandMessageTYPE_INVALID

The command type is not valid. This message code is returned by mpiCommandCreate(...) if the command type is not a member of the MPICommandType enumeration.

MPICommandMessagePARAM_INVALID

Currently not supported and is reserved for future use.

See Also [MPICommandType](#)

MPICommandMotion

MPICommandMotion

```
typedef enum {
    MPICommandMotionABORT,
    MPICommandMotionE_STOP,
    MPICommandMotionE_STOP_ABORT,
    MPICommandMotionE_STOP_CMD_EQ_ACT,
    MPICommandMotionMODIFY,
    MPICommandMotionRESET,
    MPICommandMotionRESUME,
    MPICommandMotionSTART,
    MPICommandMotionSTOP,
} MPICommandMotion;
```

Description

CommandMotion is an enumeration of motion specific controller commands that can be used in a program sequence. It specifies a single motion action for the controller to execute. The CommandMotion also defines the command parameters that must be passed to `mpiCommandCreate(...)`. For `MPICommandMotion`, there is a corresponding `motion{...}` structure in the `MPICommandParams{...}` structure.

MPICommandMotionABORT	Commands an Abort action on the motion supervisor associated with the motion object. See mpiMotionAction(...) , <code>MPIActionABORT</code> for details.
MPICommandMotionE_STOP	Commands an E-Stop action on the motion supervisor associated with the motion object. See mpiMotionAction(...) , <code>MPIActionE_STOP</code> for details.
MPICommandMotionE_STOP_ABORT	Commands an E-Stop, then Abort action on the motion supervisor associated with the motion object. See mpiMotionAction(...) , <code>MPIActionE_STOP_ABORT</code> for details.
MPICommandMotionE_STOP_CMD_EQ_ACT	Commands an E-Stop (command position = actual position) action on the motion supervisor associated with the motion object. See mpiMotionAction(...) , <code>MPIActionE_STOP_CMD_EQ_ACT</code> for details.

MPICommandMotionMODIFY	Commands a Motion Modify on the motion supervisor associated with the motion object. Make sure to specify the MPIMotionType and MPIMotionParams in the MPICommandParams{...} structure. See mpiMotionModify(...) for details.
MPICommandMotionRESET	Commands a Reset action on the motion supervisor associated with the motion object. See mpiMotionAction(...) , MPIActionRESET for details.
MPICommandMotionRESUME	Commands a Resume action on the motion supervisor associated with the motion object. See mpiMotionAction(...) , MPIActionRESUME for details.
MPICommandMotionSTART	Commands a Motion Start on the motion supervisor associated with the motion object. Make sure to specify the MPIMotionType and MPIMotionParams in the MPICommandParams{...} structure. See mpiMotionStart(...) for details.
MPICommandMotionSTOP	Commands a Stop action on the motion supervisor associated with the motion object. See mpiMotionAction(...) , MPIActionSTOP for details.

See Also [MPIAction](#) | [MPICommand](#) | [MPICommandParams](#)

MPICommandOperator

MPICommandOperator

```
typedef enum {
    /* 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 division. 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;
        MPIControl          control; /* 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

[CommandParams](#) holds the parameters used by an MPICommand. Each element in the MPICommandParams union corresponds to different types of commands (specified by the MPICommandType enumeration).

Element	Description	Supported by
assign	Assign a value to a particular controller address: *dst = value assign.control is currently not supported and is reserved for future use.	MPICommandTypeASSIGN MPICommandTypeASSIGN_FLOAT

branch	Branch to a particular command (similar to a <i>goto</i> statement) if a particular comparison evaluates to TRUE: branch to label on expr If <i>label</i> = 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.	MPICommandTypeBRANCH MPICommandTypeBRANCH_REF MPICommandTypeBRANCH_FLOAT MPICommandTypeBRANCH_FLOAT_REF
branchEvent	Branch to a particular command (similar to a <i>goto</i> statement) if a particular event occurs or has occurred: branch to label on MPIEventMask(handle) oper mask If <i>label</i> = NULL, then no more commands will be executed if a particular event occurs or has occurred.	MPICommandTypeBRANCH_EVENT
branchIO	Branch to a particular command (similar to a <i>goto</i> statement) if a particular i/o state matches a specified condition: branch to <i>label</i> on <i>Io.input oper mask</i> If <i>label</i> = NULL, then no more commands will be executed if a particular i/o state matches a specified condition.	MPICommandTypeBRANCH_IO
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.	MPICommandTypeCOMPUTE MPICommandTypeCOMPUTE_REF MPICommandTypeCOMPUTE_FLOAT MPICommandTypeCOMPUTE_FLOAT_REF
computeIO	Performs a computation on a set of i/o bits: <i>Io.output</i> = <i>Io.output oper mask</i>	MPICommandType_IO
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.	MPICommandTypeCOPY
delay	Delays execution of the next command <i>delay</i> seconds.	MPICommandTypeDELAY
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.	MPICommandTypeEVENT
motion	Commands a motion action (See MPICommandMotion): mpiMotionStart (motion, type, params); or mpiMotionAction(motion, MPIAction[ABORT E_STOP E_STOP_ABORT RESET RESUME STOP]);	MPICommandTypeMOTION
wait	Delays execution of the next command until a particular comparison evaluates to TRUE: wait until <i>expr</i> wait.control is currently not supported and is reserved for future use.	MPICommandTypeWAIT MPICommandTypeWAIT_REF MPICommandTypeWAIT_FLOAT MPICommandTypeWAIT_FLOAT_REF
waitEvent	Delays execution of the next command until a particular event occurs: wait until MPIEventMask(handle) oper mask	MPICommandTypeWAIT_EVENT
waitIO	Delays execution of the next command until a particular i/o state matches a specified condition: wait until <i>Io.input oper mask</i>	MPICommandTypeWAIT_IO

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

CommandType is an enumeration of controller commands that can be used in a program sequence. It specifies a single instruction for the controller to execute. The CommandType also defines the command parameters that must be passed to `mpiCommandCreate(...)`. For each MPICommandType there is a corresponding structure in the `MPICommandParams{...}` union. For example, when the MPICommandTypeASSIGN is specified, the `assign{...}` structure in `MPICommandParams{...}` must be filled in to specify the address and value.

Commands must be created with `mpiCommandCreate(...)` and then added to a sequence using `mpiSequenceCommandAppend(...)`, `mpiSequenceCommandInsert(...)`, or `mpiSequenceCommandListSet(...)`. Then the command sequence can be loaded into the controller with `mpiSequenceLoad(...)` and started with `mpiSequenceStart(...)`.

Enumeration	Description	Associated MPICommandParams structure
MPICommandTypeASSIGN	Writes a constant value (long or float) into the controller's memory at the specified address.	assign
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.	branch
MPICommandTypeBRANCH_REF	Branch to a particular command if the comparison evaluates to TRUE. 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 occurred.	branchEvent
MPICommandTypeBRANCH_IO	Branch to a particular command (similar to a goto statement) if a particular I/O state matches a specified condition.	branchIO

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.	compute
MPICommandTypeCOMPUTE_REF		
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.	computeIO
MPICommandTypeCOPY	Copies controller memory from one place to another.	copy
MPICommandTypeDELAY	Delays execution of the next command.	delay
MPICommandTypeEVENT	Generate an event.	event
MPICommandTypeMOTION	Commands a motion action. See MPICommandMotion .	motion
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.	wait
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.	waitEvent

MPICommandTypeWAIT_IO

Delays execution of the next command until a particular I/O state matches a specified condition.

waitIO

See Also

[MPICommand](#) | [MPICommandMotion](#) | [MPICommandParams](#) | [mpiCommandCreate](#) | [mpiCommandType](#) | [mpiSequenceCommandAppend](#) | [mpiSequenceCommandInsert](#) | [mpiSequenceCommandListSet](#) | [mpiSequenceLoad](#) | [mpiSequenceStart](#) |

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

<u>mpiCompareCreate</u>	Create Compare object
<u>mpiCompareDelete</u>	Delete Compare object
<u>mpiCompareValidate</u>	Validate Compare object

Configuration and Information Methods

<u>mpiCompareConfigGet</u>	Get Compare configuration
<u>mpiCompareConfigSet</u>	Set Compare configuration
<u>mpiCompareFlashConfigGet</u>	Get flash configuration for Compare
<u>mpiCompareFlashConfigSet</u>	Set flash configuration for Compare
<u>mpiCompareStatus</u>	Get Compare's status

Memory Methods

<u>mpiCompareMemory</u>	Set Compare memory address
<u>mpiCompareMemoryGet</u>	Copy bytes of Compare memory to application memory
<u>mpiCompareMemorySet</u>	Copy bytes of application memory to Compare memory

Action Methods

<u>mpiCompareArm</u>	Arm Compare object
<u>mpiCompareLoad</u>	

Relational Methods

<u>mpiCompareControl</u>	Return handle of Control that is associated with Compare

[mpiCompareNumber](#)

Get index of Compare (for Control list)

Data Types

[MPICompareConfig](#) / [MEICompareConfig](#)

[MPICompareMessage](#)

[MPICompareParams](#)

[MPIComparePosition](#)

[MPICompareState](#)

[MPICompareStatus](#)

Constants

[MPIComparePositionCountMAX](#)

Required Header

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.

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

file:///D:/pdfs/030100/html/Software-MPI/docs/Compare/Method/create1.htm [7/22/2004 3:00:35 PM]

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 <i>CompareDelete</i> 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
MPICompareConfig
void
compare,
*config,
*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 <i>CompareConfigGet</i> successfully writes the Compare object's configuration to the structure(s)

See Also

[mpiCompareConfigSet](#) | [MEICompareConfig](#)


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

Description	CompareConfigSet sets a Compare object's (<i>compare</i>) configuration using data from the structure pointed to by <i>config</i> , and also using data from the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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).

external either points to a structure of type **MEICompareConfig** or is NULL.

Return Values

MPIMessageOK	if <i>CompareConfigSet</i> successfully sets the Compare object's configuration using data from the structure(s)

See Also [mpiCompareConfigGet](#) | [MEICompareConfig](#)

mpiCompareFlashConfigGet

Declaration

long mpiCompareFlashConfigGet(MPICompare
void
MPICompareConfig
void
compare,
*flash,
*config,
*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 <i>CompareFlashConfigGet</i> successfully writes the Compare object’s flash configuration to the structure(s) <i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

[MEIFlash](#) | [mpiCompareFlashConfigSet](#) | [MEICompareConfig](#)

Required Header

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).

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

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

file:///D:/pdfs/030100/html/Software-MPI/docs/Compare/Method/flacfset1.htm [7/22/2004 3:00:37 PM]

Declaration

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

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

compare	a handle to a Compare object
*status	a pointer to Compare's status structure
*external	a pointer to an implementation-specific structure

external should always be set to NULL.

Return Values

MPIMessageOK	if <i>CompareStatus</i> successfully writes the status of a Compare object to the structure(s)
MPIMessageARG_INVALID	if the <i>status</i> pointer is NULL.

See Also

mpiCompareMemory

```
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 <i>CompareMemory</i> successfully writes the Compare object's memory address to the contents of memory

See Also [mpiCompareMemoryGet](#) | [mpiCompareMemorySet](#)

mpiCompareMemoryGet

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

Required Header

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

Return Values

MPIMessageOK	if <i>CompareMemoryGet</i> 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

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 <i>CompareMemoryGet</i> successfully copies data from application memory to Compare memory

See Also [mpiCompareMemoryGet](#) | [mpiCompareMemory](#)

mpiCompareArm

```
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

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

Required Header

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

Return Values

MPIMessageOK	if <i>CompareLoad</i> successfully loads the Compare object's parameters using the data from the structure.

See Also

mpiCompareControl

Declaration [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

Required Header

Return Values

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,

    MPICompareStateIDLE,
    MPICompareStateARMED,
    MPICompareStateCOMPARED,
} 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

Compensator Objects

Introduction

A **Compensator** object manages a single compensation table. Its primary function is to provide an interface to configure both the compensating axes and the compensated axis. It also provides an interface for loading the on-controller compensation tables. The Compensator object is a host-based object that has a corresponding compensator object embedded on the controller. The embedded compensator handles the real-time issues associated with axis position compensation.

Before creating the MPI Compensator object, the corresponding embedded compensator object on the controller must be enabled. Also, before configuring the MPI Compensator object, the controller's compensation table must be allocated with a sufficient size to hold all required compensation values (or points). Both of these items can be configured using `mpiControlConfigGet/Set(...)` methods.

NOTE: Configuring the compensator table size using `mpiControlConfigSet(...)` will reallocate the controller's dynamic memory. Reallocating dynamic memory on the controller affects multiple objects and should only be done at the very beginning of your application.

For more information on determining compensation table size please see [Determining Required Compensator Table Size](#).

Methods

Create, Delete, Validate Methods

<code>mpiCompensatorCreate</code>	Create Compensator object
<code>mpiCompensatorDelete</code>	Delete Compensator object
<code>mpiCompensatorValidate</code>	Validate Compensator object

Configuration and Information Methods

<code>mpiCompensatorConfigGet</code>	Get Compensator configuration
<code>mpiCompensatorConfigSet</code>	Set Compensator configuration
<code>meiCompensatorInfo</code>	Get Compensator information
<code>meiCompensatorTableGet</code>	Get Compensator table
<code>meiCompensatorTableSet</code>	Set Compensator table

Memory Methods

[meiCompensatorMemory](#)

Set address to be used to access Compensator memory

[meiCompensatorMemoryGet](#)

Get bytes of Compensator memory and place it into application memory

[meiCompensatorMemorySet](#)

Put (set) bytes of application memory into Compensator memory

Relational Methods

[meiCompensatorControl](#)

Return handle of Control object associated with Compensator

[meiCompensatorNumber](#)

Get number of Compensator

Data Types

[MPICompensatorConfig](#)

[MPICompensatorDimension](#)

[MPICompensatorInfo](#)

[MPICompensatorInputAxis](#)

[MPICompensatorMessage](#)

[MPICompensatorRange](#)

Constants

[MPICompensatorDimensionsMAX](#)

Required Header

Valid compensator numbers are zero (0) to MPIControlMAX_COMPENSATORS.

Before creating a Compensator object, the controller compensation objects must be enabled using `MPIControlConfig.compensatorCount`, or the host object will be invalid.

handle	handle to a Compensator object

See Also [mpiCompensatorDelete](#) | [mpiCompensatorValidate](#) | [MPIControlConfig](#)

mpiCompensatorDelete

Declaration long **mpiCompensatorDelete**([MPICompensator](#) **compensator**) ;

Required Header stdmpi.h

Description **CompensatorDelete** deletes a host Compensator object (*compensator*) and invalidates its handle.

CompensatorDelete is the equivalent of a C++ destructor.

Return Values

MPIMessageOK	if <i>CompensatorDelete</i> successfully deletes a Compensator object and invalidates its handle

See Also [mpiCompensatorCreate](#) | [mpiCompensatorValidate](#)

mpiCompensatorValidate

Declaration long [mpiCompensatorValidate](#)([MPICompensator](#) compensator);

Required Header stdmpi.h

Description [CompensatorValidate](#) CompensatorValidate validates the Compensator object (*compensator*) and its handle. Always call mpiCompensatorValidate after creating a new Compensator object.

Return Values	
MPIMessageOK	if Compensator is a handle to a valid object.
MPICompensatorMessageCOMPENSATOR_INVALID	If the <i>number</i> used in mpiCompensatorCreate() was not a valid number. Valid numbers range from 0 to MPIControlMAX_COMPENSATORS
MPICompensatorMessageNOT_ENABLED	If the corresponding controller compensation object number is valid but is disabled on the controller.

See Also [mpiCompensatorCreate](#) | [mpiCompensatorDelete](#)

mpiCompensatorConfigGet

Declaration

```
long  mpiCompensatorConfigGet(MPICompensator      compensator,  
                               MPICompensatorConfig *config,  
                               void                *external);
```

Required Header stdmpi.h

Description **CompensatorConfigGet** gets the configuration of a Compensator object (*compensator*) 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 intended for future use and is not currently used. Set this value to NULL.

Return Values	
MPIMessageOK	if <i>CompensatorConfigGet</i> successfully gets the configuration of a Compensator object and writes it into the structure(s).
MPIMessagePARAM_INVALID	If <i>*config</i> parameter is NULL, or if <i>*external</i> parameter is NOT NULL.

See Also [mpiCompensatorConfigSet](#) | [MEICompensatorConfig](#)

mpiCompensatorConfigSet

Declaration

```
long mpiCompensatorConfigSet(MPICompensator compensator,
                             MPICompensatorConfig *config,
                             void *external);
```

Required Header stdmpi.h

Description **CompensatorConfigSet** sets (writes) the configuration of a Compensator object (*compensator*) 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: *config* or *external* can be NULL (but both cannot be NULL).

XMP Only

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

Return Values

MPIMessageOK	if <i>CompensatorConfigSet</i> successfully writes the configuration of a Compensator object using data from the structure(s)
MPIMessagePARAM_INVALID	If <i>*config</i> parameter is NULL, or if <i>*external</i> parameter is NOT NULL
MPIMessageARG_INVALID	The axisOutNumber or any of the inputAxis[n].axisNumbers correspond to invalid Axis objects on the controller. See MPICompensatorConfig .
MPICompensatorMessageDIMENSION_NOT_SUPPORTED	dimensionCount is out of range. See MPICompensatorConfig .

MPICompensatorMessage AXIS_NOT_ENABLED	The axisOutNumber or any of the inputAxis[n].axisNumbers correspond to disabled Axis objects on the controller. See MPICompensatorConfig .
MPICompensatorMessage POSITION_DELTA_INVALID	The <i>positionDelta</i> argument is either out of range or not a multiple of the range. See MPICompensatorConfig .
MPICompensatorMessage TABLE_SIZE_ERROR	The host compenstation table will not fit within the controller's configured compensation table. See Determining Required Compensator Table Size .

See Also [mpiCompensatorConfigGet](#) | [MEICompensatorConfig](#)

mpiCompensatorInfo

Declaration

```
long  mpiCompensatorInfo(MPICompensator      compensator,  
                        MPICompensatorInfo *info);
```

Required Header

stdmpi.h

Description

CompensatorInfo reads the static information about the compensator object, and writes it into the structure pointed to by *info*.

compensator	a handle to the Compensator object.
*info	a pointer to a compensator information structure.

Return Values

MPIMessageOK	if <i>CompensatorInfo</i> successfully writes the Compensator object information into the <i>info</i> structure.
MPICompensatorMessageNOT_CONFIGURED	MPI Compensator object must be configured before calling mpiCompensatorTableGet/Set or mpiCompensatorInfo.

See Also

[MPICompensatorInfo](#) | [MPIControlConfig](#) | [mpiCompensatorTableGet](#) | [mpiCompensatorTableSet](#) | [mpiCompensatorInfo](#)

mpiCompensatorTableGet

Declaration

```
long  mpiCompensatorTableGet( MPICompensator
                             long  compensator,
                             *table);
```

Required Header stdmpi.h

Description **CompensatorTableGet** reads the NxM Compensator table stored on the controller whose dimensions are defined by the values in the MPICompensatorConfig structure. These values are written into the location specified by **table*.

NOTE: The array pointed to **table* must have enough memory allocated to hold the entire size of the configured compensation table.

Return Values	
MPIMessageOK	
MPICompensatorMessageNOT_CONFIGURED	MPI Compensator object must be configured before calling mpiCompensatorTableGet/Set or mpiCompensatorInfo.
MPIMessagePARAM_INVALID	If <i>*table</i> is NULL.

See Also [mpiCompensatorTableSet](#) | [MPICompensatorConfig](#)

mpiCompensatorTableSet

Declaration

```
long  mpiCompensatorTableSet(MPICompensator
                             long            compensator,
                             *table);
```

Required Header stdmpi.h

Description [CompensatorTableSet](#) writes the values stored in the location specified by **table* to the Compensator table stored on the controller.

NOTE: The array pointed to **table* must have a size large enough to fill the configured compensation table size (as defined by the [MPICompensatorConfig](#) structure) or memory access violations may occur.

Return Values	
MPIMessageOK	
MPICompensatorMessageNOT_CONFIGURED	MPI Compensator object must be configured before calling mpiCompensatorTableGet/Set or mpiCompensatorInfo.
MPIMessagePARAM_INVALID	If <i>*table</i> is NULL.

See Also [mpiCompensatorTableGet](#) | [MPICompensatorConfig](#)

mpiCompensatorMemory

Declaration

```
long  mpiCompensatorMemory(MPICompensator compensator,  
                             void **memory);
```

Required Header stdmpi.h

Description [CompensatorMemory](#) sets (writes) an address (used to access a Compensator object's memory) to the contents of *memory*.

Return Values	
MPIMessageOK	if <i>CompensatorMemory</i> successfully writes the Compensator's address to the contents of <i>memory</i> .

See Also [mpiCompensatorMemoryGet](#) | [mpiCompensatorMemorySet](#)

mpiCompensatorMemoryGet

Declaration

```
long  mpiCompensatorMemoryGet(MPICompensator    compensator,  
                                void              *dst,  
                                void              *src,  
                                long               count);
```

Required Header stdmpi.h

Description **CompensatorMemoryGet** copies *count* bytes of a Compensator’s (*compensator*) memory (starting at address *src*) to application memory (starting at address *dst*).

Return Values

MPIMessageOK	if <i>CompensatorMemoryGet</i> successfully copies data from Compensator memory to application memory.

See Also [mpiCompensatorMemorySet](#) | [mpiCompensatorMemory](#)

mpiCompensatorMemorySet

Declaration

```
long  mpiCompensatorMemorySet(MPICompensator    compensator,  
                                void              *dst,  
                                void              *src,  
                                long              count);
```

Required Header stdmpi.h

Description [CompensatorMemorySet](#) copies *count* bytes of application memory (starting at address *src*) to a Compensator’s (*compensator*) memory (starting at address *dst*).

Return Values

MPIMessageOK	if <i>CompensatorMemorySet</i> successfully copies data from application memory to Compensator memory.

See Also [mpiCompensatorMemoryGet](#) | [mpiCompensatorMemory](#)

mpiCompensatorControl

Declaration

```
MPIControl mpiCompensatorControl(MPICompensator compensator);
```

Required Header stdmpi.h

Description **CompensatorControl** returns a handle to the Control object with which the compensator is associated.

compensator	a handle to the Compensator object

Return Values

MPIControl	handle to a Control object
MPIHandleVOID	if <i>compensator</i> is invalid

See Also [mpiCompensatorCreate](#) | [mpiControlCreate](#)

mpiCompensatorNumber

Declaration

```
long  mpiCompensatorNumber (MPICompensator  
                             long      compensator,  
                             *number );
```

Required Header stdmpi.h

Description **CompensatorNumber** writes the index of a compensation object (object on the motion controller that the Compensator object is associated with) to the contents of *number*.

Return Values

MPIMessageOK	if <i>CompensatorNumber</i> successfully writes the index of a compensation object to the contents of <i>number</i> .

See Also [mpiCompensatorCreate](#)

MPICompensatorConfig

MPICompensatorConfig

```
typedef struct MPICompensatorConfig {
    long                dimensionCount,
    MPICompensatorInputAxis inputAxis[MPICompensatorDimensionMAX],
    long                outputAxisNumber,
} MPICompensatorConfig;
```

Description

dimensionCount	The input dimension count of the compensation table. Valid values are from zero (0) to MPICompensatorDimensionsMAX. A value of zero (0) effectively disables the compensation object
inputAxis	The substructure used to configure each input dimension of the Compensator object.
outputAxisNumber	This specifies the axis number of the Axis to be compensated by the Compensator object. This number must correspond to a valid (existing) and enabled Axis on the controller.

See Also [MPIControlConfig](#) | [mpiCompensatorConfigGet](#) | [mpiCompensatorConfigSet](#) | [MPICompensatorDimension](#) | [MPICompensatorDimensionsMAX](#)

MPICompensatorDimension

MPICompensatorDimension

```
typedef struct MPICompensatorDimension {  
    MPICompensatorDimensionX,  
    MPICompensatorDimensionY,  
} MPICompensatorDimension;
```

Description **CompensatorDimension** an enumeration of valid Compensator dimensions.

MPICompensatorDimensionX	First Compensating Dimension
MPICompensatorDimensionY	Second Compensating Dimension

See Also [MPICompensatorConfig](#) | [MPICompensatorInfo](#)

MPICompensatorInfo

MPICompensatorInfo

```
typedef struct MPICompensatorInfo {  
    long    tableDimensions[MPICompensatorDimensionMAX],  
    long    tableSizeBytes,  
} MPICompensatorInfo;
```

Description

tableDimensions	The dimensions along each axis of the table. This value is affected by the values set in the MPICompensatorConfig structure.
tableSizeBytes	The size (in Byte) required to store the entire compensation table in a host resident structure or array. This value is affected by the values set in the MPICompensatorConfig structure. This is NOT the amount of memory allocated on the controller by setting the MPIControlConfig.compensatorPointCount value.

See Also

[MPICompensatorConfig](#) | [MPIControlConfig](#) | [MPICompensatorDimension](#) | [MPICompensatorDimensionMAX](#)

MPICompensatorInputAxis

MPICompensatorInputAxis

```
typedef struct MPICompensatorInputAxis {
    long                axisNumber,
    MPICompensatorRange range,
    long                positionDelta,
} MPICompensatorInputAxis;
```

Description

axisNumber	This specifies the axis number of the compensating Axis object. The position from this Axis will be used to index a single dimension of the compensation table. This number must correspond to a valid (existing) and enabled Axis on the controller.
range	Used to configure the feedback positions along the compensation axis where compensation will start and end.
positionDelete	Spacing between compensation positions on the compensating axis. NOTE: This value must be an exact multiple of the range (i.e. the difference between range.positionMax – range.positionMin)

See Also [MPICompensatorConfig](#) | [mpiCompensatorConfigGet](#) | [mpiCompensatorConfigSet](#)

MPICompensatorMessage

MPICompensatorMessage

```
typedef enum {
    MPICompensatorMessageCOMPENSATOR_INVALID,
    MPICompensatorMessageNOT_CONFIGURED,
    MPICompensatorMessageNOT_ENABLED,
    MPICompensatorMessageAXIS_NOT_ENABLED,
    MPICompensatorMessageTABLE_SIZE_ERROR,
    MPICompensatorMessagePOSITION_DELTA_INVALID,
    MPICompensatorMessageDIMENSION_NOT_SUPPORTED,
} MPICompensatorMessage;
```

Description

MPICompensatorMessageCOMPENSATOR_INVALID

If the number used to in mpiCompensatorCreate(...) was not a valid number. Valid numbers range from 0 to MPIControlMAX_COMPENSATORS.

MPICompensatorMessageNOT_CONFIGURED

MPI Compensator object must be configured before calling mpiCompensatorTableGet/Set or mpiCompensatorInfo.

MPICompensatorMessageNOT_ENABLED

If the corresponding controller compensation object number is valid, but is disabled on the controller.

MPICompensatorMessageAXIS_NOT_ENABLED

The axisOutNumber or any of the inputAxis[n].axisNumbers correspond to disabled Axis objects on the controller. See [MPICompensatorConfig{}](#).

MPICompensatorMessageTABLE_SIZE_ERROR

The host compensation table will not fit within the controller's configured compensation table. See [Determining Required Compensation Table Size](#).

MPICompensatorMessagePOSITION_DELTA_INVALID

The positionDelta argument is either out of range or is not a multiple of the range. See [MPICompensatorConfig{}](#).

MPICompensatorMessageDIMENSION_NOT_SUPPORTED

dimensionCount is out of range. See [MPICompensatorConfig{}](#) for more information.

MPICompensatorRange

MPICompensatorRange

```
typedef struct MPICompensatorRange {  
    long    positionMin,  
    long    positionMax,  
} MPICompensatorRange;
```

Description

positionMin	The minimum feedback position (counts) along the compensation axis where compensation will occur.
positionMax	The maximum feedback position (counts) along the compensation axis where compensation will occur.

See Also [MPICompensatorConfig](#) | [mpiCompensatorConfigGet](#) | [mpiCompensatorConfigSet](#)

MPIControlDimensionsMAX

Declaration

```
#define MPIControlDimensionsMAX (MPICompensatorDimensionLAST)
```

Required Header stdmpi.h

Description **MPIControlDimensionsMAX** defines the maximum number of dimensions supported by the Compensator object's compensation tables.

See Also [MPICompensatorDimension](#) | [MPICompensatorInfo](#) | [MPICompensatorConfig](#)

Determining Required Compensator Table Size

The compensator table size is dependent on the number of dimensions (1D or 2D), the position range, and the resolution (or granularity) of the compensation points. The compensator uses linear interpolation to calculate the compensation values between each distinct compensation point.

For each compensation axis there are three position values: Min, Max, and Delta. The compensating range for an axis is specified by the Min and Max positions along the axis. The range (Max – Min) divided by Delta, determines the number of required points for the compensator. You can calculate the number of required compensator points by using the following equations:

1D Compensation: $\text{Points} = (\text{positionMax} - \text{positionMin}) / \text{positionDelta}$

2D Compensation: $\text{Points} = \text{PointsX} * \text{PointsY}$

NOTE: Delta must be an exact multiple of the difference between Min and Max.

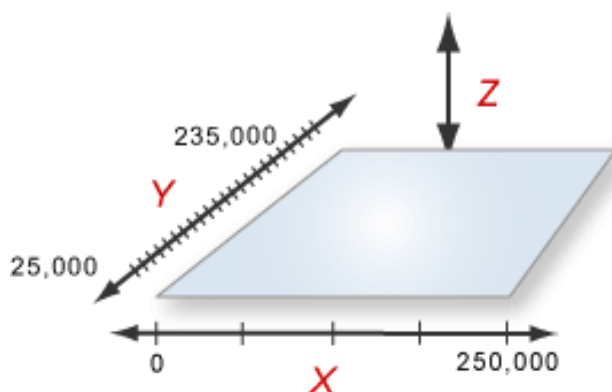
Example: (taken from comp.c sample application)

To compensate a Z (vertical) axis for X-Y surface irregularities, first define the X-Y area to be compensated (Xmin to Xmax, Ymin to Ymax). Then define the spacing of the measuring points (delta) for the X and Y axes to determine the compensation table size.

For the X-Y table diagram below we have:

Xmin = 0, Xmax = 250000, Xdelta = 50000

Ymin = 25000, Ymax = 235000, Ydelta = 10000



For this table our X & Y dimensions are:

$\text{X\_DIM} = (250000 - 0) / 50000 = 5$

$$Y\_DIM = (235000-25000)/10000 = 21$$

which requires a table point count of:

$$Points = X\_DIM * Y\_DIM = 105$$

With this information we can now configure the size of our compensation table on the controller using `MPIControlConfig.compensatorPointCount = 105;`

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

<u>mpiControlCreate</u>	Create Control object
<u>mpiControlDelete</u>	Delete Control object
<u>mpiControlValidate</u>	Validate Control object

Configuration and Information Methods

<u>mpiControlAddress</u>	Get original address of Control object (when it was created)
<u>mpiControlConfigGet</u>	Get Control config
<u>mpiControlConfigSet</u>	Set Control config
<u>meiControlFPGADefaultGet</u>	
<u>meiControlFPGAFileOverride</u>	
<u>meiControlExtMemAvail</u>	
<u>mpiControlFlashConfigGet</u>	Get Control flash config
<u>mpiControlFlashConfigSet</u>	Set Control flash config
<u>meiControlGateGet</u>	Get the closed state (TRUE or FALSE)
<u>meiControlGateSet</u>	Set the closed state (TRUE or FALSE)
<u>meiControlInfo</u>	
<u>mpiControlIoGet</u>	

[mpiControlIoSet](#)[meiControlIoBitGet](#)[meiControlIoBitSet](#)[meiControlSampleCounter](#)[meiControlSamplestoSeconds](#)[meiControlSampleWait](#)[meiControlSecondstoSamples](#)[mpiControlType](#)

Converts samples to seconds

Converts seconds to samples

Get type of Control object (used to create Command object)

Memory Methods

[mpiControlMemory](#)

Get address of Control memory

[mpiControlMemoryAlloc](#)

Allocate bytes of firmware memory

[mpiControlMemoryCount](#)

Get number of bytes available in firmware

[mpiControlMemoryFree](#)

Free bytes of firmware memory

[mpiControlMemoryGet](#)

Copy count bytes of Control memory to application memory

[mpiControlMemorySet](#)

Copy count bytes of application memory to Control memory

Relational Methods

[meiControlPlatform](#)

Action Methods

[mpiControlCycleWait](#)

Wait for Control to execute count cycles

[mpiControlInit](#)

Initialize Control object

[mpiControlInterruptEnable](#)

Enable interrupts to Control object

[mpiControlInterruptWait](#)

Wait for controller interrupt

[mpiControlInterruptWake](#)

Wake all threads waiting for controller interrupt

[mpiControlReset](#)

Reset controller hardware

[meiControlSampleWait](#)

Specify how many samples the host waits for, while the XMP executes

[meiControlVersionMismatchOverride](#)

Data Types

[MPIControlAddress](#)[MPIControlConfig](#) / [MEIControlConfig](#)[MEIControlFPGA](#)[MEIControlInfo](#)[MEIControlInfoDriver](#)[MEIControlInfoFirmware](#)

[MEIControlInfoHardware](#)

[MEIControlInfoMpi](#)

[MEIControlInfoPld](#)

[MEIControlInfoRincon](#)

[MEIControlInput](#)

[MPIControlIo](#)

[MEIControlIoBit](#)

[MPIControlIoWords](#)

[MPIControlMessage](#) / [MEIControlMessage](#)

[MPIControlMemoryType](#)

[MEIControlOutput](#)

[MEIControlTrace](#)

[MPIControlType](#)

Constants

[MPIControlMAX_AXES](#)

[MPIControlMAX_COMPENSATORS](#)

[MPIControlMAX_RECORDERS](#)

[MPIControlMIN_AXIS_FRAME_COUNT](#)

[MEIControlSTRING_MAX](#)

mpiControlCreate

Declaration `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
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
DEFAULT	Null address	0 address.number	default address parameter default address parameter
DEVICE	Null address	0 address.number	default device driver address.type.device (if address.type.device is Null, then default device driver)
CLIENT	address	specified by server	address.type.client (NOTE: address.number should be set to zero)

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 <i>ControlDelete</i> 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

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

Return Values

MPIMessageOK	if <i>ControlAddress</i> successfully writes the address (used when <i>control</i> was created) to the contents of <i>address</i>

See Also

Required Header

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).

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

Sample Code

file:///D:/pdfs/030100/html/Software-MPI/docs/Control/Method/cfget1.htm (1 of 2) [7/22/2004 3:14:14 PM]

Return Values

MPIMessageOK	if <i>ControlConfigGet</i> successfully gets the <i>control</i> configuration and writes it in the structure(s)

See Also [mpiControlConfigSet](#) | [MEIControlConfig](#) |
[Special Note](#) on Dynamic Allocation of External Memory Buffers.

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

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

XMP Only

external either points to a structure of type **MEIControlConfig** or is NULL.

Sample Code

file:///D:/pdfs/030100/html/Software-MPI/docs/Control/Method/cfset1.htm (1 of 2) [7/22/2004 3:14:15 PM]

Return Values

MPIMessageOK	if <i>ControlConfigSet</i> 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.

Required Header `stdmei.h`

Return Values

See Also [MPIControlConfig](#)

Declaration long [mpiControlFlashConfigGet](#)([MPIControl](#) control,
 void *flash,
 [MPIControlConfig](#) *config,
 void *external)

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

Sample Code

```

/*
    Write a value to element index of the user buffer.
    Make sure to save topology to flash before doing this.
*/
void write2UserBufferFlash(MPIControl control, long value, long index)
{
    MPIControlConfig config;
    MEIControlConfig external;
    long returnValue;

    if((index < MEIXmpUserDataSize) && (index >= 0))
    {
        /* Make sure to save topology to flash before doing this */
        returnValue = mpiControlFlashConfigGet(control,
            MPIHandleVOID,
            &config,
            &external);
        msgCHECK(returnValue);

        external.UserBuffer.Data[index] = value;

        returnValue = mpiControlFlashConfigSet(control,
            MPIHandleVOID,
            &config,

```

```
        &external);  
    msgCHECK(returnValue);  
}  
}
```

Return Values

MPIMessageOK	if <i>ControlFlashConfigGet</i> 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.

Sample Code

```
/*
 * Write a value to element index of the user buffer.
 * Make sure to save topology to flash before doing this.
 */
void write2UserBufferFlash(MPIControl control, long value, long index)
{
    MPIControlConfig config;
    MEIControlConfig external;
    long returnValue;

    if((index < MEIXmpUserDataSize) && (index >= 0))
    {
        /* Make sure to save topology to flash before doing this */
        returnValue = mpiControlFlashConfigGet(control,
            MPIHandleVOID,
            &config,
            &external);
        msgCHECK(returnValue);

        external.UserBuffer.Data[index] = value;

        returnValue = mpiControlFlashConfigSet(control,
            MPIHandleVOID,
```

```
        &config,  
        &external);  
    msgCHECK(returnValue);  
}  
}
```

Return Values

MPIMessageOK

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

See Also

[MEIFlash](#) | [mpiControlFlashConfigGet](#) | [MEIControlConfig](#)

meiControlFPGADefaultGet

Declaration

```
long meiFPGADefaultGet(MPIControl
                        MEIPlatformSocketInfo
                        MEIControlFPGA
                        control,
                        *socketInfo,
                        *fpga)
```

Required Header stdmei.h

Description **FPGADefaultGet** creates a default FPGA filename based on the *socketInfo*.

control	a handle to the Control object
*socketInfo	tells the function which type of FPGA is physically on the board.
*fpga	a pointer to a MEIControlFPGA object that contains a string that is the filename. To get the proper <i>fpga</i> , pass in <i>control</i> and valid <i>socketInfo</i> .

Return Values

MPIMessageOK	if <i>ControlFPGADefaultGet</i> successfully gets creates a default FPGA filename.

See Also [MEIControlFPGA](#)

meiControlFPGADefaultOverride

Declaration

```
long meiFPGADefaultOverride(MPIControl control,
                             MEIControlFPGA *fpga,
                             const char *overrideFile,
                             MEIPlatformSocketInfo *socketInfo)
```

Required Header

stdmei.h

Description

FPGADefaultOverride checks to see if the *socketInfo* fits the board's physical configuration. If so, the FPGA filename is replaced with the *overrideFile*. This allows the user to specify FPGA files instead of using the MPI's default FPGA file.

control	a handle to the Control object.
*fpga	a pointer to MEIControlFPGA struct that contains the current file name string.
*overrideFile	is a character string that contains a desired filename.
*socketInfo	is a pointer to valid socket information.

Return Values

MPIMessageOK	if <i>ControlFPGAOverride</i> successfully replaces the default FPGA file with the desired overrideFile.

See Also

[MEIControlFPGA](#)

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 <i>ControlGateGet</i> 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](#)

Required Header

control	a handle to the Control object
*info	a pointer to MEIControlInfo that gets completed with the appropriate controller information.

MPIMessageOK	If meiControlInfo successfully retrieves the controller information.
MPIHandleVOID	if control is invalid

file:///D:/pdfs/030100/html/Software-MPI/docs/Control/Method/inf2.htm [7/22/2004 3:14:18 PM]

Required Header

control	a handle to a Control object
*io	a pointer to a structure containing the digital input and output values.

MPIMessageOK	if <i>ControlIoGet</i> successfully gets the I/O bits from controller and puts (writes) them in the structure.
MPIMessageARG_INVALID	if the <i>io</i> pointer points to NULL.

file:///D:/pdfs/030100/html/Software-MPI/docs/Control/Method/ioget1.htm [7/22/2004 3:14:18 PM]

mpiControlIoSet

```
Declaration      long mpiControlIoGet(MPIControl      control,  
                  MPIControlIo      *io);
```

Required Header `stdmpi.h`

Description	ControlIoSet writes the states of a controller's digital I/O using data from a structure pointed to by io . Some controller models have local digital I/O. Please see the controller hardware documentation for details.

control	a handle to a Control object
*io	a pointer to a structure containing the digital input and output values.

Return Values

MPIMessageOK	if <i>ControlIoSet</i> successfully gets the I/O bits from controller and puts (writes) them in the structure

See Also [mpiControlIoGet](#) | [MPIControlInput](#) | [MPIControlOutput](#)

meiControlIoBitGet

Declaration

```
meiControlIoBitGet(MPIControl  
MEIControlIoBit  
long control,  
bit,  
*value);
```

Required Header

```
stdmpi.h
```

Description

ControlIoBitGet reads the state of a controller digital input bit and writes it into a long pointed to by **value**. Some controller models have local digital I/O. Please see the controller hardware documentation for details

control	a handle to the Control object
bit	an enumerated bit number
*value	a pointer to a long. The value contains the state of the bit.

Return Values

MPIMessageOK	if <i>ControlIoBitGet</i> successfully gets the Control configuration and writes it into the structure(s).

See Also

[meiControlIoBitSet](#) | [MEIControlIoBit](#)

meiControlIoBitSet

Declaration

meiControlIoBitSet([MPIControl](#)
[MEIControlIoBit](#)
long
control,
bit,
*value);

Required Header

stdmpi.h

Description

ControlIoBitSet writes the state of a controller digital output bit using data from a value pointed to by a long. Some controller models have local digital I/O. Please see the controller hardware documentation for details

control	a handle to the Control object
bit	an enumerated bit number
*value	a pointer to a long. The value contains the state of the bit.

Return Values	
MPIMessageOK	if <i>ControlIoBitSet</i> successfully sets the Control configuration and writes it into the structure(s).

See Also

[meiControlIoBitGet](#) | [MEIControlIoBit](#)

Required Header `stdmei.h`

Return Values

See Also [meiControlSecondstoSamples](#) | [meiControlSamplestoSeconds](#) | [meiControlSampleWait](#)

Required Header

Return Values

See Also [meiControlSecondstoSamples](#) | [meiControlSampleCounter](#)

meiControlSampleWait

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

Required Header

Description	ControlSampleWait waits for <i>count</i> samples while the XMP motion controller (associated with <i>control</i>) 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 <i>ControlSampleWait</i> successfully waits for count samples while the XMP motion controller executes

See Also [meiControlSamplestoSeconds](#) | [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 <i>ControlSecondsToSamples</i> successfully converts the seconds to samples.

See Also [meiControlSamplestoSeconds](#) | [meiControlSampleCounter](#) | [meiControlSampleWait](#)

mpiControlType

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

Required Header

Description	When a Control object (<i>control</i>) is created, a type is used. ControlType writes this type to the contents of <i>type</i> .

Return Values

MPIMessageOK	if <i>ControlType</i> successfully gets the type (used when <i>control</i> was created) to the contents of <i>type</i>

See Also

Required Header

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(...)`.

```

/* Simple code to increment userbuffer[0] */
    MEIXmpData      *firmware;
    MEIXmpBufferData *buffer;

    long returnValue, tempBuffer;

    /* Get memory pointers */
    returnValue =
        mpiControlMemory(control,
            &firmware,
            &buffer);
    msgCHECK(returnValue);

    returnValue = mpiControlMemoryGet(control,
        &tempBuffer,
        &buffer->UserBuffer.Data[0],
        sizeof(buffer->UserBuffer.Data[0]));
    msgCHECK(returnValue);

    tempBuffer++;

    returnValue = mpiControlMemorySet(control,
        &buffer->UserBuffer.Data[0],
        &tempBuffer,
        sizeof(buffer->UserBuffer.Data[0]));
    msgCHECK(returnValue);

```

Return Values

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

See Also [mpiControlMemoryGet](#) | [mpiControlMemorySet](#) | [mpiControlMemoryAlloc](#) | [mpiControlMemoryCount](#) | [mpiControlMemoryFree](#)

Required Header

Return Values

See Also [mpiControlMemoryGet](#) | [mpiControlMemorySet](#) | [mpiControlMemory](#) | [mpiControlMemoryCount](#) | [mpiControlMemoryFree](#)

Required Header `control.h`

Return Values

See Also

Required Header

Return Values

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**).

Sample Code

```
/* Simple code to increment userbuffer[0] */
MEIXmpData      *firmware;
MEIXmpBufferData *buffer;

long returnValue, tempBuffer;

/* Get memory pointers */
returnValue =
    mpiControlMemory(control,
                      &firmware,
                      &buffer);
msgCHECK(returnValue);

returnValue = mpiControlMemoryGet(control,
                                  &tempBuffer,
                                  &buffer->UserBuffer.Data[0],
                                  sizeof(buffer->UserBuffer.Data[0]));
msgCHECK(returnValue);

tempBuffer++;

returnValue = mpiControlMemorySet(control,
                                  &buffer->UserBuffer.Data[0],
                                  &tempBuffer,
                                  sizeof(buffer->UserBuffer.Data[0]));
msgCHECK(returnValue);
```

Return Values

MPIMessageOK	if <i>ControlMemoryGet</i> 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**).

Sample Code

```
/* Simple code to increment userbuffer[0] */
MEIXmpData            *firmware;
MEIXmpBufferData *buffer;

long returnValue, tempBuffer;

/* Get memory pointers */
returnValue =
    mpiControlMemory(control,
        &firmware,
        &buffer);
msgCHECK(returnValue);

returnValue = mpiControlMemoryGet(control,
    &tempBuffer,
    &buffer->UserBuffer.Data[0],
    sizeof(buffer->UserBuffer.Data[0]));
msgCHECK(returnValue);

tempBuffer++;

returnValue = mpiControlMemorySet(control,
    &buffer->UserBuffer.Data[0],
    &tempBuffer,
    sizeof(buffer->UserBuffer.Data[0]));
msgCHECK(returnValue);
```

Return Values

MPIMessageOK	if <i>ControlMemorySet</i> successfully sets (writes) count bytes of application memory to control memory

See Also [mpiControlMemoryGet](#) | [mpiControlMemory](#) | [mpiControlMemoryAlloc](#) |
 [mpiControlMemoryCount](#) | [mpiControlMemoryFree](#)

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 <i>control</i> is invalid

See Also [mpiControlCreate](#)

meiControlCycleWait

```
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 <i>count</i> cycles

See Also

mpiControlInit

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 <i>ControlInit</i> successfully initializes the motion control device control

See Also [mpiControlDelete](#)

Required Header `stdmpi.h`

Return Values

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(...)**.

If **timeout** is **MPIWaitPOLL (0)**, then *ControlInterruptWait* will return immediately.

If **timeout** is **MPIWaitFOREVER (-1)**, then *ControlInterruptWait* will wait forever for an interrupt.

Otherwise, *ControlInterruptWait* will wait **timeout** milliseconds for an interrupt.

NOTE: For Windows operating systems, only **MPIWaitPOLL** and **MPIWaitFOREVER** are valid timeout values.

Return Values	
MPIMessageOK	if <i>ControlInterruptWait</i> waits for an interrupt from the motion controller
MPIMessageTIMEOUT	if <i>ControlInterruptWait</i> did not receive an interrupt within timeout ms.

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 <i>ControlInterruptWake</i> 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 <i>ControlReset</i> successfully resets the motion controller board

See Also

meiControlVersionMismatchOverride

Declaration long `meiControlVersionMismatchOverride`([MPIControl](#) `control`);

Required Header `stdmei.h`

Description [ControlMismatchOverride](#) overrides the version mismatch between the MPI and the Xmp.

This function is reserved for MEI use only and should not be used by a customer.

Return Values

MPIMessageOK	if the motion controller successfully overrides the version mismatch between the MPI and the Xmp.

See Also

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    axisCount;
    long    axisFrameCount[MPIControlMAX_AXES];
    long    captureCount;
    long    compareCount;
    long    compensatorCount;
    long    compensatorPointCount[MPIControlMAX_COMPENSATORS];
    long    cmdDacCount;
    long    auxDacCount;
    long    filterCount;
    long    motionCount;
    long    motorCount;
    long    recorderCount;
    long    recordCount[MPIControlMAX\_RECORDERS];
    long    sequenceCount;
    long    userVersion;
    long    sampleRate;
} MPIControlConfig;
```

Description

The **ControlConfig** structure specifies the controller configurations. It allocates the number of resources and configurations for the controller's operation. The controller's performance is inversely related to the DSP's load. The controller configuration structure allows the user to disable/enable objects for optimum performance.

For SynqNet controllers, changing the `sampleRate` or `TxTime` will cause the SynqNet network to be shutdown and re-initialized using the new `sampleRate` or `TxTime` values.

axisCount	Number of axis objects enabled for the controller. The controller's axis object handles the trajectory calculations for command position. For simple systems, set the <i>axisCount</i> equal to the <i>motorCount</i> .

axisFrameCount[]	An array of the number of frames for each axis frame buffer. Each frame is the size of MEIXmpFrame. The controller's frame buffers are dynamically allocated by changing the axisFrameCount. The default axisFrameCount is reasonable for most applications. A larger frame buffer may be required for long multi-point or cam motion profiles. The valid range is from 16 to the available memory. Use meiControlExtMemAvail(...) to determine the controller's available memory. Be sure to leave some free memory for potential future features.
captureCount	Number of capture objects enabled for the controller. The controller supports up to 16 captures. The controller's capture object manages the hardware resources to latch a motor's position feedback, triggered by a motor's input.
compareCount	Number of compare objects enabled for the controller. The controller's compare object manages the hardware resources to trigger a motor's output, triggered by a comparison between the motor's feedback and a pre-loaded position value.
compensatorCount	This value defines the number of enabled compensators.
compensatorPointCount	The number of points in the compensation table for each compensator. See Determining Required Compensator Table Size for more information. An array of the number of points in the compensation table for each compensator. Each point is 32bits. The controller's compensation tables are dynamically allocated by changing the <i>compensatorPointCount</i>. When using compensator objects, see Determining Required Compensator Table Size for more information on a proper value for the point count.
cmdDacCount	Number of command DACs (digital to analog converter) enabled for the controller. The controller's cmdDac transmits and scales a torque demand value to a SynqNet servo drive or a physical DAC circuit. There is one cmdDac per motor. Normally, the cmdDacCount should be equal to the motorCount.
auxDacCount	Number of auxilliary DACs (digital to analog converter) enabled for the controller. The controller's auxDac transmits and scales a torque demand value to a SynqNet servo drive or a physical DAC circuit. Auxilliary DACs can be used for sinusoidal motor commutation, where the cmdDac and auxDac provide the commutation phases. Or, auxilliary DACs can be used for general purpose analog outputs. There is one auxDac per motor.
filterCount	Number of filter objects enabled for a controller. The filter object handles the closed-loop servo calculations to control the motor. For simple systems, set the filterCount equal to the motorCount.

motionCount	Number of motion supervisor objects enabled for a controller. The controller's motion supervisor handles coordination of motion and events for an axis or group of axes. For simple systems, set the motionCount equal to the axisCount.
motorCount	Number of motor objects enabled for a controller. The controller's motor object handles the interface to the servo or stepper drive, dedicated I/O and general purpose motor related I/O. For simple systems, the motorCount should equal the number of physical motors connected to the controller (either directly or via SynqNet).
recorderCount	Number of data recorder objects enabled for a controller. The controller's recorder object handles collecting and buffering any data in controller memory. The enabled data recorders can collect up to a total of 32 addresses each sample. The valid range for the recordCount is 0 to 32.
recordCount	An array of the number of records for each data recorder buffer. Each data record is 32 bits. The controller's data recorder buffers can be dynamically allocated by changing the recordCount. A larger data recorder buffer may be required for higher sample rates, slow host computers, when running via client/server, or when a large number of data fields are being recorded. The valid range is 0 to the available memory. Use meiControlExtMemAvail(...) to determine the controller's available external memory.
sequenceCount	Number of sequence objects enabled for the controller. The controller's sequence object executes and manages a sequence of pre-compiled controller commands.
userVersion	A 32 bit user defined field. The userVersion can be used to mark a firmware image with an identifier. This is useful if multiple controller firmware images are saved to a file.
sampleRate	Number of controller foreground update cycles per second. For SynqNet controllers, this is also the cyclic update rate for the SynqNet network. During the controller's foreground cycle, the axis trajectories are calculated, the filters (closed-loop servo control) are calculated, motion is coordinated, the SynqNet data buffers are updated, and other time critical operations are performed. The default sample rate is 2000 (period = 500 microseconds). The minimum sampleRate for SynqNet systems is 1000 (period = 1 millisecond). The maximum is dependent on the controller hardware and processing load. There are several factors that must be considered to find an appropriate sampleRate for a system. The servo performance, the motion profile accuracy, the SynqNet network cyclic rate, the SynqNet drive update rates, controller background cycle update rate, and controller/application performance. For SynqNet systems, select a sampleRate that is a common multiple of the SynqNet drives connected to the network. For example, if the drive update rate is 8kHz, then appropriate controller sample rates are: 16000, 8000, 5333, 4000, 3200, 2667, 2286, 2000, 1778, 1600, 1455,

	1333, 1231, 1067, and 1000

MEIControlConfig

```
typedef struct MEIControlConfig {
    long                preFilterCount;
    long                TxTime;
    MEIXmpPreFilter     PreFilter[MEIXmpMAX_PreFilters];
    MEIXmpUserBuffer    UserBuffer;
} MEIControlConfig;
```

Description

preFilterCount	This value defines the number of enabled pre-filters.
TxTime	This value determines the controller's transmit time for the SynqNet data. The units are a percentage of the sample period. The default is 75%. Smaller TxTime values will reduce the latency between when the controller receives the data, calculates the outputs, and transmits the data. If the TxTime is too small, the data will be sent before the controller updates the buffer, which will cause a TX_FAILURE event.
PreFilter	This array defines the configuration for each pre-filter.
UserBuffer	This structure defines the controller's user buffer. This is used for custom features that require a controller data buffer.

Sample Code

```
/*
    Write a value to element index of the user buffer.
    Make sure to save topology to flash before doing this.
*/
void write2UserBufferFlash(MPIControl control, long value, long index)
{
    MPIControlConfig config;
    MEIControlConfig external;
    long returnValue;

    if((index < MEIXmpUserDataSize) && (index >= 0))
    {
        /* Make sure to save topology to flash before doing this */
        returnValue = mpiControlFlashConfigGet(control,
            MPIHandleVOID,
            &config,
            &external);
        msgCHECK(returnValue);
    }
}
```

```
        external.UserBuffer.Data[index] = value;

        returnValue = mpiControlFlashConfigSet(control,
            MPIHandleVOID,
            &config,
            &external);
        msgCHECK(returnValue);
    }
}
```

See Also [mpiControlConfigGet](#) | [mpiControlConfigSet](#) | [meiControlExtMemAvail](#) | [Dynamic Allocation of External Memory Buffers](#)

MEIControlInfo

MEIControlInfo

```
typedef struct MEIControlInfo {
    MEIControlInfoMpi      mpi;
    MEIControlInfoFirmware firmware;
    MEIControlInfoPld      pld;
    MEIControlInfoRincon   rincon;
    MEIControlInfoHardware hardware;
    MEIControlInfoDriver   driver;
}MEIControlInfo;
```

Description **ControlInfo** contains the information about the motion controller being used.

mpi	Information about the MPI software located on the host computer.
firmware	Information about the Firmware running on the controller.
pld	Information about the PLD located in the controller.
rincon	Information about the Rincon FPGA located on the controller.
hardware	Production information about the hardware stored in the controller.
driver	Information about the Driver, running on the host, used to interface with the controller.

See Also

MEIControlInfoDriver

Declaration

```
typedef struct MEIControlInfoDriver {
    char    version[MEIControlSTRING\_MAX];
} MEIControlInfoDriver;
```

Required Header stdmei.h

Description **ControlInfoDriver** is a structure that contains the version information of the connected hardware.

version	The version of the Driver the host uses to interface with the controller.

See Also

MEIControlInfoFirmware

Declaration

```
typedef struct MEIControlInfoFirmware {
    long    version;      /* MEIXmpVERSION_EXTRACT(SoftwareID) */
    long    option;       /* MEIXmpOPTION_EXTRACT(Option) */
    char    revision;     /* ('A' - 1) + MEIXmpREVISION_EXTRACT(SoftwareID) */
    long    subRevision;  /* MEIXmpSUB_REV_EXTRACT(Option) */
    long    branchId;
} MEIControlInfoFirmware;
```

Required Header stdmei.h

Description **ControlInfoFirmware** is a structure that contains read-only version information for the firmware running in the controller.

version	The major version number for the controller's firmware. To be compatible with the MPI library, this number must match the fwVersion in the MEIControlInfoMpi structure.
option	The firmware option number. Special or custom firmware is given a unique option number. An application or user can identify optional firmware from this value.
revision	The minor version number for the controller's firmware. Indicates a minor change or bug fix to the firmware code.
subRevision	The micro version value for the controller's firmware. Indicates a very minor change or bug fix to the firmware code.
branchId	Identifies an intermediate branch software revision. 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.

See Also [MEIControlInfoMPI](#)

MEIControlInfoHardware

Declaration

```
typedef struct MEIControlInfoHardware {
    char    modelName[MEIControlSTRING_MAX];
    char    serialNumber[MEIControlSTRING_MAX];
    char    type[MEIControlSTRING_MAX];
} MEIControlInfoHardware;
```

Required Header stdmei.h

Description **ControlInfoHardware** is a structure that contains the version information of the connected hardware.

modelName	The Controller's model number or t-level number (ex: T001-0001) which is stored on the hardware.
serialNumber	The Controller's serial number, which is unique to each controller.
type	The type of Controller (XMP or ZMP).

See Also

MEIControlInfoMpi

Declaration

```
typedef struct MEIControlInfoMpi {
    char        version[MEIControlSTRING_MAX];
    long        fwVersion;
    long        fwOption;
} MEIControlInfoMpi;
```

Required Header stdmei.h

Description **ControlInfoMpi** is a structure that contains read-only version information for the 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.
fwVersion	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.
fwOption	The firmware option number. Special or custom firmware is given a unique option number. An MPI library that requires optional firmware will have a value that must match the firmware's option number.

See Also [MEIControlInfoFirmware](#) | [MEIControlInfo](#)

MEIControlInfoPld

Declaration

```
typedef struct MEIControlInfoPld {
    char    version[MEIControlSTRING\_MAX];
    char    option[MEIControlSTRING\_MAX];
} MEIControlInfoPld;
```

Required Header `stdmei.h`

Description **ControlInfoPld** is a read-only structure that contains PLD version information. The PLD is a hardware component that contains logic to handle the controller's internal operation.

version	This is an 8-bit value in the hardware. The version string for the PLD. The PLD image is downloaded to the controller during manufacturing.
option	This is a 16-bit value (actually 2 8 bit values) in the hardware. The build option string for the PLD. The PLD option number is a coded value that describes the PLD image build type and target component. For XMP controllers, the option field has bits defining various features on the PCB - for example, the presence of the CAN interface, or the type of FPGA on the PCB.

See Also [MEIControlInfo](#)

MEIControlInfoRincon

Declaration

```
typedef struct MEIControlInfoRincon {
    char    version[MEIControlSTRING_MAX];
    char    package[MEIControlSTRING_MAX];
} MEIControlInfoRincon;
```

Required Header stdmei.h

Description **ControlInfoRincon** is a structure that contains read-only version information for the controller's Rincon image. The Rincon image contains the logic to operate a controller's SynqNet interface.

version	This is a 16-bit value in the hardware. The version string for the Rincon image on the controller.
package	This is a 16-bit value in the hardware. The package string identification for the Rincon. The package string is a coded value that describes the Rincon image build type and target component. Existing types are: 9201 - Rincon for XMP, XC2S100, PQ208 package 9601 - Rincon for XMP, XC2S100, FG256 package A102 - RinconZ for ZMP, XC2S300E, FT256 package A301 - RinconZ for ZMP, XC3S200, FT256 package The package and version data can be used to create the FPGA filename. For example, 221_9201.fpg is Rincon type 9201, version 221.

See Also [MEIControlInfo](#)

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,  
    MEIControlInputXESTOP    = MEIXmpControlIOMaskXESTOP,  
} MEIControlInput;
```

Description

ControlInput is an enumeration of a controller's local digital input bit masks. Each mask represents a discrete input.

See Also

[mpiControlIoGet](#) | [mpiControlIoSet](#) | [MEIControlOutput](#)

MPIControlIo

MPIControlIo

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

Description

The **ControlIo** structure contains controller's local digital input and output states. The digital inputs can be read and the digital outputs can be read or written through this structure.

input	An array of digital input values. Each bit mask is defined by the MEIControlInput enumeration.
output	An array of digital output values. Each bit mask is defined by the MEIControlOutput enumeration.

See Also

[MEIControlOutput](#) | [MEIControlInput](#) | [mpiControlIoGet](#) | [mpiControlIoSet](#)

MEIControlIoBit

Declaration

```
typedef enum {  
  
    MEIControlIoBitUSER_0_IN,  
    MEIControlIoBitUSER_1_IN,  
    MEIControlIoBitUSER_2_IN,  
    MEIControlIoBitUSER_3_IN,  
    MEIControlIoBitUSER_4_IN,  
    MEIControlIoBitUSER_5_IN,  
    MEIControlIoBitUSER_0_OUT,  
    MEIControlIoBitUSER_1_OUT,  
    MEIControlIoBitUSER_2_OUT,  
    MEIControlIoBitUSER_3_OUT,  
    MEIControlIoBitUSER_4_OUT,  
    MEIControlIoBitUSER_5_OUT,  
  
} MEIControlIoBit;
```

Required Header

stdmpi.h

Description

ControlIoBit is an enumeration of a controller's local digital I/O bit numbers.

MEIControlIoBitUSER_0_IN	controller's local input, bit number 0
MEIControlIoBitUSER_1_IN	controller's local input, bit number 1
MEIControlIoBitUSER_2_IN	controller's local input, bit number 2
MEIControlIoBitUSER_3_IN	controller's local input, bit number 3
MEIControlIoBitUSER_4_IN	controller's local input, bit number 4
MEIControlIoBitUSER_5_IN	controller's local input, bit number 5
MEIControlIoBitUSER_0_OUT	controller's local output, bit number 0
MEIControlIoBitUSER_1_OUT	controller's local output, bit number 1
MEIControlIoBitUSER_2_OUT	controller's local output, bit number 2
MEIControlIoBitUSER_3_OUT	controller's local output, bit number 3
MEIControlIoBitUSER_4_OUT	controller's local output, bit number 4
MEIControlIoBitUSER_5_OUT	controller's local output, bit number 5

Return Values

MPIMessageOK	if <i>ControlIoGet</i> successfully gets the I/O bits from controller and puts (writes) them in the structure.
MPIMessageARG_INVALID	if the <i>io</i> pointer points to NULL.

See Also [meiControlIoBitGet](#) | [meiControlIoBitSet](#)

MPIControlIoWords

MPIControlIoWords

```
#define MPIControlIoWords (1)
```

Description

ControlIoWords defines the number of 32 bit Input and Output words on the controller.

See Also

[MPIControlIo](#) | [MEIControlIoBit](#) | [MEIControlInput](#) | [MEIControlOutput](#)

MPIControlMemoryType

MPIControlMemoryType

```
typedef enum {
    MPIControlMemoryTypeUSER,
    MPIControlMemoryTypeDEFAULT = MPIControlMemoryTypeUSER
} MPIControlMemoryType;
```

Description

ControlMemoryType is an enumeration of controller memory types. The controller memory contains static and dynamic regions. The controller firmware defines the regions and the MPI configures the dynamic memory.

MPIControlMemoryTypeUSER	The dynamic portion of the controller's external memory that is not in use by the controller.
MPIControlMemoryTypeDEFAULT	Defined as MPIControlMemoryTypeUSER.

See Also

[mpiControlMemoryAlloc](#) | [mpiControlMemoryCount](#) | [mpiControlMemoryFree](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

MPIControlMessage / MEIControlMessage

MPIControlMessage

```
typedef enum {
    MPIControlMessageLIBRARY_VERSION,
    MPIControlMessageADDRESS_INVALID,
    MPIControlMessageCONTROL_INVALID,
    MPIControlMessageCONTROL_NUMBER_INVALID,
    MPIControlMessageTYPE_INVALID,
    MPIControlMessageINTERRUPTS_DISABLED,
    MPIControlMessageEXTERNAL_MEMORY_OVERFLOW,
    MPIControlMessageADC_COUNT_INVALID,
    MPIControlMessageAXIS_COUNT_INVALID,
    MPIControlMessageAXIS_FRAME_COUNT_INVALID,
    MPIControlMessageCAPTURE_COUNT_INVALID,
    MPIControlMessageCOMPARE_COUNT_INVALID,
    MPIControlMessageCMDDAC_COUNT_INVALID,
    MPIControlMessageAUXDAC_COUNT_INVALID,
    MPIControlMessageFILTER_COUNT_INVALID,
    MPIControlMessageMOTION_COUNT_INVALID,
    MPIControlMessageMOTOR_COUNT_INVALID,
    MPIControlMessageSAMPLE_RATE_TOO_LOW,
    MPIControlMessageRECORDER_COUNT_INVALID,
    MPIControlMessageCOMPENSATOR_COUNT_INVALID,
    MPIControlMessageAXIS_RUNNING,
    MPIControlMessageRECORDER_RUNNING,
} MPIControlMessage;
```

Description

MPIControlMessageLIBRARY_VERSION

The MPI Library does not match the application. This message code is returned by [mpiControlInit\(...\)](#) if the MPI's library (DLL) version does not match the MPI header files that were compiled with the application. To correct this problem, the application must be recompiled using the same MPI software installation version that the application uses at run-time.

MPIControlMessageADDRESS_INVALID

The controller address is not valid. This message code is returned by [mpiControlInit\(...\)](#) if the controller address is not within a valid memory range. [mpiControlInit\(...\)](#) only requires memory addresses for certain operating systems. To correct this problem, verify the controller memory address.

MPIControlMessageCONTROL_INVALID

Currently not supported.

MPIControlMessageCONTROL_NUMBER_INVALID

The controller number is out of range. This message code is returned by [mpiControlInit\(...\)](#) if the controller number is less than zero or greater than or equal to MaxBoards(8).

MPIControlMessageTYPE_INVALID

The controller type is not valid. This message code is returned by [mpiControlInit\(...\)](#) if the controller type is not a member of the MPIControlType enumeration.

MPIControlMessageINTERRUPTS_DISABLED

The controller interrupt is disabled. This message code is returned by [mpiControlInterruptWait\(...\)](#) if the controller's interrupt is not enabled. This prevents an application from waiting for an interrupt that will never be generated. To correct this problem, enable controller interrupts with [mpiControlInterruptEnable\(...\)](#) before waiting for an interrupt.

MPIControlMessageEXTERNAL_MEMORY_OVERFLOW

The controller's external memory will overflow. This message code is returned by [mpiControlConfigSet\(...\)](#) if the dynamic memory allocation exceeds the external memory available on the controller. To correct the problem, reduce the number/size of control configuration resources or use a controller model with a larger static memory component.

MPIControlMessageADC_COUNT_INVALID

The ADC count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of ADCs is greater than MEIXmpMAX_ADCs.

MPIControlMessageAXIS_COUNT_INVALID

The axis count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of axes is greater than MEIXmpMAX_Axes.

MPIControlMessageAXIS_FRAME_COUNT_INVALID

This message is returned from [mpiControlConfigSet\(...\)](#) if the value for MPIControlConfig.axisFrameCount is not a power of two or if axisFrameCount is less than MPIControlMIN_AXIS_FRAME_COUNT.

MPIControlMessageCAPTURE_COUNT_INVALID

The capture count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of captures is greater than MEIXmpMAX_Captures.

MPIControlMessageCOMPARE_COUNT_INVALID

The compare count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of compares is greater than MEIXmpMAX_Compare.

MPIControlMessageCMDDAC_COUNT_INVALID

The command DAC count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of command DACs is greater than MEIXmpMAX_DACs.

MPIControlMessageAUXDAC_COUNT_INVALID

The auxiliary DAC count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of auxiliary DACs is greater than MEIXmpMAX_DACs.

MPIControlMessageFILTER_COUNT_INVALID

The filter count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of filters is greater than MEIXmpMAX_Filters.

MPIControlMessageMOTION_COUNT_INVALID

The motion count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of motions is greater than MEIXmpMAX_MSs.

MPIControlMessageMOTOR_COUNT_INVALID

The motor count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of motors is greater than MEIXmpMAX_Motors.

MPIControlMessageSAMPLE_RATE_TO_LOW

The controller sample rate is too small. This message code is returned by [mpiControlConfigSet\(...\)](#) if the sample rate is less than 1000. SynqNet does not support cyclic data rates below 1kHz. The controller's sample rate specifies the SynqNet cyclic rate.

MPIControlMessageRECORDER_COUNT_INVALID

The recorder count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of recorders is greater than MEIXmpMAX_Recorders.

MPIControlMessageCOMPENSATOR_COUNT_INVALID

The compensator count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of compensators is greater than MPIControlMAX_COMPENSATORS.

MPIControlMessageAXIS_RUNNING

Attempting to configure the control object while axes are running. It is recommended that all configuration of the control object occur prior to commanding motion.

MPIControlMessageRECORDER_RUNNING

Attempting to configure the control object while a recorder is running. It is recommended that all configuration of the control object occur prior to operation of any recorder objects.

MEIControlMessage

```
typedef enum {
    MEIControlMessageFIRMWARE_INVALID,
    MEIControlMessageFIRMWARE_VERSION_NONE,
    MEIControlMessageFIRMWARE_VERSION,
    MEIControlMessageFPGA_SOCKETS,
    MEIControlMessageBAD_FPGA_SOCKET_DATA,
    MEIControlMessageNO_FPGA_SOCKET,
    MEIControlMessageINVALID_BLOCK_COUNT,
    MEIControlMessageSYNQNET_OBJECTS,
    MEIControlMessageSYNQNET_STATE,
    MEIControlMessageIO_BIT_INVALID,
} MEIControlMessage;
```

Description

MEIControlMessageFIRMWARE_INVALID

The controller firmware is not valid. This message code is returned by [mpiControlInit\(...\)](#) if the MPI library does not recognize the controller signature. After power-up or reset, the controller loads the firmware from flash memory. When the firmware executes, it writes a signature value into external memory. If [mpiControlInit\(...\)](#) does not recognize the signature, then the firmware did not execute properly. To correct this problem, download firmware and verify the controller hardware is working properly.

MEIControlMessageFIRMWARE_VERSION_NONE

The controller firmware version is zero. This message code is returned by control methods do not find a firmware version. This indicates the firmware did not execute at controller power-up or reset. To correct this problem, download firmware and verify the controller hardware is working properly.

MEIControlMessageFIRMWARE_VERSION

The controller firmware version does not match the software version. This message code is returned by control methods if the firmware version is not compatible with the MPI library. To correct this problem, either download compatible firmware or install a compatible MPI run-tim library.

MEIControlMessageFPGA_SOCKETS

The maximum number of FPGA socket types has been exceeded. This message code is returned by [meiFlashMemoryFromFile\(...\)](#) if the controller has more FPGA types than the controller has flash memory space to support them.

MEIControlMessageBAD_FPGA_SOCKET_DATA

Not supported.

MEIControlMessageNO_FPGA_SOCKET

The FPGA socket type does not exist. This message code is returned by [meiFlashMemoryFromFile\(...\)](#) if the controller does not support the FPGA type that was specified in the FPGA image file. To correct this problem, use a different FPGA image that is compatible with the controller.

MEIControlMessageINVALID_BLOCK_COUNT

Not supported.

MEIControlMessageSYNQNET_OBJECTS

Not supported.

MEIControlMessageSYNQNET_STATE

The controller's SynqNet state is not expected. This message code is returned by [mpiControlInit\(...\)](#), [mpiControlReset\(...\)](#) and [mpiControlConfigSet\(...\)](#) if the SynqNet network initialization fails to reach the SYNQ state. To correct this problem, check your node hardware and network connections.

MEIControlMessageIO_BIT_INVALID

The controller I/O bit is not valid. This message code is returned by [meiControlIoBitGet\(...\)](#) or [meiControlIoBitSet\(...\)](#) if the controller I/O bit is not a member of the MEIControlIoBit enumeration.

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 is an enumeration of a controller's local digital output bit masks. Each mask represents a discrete output.

See Also

[mpiControlIoGet](#) | [mpiControlIoSet](#) | [MEIControlInput](#)

MEIControlTrace

MEIControlTrace

```
typedef enum {  
    MEIControlTraceDYNA_ALLOC = MEIControlTraceFIRST << 0,  
} MEIControlTrace;
```

Description **ControlTrace** is an enumeration of control object trace bits to enable debug tracing.

MEIControlTraceDYNA_ALLOC	This trace bit enables tracing for calls that dynamically allocate controller memory.

See Also

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](#)

MPIControlMAX_AXES

Declaration `#define MPIControlMAX_AXES (32)`

Required Header `stdmpi.h`

Description Defines the maximum number of axes available on one controller.

See Also [MPIAxis](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

MPIControlMAX_COMPENSATORS

Declaration `#define MPIControlMAX_COMPENSATORS (4)`

Required Header `stdmpi.h`

Description Defines the maximum number of compensator objects available on one controller.

See Also [MPICompensator](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

MPIControlMAX_RECORDERS

Declaration `#define MPIControlMAX_RECORDERS (32)`

Required Header `stdmpi.h`

Description Defines the maximum number of recorder objects available on one controller.

See Also

MPIControlMIN_AXIS_FRAME_COUNT

Declaration `#define MPIControlMIN_AXIS_FRAME_COUNT (128)`

Required Header `stdmpi.h`

Description Defines the the minimum allowed value for which
MPIControlConfig.axisFrameCount can be set.

See Also [MPIControlConfig](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

MEIControlSTRING_MAX

Declaration `#define MEIControlSTRING_MAX (16)`

Required Header `stdmei.h`

Description Defines the maximum number of characters in MEIControlInfo strings.

See Also [MEIControlInfo](#) | [MEIControlInfoHardware](#)

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 and Record 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 [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](#)

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](#)

DriveMap Objects

Introduction

A **DriveMap** object is used to access information in a drive map file.

Methods

Create, Delete, Validate Methods

[meiDriveMapCreate](#)

[meiDriveMapDelete](#)

[meiDriveMapValidate](#)

Configuration and Information Methods

[meiDriveMapParamCount](#)

[meiDriveMapParamList](#)

[meiDriveMapConfigCount](#)

[meiDriveMapConfigList](#)

Data Types

[MEIDriveMapMessage](#)

[MEIDriveMapParamAccess](#)

[MEIDriveMapParamInfo](#)

[MEIDriveMapParamType](#)

[MEIDriveMapParamValue](#)

Constants

[MEIDriveMapParamMAX_STRING_LENGTH](#)

meiDriveMapCreate

Declaration `meiDriveMapCreate(char *fileName)`

Required Header `stdmei.h`

Description `DriveMapCreate` creates a DriveMap object associated with the file specified by *fileName*.

DriveMapCreate is the equivalent of a C++ constructor.

*filename	the drive map file.

Return Values

handle	to a DriveMap object. After creating a DriveMap object, it must be validated using <code>meiDriveMapValidate()</code> .
MPIHandleVOID	if the object could not be created.

See Also [meiDriveMapDelete](#) | [meiDriveMapValidate](#)

meiDriveMapDelete

Declaration `meiDriveMapDelete(MEIDriveMap driveMap);`

Required Header `stdmei.h`

Description **DriveMapDelete** deletes a DriveMap object and invalidates its handle.

DriveMapDelete is the equivalent of a C++ destructor.

driveMap	a handle of the DriveMap object to delete in the reverse order to avoid memory leaks.

Return Values

MPIMessageOK	if <i>DriveMapDelete</i> successfully deleted the object.

See Also [meiDriveMapCreate](#) | [meiDriveMapValidate](#)

meiDriveMapValidate

Declaration long **meiDriveMapValidate**([MEIDriveMap](#) **driveMap**);

Required Header stdmei.h

Description **DriveMapValidate** validates a DriveMap object and its handle.

DriveMapValidate is the equivalent of a C++ constructor.

driveMap	a handle to SqNode object.

Return Values

MPIMessageOK	if <i>driveMap</i> is a handle to a valid object.

See Also [meiDriveMapCreate](#) | [meiDriveMapDelete](#)

meiDriveMapParamCount

Declaration

```
long meiDriveMapParamCount (MEIDriveMap driveMap,
                             char *nodeName,
                             char *firmwareVersion,
                             long *paramsCount );
```

Required Header stdmei.h

Description **DriveMapParamCount** scans the drive map file for a drive entry that matches a particular drive. If an entry is found, then this function returns the number of drive parameters that need to be preserved for the configuration of the drive.

This function is normally used with the [meiDriveMapParamList](#) function. First, this function is called in order to get the size of the drive parameter list. Then the user can use this size to allocate enough memory to hold the complete parameter list before calling meiDriveMapParamList to fill in the list.

driveMap	a handle to a DriveMap object.
nodeName	the product/manufacturing text string of the node to search for. The nodeName of an SqNode object can be retrieved by calling meiSqNodeInfo. See MEISqNodeInfo .
firmwareVersion	The firmware version of the drive to search for. This information can be retrieved from an SqNode object by calling meiSqNodeDriveInfo. See MEISqNodeDriveInfo .
*paramsCount	pointer to the variable that will be set by this function.

Return Values

MPIMessageOK	if <i>meiDriveMapParamCount</i> successfully scans the drive map file for a drive entry that matches this node on the network and returns the number of drive parameters that need to be preserved for the configuration of the drive.

See Also [meiDriveMapParamList](#)

meiDriveMapParamList

Declaration

```
long meiDriveMapParamList (MEIDriveMap      driveMap,
                           char              *nodeName,
                           char              *firmwareVersion,
                           long              paramCount,
                           MEIDriveParamInfo *driveParamInfo);
```

Required Header stdmei.h

Description [SqNodeDriveMapScanParamsList](#) scans the drive map file for an entry that matches a particular drive. If a drive entry is found, this function writes the drive parameter information about each of the drive parameters to the *driveParamInfo* list.

This function is normally used with the meiDriveMapParamCount function. The meiDriveMapParamCount function is called first to get the size of the parameter list, the user can then use this size to allocate enough memory to hold the complete parameter list before calling this function to fill in the parameter list.

driveMap	a handle to a DriveMap object.
*nodeName	the product/manufacturing text string of the node to search for. The nodeName of an SqNode object can be retrieved by calling meiSqNodeInfo. See MEISqNodeInfo .
*firmwareVersion	The firmware version of the drive to search for. This information can be retrieved from an SqNode object by calling meiSqNodeDriveInfo. See MEISqNodeDriveInfo .
paramCount	the number of drive parameter information records that can be written to the driveParamInfo list.
*driveParamInfo	pointer to the list of drive parameter information records that will be filled in by this function.

Return Values

MPIMessageOK	if <i>meiDriveMapParamList</i> successfully scans the drive map file for an entry that matches the node on the network and writes the information about each of the drive parameters to the <i>driveParamInfo</i> list.

See Also [meiDriveMapParamCount](#)

meiDriveMapConfigCount

Declaration

```
long meiDriveMapConfigCount (MEIDriveMap driveMap,
                             char *nodeName,
                             char *firmwareVersion,
                             long *configCount);
```

Required Header stdmei.h

Description **DriveMapConfigCount** scans the drive map file for a drive entry that matches a particular drive. If an entry is found, this function returns the number of drive parameters that need to be preserved for the configuration of the drive.

This function is normally used with the meiDriveMapConfigList function. This function is called first in order to get the size of the drive configuration list. Then the user can use this size to allocate enough memory to hold the complete configuration list before calling meiDriveMapConfigList to fill in the list.

driveMap	a handle to a DriveMap object.
nodeName	the product/manufacturing text string of the node to search for. The nodeName of an SqNode object can be retrieved by calling meiSqNodeInfo. See MEISqNodeInfo .
firmwareVersion	The firmware version of the drive to search for. This information can be retrieved from an SqNode object by calling meiSqNodeDriveInfo. See MEISqNodeDriveInfo .
*configCount	pointer to the variable that will be set by this function.

Return Values

MPIMessageOK	if meiDriveMapConfigCount successfully scans the drive map file for a drive entry that matches this node on the network and returns the number of drive parameters that need to be preserved for the configuration of the drive.

See Also [meiDriveMapConfigList](#)

meiDriveMapConfigList

Declaration

```
long meiDriveMapConfigList (MEIDriveMap driveMap,
                             char *nodeName,
                             char *firmwareVersion,
                             long configCount,
                             long *configList);
```

Required Header stdmei.h

Description **DriveMapConfigList** scans the drive map file for a drive entry that matches a particular drive. If an entry is found, this function returns the list of drive parameters that need to be preserved for the configuration of the drive.

This function is normally used with the [meiDriveMapConfigCount](#) function. The meiDriveMapConfigCount function is called first in order to get the size of the drive configuration list. Then the user can use this size as a guide to allocate enough memory to hold the complete configuration list before calling this function to fill in the list.

driveMap	a handle to a DriveMap object.
*nodeName	the product/manufacturing text string of the node to search for. The nodeName of an SqNode object can be retrieved by calling meiSqNodeInfo. See MEISqNodeInfo .
*firmwareVersion	The firmware version of the drive to search for. This information can be retrieved from an SqNode object by calling meiSqNodeDriveInfo. See MEISqNodeDriveInfo .
configCount	the number of drive parameter information records that can be written to the configList list.
*configList	pointer to the list of drive parameters that make up the drive configuration that will be filled in by this function.

Return Values

MPIMessageOK	if <i>meiDriveMapConfigList</i> successfully scans the drive map file for a drive entry that matches this node on the network and returns the list of drive parameters that need to be preserved for the configuration of the drive.

See Also [meiDriveMapConfigCount](#)

MEIDriveMapMessage

MEIDriveMapMessage

```
typedef enum {
    MEIDriveMapMessageMAP_FILE_OPEN_ERROR,
    MEIDriveMapMessageMAP_FILE_FORMAT_INVALID,
    MEIDriveMapMessageNODE_NOT_FOUND_IN_MAP,
    MEIDriveMapMessageVERSION_NOT_FOUND_IN_MAP,
    MEIDriveMapMessageDRIVE_PARAM_READ_ONLY,
} MEIDriveMapMessage;
```

Description

MEIDriveMapMessageMAP_FILE_OPEN_ERROR

There was an error when opening the drive map file. The file may not exist, or access to the directory may not be allowed.

MEIDriveMapMessageMAP_FILE_FORMAT_INVALID

The format of the drive map file is invalid.

MEIDriveMapMessageNODE_NOT_FOUND_IN_MAP

The node type specified by the nodeName parameter was not found in the driveMap file.

MEIDriveMapMessageVERSION_NOT_FOUND_IN_MAP

The drive firmware version specified by the firmwareVersion parameter was not found in the driveMap file.

MEIDriveMapMessageDRIVE_PARAM_READ_ONLY

A read-only parameter is included in the configuration list for the specified drive in the driveMap file.

See Also

MEIDriveMapParamAccess

MEIDriveMapParamAccess

```
typedef enum MEISqNodeDriveParamAccess {  
    MEIDriveMapParamAccessREAD_WRITE,  
    MEIDriveMapParamAccessREAD_ONLY,  
} MEIDriveMapParamAccess;
```

Description

The **DriveMapParamAccess** structure indicates what type of access is possible with the drive parameter.

This field of the [MEIDriveMapParamInfo](#) structure indicates what type of access is possible with this drive parameter.

See Also

[MEIDriveMapParamInfo](#)

MEIDriveMapParamInfo

MEIDriveMapParamInfo

```
typedef struct MEISqNodeDriveParamInfo {
    unsigned long      parameter;
    char               name[MEISqNodeDriveParamMAX_STRING_LENGTH];
    MEIDriveMapParamAccess access; /* 0=>rw, 1=>ro */
    MEIDriveMapParamType type;
    char               validValues[MEISqNodeDriveParamMAX_STRING_LENGTH];
    long               defaultValue;
    char               help[MEISqNodeDriveParamMAX_STRING_LENGTH];
} MEIDriveMapParamInfo;
```

Description

The [DriveMapParamInfo](#) structure holds a set of information describing a drive parameter. This structure is read from the drives map file "drives.dm."

parameter	the number used to address this drive parameter.
name	a null terminated string giving a name for this drive parameter.
access	defines if this drive parameter is read-only or read-write.
type	the data type for this drive parameter. (ex: signed32, unsigned32, float, etc.)
validValues	a string describing the possible valid values for this drive parameter.
defaultValue	The factory default value this drive parameter.
help	A string describing this drive parameter.

See Also

[MEIDriveMapParamType](#)

MEIDriveMapParamType

MEIDriveMapParamType

```
typedef enum MEIDriveParamType {
    MEISqNodeDriveParamTypeSIGNED32,
    MEISqNodeDriveParamTypeSIGNED16,
    MEISqNodeDriveParamTypeSIGNED8,
    MEISqNodeDriveParamTypeUNSIGNED32,
    MEISqNodeDriveParamTypeUNSIGNED16,
    MEISqNodeDriveParamTypeUNSIGNED8,
    MEISqNodeDriveParamTypeHEX,
    MEISqNodeDriveParamTypeENUMERATED,
    MEISqNodeDriveParamTypeMASK,
    MEISqNodeDriveParamTypeCHARACTER,
    MEISqNodeDriveParamTypeSTRING,
    MEISqNodeDriveParamTypeSINGLE,
    MEISqNodeDriveParamTypeACTION,
} MEIDriveMapParamType;
```

Description [DriveMapParamType](#) is an enumeration that indicates which data format should be used with the drive parameter.

MEISqNodeDriveParamTypeSIGNED32	A signed 32bit integer.
MEISqNodeDriveParamTypeSIGNED16	A signed 16bit integer.
MEISqNodeDriveParamTypeSIGNED8	A signed 8bit integer.
MEISqNodeDriveParamTypeUNSIGNED32	An unsigned 32bit integer.
MEISqNodeDriveParamTypeUNSIGNED16	An unsigned 16bit integer.
MEISqNodeDriveParamTypeUNSIGNED8	An unsigned 8bit integer.
MEISqNodeDriveParamTypeHEX	An integer representred in hexadecimal notation.
MEISqNodeDriveParamTypeENUMERATED	An integer where each value can be represented by a different name.
MEISqNodeDriveParamTypeMASK	An integer where each bit has a unique name.
MEISqNodeDriveParamTypeCHARACTER	An single ASCII character.
MEISqNodeDriveParamTypeSTRING	A null terminated string of characters.
MEISqNodeDriveParamTypeSINGLE	A single precision floating point number.
MEISqNodeDriveParamTypeACTION	Writing to this parameter will perform an action on the drive. No data is assiated with this type.

MEIDriveMapParamValue

MEIDriveMapParamValue

```
typedef union {
    long          signed32;
    short         signed16;
    char          signed8;
    unsigned long unsigned32;
    unsigned short unsigned16;
    unsigned char unsigned8;
    unsigned long enumerated;
    unsigned long hex;
    unsigned long mask;
    char          character;
    char          * string;
    float         single;
} MEIDriveMapParamValue;
```

Description

The **MEIDriveMapParamValue** union holds the value of a drive parameter. The different fields allow this data type to hold all the different types of drive parameters. The MEISqNodeDriveParamType enumeration is used to identify which of the fields within the MEISqNodeDriveParamValue union to use.

See Also [MEIDriveMapParamType](#)

MEIDriveMapParamMAX_STRING_LENGTH

MEIDriveMapParamMAX_STRING_LENGTH

```
#define  MEIDriveMapParamMAX_STRING_LENGTH  (256)
```

Description

The **DriveMapParamMAX_STRING_LENGTH** macro defines the maximum length of a string that can be read from or written to a drive parameter.

See Also

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

<u>mpiEventStatusGet</u>	Get Event status
<u>mpiEventStatusSet</u>	Set Event status

Data Types

[MPIEventMessage](#)
[MEIEventNotifyData](#)
[MPIEventStatus](#)
[MEIEventStatusInfo](#)
[MPIEventType](#) / [MEIEventType](#)

Constants

<u>MPIEventStatusINFO_COUNT_MAX</u>	defines the size of the MPIEventStatus.info[] array.

Required Header `event.h`

Return Values

See Also [mpiEventStatusSet](#) | [meiEventStatusInfo](#)

mpiEventStatusSet

Declaration

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

Required Header

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

Return Values

MPIMessageOK	if <i>EventStatusSet</i> 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

The event type is not valid. This message code is returned by `mpiEventStatusSet(...)` if the event type is not a member of the [MPIEventType](#) or `MEIEventType` enumerations.

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.
info	Contains information on what generated the event and the conditions under which it was generated. MEIEventStatusInfo simplifies decoding this array. Sample code is shown on the MEIEventStatusInfo page.

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        node;    /* MEIEventTypeCAN_FIRST... MEIEventTypeCAN_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 {
            /* Data associated with the CAN event. */
            long data[4];
        } can;
        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.node	The CAN Node number of the MEICan object that generated the event.
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.can.data	A union that contains extra data about the CAN event that 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 */
    MPIEventTypeAMP_WARNING,          /* 9 */

    /* Motion events */
    MPIEventTypeMOTION_DONE,          /* 10 */
    MPIEventTypeMOTION_AT_VELOCITY,   /* 11 */

    /* Recorder events */
    MPIEventTypeRECORDER_HIGH,        /* 12 */
    MPIEventTypeRECORDER_FULL,        /* 13 */
    MPIEventTypeRECORDER_DONE,        /* 14 */

    /* External events */
    MPIEventTypeEXTERNAL,             /* 15 */
} 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.
MPIEventTypeAMP_WARNING	This event type indicates an Amp Warning 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_HIGH	This event type indicates that the controller's recorded data exceeded the buffer's high limit.
MPIEventTypeRECORDER_FULL	This event type indicates that the controller's recorded data has filled the buffer.
MPIEventTypeRECORDER_DONE	This event type indicates that the controller has recorded the number of requested data records.
MPIEventTypeEXTERNAL	This event type indicates an External event was generated from an external source.

MEIEventType

```
typedef enum {
    /* Motor events */
    MEIEventTypeLIMIT_USER0,
    MEIEventTypeLIMIT_USER1,
    MEIEventTypeLIMIT_USER2,
    MEIEventTypeLIMIT_USER3,
    MEIEventTypeLIMIT_USER4,
    MEIEventTypeLIMIT_USER5,
    MEIEventTypeLIMIT_USER6,
    MEIEventTypeLIMIT_USER7,
    MEIEventTypeLIMIT_USER8,
    MEIEventTypeLIMIT_USER9,
    MEIEventTypeLIMIT_USER10,
    MEIEventTypeLIMIT_USER11,
    MEIEventTypeLIMIT_USER12,
    MEIEventTypeLIMIT_USER13,
    MEIEventTypeLIMIT_USER14,
    MEIEventTypeLIMIT_USER15,
```

```

/* Motion events */
MEIEventTypeMOTION_OUT_OF_FRAMES,
MEIEventTypeMOTION_RESERVED0,

/* Axis events */
MEIEventTypeIN_POSITION_COARSE,
MEIEventTypeIN_POSITION_FINE,
MEIEventTypeSETTLED
MEIEventTypeAT_TARGET,
MEIEventTypeFRAME,
MEIEventTypeAXIS_RESERVED0,
MEIEventTypeAXIS_RESERVED1,

/* SynqNet events */
MEIEventTypeSYNQNET_DEAD,
MEIEventTypeSYNQNET_RX_FAILURE,
MEIEventTypeSYNQNET_TX_FAILURE,
MEIEventTypeSYNQNET_NODE_FAILURE,
MEIEventTypeSYNQNET_RECOVERY,

/* SqNode events */
MEIEventTypeSQNODE_IO_ABORT,
MEIEventTypeSQNODE_NODE_DISABLE,
MEIEventTypeSQNODE_NODE_ALARM,
MEIEventTypeSQNODE_ANALOG_POWER_FAULT,
MEIEventTypeSQNODE_USER_FAULT,
MEIEventTypeSQNODE_NODE_FAILURE,

/* CAN events */
MEIEventTypeCAN_BUS_STATE,
MEIEventTypeCAN_RECEIVE_OVERRUN,
MEIEventTypeCAN_EMERGENGY,
MEIEventTypeCAN_NODE_BOOT,
MEIEventTypeCAN_HEALTH,
MEIEventTypeCAN_DIGITAL_INPUT,
MEIEventTypeCAN_ANALOG_INPUT,

} 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. User Limit number 0.

MEIEventTypeLIMIT_USER1

This event type indicates a User Limit event was generated from a Motor object. User Limit number 1.

MEIEventTypeLIMIT_USER2

This event type indicates a User Limit event was generated from a Motor object. User Limit number 2.

MEIEventTypeLIMIT_USER3

This event type indicates a User Limit event was generated from a Motor object. User Limit number 3.

MEIEventTypeLIMIT_USER4

This event type indicates a User Limit event was generated from a Motor object. User Limit number 4.

MEIEventTypeLIMIT_USER5

This event type indicates a User Limit event was generated from a Motor object. User Limit number 5.

MEIEventTypeLIMIT_USER6

This event type indicates a User Limit event was generated from a Motor object. User Limit number 6.

MEIEventTypeLIMIT_USER7

This event type indicates a User Limit event was generated from a Motor object. User Limit number 7.

MEIEventTypeLIMIT_USER8

This event type indicates a User Limit event was generated from a Motor object. User Limit number 8.

MEIEventTypeLIMIT_USER9

This event type indicates a User Limit event was generated from a Motor object. User Limit number 9.

MEIEventTypeLIMIT_USER10

This event type indicates a User Limit event was generated from a Motor object. User Limit number 10.

MEIEventTypeLIMIT_USER11

This event type indicates a User Limit event was generated from a Motor object. User Limit number 11.

MEIEventTypeLIMIT_USER12

This event type indicates a User Limit event was generated from a Motor object. User Limit number 12.

MEIEventTypeLIMIT_USER13

This event type indicates a User Limit event was generated from a Motor object. User Limit number 13.

MEIEventTypeLIMIT_USER14

This event type indicates a User Limit event was generated from a Motor object. User Limit number 14.

MEIEventTypeLIMIT_USER15

This event type indicates a User Limit event was generated from a Motor object. User Limit number 15.

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 that an In Fine Position event was generated from an Axis object.

MEISynqNetMessageREADY_TIMEOUT

The node failed to be ready for a service command within the timeout. This message code is returned by MPI methods that fail a service command transaction because the node is not ready to accept service commands. To correct this problem, check your node hardware. There are 32 possible message codes for this error. Each message code specifies a different node, from node number 0 to 31.

MEIEventTypeSETTLED

Equivalent to MEIEventTypeIN_POSITION_FINE.

MEIEventTypeAT_TARGET

Reserved Frame Event.

MEIEventTypeFRAME

This event type 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 currently not supported and is reserved for future use or custom axis events.

MEIEventTypeAXIS_RESERVED1

This event type indicates that a Reserved Axis event was generated from an Axis object.

This event type is currently not supported and is reserved for future use or custom axis events.

MEIEventTypeSYNQNET_DEAD

The SynqNet network was shutdown due to a communication failure. This status/event occurs when the controller fails to read/write data to the SynqNet network interface from an RX_FAILURE or a TX_FAILURE. To recover from a DEAD event, the network must be shutdown and reinitialized. SYNQNET_DEAD is latched by the controller, use [meiSynqNetEventReset\(...\)](#) to clear the status/event bit.

MEIEventTypeSYNQNET_RX_FAILURE

SynqNet network data receive failure. Generated when the controller fails to receive the packet data buffer (Rincon DMA to internal memory) in two successive controller samples. A SYNQNET_RX_FAILURE is most likely caused by an incorrect RX_COPY_TIMER value (internal) or a timing problem. To recover from an RX_FAILURE event, the network must be shutdown and reinitialized. SYNQNET_RX_FAILURE is latched by the controller, use [meiSynqNetEventReset\(...\)](#) to clear the status/event bit.

MEIEventTypeSYNQNET_TX_FAILURE

SynqNet network data transmission failure. Generated when the controller fails to transmit the packet data buffer in two successive controller samples. This occurs when the maximum foreground time exceeds the Tx time percentage of the controller's sample period. The default Tx time value is 75% of the controller's sample period. To correct Tx failures, either increase the Tx time or decrease the controller's sample rate. To recover from a TX_FAILURE event, the network must be shutdown and reinitialized. SYNQNET_TX_FAILURE is latched by the controller, use [meiSynqNetEventReset\(...\)](#) to clear the status/event bit.

MEIEventTypeSYNQNET_NODE_FAILURE

SynqNet node failure. Generated when any node's upstream or downstream packet error rate counters exceed the failure limit. The failure limits are configured with [meiSqNodeConfigSet\(...\)](#). Use [meiSynqNetStatus\(...\)](#) to read the nodeFailedMask to identify the failed nodes. Also, a SQNODE_NODE_FAILURE will be generated for each node that fails. SYNQNET_NODE_FAILURE is latched by the controller, use [meiSynqNetEventReset\(...\)](#) to clear the status/event bit. To recover from a node failure, the network must be shutdown and reinitialized.

MEIEventTypeSYNQNET_RECOVERY

SynqNet fault recovery. Generated when any node's upstream or downstream packet error rate counters exceed the fault limit and the data traffic is redirected around the fault. The fault limits are configurable via [meiSqNodeConfigSet\(...\)](#). SYNQNET_RECOVERY is latched by the controller. Use [meiSynqNetEventReset\(...\)](#) to clear the status/event bit.

MEIEventTypeSQNODE_IO_ABORT

SynqNet node I/O abort. Generated when the node I/O Abort is activated. When the I/O Abort is triggered, the node's outputs are disabled (set to the power-on condition). The node I/O Abort can be configured to trigger when either a Synq Lost occurs, Node Disable is active, a Power Fault occurs, or a User Fault is triggered. See MEISqNodeConfigIoAbort{...} for more details.

MEIEventTypeSQNODE_NODE_DISABLE

SynqNet node's Node Disable input is activated. Generated when the Node Disable input signal transitions from inactive to active. This signal is latched in hardware. Use [meiSqNodeEventReset\(...\)](#) to clear the status/event and the hardware latch.

MEIEventTypeSQNODE_NODE_ALARM

SynqNet node alarm digital output. Generated when the node alarm output signal transitions from inactive to active. This signal is not latched.

MEIEventTypeSQNODE_ANALOG_POWER_FAULT

SynqNet node analog power failure. Generated when the node's power failure input bit transitions from inactive to active. The power fault circuit is node specific, but is typically connected to an analog power monitor. This signal is latched in hardware. Use [meiSqNodeEventReset\(...\)](#) to clear the status/event and the hardware latch.

MEIEventTypeSQNODE_USER_FAULT

SynqNet node user fault. Generated when the node's user configurable fault is triggered. The user fault can be configured to monitor any controller memory address and compare the masked value to a specified pattern. This signal is latched by the controller, use [meiSqNodeEventReset\(...\)](#) to clear the status/event bit.

MEIEventTypeSQNODE_NODE_FAILURE

SynqNet node failure. Generated when a node's upstream or downstream packet error rate counters exceed the failure limit. The failure limits are configured with [meiSqNodeConfigSet\(...\)](#). SQNODE_NODE_FAILURE is latched by the controller, use [meiSqNodeEventReset\(...\)](#) to clear the status/event bit. To recover from a node failure, the network must be shutdown and reinitialized.

MEIEventTypeCAN_BUS_STATE

The BusState has changed. Data[0] contains the new bus state.

MEIEventTypeCAN_RECEIVE_OVERRUN

The CAN hardware detected a receive overrun.

MEIEventTypeCAN_EMERGENCY

An emergency message was received from a node. Data[0] contains the node number. Data[1 to 4] contains the contents of the emergency message.

MEIEventTypeCAN_NODE_BOOT

A node boot message was received from a node. Data[0] contains the node number.

MEIEventTypeCAN_HEALTH

The health of a node has changed. Data[0] contains the node number. Data[1] contains the new node health.

MEIEventTypeCAN_DIGITAL_INPUT

A digital input event was received from a node. Data[0] contains the node number. Data[1 to 4] contains the new input state.

MEIEventTypeCAN_ANALOG_INPUT

An analog input event was received from a node. Data[0] contains the node number. Data[1 to 4] contains the new input state.

See Also [MPIEventMask](#) | [MPIEventMgr](#) | [MPINotify](#) | [MPIEventStatus](#) | [meiSynqNetEventReset](#) | [Special Note](#) on the use of MPIEventTypeENCODER_FAULT

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](#)

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](#)

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

[mpiEventMaskALL](#) / [meiEventMaskALL](#)

Set all events within the event mask.

[mpiEventMaskAND_ASSIGN](#)

Assign dst all events masked by both *src* and *dst*.

[mpiEventMaskASSIGN](#)

Assign the value of event mask *src* to the event mask *dst*.

[mpiEventMaskAXIS](#) / [meiEventMaskAXIS](#)

Set all MPIAxis events within the event mask.

[mpiEventMaskBIT](#)

Set mask to only handle events of type *type*.

[mpiEventMaskBitGET](#)

Reports if a mask is set to handle events of type *type*.

[mpiEventMaskBitSET](#)

Sets mask to handle events of type *type*.

[mpiEventMaskBIT_POSITION](#)

Returns the bit number that is associated with MPI/MEI event type.

[mpiEventMaskBIT_POSITION_MASK](#)

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

[mpiEventMaskCLEAR](#)

Set mask to handle no events.

[mpiEventMaskCOMPLEMENT](#)

Change the value of every bit within the event mask.

[mpiEventMaskEXTERNAL](#)

Set external events within the event mask.

[mpiEventMaskGET](#)

Reports if a mask is set to handle events of type *type*.

[mpiEventMaskIS_CLEAR](#)

[mpiEventMaskIS_EQUAL](#)

Tests the equality of two event masks.

[mpiEventMaskMOTION](#) / [meiEventMaskMOTION](#)

Set all MPIMotion events within the event mask.

[mpiEventMaskMOTOR](#) / [meiEventMaskMOTOR](#)

Set all MPIMotor events within the event mask.

[mpiEventMaskOR_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*.

[meiEventMask**SYNQNET**](#)

Set all MEISynqNet events within the event mask.

[meiEventMask**SQNODE**](#)

Set all MEISqNode events within the event mask.

[mpiEventMask**WORD**](#)

Constants

[MEIEventMask**BITS_IN_ELEMENT**](#)

Define the number of bits in each data element of MPIEventMask.

[MPIEventMask**ELEMENTS**](#)

Define the number of data elements in a MPIEventMask.

[MPIEventMask**ELEMENT_TYPE**](#)

Define what the data type MPIEventMask is comprised of.

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](#) | [MPI EventMask Objects](#)

mpiEventMaskALL / meiEventMaskALL

mpiEventMaskALL

Declaration

```
mpiEventMaskALL(mask)
```

Required Header stdmpi.h

Description

EventMaskALL is a macro that sets all the bits associated with MPI events in the event mask. The MPI event types are defined in the MPIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiXxxxxEventNotifySet(...) method which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

meiEventMaskALL

Declaration

```
meiEventMaskALL(mask)
```

Required Header stdmei.h

Description

EventMaskALL is a macro that sets all the bits associated with MEI events in the event mask. The MEI event types are defined in the MEIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also [MPIEventType](#) | [MPIEventMask](#)

mpiEventMaskAND_ASSIGN

Declaration

```
mpiEventMaskAND_ASSIGN(dst, src)
```

Required Header stdmpi.h

Description **EventMaskOR_ASSIGN** is a macro that bitwise ANDs all the bits associated with MPI/MEI events in the event mask *src* with *dst* and assigns the result to *dst*. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

dst	An array of unsigned longs. Use MPIEventMask to declare the dst. Each bit in the array represents a mask for a particular event.
src	An array of unsigned longs. Use MPIEventMask to declare the src. Each bit in the array represents a mask for a particular event.

See Also [MPIEventMask](#) | [MPIEventMaskASSIGN](#) | [MPIEventMaskOR_ASSIGN](#) |

mpiEventMaskASSIGN

Declaration

```
mpiEventMaskASSIGN(dst, src)
```

Required Header

stdmpi.h

Description

EventMaskASSIGN is a macro that assigns all the bits associated with MPI/MEI events in the event mask *src* to the event mask *dst*. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

dst	An array of unsigned longs. Use MPIEventMask to declare the dst. Each bit in the array represents a mask for a particular event.
src	An array of unsigned longs. Use MPIEventMask to declare the src. Each bit in the array represents a mask for a particular event.

See Also

[MPIEventMask](#) | [mpiEventMaskOR_ASSIGN](#) | [mpiEventMaskAND_ASSIGN](#)

mpiEventMaskAXIS / meiEventMaskAXIS

mpiEventMaskAXIS

Declaration

mpiEventMaskAXIS(mask)

Required Header stdmpi.h

Description

EventMaskAXIS is a macro that assigns all the bits associated with MPI Axis object events to the event *mask*. The MPI event types are defined in the MPIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

meiEventMaskAXIS

Declaration

meiEventMaskAXIS(mask)

Required Header stdmei.h

Description

EventMaskAXIS is a macro that assigns all the bits associated with MEI Axis object events to the event *mask*. The MEI event types are defined in the MEIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also

[MPIEventMask](#) | [MEIEventType](#)

mpiEventMaskBIT

Declaration

```
mpiEventMaskBIT(mask, type)
```

Required Header stdmpi.h

Description **EventMaskBIT** is a macro that assigns a bit associated with MPI/MEI event type in the event mask. The event types are defined in the MPIEventType and MEIEventType enumerations. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.
type	An enumerated event type. Use an enumerated value from MPIEventType or MEIEventType.

See Also [MPIEventMask](#) | [MPIEventType](#) | [MEIEventType](#) | [mpiEventMaskBitGET](#) | [mpiEventMaskBitSET](#) |

mpiEventMaskBitGET

Declaration

```
mpiEventMaskBitGET(mask, bit)
```

Required Header stdmpi.h

Description **EventMaskBitGet** is a macro that returns TRUE if the *bit* associated with a MPI/MEI event in the event *mask* is TRUE. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.
bit	A bit number associated with an event type in an event mask.

Returns	TRUE if the specified bit is TRUE in the event mask. FALSE if the specified bit is FALSE in the event mask.

See Also [MPIEventMask](#) | [mpiEventMaskBIT](#) | [mpiEventMaskBitSET](#) |

mpiEventMaskBitSET

Declaration

```
mpiEventMaskBitSET(mask, bit, value)
```

Required Header

stdmpi.h

Description

EventMaskBitSET is a macro that sets (value = TRUE) or clears (value = FALSE) the *bit* associated with a MPI/MEI event in the event *mask*. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.
bit	A bit number associated with an event type in an event mask.
value	TRUE to set a bit, FALSE to clear a bit.

See Also

[MPIEventMask](#) | [mpiEventMaskBitGET](#) | [mpiEventMaskBIT](#) |

mpiEventMaskBIT_POSITION

Declaration

```
mpiEventMaskBIT_POSITION ( type )
```

Required Header

```
stdmpi.h
```

Description

EventMaskBIT_POSITION is a macro that returns the bit number that is associated with MPI/MEI event *type*. The event types are defined in the MPIEventType and MEIEventType enumerations.

type	An enumerated event type. Use an enumerated value from MPIEventType or MEIEventType.

Returns

A bit number associated with the event type.

See Also

[MPIEventMask](#) | [MPIEventType](#) | [MEIEventType](#) | [mpiEventMaskBIT](#) | [mpiEventMaskBitGET](#) | [mpiEventMaskBitSET](#) |

mpiEventMaskBIT_POSITION_MASK

Declaration

```
mpiEventMaskBIT_POSITION_MASK ( type )
```

Required Header

stdmpi.h

Description

EventMaskBIT_POSITION_MASK is a macro that returns an event mask with a bit set that is associated with MPI/MEI event *type*. The event types are defined in the MPIEventType and MEIEventType enumerations. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method which configures the controller to generate events.

type	An enumerated event type. Use an enumerated value from MPIEventType or MEIEventType.

Returns

An event mask. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also

[MPIEventMask](#) | [MPIEventType](#) | [MEIEventType](#) | [mpiEventMaskBIT](#) | [mpiEventMaskBitGET](#) | [mpiEventMaskBitSET](#) |

mpiEventMaskCLEAR

Declaration

mpiEventMaskCLEAR(*mask*)

Required Header stdmpi.h

Description **EventMaskCLEAR** is a macro that clears all the bits in an event *mask*.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also [MPIEventMask](#) | [mpiEventMaskIS_CLEAR](#) | [mpiEventMaskSET_ALL](#)

mpiEventMaskCOMPLEMENT

Declaration

mpiEventMaskCOMPLEMENT(*mask*)

Required Header stdmpi.h

Description

EventMaskCOMPLEMENT is a macro that inverts all the bits in an event *mask*. If a bit associated with an event type is TRUE, EventMaskCOMPLEMENT will set the bit to FALSE. And likewise, if a bit associated with an event type is FALSE, EventMaskCOMPLEMENT will set the bit to TRUE.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also

[MPIEventMask](#) | [mpiEventMaskCLEAR](#) | [mpiEventMaskSET_ALL](#)

mpiEventMaskEXTERNAL

Declaration `mpiEventMaskEXTERNAL(mask)`

Required Header `stdmpi.h`

Description `EventMaskEXTERNAL` is a macro that assigns all the bits associated with MPI External events to the event *mask*. After the event mask bits are initialized, the mask can be passed to a `mpiObjectEventNotifySet(...)` method, which configures the controller to generate events.

mask	An array of unsigned longs. Use <code>MPIEventMask</code> to declare the mask. Each bit in the array represents a mask for a particular event.

See Also [MPIEventMask](#)

mpiEventMaskGET

Declaration

```
mpiEventMaskGET(mask, type)
```

Required Header stdmpi.h

Description **EventMaskGET** is a macro that returns TRUE if the bit associated with the event *type* in the event *mask* is TRUE. The event types are defined in the MPIEventType and MEIEventType enumerations. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.
type	An enumerated event type. Use an enumerated value from MPIEventType or MEIEventType.

Returns	TRUE if the bit associated with the event type is TRUE in the event mask. FALSE if the bit associated with the event type is FALSE in the event mask.

See Also [MPIEventMask](#) | [MPIEventType](#) | [MEIEventType](#) | [mpiEventMaskSET](#) | [mpiEventMaskSET_ALL](#) |

mpiEventMaskIS_CLEAR

Declaration

mpiEventMaskIS_CLEAR(mask)

Required Header stdmpi.h

Description **EventMaskIS_CLEAR** is a macro that returns TRUE if all the bits in an event *mask* are FALSE.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

Returns	TRUE is all the bits in an event mask are FALSE. FALSE if any bit in an event mask is TRUE.

See Also [MPIEventMask](#) | [mpiEventMaskCLEAR](#) | [mpiEventMaskSET_ALL](#)

mpiEventMaskIS_EQUAL

Declaration

```
mpiEventMaskIS_EQUAL(mask1, mask2)
```

Required Header

stdmpi.h

Description

EventMaskIS_EQUAL is a macro that returns TRUE if event *mask1* is the same as event *mask2*. The MPI event types are defined in the MPIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method which configures the controller to generate events.

mask1	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.
mask2	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

Returns

TRUE if event mask1 is the same as event mask2.
FALSE if event mask1 is different from event mask2.

See Also

[MPIEventMask](#) | [mpiEventMaskGET](#) | [mpiEventMaskSET](#)

mpiEventMaskMOTION / meiEventMaskMOTION

mpiEventMaskMOTION

Declaration

mpiEventMaskMOTION(*mask*)

Required Header stdmpi.h

Description

EventMaskMOTION is a macro that assigns all the bits associated with MPI Motion object events to the event *mask*. The MPI event types are defined in the MPIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

meiEventMaskMOTION

Declaration

meiEventMaskMOTION(*mask*)

Required Header stdmei.h

Description

EventMaskMOTION is a macro that assigns all the bits associated with MPI Motion object events to the event *mask*. The MPI event types are defined in the MPIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also

[MPIEventMask](#) | [MPIEventType](#)

mpiEventMaskMOTOR / meiEventMaskMOTOR

mpiEventMaskMOTOR

Declaration

```
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), \
                           mpiEventMaskBitSET((mask), MPIEventTypeAMP_WARNING, 1)
```

Required Header stdmpi.h

Description **EventMaskMOTOR** is a macro that assigns all the bits associated with MPI Motor object events to the event *mask*. The MPI event types are defined in the MPIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

meiEventMaskMOTOR

Declaration **meiEventMaskMOTOR**(mask)

Required Header stdmei.h

Description **EventMaskMOTOR** is a macro that assigns all the bits associated with MEI Motor object events to the event *mask*. The MEI event types are defined in the MEIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also [MPIEventMask](#) | [MEIEventType](#)

mpiEventMaskOR_ASSIGN

Declaration

```
mpiEventMaskOR_ASSIGN(dst, src)
```

Required Header stdmpi.h

Description **EventMaskIS_EQUAL** is a macro that bitwise ORs all the bits associated with MPI/MEI events in the event mask *src* with *dst* and assigns the result to *dst*. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

dst	An array of unsigned longs. Use MPIEventMask to declare the dst. Each bit in the array represents a mask for a particular event.
src	An array of unsigned longs. Use MPIEventMask to declare the src. Each bit in the array represents a mask for a particular event.

See Also [MPIEventMask](#) | [MPIEventMaskASSIGN](#) | [MPIEventMaskAND_ASSIGN](#)

mpiEventMaskRECORDER

Declaration

```
mpiEventMaskRECORDER(mask)  mpiEventMaskBitSET((mask), MPIEventTypeRECORDER_HIGH, 1),
\
                             mpiEventMaskBitSET((mask), MPIEventTypeRECORDER_FULL, 1),
\
                             mpiEventMaskBitSET((mask), MPIEventTypeRECORDER_DONE, 1)
```

Required Header stdmpi.h

Description **EventMaskRECORDER** is a macro that assigns all the bits associated with MPI Recorder object events to the event *mask*. The MPI event types are defined in the MPIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also [MPIEventMask](#) | [MPIEventType](#)

mpiEventMaskSET

Declaration `mpiEventMaskSET(mask, type)`

Required Header stdmpi.h

Description `EventMaskSET` is a macro that sets the *bit* associated with MPI event type in the event *mask*. The event types are defined in the MPIEventType and MEIEventType enumerations. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.
type	An enumerated event type. Use an enumerated value from MPIEventType or MEIEventType.

See Also [MPIEventMask](#) | [mpiEventMaskGET](#) | [MPIEventType](#) | [MEIEventType](#) | [mpiEventMaskSET_ALL](#)

mpiEventMaskSET_ALL

Declaration

```
mpiEventMaskSET_ALL(mask, type)
```

Required Header stdmpi.h

Description **EventMaskSET_ALL** is a macro that sets all the *bits* associated with MPI/MEI event types that are less than the specified event *type*. The event types are defined in the MPIEventType and MEIEventType enumerations. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.
type	An enumerated event type. Use an enumerated value from MPIEventType or MEIEventType.

See Also [MPIEventMask](#) | [MPIEventType](#) | [MEIEventType](#) | [mpiEventMaskGET](#) | [mpiEventMaskSET](#)

meiEventMaskSYNQNETH

Declaration

```
meiEventMaskSYNQNETH(mask)
```

Required Header stdmei.h

Description **EventMaskSYNQNETH** is a macro that assigns all the bits associated with MEI SynqNet object events to the event mask. The MEI event types are defined in the MEIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also [MPIEventMask](#) | [MEIEventType](#)

meiEventMaskSQNODE

Declaration

meiEventMaskSQNODE(*mask*)

Required Header stdmei.h

Description

EventMaskSQNODE is a macro that assigns all the bits associated with MEI SqNode object events to the event *mask*. The MEI event types are defined in the MEIEventType enumeration. After the event mask bits are initialized, the mask can be passed to a mpiObjectEventNotifySet(...) method, which configures the controller to generate events.

mask	An array of unsigned longs. Use MPIEventMask to declare the mask. Each bit in the array represents a mask for a particular event.

See Also [MPIEventMask](#) | [MEIEventType](#)

mpiEventMaskWORD

Declaration

mpiEventMaskWORD(**type**)

Required Header stdmpi.h

Description **EventMaskWORD** is a macro that returns the word number that is associated with MPI/MEI event ***type***. The event types are defined in the MPIEventType and MEIEventType enumerations.

type	An enumerated event type. Use an enumerated value from MPIEventType or MEIEventType.

Returns

The word number associated with the event type.

See Also [MPIEventMask](#) | [MPIEventType](#) | [MEIEventType](#) | [mpiEventMaskBIT_POSITION](#)

MEIEventMaskBITS_IN_ELEMENT

MEIEventMaskBITS_IN_ELEMENT

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

Description [**EventMaskBITS_IN_ELEMENT**](#) defines the number of bits in each data element of MPIEventMask.

NOTE: MEIEventMaskBITS_IN_ELEMENT replaced
mpiEventMaskBITS_IN_ELEMENT.

See Also [MPIEventMask](#) | [mpiEventMaskBIT](#) | [mpiEventMaskBitGET](#) | [mpiEventMaskBitSET](#) |
[mpiEventMaskBIT_POSITION_MASK](#)

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** object manages the collection and distribution of event messages from the controller to the host. Events include normal motion completion, motor limits, fault conditions, data recorder buffer full, network node faults, etc. The EventNotify methods enable the application to request host notification for specified events, while ignoring other events. The EventMgr receives the event and then distributes the event via the Notify object to the tasks that are waiting for the event. Events that are faults are latched; the fault condition and the event must be cleared before the event can be generated again.

To collect events, the EventMgr must be serviced via the [mpiEventMgrService\(...\)](#) routine. The EventMgr can be polled by periodically by calling the mpiEventMgrService(...) routine or can be placed in an interrupt service routine. For your convenience, there is an apputil module that provides a serviceCreate(...) function that creates an EventMgr service routine for Windows OSs. The Service thread can be configured for polling or interrupts. Sources for the apputil module are included with the software distribution, so you can port it to other OSs.

The controller has a circular buffer which holds up to 128 event messages. If the EventMgr is not serviced within 128 events, event messages will be overwritten as new events occur. For interrupt driven EventMgr service threads, this is not an issue. For a polling EventMgr service thread, this could be an issue if the service thread does not have enough CPU cycles. In the worst-case scenario, events could be lost.

The best way to avoid lost events is to use an interrupt driven EventMgr service routine. If you're using polling, the next best thing is to make sure the EventMgr services the events at a high enough rate to avoid controller message buffer rollover. Determining the worst case EventMgr service latency and the maximum event message rate can be tricky. A simple method is to estimate the event message frequency and make sure each EventMgr can poll at least once for every 32 events (safety factor of 4). If you know that some events occur more frequently than others, then you may want to increase the polling frequency for EventMgrs that process the most frequent events and decrease the polling frequency for EventMgrs that process less frequently.

Generally, use only one EventMgr in your application. Do not use multiple interrupt driven EventMgr service threads to collect the same event. Suppose you configure an EventMgr to service Motion Done events from MS 0 and you configure another EventMgr with MEIEventMgrServiceConfig.allProcesses = TRUE. If one EventMgr collects the event message and acknowledges the interrupt before the other EventMgr can respond, it will miss the event.

If you want to monitor ALL events with MEIEventMgrServiceConfig.allProcesses =

TRUE, then use polling (not interrupts). See the sample app EventLog.c for an example.

Methods

Create, Delete, Validate Methods

<u>mpiEventMgrCreate</u>	Create EventMgr object
<u>mpiEventMgrDelete</u>	Delete EventMgr object
<u>mpiEventMgrValidate</u>	Validate EventMgr object

Configuration and Information Methods

<u>mpiEventMgrConfigGet</u>	Get EventMgr config
<u>mpiEventMgrConfigSet</u>	Set EventMgr config
<u>mpiEventMgrEvent</u>	Request event notification for all Notify objects on EventMgr's list
<u>mpiEventMgrServiceConfigGet</u>	Get processes that EventMgr will service
<u>mpiEventMgrServiceConfigSet</u>	Set processes that EventMgr will service

Action Methods

<u>mpiEventMgrFlush</u>	Flush pending EventMgr events
<u>mpiEventMgrService</u>	Get list of all pending asynchronous events

Relational Methods

List Methods- for Control Objects

<u>mpiEventMgrControl</u>	Return handle of indexth Control object in list
<u>mpiEventMgrControlAppend</u>	Append Control's handle to list
<u>mpiEventMgrControlCount</u>	Count the number of Control objects associated with EventMgr (in list)
<u>mpiEventMgrControlFirst</u>	Return handle to first Control object in list
<u>mpiEventMgrControlIndex</u>	Return the index of a Control object in list
<u>mpiEventMgrControlInsert</u>	Insert Control handle into list
<u>mpiEventMgrControlLast</u>	Get handle to last Control object in list
<u>mpiEventMgrControlListGet</u>	Get list of Control objects associated with EventMgr
<u>mpiEventMgrControlListSet</u>	Create a list of Control objects associated with EventMgr
<u>mpiEventMgrControlNext</u>	Get handle to next Control object in list
<u>mpiEventMgrControlPrevious</u>	Get handle to previous Control object in list
<u>mpiEventMgrControlRemove</u>	Remove a Control object's handle from list

List Methods- for Notify Objects

<u>mpiEventMgrNotify</u>	Return handle to a Notify object associated with EventMgr
<u>mpiEventMgrNotifyAppend</u>	Append Notify object to list
<u>mpiEventMgrNotifyCount</u>	Return number of Notify objects in list
<u>mpiEventMgrNotifyFirst</u>	Get first Notify object in list

[mpiEventMgrNotifyIndex](#)

Get index value for a Notify object in list

[mpiEventMgrNotifyInsert](#)

Place a Notify object after another Notify object in list

[mpiEventMgrNotifyLast](#)

Get handle to the Notify object that is last on the list

[mpiEventMgrNotifyListGet](#)

Get a list of Notify objects

[mpiEventMgrNotifyListSet](#)

Create a list of Notify objects

[mpiEventMgrNotifyNext](#)

Get the Notify object just after notify in list

[mpiEventMgrNotifyPrevious](#)

Get the Notify object just before notify in list

[mpiEventMgrNotifyRemove](#)

Remove a Notify object from list

Data Types

[MPIEventMgrMessage](#)

[MEIEventMgrServiceConfig](#)

mpiEventMgrCreate

Declaration `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 <i>EventMgrDelete</i> 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](#)

Required Header `stdmpi.h`

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).

external either points to a structure of type **MEIEventManagerConfig** or is NULL.

Return Values

MPIMessageOK	if <i>EventManagerConfigGet</i> 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 <i>EventMgrConfigSet</i> successfully sets (writes) the EventMgr's configuration using data from the structure(s)

See Also

[mpiEventMgrConfigGet](#)

Required Header

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.

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

file:///D:/pdfs/030100/html/Software-MPI/docs/EventMgr/Method/evt1.htm [7/22/2004 5:28:40 PM]

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 <i>EventMgrServiceConfigGet</i> 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 <i>EventMgrFlush</i> successfully flushes any pending EventMgr events

See Also

Required Header

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.

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

Return Values

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

See Also

Required Header

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

MPIMessageOK	if <i>EventManagerControlAppend</i> successfully appends control to the list of motion controllers associated with an <i>EventManager</i>
MPIMessageHANDLE_INVALID	Either <i>eventMgr</i> or <i>control</i> is an invalid handle.
MPIMessageOBJECT_ON_LIST	<i>control</i> is already on the list.
MPIMessageNO_MEMORY	Not enough memory was available.

file:///D:/pdfs/030100/html/Software-MPI/docs/EventMgr/Method/cnlapd1.htm [7/22/2004 5:28:41 PM]

mpiEventMgrControlCount

Declaration long `mpiEventMgrControlCount`([MPIEventMgr](#) `eventMgr`)

Required Header stdmpi.h

Description Returns the number of elements on the list.

<code>eventMgr</code>	a handle to the EventMgr object.

Return Values

number	of motion controllers associated with an EventMgr (<i>eventMgr</i>)
-1	if <i>eventMgr</i> is an invalid handle
0	if <i>eventMgr</i> 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 conjunction 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 (<i>eventMgr</i>)
MPIHandleVOID	if <i>eventMgr</i> is invalid if <i>eventMgr</i> has no associated motion controllers
MPIMessageHANDLE_INVALID	if <i>eventMgr</i> is an invalid handle.

See Also [mpiEventMgrControlLast](#)

Required Header

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

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

file:///D:/pdfs/030100/html/Software-MPI/docs/EventMgr/Method/cnlinux1.htm [7/22/2004 5:28:42 PM]

Required Header

Return Values

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 conjunction 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 (<i>eventMgr</i>)
MPIHandleVOID	if <i>eventMgr</i> is invalid if <i>eventMgr</i> has no associated motion controllers
MPIMessageHANDLE_INVALID	if <i>eventMgr</i> is an invalid handle.

See Also [mpiEventMgrControlFirst](#)

Required Header

Return Values

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

file:///D:/pdfs/030100/html/Software-MPI/docs/EventMgr/Method/cnllisget1.htm [7/22/2004 5:28:42 PM]

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 <i>EventMgrControlListSet</i> 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 conjunction 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 <i>control</i> in the list of motion controllers associated with an EventMgr (<i>eventMgr</i>)
MPIHandleVOID	if <i>eventMgr</i> is invalid if <i>control</i> is the last motion controller
MPIMessageHANDLE_INVALID	Either <i>eventMgr</i> or <i>control</i> is an invalid handle.

See Also [mpiEventMgrControlPrevious](#)

Required Header

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

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.

file:///D:/pdfs/030100/html/Software-MPI/docs/EventMgr/Method/cnlpre1.htm [7/22/2004 5:28:43 PM]

Required Header

Return Values

See Also [mpiEventManagerControl](#)

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 <i>index</i> th Notify object associated with an EventMgr (<i>eventMgr</i>)
MPIHandleVOID	if <i>eventMgr</i> is invalid if <i>index</i> is less than 0 if <i>index</i> is greater than or equal to mpiEventMgrCount(eventMgr)
MPIMessageARG_INVALID	<i>index</i> is a negative number.
MEIListMessageELEMENT_NOT_FOUND	<i>index</i> is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	<i>eventMgr</i> is an invalid handle.

See Also

Required Header

eventMgr	a handle to the EventMgr object.
notify	a handle to a Notify object.

Return Values

MPIMessageOK	if <i>EventManagerNotifyAppend</i> successfully appends a Notify object to the list of Notify objects maintained by an EventMgr object
MPIMessageHANDLE_INVALID	Either <i>eventMgr</i> or <i>notify</i> is an invalid handle.
MPIMessageOBJECT_ON_LIST	if <i>notify</i> 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 (<i>eventMgr</i>)
-1	if <i>eventMgr</i> 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 conjunction 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 (<i>eventMgr</i>)
MPIHandleVOID	if <i>eventMgr</i> is invalid if the list (of Notify objects) is empty
MPIMessageHANDLE_INVALID	if <i>eventMgr</i> is an invalid handle.

See Also

[mpiEventMgrNotifyLast](#)

Required Header

eventMgr	a handle to the EventMgr object.
notify	a handle to the EventMgr object.

index	of <i>notify</i> in the list (of Notify objects) maintained by an EventMgr (<i>eventMgr</i>)
-1	if eventMgr is invalid if <i>notify</i> was not found in the list

file:///D:/pdfs/030100/html/Software-MPI/docs/EventMgr/Method/nfyinx1.htm [7/22/2004 5:28:45 PM]

Required Header

Return Values

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 conjunction 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 (<i>eventMgr</i>)
MPIHandleVOID	if <i>eventMgr</i> is invalid if the list (of Notify objects) is empty
MPIMessageHANDLE_INVALID	if <i>eventMgr</i> is an invalid handle.

See Also [mpiEventMgrNotifyFirst](#)

Required Header `stdmpi.h`

Return Values

See Also [mpiEventMgrNotifyListSet](#)

Declaration long `mpiEventMgrNotifyListSet`([MPIEventMgr](#) `eventMgr`,
 long `notifyCount`,
 [MPINotify](#) `*notifyList`)

Required Header

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 <i>EventMgrNotifyListSet</i> successfully creates a list of Notify objects using the Notify handles specified by <i>notifyList</i>

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 conjunction 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 conjunction 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](#)

Required Header

Return Values

See Also

MPIEventMgrMessage

MPIEventMgrMessage

```
typedef enum {

    MPIEventMgrMessageEVENTMGR_INVALID,
} MPIEventMgrMessage;
```

Description

MPIEventMgrMessageEVENTMGR_INVALID

Not supported.

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

<u>mpiFilterCreate</u>	Create Filter object
<u>mpiFilterDelete</u>	Delete Filter object
<u>mpiFilterValidate</u>	Validate Filter object

Configuration and Information Methods

<u>mpiFilterConfigGet</u>	Get Filter configuration
<u>mpiFilterConfigSet</u>	Set Filter configuration
<u>mpiFilterFlashConfigGet</u>	Get flash configuration for Filter
<u>mpiFilterFlashConfigSet</u>	Set flash configuration for Filter
<u>mpiFilterGainGet</u>	Get gain coefficients
<u>mpiFilterGainSet</u>	Set current gain index
<u>mpiFilterGainIndexGet</u>	Get current gain index
<u>mpiFilterGainIndexSet</u>	Set current gain index

Memory Methods

<u>mpiFilterMemory</u>	Get address to Filter memory
<u>mpiFilterMemoryGet</u>	Copy data from Filter memory to application memory
<u>mpiFilterMemorySet</u>	Copy data from application memory to Filter memory

Relational Methods

<u>mpiFilterAxisMapGet</u>	Get object map of axes associated with Filter
<u>mpiFilterAxisMapSet</u>	Set axes associated with Filter
<u>mpiFilterControl</u>	Return handle of Control that is associated with Filter

<u>mpiFilterMotorMapGet</u>	Get object map of Motors associated with Filter
<u>mpiFilterMotorMapSet</u>	Set Motors to be associated with Filter
<u>mpiFilterNumber</u>	Get index of Filter (for Control list)

Action Methods

<u>mpiFilterIntergratorReset</u>	Reset the integrators of filter.

Postfilter Methods

<u>meiFilterPostfilterGet</u>	Reads postfilter information.
<u>meiFilterPostfilterSet</u>	Writes postfilter information.
<u>meiFilterPostfilterSectionGet</u>	Reads postfilter section information.
<u>meiFilterPostfilterSectionSet</u>	Writes postfilter section information.

Data Types

[MPIFilter**Coeff**](#)
[MPIFilter**Config**](#) / [MEIFilter**Config**](#)
[MEIFilter**Form**](#)
[MPIFilter**Gain**](#)
[MEIFilter**GainIndex**](#)
[MEIFilter**GainPID**](#)
[MEIFilter**GainPIDCoeff**](#)
[MEIFilter**GainPIV**](#)
[MEIFilter**GainPIVCoeff**](#)
[MEIFilter**GainTypePID**](#)
[MEIFilter**GainTypePIV**](#)
[MPIFilter**Message**](#)
[MEIFilter**Type**](#)
[MEI**PostfilterSection**](#)

Constants

[MPIFilter**CoeffCOUNT_MAX**](#)
[MPIFilter**GainCOUNT_MAX**](#)
[MEI**MaxBiQuadSections**](#)

mpiFilterCreate

Declaration	<code>MPIFilter mpiFilterCreate(MPIControl control, long number)</code>

Required Header

Description	FilterCreate creates a Filter object associated with a filter (<i>number</i>), that is located on a motion controller (<i>control</i>). <i>FilterCreate</i> 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 <i>FilterDelete</i> 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

Description	FilterConfigGet gets a Filter's (<i>filter</i>) configuration and writes it into the structure pointed to by <i>config</i> , and also writes it into the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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 <i>FilterConfigGet</i> 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

Description	FilterConfigSet sets a Filter's (<i>filter</i>) configuration using data from the structure pointed to by <i>config</i> , and from the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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 <i>FilterConfigSet</i> 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 <i>FilterFlashConfigGet</i> successfully writes the Filter’s flash configuration to the structure(s) <i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> 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 <i>FilterFlashConfigSet</i> successfully sets the Filter’s flash configuration using data from the structure(s) <i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

[MEIFlash](#) | [mpiFilterFlashConfigGet](#) | [MEIFilterConfig](#)

mpiFilterGainGet

[illegible]

Required Header

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 <i>FilterGainGet</i> successfully writes the gain coefficients to the structure

See Also [mpiFilterGainSet](#)

mpiFilterGainSet

```

Declaration
long mpiFilterGainSet(MPIFilter filter,
                      long gainIndex,
                      MPIFilterGain *gain)

```

Required Header

Description	FilterGainSet sets the gain coefficients of a Filter (<i>filter</i> , for the gain index specified by <i>gainIndex</i>) using data from the structure pointed to by <i>gain</i> .

Return Values

MPIMessageOK	if <i>FilterGainSet</i> successfully sets the gain coefficients of a Filter using data from the structure

See Also [mpiFilterGainGet](#)

Required Header

If the filter is in state `MEIXmpSwitchType MEIXmpSwitchTypeMOTION_ONLY`, the gain index is automatically changed by the firmware as described at [MEIXmpSwitchType](#). When the filter is in state `MEIXmpSwitchType MEIXmpSwitchTypeNONE`, the gain index is controlled by the user.

Gain Scheduling is a feature that switches filter gains for the acceleration, deceleration, constant velocity, and idle states of motion. The post filters are not affected by gain scheduling. Standard algorithms are used with gain scheduling (PID, PIV).

Return Values

MPIMessageOK	if <i>FilterGainIndexGet</i> successfully writes the gain index to the location

See Also [MPIFilterConfig](#) | [mpiFilterConfigGet](#) | [mpiFilterConfigSet](#) | [MEIFilterGainIndex](#) | [MEIXmpSwitchType](#) | [mpiFilterGainIndexSet](#) | [mpiFilterGainIndexGet](#) | [mpiFilterGainGet](#) | [mpiFilterGainSet](#)

mpiFilterGainIndexSet

Declaration	long mpiFilterGainIndexSet (MPIFilter filter , long gainIndex)
Required Header	stdmpi.h
Description	FilterGainIndexSet sets the current gain index of a Filter (<i>filter</i>) to <i>gainIndex</i>. Writing the gain index controls what gain table is currently being used. If the filter is in state <code>MEIXmpSwitchType</code> MEIXmpSwitchTypeMOTION_ONLY, the gain index is changed automatically by the firmware as described at MEIXmpSwitchType. Be aware that the filter can change the gain index in real-time, thereby overwriting your changes in this mode. When the filter is in state <code>MEIXmpSwitchType</code> MEIXmpSwitchTypeNONE, the gain index is controlled by the user. This is the normal state when using <code>FilterGainIndexSet(...)</code>. Gain Scheduling is a feature that switches filter gains for the acceleration, deceleration, constant velocity, and idle states of motion. The post filters are not affected by gain scheduling. Standard algorithms are used with gain scheduling (PID, PIV).

Return Values	
<code>MPIMessageOK</code>	if <i>FilterGainIndexSet</i> successfully sets the current gain index to <i>gainIndex</i>
See Also	MPIFilterConfig mpiFilterConfigGet mpiFilterConfigSet MEIFilterGainIndex MEIXmpSwitchType mpiFilterGainIndexSet mpiFilterGainIndexGet mpiFilterGainGet mpiFilterGainSet

Required Header

Return Values

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 <i>FilterMemoryGet</i> 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 <i>FilterMemoryGet</i> successfully copies data from application memory to Filter memory

See Also

[mpiFilterMemoryGet](#) | [mpiFilterMemory](#)

mpiFilterAxisMapGet

```
long mpiFilterAxisMapGet(MPIFilter filter, MPIObjectMap *axisMap)
```

Required Header

Description	FilterAxisMapGet gets the object map of the Axes that are associated with a Filter (<i>filter</i>), and writes it into the structure pointed to by <i>axisMap</i> .

Return Values

MPIMessageOK	if <i>FilterAxisMapGet</i> successfully writes the object map of Axes to the structure

See Also [mpiFilterAxisMapSet](#)

Required Header

Return Values

MPIMessageOK	if <i>FilterAxisMapSet</i> successfully sets the Axes using the object map

file:///D:/pdfs/030100/html/Software-MPI/docs/Filter/Method/axmapset1.htm [7/22/2004 5:37:12 PM]

mpiFilterControl

Declaration [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 *motorMap)
```

Required Header

Description	FilterMotorMapGet
	gets the object map of the Motors associated with the Filter (<i>filter</i>), and writes it into the structure pointed to by <i>motorMap</i> .

Return Values

MPIMessageOK	if <i>FilterMotorMapGet</i> successfully writes the object map of the Motors to the structure

See Also [mpiFilterMotorMapSet](#)

mpiFilterMotorMapSet

[illegible]

Required Header

Description **FilterMotorMapSet** sets the Motors associated with the Filter (*filter*) using data from the object map specified by *motorMap*.

Return Values

MPIMessageOK	if <i>FilterMotorMapGet</i> successfully sets the Motors using data from the object map

See Also [mpiFilterMotorMapGet](#)

mpiFilterNumber

```
long mpiFilterNumber(MPIFilter filter,
                    long *number)
```

Required Header stdmpi.h

Description	For a motion controller that <i>filter</i> is associated with, FilterNumber writes the index of <i>filter</i> to the contents of <i>number</i> .

Return Values

MPIMessageOK	if <i>FilterNumber</i> 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 <i>mpiFilterIntegratorReset</i> successfully clears the integrators of <i>filter</i> .
MPIFilterMessageINVALID_ALGORITHM	if the <i>filter's</i> 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.

meiFilterPostfilterGet

Declaration

long `meiFilterPostfilterGet`(`MPIFilter` `filter`,
long `*sectionCount`,
`MEIPostfilterSection` `*sections`);

Required Header

`stdmei.h`

Description

`PostfilterGet` reads an `MPIFilter` object's postfilter configuration. It writes to *sectionCount* the number of sections within a postfilter if *sectionCount* is not NULL. It also writes to *sections* the current array of *filter*'s postfilter sections if sections is not NULL.

The MPI calculates the post filter coefficients and takes into consideration the sample rate of the controller at that time. If you change the sample rate of the controller, you will need to recalculate the post filters. This can be done for all filters specified in Hertz by setting the filters again with the MPI. The MPI will calculate the filters using the current servo sample rate.

Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

filter	the handle of the <code>MPIFilter</code> object whose postfilter configuration is to be read.
*sectionCount	the data location where the postfilter's current section count will be written.
*sections	the data location where the postfilter's current section configuration data will be written.

Sample Code

```
/* Count the number of resonator sections in a MPIFilter object's postfilter.
Sample usage:

returnValue =
    filterResonatorCount(filter, &resonatorCount);
*/

long filterResonatorCount(MPIFilter filter, long* count)
{
    MPIFilterConfig config;
    MEIPostfilterSection sections[MEIMaxBiQuadSections];
    long sectionCount, index;
    long returnValue = (count==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue =
            meiFilterPostfilterGet(filter, &sectionCount, sections);

        if (returnValue == MPIMessageOK)
        {
```



```

    for (*count=0, index=0; index sectionCount; ++index)
    {
        if (section[index].type == MEIFilterTypeRESONATOR) ++(*count);
    }
}
return returnValue;
}
```

Return Values

MPIMessageOK	if <i>PostfilterGet</i> successfully retrieves postfilter information.
MPIFilterMessageCONVERSION_DIV_BY_0	Returned when meiFilterPostfilterGet(...) cannot convert digital coefficients to analog coefficients. When this error occurs, the offending section(s) will report its type as MEIFilterTypeUNKNOWN and will not contain any analog data.
MPIFilterMessagePOSTFILTER_NOT_ENABLED	Returned when sections is not NULL and when no postfilter sections are enabled.
MPIFilterMessageINVALID_FILTER_FORM	Returned when meiFilterPostfilterGet(...) cannot interpret the current postfilter's form (when the form is something other than NONE, IIR, or BIQUAD).

See Also [MEIPostfilterSection](#) | [meiFilterPostfilterGet](#) | [meiFilterPostfilterSet](#) | [meFilterPostfilterSectionGet](#) | [MEIMaxBiQuadSections](#) |

[Post Filter Theory](#) section.

Declaration	long meiFilterPostfilterSet (MPIFilter	filter,
	long	*sectionsCount,
	MEIPostfilterSection	*sections);

Description	PostfilterSet sets the number of postfilter sections within an MPIFilter object and configures each postfilter section as well. If numberOfSections equals zero, then sections can be NULL and the postfilter will be disabled.

The MPI calculates the post filter coefficients and takes into consideration the sample rate of the controller at that time. If you change the sample rate of the controller, you will need to recalculate the post filters. This can be done for all filters specified in Hertz by setting the filters again with the MPI. The MPI will calculate the filters using the current servo sample rate.

Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

filter	the handle of the MPIFilter object whose postfilter sections will be configured.
*sectionsCount	the number of postfilter sections to set in the <i>filter</i> object.
*sections	a pointer to an array of MEIPostfilterSection data structures to be set in <i>filter</i> .

Sample Code

```

/* Set a 4th order low-pass post-filter by using two 2nd order low-pass sections.
   Sample usage:

   returnValue =
       fourthOrderLowPass(filter, 300 /* Hz */);
*/
long filterFourthOrderLowpass(MPIFilter filter, long breakPointFrequency)
{
    MPIFilterConfig config;
    MEIPostfilterSection section[MEIMaxBiQuadSections];
    long returnValue;

    section[0].type = MEIFilterTypeLOW_PASS;
    section[0].form = MEIFilterFormINT_BIQUAD;
    section[0].data.lowPass.breakpoint = breakPointFrequency;
    section[1] = section[0]; /* copy first section */

    returnValue =
        meiFilterPostfilterSet(filter, 2, section);

    return returnValue;
}

```

Return Values

MPIMessageOK	if <i>PostfilterSet</i> successfully writes postfilter information.

See Also [MEIPostfilterSection](#) | [meiFilterPostfilterGet](#) | [meFilterPostfilterSectionSet](#) | [MEIMaxBiQuadSections](#) | [Post Filter Theory](#) section.

meiFilterPostfilterSectionGet

Declaration

```
long meiFilterPostfilterSectionGet(MPIFilter filter,
                                   long      sectionNumber,
                                   MEIPostfilterSection *section);
```

Required Header stdmei.h

Description **PostfilterSectionGet** reads the configuration of a single section of an MPIFilter object's postfilter. It writes to ***section** the configuration of **filter**'s postfilter **sectionNumber**th section.

The MPI calculates the post filter coefficients and takes into consideration the sample rate of the controller at that time. If you change the sample rate of the controller, you will need to recalculate the post filters. This can be done for all filters specified in Hertz by setting the filters again with the MPI. The MPI will calculate the filters using the current servo sample rate.

Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

filter	the handle of the MPIFilter object whose postfilter section configuration is to be read.
sectionNumber	the index of the postfilter section whose configuration is to be read.
section	the data location where the postfilter's current section configuration data will be written.

Sample Code

```
/* Test a section of a MPIFilter object's postfilter to see if it is a notch type.
   Sample usage:

   returnValue =
       isSectionTypeNotch(filter, 0, &isNotch);
*/
long isSectionTypeNotch(MPIFilter filter, long sectionIndex, long* isNotch)
{
    MPIFilterConfig config;
    MEIPostfilterSection section;
    long returnValue = (isNotch==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue =
            meiFilterPostfilterSectionGet(filter, sectionIndex, &section);
        if (returnValue == MPIMessageOK)
        {
            *isNotch = (section.type == MEIFilterTypeNOTCH) ? TRUE : FALSE;
        }
    }

    return returnValue;
}
```

Return Values	
MPIMessageOK	if <i>PostfilterSectionGet</i> successfully reads postfilter section information.
MPIFilterMessageCONVERSION_DIV_BY_0	Returned when meiFilterPostfilterSectionGet(...) cannot convert digital coefficients to analog coefficients. When this error occurs, the section will report its type as MEIFilterTypeUNKNOWN and will not contain any analog data.
MPIFilterMessageSECTION_NOT_ENABLED	Returned when no postfilter sections are enabled.
MPIFilterMessageINVALID_FILTER_FORM	Returned when meiFilterPostfilterSectionGet(...) cannot interpret the current postfilter's form (when the form is something other than NONE, IIR, or BIQUAD).

See Also [MEIPostfilterSection](#) | [meiFilterPostfilterGet](#) | [meFilterPostfilterSectionSet](#) | [MEIMaxBiQuadSections](#) | [Post Filter Theory](#) section.

meiFilterPostfilterSectionSet

Declaration

```
long meiFilterPostfilterSectionSet(MPIFilter filter,
                                   long      sectionNumber,
                                   MEIPostfilterSection *section);
```

Required Header stdmei.h

Description **PostfilterSectionSet** sets the configuration of a single section of an MPIFilter object's postfilter. It sets *filter*'s postfilter *sectionNumber*th section to the configuration specified in **section*. If the postfilter type is IIR, then this method is essentially equivalent to meiFilterPostfilterSet().

The MPI calculates the post filter coefficients taking into consideration the sample rate of the controller at that time. If you change the change the sample rate of the controller, you will need to recalculate your post filters. This can be done for all filters specified in Hertz by setting the filters again using the MPI. The MPI will calculate the filters using the current servo sample rate.

Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

filter	the handle of the MPIFilter object whose postfilter section configuration is to be set.
sectionNumber	the index of the postfilter section whose configuration is to be set.
*section	the data location of the section configuration to copy to the controller.

Sample Code

```
/* Set a section of a MPIFilter object's postfilter to a unity gain filter type.
Sample usage:

returnValue =
    setSectionTypeUnityGain(filter, 3);
*/
long setSectionTypeUnityGain(MPIFilter filter, long sectionIndex)
{
    MPIFilterConfig config;
    MEIPostfilterSection section;
    long returnValue;

    section.type = MEIFilterTypeUNITY_GAIN;
    section.form = MEIFilterFormBIQUAD;

    returnValue =
        meiFilterPostfilterSectionSet(filter, sectionIndex, &section);

    return returnValue;
}
```

Return Values

MPIMessageOK if *PostfilterSectionSet* successfully writes postfilter section information.

See Also [MEIPostfilterSection](#) | [meiFilterPostfilterSet](#) | [meFilterPostfilterSectionGet](#) | [MEIMaxBiQuadSections](#) |

[Post Filter Theory](#) section.

MPIFilterCoeff

MPIFilterCoeff

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

Description

f	float coefficient
l	long coefficient

See Also [MPIFilterCoeffCOUNT_MAX](#) | [MEIFilterGainPIDCoeff](#) | [MEIFilterGainPIVCoeff](#)

MPIFilterConfig / MEIFilterConfig

MPIFilterConfig

```
typedef struct MPIFilterConfig {
    long          gainIndex;
    MPIFilterGain gain[MPIFilterGainCOUNT\_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
axisMap	see MPIObjectMap
motorMap	see MPIObjectMap

MEIFilterConfig

```
typedef struct MEIFilterConfig {
    MEIXmpAlgorithm      Algorithm;

    MEIXmpAxisInput      Axis[MEIXmpFilterAxisInputs];

    long                 *VelPositionPtr;

    MEIXmpSwitchType     GainSwitchType;
    float                GainDelay;
    long                 GainWindow;
    MEIXmpSwitchType     PPISwitchType;
    MEIXmpPPIMode        PPIMode;
    float                PPIDelay;
    long                 PPIWindow;
    MEIXmpIntResetConfig ResetIntegratorConfig;
    float                ResetIntegratorDelay;

    MEIXmpFilterForm     PostFilterForm;
    MEIXmpPostFilter     PostFilter;
} MEIFilterConfig;
```

Description

MEIFilterConfig contains configuration information specific to a controller. With the exception of the Algorithm element, MEIFilterConfig contains configuration information that are more intuitively accessed by other means (Postfilter parameter) or information for advanced setups and custom controller firmware.

Algorithm	This value defines the algorithm that the filter is executing every servo cycle. The most common values are: <table> <tr> <td>MEIXmpAlgorithmPID</td><td>PID algorithm</td></tr> <tr> <td>MEIXmpAlgorithmPIV</td><td>PIV algorithm</td></tr> <tr> <td>MEIXmpAlgorithmNONE</td><td>No control algorithm</td></tr> </table>	MEIXmpAlgorithmPID	PID algorithm	MEIXmpAlgorithmPIV	PIV algorithm	MEIXmpAlgorithmNONE	No control algorithm
MEIXmpAlgorithmPID	PID algorithm						
MEIXmpAlgorithmPIV	PIV algorithm						
MEIXmpAlgorithmNONE	No control algorithm						
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.						
PostFilterForm	This value defines the form for postfilters when they are configured using mpiFilterConfigGet/Set(). Supported values are: • MEIXmpFilterFormIIR, IIR Filter • MEIXmpFilterFormBIQ, Bi-Quad Filter • MEIXmpFilterFormSS_BIQ, State Space form of Bi-Quad Filter						

	• MEIXmpFilterFormINT_BIQ, Integer (64-bit) Bi-Quad Filter • MEIXmpFilterFormINT_SS_BIQ, Integer State Space form of Bi-Quad Filter Though the postfilter may be configured through this parameter, it is strongly recommended that users use the <code>meiFilterPostfilter...()</code> methods instead for a more intuitive and user-friendly interface.
PostFilter	This array defines the configuration for the filter's postfilter (the type, the length and values for the post filter coefficients). Though the postfilter may be configured through this parameter, it is strongly recommended that users use the <code>meiFilterPostfilter...()</code> methods instead for a more intuitive interface. Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

Sample Code

```

/*  Test whether an MPIFilter object's control loop algorithm is PID.
    Sample usage:

    returnValue =
        isAlgorithmPid(filter, &isPid);
*/

long isAlgorithmPid(MPIFilter filter, long* isPid)
{
    MEIFilterConfig xmpConfig;
    long returnValue = (isPid==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue =
            mpiFilterConfigGet(filter, NULL, &xmpConfig);
        if (returnValue == MPIMessageOK)
        {
            *isPid = (xmpConfig.Algorithm == MEIXmpAlgorithmPID) ? TRUE : FALSE;
        }
    }

    return returnValue;
}

```

See Also [mpiFilterConfigGet](#) | [mpiFilterConfigSet](#) | [meiFilterPostfilterGet](#) | [meiFilterPostfilterSet](#) | [meiFilterPostfilterSectionGet](#) | [meiFilterPostfilterSectionSet](#) |

MEIFilterForm

MEIFilterForm

```
typedef enum{
    MEIFilterFormIIR,
    MEIFilterFormBIQUAD,
    MEIFilterFormSS_BIQUAD,
    MEIFilterFormINT_BIQUAD,
    MEIFilterFormINT_SS_BIQUAD,
} MEIFilterForm;
```

Description

FilterForm describes the form that a digital filter takes on the controller. Please note that the equations listed below use the coefficients loaded onto the controller, not necessarily the coefficients used by the MPI. A user may specify a low pass filter with only a single parameter (the breakpoint) and request that the form of the filter be a space-state biquad form on the controller.

Digital filtering on the XMP is accomplished through 32-bit words. This equates to the use of single precision floating point numbers - a 24-bit mantissa or about 7 decimal places of accuracy. This lack of precision can cause errors in the filtering process normally appearing as DC gain shifts or limit cycling, this especially true when the filter requires more than one section, a 6th order low pass filter would be one example. Filter forms using integer math can provide more internal precision for coefficients and internal registers but at the cost of less dynamic range. Filter forms using integer math take more processing time for the controller and can potentially limit the maximum sample rate of the controller.

The state-space (SS) filter forms allow the scaling of the input and the output, whereas the non-state-space forms only allow output scaling. This helps to prevent the loss of precision of the internal registers while still maintaining a very large dynamic range. Filter forms using state-space forms take more processing time for the controller and can potentially limit the maximum sample rate of the controller. However, a non-integer state-space filter form takes less processing power than an integer non-state-space filter form.

MEIFilterFormIIR

Deprecated. Cascaded biquad sections offer better precision and better calculation performance.

MEIFilterFormBIQUAD	Second Order digital filter form, for implementing low/high pass, notch, lead/lag and custom filters. The filter is a single precision floating point canonical form. The biquad filter is defined by the following discrete transfer function: The XMP's representation of this filter is: w0: Intermediate result u(k): filter input a1, a2, b0, b1, and b2: discrete biquad coefficients y(k):filter output x1k and x2k: filter states
MEIFilterFormSS_BIQUAD	Second order digital filter form, for implementing low/high pass, notch, lead/lag and custom filters. The filter is a single precision, floating point state space implementation. This filter applies input and output scaling to the canonical form. The XMP's state space representation of this filter is: u(k): filter input d1, c1, c2, a2, a1,b1: discrete biquad coefficients y(k):filter output p1k and p2k: filter states
MEIFilterFormINT_BIQUAD	Second Order digital filter form, for implementing low/high pass, notch, lead/lag and custom filters. The filter is a fixed point canonical form state space implementation. This form is a fixed point implementation of the floating point form MEIFilterFormBIQUAD. See the definition of MEIFilterFormBIQUAD above for the defining equations for this filter. The input coefficients for this filter (b0, b1, b2, a1 and a2) should all be greater than -2, and less than 2. The coefficients are represented as 32 bit 2's complement, with $1=2^{30}$. The coefficient's numerical format is 1.29 (1 bit whole, 29 bits fractional), and the controller uses an 80 bit accumulator. Only the 32 bit result of the multiplication is output from each section.

MEIFilterFormINT_SS_BIQUAD

Second Order digital filter form, for implementing low/high pass, notch, lead/lag and custom filters. The filter is a fixed point canonical form state space implementation. This form is a fixed point implementation of the floating point form MEIFilterFormSS_BIQUAD. See the definition of MEIFilterFormSS_BIQUAD above for the defining equations for this filter.

The input coefficients for this filter (d1, c1, c2, a2, a1 and b1) should all be greater than -2, and less than 2. The coefficients are represented as 32 bit 2's complement, with $1=2^{30}$. The coefficient's numerical format is 1.29 (1 bit whole, 29 bits fractional), and the controller uses an 80 bit accumulator. Only the 32 bit result of the multiplication is output from each section.

See Also [MEIPostfilterSection](#)

MPIFilterGain

MPIFilterGain

```
typedef struct MPIFilterGain {  
    MPIFilterCoeff    coeff[MPIFilterCoeffCOUNT_MAX];  
} MPIFilterGain;
```

Description

coeff	see MPIFilterCoeff

See Also [MPIFilterGainCOUNT_MAX](#) | [MEIFilterGainPIDCoeff](#) | [MEIFilterGainPIVCoeff](#)

MEIFilterGainIndex

MEIFilterGainIndex

```
typedef enum {

    /* Gain table index for normal firmware. */
    MEIFilterGainIndexNO_MOTION = MEIXmpGainNOT_MOVING,
    MEIFilterGainIndexACCEL      = MEIXmpGainACCEL,
    MEIFilterGainIndexDECEL      = MEIXmpGainDECEL,
    MEIFilterGainIndexVELOCITY   = MEIXmpGainCONSTANT_VEL,

    /* Gain table index for Custom 1 firmware. */
    MEIFilterGainIndexSTOPPING2 = MEIXmpGainSTOPPED2,
    MEIFilterGainIndexSTOPPING1 = MEIXmpGainSTOPPED1,
    MEIFilterGainIndexSETTLING   = MEIXmpGainSETTLING,
    MEIFilterGainIndexMOVING     = MEIXmpGainMOVING,
    MEIFilterGainIndexSTOPPING3 = MEIXmpGainSTOPPED3,

    /* Gain table index for Custom 5 firmware. */
    MEIFilterGainIndexMIN        = MEIXmpGainMIN,
    MEIFilterGainIndexMAX        = MEIXmpGainMAX,
    MEIFilterGainIndexNONE       = MEIXmpGainNONE,
    MEIFilterGainIndexSLOPE      = MEIXmpGainSLOPE,

    MEIFilterGainIndexLAST       = MEIXmpGainLAST,
    MEIFilterGainIndexALL        = MEIFilterGainIndexLAST,
                                /* used for gain get/set() */
    MEIFilterGainIndexFIRST      = MEIFilterGainIndexINVALID + 1,

    MEIFilterGainIndexDEFAULT    = MEIFilterGainIndexNO_MOTION,
} MEIFilterGainIndex;
```

Description

FilterGainIndex is an enumeration for the gain index used in gain scheduling.

In standard firmware, only

MEIFilterGainIndexNO_MOTION,
MEIFilterGainIndexACCEL,
MEIFilterGainIndexDECEL, and
MEIFilterGainIndexVELOCITY

are used. The gain index that is currently used can be found with [mpiFilterGainIndexGet\(...\)](#).

Gain Scheduling is a feature that switches filter gains for the acceleration, deceleration, constant velocity, and idle states of motion. The post filters are not affected by gain scheduling. Standard algorithms are used with gain scheduling (PID, PIV). To change the gain scheduling type from NONE (uses only the gains in gain table index 0), use [MEIFilterConfig](#). GainSwitchType is set with [mpiFilterConfigSet\(...\)](#).

When setting filter gain parameters using [mpiFilterGainGet\(...\)](#) and [mpiFilterGainSet\(...\)](#), use the gain index value to write to a gain index of your choosing.

MEIFilterGainIndexNO_MOTION	No commanded motion. Trajectory parameters Velocity, Acceleration, and Jerk equal zero.
MEIFilterGainIndexACCEL	Acceleration portion of the commanded move.
MEIFilterGainIndexDECEL	Deceleration portion of the commanded move.
MEIFilterGainIndexVELOCITY	Constant velocity portion of the commanded move. Gain switching is configured by setting the GainSwitichType, GainDelay, and GainWindow in the MEIFilterConfig{...} structure and calling mpiFilterConfigGet/Set(...). The GainSwitchType has the following options:

See Also [MEIFilterConfig](#) | [mpiFilterConfigGet](#) | [mpiFilterConfigSet](#) | [MEIXmpSwitchType](#) | [mpiFilterGainIndexSet](#) | [mpiFilterGainIndexGet](#) | [mpiFilterGainGet](#) | [mpiFilterGainSet](#)

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](#)

MEIFilterGainTypePID

MEIFilterGainTypePID

```
static MEIDataType MEIFilterGainTypePID[MPIFilterCoeffCOUNT_MAX] =
{
    MEIDataTypeFLOAT, /* Kp                */ /*
    MEIDataTypeFLOAT, /* Ki                */ /*
    MEIDataTypeFLOAT, /* Kd                */ /*

    MEIDataTypeFLOAT, /* Kpff             */ /*
    MEIDataTypeFLOAT, /* Kvff             */ /*
    MEIDataTypeFLOAT, /* Kaff             */ /*
    MEIDataTypeFLOAT, /* Kfff             */ /*

    MEIDataTypeFLOAT, /* MovingIMax       */ /*
    MEIDataTypeFLOAT, /* RestIMax         */ /*

    MEIDataTypeLONG,  /* DRate            */ /*

    MEIDataTypeFLOAT, /* OutputLimit       */ /*
    MEIDataTypeFLOAT, /* OutputLimitHigh   */ /*
    MEIDataTypeFLOAT, /* OutputLimitLow    */ /*
    MEIDataTypeFLOAT, /* OutputOffset      */ /*
    MEIDataTypeFLOAT, /* Ka0               */ /*
    MEIDataTypeFLOAT, /* Ka1               */ /*
    MEIDataTypeFLOAT, /* Ka2               */ /*
};
```

Description

FilterGainTypePID is a static array that describes the data type of the coefficients for the PID algorithm. Specifically, an element of MEIFilterGainTypePID describes which member of the union MPIFilterCoeff to access when using the data structure MPIFilterCoeff.

MEIFilterGainTypePID allows for a more simple design of general case utilities and configuration routines. If it is known that only the PID parameters will be used, then the data structure MEIFilterGainPID can be used directly without having to manipulate MPIFilterCoeff, MPIFilterCoeff, and MEIFilterGainTypePID.

Sample Code

```

/*  Read the current value of a filter's PID coefficient.  Sample usage:

    returnValue =
        getPidFilterCoeff(filter, MEIFilterGainPIDCoeffGAIN_PROPORTIONAL, &kp);
*/
long getPidFilterCoeff(MPIFilter filter, long index, double* value)
{
    MPIFilterConfig config;
    long returnValue = (value==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue = mpiFilterConfigGet(filter, &config, NULL);

        if (returnValue == MPIMessageOK)
        {
            switch(MEIFilterGainTypePID[index])
            {
                case MEIDataTypeLONG:
                    *value = config.gain[config.gainIndex].coeff[index].l;
                    break;
                case MEIDataTypeFLOAT:
                    *value = config.gain[config.gainIndex].coeff[index].f;
                    break;
                default:
                    returnValue = MPIMessageARG_INVALID;
            }
        }
    }
    return returnValue;
}

```

See Also

[MPIFilterCoeff](#) |
 [MEIFilterGainTypePIV](#) |
 [MEIFilterGainPID](#) |
 [MEIDataType](#) |
 [MPIFilterGain](#) |

MEIFilterGainTypePIV

MEIFilterGainTypePIV

```
static MEIDataType MEIFilterGainTypePIV[MPIFilterCoeffCOUNT_MAX] =
{
    MEIDataTypeFLOAT, /* Kpp                */ /*
    MEIDataTypeFLOAT, /* Kip                */ /*

    MEIDataTypeFLOAT, /* Kpv                */ /*

    MEIDataTypeFLOAT, /* Kpff               */ /*
    MEIDataTypeFLOAT, /* Kvff               */ /*
    MEIDataTypeFLOAT, /* Kaff               */ /*
    MEIDataTypeFLOAT, /* Kfff               */ /*

    MEIDataTypeFLOAT, /* MovingIMax         */ /*
    MEIDataTypeFLOAT, /* RestIMax           */ /*

    MEIDataTypeFLOAT, /* Kdv                */ /*

    MEIDataTypeFLOAT, /* OutputLimit         */ /*
    MEIDataTypeFLOAT, /* OutputLimitHigh     */ /*
    MEIDataTypeFLOAT, /* OutputLimitLow      */ /*
    MEIDataTypeFLOAT, /* OutputOffset        */ /*

    MEIDataTypeFLOAT, /* Kiv                */ /*
    MEIDataTypeFLOAT, /* Vintmax             */ /*
    MEIDataTypeFLOAT, /* Ka0                 */ /*
    MEIDataTypeFLOAT, /* Ka1                 */ /*
};
```

Description

FilterGainTypePIV is a static array that describes the data type of the coefficients for the PIV algorithm. Specifically, an element of MEIFilterGainTypePIV describes which member of the union MPIFilterCoeff to access when using the data structure MPIFilterCoeff.

MEIFilterGainTypePIV allows for a more simple design of general case utilities and configuration routines. If it is known that only the PIV parameters will be used, then the data structure MEIFilterGainPIV can be used directly without having to manipulate MPIFilterCoeff, MPIFilterCoeff, and MEIFilterGainTypePIV.

Sample Code

```

/*  Read the current value of a filter's PIV coefficient.  Sample usage:
    returnValue =
        getPivFilterCoeff(filter, MEIFilterGainPIVCoeffGAINVELOCITY_PROPORTIONAL,
&kpv);
*/
long getPivFilterCoeff(MPIFilter filter, long index, double* value)
{
    MPIFilterConfig config;
    long returnValue = (value==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue = mpiFilterConfigGet(filter, &config, NULL);

        if (returnValue == MPIMessageOK)
        {
            switch(MEIFilterGainTypePIV[index])
            {
                case MEIDataTypeLONG:
                    *value = config.gain[config.gainIndex].coeff[index].l;
                    break;
                case MEIDataTypeFLOAT:
                    *value = config.gain[config.gainIndex].coeff[index].f;
                    break;
                default:
                    returnValue = MPIMessageARG_INVALID;
            }
        }
    }

    return returnValue;
}

```

See Also [MPIFilterCoeff](#) | [MEIFilterGainTypePID](#) | [MEIFilterGainPIV](#) | [MEIDataType](#) | [MPIFilterGain](#) |

MPIFilterMessage

MPIFilterMessage

```
typedef enum {
    MPIFilterMessageFILTER_INVALID,
    MPIFilterMessageINVALID_ALGORITHM,
    MPIFilterMessageINVALID_DRATE,
    MPIFilterMessageCONVERSION_DIV_BY_0,
    MPIFilterMessagePOSTFILTER_NOT_ENABLED,
    MPIFilterMessageINVALID_FILTER_FORM,
} MPIFilterMessage;
```

Description

MPIFilterMessageFILTER_INVALID

The filter number is out of range. This message code is returned by [mpiFilterCreate\(...\)](#) if the filter number is less than zero or greater than or equal to MEIXmpMAX_Filters.

MPIFilterMessageINVALID_ALGORITHM

The filter algorithm is not valid. This message code is returned by [mpiFilterIntegratorReset\(...\)](#) if the filter algorithm is not a member of the MEIXmpAlgorithm enumeration (does not support integrators). This problem occurs if the filter type is set to user or an unknown type with [mpiFilterConfigSet\(...\)](#).

MPIFilterMessageINVALID_DRATE

The filter derivative rate is not valid. This message code is returned by [mpiFilterConfigSet\(...\)](#) if the filter derivative rate is less than 0 or greater than 7.

NOTE: The derivative rate for all gain tables must be in the range [0,7], not just the derivative rate for the current gain table.

MPIFilterMessageCONVERSION_DIV_BY_0

Returned when [meiFilterPostfilterGet\(...\)](#) or [meiFilterPostfilterSectionGet\(...\)](#) cannot convert digital coefficients to analog coefficients. When this error occurs, the offending section(s) will report its type as MEIFilterTypeUNKNOWN and will not contain any analog data.

MPIFilterMessagePOSTFILTER_NOT_ENABLED

Returned when [meiFilterPostfilterGet\(...\)](#) or [meiFilterPostfilterSectionGet\(...\)](#) attempt to read postfilter data when no postfilter sections are enabled.

MPIFilterMessageINVALID_FILTER_FORM

Returned when [meiFilterPostfilterGet\(...\)](#) or [meiFilterPostfilterSectionGet\(...\)](#) cannot interpret the current postfilter's form (when the form is something other than NONE, IIR, or BIQUAD).

See Also [mpiFilterCreate](#)

MEIFilterType

MEIFilterType

```
typedef enum {
    MEIFilterTypeUNITY_GAIN,
        /* B0 = 1      B1=B2=A1=A2 = 0  (effectively acting as no filter) */
    MEIFilterTypeSINGLE_ORDER,
    MEIFilterTypeLOW_PASS,
    MEIFilterTypeHIGH_PASS,
    MEIFilterTypeNOTCH,
    MEIFilterTypeRESONATOR,
    MEIFilterTypeLEAD_LAG,
    MEIFilterTypeZERO_GAIN,
        /* b0=b1=b2=a1=a2 = 0  (this does act as a filter.... zeroing the output) */
    MEIFilterTypeBIQUAD,
        /* Only valid for setting.  Reading will not return these types */
    MEIFilterTypeDIGITAL_BIQUAD,
    MEIFilterTypePOLES_ZEROS,
    MEIFilterTypeDIGITAL_POLES_ZEROS,
    MEIFilterTypeUNKNOWN,
        /* algorithm couldn't figure out what this filter was from the coeffs! */
} MEIFilterType;
```

Description

NOTE: The MPI will attempt to return analog & digital biquad and pole/zero information from [meiFilterPostfilterGet\(...\)](#) and [meiFilterPostfilterSectionGet\(...\)](#). However, the filter types MEIFilterTypeDIGITAL_BIQUAD, MEIFilterTypePOLES_ZEROS, and MEIFilterTypeDIGITAL_POLES_ZEROS are never returned by get() calls -- they are used only for setting postfilters. MEIFilterTypeBIQUAD will only be returned by meiFilterPostfilterGet(...) and meiFilterPostfilterSectionGet(...) if the analog coefficients can be calculated (there is no division by 0) and the section cannot be identified as one of the other analog filter types.

MEIFilterTypeUNITY_GAIN	A unity gain filter. This effectively performs no filtering.
MEIFilterTypeSINGLE_ORDER	A single order filter
MEIFilterTypeLOW_PASS	A low pass filter
MEIFilterType_HIGH_PASS	A high pass filter.
MEIFilterTypeNOTCH	A notch filter
MEIFilterTypeRESONATOR	A resonator filter.
MEIFilterTypeLEAD_LAG	A lead or lag filter.
MEIFilterTypeZERO_GAIN	Zeros the output of a filter.
MEIFilterTypeBIQUAD	An analog biquad filter. When reading postfilter data, this type means that the postfilter section could not be identified as a standard filter type.
MEIFilterTypeDIGITAL_BIQUAD	A digital biquad filter. This is only used for setting postfilter sections.

MEIFilterTypePOLES_ZERO	Analog poles and zeros filter (maximum of two poles and zeros) with unity zero-frequency amplitude. This is only used for setting postfilter sections.
MEIFilterTypeDIGITAL_POLES_ZEROS	Digital poles and zeros filter (maximum of two poles and zeros) with unity zero-frequency amplitude. This is only used for setting postfilter sections.
MEIFilterTypeUNKNOWN	Returned by <code>meiFilterPostfilterGet(...)</code> and <code>meiFilterPostfilterSectionGet(...)</code> if analog coefficients cannot be found. only digital data will be available.

See Also

[MEIPostfilterSection](#) | [meiFilterPosterfilterGet](#) | [meiFilterPosterfilterSet](#) | [meiFilterPosterfilterSectionGet](#) | [meiFilterPosterfilterSectionSet](#)

MEIPostfilterSection

MEIPostfilterSection

```
typedef struct MEIPostfilterSection {
    MEIFilterType    type;
    MEIFilterForm    form;
    struct {
        struct {
            double breakPoint;      /* Hz */
        } lowPass;

        struct {
            double breakPoint;      /* Hz */
        } highPass;

        struct {
            double centerFrequency; /* Hz */
            double bandwidth;      /* Hz */
        } notch;

        struct {
            double centerFrequency; /* Hz */
            double bandwidth;      /* Hz */
            double gain;           /* dB */
        } resonator;

        struct {
            double lowFrequencyGain; /* dB */
            double highFrequencyGain; /* dB */
            double centerFrequency; /* Hz */
        } leadLag;

        struct {
            double a1;
            double a2;
            double b0;
            double b1;
            double b2;
        } biquad;

        struct {
            double a1;
            double a2;
            double b0;
            double b1;
            double b2;
        } digitalBiquad;

        struct {
            long poleCount;
            long zeroCount;
            struct {
                double real;
                double imag;
            } pole[2];
        }
    }
};
```

```

        struct {
            double  real;
            double  imag;
        } zero[2];
    } polesZeros;

    struct {
        long poleCount;
        long zeroCount;
        struct {
            double real;
            double imag;
        } pole[2];
        struct {
            double real;
            double imag;
        } zero[2];
    } digitalPolesZeros;

    struct {
        double d1;
        double c1;
        double c2;
        double a2;
        double a1;
        double b1;
    } stateSpaceBiquad;

    struct {
        long    numberOfCoefficients;
        double  coeff[MEIXmpMAX_PostFilterSize];
    } iir;
} data;
} MEIPostfilterSection;

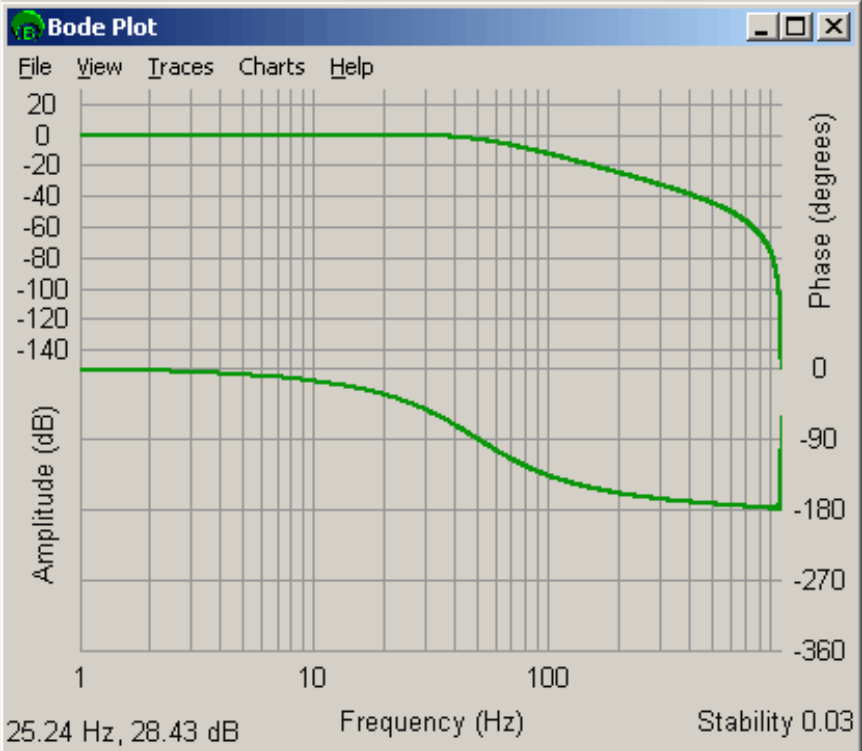
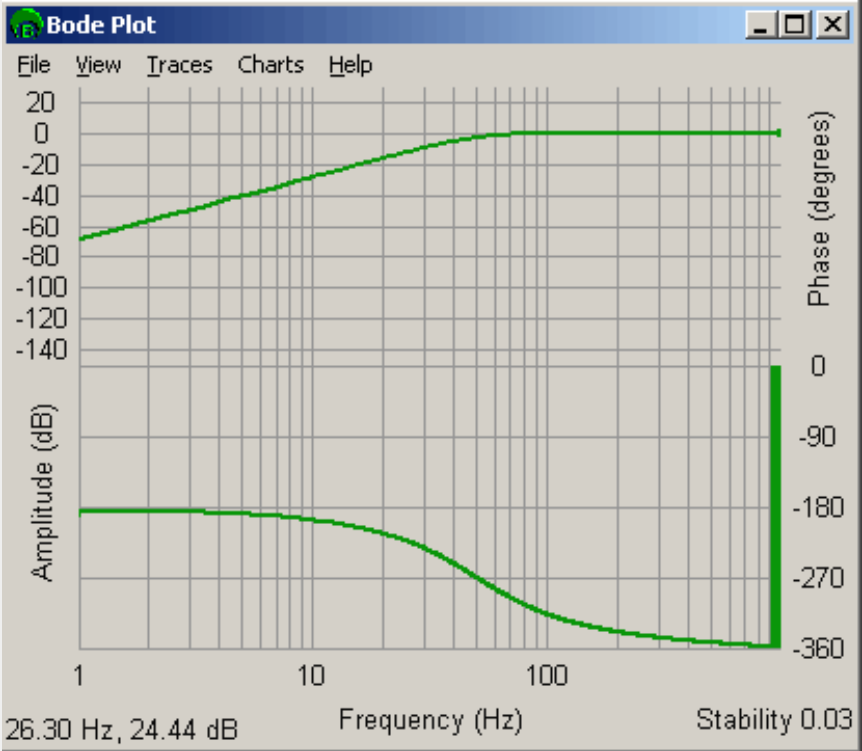
```

Description

MEIPostfilterSection holds the configuration data for a single section of an MPIFilter object's postfilter. The MPI calculates the post filter coefficients and takes into consideration the sample rate of the controller at that time. If you change the sample rate of the controller, you will need to recalculate the post filters. This can be done for all filters specified in Hertz by setting the filters again with the MPI. The MPI will calculate the filters using the current servo sample rate.

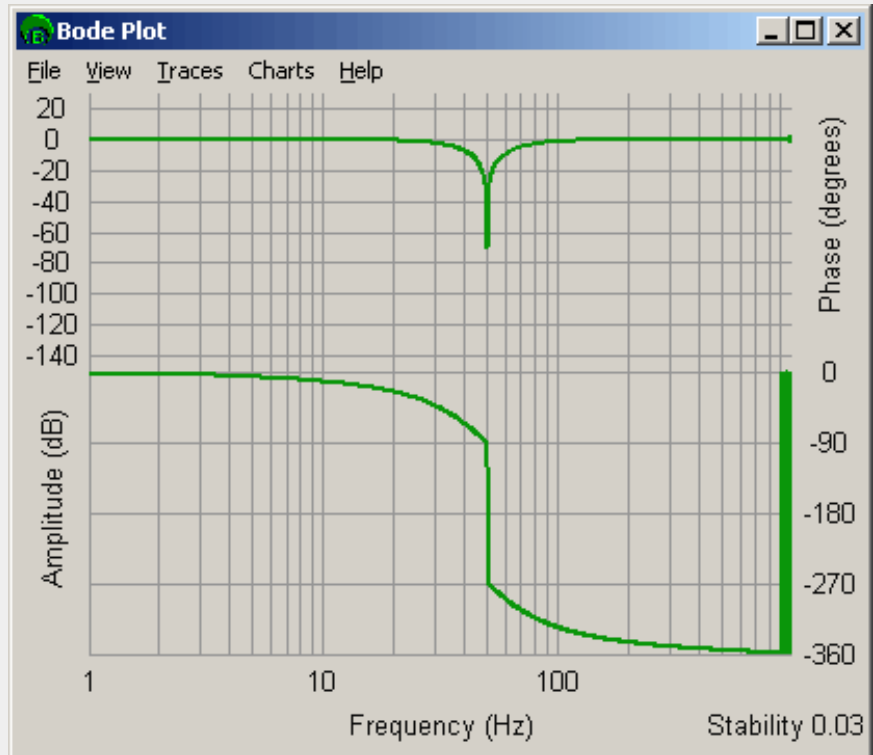
Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

type	The postfilter section type. This field determines which field of the MEIPostfilterSection.data union is used by meiFilterPostfilter...() methods. More information about particular filter types can be found below and in the MEIFilterType documentation.

form	The form of a postfilter section. The form determines how a particular postfilter section is calculated on the controller. All forms have certain limitations and tradeoffs. Please refer to MEIFilterForm for more information.
lowPass.breakpoint	The break point (measured in Hertz) of a low pass postfilter section.  Example of a 50 Hz low pass filter.
highPass.breakpoint	The break point (measured in Hertz) of a high pass postfilter section.  Example of a 50 Hz High pass filter

notch.centerFrequency

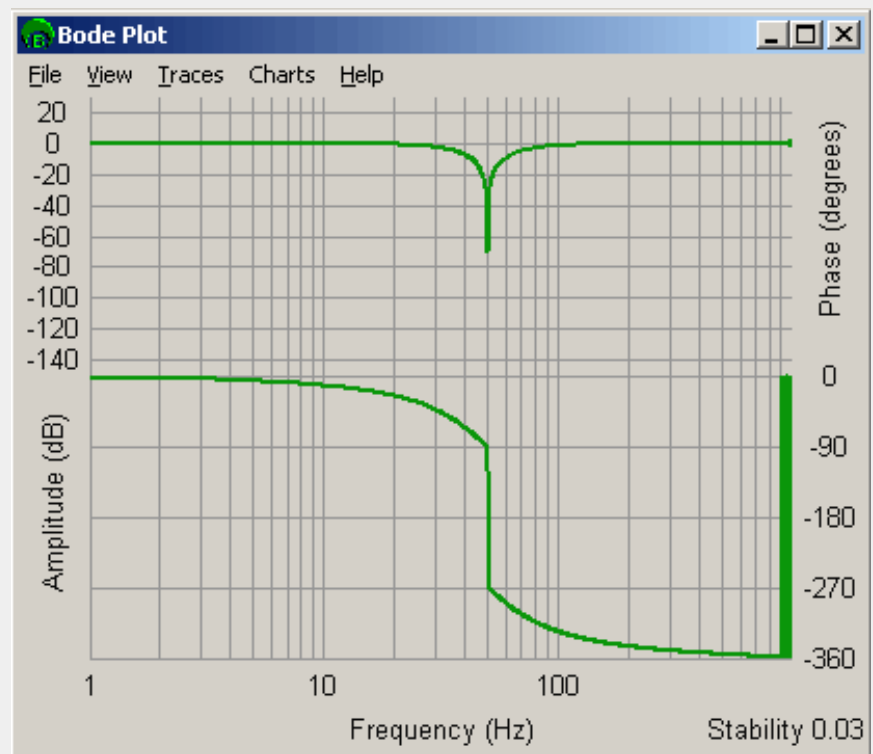
The bandwidth (measured in Hertz) of a notch postfilter section.



Example of a 50 Hz Center / 50 Hz Bandwidth Notch filter. Note that phase wrapping gives the illusion that phase drops 180 degrees after the center frequency. The phase raises by 180 degrees.

notch.bandwidth

The center frequency (measured in Hertz) of a notch postfilter section.

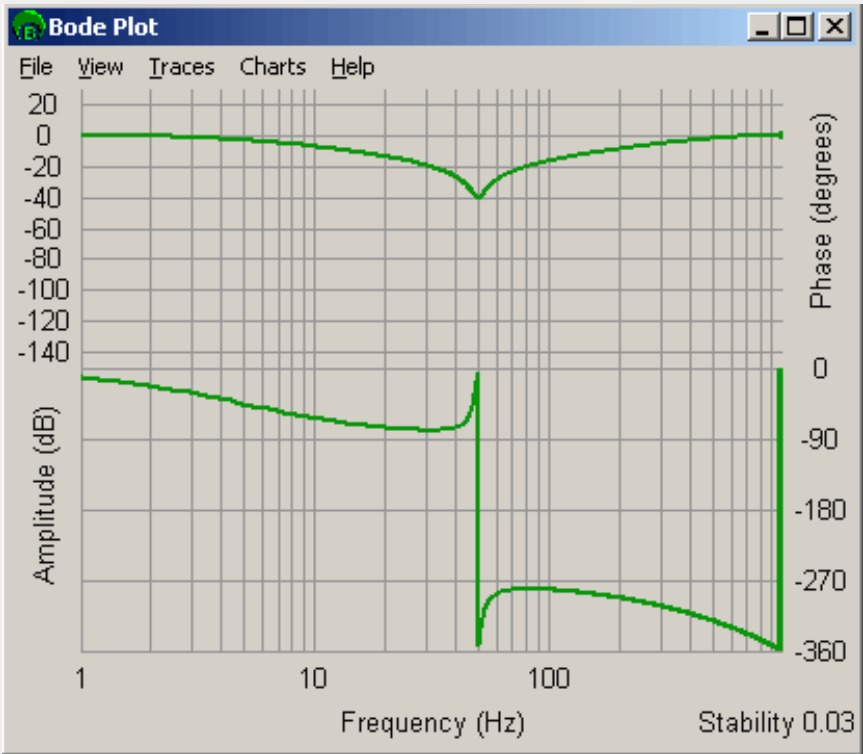


Example of a 50 Hz Center / 50 Hz Bandwidth Notch filter. Note that phase wrapping

gives the illusion that phase drops 180 degrees after the center frequency. The phase raises by 180 degrees.

resonator.centerFrequency

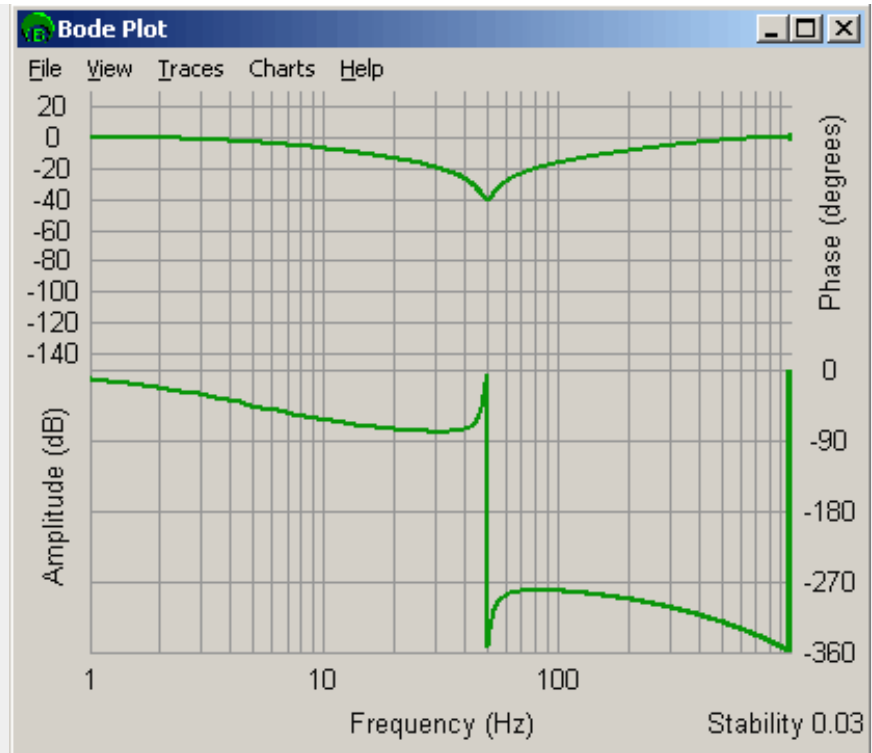
The center frequency (measured in Hertz) of a resonator postfilter section.



Example of a 50 Hz center / 50 Hz Bandwidth / -40 dB Gain Resonator filter. Note that phase wrapping gives the illusion that the phase drops 360 degrees after the center frequency.

resonator.bandwidth

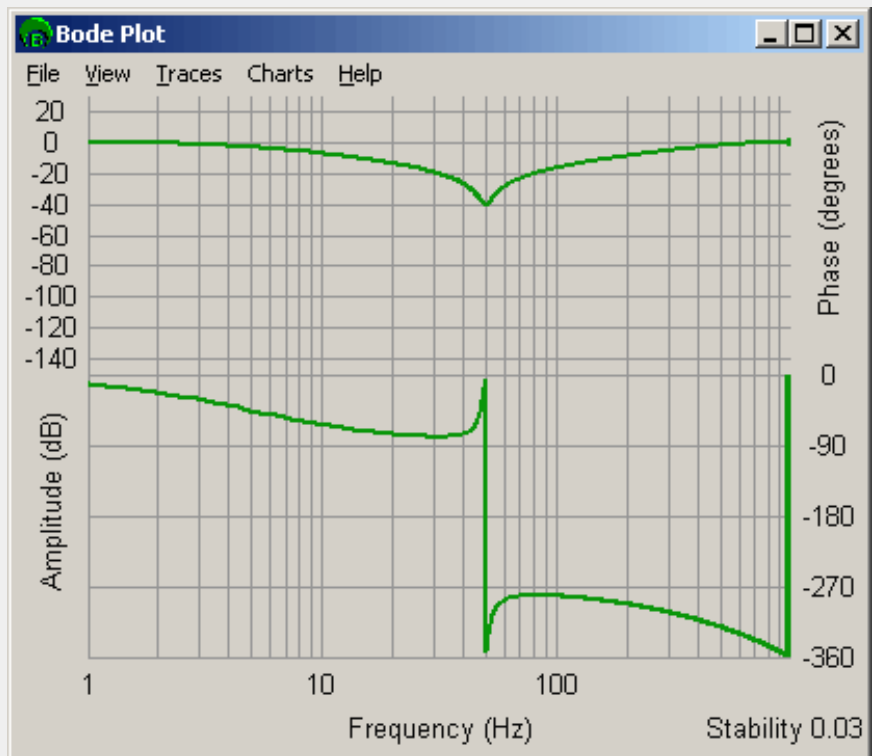
The bandwidth (measured in Hertz) of a resonator postfilter section.



Example of a 50 Hz center / 50 Hz Bandwidth / -40 dB Gain Resonator filter. Note that phase wrapping gives the illusion that the phase drops 360 degrees after the center frequency.

resonator.gain

The center frequency gain (measured in dB) of a resonator postfilter section.

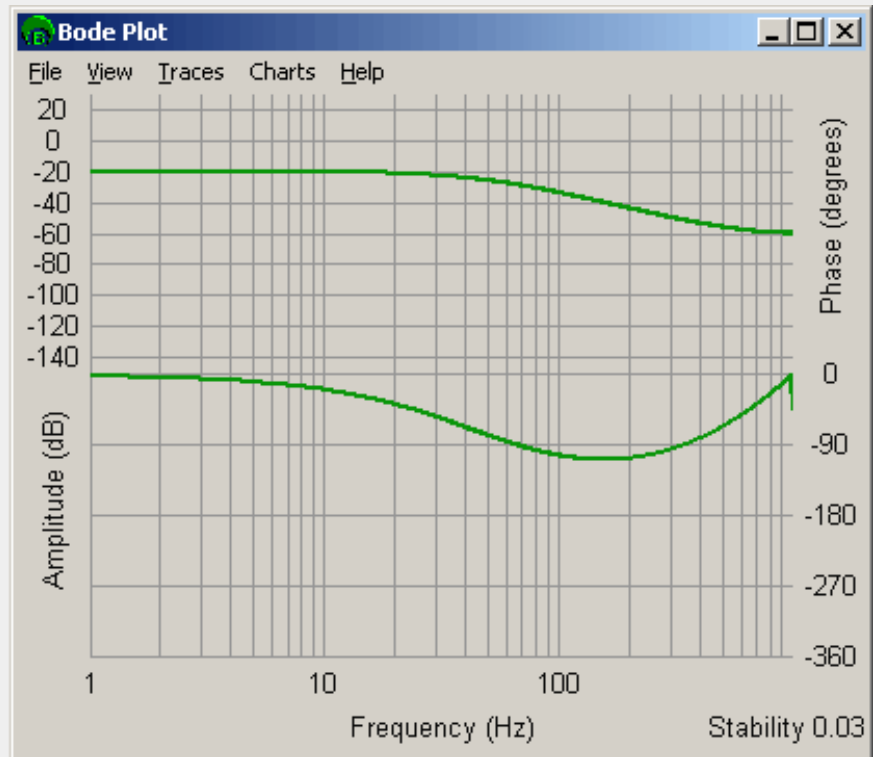


Example of a 50 Hz center / 50 Hz Bandwidth / -40 dB Gain Resonator filter. Note that phase wrapping gives the illusion that the phase drops 360 degrees after the center frequency.

leadLad.centerFrequency

The center frequency (measured in Hertz) of a lead or lag postfilter section. The amplitude at this frequency is the average amplitude of the low and high frequency amplitudes. The gain (measured in dB) at this point is given by:

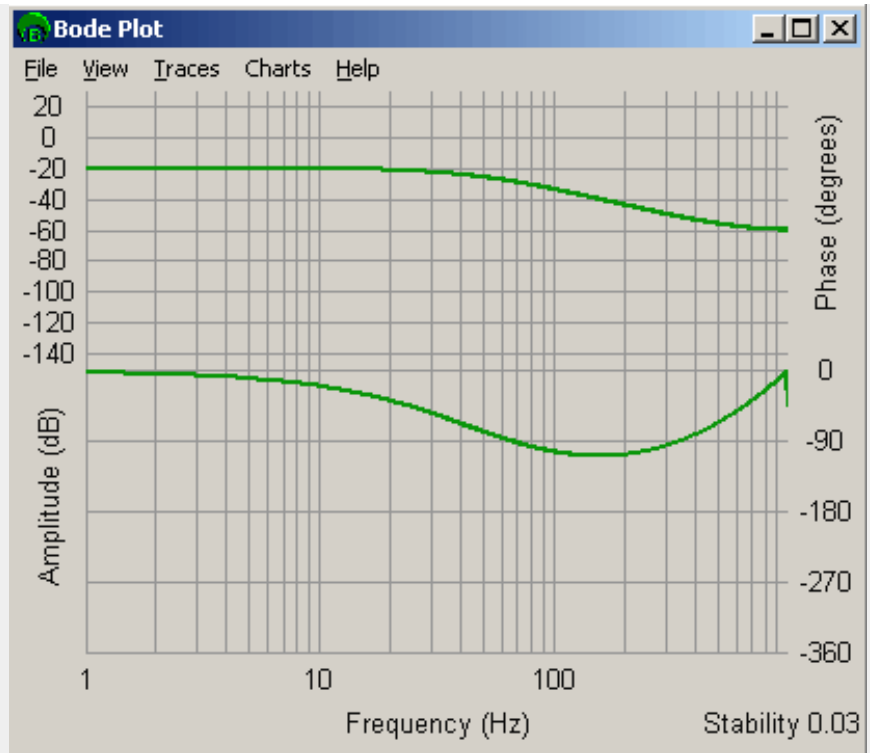
$$centerFrequencyGain = 20 \cdot \log_{10} \left(\frac{10^{\frac{lowFrequencyGain}{20}} + 10^{\frac{highFrequencyGain}{20}}}{2} \right)$$



Example of a -20 dB low frequency gain / -60 dB high frequency gain / 50 Hz center lead lag filter.

leadLag.lowFrequencyGain

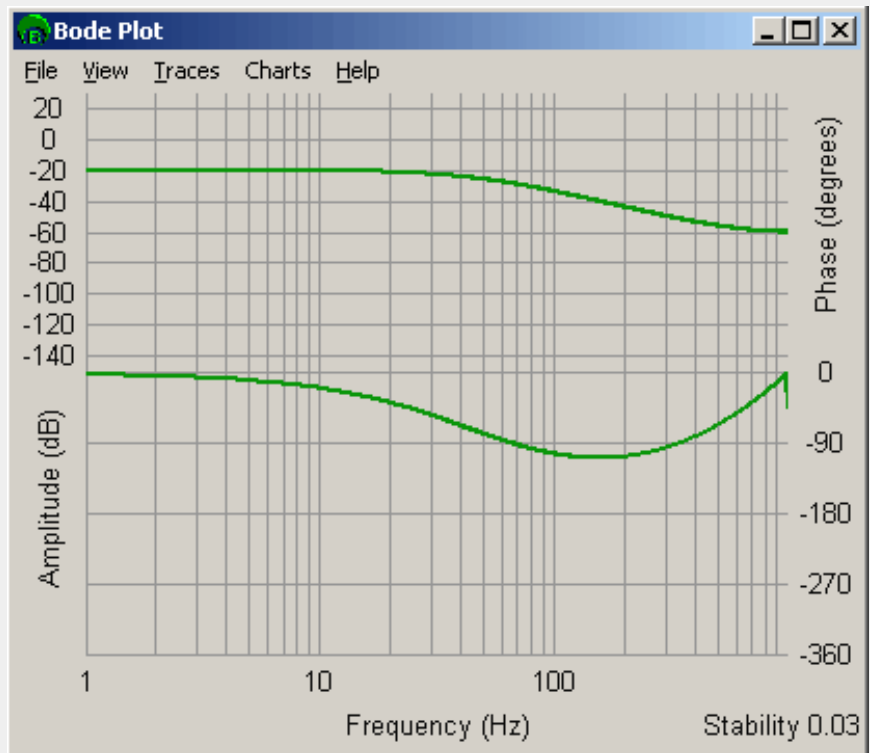
The low frequency gain (measured in dB) of a lead or lag postfilter section.



Example of a -20 dB low frequency gain / -60 dB high frequency gain / 50 Hz center lead lag filter.

leadLag.highFrequencyGain

The high frequency gain (measured in dB) of a lead or lag postfilter section.



Example of a -20 dB low frequency gain / -60 dB high frequency gain / 50 Hz center lead lag filter.

biquad.a1	The analog coefficients of a single order or bi-quad postfilter section. Analog values of the postfilter coefficients are produced as parts of a Laplace Transform: $H(s) = \frac{b_0 + b_1 \cdot s + b_2 \cdot s^2}{1 + a_1 \cdot s + a_2 \cdot s^2} \text{ where } s = j \cdot \omega_{warped}$ and $\omega_{warped} = 2 \cdot sampleRate \cdot \tan\left(\frac{\pi \cdot f}{sampleRate}\right)$
biquad.a2	
biquad.b0	
biquad.b1	
biquad.b2	
digitalBiquad.a1	The digital coefficients of a single order or bi-quad postfilter section.
digitalBiquad.a2	
digitalBiquad.b0	
digitalBiquad.b1	
digitalBiquad.b2	
digitalBiquad.d1	The digital coefficients of a state-space bi-quad postfilter section.
digitalBiquad.c1	
digitalBiquad.c2	
digitalBiquad.a2	
digitalBiquad.a1	
digitalBiquad.b1	Analog poles and zeros.
polesZeros.poleCount	
polesZeros.zeroCount	
polesZeros.pole[].real	
polesZeros.pole[].imag	
digitalPolesZeros.poleCount	Digital poles and zeros.
digitalPolesZeros.zeroCount	
digitalPolesZeros.pole[].real	
digitalpolesZeros.pole[].imag	
stateSpaceBiquad.d1	
stateSpaceBiquad.c1	State space coefficients.
stateSpaceBiquad.c2	
stateSpaceBiquad.a2	
stateSpaceBiquad.a1	
stateSpaceBiquad.b1	
iir.numberOfCoefficients	Currently not supported. Reserved for future use.
iir.coeff	

Sample Code

```

/*  Set a 4th order low-pass post-filter by using two 2nd order low-pass sections.
    Sample usage:

    returnValue =
        fourthOrderLowPass(filter, 300 /* Hz */);
*/
long filterFourthOrderLowpass(MPIFilter filter, long breakPointFrequency)
{
    MPIFilterConfig config;
    MEIPostfilterSection sections[2];
    long returnValue;

    section[0].type = MEIFilterTypeLOW_PASS;
    section[0].form = MEIFilterFormINT_BIQUAD;
    section[0].lowPass.breakpoint = breakPointFrequency;
    section[1] = section[0]; /* copy first section */

    returnValue =
        meiFilterPostfilterSet(filter, 2, sections);

    return returnValue;
}

```

See Also

[MEIFilterType](#) | [MEIFilterForm](#) | [MEIMaxIIRCoefficients](#) | [meiFilterPostfilterGet](#) | [meiFilterPostfilterSet](#) | [meiFilterPostfilterSectionGet](#) | [meiFilterPostfilterSectionSet](#) |

[Post Filter Theory](#) section.

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](#)

MEIMaxBiQuadSections

MEIMaxBiQuadSections

```
#define MEIMaxBiQuadSections (6)
```

Description

MEIMaxBiQuadSections is the maximum number of Bi-Quad sections a postfilter can use.

NOTE: The PIV algorithm uses the last Bi-Quad section internally. Thus a user can only use (MEIMaxBiQuadSections - 1) Bi-quad sections with the PIV algorithm.

See Also

[MEIPostFilterSection](#) | [meiFilterPostfilterGet](#) | [meiFilterPostfilterSet](#) | [meiFilterPostfilterSectionGet](#) | [meiFilterPostfilterSectionSet](#)

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](#)

Post Filter Theory

[Laplacian Space](#) | [Z Space](#) | [Z Transform Stability](#)

Laplacian Space

Analog values of the postfilter coefficients are produced as parts of a Laplace Transform:

$$H(s) = \frac{b_0 + b_1 \cdot s^{-1} + b_2 \cdot s^{-2}}{1 + a_1 \cdot s^{-1} + a_2 \cdot s^{-2}} \text{ where } s = j \cdot \omega_{warped}$$

and

$$\omega_{warped} = 2 \cdot sampleRate \cdot \tan\left(\frac{\pi \cdot f}{sampleRate}\right)$$

The amplitude and phase of the filter can be derived from the above by:

$$A^2 = \frac{(b_2 - b_0 \omega^2)^2 + (b_1 \omega)^2}{(a_2 - \omega^2)^2 + (a_1 \omega)^2}$$

$$phase = \arctan2\left(\left[(b_2 - b_0 \omega^2)(a_2 - \omega^2) + a_1 b_1 \omega^2\right], \omega[b_1(a_2 - \omega^2) - a_1(b_2 - b_0 \omega^2)]\right)$$

$\arctan2(x, y)$ is similar to $\arctan\left(\frac{y}{x}\right)$ except that the returned angle can be in the range from $-\pi$ to π .

From here we can calculate the gain (in dB) of the filter:

$$gain = 10 \cdot \log_{10} \left(\frac{(b_2 - b_0 \omega^2)^2 + (b_1 \omega)^2}{(a_2 - \omega^2)^2 + (a_1 \omega)^2} \right)$$

The filter types are designed as follows:

Low Pass	$\Omega = 2 \cdot \text{sampleRate} \cdot \tan\left(\frac{\pi \cdot f_{\text{breakPt}}}{\text{sampleRate}}\right)$ $a_1 = \sqrt{2} \cdot \Omega$ $a_2 = \Omega^2$ $b_0 = 0$ $b_1 = 0$ $b_2 = \Omega^2$
High Pass	$\Omega = 2 \cdot \text{sampleRate} \cdot \tan\left(\frac{\pi \cdot f_{\text{breakPt}}}{\text{sampleRate}}\right)$ $a_1 = \sqrt{2} \cdot \Omega$ $a_2 = \Omega^2$ $b_0 = -1$ $b_1 = 0$ $b_2 = 0$
Notch	$\Omega = 2 \cdot \text{sampleRate} \cdot \tan\left(\frac{\pi \cdot f_{\text{breakPt}}}{\text{sampleRate}}\right)$ $a_1 = 2 \cdot \pi \cdot \text{bandwidth} \cdot \left(1 + \tan^2\left(\frac{\pi \cdot f_{\text{center}}}{\text{sampleRate}}\right)\right)$ $a_2 = \Omega^2$ $b_0 = 1$ $b_1 = 0$ $b_2 = \Omega^2$
Resonator	$\Omega = 2 \cdot \text{sampleRate} \cdot \tan\left(\frac{\pi \cdot f_{\text{center}}}{\text{sampleRate}}\right)$ $a_1 = 2 \cdot \pi \cdot \text{bandwidth} \cdot \left(1 + \tan^2\left(\frac{\pi \cdot f_{\text{center}}}{\text{sampleRate}}\right)\right) / 10^{\left(\frac{\text{gain}}{40}\right)}$ $a_2 = \Omega^2$ $b_0 = 1$ $b_1 = a_1 \cdot 10^{\left(\frac{\text{gain}}{20}\right)}$ $b_2 = \Omega^2$

Lead, Lag	$\Omega = 2 \cdot \text{sampleRate} \cdot \tan\left(\frac{\pi \cdot f_{\text{center}}}{\text{sampleRate}}\right)$ $a_1 = 2 \cdot \Omega$ $a_2 = \Omega^2$ $b_0 = 10^{\left(\frac{\text{highFrequencyGain}}{20}\right)}$ $b_1 = 2 \cdot \Omega \cdot 10^{\left(\frac{\text{lowFrequencyGain} + \text{highFrequencyGain}}{40}\right)}$ $b_2 = \Omega^2 \cdot 10^{\left(\frac{\text{lowFrequencyGain}}{20}\right)}$

Additional Notes:

For the resonator filter, the maximum and minimum phase changes will occur at:

$$f = \frac{\text{SampleRate}}{\pi} \tan^{-1} \left(\frac{1}{2\sqrt{2} \cdot \text{SampleRate}} \sqrt{(a_1 b_1 - 2a_2) \pm \sqrt{a_1 b_1 (4a_2 + a_1 b_1)}} \right)$$

These frequencies also happen to be the half-gain points measured in dB (or the root-amplitude gain points).

For the lead/lag filters, the maximal phase change will occur at:

$$f = \frac{\text{SampleRate}}{\pi} \tan^{-1} \left(\frac{a_1}{4 \cdot \text{SampleRate}} \cdot 4 \sqrt{\frac{b_2}{a_2 b_0}} \right)$$

This frequency also happens to be the dB gain mean point measured (or the amplitude gain geometric mean point).

Z Space

Though Laplacian Space is useful for designing or quickly analyzing a bi-quad filter's design, it does not accurately model digital bi-quad filters. Digital filters are described naturally by Z transforms. It is possible to convert a filter from a Laplace transform to a Z

transform, as will be described below, while maintaining the same general characteristics. The amplitude and phase information will be slightly warped by moving into Z space. One should note, however, that for the filters listed above the characteristics of gains, bandwidths, and center or breakpoint frequencies are unchanged.

Bi-quad filters are described by the following Z transform:

$$H(Z) = \frac{B_0 + B_1 \cdot Z^{-1} + B_2 \cdot Z^{-2}}{1 + A_1 \cdot Z^{-1} + A_2 \cdot Z^{-2}} \quad \text{where } Z = e^{j \left(\frac{2 \cdot \pi \cdot f}{\text{sampleRate}} \right)}$$

One should note that only filters where the roots of the denominator lie within the unit circle are stable. Though digital filters can be constructed where the equations for amplitude and phase for both the Z transform version and the Laplace transform version may converge, the filter itself will be unstable, continually adding energy to the system. Please see the Z Transform Stability Section below.

The equations for amplitude, phase and dB gain can be derived from the above Z transform:

$$A^2 = \frac{(B_0^2 + B_1^2 + B_2^2) + 2 \cdot B_1 \cdot (B_0 + B_2) \cdot \cos\left(\frac{2 \cdot \pi \cdot f}{\text{sampleRate}}\right) + 2 \cdot B_0 \cdot B_2 \cdot \cos\left(\frac{4 \cdot \pi \cdot f}{\text{sampleRate}}\right)}{(1 + A_1^2 + A_2^2) + 2 \cdot A_1 \cdot (1 + A_2) \cdot \cos\left(\frac{2 \cdot \pi \cdot f}{\text{sampleRate}}\right) + 2 \cdot A_2 \cdot \cos\left(\frac{4 \cdot \pi \cdot f}{\text{sampleRate}}\right)}$$

$$\text{phase} = \arctan2 \left(\frac{\left((B_0 + A_1 \cdot B_1 + A_2 \cdot B_2) + (A_1 \cdot B_0 + B_1 + A_2 \cdot B_1 + A_1 \cdot B_2) \cdot \cos\left(\frac{2 \cdot \pi \cdot f}{\text{sampleRate}}\right) + (B_0 \cdot A_2 + B_2) \cdot \cos\left(\frac{4 \cdot \pi \cdot f}{\text{sampleRate}}\right) \right)}{\left((A_1 \cdot B_0 - B_1 + A_2 \cdot B_1 - A_1 \cdot B_2) \cdot \sin\left(\frac{2 \cdot \pi \cdot f}{\text{sampleRate}}\right) + (B_0 \cdot A_2 - B_2) \cdot \sin\left(\frac{4 \cdot \pi \cdot f}{\text{sampleRate}}\right) \right)} \right)$$

$$\text{gain} = 10 \cdot \log_{10} \left(\frac{(B_0^2 + B_1^2 + B_2^2) + 2 \cdot B_1 \cdot (B_0 + B_2) \cdot \cos\left(\frac{2 \cdot \pi \cdot f}{\text{sampleRate}}\right) + 2 \cdot B_0 \cdot B_2 \cdot \cos\left(\frac{4 \cdot \pi \cdot f}{\text{sampleRate}}\right)}{(1 + A_1^2 + A_2^2) + 2 \cdot A_1 \cdot (1 + A_2) \cdot \cos\left(\frac{2 \cdot \pi \cdot f}{\text{sampleRate}}\right) + 2 \cdot A_2 \cdot \cos\left(\frac{4 \cdot \pi \cdot f}{\text{sampleRate}}\right)} \right)$$

The equations for converting between the analog (Laplace transform) coefficients and the digital (Z transform) coefficients are handled internally by the MPI, but are listed below so that one can accurately analyze the performance of the bi-quad filters.

Bi-quad Postfilter

$$D = 4 + \frac{2 \cdot a_1}{\text{sampleRate}} + \frac{a_2}{\text{sampleRate}^2}$$

$$A_1 = \frac{\left(\frac{2 \cdot a_2}{\text{sampleRate}^2} - 8 \right)}{D}$$

$$A_2 = \frac{\left(4 - \frac{2 \cdot a_1}{\text{sampleRate}} + \frac{a_2}{\text{sampleRate}^2} \right)}{D}$$

$$B_0 = \frac{\left(4 \cdot b_0 + \frac{2 \cdot b_1}{\text{sampleRate}} + \frac{b_2}{\text{sampleRate}^2} \right)}{D}$$

$$B_1 = \frac{\left(\frac{2 \cdot b_2}{\text{sampleRate}^2} - 8 \cdot b_0 \right)}{D}$$

$$B_2 = \frac{\left(4 \cdot b_0 - \frac{2 \cdot b_1}{\text{sampleRate}} + \frac{b_2}{\text{sampleRate}^2} \right)}{D}$$

Bi-linear Postfilter ($a_2 = b_2 = 0$)

$$D = \frac{2 \cdot a_1}{\text{sampleRate}}$$

$$A_1 = \frac{\left(\frac{a_1}{\text{sampleRate}} - 2 \right)}{D}$$

$$A_2 = 0$$

$$B_0 = \frac{\left(\frac{b_1}{\text{sampleRate}} + 2 \cdot b_0 \right)}{D}$$

$$B_1 = \frac{\left(\frac{b_1}{\text{sampleRate}} - 2 \cdot b_0 \right)}{D}$$

$$B_2 = 0$$

Z Transform Stability

As briefly described in the last section, it is possible for the digital filters constructed from analog filters to be unstable. One needs to ensure that:

- The filter does not continually add energy to a system.
- The filter has no phase lag at 0 frequency. (A filter with 180° phase lag will create unstable closed loop systems.)

To guarantee a filter does not continually add energy to a system, the following relationship must be satisfied by the Z transform coefficients:

$$\left| \frac{A_1}{2} \pm \sqrt{\frac{A_1^2}{2} - A_2} \right| < 1$$

To guarantee a filter has no phase lag at 0 frequency, the following relationship must be satisfied by the Z transform coefficients:

$$(1 + A_1 + A_2) \cdot (B_0 + B_1 + B_2) \geq 0$$

If it is found that this last condition is not true, then one should change the sign on all B_n coefficients. Equivalently, one can change the sign of all b_n coefficients for the Laplace (analog) transform.

[Return to Filter Objects](#)

Flash Objects

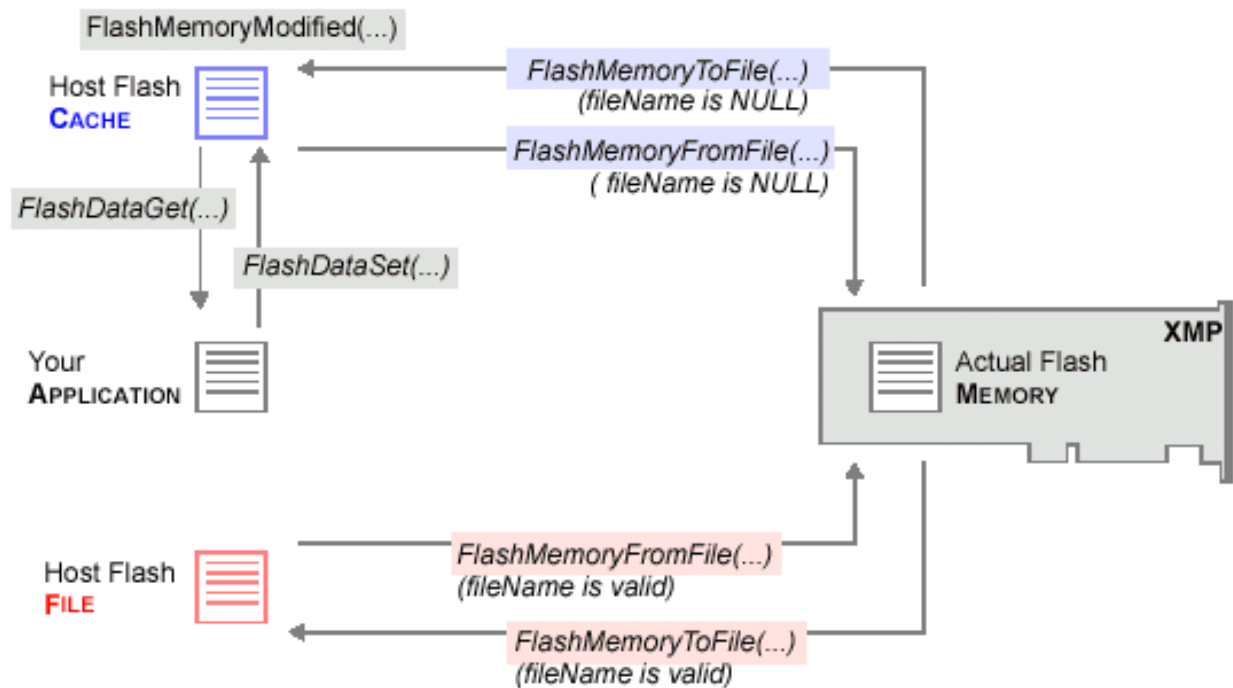
Introduction

A **Flash** object manages nonvolatile flash memory on the XMP/ZMP motion controllers. To optimize flash memory, a host-resident flash memory cache is used to provide faster writing performance.

After your application creates a Flash object (using `meiFlashCreate(...)`), the Flash object then creates a host-resident flash memory cache. `meiFlashCreate(...)` will create the flash object and initialize an internal (host-resident) cache with a copy of the flash on the board. All Flash functions (with the exception of `meiFlashMemoryFromFile(...)`) will modify the host resident cache. When you have finished modifying the host resident cache, use `meiFlashMemoryToFile(fileName=NULL)` to write the host cache data to the board's flash memory. `meiFlashMemoryToFile(...)` can be used to copy the board's flash to a user specified file. (If the filename is NULL, the data is copied to the host resident cache.) Host resident cache is deleted when `meiFlashDelete(...)` is called.

Use the `meiFlashDataGet/Set(...)` methods to move data between your application and the flash cache. Use the `meiFlashMemoryToFile` and `meiFlashMemory/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

meiFlashCreate	Create Flash object
meiFlashDelete	Delete Flash object
meiFlashValidate	Validate Flash object

Configuration and Information Methods

meiFlashConfigGet	Copy flash config from cache to application memory
meiFlashConfigSet	Copy flash config from application memory to cache
meiFlashDataGet	Get count bytes of flash data memory and write them in application memory
meiFlashDataSet	Set count bytes of flash data memory using application memory

Memory Methods

meiFlashMemoryFromFileImage	Write actual flash memory using the binary image contained in filename
meiFlashMemoryFromFileType	Copy count bytes of flash memory to application memory
meiFlashMemoryGet	Determine if flash cache has been modified
meiFlashMemoryModified	Copy count bytes of application memory to flash memory
meiFlashMemorySet	Write actual flash memory to cache or to file
meiFlashMemoryFromFile	Save actual flash memory to cache or to file
meiFlashMemoryToFile	
meiFlashMemoryVerify	

Relational Methods

meiFlashControl	Return handle of Control that is associated with Flash

Data Types

[MEIFlashConfig](#)

[MEIFlashFileMaxNameChars](#)

[MEIFlashFileMaxChars](#)

[MEIFlashFileMaxPathChars](#)

[MEIFlashFiles](#)

[MEIFlashFileType](#)

[MEIFlashMessage](#)

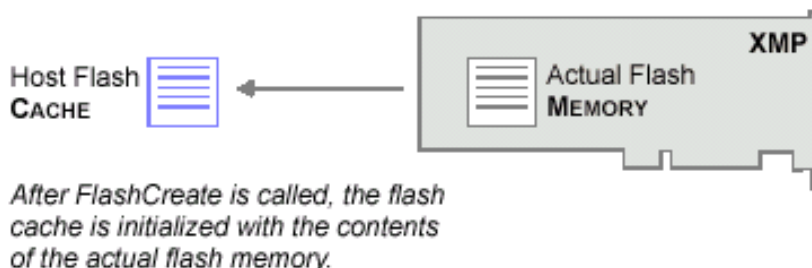
[MEIFlashSection](#)

meiFlashCreate

Declaration [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.



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 <i>FlashDelete</i> 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](#)

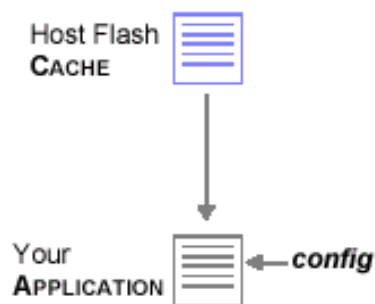
meiFlashConfigGet

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 <i>config</i>. The flash configuration includes data about the actual flash memory device (its type and size).



Return Values

MPIMessageOK	if <i>FlashConfigGet</i> successfully gets the flash configuration and writes it into the structure

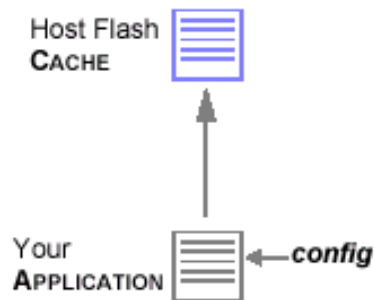
See Also [meiFlashConfigSet](#)

meiFlashConfigSet

```
Declaration      long meiFlashConfigSet(MEIFlash flash,
                           MEIFlashConfig *config)
```

Required Header

Description	FlashConfigSet sets (reads) the control configuration from the structure pointed to by <i>config</i> .



Return Values

MPIMessageOK	if <i>FlashConfigSet</i> 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

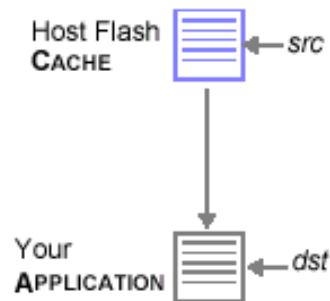
stdmei.h

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 <i>FlashDataGet</i> 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

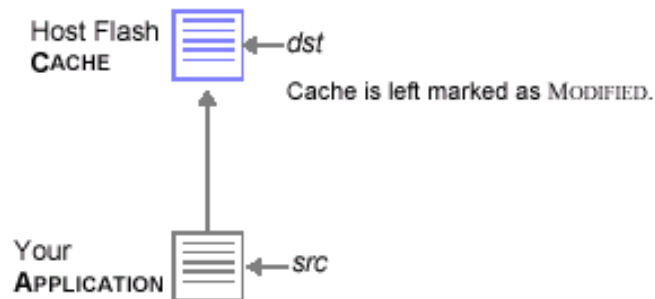
stdmei.h

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 <i>FlashDataSet</i> successfully writes to the flash cache

See Also [mpiControlMemory](#) | [meiFlashDataSetGet](#)

meiFlashMemoryFromFileImage

Declaration

```
long meiFlashMemoryFromFileImage(MEIFlash flash,
                                  const char *fileImage,
                                  MEIFlashFileType *fileType);
```

Required Header

stdmei.h

Description

FlashMemoryFromFileImage is used to erase and program controller flash memory. The values to be programmed are stored in the byte array fileImage. The parameter *fileType* is used by this method to determine the flash sector and size of the flash memory region to be programmed.

Supported values for *fileType* are:

- **MEIFlashFileTypeCode**, Code
- **MEIFlashFileTypeDataInt**, Internal Data
- **MEIFlashFileTypeDataExt**, External Data
- **MEIFlashFileTypeSynqNet**, SynqNet
- **MEIFlashFileTypeCodeAndData**, Code, Internal and External Data.
- **MEIFlashFileTypeFPGA0**, FPGA (Rincon)
- **MEIFlashFileTypeALL**, Code, Internal Data, External Data, and FPGA.

This method exists primarily for compatibility with older software. For new designs, it is recommended that you use the the [meiFlashMemoryFromFileType\(\)](#) method instead.

flash	a handle to a flash object
*fileImage	a pointer to data in memory to be copied to flash.
*fileType	where the fileImage will be copied to in flash.

Return Values

MPIMessageOK	if <i>FlashMemoryFromFileImage</i> successfully erases/programs the controller flash memory.
MEIFlashMessageFLASH_INVALID	if the flash handle is invalid.
MPIMessageNO_MEMORY	the system could not allocate enough memory for temporary storage of the flash image.

MPIMessageTIMEOUT	the flash could not be programmed.
MEIFlashMessageFLASH_WRITE_ERROR	the programmed flash did not match the flash image in verification.

See Also [meiFlashMemoryFromFileType](#)

Declaration long **meiFlashMemoryFromFileType**([MEIFlash](#) flash,
 const char *fileName,
 [MEIFlashFileType](#) fileType)

Description	FlashMemoryFromFileType writes actual flash memory using the binary image contained in <i>fileName</i> , 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 <i>FlashMemoryFromFileType</i> 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)
```

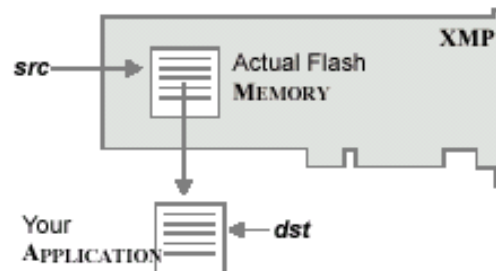
Required Header stdmei.h

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](#)

Required Header `stdmei.h`

file:///D:/pdfs/030100/html/Software-MPI/docs/Flash/Method/memmod2.htm [7/22/2004 5:58:52 PM]

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 <i>FlashMemorySet</i> successfully copies application memory to flash memory

See Also [meiFlashMemoryGet](#)

Required Header

flash	a handle to a flash object
*filesIn	a pointer to an array of flash filename structures to download to the controller. See MEIFlashFiles .
*filesOut	a pointer to an array of flash filename structures that were successfully downloaded to the controller. See MEIFlashFiles .

MPIMessageOK	if <i>FlashMemoryFromFile</i> successfully writes to actual flash memory using the file or using flash cache.

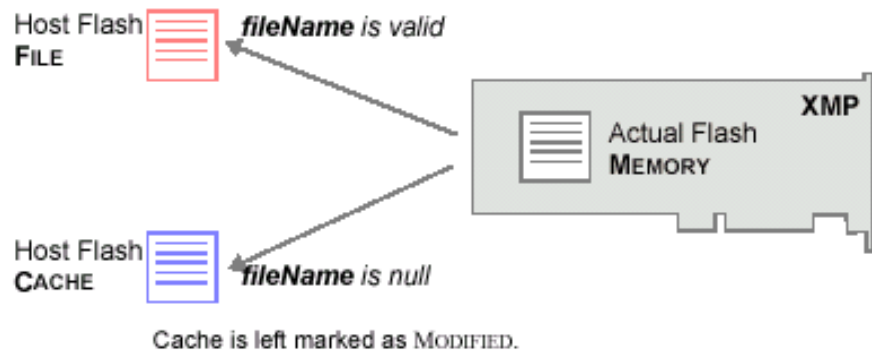
file:///D:/pdfs/030100/html/Software-MPI/docs/Flash/Method/memfmfil2.htm [7/22/2004 5:58:51 PM]

meiFlashMemoryToFile

Declaration	long	meiFlashMemoryToFile	(	MEIFlash	flash,
			const char	*fileName,	
			MEIFlashFileType	fileType)	

Required Header `stdmei.h`

Description	FlashMemoryToFile saves actual <i>flash</i> memory to a binary image contained in <i>fileName</i> , or to the flash memory cache.



Return Values

MPIMessageOK	if <i>FlashMemoryToFile</i> successfully saves actual flash memory to file or to cache

See Also

meiFlashMemoryVerify

Declaration

long **meiFlashMemoryVerify**([MEIFlash](#) **flash**,
const char ***fileName**,
[MEIFlashFileType](#) **fileType**);

Required Header

stdmei.h

Description

FlashMemoryVerify is function that verifies Flash memory against an input file. It takes a flash object, the filename of the input file, and a file type.

The **fileType** is used to tell the function what part of flash memory to verify (Code space, data space, both, etc... see MEIFlashFileType enum in flash.h).

NOTE: meiFlashMemoryVerify has been added back to the MPI in order to allow customers who wish to verify that the file loaded in flash is the same as the default file.

flash	a handle to a Flash object
*filename	a string
fileType	an enumeration corresponding to the flash file types

Return Values	
MPIMessageOK	if <i>FlashMemoryVerify</i> successfully verifies the Flash memory against an input file.
MEIFlashMessageFLASH_READ_ERROR	if the input file does not match what is in flash memory.

See Also

meiFlashControl

Declaration `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

MEIFlashConfig

MEIFlashConfig

```
typedef struct MEIFlashConfig {
    long                wordSize;
    long                sectorSize;
    long                extMemSize;
    MEIFlashSection    all;
    MEIFlashSection    code;
    MEIFlashSection    codeBoot0;
    MEIFlashSection    codeBoot;
    MEIFlashSection    codeMain;
    MEIFlashSection    data;
    MEIFlashSection    dataExt;
    MEIFlashSection    synqNet;
    MEIFlashSection    FPGA0;
} MEIFlashConfig;
```

Description

The **FlashConfig** structure contains the flash configuration parameters. This data is stored in the host's memory and is internally used by the MPI library to manage flash reads/writes. Typically, applications do not need to access the flash configurations.

WARNING:
This data structure is for MEI internal purposes only and should not be configured by a customer.

wordSize	The size of a flash word in bytes.
sectorSize	The size of a flash sector in bytes.
extMemSize	The size of the controller's external memory in bytes.
all	The flash section parameters for the controller's code, data, and local FPGA images.
code	The flash section parameters for the controller's boot and main code.
codeBoot0	The flash section parameters for the controller's boot and main code. (ZMP)
codeBoot	The flash section parameters for the controller's boot code.
codeMain	The flash section parameters for the controller's main code.
data	The flash section parameters for the controller's internal data memory.
dataExt	The flash section parameters for the controller's external data memory.
synqNet	This is for the controller's synqnet data memory.
FPGA0	The flash section parameters for the local FPGA image number 0.

See Also

[MEIFlashSection](#) | [MEIFlashFileType](#) | [meiFlashConfigGet](#) | [meiFlashConfigSet](#)

MEIFlashFileMaxNameChars

Declaration `#define MEIFlashFileMaxNameChars      (12)   /*8.3 format */`

Required Header `stdmei.h`

Description The **FlashFileMaxNameChars** define is used internally by the MPI when linking a series of strings to create the full path name of the flash file. If the user tries to use a file with more than 12 characters (8.3 format), then the MPI will not be able to create the correct filename and may result in problems when opening the file.

See Also [MEIFlashFileMaxChars](#) | [MEIFlashFileMaxPathChars](#)

MEIFlashFileMaxChars

Declaration `#define MEIFlashFileMaxChars      (120)`

Required Header `stdmei.h`

Description The MPI has to create filenames with the full path in order to open the flash files. **FlashFileMaxChars** is an arbitrary size that is used by the MPI to allocate memory space for the filename. Full path filenames should not exceed 120 characters.

See Also [MEIFlashFileMaxNameChars](#) | [MEIFlashFileMaxPathChars](#)

MEIFlashFileMaxPathChars

Declaration

```
#define MEIFlashFileMaxPathChars (MEIFlashFileMaxChars -  
MEIFlashFileMaxNameChars)
```

Required Header `stdmei.h`

Description **FlashFileMaxPathChars** is used internally by the MPI when creating full path filenames. This is the number of characters available for the path, not including the actual filename.

See Also [MEIFlashFileMaxNameChars](#) | [MEIFlashFileMaxChars](#)

MEIFlashFiles

MEIFlashFiles

```
typedef struct MEIFlashFiles {
    char binFile[MEIFlashFileMaxChars];
    char FPGAFile[MEIXmpFlashMaxFPGAFiles][MEIFlashFileMaxChars];
} MEIFlashFiles;
```

Description The **FlashFiles** structure specifies the binary files to be downloaded to the controller.

binFile	A controller firmware filename. The firmware file contains configuration data and executable code. The firmware file format is binary.
FPGAFile	An array of FPGA filenames. The FPGA file contains the binary image that is loaded into the FPGA component. Only the appropriate FPGA file can be loaded.

See Also [meiFlashMemoryFromFile](#)

MEIFlashFileType

MEIFlashFileType

```
typedef enum {
    MEIFlashFileTypeNONE = 0,
    MEIFlashFileTypeCode,
    MEIFlashFileTypeDataInt,
    MEIFlashFileTypeDataExt,
    MEIFlashFileTypeSynqNet,
    MEIFlashFileTypeCodeAndData,
    MEIFlashFileTypeFPGA0,
    MEIFlashFileTypeALL      /* Loads Code and all FPGAs
                               (for .bin files that include
                               the FPGA images) */
} MEIFlashFileType;
```

Description **FlashFileType** is an enumeration of file types. Each file type contains code or data for the controller's flash memory. This enumeration is used to specify what code/data is to be copied to flash memory with `meiFlashMemoryFromFileType(...)` or copied from flash memory with `meiFlashMemoryToFile(...)`.

MEIFlashFileTypeCode	Controller processor code only.
MEIFlashFileTypeDataInt	Controller internal data only.
MEIFlashFileTypeDataExt	Controller external data only.
MEIFlashFileTypeSynqNet	Used for flashing the synqNet page of flash.
MEIFlashFileTypeCodeAndData	Controller processor code and data.
MEIFlashFileTypeFPGA0	Local FPGA image number 0.
MEIFlashFileTypeALL	All code, data, and FPGA images.

See Also [meiFlashMemoryFromFileType](#) | [meiFlashMemoryToFile](#) | [meiFlashMemoryVerify](#)

MEIFlashMessage

MEIFlashMessage

```
typedef enum {
    MEIFlashMessageFLASH_INVALID,
    MEIFlashMessageFLASH_VERIFY_ERROR,
    MEIFlashMessageFLASH_WRITE_ERROR,
    MEIFlashMessagePATH,
    MEIFlashMessageNETWORK_TOPOLOGY_ERROR,
} MEIFlashMessage;
```

Description **FlashMessage** lists the error messages returned by the MEIFlash module.

MEIFlashMessageFLASH_INVALID

The flash object is not valid. This message code is returned by [meiFlashMemoryFromFile\(...\)](#), [meiFlashMemoryFromFileType\(...\)](#), [meiFlashMemoryToFile\(...\)](#), or [meiFlashMemoryVerify\(...\)](#) if the flash object's memory cache has not been allocated. The flash memory cache is allocated in [mpiFlashCreate\(...\)](#). To prevent this problem, create flash objects with [mpiFlashCreate\(...\)](#) and check the return value.

MEIFlashMessageFLASH_VERIFY_ERROR

The flash memory verify failed. This message code is returned by [meiFlashMemoryVerify\(...\)](#) if the read from flash memory does not match the read from the file. This indicates that the specified flash file is different from the flash memory. This message code can also be returned from [meiFlashMemoryFromFile\(...\)](#), since it calls [meiFlashMemoryVerify\(...\)](#). To correct this problem, first check that the specified file is the one you intended. If the specified file is correct, use [meiFlashMemoryFromFile\(...\)](#) to download the file.

MEIFlashMessageFLASH_WRITE_ERROR

The flash memory write failed. This message code is returned by [meiFlashMemoryFromFile\(...\)](#) or [meiFlashMemoryFromFileType\(...\)](#) if the flash memory write fails. This indicates a problem in the flash memory component or subsystem.

MEIFlashMessagePATH

The flash file path is too long. This message code is returned by [meiFlashMemoryFromFile\(...\)](#) if the file path is longer than MEIFlashFileMaxPathChars. To correct this problem, use a shorter path.

MEIFlashMessageNETWORK_TOPOLOGY_ERROR

The network topology has not been saved to flash. This message code is returned by object flash config set methods if the network topology has not been previously saved to flash. This message code serves as a warning of a potential safety problem. Saving object configurations to flash will cause those values to be sent to nodes during the next network initialization, even if the flash configurations are not compatible with the network topology. If the network topology is saved to flash, then the controller will automatically verify the topology before sending object configuration values to the nodes.

See Also [MEIFlash](#)

MEIFlashSection

MEIFlashFiles

```
typedef struct MEIFlashSection {  
    unsigned char    *address;  
    long             size;  
    long             sectorIndex;  
} MEIFlashSection;
```

Description

The **FlashSection** structure contains the flash configuration parameters for a specified section.

*address	A pointer to a flash sector address.
size	Length of the flash memory in bytes.
sectorIndex	The flash sector number.

See Also

[MEIFlashConfig](#) | [meiFlashConfigGet](#) | [meiFlashConfigSet](#)

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`.

Data Types

[`MPIAction` / `MEIAction`](#)

[`MEIDataType`](#)

[`MPIIoSource`](#)

[`MPIIoType`](#)

[`MPIIoTrigger`](#)

[`MEIMaxBiQuadSections`](#)

[`MPIModuleId` / `MEIModuleId`](#)

[`MEINetworkPort`](#)

[`MEINetworkType`](#)

[`MPIState`](#)

[`MPIStatus`](#)

[`MEIStatusFlag`](#)

[`MPIStatusMask` / `MEIStatusMask`](#)

[`MPITrajectory`](#)

[`MPIWait`](#)

Macros

[`mpiStatusMaskBIT`](#)

MPIAction / MEIAction

MPIAction

```
typedef enum {
    MPIActionINVALID,

    MPIActionNONE,
    MPIActionSTOP,
    MPIActionE_STOP,
    MPIActionE_STOP_ABORT,
    MPIActionE_STOP_CMD_EQ_ACT,
    MPIActionABORT,

    MPIActionDONE,
    MPIActionSTART,
    MPIActionRESUME,
    MPIActionRESET,
    MPIActionCANCEL_REPEAT,

} 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.
MPIActionE_STOP_CMD_EQ_ACT	The command position is set equal to the previous sample's actual position. After the E_STOP time expires, the AmpEnable is disabled. The settling parameters are supported for this action, when settleOnEstopCmdEqAct in the MPIAxisConfig structure is enabled.
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(...).
MPIActionCANCEL_REPEAT	This action makes repeating cam finish at the end of the next cycle. i.e. The cam will continue executing until it passes the start of finish of the cam table. See Repeating Cams .

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

MEIAction

```
typedef enum {
    MEIActionMAP = MPIActionLAST,

} MEIAction;
```

Description

MEIActionMAP	MotionAction will write the axis mapping relationship of the motion supervisor to the controller. This mapping is written automatically when mpiMotionStart() is called.

See Also

[mpiMotionAction](#) | [MPIMotionDecelTime](#) | [MPIMotorEventConfig](#)
[mpiMotorEventConfigGet](#) | [mpiMotorEventConfigSet](#) | [MPIEvent](#) | [MPIAxisConfig](#)

MEIDataType

```
typedef enum {
    MEIDataTypeINVALID = 0, /* this should stay zero - static arrays init
                             to zero by default */

    MEIDataTypeCHAR,
    MEIDataTypeSHORT,
    MEIDataTypeUSHORT,
    MEIDataTypeLONG,
    MEIDataTypeULONG,
    MEIDataTypeFLOAT,
    MEIDataTypeDOUBLE,
} MEIDataType;

static MEIDataType MEIFilterGainTypePID[MPIFilterCoeffCOUNT_MAX] =
{
    MEIDataTypeFLOAT, /* Kp          */ /* */
    MEIDataTypeFLOAT, /* Ki          */ /* */
    MEIDataTypeFLOAT, /* Kd          */ /* */

    MEIDataTypeFLOAT, /* Kpff        */ /* */
    MEIDataTypeFLOAT, /* Kvff        */ /* */
    MEIDataTypeFLOAT, /* Kaff        */ /* */
    MEIDataTypeFLOAT, /* Kfff        */ /* */

    MEIDataTypeFLOAT, /* MovingIMax  */ /* */
    MEIDataTypeFLOAT, /* RestIMax    */ /* */

    MEIDataTypeLONG,  /* DRate       */ /* */

    MEIDataTypeFLOAT, /* OutputLimit  */ /* */
    MEIDataTypeFLOAT, /* OutputLimitHigh */ /* */
    MEIDataTypeFLOAT, /* OutputLimitLow */ /* */
    MEIDataTypeFLOAT, /* OutputOffset */ /* */
    MEIDataTypeFLOAT, /* Ka0          */ /* */
    MEIDataTypeFLOAT, /* Ka1          */ /* */
    MEIDataTypeFLOAT, /* Ka2          */ /* */
};
```

Description

DataType is an enumeration of data types for the filter coefficients.

MEIDataTypeCHAR	character filter data type
MEIDataTypeSHORT	short integer filter data type
MEIDataTypeUSHORT	unsigned short integer filter data type
MEIDataTypeLONG	long integer filter data type
MEIDataTypeULONG	unsigned long filter data type
MEIDataTypeFLOAT	floating point filter data type
MEIDataTypeDOUBLE	double precision floating point filter data type

See Also

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

MEIMaxBiQuadSections

MEIMaxBiQuadSections

#define **MEIMaxBiQuadSections** (6)

Description

MaxBiQuadSections defines the maximum number of biquad sections in an MPIFilter's postfilter. Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

NOTE: The PIV algorithm uses the last biquad section internally and so the user can only use (MEIMaxBiQuadSections-1) sections if the PIV algorithm is the current control algorithm.

See Also

[meiFilterPostfilterGet](#) | [meiFilterPostfilterSet](#) | [meiFilterPostfilterSectionGet](#) | [meiFilterPostfilterSectionSet](#)

MPIModuleId / MEIModuleId

MPIModuleId

```
typedef enum {
    MPIModuleIdINVALID,

    MPIModuleIdMESSAGE,
    MPIModuleIdAXIS,
    MPIModuleIdCAPTURE,
    MPIModuleIdCOMMAND,
    MPIModuleIdCOMPARE,
    MPIModuleIdCOMPENSATOR,
    MPIModuleIdCONTROL,

    MPIModuleIdEVENT,
    MPIModuleIdEVENTMGR,
    MPIModuleIdFILTER,

    MPIModuleIdMOTION,
    MPIModuleIdMOTOR,

    MPIModuleIdNOTIFY,
    MPIModuleIdPATH,
    MPIModuleIdPROBE,
    MPIModuleIdRECORDER,
    MPIModuleIdSEQUENCE,

    MPIModuleIdEXTERNAL = 0x80,

    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,  
  
    MEIModuleIdCAN,  
  
    MEIModuleIdCLIENT,  
    MEIModuleIdELEMENT,  
    MEIModuleIdFLASH,  
    MEIModuleIdLIST,  
    MEIModuleIdMAP,  
    MEIModuleIdPACKET,  
    MEIModuleIdSERVER,  
  
    MEIModuleIdSYNQNET,  
    MEIModuleIdSQNODE,  
    MEIModuleIdDRIVE_MAP,  
  
    MEIModuleIdBLOCK = MEIModuleIdSQNODE,  
}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](#)

MEINetworkPort

MEINetworkPort

```
typedef enum MEINetworkPort {  
    MEINetworkPortIN0,  
    MEINetworkPortOUT0,  
} MEINetworkPort;
```

Description

MEINetworkPort enumerations are used to specify a network port. Network ports represent the physical network connections into which the cables are plugged. These ports are commonly found in pairs on SynqNet controllers and SynqNet nodes. OUT ports from the controller or an upstream node connect to the IN port of a downstream node. When the OUT port of the last downstream node is connected back to the IN port of the controller, the network is considered to be configured as a RING network type.

MEINetworkPortIN0	"IN" Network port 0
MEINetworkPortOUT0	"OUT" Network port 0

See Also [MEINetworkType](#)

MEINetworkType

MEINetworkType

```
typedef enum MEINetworkType {
    MEINetworkTypeINVALID = -1,    /* no nodes found */
    MEINetworkTypeSTRING,
    MEINetworkTypeSTRING_TERMINATED,
    MEINetworkTypeRING,
} MEINetworkType;
```

Description

The **NetworkType** enumeration lists all network topologies supported by this MPI release.

NOTE: Due to the un-terminated nature of the MEINetworkTypeSTRING network type discovering a topology of this type requires a timeout period to detect the end of the string. Thus discovering a string topology will take a longer time than other networks types.

MEINetworkTypeINVALID	No nodes were found on the network
MEINetworkTypeSTRING	The network topology type is a string of nodes
MEINetworkTypeSTRING_TERMINATED	The network topology type is a string of nodes. The last node on the string is terminated.
MEINetworkTypeRING	The network topology type is a ring of nodes.

See Also [MEISynqNetInfo](#) | [meiSynqNetInfo](#)

MPIState

MPIState

```
typedef enum {
    MPIStateIDLE,
    MPIStateMOVING,
    MPIStateSTOPPING,
    MPIStateSTOPPED,
    MPIStateSTOPPING_ERROR,
    MPIStateERROR,
} MPIState;
```

Description

State is an MPI enum that is used to describe the current state of the controller's motion state machine. MPIState resides in the MPIStatus structure. Currently MPIState is only used with motion module.

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. The stop command could have come from the firmware or MPI.
MPIStateSTOPPED	The move has stopped due to a STOP command. It requires a MotionReset to go back to IDLE OR a MotionResume to resume stopped motion. MotionStart can also be called at any time to start a new move.
MPIStateSTOPPING_ERROR	The state of the motion is performing an emergency stop and/or abort on all axes. The move is stopping due to an error (ESTOP, ABORT, etc...).
MPIStateERROR	The state of the motion is in error. The error state is generated from an E_Stop or Abort event. It requires a MotionReset to get back to IDLE before loading another move.

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).
statusMask	Value is an enumeration of bit masks for the MEIStatusFlags. The status masks represent the present condition for an object.

See Also [MPIState](#) | [MPIStatusMask](#)

MEIStatusFlag

MEIStatusFlag

```
typedef enum {  
    MEIStatusFlagBROKEN_WIRE,          /* 1 */  
    MEIStatusFlagILLEGAL_STATE,        /* 2 */  
    MEIStatusFlagABS_ENCODER_FAULT,    /* 3 */  
    MEIStatusFlagABS_ENCODER_TIMEOUT,  /* 4 */  
    MEIStatusFlagBROKEN_WIRE_SECONDARY, /* 5 */  
    MEIStatusFlagILLEGAL_STATE_SECONDARY, /* 6 */  
} MEIStatusFlag;
```

Description

StatusFlag is an enumeration of status flags (bits) for an object. The status flags represent the present condition for an object.

Please see [MEIStatusMask](#) data type for more information.

See Also

[MEIStatusMask](#) | [MPIStatusMask](#)

MPIStatusMask / MEIStatusMask

MPIStatusMask

```
typedef enum {
    MPIStatusMaskNONE    = 0x0,

    MPIStatusMaskMOTOR    = MPIStatusMaskNONE, /* 0x00000001 */

    MPIStatusMaskALL      = mpiStatusMaskBIT(MPIStatusFlagLAST) - 1
                          /* 0x00000001 */
} MPIStatusMask;
```

Description **StatusMask** is an enumeration of bit masks for the MEIStatusFlags. The status masks represent the present condition for an object.

MPIStatusMaskMOTOR Value specifies the motor's status mask.

MPIStatusMaskALL Value specifies the status mask that encompasses all the possible status flags.

See Also [MPIStatus](#)

MEIStatusMask

```
typedef enum {
    MEIStatusMaskBROKEN_WIRE          =
    mpiStatusMaskBIT(MEIStatusFlagBROKEN_WIRE),          /* 0x00000002 */
    MEIStatusMaskILLEGAL_STATE        =
    mpiStatusMaskBIT(MEIStatusFlagILLEGAL_STATE),          /* 0x00000004 */
    MEIStatusMaskABS_ENCODER_FAULT    =
    mpiStatusMaskBIT(MEIStatusFlagABS_ENCODER_FAULT),      /* 0x00000008 */
    MEIStatusMaskABS_ENCODER_TIMEOUT  =
    mpiStatusMaskBIT(MEIStatusFlagABS_ENCODER_TIMEOUT),    /* 0x00000010 */
    MEIStatusMaskBROKEN_WIRE_SECONDARY =
    mpiStatusMaskBIT(MEIStatusFlagBROKEN_WIRE_SECONDARY),  /* 0x00000020 */
    MEIStatusMaskILLEGAL_STATE_SECONDARY =
    mpiStatusMaskBIT(MEIStatusFlagILLEGAL_STATE_SECONDARY), /* 0x00000040 */

    MEIStatusMaskMOTOR = /* 0x0000001E */
                        (MEIStatusMaskBROKEN_WIRE |
                        MEIStatusMaskILLEGAL_STATE |
                        MEIStatusMaskABS_ENCODER_FAULT |
                        MEIStatusMaskABS_ENCODER_TIMEOUT),

    MEIStatusMaskALL    = /* 0x0000001E */
```



```

        (mpiStatusMaskBIT(MEIStatusFlagLAST) - 1) & ~MPIStatusMaskALL
    } MEIStatusMask;

```

Description **StatusMask** is an enumeration of bit masks for the MEIStatusFlags. The status masks represent the present condition for an object.

MEIStatusMaskBROKEN_WIRE	Broken wire on the primary encoder input signals. Occurs when any of the differential encoder input channels (A+ and A-, B+ and B-, or I+ and I-), have the same logic state. This mask indicates either a floating or shorted encoder input signal.
MEIStatusMaskILLEGAL_STATE	Illegal encoder logic state on the primary encoder input signals. Occurs when the A and B encoder input channels transition simultaneously. This mask indicates either faulty encoder signal logic, encoder frequencies that are too high, or noisy encoder signals.
MEIStatusMaskABS_ENCODER_FAULT	Absolute encoder initialization failure. Occurs when the hardware fails to read the absolute position information from an encoder.
MEIStatusMaskABS_ENCODER_TIMEOUT	Absolute encoder response timeout. Occurs when the encoder fails to respond to a request for absolute position data.
MEIStatusMaskBROKEN_WIRE_SECONDARY	Broken wire on the secondary encoder input signals. Occurs when any of the differential encoder input channels (A+ and A-, B+ and B-, or I+ and I-), have the same logic state. This mask indicates either a floating or shorted encoder input signal.
MEIStatusMaskILLEGAL_STATE_SECONDARY	Illegal encoder logic state on the secondary encoder inputs. Occurs when the A and B encoder input channels transition simultaneously. This flag indicates either faulty encoder signal logic, encoder frequencies that are too high, or noisy encoder signals.
MEIStatusMaskMOTOR	Bit mask containing all of the motor specific MEIStatusFlags set.
MEIStatusMaskALL	Bit mask containing all of the MEIStatusFlags set.

See Also [MPIStatus](#) | [MEIStatusFlag](#) | [MPIStatusFlag](#) | [MPIStatusMask](#)

MPITrajectory

MPITrajectory

```
typedef struct MPITrajectory {
    double    velocity;
    double    acceleration;
    double    deceleration;
    double    jerkPercent;
    double    accelerationJerk;
    double    decelerationJerk;
} MPITrajectory;
```

Description

The **Trajectory** structure contains the motion profile parameters for simple point to point type motion.

NOTE: Not all firmware binaries support S_CURVE_JERK and VELOCITY_JERK motion types due to code space limitations. mpiMotionStart(...) and mpiMotionModify(...) will return MPIMotionMessagePROFILE_NOT_SUPPORTED if the controller does not support the requested move type.

velocity	Rate of change of position. Specifies the constant slew rate for S_CURVE, TRAPEZOIDAL, S_CURVE_JERK, VELOCITY, and VELOCITY_JERK motion types. Units are counts per second.
acceleration	Rate of change of velocity. Specifies the initial ramp to reach constant velocity for TRAPEZOIDAL and VELOCITY motion types. Also, specifies the ramp from the initial jerk to the next jerk before constant velocity for S_CURVE, S_CURVE_JERK and VELOCITY_JERK motion types. Units are counts per second * second.
deceleration	Rate of change of velocity. Specifies the final ramp to reach zero velocity for TRAPEZOIDAL and VELOCITY motion types. Also, specifies the ramp from the jerk after constant velocity to the final jerk for S_CURVE and S_CURVE_JERK motion types. Units are counts per second * second.
jerkPercent	Portion of acceleration and deceleration ramp to perform jerk profile for S_CURVE, VELOCITY, S_CURVE_JERK, and VELOCITY_JERK motion types. Units are in percent. Range is 0.0 to 100.0.
accelerationJerk	Rate of change of acceleration. Specifies the initial jerk rates to reach constant velocity for S_CURVE_JERK and VELOCITY_JERK motion types. Units are counts per second * second * second.
decelerationJerk	Rate of change of deceleration. Specifies the final jerk rate to reach zero velocity for S_CURVE_JERK motion types. Units are counts per second * second * second.

Sample Code

```
MPITrajectory trajectory;

mpiAxisTrajectory(axis, &trajectory);

printf("Velocity %.3f\n"
       "Acceleration %.3f\n",
       trajectory.velocity,
       trajectory.acceleration);
```

See Also

[MPIMotionType](#) | [mpiMotionTrajectory](#) | [mpiMotionStart](#) | [mpiMotionModify](#)

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](#) | [mpiPlatformKey](#)

mpiStatusMaskBIT

Declaration `#define      mpiStatusMaskBIT(flag)      (0x1 << (flag))`

Required Header `stdmpi.h`

Description **StatusMaskBIT** converts the status flag into the status mask.

See Also [MPIStatusMask](#)

Message Objects

Introduction

The **Message** module manages text strings associated with the error/warning code return values that can be returned from MPI methods. Each method return value has a unique value which can be decoded using the Message methods and macros.

For example, a return value of MEISynqNetMessagePLL_ERROR has a defined value of 0x192E. Passing this value to the `mpiMessage(...)` function returns the following text string:

"SynqNet: node PLL unable to lock with drive"

If MEISynqNetMessagePLL_ERROR was the last return value received from an MPI method, the string returned from `mpiMessage(...)` would also contain extended message information. The extended message information provides greater insight into the problem that has just occurred. In this case, the extended message information would indicate which node had the PLL locking problem:

"SynqNet: node PLL unable to lock with drive : Node 3"

Notice that the extended message information is delimited from the more generic message by a colon surrounded by spaces " : "

Extended message information may not always be returned. Extended information is only available for the very last return value from the MPI -- if a second (non-zero) return value has been returned between the time when the first return value was returned and the call to `mpiMessage(...)`, then no extended information would be returned. In addition, not all messages return extended error information.

Methods

Configuration and Information Methods

[mpiMessage](#)

[mpiMessageFunction](#)

Associate message text with a function

Data Types

[MPIMessageFunction](#)

Macros

[mpiMessageID](#)

[mpiMessageMODULE](#)

[mpiMessageNUMBER](#)

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 " <i>messageText</i> " is	Then
NULL	<i>Message</i> returns a pointer to message text resident in the library
not NULL	<i>Message</i> returns the pointer <i>messageText</i> , 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 <i>messageId</i> is NULL.
Pointer to a non-zero-length string	if the mpiMessage successssfully retrieves the message associated with <i>messageId</i> .

See Also

[mpiMessageFunction](#)

mpiMessageFunction

[illegible]

Required Header

Description	MessageFunction registers <i>function</i> as the function to be called by <code>mpiMessage(...)</code> (in order to obtain the text for a message associated with module <i>moduleId</i>).

MessageFunction is typically called internally by object create methods. Applications generally do not need to call *MessageFunction* directly.

Return Values

MPIMessageOK	if <i>MessageFunction</i> successfully registers <i>function</i> as the function to be called by mpiMessage(...)

See Also [mpiMessage](#)

MPIMessageFunction

MPIMessageFunction

```
typedef const char *(*MPIMessageFunction)(long);
```

Description

MessageFunction is the type definition for the callback function used by `mpiMessage(...)`. A default callback function is provided internally to all MPI/MEI modules, but an application can also be written to override it and provide a custom message function instead.

See Also

[mpiMessage](#) | [mpiMessageFunction](#)

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 << 16)) >> 16)
```

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 using MPI Motion Attributes and MEI Motion Attributes, [click here](#).

| [Error Messages](#) |

Methods

Create, Delete, Validate Methods

mpiMotionCreate	Create Motion object
mpiMotionDelete	Delete Motion object
mpiMotionValidate	Validate Motion object

Configuration and Information Methods

mpiMotionConfigGet	Get Motion configuration
mpiMotionConfigSet	Set Motion configuration
mpiMotionFlashConfigGet	Get Motion flash config
mpiMotionFlashConfigSet	Set Motion flash config
meiMotionFrameBufferLoad	
meiMotionFrameBufferStatus	Get a Motion object (motion) frame buffer status.
meiMotionLogClose	Frees memory buffer for motion logging
meiMotionLogOpen	Allocates a memory buffer for motion logging.
meiMotionLogPrint	Writes motion log buffer data

[mpiMotionParamsGet](#)

Get Motion parameters

[mpiMotionParamsSet](#)

Set Motion parameters

[mpiMotionParamsValidate](#)

Validate Motion parameters

[mpiMotionPositionGet](#)

Get position parameters of all axes associated with Motion

[mpiMotionPositionSet](#)

Set position parameters of all axes associated with Motion

[mpiMotionStatus](#)

Get Motion status

[mpiMotionTrajectory](#)

Get trajectories for all Axis associated with Motion

Event Methods

[mpiMotionEventNotifyGet](#)

Get event mask

[mpiMotionEventNotifySet](#)

Set event mask

[mpiMotionEventReset](#)

Reset events specified in event mask

Action Methods

[mpiMotionAction](#)

Perform an action on a Motion

[mpiMotionActionFunction](#)[mpiMotionModify](#)

Modify parameters of Motion while it is executing

[mpiMotionStart](#)

Start Motion (idle state > moving state)

Memory Methods

[mpiMotionMemory](#)

Set address to be used to access Motion memory

[mpiMotionMemoryGet](#)

Get bytes of Motion memory and place it into application memory

[mpiMotionMemorySet](#)

Put (set) bytes of application memory into Motion memory

Relational Methods

[mpiMotionControl](#)

Return handle of Control object associated with Motion

[mpiMotionNumber](#)

Get index of Motion

[mpiMotionAxis](#)

Return handle of axis by index number

[mpiMotionAxisAppend](#)

Append axis to list

[mpiMotionAxisCount](#)

Return number of axes in list

[mpiMotionAxisFirst](#)

Return handle to first axis in list

[mpiMotionAxisIndex](#)

Return index value of an axis in list

[mpiMotionAxisInsert](#)

Insert axis into list associated with Motion

[mpiMotionAxisLast](#)

Return handle to last axis in list

[mpiMotionAxisListGet](#)

Get list of axes associated with Motion

[mpiMotionAxisListSet](#)

Create a list of axes associated with Motion

[mpiMotionAxisNext](#)

Get handle to next axis in list

[mpiMotionAxisPrevious](#)

Get handle to previous axis in list

[mpiMotionAxisRemove](#)

Remove handle to axis in list

Data Types

[*MEIMotionActionFunction](#)
[MPIMotionAttr](#) / [MEIMotionAttr](#)
[MEIMotionAttrHold](#)
[MEIMotionAttrHoldSource](#)
[MEIMotionAttrHoldType](#)
[MPIMotionAttrMask](#) / [MEIMotionAttrMask](#)
[MEIMotionAttrOutput](#)
[MEIMotionAttrOutputType](#)
[MPIMotionAttributes](#) / [MEIMotionAttributes](#)
[MPIMotionBESSEL](#)
[MPIMotionBSPLINE](#)
[MPIMotionCam](#)
[MPIMotionConfig](#) / [MEIMotionConfig](#)
[MPIMotionDecelTime](#)
[MEIMotionFrame](#)
[MEIMotionFrameBufferStatus](#)
[MPIMotionMessage](#) / [MEIMotionMessage](#)
[MPIMotionParams](#) / [MEIMotionParams](#)
[MPIMotionPoint](#)
[MPIMotionPT](#)
[MPIMotionPTF](#)
[MPIMotionPVT](#)
[MPIMotionPVTF](#)
[MPIMotionSCurve](#)
[MPIMotionSCurveJerk](#)
[MPIMotionSPLINE](#)
[MEIMotionTrace](#)
[MPIMotionTrapezoidal](#)
[MPIMotionType](#) / [MEIMotionType](#)
[MPIMotionVelocity](#)

Macros

[mpiMotionATTR](#)
[mpiMotionAttrMaskBIT](#)

[mpiMotionTYPE](#)

Required Header

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).

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

file:///D:/pdfs/030100/html/Software-MPI/docs/Motion/Method/create1.htm [7/26/2004 1:21:05 PM]

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 <i>MotionDelete</i> 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

Description	MotionConfigGet gets the configuration of a Motion object (<i>motion</i>) and puts (writes) it in the structure pointed to by <i>config</i> , and also writes it into the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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 <i>MotionConfigGet</i> 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

Description	MotionConfigSet sets (writes) the configuration of a Motion object (<i>motion</i>) using data from the structure pointed to by <i>config</i> , and also using data from the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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 <i>MotionConfigSet</i> 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 <i>MotionFlashConfigGet</i> 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 <i>MotionFlashConfigSet</i> successfully sets (writes) the Motion’s flash configuration using data from the structure(s)

See Also

[MEIMotionConfig](#) | [MEIFlash](#) | [mpiMotionFlashConfigGet](#)

meiMotionFrameBufferStatus

Declaration

```
long meiMotionFrameBufferStatus(MPI_Motion motion,
                                MEIMotionFrameBufferStatus *status)
```

Required Header stdmpi.h

Description **MotionFrameBufferStatus** gets a Motion object (motion) frame buffer status and writes it to the structure pointed to by status.

motion	a handle to the Motion object
*status	a pointer to the motion frame buffer status structure returned by the method

Return Values

MPIMessageOK	if <i>MotionFrameBufferStatus</i> successfully gets the frame buffer status

See Also [MEIMotionFrameBufferStatus](#)

meiMotionLogClose

Declaration long **meiMotionLogClose**([MPI_Motion](#) **motion**);

Required Header stdmpi.h

Description **MotionLogClose** frees the memory buffer for motion logging.

motion	a handle to the Motion object

Return Values

MPIMessageOK	if <i>MotionLogClose</i> successfully frees the memory buffer for motion logging.

See Also [meiMotionLogOpen](#) | [meiMotionLogPrint](#)

Required Header `stdmpi.h`

motion	a handle to the Motion object
maxEntries	The maximum number of log entries. This value determines the log buffer memory size.

file:///D:/pdfs/030100/html/Software-MPI/docs/Motion/Method/logopen2.htm [7/26/2004 1:21:08 PM]

Required Header

motion	a handle to the Motion object
*filename	a pointer to a file name string.

MPIMessageOK	if <i>MotionLogClose</i> successfully writes the data into a file specified by the <i>fileName</i> .

file:///D:/pdfs/030100/html/Software-MPI/docs/Motion/Method/logprint2.htm [7/26/2004 1:21:08 PM]

Required Header

```
stdmpi.h
```

motion	a handle to the Motion object
*params	a pointer to the motion parameters structure returned by the method

MPIMessageOK	if <i>MotionParamsGet</i> successfully returns the parameters associated with a Motion object

See Also [mpiMotionParamsSet](#) | [mpiMotionStart](#) | [meiMotionParamsValidate](#)

mpiMotionParamsSet

[illegible]

Required Header

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 <i>MotionParamsSet</i> 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 " <i>*point</i> " is	Then
not Null	the number of points specified by <i>params</i> is written to the location (pointed to by <i>points</i>)

Return Values	
MPIMessageOK	if <i>MotionParamsValidate</i> successfully validates the type-specific motion parameters using the coordinate system of <i>motion</i>

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 <i>MotionPositionGet</i> 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 <i>MotionPositionSet</i> 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).

motion	a handle to a Motion object
*status	a pointer to MPIStatus structure
*external	a pointer to an implementation-specific structure

XMP Only

external should always be set to NULL.

Return Values	
MPIMessageOK	if <i>MotionStatus</i> successfully writes the Motion’s status to the structure(s).
MPIMessageARG_INVALID	if the <i>status</i> pointer is NULL.

See Also

Required Header

NOTE: deceleration, jerkPercent, accelerationJerk, and decelerationJerk fields cannot be read from the controller and consequently are set to zero.

```
MPITrajectory trajectory;

    mpiAxisTrajectory(axis, &trajectory);

    printf("Velocity %.3f\n"
           "Acceleration %.3f\n",
           trajectory.velocity,
           trajectory.acceleration);
```

MPIMessageOK	if <i>MotionTrajectory</i> successfully gets the trajectories for all axes associated with a Motion object

file:///D:/pdfs/030100/html/Software-MPI/docs/Motion/Method/trj1.htm [7/26/2004 1:21:11 PM]

Required Header

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 disabled for event types not specified in *eventmask*.

external either points to a structure of type **MEIEEventNotifyData{}** or is NULL. The **MEIEEventNotifyData{}** structure is an array of firmware addresses, whose contents are placed into the **MEIEEventStatusInfo{}** structure (of all events generated by this object).

MPIMessageOK	if <i>MotionEventNotifyGet</i> successfully writes the event mask to the location(s)

file:///D:/pdfs/030100/html/Software-MPI/docs/Motion/Method/evtnfyget1.htm [7/26/2004 1:21:11 PM]

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 <i>MotionEventNotifySet</i> successfully requests host notification of the event(s) that are specified by <i>eventMask</i> and generated by <i>motion</i>

See Also

[MPIEventType](#) | [MEIEventNotifyData](#) | [MEIEventStatusInfo](#)
[mpiEventMaskMOTION](#) | [mpiEventMaskAXIS](#) | [mpiMotionEventNotifyGet](#)

mpiMotionEventReset

Declaration

```
long mpiMotionEventReset(MPIMotion motion, MPIEventMask eventMask)
```

Required Header

Description	MotionEventReset resets the event(s) that are specified in <i>eventMask</i> and generated by <i>motion</i> . Your application must call <i>MotionEventReset</i> only after one or more latchable events have occurred.

Return Values

MPIMessageOK	if <i>MotionEventReset</i> successfully resets the event(s) that are specified in <i>eventMask</i> and generated by <i>motion</i>

See Also

Required Header

The deceleration stop and Estop times can be set with:

where the structure `MPIMotionConfig`, contains the elements:

In the 20030818 and later releases, `mpiMotionAction(...)` was optimized for the `MPIActionRESET` action type. To make the execution as fast as possible, service commands to clear node status, CRCs, and the packet error counters were removed. Service commands to clear amp and encoder faults are only sent to the node when an amp fault or encoder fault is active.

The method, [meiSqNodeStatusClear\(...\)](#) was added to clear node faults. It sends service commands to the node in order to clear the node status, CRCs, packet error counters, and hardware latches for the nodeDisable and powerFault.

file:///D:/pdfs/030100/html/Software-MPI/docs/Motion/Method/acn1.htm (1 of 2) [7/26/2004 1:21:12 PM]

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.
MEIActionMAP	MotionAction will write the axis mapping relationship of the motion supervisor to the controller. This mapping is written automatically when mpiMotionStart() is called.

Return Values

MPIMessageOK if *MotionAction* successfully performs the specified action on the Motion object

See Also [meiSqNodeStatusClear](#)

meiMotionActionFunction

Declaration

```
MEIMotionActionFunction  meiMotionActionFunction(MPIMotion      motion,  
                                                    MEIMotionActionFunction function);
```

Required Header stdmpi.h

Description [MotionActionFunction](#)

motion	a handle to the Motion object
function	

Return Values

	if <i>MotionActionFunction</i> successfully
MPIMessageOK	

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).

Each axis has a 128 frame buffer (FIFO). [mpiMotionStart](#) and mpiMotionModify calls will load up to 10 frames. No provision has been made to check if the new frames will overwrite the currently executing frames. This could happen after about 12 Start/Modify calls are made with the APPEND attribute.

The XMP firmware velocity frame execution time for point-to-point moves cannot exceed 16,384,000 samples. With the sample rate configured for 2000 (default), the maximum velocity time is 2.27 hours. If the commanded motion exceeds the maximum frame time, the axes will stop abruptly after 16,384,000 samples. The motors will still maintain servo control and no errors are reported.

Return Values	
MPIMessageOK	if <i>MotionModify</i> successfully modifies the parameters of a Motion object
MPIMotionMessageIDLE	if no Motion was in progress when <i>MotionModify</i> was called. In order for <i>MotionModify</i> to work, there must be a Motion in progress.
MPIMotionMessageAUTO_START	if <i>MotionModify</i> 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	

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.

Each axis has a 128 frame buffer (FIFO). mpiMotionStart and [mpiMotionModify](#) calls will load up to 10 frames. No provision has been made to check if the new frames will overwrite the currently executing frames. This could happen after about 12 Start/Modify calls are made with the APPEND attribute.

If E-stop deceleration rates are not set high enough to stop within the number of frames specified by the empty frame limit, the axis jumps on a frame underflow. The axis will E-stop along the path of the last frames in the buffer, then continue onto the next frames (which are the frames from 128 frames ago). This can potentially cause a dangerous condition.

When successive non-integer length relative motions are commanded, the fractional portion is truncated and discarded. This may cause problems if the fractional value needs to be summed over multiple moves.

The XMP firmware velocity frame execution time for point-to-point moves cannot exceed 16,384,000 samples. With the sample rate configured for 2000 (default), the maximum velocity time is 2.27 hours. If the commanded motion exceeds the maximum frame time, the axes will stop abruptly after 16,384,000 samples. The motors will still maintain servo control and no errors are reported.

Return Values	
MPIMessageOK	<i>MotionStart</i> successfully changes a Motion object from the idle state to the moving state
MPIMotionMessageMOVING	if <i>MotionStart</i> was called when a Motion is in progress
For more Returns, click here	

See Also [mpiMotionParamsSet](#) | [MPIState](#) | See [diagram](#) on how mpiMotionStart works

Returns for mpiMotionStart and mpiMotionModify

Both mpiMotionStart and mpiMotionModify use motion attributes, which are summarized in the table below.

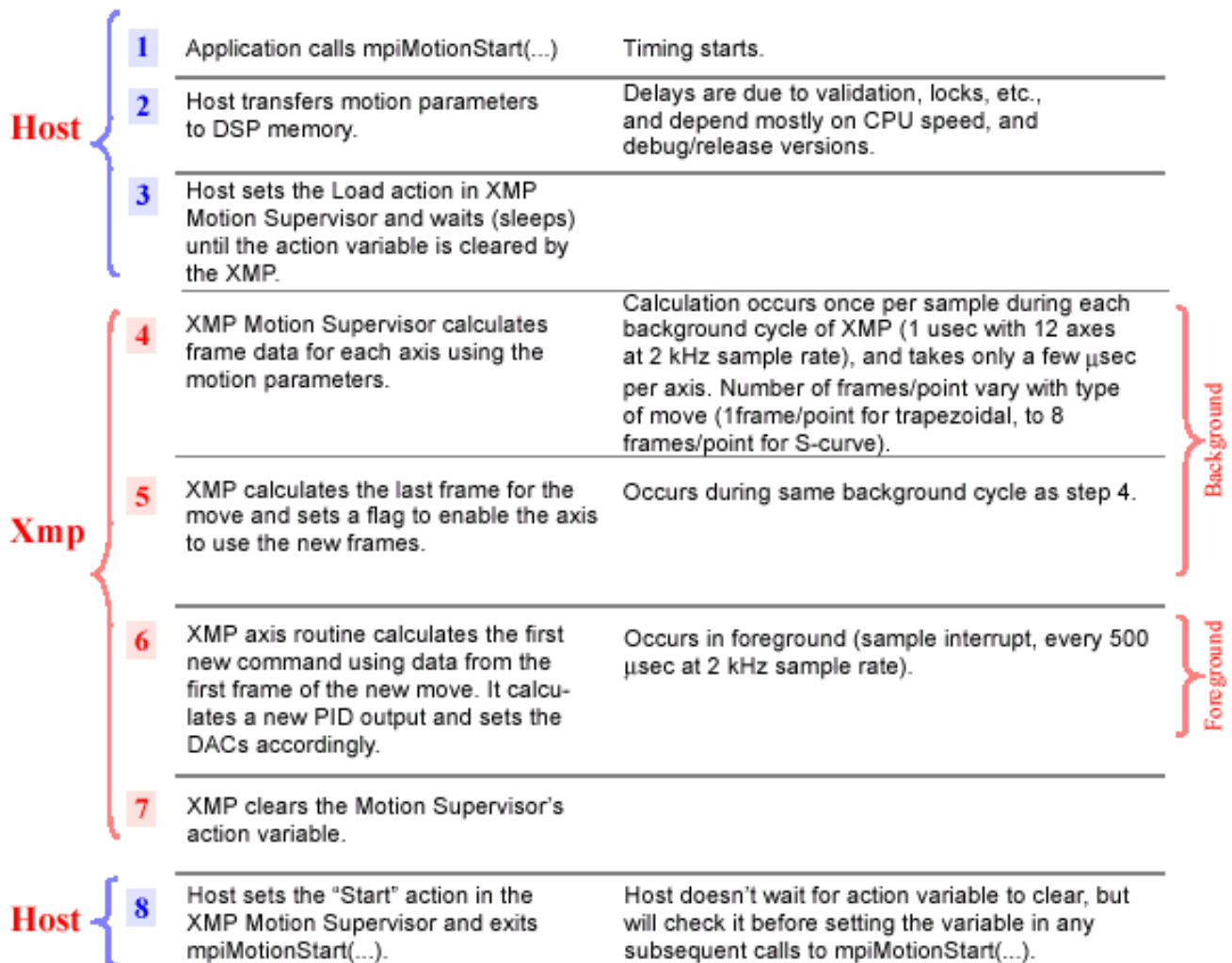
Return Values - for both mpiMotionStart and MotionModify	
MPIMotionMessageERROR	if <i>MotionStart</i> is called when the Motion Supervisor is in the error state.
MPIMotionMessageAXIS_COUNT	if the motion object has no axes.
MPIMotionMessagePROFILE_ERROR	if the Motion profile specified by the parameters cannot be calculated correctly.
MPIMotionMessageATTRIBUTE_INVALID	if the specified motion attribute mask is not valid with the motion type. Please see possible causes for receiving this message.
MPIMessagePARAM_INVALID	if the specified motion parameters are not valid with the Motion type. Please see possible causes for receiving this message.
MPIMessageARG_INVALID	if any of the <i>MotionStart</i> arguments are not valid. Possible Causes: Number of axes on the motion commanded exceeds MEIXmpMAX_COORD_AXES.
MPIMessageUNSUPPORTED	if the motion type is not supported by the controller. Possible Causes: Unsupported motion type specified. List of supported motion types: MPIMotionTypePT MPIMotionTypePTF MPIMotionTypePVT MPIMotionTypePVTF MPIMotionTypeSPLINE MPIMotionTypeBESSEL MPIMotionTypeBSPLINE MPIMotionTypeBSPLINE2 MPIMotionTypeS_CURVE MPIMotionTypeS_CURVE_JERK MPIMotionTypeTRAPEZOIDAL MPIMotionTypeVELOCITY

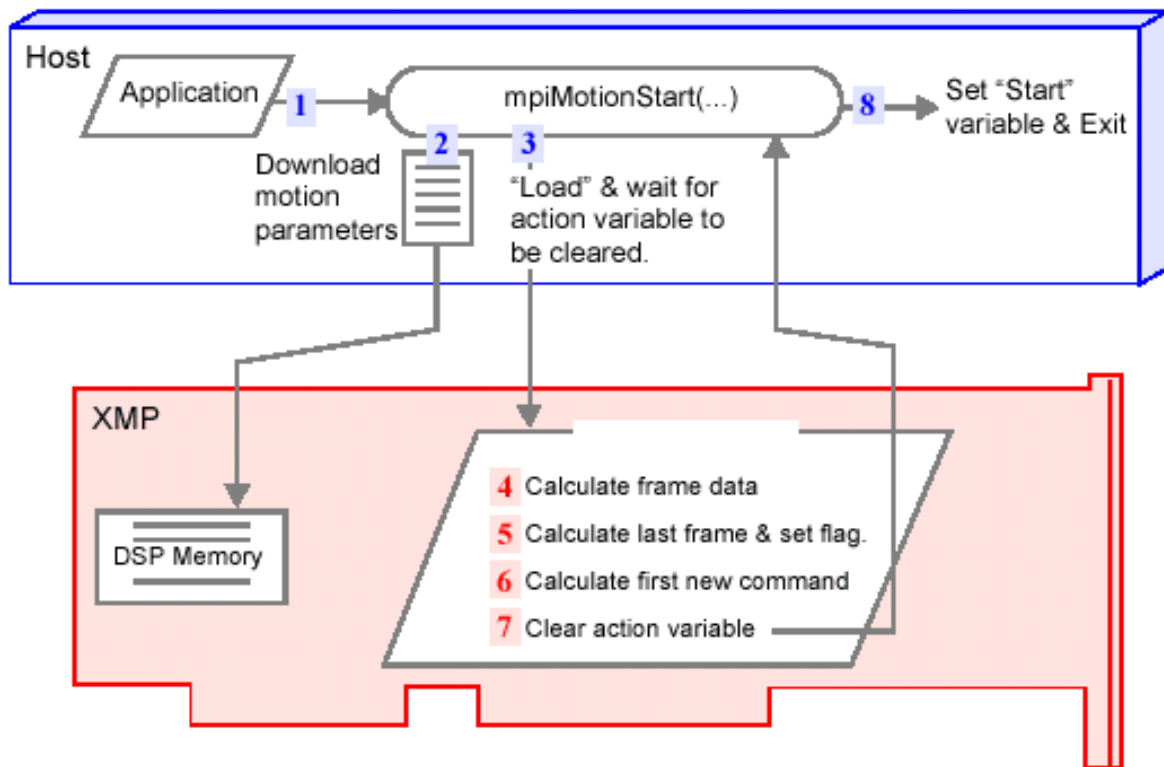
	MPIMotionTypeVELOCITY_JERK
	MEIMotionTypeFRAME

Return to [mpiMotionStart](#) or [mpiMotionModify](#)

Diagram of mpiMotionStart

The diagram below explains how mpiMotionStart(...) executes.





Return to [mpiMotionStart](#)

mpiMotionMemory

```
long mpiMotionMemory(MPI_Motion motion,
                     void **memory)
```

Required Header

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 <i>MotionMemory</i> 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 <i>MotionMemoryGet</i> successfully copies <i>count</i> 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 <i>MotionMemorySet</i> successfully copies <i>count</i> bytes of application memory to a Motion’s memory

See Also

[mpiMotionMemoryGet](#) | [mpiMotionMemory](#)

mpiMotionControl

Declaration `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 <i>motion</i> is invalid

See Also [mpiMotionCreate](#) | [mpiControlCreate](#)

mpiMotionNumber

```
long mpiMotionNumber(MPI_Motion motion,
                    long *number)
```

Required Header

Description	MotionNumber writes the index of a Motion object (<i>motion</i> , on the motion controller that the Motion object is associated with) to the contents of <i>number</i> .

Return Values

MPIMessageOK	if <i>MotionNumber</i> successfully writes the index of a Motion object to the contents of <i>number</i>

See Also

mpiMotionAxis

Declaration `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 <i>index</i> th Axis of a Motion (<i>motion</i>)
MPIHandleVOID	if <i>motion</i> is invalid if <i>index</i> is less than 0 if <i>index</i> is greater than or equal to <code>mpiMotionAxisCount(motion)</code>
MPIMessageARG_INVALID	<i>index</i> is a negative number.
MEIListMessageELEMENT_NOT_FOUND	<i>index</i> is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	<i>motion</i> is an invalid handle.

See Also

Required Header

When using multiple Motion Supervisors that share axes, Motion events (Done, AtVelocity) are sent to both Motion objects, no matter which Motion Supervisor commanded the motion. This occurs, because the Motion events are derived from the Motion Supervisor status, which is derived from each axis' status.

motion	a handle to the Motion object.
axis	a handle to an Axis object.

Return Values

MPIMessageOK	if <i>MotionAxisAppend</i> successfully appends an Axis to a Motion object
MPIMessageHANDLE_INVALID	Either <i>motion</i> or <i>axis</i> is an invalid handle.
MPIMessageUNSUPPORTED	The list already contains the maximum number of elements (MEIXmpMAX_COORD_AXES). -or- <i>motion</i> and <i>axis</i> are on different controllers.
MPIMessageOBJECT_NOT_ENABLED	<i>axis</i> is not an enabled axis.
MPIMessageOBJECT_ON_LIST	<i>axis</i> 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 (<i>motion</i>)
-1	if <i>motion</i> is invalid
0	if <i>motion</i> is empty

See Also [mpiMotionAxis](#) | [mpiMotionAxisAppend](#)

mpiMotionAxisFirst

Declaration `MPIAxis mpiMotionAxisFirst(MPIMotion motion)`

Required Header `stdmpi.h`

Description `MotionAxisFirst` return the first element in the list. This function can be used in conjunction 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 (<i>motion</i>)
MPIHandleVOID	if <i>motion</i> is invalid if <i>motion</i> is empty
MPIMessageHANDLE_INVALID	<i>motion</i> is an invalid handle.

See Also [mpiMotionAxisNext](#) | [mpiMotionAxisLast](#)

Required Header

motion	a handle to the Motion object.
axis	a handle to an Axis object.

index	of an Axis (<i>axis</i>) in a Motion (<i>motion</i>)
-1	if <i>motion</i> is invalid if the Axis (axis) was not found in the Motion (<i>motion</i>)

file:///D:/pdfs/030100/html/Software-MPI/docs/Motion/Method/axinx1.htm [7/26/2004 1:21:17 PM]

mpiMotionAxisInsert

Declaration

```
long mpiMotionAxisInsert(MPIMotion motion,  
                        MPIAxis axis,  
                        MPIAxis insert)
```

Required Header

Description	MotionAxisInsert inserts an Axis (<i>insert</i>) in a Motion (<i>motion</i>), just after the specified Axis (<i>axis</i>).

Return Values

MPIMessageOK	if <i>MotionAxisInsert</i> successfully inserts an Axis (<i>insert</i>) in a Motion (<i>motion</i>) following a specified Axis (<i>axis</i>)

See Also [mpiMotionAxis](#)

mpiMotionAxisLast

Declaration `MPIAxis mpiMotionAxisLast(MPIMotion motion)`

Required Header `stdmpi.h`

Description `MotionAxisLast` returns the last element in the list. This function can be used in conjunction 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 (<i>motion</i>)
MPIHandleVOID	if <i>motion</i> is invalid if <i>motion</i> is empty
MPIMessageHANDLE_INVALID	Either <i>motion</i> or <i>axis</i> is an invalid handle.

See Also [mpiMotionAxisPrevious](#)

mpiMotionAxisListGet

```
Declaration      long mpiMotionAxisListGet(MPIMotion motion,  
                  long *axisCount,  
                  MPIAxis *axisList)
```

Required Header

Description	MotionAxisListGet gets the coordinate system of a Motion object (<i>motion</i>).

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 <i>MotionAxisListGet</i> successfully gets the coordinate system of a Motion object

See Also [mpiMotionAxisListSet](#) | [mpiMotionAxis](#)

Declaration long `mpiMotionAxisListSet`([MPIMotion](#) `motion`,
 long `axisCount`,
 [MPIAxis](#) `*axisList`)

Description	MotionAxisListSet creates a coordinate system of <i>axisCount</i> dimensions, using the Axis handles specified by <i>axisList</i> . Any existing coordinate system is completely replaced.

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.

```

/* Creates a 1:1 mapping between motion supervisor and axis */
MPIMotion motion;
MPIAxis axis;
long returnValue;

axis = mpiAxisCreate(control, axisNumber);
returnValue = mpiAxisValidate(axis);
msgCHECK(returnValue);

motion = mpiMotionCreate(control,
                          axisNumber,
                          NULL);
returnValue = mpiMotionValidate(motion);
msgCHECK(returnValue);

returnValue = mpiMotionAxisListSet(motion,
                                    1,
                                    &axis);
msgCHECK(returnValue);

returnValue = mpiMotionAction(motion, MEIActionMAP);
msgCHECK(returnValue);

/* Don't forget to delete your motion supervisor and axis objects */

```

Return Values

MPIMessageOK	if <i>MotionAxisListSet</i> successfully creates a coordinate system

See Also [mpiMotionCreate](#) | [mpiMotionAxisListGet](#) | [mpiMotionAxis](#)

mpiMotionAxisNext

Declaration `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 conjunction 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 (<i>axis</i>) in a Motion (<i>motion</i>)
MPIHandleVOID	if <i>motion</i> is invalid if the specified Axis (<i>axis</i>) is the last axis in a Motion (<i>motion</i>)
MPIMessageHANDLE_INVALID	Either <i>motion</i> or <i>axis</i> is an invalid handle.

See Also [mpiMotionAxisFirst](#) | [mpiMotionAxisPrevious](#)

mpiMotionAxisPrevious

Declaration `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 conjunction 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 specified Axis (axis) is the first Axis in a Motion (motion)
MPIMessageHANDLE_INVALID	Either motion or axis is an invalid handle.

See Also [mpiMotionAxisLast](#) | [mpiMotionAxisNext](#)

Required Header

Return Values

See Also

MEIMotionActionFunction

MEIMotionActionFunction

```
typedef long      /* Callback prototype */
    (*MEIMotionActionFunction)(MPI_Motion      motion,
                                MPI_MotionType  type,
                                MPI_XmpAction   xmpAction);
```

Description

The **ActionFunction** is a function prototype definition. This is the prototype for a user defined callback function in the mpiMotionAction(...) method. If a callback function is set using meiMotionActionFunction(...), then every time the application calls mpiMotionAction(...), the callback function will be executed. Your motion action callback function must implement the exact interface of the above prototype.

Using a callback function can be useful to log details of mpiMotionAction(...) calls.

WARNING:

Defining a callback function will affect the execution time of the mpiMotionAction(...) method call. In order to preserve system performance, the callback function should be written to execute as fast as possible.

motion	The motion object handle used to call mpiMotionAction() will be passed to the callback function.
type	The type of motion that the action was called to act upon will be passed to the callback function.
xmpAction	The type of action that was requested will be passed to the callback function.

See Also [mpiMotionAction](#) | [meiMotionActionFunction](#)

MPIMotionAttr / MEIMotionAttr

MPIMotionAttr

```
typedef enum {  
  
    MPIMotionAttrAPPEND,  
    MPIMotionAttrAUTO_START,  
    MPIMotionAttrDELAY,  
    MPIMotionAttrID,  
    MPIMotionAttrELEMENT_ID,  
    MPIMotionAttrRELATIVE,  
    MPIMotionAttrSYNC_END,  
    MPIMotionAttrSYNC_START,  
    MPIMotionAttrREPEAT,  
    MPIMotionAttrMASTER_START,  
    MPIMotionAttrNO_BACKTRACK,  
    MPIMotionAttrNO_BACKTRACK_HOLD,  
  
    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.

MPIMotionAttrAPPEND	This bit 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 bit can be used with mpiMotionStart(...) or mpiMotionModify(...).
MPIMotionAttrAUTO_START	This bit 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.

MPIMotionAttrDELAY	This bit enables a time delay (seconds) before the motion profile begins. This mask can be used with <code>mpiMotionStart(...)</code> . Motion with Modify is not supported with the DELAY attribute. Please see MPIMotionAttributes for more information.
MPIMotionAttrID	This bit enables an identification tag to be stored in the motion profile. Please see MPIMotionAttributes for more information. This bit can be used with <code>mpiMotionStart(...)</code> and <code>mpiMotionModify(...)</code> .
MPIMotionAttrELEMENT_ID	This bit enables an identification tag to be stored in the path motion profiles. Please see MPIMotionAttributes for more information.
MPIMotionAttrRELATIVE	This bit changes the profile target position from absolute to relative coordinates. Currently only supports APPEND and ID attributes. Support for PT, PVT, SPLINE, BESSEL, or BSPLINE motion types will be added in the future.
MPIMotionAttrSYNC_END	This bit 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.
MPIMotionAttrSYNC_START	This bit 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.
MPIMotionAttrREPEAT	This attribute generates a repeating cam motion. If you use this attribute you need to fill in the repeatFrom field in the MPIMotionAttributes structure. See Repeating Cams .
MPIMotionAttrMASTER_START	This attribute specifies the position of the master that a cam will start. If you use this attribute you need to fill in the masterStart field in the MPIMotionAttributes structure. See Starting at Specific Master Positions .

MPIMotionAttrREPEAT_NO_BACKTRACK	This attribute modifies a cam motion so that when the slave axes will only progress if the master is moving in a positive direction, if the master changes direction and moves backwards the slave axes will hold that position until the master velocity returns to the original direction. This attribute cannot be specified with the MPIMotionAttrNO_BACKTRACK attribute. See Reversal of the Master - Backtracking .
MPIMotionAttrREPEAT_NO_BACKTRACK_HOLD	This attribute modifies a cam motion so that when the master changes direction the slave axes will hold that position until the master returns to the point where the direction changed. This attribute cannot be specified with the MPIMotionAttrFORWARD_ONLY attribute. See Reversal of the Master with NO_BACKTRACK_HOLD .

MEIMotionAttr

```
typedef enum {
    MEIMotionAttrEVENT,
    MEIMotionAttrFINAL_VEL,
    MEIMotionAttrNO_REVERSAL,
    MEIMotionAttrHOLD,
    MEIMotionAttrHOLD_LESS,
    MEIMotionAttrHOLD_GREATER,
    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.

MEIMotionAttrEVENT	This bit allows the user to specify an MPIEventMask during a motion.
MEIMotionAttrFINAL_VEL	This bit allows the user to specify a non-zero target velocity for point to point motion types.
MEIMotionAttrNO_REVERSAL	This bit prevents a motion profile from changing direction.
MEIMotionAttrHOLD	This bit prevents a motion profile from executing until the specified trigger conditions are met. MPIMotionMessageATTRIBUTE_INVALID will be returned if MEIMotionAttrHOLD is used with a mpiMotionModify(...) method.
MEIMotionAttrHOLD_LESS	Motion attribute bit for less than or equal hold logic. MPIMotionMessageATTRIBUTE_INVALID will be returned if MEIMotionAttrHOLD_LESS is used with a mpiMotionModify(...) method.
MEIMotionAttrHOLD_GREATER	Motion attribute bit for greater than or equal hold logic. MPIMotionMessageATTRIBUTE_INVALID will be returned if MEIMotionAttrHOLD_GREATER is used with a mpiMotionModify(...) method.
MEIMotionAttrOUTPUT	This bit allows the user to set or clear bits during a motion.

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 (samples). 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;
    struct {
        long   number;
        long   position;
    } axis;
    struct {
        long   *address;
        long   mask;
        long   pattern;
    } user;
} 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.
input.mask	This value specifies the AND mask when MEIMotionAttrHoldTypeINPUT is used.
input.pattern	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.
motor.number	This value specifies the motor number when MEIMotionAttrHoldTypeMOTOR is used.
motor.mask	This value specifies the AND mask when MEIMotionAttrHoldTypeMOTOR is used.
motor.pattern	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.
axis.number	An axis's number (0, 1, 2, etc.). Must be specified when the AXIS_POSITION_COMMAND or AXIS_POSITION_ACTUAL motion hold types are used.

MEIMotionAttrHoldType

MEIMotionAttrHoldType

```
typedef enum {
    MEIMotionAttrHoldTypeINVALID,
    MEIMotionAttrHoldTypeNONE,
    MEIMotionAttrHoldTypeGATE,
    MEIMotionAttrHoldTypeINPUT,
    MEIMotionAttrHoldTypeMOTOR,
    MEIMotionAttrHoldTypeAXIS_POSITION_ACTUAL,
    MEIMotionAttrHoldTypeAXIS_POSITION_COMMAND,
    MEIMotionAttrHoldTypeUSER_ADDRESS,
} 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.
MEIMotionAttrHoldTypeAXIS_POSITION_ACTUAL	Motion hold input trigger from an axis's actual position.
MEIMotionAttrHoldTypeAXIS_POSITION_COMMAND	Motion hold input trigger from an axis's command position.
MEIMotionAttrHoldTypeUSER_ADDRESS	Motion hold input trigger from a user specifiable controller memory address.

Remarks

These types specify the motion profile trigger condition. The hold trigger value is specified with the MEIMotionAttrHoldSource data type.

See Also [MEIMotionAttrHoldSource](#) | [MEIMotionAttrHold](#)

axis.position	A position comparison value. Must be specified when the AXIS_POSITION_COMMAND or AXIS_POSITION_ACTUAL motion hold types are used.
user.address	A controller memory address. Must be specified when the USER_ADDRESS motion hold type is used.
user.mask	A bitwise AND mask for the user specified controller memory address. Must be specified when the USER_ADDRESS motion hold type is used.
user.pattern	A comparison value for the masked user.address. When the pattern matches the masked value, the motion will be triggered. Must be specified when the USER_ADDRESS motion hold type is used.

See Also [MEIMotionAttrHoldType](#) | [meiControlGateGet](#) | [meiControlGateSet](#)

MPIMotionAttrMask / MEIMotionAttrMask

MPIMotionAttrMask

```
typedef enum {
    MPIMotionAttrMaskAPPEND,
    MPIMotionAttrMaskAUTO_START,
    MPIMotionAttrMaskDELAY,
    MPIMotionAttrMaskID,
    MPIMotionAttrMaskELEMENT_ID,
    MPIMotionAttrMaskRELATIVE,
    MPIMotionAttrMaskSYNC_END,
    MPIMotionAttrMaskSYNC_START,
    MPIMotionAttrMaskREPEAT,
    MPIMotionAttrMaskMASTER_START,
    MPIMotionAttrMaskNO_BACKTRACK,
    MPIMotionAttrMaskNO_BACKTRACK_HOLD,

    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. This mask can be used with mpiMotionStart(...). Motion with Modify is not supported with the DELAY attribute. Please see MPIMotionAttributes for more information.

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 <code>mpiMotionStart(...)</code> and <code>mpiMotionModify(...)</code> .
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. Currently only supports APPEND and ID attributes. Support for PT, PVT, SPLINE, BESSEL, or BSPLINE motion types will be added in the future.
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.
MPIMotionAttrMaskREPEAT	This attribute generates a repeating cam motion. If you use this attribute you need to fill in the repeatFrom field in the MPIMotionAttributes structure. See Repeating Cams .
MPIMotionAttrMaskMASTER_START	This attribute specifies the position of the master that a cam will start. If you use this attribute you need to fill in the masterStart field in the MPIMotionAttributes structure. See Starting at Specific Master Positions .

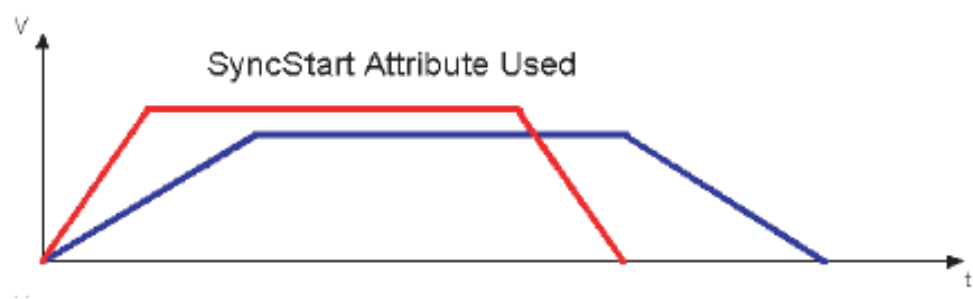
MPIMotionAttrMaskNO_BACKTRACK	This attribute modifies a cam motion so that when the slave axes will only progress if the master is moving in a positive direction, if the master changes direction and moves backwards the slave axes will hold that position until the master velocity returns to the original direction.This attribute cannot be specified with the MPIMotionAttrNO_BACKTRACK attribute. See Reversal of the Master - Backtracking.
MPIMotionAttrMaskNO_BACKTRACK_HOLD	This attribute modifies a cam motion so that when the master changes direction the slave axes will hold that position until the master returns to the point where the direction changed.This attribute cannot be specified with the MPIMotionAttrFORWARD_ONLY attribute. See Reversal of the Master with NO_BACKTRACK_HOLD.

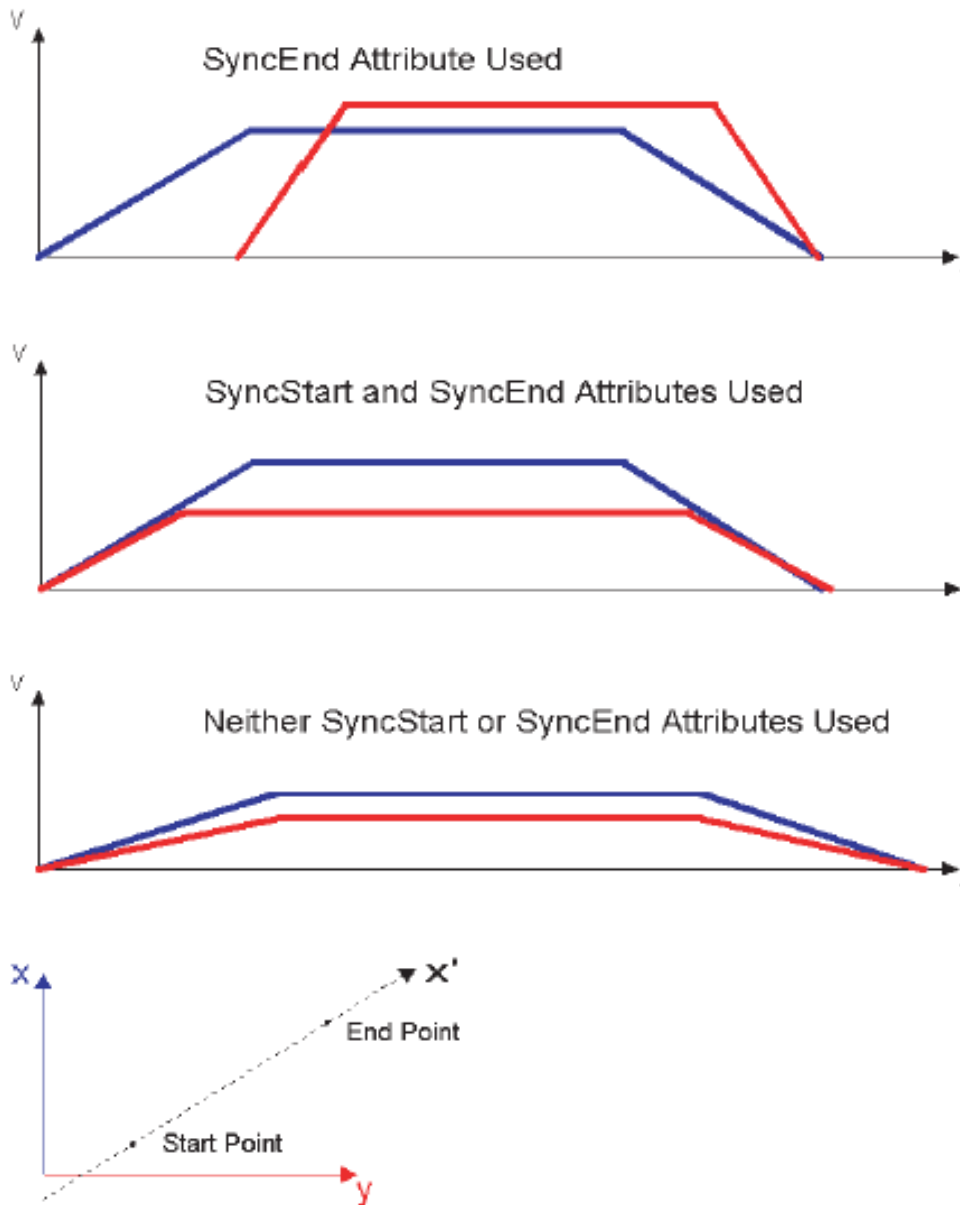
Remarks

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.

For the motion types MPIMotionTypeS_CURVE_JERK, MPIMotionTypeS_CURVE, MPIMotionTypeTRAPEZOIDAL, if neither MPIMotionAttrMaskSYNC_START nor MPIMotionAttrMaskSYNC_END are specified, then only one MPITrajectory structure will be used for by mpiMotionStart() and mpiMotionModify(). Please refer to MPIMotionAttr for more information.

Trajectory





MEIMotionAttrMask

```
typedef enum {
    MEIMotionAttrMaskeVENT,
    MEIMotionAttrMaskFINAL_VEL,
    MEIMotionAttrMaskNO_REVERSAL,
    MEIMotionAttrMaskHOLD,
    MEIMotionAttrMaskHOLD_LESS,
    MEIMotionAttrMaskHOLD_GREATER,
    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.
MEIMotionAttrMaskHOLD_LESS	Motion attribute mask for less than or equal hold logic.
MEIMotionAttrMaskHOLD_GREATER	Motion attribute mask for greater than or equal hold logic.
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    offMask;
    long    onMask;
    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.
offMask	This value specifies the bits to be turned OFF when MEIMotionAttrOutputTypeOFFMASK is used.
onMask	This value specifies the bits to be turned ON when MEIMotionAttrOutputTypeONMASK is used.
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 */
    long    masterStart; /* MPIMotionAttrMaskMASTER_START */
    long    repeatFrom; /* MPIMotionAttrMaskREPEAT */
} 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. The id is limited to 16-bit resolution by the controller firmware.
elementId	This array defines the identity for each element of a path motion. The elementId is limited to 16-bit resolution by the controller firmware.
masterStart	This field is only used with the MASTER_START motion attribute and when commanding a cam motion. This field defines the position of the master at the start of the cam motion. See Starting at Specific Master Positions .
repeatFrom	This field is only used with the REPEAT motion attribute and when commanding a cam motion. This field defines the first frame that is repeated. Any frames before this frame will act as a run-in sequence. See Repeating Cams .

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 previous 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 previous 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](#)

MPIMotionCam

MPIMotionCam

```
typedef struct MPIMotionDecelTime {
    long      pointCount;
    long      **slavePosition;
    long      *masterDistance;
    double    **gearRatio;
    /* This field is only used with MPIMotionTypeCUBIC_CAM. */
} MPIMotionDecelTime;
```

Description

MPIMotionType contains the cam segments for a linear or cubic interpolated cam move.

pointCount	This field defines the type of master position source is being used.
**slavePosition	This field defines the distance that each slave axis will move during each segment of the cam.
*masterDistance	An array containing pointCount number of elements. Each element is the distance the master axis will move during each segment of the cam.
**gearRatio	This field is only used with cubic cams, with linear cams this field is completely ignored and can be either be set to zero or not initialised.This field defines the gear ratio (first derivative of the slave position with respect to the master position) to the cam at each of the transitions between cam segments. The initial gear ration is assumed to be zero.

See Also

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,
    MPIMotionMessageIDLE,
    MPIMotionMessageMOVING,
    MPIMotionMessageSTOPPING,
    MPIMotionMessageSTOPPED,
    MPIMotionMessageSTOPPING_ERROR,
    MPIMotionMessageERROR,
    MPIMotionMessageAUTO_START,
    MPIMotionMessagePROFILE_ERROR,
    MPIMotionMessagePROFILE_NOT_SUPPORTED,
    MPIMotionMessagePATH_ERROR,
    MPIMotionMessageFRAMES_LOW,
    MPIMotionMessageFRAMES_EMPTY,
    MPIMotionMessageFRAME_BUFFER_TOO_SMALL,
} MPIMotionMessage;
```

Description

MPIMotionMessageMOTION_INVALID

The motion supervisor number is out of range. This message code is returned by `mpiMotionCreate(...)` if the motion supervisor number is less than zero or greater than or equal to `MEIXmpMAX_MSs`.

MPIMotionMessageAXIS_NOT_FOUND

The specified axis object is not available. This message is returned from [mpiMotionAxisRemove\(...\)](#) if the axis that is being removed is not a member of the motion object.

MPIMotionMessageAXIS_COUNT

The number of axes is out of range. This message is returned from [mpiMotionConfigSet\(...\)](#), [mpiMotionStart\(...\)](#), or [mpiMotionModify\(...\)](#) if there are no axes associated with the motion object or if the axis count exceeds `MEIXmpMAX_COORD_AXES`.

MPIMotionMessageTYPE_INVALID

The motion type or motion attribute is not valid. This message code is returned from [mpiMotionStart\(...\)](#) or [mpiMotionModify\(...\)](#) if the motion type or motion attribute mask is not recognized by the library. To correct the problem, select a motion type/attribute from the MPI and/or MEI enumerations.

MPIMotionMessageATTRIBUTE_INVALID

The motion attribute is not valid. This message code is returned from [mpiMotionStart\(...\)](#) or [mpiMotionModify\(...\)](#) if the motion attribute mask is not compatible with the specified motion type. To correct the problem, do not use the motion attribute mask with the specified motion type or select a different motion type.

Please see [possible causes](#) for receiving this message.

MPIMotionMessageIDLE

All motion supervisor axes are not moving and are ready to move. This message code is returned from [mpiMotionModify\(...\)](#) when the motion supervisor is in the IDLE state. A motion cannot be modified if no motion is in progress. To correct the problem, use the AUTO_START attribute for [mpiMotionModify\(...\)](#) or use [mpiMotionStart\(...\)](#) instead. This message code is also returned from [mpiMotionAction\(...\)](#) when a STOP or RESUME is commanded when the motion supervisor is in the IDLE state. A motion cannot be stopped if there is no motion in progress and a motion cannot be resumed if there is no motion profile pending.

MPIMotionMessageMOVING

At least one motion supervisor axis is moving. This message code is returned from [mpiMotionStart\(...\)](#) when the motion supervisor is in the MOVING state. A motion cannot be started if a motion is in progress. To correct the problem, use [mpiMotionModify\(...\)](#) instead. This message code is also returned from [mpiMotionAction\(...\)](#) when a RESUME or RESET is commanded when the motion supervisor is in the MOVING state. A motion cannot be resumed or reset while a motion is in progress.

MPIMotionMessageSTOPPING

Motion supervisor axes are stopping due to a STOP action. This message code is returned from [mpiMotionStart\(...\)](#) or [mpiMotionModify\(...\)](#) when the motion supervisor is in the STOPPING state. A motion cannot be commanded when a stop is in progress. This message code is also returned from [mpiMotionAction\(...\)](#) when a RESUME or RESET is commanded when the motion supervisor is in the STOPPING state. A motion cannot be resumed or reset while a stop is in progress.

MPIMotionMessageSTOPPED

Motion supervisor axes are stopped due to a STOP action. This message code is returned from [mpiMotionAction\(...\)](#) when a STOP action is commanded while the motion supervisor is already in the STOPPED state.

MPIMotionMessageSTOPPING_ERROR

Motion supervisor axes are stopping due to an E_STOP or ABORT action. This message code is returned from [mpiMotionStart\(...\)](#) or [mpiMotionModify\(...\)](#) when the motion supervisor is in the STOPPING_ERROR state. A motion cannot be commanded when a stopping on error action is in progress. This message code is also returned from [mpiMotionAction\(...\)](#) when a RESUME or RESET is commanded when the motion supervisor is in the STOPPING state. A motion cannot be resumed or reset while a stopping on error action is in progress.

MPIMotionMessageERROR

Motion supervisor axes are in an error state due to an E_STOP or ABORT action. This message code is returned from [mpiMotionStart\(...\)](#) or [mpiMotionModify\(...\)](#) when the motion supervisor is in the ERROR state. A motion cannot be commanded when the motion supervisor is in an error state. This message code is also returned from [mpiMotionAction\(...\)](#) when a STOP or RESUME is commanded when the motion supervisor is in the ERROR state. To correct the problem, fix the condition that caused the E_STOP or ABORT action and then clear the ERROR state using [mpiMotionAction\(...\)](#) with a RESET.

MPIMotionMessageAUTO_START

The motion modify was automatically converted to a motion start. This message code is returned from [mpiMotionModify\(...\)](#) when the AUTO_START attribute mask was specified and the motion was commanded while the motion supervisor is in the IDLE state. This message code is useful for notifying an application of an auto-start, it is not an error condition.

MPIMotionMessagePROFILE_ERROR

The motion profile is not possible with the specified constraints. This message code is returned by [mpiMotionModify\(...\)](#) when the NO_REVERSAL attribute mask is specified, but the specified target position is in the reverse direction. To correct the problem, either remove the NO_REVERSAL constraint or specify a target position that is in the same direction as the motion profile in progress.

MPIMotionMessagePROFILE_NOT_SUPPORTED

The controller firmware does not support the specified motion profile type. This message code is returned by [mpiMotionStart\(...\)](#) or [mpiMotionModify\(...\)](#) when an S_CURVE_JERK or VELOCITY_JERK motion type is commanded, but not supported by the controller firmware. Due to programming memory space constraints, specific revisions of controller firmware may not support jerk motion types. To correct the problem, use the S_CURVE or VELOCITY motion instead.

MPIMotionMessagePATH_ERROR

The specified multi-point motion path is not valid. This message code is returned by [mpiMotionStart\(...\)](#) or [mpiMotionModify\(...\)](#) when the final point is not specified or the time slice is less than one controller sample.

MPIMotionMessageFRAMES_LOW

The controller's frame buffer is low. This message code is returned by the internal method, `meiMotionFramesLow(...)`. This is an internal message code used by the library to manage the frame buffering.

MPIMotionMessageFRAMES_EMPTY

The controller's frame buffer is empty. This message code is returned by the internal method, `meiMotionFramesLow(...)`. This is an internal message code used by the library to manage the frame buffering.

MPIMotionMessageFRAME_BUFFER_TOO_SMALL

When trying to start a cam, an insufficient amount of space was found on the controller. To remedy this problem, you can either reduce the number of points within the cam (See [Caming](#)) or increase the number of points in the frame buffer. See [Increasing the Maximum Cam Table Size](#).

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	The motion object has no axes. This message code is returned by mpiMotionStart(...), mpiMotionModify(...) or mpiMotionAction(...) if there are no axes mapped to the motion object. To correct this problem, make sure there is at least one axis object associated with the motion object before commanding any motion or motion actions. Possible Causes: No axes are mapped to the motion supervisor that the move was commanded on.

See Also [MPIMotionType](#) | [MPIMotionAttrMask](#) | [mpiMotionModify](#) | [mpiMotionAction](#)
[mpiMotionStart](#)

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.

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

MEIMotionAttributes

MEIMotionAttrEVENT

The MPI has been extended to support `mpiMotionStart(...)` with the `MEIMotionAttrMaskEVENT` attribute. This mask allows the user to specify an `MPIEventMask` during a motion.

MEIMotionAttrFINAL_VEL

The MPI has been extended to support `mpiMotionStart(...)` with the `MEIMotionAttrMaskFINAL_VEL` attribute. This mask allows the user to specify a non-zero target velocity for point to point motion types.

MEIMotionAttrNO_REVERSAL

The MPI has been extended to support `mpiMotionStart(...)` with the `MEIMotionAttrMaskNO_REVERSAL` attribute. This mask prevents a motion profile from changing direction.

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 mask allows the user to set or clear bits during a motion. 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.

MPIMotionMessageATTRIBUTE_INVALID

Back to [MPIMotionMessage](#)

Possible causes for receiving an MPIMotionMessageATTRIBUTE_INVALID:

PT Moves are only permitted to have motion attributes from the following list:

- MPIMotionAttrMaskAPPEND
- MPIMotionAttrMaskRELATIVE
- MPIMotionAttrMaskID
- MPIMotionAttrMaskELEMENT_ID
- MEIMotionAttrMaskHOLD
- MEIMotionAttrMaskHOLD_LESS
- MEIMotionAttrMaskHOLD_GREATER
- MEIMotionAttrMaskOUTPUT

See Also: [MPIMotionAttrMask](#).

PTF Moves are only permitted to have motion attributes from the following list:

- MPIMotionAttrMaskAPPEND
- MPIMotionAttrMaskRELATIVE
- MPIMotionAttrMaskID
- MPIMotionAttrMaskELEMENT_ID
- MEIMotionAttrMaskHOLD
- MEIMotionAttrMaskHOLD_LESS
- MEIMotionAttrMaskHOLD_GREATER
- MEIMotionAttrMaskOUTPUT

See Also: [MPIMotionAttrMask](#).

PVT Moves are only permitted to have motion attributes from the following list:

MPIMotionAttrMaskAPPEND
MPIMotionAttrMaskRELATIVE
MPIMotionAttrMaskID
MPIMotionAttrMaskELEMENT_ID
MEIMotionAttrMaskHOLD
MEIMotionAttrMaskHOLD_LESS
MEIMotionAttrMaskHOLD_GREATER
MEIMotionAttrMaskOUTPUT

See Also: [MPIMotionAttrMask](#).

PVTF Moves are only permitted to have motion attributes from the following list:

MPIMotionAttrMaskAPPEND
MPIMotionAttrMaskRELATIVE
MPIMotionAttrMaskID
MPIMotionAttrMaskELEMENT_ID
MEIMotionAttrMaskHOLD
MEIMotionAttrMaskHOLD_LESS
MEIMotionAttrMaskHOLD_GREATER
MEIMotionAttrMaskOUTPUT

See Also: [MPIMotionAttrMask](#).

SPLINE Moves are only permitted to have motion attributes from the following list:

MPIMotionAttrMaskRELATIVE
MPIMotionAttrMaskID
MPIMotionAttrMaskELEMENT_ID
MEIMotionAttrMaskHOLD
MEIMotionAttrMaskHOLD_LESS
MEIMotionAttrMaskHOLD_GREATER
MEIMotionAttrMaskOUTPUT

See Also: [MPIMotionAttrMask](#).

BESSEL Moves are only permitted to have motion attributes from the following list:

MPIMotionAttrMaskAPPEND
 MPIMotionAttrMaskRELATIVE
 MPIMotionAttrMaskID
 MPIMotionAttrMaskELEMENT_ID
 MEIMotionAttrMaskHOLD
 MEIMotionAttrMaskHOLD_LESS
 MEIMotionAttrMaskHOLD_GREATER
 MEIMotionAttrMaskOUTPUT

See Also: [MPIMotionAttrMask](#).

BSPLINE, BSPLINE2 Moves are only permitted to have motion attributes from the following list:

MPIMotionAttrMaskAPPEND
 MPIMotionAttrMaskRELATIVE
 MPIMotionAttrMaskID
 MPIMotionAttrMaskELEMENT_ID
 MEIMotionAttrMaskHOLD
 MEIMotionAttrMaskHOLD_LESS
 MEIMotionAttrMaskHOLD_GREATER
 MEIMotionAttrMaskOUTPUT

See Also: [MPIMotionAttrMask](#).

VELOCITY, VELOCITY_JERK Moves are not permitted to have the following motion attributes:

MPIMotionAttrMaskSYNC_END
 MPIMotionAttrMaskSYNC_START
 MEIMotionAttrMaskFINAL_VEL
 MEIMotionAttrMaskNO_REVERSAL

See Also: [MPIMotionAttrMask](#).

FRAME Moves are only permitted to have motion attributes from the following list:

MPIMotionAttrMaskAPPEND
 MPIMotionAttrMaskID

See Also: [MPIMotionAttrMask](#).

DELAY attribute specified and attributes delay address is NULL.

FINAL_VEL attribute specified and attributes finalVelocity address is NULL.

HOLD, HOLD_LESS, or HOLD_GREATER attribute specified and attributes hold address is NULL.

OUTPUT attribute specified and attributes output address is NULL.

HOLD, HOLD_LESS, and HOLD_GREATER are not permitted during motion modifies.

Address of params passed to [mpiMotionStart](#) is NULL.

MPIMotionParams / MEIMotionParams

MPIMotionParams

```
typedef struct MPIMotionParams {

    MPIMotionPT          pt;
    MPIMotionPTF         ptf;
    MPIMotionPVT         pvt;
    MPIMotionPVTF        pvtf;
    MPIMotionSPLINE      spline;
    MPIMotionBESSEL      besse1;
    MPIMotionBSPLINE     bspline;

    MPIMotionSCurve      sCurve;
    MPIMotionSCurve      sCurveJerk;
    MPIMotionTrapezoidal trapezoidal;

    MPIMotionVelocity    velocity;
    MPIMotionVelocity    velocityJerk;

    MPIMotionCam         cam;

    MPIMotionAttributes   attributes;

    void                  *external;
} MPIMotionParams;
```

Description

The **Params** structure contains the motion trajectory parameters for each motion type and the motion attributes.

pt	This structure contains the parameters for a PT motion type. Please see MPIMotionPT data type for more information.
ptf	This structure contains the parameters for a PTF (position, time, and feedforward) motion type. Please see MPIMotionPTF data type for more information.
pvt	This structure contains the parameters for a PVT motion type. Please see MPIMotionPVT data type for more information.
pvtf	This structure contains the parameters for a PTF (position, velocity, time, and feedforward) motion type. Please see MPIMotionPVTF 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.
sCurveJerk	This structure contains the parameters for an S-Curve Jerk point to point motion type. Please see MPIMotionSCurveJerk data type for more information.
trapezoidal	This structure contains the parameters for a Trapezoidal point to point motion type. Please see MPIMotionTrapezoidal 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.
cam	This structure contains the parameters for a cam motion type. Please see MPIMotionCAM 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](#)

For more information, please see the [PT and PVT Path Motion for the XMP](#) application note.

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 previous 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](#) | [Streaming Point Motion Type Calculations](#)

For more information, please see the [PT and PVT Path Motion for the XMP](#) application note.

MPIMotionPTF

MPIMotionPTF

```
typedef struct MPIMotionPTF {
    long          pointCount;
    double        *position;
    double        *feedforward;
    double        *time;
    MPIMotionPoint point;
} MPIMotionPTF;
```

Description

The **MotionPTF** structure contains the motion parameters for the MPIMotionTypePTF.

The feedforward values are interpolated linearly over the PT or PVT time intervals. The feedforward values correspond to the P or PV values. (i.e. When the motion reaches a specified position (PTF) or position and velocity (PVTF), the interpolated feedforward value will be equal to what is specified in the motion parameters.)

The feedforward values are not set to zero at the beginning of the move; they retain the last value that is specified in the PTF or PVTF motion parameters.

The feedforward values are not zeroed by non-PTF or PVTF moves (i.e. PT, PVT, Spline, S-Curve, etc.).

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.
*feedforward	This array stores the feedforward values for the motion profile. There is one feedforward value per point, per axis. The length of the array must be equal to pointCount multiplied by the number of axes. The feedforward values are interleaved in the array by the axis index. The units are raw DAC counts (range - 32768 to +32767).
*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 previous 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

[MPIMotionParams](#) | [MPIMotionType](#) | [MPIMotionPoint](#) | [mpiMotionStart](#) | [mpiMotionModify](#)

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 previous 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](#) | [Streaming Point Motion Type Calculations](#)

For more information, please see the [PT and PVT Path Motion for the XMP](#) application note.

MPIMotionPVTF

MPIMotionPVTF

```
typedef struct MPIMotionPVTF {
    long          pointCount;
    double        *position;
    double        *velocity;
    double        *feedforward;
    double        *time;
    MPIMotionPoint point;
} MPIMotionPVTF;
```

Description

The **MotionPVTF** structure contains the motion parameters for the MPIMotionTypePVTF.

The feedforward values are interpolated linearly over the PT or PVT time intervals. The feedforward values correspond to the P or PV values. (i.e. When the motion reaches a specified position (PTF) or position and velocity (PVTF), the interpolated feedforward value will be equal to what is specified in the motion parameters.)

The feedforward values are not set to zero at the beginning of the move; they retain the last value that is specified in the PTF or PVTF motion parameters.

The feedforward values are not zeroed by non-PTF or PVTF moves (i.e. PT, PVT, Spline, S-Curve, etc.).

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.
*feedforward	This array stores the feedforward values for the motion profile. There is one feedforward value per point, per axis. The length of the array must be equal to pointCount multiplied by the number of axes. The feedforward values are interleaved in the array by the axis index. The units are raw DAC counts (range - 32768 to +32767).

*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 previous 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 [MPIMotionParams](#) | [MPIMotionType](#) | [MPIMotionPoint](#) | [mpiMotionStart](#) | [mpiMotionModify](#)

MPIMotionSCurve

MPIMotionSCurve

```
typedef struct MPIMotionSCurve {  
    MPITrajectory    *trajectory;  
    double          *position;  
} MPIMotionSCurve;
```

Description

The [SCurve](#) structure contains the motion trajectory parameters and final target positions that specify the point-to-point motion profile. Only the velocity, acceleration, deceleration, and jerkPercent trajectory parameters are applicable to S-Curve motion profiles.

*trajectory	A pointer to an array of trajectory structures. Each trajectory structure contains the motion profile parameters for each axis associated with the motion object. If more than one axis is associated with the motion and neither the MPIMotionAttrMaskSYNC_START nor MPIMotionAttrMaskSYNC_END attributes are specified, then only the first trajectory structure will be applied as the vector trajectory for all the axes.
*position	A pointer to an array of target positions. Each position value specifies the target for each axis associated with the motion object. Some motion types, like VELOCITY and VELOCITY_JERK do not require target positions and will ignore these values.

See Also

[MPITrajectory](#) | [MPIMotionType](#) | [mpiMotionStart](#) | [mpiMotionModify](#)

MPIMotionSCurveJerk

MPIMotionSCurveJerk

```
typedef MPIMotionSCurve MPIMotionSCurveJerk;
```

Description

The **SCurveJerk** structure contains the motion trajectory parameters and final target positions that specify the point-to-point motion profile. Only the velocity, acceleration, deceleration, accelerationJerk, and decelerationJerk trajectory parameters are applicable to S-Curve Jerk motion profiles.

See Also

[MPITrajectory](#) | [MPIMotionType](#) | [mpiMotionStart](#) | [mpiMotionModify](#)

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 previous 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 MPIMotionSCurve MPIMotionTrapezoidal;
```

Description

The **Trapezoidal** structure contains the motion trajectory parameters and final target positions that specify the point-to-point motion profile. Only the velocity, acceleration, and deceleration trajectory parameters are applicable to Trapezoidal motion profiles.

See Also

[MPITrajectory](#) | [MPIMotionType](#) | [mpiMotionStart](#) | [mpiMotionModify](#)

MPIMotionType / MEIMotionType

MPIMotionType

```
typedef enum {
    MPIMotionTypeINVALID,

    MPIMotionTypePT,
    MPIMotionTypePTF,
    MPIMotionTypePVT,
    MPIMotionTypePVTF,
    MPIMotionTypeSPLINE,
    MPIMotionTypeBESSEL,
    MPIMotionTypeBSPLINE,
    MPIMotionTypeBSPLINE2,

    MPIMotionTypeS_CURVE,
    MPIMotionTypeTRAPEZOIDAL,
    MPIMotionTypeS_CURVE_JERK,

    MPIMotionTypeVELOCITY,
    MPIMotionTypeVELOCITY_JERK,
    MPIMotionTypeCAM_LINEAR,
    MPIMotionTypeCAM_CUBIC,

    MPIMotionTypeMASK = 0xFF,
} MPIMotionType;
```

Description

MPIMotionType is an enumeration of motion profile types. It specifies the motion profile to be generated with `mpiMotionStart(...)` and `mpiMotionModify(...)`. The motion type also defines the trajectory parameters that must be passed to `mpiMotionStart(...)` and `mpiMotionModify(...)`. For each **MPIMotionType** and **MEIMotionType** there is a corresponding element in the **MPIMotionParams{...}** or **MEIMotionParams{...}** structure. For example, when **MPIMotionTypeS_CURVE** is specified, the **MPIMotionSCurve{...}** structure must be filled in to specify the trajectory for the S-Curve profile.

The motion profile characteristics can be specified by bitwise OR-ing the **MPIMotionType** with one or more motion attribute masks. The **MPIMotionAttrMask** and **MEIMotionAttrMask** enumerations contain the attribute masks. Attribute masks can be used to enable special features or configure the motion profile calculation and execution properties.

Internally, the motion profile is calculated and broken up into smaller trajectory segments called frames. The frames contain position, time, velocity, acceleration, and jerk trajectory information. The controller buffers the frames in its memory and executes them by loading the values into its trajectory calculator. For simple MotionTypes, like **VELOCITY**, **TRAPEZOIDAL**, and **S_CURVE**, the frames are calculated by the controller. For complex, multi-point motions, like **PT**, **PVT**, **SPLINE**, etc. the frames are calculated in the MPI Library. MotionTypes that are

calculated in the controller can be modified on-the-fly with `mpiMotionModify(...)`.

The move types, `MPIMotionTypeS_CURVE_JERK` and `MPIMotionTypeVELOCITY_JERK` are only available with custom firmware (option 21). In the custom firmware, the Velocity/Trapezoidal/S-Curve algorithm is replaced with the S-Curve Jerk algorithm. In the standard firmware, the `MPIMotionTypeS_CURVE_JERK` and `MPIMotionTypeVELOCITY_JERK` motion types are not supported and if specified, the MPI will return a "profile not supported" error.

For details, see [S-Curve Jerk Algorithm and Attributes](#).

MPIMotionTypePT	This type fits constant velocity profile segments through a list of position and time points. Not supported in motion sequences. Please see MPIMotionPT for more information.
MPIMotionTypePTF	PTF motion is identical to PT except host-calculated feedforward values can be specified for each PTF motion segment.
MPIMotionTypePVT	This type fits jerk profile segments through a list of position, velocity and time points. Not supported in motion sequences. Please see MPIMotionPVT for more information.
MPIMotionTypePVTF	PVTF motion is identical to PVT except host-calculated feedforward values can be specified for each PVTF motion segment.
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.
MPIMotionTypeCAM_LINEAR	This type generates linear interpolated cam motion. See Camming .
MPIMotionTypeCAM_CUBIC	This type generates cubic interpolated cam motion. See Camming .

MEIMotionType

```
typedef enum {  
    MEIMotionTypeFRAME,  
} MEIMotionType;
```

Description **MEIMotionType** is an enumeration of the controller specific motion profile types. See [MPIMotionType](#) for details.

MEIMotionTypeFRAME	This motion type is used to construct motion profiles at the frame level.

See Also [mpiMotionStart](#) | [mpiMotionModify](#) | [mpiMotionTYPE](#) | [MPIMotionParams](#) | [MEIMotionParams](#) | [MPIMotionAttr](#) | [MEIMotionAttr](#) | [Streaming Point Motion Type Calculations](#)

Streaming Point Motion Type Calculations

Overview

This section is intended to provide information about how one can determine the paths (equations of motion) that will be produced by the various streaming point motion types. These equations can be used to answer several important questions such as:

- Will a certain motion cause a collision?
- What will the maximum acceleration be?

Since all motion is specified by constant-jerk segments of motion, it is fairly easy to calculate the maximum acceleration for any given coordinate. Since all motion segments will have accelerations that change linearly with time, simply calculate the maximum accelerations at the beginning and end of every segment. Please note that accelerations are not continuous for all motion types, so one must make two comparisons for each segment. If you are working with a multi-axis system, then this rule will also apply to a system with Cartesian axes. Although, this may not be true for non-Cartesian coordinate systems.

There is one important note about the time-delta arrays that the MEI motion types accept. Any given dt defining a point also species how long it will take to arrive at that point:

$$dt_i = (\text{time of point } x_i) - (\text{time of point } x_{i-1})$$

PT Moves - **MPIMotionTypePT**

This is the most simple streaming point algorithm. This motion type will interpolate all points using constant velocity segments. The most important consequence of this motion type is that at each point defining the move, there is an instantaneous change in velocity. Therefore, one should only use PT motion when streaming points frequently—once every few servo samples. This will require several hundred, perhaps thousands of points to define a single move. If moves are to be defined using only a few points, then the other streaming point motion types should be used.

PVT Moves - **MPIMotionTypePVT**

PVT is the most flexible motion type because one can define both position and velocity at any given time. PVT motion guarantees continuity of velocity, but may have discontinuities in acceleration. The drawback to PVT motion is that velocities must be specified. There are a couple of major advantages of PVT motion:

- The equations of motion for any given segment are defined by the two points at the ends of the segment.
- PVT moves are easily appended to, without any major consequences, such as a discontinuity in velocity that are evident in some of the spline motion types.

Given x_i , v_i , and dt_i , and requiring that the velocity be continuous, it is simple to calculate the equations for motion at any given interval between the specified PVT points.

$$\begin{array}{ll}
 x_{0i} = x_i & x_0 = x \\
 v_{0i} = v_i & v_0 = v \\
 a_{0i} = 6 \cdot \left(\frac{x_{i+1} - x_i}{dt_{i+1}^2} \right) - \left(\frac{2 \cdot v_{i+1} + 4 \cdot v_i}{dt_{i+1}} \right) & \text{or} \quad a_0 = 6 \cdot \frac{dx}{dt^2} - \left(\frac{6 \cdot v + 2 \cdot dv}{dt} \right) \\
 j = -12 \cdot \left(\frac{x_{i+1} - x_i}{dt_{i+1}^3} \right) + 6 \cdot \left(\frac{v_{i+1} + v_i}{dt_{i+1}^2} \right) & j = -12 \cdot \frac{dx}{dt^3} + \left(\frac{12 \cdot v + 6 \cdot dv}{dt^2} \right)
 \end{array}$$

Cubic Spline Moves - MPIMotionTypeSPLINE

The MPIMotionTypeSPLINE generates a "natural" cubic spline.

- A cubic spline will pass through each point defining the spline.
- The acceleration at the beginning and end of the motion is zero.
- The velocity of a cubic spline may not be zero at the beginning and end of the motion.
- The equations for motion of any segment depends on all points defining the spline.

Some important consequences of these characteristics are:

- When using cubic spline motion, at least two closely spaced points should be used at the beginning and at the end of a spline to help define the starting and ending velocity.
- Because the end-point velocities are not intuitive, there may be instantaneous changes in velocity when starting and/or appending new spline segments.

An algorithm that gives identical results to MEI's internal algorithm is given by Numerical Recipes. [Numerical Recipes for C section 3.3](#) lists a function called spline() which outputs identical results to MEI's algorithm.

PLEASE NOTE:

Numerical Recipes for C assumes one-based arrays rather than the typical zero-based arrays usually used in C. spline() takes x_i and t_i and returns a_i for all points used to define the spline. From here we can solve some algebraic equations to find v_i , j_i , dt_i , and t_i .

$$\begin{array}{ll}
 x_{0i} = x_i & \\
 v_{0i} = \left(\frac{x_{i+1} - x_i}{dt_{i+1}} \right) - \frac{1}{6} \cdot (a_{i+1} + 2 \cdot v_i) \cdot dt_{i+1} & \\
 a_{0i} = a_i & \text{or} \\
 j = \left(\frac{a_{i+1} - a_i}{dt_{i+1}} \right) & \\
 & \begin{array}{l}
 x_0 = x \\
 v_0 = \frac{dx}{dt} - \frac{1}{6} \cdot (3a + da) \cdot dt \\
 a_0 = a \\
 j = \frac{da}{dt}
 \end{array}
 \end{array}$$

MPIMotionVelocity

MPIMotionVelocity

```
typedef struct MPIMotionVelocity {
    MPITrajectory      *trajectory;
} MPIMotionVelocity;
```

Description

The **Velocity** structure contains the motion trajectory parameters that specify velocity motion profiles. Only the velocity, acceleration, jerkPercent, and accelerationJerk trajectory parameters are applicable to velocity and velocity-jerk motion profiles.

***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](#) | [MPIMotionType](#) | [mpiMotionStart](#) | [mpiMotionModify](#)

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](#)

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

<u>mpiMotorCreate</u>	Create Motor object
<u>mpiMotorDelete</u>	Delete Motor object
<u>mpiMotorValidate</u>	Validate Motor object

Configuration and Information Methods

<u>mpiMotorAmpEnableGet</u>	Get state of amp enable output
<u>mpiMotorAmpEnableSet</u>	Set state of amp enable output
<u>meiMotorAmpFault</u>	Writes the amp fault buffer data of a motor.
<u>meiMotorAmpFaultClear</u>	Clears the amp fault buffer data of a motor.
<u>meiMotorAmpWarning</u>	Writes the amp warning buffer of the motor.
<u>meiMotorAmpWarningClear</u>	Clears the motor's amp warning message buffer.
<u>mpiMotorAxisMapGet</u>	Get object map of axes
<u>meiMotorCommutationModeGet</u>	Gets the commutation mode of a motor.
<u>meiMotorCommutationModeSet</u>	Sets the commutation mode of a motor.
<u>mpiMotorConfigGet</u>	Get motor configuration
<u>mpiMotorConfigSet</u>	Set motor configuration
<u>meiMotorConfigStepper</u>	Configures a motor for stepper mode.
<u>meiMotorDacConfigGet</u>	Get a Motor's (motor) Dac configuration
<u>meiMotorDacConfigSet</u>	Set a Motor's (motor) Dac configuration
<u>meiMotorDacFlashConfigGet</u>	
<u>meiMotorDacFlashConfigSet</u>	
<u>mpiMotorFeedback</u>	Get feedback position

[mpiMotorFlashConfigGet](#)

Get flash config of motor

[mpiMotorFlashConfigSet](#)

Set flash config of motor

[meiMotorInfo](#)

Get information about the network, node, and drive interface

[mpiMotorIoGet](#)

Get dedicated I/O bits

[mpiMotorIoSet](#)

Set dedicated I/O bits

[mpiMotorStatus](#) / [meiMotorStatus](#)

Get motor status

Event Methods

[mpiMotorEventConfigGet](#)

Get motor's event configuration

[mpiMotorEventConfigSet](#)

Set motor's event configuration

[mpiMotorEventNotifyGet](#)

Get motor's event mask for host notification.

[mpiMotorEventNotifySet](#)

Set motor's event mask for host notification.

[mpiMotorEventReset](#)

Reset events specified in event mask

Memory Methods

[mpiMotorMemory](#)

Get address of motor memory

[mpiMotorMemoryGet](#)

Copy motor memory to application memory

[mpiMotorMemorySet](#)

Copy application memory to motor memory

Action Methods

[meiMotorEncoderReset](#)

Clears encoder faults.

[meiMotorMultiTurnReset](#)

Clears the SynqNet drive multi-turn data for absolute type encoders.

Relational Methods

[mpiMotorControl](#)

Get handle to associated Control object

[mpiMotorFilterMapGet](#)

Get object map of associated Filters

[mpiMotorFilterMapSet](#)

Set the Filters using object map

[mpiMotorNumber](#)

Get index number of motor (in Control list)

Other Methods

[meiMotorCompareListGet](#)

[meiMotorEncoderRatio](#)

Get encoder ratio from the XMP.

Data Types

[MEIMotorAmpFaults](#)

[MEIMotorAmpFaultMsg](#)

[MEIMotorAmpWarnings](#)

[MEIMotorAmpWarningMsg](#)

[MPIMotorBrake](#)

[MPIMotorBrakeMode](#)

[MPIMotorConfig](#) / [MEIMotorConfig](#)

[MEIMotorDacConfig](#)

[MEIMotorDacChannelConfig](#)

[MEIMotorDacChannelStatus](#)

[MEIMotorDacStatus](#)

[MEIMotorDedicatedIn](#)

[MEIMotorDedicatedOut](#)

[MEIMotorDisableAction](#)

[MPIMotorEncoder](#) / [MEIMotorEncoder](#)

[MPIMotorEncoderFault](#)

[MPIMotorEncoderFaultMask](#)

[MEIMotorEncoderRatio](#)

[MEIMotorEncoderReverseModulo](#)

[MEIMotorEncoderType](#)

[MPIMotorEventConfig](#) / [MEIMotorEventConfig](#)

[MPIMotorEventTrigger](#)

[MEIMotorFaultBit](#)

[MEIMotorFaultConfig](#)

[MEIMotorFaultMask](#)

[MPIMotorFeedback](#)

[MEIMotorInfo](#)

[MEIMotorInfoNodeType](#)

[MPIMotorIo](#)

[MEIMotorIoConfig](#)

[MEIMotorIoConfigIndex](#)

[MEIMotorIoMask](#)

[MEIMotorIoType](#)

[MPIMotorMessage](#) / [MEIMotorMessage](#)

[MEIMotorStatus](#)

[MEIMotorStatusOutput](#)

[MEIMotorStepper](#)

[MEIMotorStepperPulse](#)

[MEIMotorStepperPulseType](#)

[MEIMotorStepperStatus](#)

[MPIMotorType](#)

Macros

[mpiMotorEncoderFaultMaskBIT](#)

Constants

[MEIMotorAmpMsgMAX](#)

[MEIMotorAmpFaultsMAX](#)

[MEIMotorAmpWarningsMAX](#)

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 <i>MotorDelete</i> 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](#)

Required Header

file:///D:/pdfs/030100/html/Software-MPI/docs/Motor/Method/ampenbget1.htm [7/23/2004 9:09:44 AM]

mpiMotorAmpEnableSet

Declaration

```
long mpiMotorAmpEnableSet(MPIMotor motor,  
                           long ampEnable)
```

Required Header

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.

<i>If "ampEnable" is</i>	<i>Then</i>
FALSE (0)	the amp will be disabled
TRUE (1)	the amp will be enabled

Return Values

MPIMessageOK	if <i>MotorAmpEnableSet</i> successfully sets the Motor's amp enable output state to ampEnable

See Also [MPIMotorConfig](#) | [mpiMotorAmpEnableGet](#)

meiMotorAmpFault

[illegible]

Required Header

Description	MotorAmpFault reads the amp fault buffer from a motor and writes the data into a structure pointed to by fault.

motor	a handle to the Motor object
*fault	a pointer to a structure containing the number of amp faults, their coded values and message strings. See MEIMotorAmpFaults .

Return Values

MPIMessageOK	if <i>MotorAmpFault</i> writes the amp fault buffer data

See Also [meiMotorAmpFaultClear](#) | [meiMotorAmpWarning](#) | [meiMotorWarningClear](#)

meiMotorAmpFaultClear

Declaration long **meiMotorAmpFaultClear**([MPIMotor](#) **motor**);

Required Header stdmei.h

Description **MotorAmpFaultClear** flushes the motor's amp fault message buffer. The number of amp faults is set to zero, the coded values are set to zero, and the messages are cleared.

motor	a handle to the Motor object

Return Values

MPIMessageOK	if <i>MotorAmpFaultClear</i> successfully flushes the Motor's amp fault message buffer.

See Also [meiMotorAmpFault](#) | [meiMotorAmpWarning](#) | [meiMotorWarningClear](#)

Required Header

motor	a handle to the Motor object
*warning	a pointer to a structure containing the number of amp warnings, their coded values and message strings. See MEIMotorAmpWarnings .

MPIMessageOK if *MotorAmpWarning* successfully writes the amp warning buffer data

file:///D:/pdfs/030100/html/Software-MPI/docs/Motor/Method/ampwarn2.htm [7/23/2004 9:09:45 AM]

meiMotorAmpWarningClear

Declaration long **meiMotorAmpWarningClear**([MPIMotor](#) **motor**);

Required Header stdmei.h

Description **MotorAmpWarningClear** flushes the motor's amp warning message buffer. The number of amp warnings is set to zero, the coded values are set to zero, and the messages are cleared.

motor	a handle to the Motor object

Return Values

MPIMessageOK	if <i>MotorAmpWarningClear</i> successfully ...

See Also [meiMotorAmpWarning](#) | [meiMotorAmpFault](#) | [meiMotorAmpFaultClear](#)

mpiMotorAxisMapGet

[illegible]

Required Header

Description	MotorAxisMapGet gets the object map of the Axes associated with a Motor (motor) and writes it into the structure pointed to by axismap .

Return Values

MPIMessageOK	if <i>MotorAxisMapGet</i> successfully writes the Motor's object map of Axes to the structure

See Also

Required Header

Return Values

See Also [meiMotorCommutationModeSet](#)

Required Header `stdmei.h`

Return Values

See Also [meiMotorCommutationModeGet](#)

mpiMotorConfigGet

```

Declaration
long mpiMotorConfigGet(MPIMotor motor,
                       MPIMotorConfig *config,
                       void *external)

```

Required Header

Description	MotorConfigGet gets a Motor's (<i>motor</i>) configuration and writes it into the structure pointed to by <i>config</i> , and also writes it into the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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 <i>MotorConfigGet</i> successfully writes the Motor's configuration to the structure(s)

See Also [MEIMotorConfig](#) | [mpiMotorConfigSet](#)

```
long mpiMotorConfigSet(MPIMotor motor,  
                      MPIMotorConfig *config,  
                      void *external)
```

Required Header

Description	MotorConfigSet sets a Motor's (<i>motor</i>) configuration using data from the structure pointed to by <i>config</i> , and also using data from the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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).

external either points to a structure of type **MEIMotorConfig** or is NULL.

Return Values

MPIMessageOK	if <i>MotorConfigSet</i> successfully sets the Motor's configuration using data from the structure(s)

See Also [mpiMotorConfigGet](#) | [MEIMotorConfig](#)
Special Note: Using mpiMotorConfigSet with Absolute Encoders

```
Declaration      long  meiMotorConfigStepper( MPIMotor      motor,
                                   MEIMotorConfig *config,
                                   long      stepperNumber )
```

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

MPIMessageOK	if <i>MotorConfigStepper</i> successfully modifies the motor configuration structure pointed to by <i>config</i>

file:///D:/pdfs/030100/html/Software-MPI/docs/Motor/Method/cfspr2.htm [7/23/2004 9:09:48 AM]

meiMotorDacConfigGet

[illegible]

Required Header `stdmei.h`

Description	MotorDacConfigGet gets a Motor's (<i>motor</i>) DAC configuration and writes it to the structure pointed to by <i>config</i> .

motor	a handle to the Motor object.
*config	a pointer to a MEIMotorDacConfig structure.

Return Values

MPIMessageOK	if <i>MotorDacConfigGet</i> successfully writes the Motor's Dac configuration to the <i>config</i> structure.

See Also [meiMotorDacConfigSet](#) | [MEIMotorDacConfig](#)

meiMotorDacConfigSet

[illegible]

Required Header `stdmei.h`

Description	MotorDacConfigSet configures a Motor's (<i>motor</i>) DAC using data from the structure pointed to by <i>config</i> .

motor	a handle to the Motor object.
*config	a pointer to a MEIMotorDacConfig structure.

Return Values

MPIMessageOK	if <i>MotorDacConfigSet</i> successfully writes the Motor's Dac configuration to the controller.

See Also [meiMotorDacConfigGet](#) | [MEIMotorDacConfig](#)

Required Header `stdmei.h`

file:///D:/pdfs/030100/html/Software-MPI/docs/Motor/Method/dacflacfgget2.htm [7/23/2004 9:09:49 AM]

meiMotorDacFlashConfigSet

[illegible]

Required Header `stdmei.h`

Description **MotorDacFlashConfigSet** sets a Motor's (*motor*) DAC configuration to flash memory using data from the structure pointed to by *config*.

motor	a handle to the Motor object
*flash	<i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally.
*config	a pointer to a MEIMotorDacConfig structure.

Return Values

MPIMessageOK	if <i>MotorDacFlashConfigSet</i> successfully writes the DAC's configuration to flash memory using data from the <i>config</i> structure.

See Also [meiMotorDacFlashConfigGet](#)

mpiMotorFeedback

[illegible]

Required Header

Description	MotorFeedback gets the feedback position of a Motor (<i>motor</i>) and writes it into the location pointed to by <i>feedback</i> .

Return Values

MPIMessageOK	if <i>MotorFeedback</i> 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 <i>MotorFlashConfigGet</i> successfully writes the Motor’s flash configuration to the structure(s) <i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> 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 <i>MotorFlashConfigSet</i> successfully sets the Motor’s flash configuration using data from the structure(s) <i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally.
MEIFlashMessageNETWORK_TOPOLOGY_ERROR	returned because the SynqNet topology has not yet been saved to flash using meiSynqNetFlashTopologySave(...).

See Also

[MEIMotorConfig](#) | [MEIFlash](#) | [mpiMotorFlashConfigGet](#) | [meiSynqNetFlashTopologySave](#)

Required Header

motor	a handle to the Motor object
*info	a pointer to a motor information structure

MPIMessageOK	if <i>MotorInfo</i> successfully writes the network and node/drive interface information into the Motor structure.

file:///D:/pdfs/030100/html/Software-MPI/docs/Motor/Method/inf2.htm [7/23/2004 9:09:50 AM]

```
long mpiMotorIoGet(MPIMotor motor,  
                  MPIMotorIo *io)
```

Required Header

Description **MotorIoGet** gets a Motor's (*motor*) dedicated I/O bits and writes them into the structure pointed to by *io*.

NOTE

When using I/O on SynqNet nodes, use the motor I/O masks in the drive's header file for clearer code. Most SynqNet nodes have a header file that defines things that are specific to that drive. The header files are found in C:\mei\XMP\sqNodeLib\include. An example is shown below that reads the hall sensors from the Trust TA802.

```

/*
    Shows the hall sensors.
    Make sure you include trust_ta800.h. Ex:
    #include "C:\mei\XMP\sqNodeLib\include\trust_ta800.h"

    The hall sensor masks are found in the enum TA800MotorIoMask.
*/

void showHalls(MPIMotor motor)
{
    MPIMotorIo io;
    long returnValue;
    long a, b, c;

    while (meiPlatformKey(MPIWaitPOLL) <= 0)
    {
        returnValue = mpiMotorIoGet(motor, &io);
        msgCHECK(returnValue);

        a = ((io.input & TA800MotorIoMaskHALL_A) == TA800MotorIoMaskHALL_A);
        b = ((io.input & TA800MotorIoMaskHALL_B) == TA800MotorIoMaskHALL_B);
        c = ((io.input & TA800MotorIoMaskHALL_C) == TA800MotorIoMaskHALL_C);

        /* Prints a 1 or 0 indicating the hall state */
        printf("Hall A %d B %d C %d\t\t\r", a, b, c);

        meiPlatformSleep(100);
    }
}

```

Sample Code

```

/* Poll io inputs for a motor and print to the screen */
void readIo(MPIMotor motor)
{
    MPIMotorIo io;
    long returnValue;

    while (meiPlatformKey(MPIWaitPOLL) <= 0)
    {
        returnValue = mpiMotorIoGet(motor, &io);
        msgCHECK(returnValue);

        printf("\rIO %x", io.input);

        meiPlatformSleep(100); /* Wait 100 mSec */
    }
}

```

Return Values

MPIMessageOK	if <i>MotorIoGet</i> 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*.

NOTE

When using I/O on SynqNet nodes, use the motor I/O masks in the drive's header file for clearer code. Most SynqNet nodes have a header file that defines things that are specific to that drive. The header files are found in C:\mei\XMP\sqNodeLib\include. An example is shown below that reads the hall sensors from the Trust TA802.

```
/*
    Shows the hall sensors.
    Make sure you include trust_ta800.h. Ex:
    #include "C:\mei\XMP\sqNodeLib\include\trust_ta800.h"

    The hall sensor masks are found in the enum TA800MotorIoMask.
*/

void showHalls(MPIMotor motor)
{
    MPIMotorIo io;
    long returnValue;
    long a, b, c;

    while (meiPlatformKey(MPIWaitPOLL) <= 0)
    {
        returnValue = mpiMotorIoGet(motor, &io);
        msgCHECK(returnValue);

        a = ((io.input & TA800MotorIoMaskHALL_A) == TA800MotorIoMaskHALL_A);
        b = ((io.input & TA800MotorIoMaskHALL_B) == TA800MotorIoMaskHALL_B);
        c = ((io.input & TA800MotorIoMaskHALL_C) == TA800MotorIoMaskHALL_C);

        /* Prints a 1 or 0 indicating the hall state */
        printf("Hall A %d B %d C %d\t\t\r", a, b, c);

        meiPlatformSleep(100);
    }
}
```

Return Values

MPIMessageOK	if <i>MotorIoSet</i> successfully sets the Motor's dedicated I/O bits using data from the structure

See Also [mpiMotorIoGet](#)

mpiMotorStatus

```
long mpiMotorStatus(MPIMotor motor,  
                   MPIStatus *status,  
                   void *external)
```

Required Header

Description	MotorStatus writes a Motor's (<i>motor</i>) status into the structure pointed to by <i>status</i> , and also into the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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 **external** can be NULL (but status must not be NULL).

motor	a handle to the Motor object
*status	a pointer to the motor status structure returned by the method
*external	a pointer to an implementation-specific structure

XMP Only

external either points to a structure of type `MEIMotorStatus{...}` or is `NULL`.

Return Values

MPIMessageOK	if <i>MotorStatus</i> successfully gets the status of a Motor object.
MPIMessageARG_INVALID	if the <i>status</i> pointer is NULL.

See Also [MPIStatus](#)

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).

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 *external* can be NULL (but status must not be 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	pointer to an implementation-specific structure

XMP Only

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

Return Values

MPIMessageOK	if <i>MotorStatus</i> successfully gets the status of a Motor object.
MPIMessageARG_INVALID	if the <i>status</i> pointer is NULL.

See Also [MPIStatus](#) | [MEIMotorStatus](#)

mpiMotorEventConfigGet

Declaration

```
long mpiMotorEventConfigGet(MPIMotor
                           MPIEventType
                           MPIMotorEventConfig
                           void
                           motor,
                           eventType,
                           *eventConfig,
                           *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.

Sample Code

```
for(index = 0; index < AXIS_COUNT; index++)
{ // turn off error limit and limit switch actions for
  motors 0 to AXIS_COUNT
    returnValue = mpiMotorEventConfigGet(motor[index],
                                          MPIEventTypeLIMIT_ERROR, &eventConfig, NULL);
    msgCHECK(returnValue);

    eventConfig.action = MPIActionNONE;

    returnValue = mpiMotorEventConfigSet(motor[index],
                                          MPIEventTypeLIMIT_ERROR, &eventConfig, NULL);
    msgCHECK(returnValue);

    returnValue = mpiMotorEventConfigGet(motor[index],
                                          MPIEventTypeLIMIT_HW_NEG, &eventConfig, NULL);
    msgCHECK(returnValue);

    eventConfig.action = MPIActionNONE;

    returnValue = mpiMotorEventConfigSet(motor[index],
                                          MPIEventTypeLIMIT_HW_NEG, &eventConfig, NULL);
    msgCHECK(returnValue);

    returnValue = mpiMotorEventConfigGet(motor[index],
```

```
        MPIEventTypeLIMIT_HW_POS, &eventConfig, NULL);
msgCHECK(returnValue);

eventConfig.action = MPIActionNONE;

returnValue = mpiMotorEventConfigSet(motor[index],
        MPIEventTypeLIMIT_HW_POS, &eventConfig, NULL);
msgCHECK(returnValue);
}
```

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](#) reads the structure pointed to by *eventConfig* and sets the Motor’s (*motor*) configuration for the event specified by *eventType*.

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 <i>MotorEventConfigGet</i> 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 <i>eventmask</i> to MPIEventMaskALL
disable host notification of all events	set <i>eventmask</i> to MPIEventTypeNONE

Sample Code

```

for(index = 0; index < AXIS_COUNT; index++)
{ // turn off error limit and limit switch actions for
  motors 0 to AXIS_COUNT
    returnValue = mpiMotorEventConfigGet(motor[index],
      MPIEventTypeLIMIT_ERROR, &eventConfig, NULL);
    msgCHECK(returnValue);

    eventConfig.action = MPIActionNONE;

    returnValue = mpiMotorEventConfigSet(motor[index],
      MPIEventTypeLIMIT_ERROR, &eventConfig, NULL);
    msgCHECK(returnValue);

    returnValue = mpiMotorEventConfigGet(motor[index],
      MPIEventTypeLIMIT_HW_NEG, &eventConfig, NULL);
    msgCHECK(returnValue);

    eventConfig.action = MPIActionNONE;

    returnValue = mpiMotorEventConfigSet(motor[index],
      MPIEventTypeLIMIT_HW_NEG, &eventConfig, NULL);
    msgCHECK(returnValue);

    returnValue = mpiMotorEventConfigGet(motor[index],
      MPIEventTypeLIMIT_HW_POS, &eventConfig, NULL);
    msgCHECK(returnValue);

    eventConfig.action = MPIActionNONE;

    returnValue = mpiMotorEventConfigSet(motor[index],
      MPIEventTypeLIMIT_HW_POS, &eventConfig, NULL);
    msgCHECK(returnValue);
  }

```

Return Values

MPIMessageOK	if <i>MotorEventNotifySet</i> successfully requests host notification of the event(s) that are specified by <i>eventMask</i> and generated by <i>motor</i>

See Also [MPIEventType](#) | [MEIEventNotifyData](#) | [MEIEventStatusInfo](#)
[mpiMotorEventNotifyGet](#)

Required Header

Return Values

See Also

mpiMotorMemory

Declaration long `mpiMotorMemory`([MPIMotor](#) `motor`,
 void `**memory`)

Required Header `stdmpi.h`

Description [MotorMemory](#) sets (writes) an address (used to access a Control object's memory) to the contents of *memory*.

Return Values

MPIMessageOK if *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 <i>MotorMemoryGet</i> 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](#)

meiMotorEncoderReset

Declaration long `meiMotorEncoderReset`([MPIMotor](#) motor)

Required Header `stdmei.h`

Description [MotorEncoderReset](#) clears the encoder fault status registers for the primary and secondary encoder associated with the motor object.

Return Values

MPIMessageOK	if <i>MotorEncoderReset(...)</i> successfully resets the motor's encoder value.

See Also

meiMotorMultiTurnReset

Declaration long `meiMotorMultiTurnReset`([MPIMotor](#) `motor`);

Required Header stdmei.h

Description `MotorMultiTurnReset` clears the SynqNet drive multi-turn data for absolute type encoders. This is only needed when configuring the zero location for an absolute encoder or when the absolute encoder's battery is replaced. `meiMotorMultiTurnReset` is an offline operation. Make sure all motors (amp enables) are disabled before executing a multi-turn reset. The SynqNet network may be shutdown (dropped from SYNQ mode) due to specific drive limitations.

Not all SynqNet drives support or require this feature. Please see the drive manufacturer's documentation for details.

<code>motor</code>	a handle to the Motor object

Return Values	
<code>MPIMessageOK</code>	if <i>MotorMultiTurnReset</i> successfully ...

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 *filtermap)
```

Required Header

Description	MotorFilterMapGet gets the object map of the Filters [that are associated with a Motor (<i>motor</i>)] and writes it into the structure pointed to by <i>filtermap</i> .

Return Values

MPIMessageOK	if <i>MotorFilterMapGet</i> successfully writes the Filter's object map (associated with a Motor) to the structure

See Also [mpiMotorFilterMapSet](#)

mpiMotorFilterMapSet

[illegible]

Required Header

Description	MotorFilterMapSet sets the Filters [that are associated with a Motor (<i>motor</i>)], using data from the object map specified by <i>filtermap</i> .

Return Values

MPIMessageOK	if <i>MotorFilterMapSet</i> successfully sets the Filters using data from the object map

See Also [mpiMotorFilterMapGet](#)

Required Header

Return Values

See Also

meiMotorCompareListGet

[illegible]

Required Header `stdmei.h`

Description	MotorCompareListGet sets <i>compareCount</i> to the number of compares that <i>motor</i> has. It sets <i>compareList</i> to the number of the compare object for each compare.

Return Values

MPIMessageOK	if <i>motor</i> is a valid MPIMotor object.

See Also

Required Header

WARNING: This is a customer-specific method that is only supported with custom firmware. To inquire about using this method, please contact an MEI Applications Engineer.

MPIMessageOK	if <i>meiMotorEncoderRatio</i> successfully gets encoder ration from the XMP.
MPIMessageARG_INVALID	if arguments are not valid

file:///D:/pdfs/030100/html/Software-MPI/docs/Motor/Method/encrto2.htm [7/23/2004 9:09:57 AM]

MEIMotorAmpFaults

MEIMotorAmpFaults

```
typedef struct MEIMotorAmpFaults {
    long          count;
    long          code[MEIMotorAmpFaultsMAX];
    MEIMotorAmpFaultMsg message[MEIMotorAmpFaultsMAX];
} MEIMotorAmpFaults;
```

Description

The **MotorAmpFaults** structure contains the amp fault information from a SynqNet drive. The amp fault messages are drive specific. Not all drives support amp fault messages. For details, please see the SqNodeLib header files and the drive manufacturer's documentation.

count	The number of amp faults in the buffer.
code	An array of drive specific amp fault coded values.
message	An array of drive specific amp fault message strings.

See Also [meiMotorAmpFault](#) | [meiMotorAmpFaultClear](#)

MEIMotorAmpFaultMsg

MEIMotorAmpFaultMsg

```
typedef char MEIMotorAmpFaultMsg[MEIMotorAmpMsgMAX];
```

Description

MotorAmpFaultMsg defines the amp fault message string definition.

See Also [MEIMotorAmpMsgMAX](#)

MEIMotorAmpWarnings

MEIMotorAmpWarnings

```
typedef struct MEIMotorAmpWarnings {
    long count;
    long code[MEIMotorAmpWarningsMAX];
    MEIMotorAmpWarningMsg message[MEIMotorAmpWarningsMAX];
} MEIMotorAmpWarnings;
```

Description

The **MotorAmpWarnings** structure contains the amp warning information from a SynqNet drive. The amp warning messages are drive specific. Not all drives support amp warning messages. For details, please see the SqNodeLib header files and the drive manufacturer's documentation.

count	The number of amp faults in the buffer.
code	An array of drive specific amp fault coded values.
message	An array of drive specific amp fault message strings.

See Also

[meiMotorAmpWarning](#) | [meiMotorAmpWarningClear](#)

MEIMotorAmpWarningsMsg

MEIMotorAmpWarningsMsg

```
typedef char MEIMotorAmpWarningsMsg[MEIMotorAmpMsgMAX];
```

Description

MotorAmpWarningsMsg defines the maximum number of amp warning messages per motor.

See Also

MPIMotorBrake

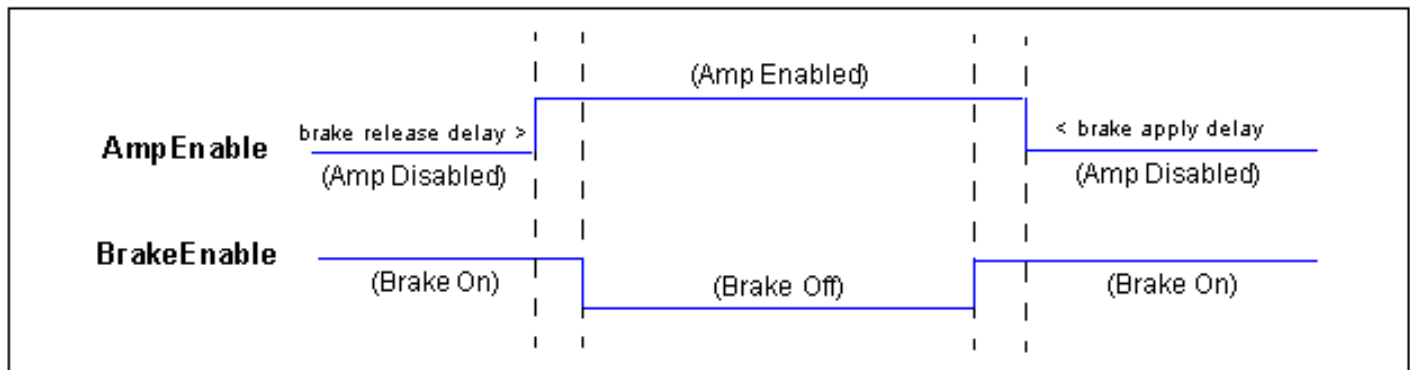
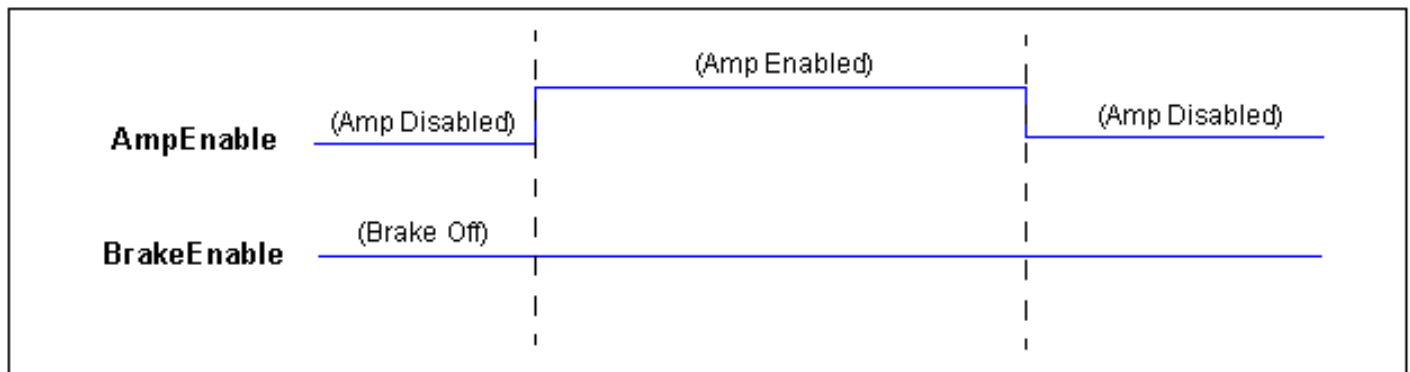
MPIMotorBrake

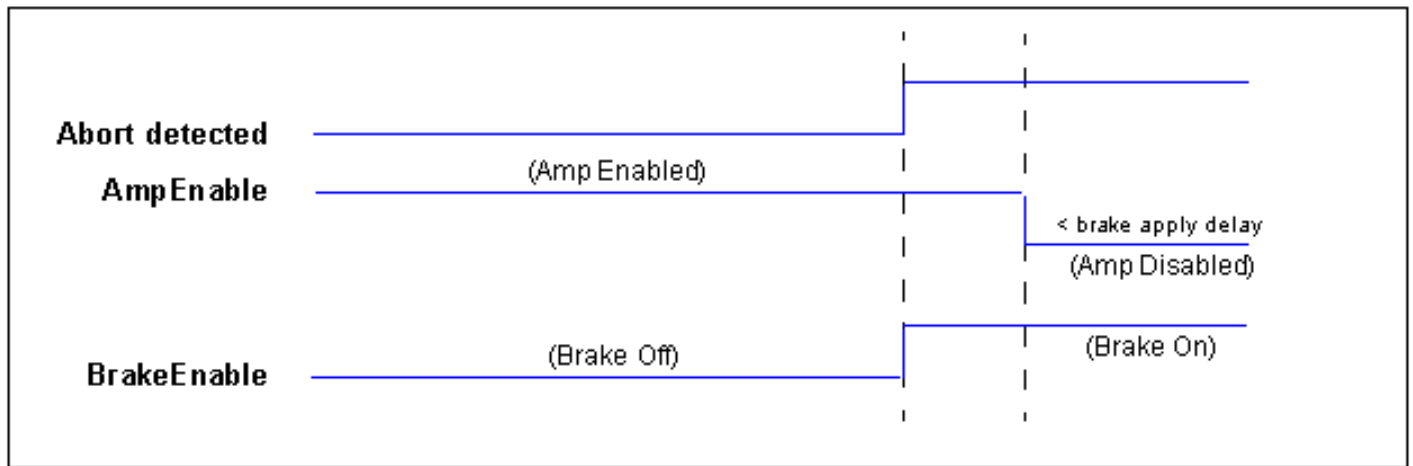
```
typedef struct MPIMotorBrake {
    MPIMotorBrakeMode    mode;
    float                applyDelay;
    float                releaseDelay;
} MPIMotorBrake;
```

Description

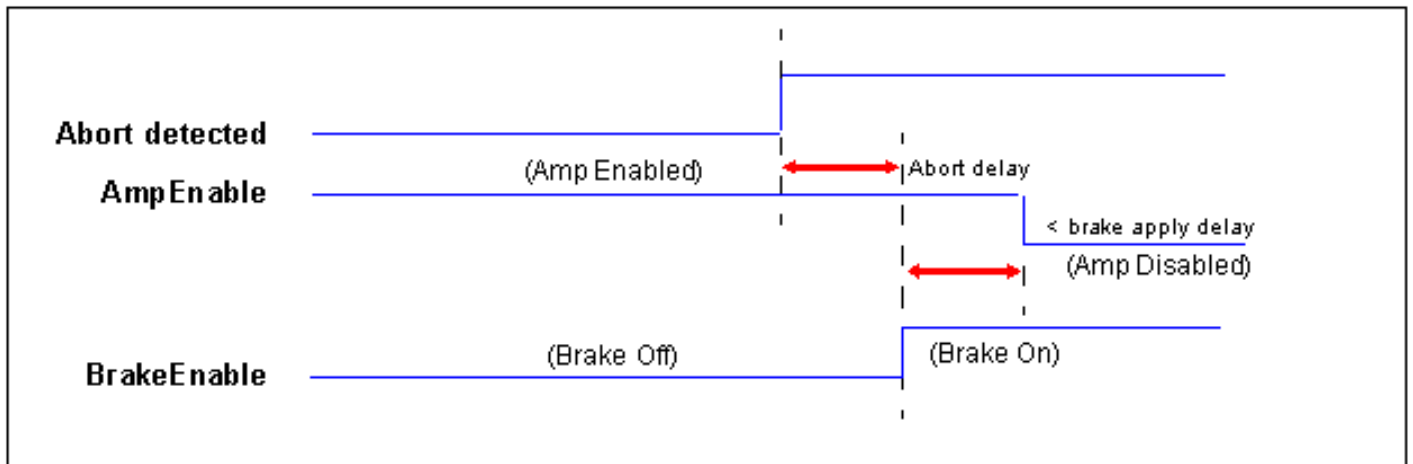
The **MotorBrake** structure specifies the configuration for a motor's dedicated brake logic. Each motor object has a dedicated brake output. The controller enables/disables the brake depending on the amp enable state and the brake configuration. When the amp enable is disabled, the brake is set to an active state. When the amp enable is enabled, the brake is set to an inactive state.

See the diagrams below for the brake logic details:





Case 3. Amp Off after ABORT Detection (in system with **NO abort delay)**



Case 4. Amp Off after ABORT Detection (in system **WITH abort delay)**

mode	An enumerated brake mode. See MPIMotorBrakeMode .
applyDelay	The time between when the brake is active and the amp enable is disabled. The units are in seconds.
releaseDelay	The time between when the amp enable is enabled and the brake is inactive. The units are in seconds.

See Also [mpiMotorConfigGet](#) | [mpiMotorConfigSet](#) | [MPIMotorConfig](#) | [MEIMotorDedicatedOut](#)

MPIMotorBrakeMode

MPIMotorBrakeMode

```
typedef enum{
    MPIMotorBrakeModeNONE,
    MPIMotorBrakeModeDELAY,

} MPIMotorBrakeMode;
```

Description **MotorBrakeMode** is an enumeration of modes for the dedicated brake signal.

MPIMotorBrakeModeNONE	Brake feature is disabled.
MPIMotorBrakeModeDELAY	Brake is enabled/disabled with specified delays.

See Also [MPIMotorBrake](#) | [MPIMotorConfig](#)

MPIMotorConfig / MEIMotorConfig

MPIMotorConfig

```
typedef struct MPIMotorConfig {
    MPIMotorType      type;

    /* Event configuration, ordered by MPIEventType */
    MPIMotorEventConfig  event[MPIEventTypeMOTOR_LAST];

    float            abortDelay;
    float            enableDelay;
    MPIMotorBrake      brake;

    MPIObjectMap       filterMap;
} MPIMotorConfig;
```

Description

event	Structure to configure various Motor Events. See MPIMotorEventConfig description.
abortDelay	Sets time value, in seconds, to delay Abort action after Event has occurred.
enableDelay	Sets time value, in seconds, to delay Enabling of the amplifier after commanded.
brake	Configures the dedicated brake logic. See MPIMotorBrake .
filterMap	Get/Set a map of Filter Objects to which the Motor is mapped. Default mapping is Filter 0 to Motor 0, Filtler 1 to Motor 1, etc. See also MPIObjectMap description in Object section.

MEIMotorConfig

```
typedef struct MEIMotorConfig {
    MEIMotorEncoder      Encoder[MEIXmpMotorEncoders];
    MEIMotorStatusOutput StatusOutput;

    MEIMotorIoConfig      Io[MEIMotorIoConfigIndexLAST];

    MEIMotorFaultConfig    faultConfig;

    MEIMotorStepper        Stepper;
    MEIMotorDacConfig      Dac;

    MEIXmpCommutationBlock Commutation;
    /* read-only from field Theta to end */
```

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](#)

```
MEIXmpLimitData      Limit[MEIXmpLimitLAST];

MPIAction             nodeFailureAction;
MPIAction             userFaultAction;
/* see MEISqNodeConfigUserFault{} structure */
MEIMotorDisableAction disableAction;

} MEIMotorConfig;
```

Description

Encoder	Structure to configure Motor Encoder type and parameters
StatusOutput	A structure to configure a motor's digital outputs to monitor axis status bits. Requires custom firmware.
Io	An array of motor I/O configuration structures.
faultConfig	A structure to configure a motor's fault bits. Support for motor fault bits is node/drive specific.
Stepper	Structure to configure Motor Stepper parameters. See MEIMotorStepper description.
Dac	Structure that includes Command and Auxiliary DAC configuration for each motor. See MEIMotorDacConfig description.
Commutation	A structure to configure controller sinusoidal commutation. This structure is controller specific. Please see the Sinusoidal Commutation Application Note for more details.
limit	Structure used to configure custom motor limits and events.
nodeFailureAction	Action applied to the motor when a Node failure occurs.
userFaultAction	Action applied to the motor when a User Fault occurs.
disableAction	Used to configure the controller to set the command position equal to the actual position while the motor is disabled.

See Also [mpiMotorConfigGet](#) | [mpiMotorConfigSet](#)

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.
Scale	Multiplier for the Dac channel. Default value is 1.0.
InputType	An enumerated value to define whether the Dac channel input is a float or a long. The input type is reserved for special or custom configurations.
Input	A pointer to a MEIXmpGenericValue, which is a union of a long and float value. The input pointer is reserved for special or custom configurations.

See Also

MEIMotorDacChannelStatus

MEIMotorDacChannelStatus

```
typedef struct MEIMotorDacChannelStatus {  
    float    level; /* volts */  
} MEIMotorDacChannelStatus;
```

Description **MotorDacChannelStatus** is a structure that returns the DAC output value.

level	<i>level</i> 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

MEIMotorDedicatedIn

MEIMotorDedicatedIn

```
typedef enum {
    MEIMotorDedicatedInAMP_FAULT          = MEIXmpMotorDedicatedFlagsMaskAMP_FAULT, /*
bit 0 */
    MEIMotorDedicatedInBRAKE_APPLIED      = MEIXmpMotorDedicatedFlagsMaskBRAKE_ON, /*
bit 1 */
    MEIMotorDedicatedInHOME                = MEIXmpMotorDedicatedFlagsMaskHOME, /*
bit 2 */
    MEIMotorDedicatedInLIMIT_HW_POS       = MEIXmpMotorDedicatedFlagsMaskPOS_LIMIT, /*
bit 3 */
    MEIMotorDedicatedInLIMIT_HW_NEG       = MEIXmpMotorDedicatedFlagsMaskNEG_LIMIT, /*
bit 4 */
    MEIMotorDedicatedInINDEX_PRIMARY      = MEIXmpMotorDedicatedFlagsMaskENC_INDEX0, /*
bit 5 */
    MEIMotorDedicatedInFEEDBACK_FAULT     = MEIXmpMotorDedicatedFlagsMaskENC_FAULT, /*
bit 6 */
    MEIMotorDedicatedInCAPTURED            = MEIXmpMotorDedicatedFlagsMaskCAPTURED, /*
bit 7 */
    MEIMotorDedicatedInHALL_A              = MEIXmpMotorDedicatedFlagsMaskHALL_A, /*
bit 8 */
    MEIMotorDedicatedInHALL_B              = MEIXmpMotorDedicatedFlagsMaskHALL_B, /*
bit 9 */
    MEIMotorDedicatedInHALL_C              = MEIXmpMotorDedicatedFlagsMaskHALL_C, /*
bit 10 */
    MEIMotorDedicatedInAMP_ACTIVE          = MEIXmpMotorDedicatedFlagsMaskAMP_ACTIVE, /*
bit 11 */
    MEIMotorDedicatedInINDEX_SECONDARY     = MEIXmpMotorDedicatedFlagsMaskENC_INDEX1,
/* bit 12 */
    MEIMotorDedicatedInWARNING              = MEIXmpMotorDedicatedFlagsMaskWARNING,
/* bit 13 */
    MEIMotorDedicatedInDRIVE_STATUS_9      = MEIXmpMotorDedicatedFlagsMaskDRIVE_STATUS9,
/* bit 14 */
    MEIMotorDedicatedInDRIVE_STATUS_10     =
MEIXmpMotorDedicatedFlagsMaskDRIVE_STATUS10, /* bit 15 */
} MEIMotorDedicatedIn;
```

Description

MotorDedicatedIn is an enumeration of bit masks for the motor's dedicated inputs. The support for dedicated inputs is node/drive specific. See the node/drive manufacturer's documentation for details.

MEIMotorDedicatedInAMP_FAULT	Generated by the masked motor fault bits. Active when one or more masked motor faults bits are active. See MEIMotorFaultConfig .
MEIMotorDedicatedInBRAKE_APPLIED	Mechanical brake state.
MEIMotorDedicatedInHOME	Position calibration sensor.
MEIMotorDedicatedInLIMIT_HW_POS	Hardware limit for the positive direction.
MEIMotorDedicatedInLIMIT_HW_NEG	Hardware limit for the negative direction.
MEIMotorDedicatedInINDEX_PRIMARY	Primary encoder index input signal.
MEIMotorDedicatedInFEEDBACK_FAULT	Position feedback status. TRUE when position feedback fails, FALSE when operating properly.
MEIMotorDedicatedInCAPTURED	Currently not supported.
MEIMotorDedicatedInHALL_A	Reflects the state of Hall Sensor A
MEIMotorDedicatedInHALL_B	Reflects the state of Hall Sensor B
MEIMotorDedicatedInHALL_C	Reflects the state of Hall Sensor C
MEIMotorDedicatedInAMP_ACTIVE	A bit set by the drive that indicates the amplifier's state. 1 = Amplifier is closing the current loop and the motor winding are energized. 0 = Amplifier is not closing the current loop and the motor windings are not energized. Support for this bit varies depending on the drive type.
MEIMotorDedicatedInINDEX_SECONDARY	Secondary encoder index input signal.
MEIMotorDedicatedInWARNING	Drive warning state. 1 = drive warning status bit is active and warning message is available 0 = drive warning status bit is not active. Support for this bit varies depending on the drive type.
MEIMotorDedicatedInDRIVE_STATUS_9	State of bit 9 in the SynqNet drive specific status register.
MEIMotorDedicatedInDRIVE_STATUS_10	State of bit 10 in the SynqNet drive specific status register.

See Also

[mpiMotorIoGet](#)

MEIMotorDedicatedOut

MEIMotorDedicatedOut

```
typedef enum {
    MEIMotorDedicatedOutAMP_ENABLE      = MEIXmpMotorDedicatedFlagsMaskAMP_ENABLE,
    /* bit 0 */
    MEIMotorDedicatedOutBRAKE_RELEASE   = MEIXmpMotorDedicatedFlagsMaskBRAKE_RELEASE,
    /* bit 1 */
} MEIMotorDedicatedOut;
```

Description **MotorDedicatedOut** is an enumeration of bit masks for the motor's dedicated outputs. The support for dedicated outputs is node/drive specific. See the node/drive manufacturer's documentation for details.

MEIMotorDedicatedOutAMP_ENABLE	Enable/disable drive or amplifier. Drive is enabled when TRUE, disabled when FALSE.
MEIMotorDedicatedOutBRAKE_RELEASE	Enable/disable mechanical brake. Brake is released (motor shaft is free) when TRUE, engaged when FALSE.

See Also [mpiMotorIoGet](#) | [mpiMotorIoSet](#)

MEIMotorDisableAction

MEIMotorDisableAction

```
typedef enum MEIMotorDisableAction {  
    MEIMotorDisableActionNONE,  
    MEIMotorDisableActionCMD_EQ_ACT,  
} MEIMotorDisableAction;
```

Description

MotorDisableAction is an enumeration of controller actions to be applied when the Amp Enable is disabled. This feature can be configured by calling mpiMotorConfigSet(...) with the disableAction element in the MEIMotorConfig structure set to one of the values defined in the MEIMotorDisableAction enumeration.

The default configuration is MEIMotorDisableActionCMD_EQ_ACT. This configuration applies some safety features which eliminate motor jumps when the Amp Enable is enabled. This is the recommended configuration for all servo motor types.

The CMD_EQ_ACT action does not operate with stepper motors. If the motor type is a stepper, make sure to set the disable action to NONE. The pulse output is based on the command position, so if the controller sets the command position equal to the actual position during motion, it will cause very unusual motion profiles.

MEIMotorDisableActionNONE	No action. When the Amp Enable is disabled (or enabled), the controller continues to calculate and apply the torque demand output value.
MEIMotorDisableActionCMD_EQ_ACT	Command position equals actual position action (default). When the Amp Enable is disabled, the controller will: 1) Disable the servo loop output (except for the offset). 2) Set the command position equal to the actual position every sample. 3) Clear the integrator error. When the Amp Enable is enabled, the controller will operate the servo loop normally.

See Also

[mpiMotorConfigSet](#) | [MEIMotorConfig](#) | [MEIMotorDisableAction](#)

MPIMotorEncoder / MEIMotorEncoder

MPIMotorEncoder

```
typedef enum {
    MPIMotorEncoderPRIMARY,
    MPIMotorEncoderSECONDARY,
} MPIMotorEncoder;
```

Description **MotorEncoder** is an enumeration of encoder feedback inputs for a motor.

MPIMotorEncoderPRIMARY	The first encoder feedback for a motor.
MPIMotorEncoderSECONDARY	The second encoder feedback for a motor.

See Also

MEIMotorEncoder

```
typedef struct MEIMotorEncoder {
    MEIMotorEncoderType
    long
    long
    long
    MEIMotorEncoderRatio
    MEIMotorEncoderReverseModulo
} MEIMotorEncoder;

type;
encoderPhase;
/* 0 => normal, else reversed */
filterDisable;
/* 0 => quad filter enabled,
else not enabled */
countsPerRev;
ratio;
reverseModulo;
```

Description

type	The type of feedback being used by the motor.
encoderPhase	Controls the direction of the encoder counts (only applicable with quadrature encoder feedback).
filterDisable	Enables/Disables the use of filters on the encoder signal.

countsPerRev	The number of encoder counts per one revolution of the motor's shaft.
ratio	Custom firmware required. Encoder feedback ratio for scaling.
reverseModulo	Custom firmware required. Reverse modulo value for the encoder to handle rollover.

See Also [mpiMotorCreate](#)

MPIMotorEncoderFault

MPIMotorEncoderFault

```
typedef enum {
    MPIMotorEncoderFaultPRIMARY,
    MPIMotorEncoderFaultSECONDARY,
    MPIMotorEncoderFaultPRIMARY_OR_SECONDARY,
} MPIMotorEncoderFault;
```

Description **MotorEncoderFault** is an enumeration of encoder fault sources for motor encoder fault events. Each motor object supports a primary and secondary encoder. The hardware may or may not support a secondary encoder.

MPIMotorEncoderFaultPRIMARY	Sets the Motor Event (Encoder Fault) to trigger from the primary encoder.
MPIMotorEncoderFaultSECONDARY	Sets the Motor Event (Encoder Fault) to trigger from the secondary encoder.
MPIMotorEncoderFaultPRIMARY_OR_SECONDARY	Sets the Motor Event (Encoder Fault) to trigger from either the primary or secondary encoder.

See Also [MPIMotorEventConfig](#) | [MPIMotorEventTrigger](#)

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

MEIMotorEncoderRatio

MEIMotorEncoderRatio

```
typedef struct MEIMotorEncoderRatio {
    long    A;    /* denominator */
    long    B;    /* numerator */
} MEIMotorEncoderRatio;
```

Description

MotorEncoderRatio is used to set the Encoder ratio. Ratio is programmed into the firmware via a denominator (A) and a numerator (B). $B/A = \text{ratio}$.

NOTE: Custom Firmware is required to support this data type.

A	the denominator.
B	the numerator.

See Also

MEIMotorEncoderReverseModulo

MEIMotorEncoderReverseModulo

```
typedef struct MEIMotorEncoderReverseModulo {
    long    *Ptr;    /* XMP Address */
    long    Rollover;
} MEIMotorEncoderReverseModulo;
```

Description

MotorEncoderReverseModulo is used to program the firmware to calculate a 32-bit encoder position based off of the data that is pointed to by ***Ptr**. Each sample, the firmware calculates a delta from the data that is pointed to by ***Ptr** and the **Rollover** value. This delta is added to a 32-bit counter to create a 32-bit position.

NOTE: Custom Firmware is required to support this data type.

*Ptr	a pointer to the XMP address.
Rollover	the value is the resolution of the incoming data pointed to by *Ptr.

See Also

MEIMotorEncoderType

MEIMotorEncoderType

```
typedef enum MEIMotorEncoderType{
    MEIMotorEncoderTypeQUAD_AB,
    MEIMotorEncoderTypeDRIVE,
    MEIMotorEncoderTypeSSI,
} MEIMotorEncoderType;
```

Description **MotorEncoderType** is the type of encoder being used for feedback. The encoder type is drive specific. Therefore, an appropriate default value should be set by the MPI.

MEIMotorEncoderTypeQUAD_AB	Quadrature encoder feedback.
MEIMotorEncoderTypeDRIVE	Drive memory interface feedback. This includes all feedback types supported by the drive.
MEIMotorEncoderTypeSSI	Synchronous Serial Interface (SSI) absolute encoder feedback.

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. NOTE: The duration parameter for the home limit should be zero for standard configurations. A non-zero value will result in the home limit being missed.

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 conditionl 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 Logic table below.

output	is a structure that allows specific action to be taken when the Limit Event is generated. In addition to generating 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.

Values 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

Values 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](#) | [MPIEncoderFault](#)

MPIMotorEventTrigger

MPIMotorEventTrigger

```
typedef union {
    long          polarity; /* 0 => active low, else active high */
    long          position; /* MPIEventTypeLIMIT_SW_[POS|NEG] */
    float         error;    /* MPIEventTypeLIMIT_ERROR */
    MPIMotorEncoderFault encoder; /* MPIEventTypeENCODER_FAULT */
} MPIMotorEventTrigger;
```

Description

polarity	Configures the polarity for the motor event trigger. If polarity = 0 (FALSE), the event will trigger on an active low signal. If polarity = 1 (TRUE), the event will trigger on an active high signal.
position	Configures the positive and negative software position limits. The controller monitors the actual position and compares it to the positive and negative software position limits. If the positive limit is exceeded the controller will generate a MPIEventTypeLIMIT_SW_POS event. If the negative limit is exceeded the controller will generate a MPIEventTypeLIMIT_SW_NEG event.
error	Configures the position error limit. The controller calculates the position error each sample period: (command position - actual position) = position error. If the position error exceeds the range of the specified error limit, the controller will generate a MPIEventTypeLIMIT_ERROR.
encoder	Configures the controller to monitor the primary or secondary for faults. The encoder should be set to one of the values defined by the MPIMotorEncoderFault{...} enumeration. When configured, if the primary, secondary, or either encoder generates a fault, the controller will generate the MPIEventTypeENCODER_FAULT.

See Also [MPIMotorEventConfig](#) | [MPIMotorEncoderFault](#) | [MPIEventType](#) | [MEIEventType](#)

MEIMotorFaultBit

MEIMotorFaultBit

```
typedef enum MEIMotorFaultBit {
    MEIMotorFaultBitINVALID,
    MEIMotorFaultBitAMP_FAULT,
    MEIMotorFaultBitDRIVE_FAULT,
    MEIMotorFaultBitWATCHDOG_FAULT,
    MEIMotorFaultBitCHECKSUM_ERROR,
    MEIMotorFaultBitFEEDBACK_FAULT,
    MEIMotorFaultBitAMP_NOT_POWERED,
    MEIMotorFaultBitDRIVE_NOT_READY,
    MEIMotorFaultBitFEEDBACK_FAULT_SECONDARY,
} MEIMotorFaultBit;
```

Description

MotorFaultBit is an enumeration of motor fault bits for the SynqNet node drive interface. The support for motor fault bits is node/drive specific. Please see the node\drive manufacturer's documentation for details.

MEIMotorFaultBitAMP_FAULT

External amp fault input pin. Indicates a fault condition in the amplifier.

MEIMotorFaultBitDRIVE_FAULT

Bit in the drive interface status register. Indicates a fault condition in the drive.

MEIMotorFaultBitWATCHDOG_FAULT

Bit in the drive interface status register. Indicates a drive failure due to a watchdog timeout. The node alternately sets/clears the watchdog each sample, the drive's processor then clears/sets the watchdog. If the drive's processor does not respond, the watchdog fault bit is set.

MEIMotorFaultBitCHECKSUM_ERROR

Bit in the drive interface status register. Indicates the data transfer (via serial or parallel memory interface) between the failed SynqNet node FPGA and drive with a checksum error.

MEIMotorFaultBitFEEDBACK_FAULT

Bit in the drive interface status register. Active when one or more of the feedback status bits is triggered. Indicates that the drive/motor position feedback system has failed.

MEIMotorFaultBitAMP_NOT_POWERED

Bit in the drive interface status register. The SynqNet drive amplifier power stage does not have sufficient voltage. The motor windings cannot be energized properly until this bit is clear.

MEIMotorFaultBitDRIVE_NOT_READY

Bit in the drive interface status register. The SynqNet drive is not ready to receive commands or servo motors. This fault indicates the drive has not completed its processor initialization or there is some other serious drive processor problem.

MEIMotorFaultBitFEEDBACK_FAULT_SECONDARY

Bit in the drive interface status register. Active when one or more of the secondary feedback status bits is triggered. Indicates that the drive/motor auxiliary position feedback system has failed.

See Also

[MEIMotorFaultMask](#) | [MEIMotorFaultConfig](#) | [meiMotorStatus](#)

MEIMotorFaultConfig

MEIMotorFaultConfig

```
typedef struct MEIMotorFaultConfig {
    MEIMotorFaultMask    faultMask; /* sets the
                                     MEIMotorDedicatedInAMP_FAULT state */
} MEIMotorFaultConfig;
```

Description The **MotorFaultConfig** structure specifies the motor fault bit configuration.

faultMask	A mask of motor fault bits. The masked motor fault bits will be monitored by the controller's motor object. If any masked motor fault bit is active (TRUE), the motor's dedicated amp fault input (MEIMotorDedicatedInAMP_FAULT) is set active (TRUE). The support for motor fault bits is node/drive specific. During SynqNet network initialization, the SqNodeLib automatically configures the faultMask based on the node type. See the node/drive manufacturer's documentation for details.

See Also [MEIMotorFaultBit](#) | [MEIMotorFaultMask](#) | [meiMotorStatus](#)

MEIMotorFaultMaskDRIVE_NOT_READY	Bit in the drive interface status register. The SynqNet drive is not ready to receive commands or servo motors. This fault indicates the drive has not completed its processor initialization or there is some other serious drive processor problem.
MEIMotorFaultMaskFEEDBACK_FAULT_SECONDARY	Bit in the drive interface status register. Active when one or more of the secondary feedback status bits is triggered. Indicates that the drive/motor auxiliary position feedback system has failed.

See Also [MEIMotorFaultBit](#) | [MEIMotorFaultConfig](#) | [meiMotorStatus](#)

MEIMotorFaultMask

MEIMotorFaultMask

```
typedef enum MEIMotorFaultMask {
    MEIMotorFaultMaskAMP                = (1<< MEIMotorFaultBitAMP_FAULT),
    MEIMotorFaultMaskDRIVE               = (1<< MEIMotorFaultBitDRIVE_FAULT),
    MEIMotorFaultMaskWATCHDOG           = (1<< MEIMotorFaultBitWATCHDOG_FAULT),
    MEIMotorFaultMaskCHECKSUM           = (1<< MEIMotorFaultBitCHECKSUM_ERROR),
    MEIMotorFaultMaskFEEDBACK           = (1<< MEIMotorFaultBitFEEDBACK_FAULT),
    MEIMotorFaultMaskAMP_NOT_POWERED    = (1<< MEIMotorFaultBitAMP_NOT_POWERED),
    MEIMotorFaultMaskDRIVE_NOT_READY    = (1<< MEIMotorFaultBitDRIVE_NOT_READY),
    MEIMotorFaultMaskFEEDBACK_FAULT_SECONDARY = (1<<
MEIMotorFaultBitFEEDBACK_FAULT_SECONDARY),
} MEIMotorFaultMask;
```

Description

MotorFaultMask is an enumeration of motor fault bit masks for the SynqNet node drive interface. The support for motor fault bits is node/drive specific. Please see the node\drive manufacturer's documentation for details.

MEIMotorFaultMaskAMP	External amp fault input pin. Indicates a fault condition in the amplifier.
MEIMotorFaultMaskDRIVE	Bit in the drive interface status register. Indicates a fault condition in the drive.
MEIMotorFaultMaskWATCHDOG	Bit in the drive interface status register. Indicates a drive failure due to a watchdog timeout. The node alternately sets/clears the watchdog each sample, the drive's processor then clears/sets the watchdog. If the drive's processor does not respond, the watchdog fault bit is set.
MEIMotorFaultMaskCHECKSUM	Bit in the drive interface status register. Indicates the data transfer (via serial or parallel memory interface) between the failed SynqNet node FPGA and drive with a checksum error.
MEIMotorFaultMaskFEEDBACK	Bit in the drive interface status register. Active when one or more of the feedback status bits is triggered. Indicates that the drive/motor position feedback system has failed.
MEIMotorFaultMaskAMP_NOT_POWERED	Bit in the drive interface status register. The SynqNet drive amplifier power stage does not have sufficient voltage. The motor windings cannot be energized properly until this bit is clear.

MPIMotorFeedback

MPIMotorFeedback

```
typedef double MPIMotorFeedback[MPIMotorEncoderLAST];
```

Description

MotorFeedback is an array of doubles used to store motor feedback (both primary and secondary).

See Also

[MPIMotorEncoder](#) | [mpiMotorFeedback](#)

MEIMotorInfo

MEIMotorInfo

```
typedef struct MEIMotorInfo {
    MEIMotorInfoNodeType    nodeType;
    struct {
        long    networkNumber;
        long    nodeNumber;
        long    driveIndex;
    } sqNode;

    long    captureCount;
    long    encoderCount;
    long    probeCount;
} MEIMotorInfo;
```

Description

The [MotorInfo](#) structure contains static data determined during network initialization. It identifies the network, node, and drive interface associated with the motor object.

nodeType	Identifies the type of node. See MEIMotorInfoNodeType .
networkNumber	An index to the SynqNet network (0, 1, 2 etc.).
nodeNumber	An index to the node (0, 1, 2, etc.).
driveIndex	An index to the drive interface (0, 1, 2, etc.).
captureCount	Counts the number of captures for the motor.
encoderCount	Counts the number of encoders for the motor.
probeCount	Counts the number of hardware Probe engines for the motor.

See Also

[meiMotorInfo](#)

MEIMotorInfoNodeType

MEIMotorInfoNodeType

```
typedef enum MEIMotorInfoNodeType {  
    MEIMotorInfoNodeTypeNONE,  
    MEIMotorInfoNodeTypeSQNODE,  
} MEIMotorInfoNodeType;
```

Description **MotorInfoNodeType** is an enumeration of node types. It specifies the node type associated with the motor.

MEIMotorInfoNodeTypeNONE	Not a network node. The node is local to the controller.
MEIMotorInfoNodeTypeSQNODE	A SynqNet node type.

See Also [MEIMotorInfo](#)

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	The input value reflects the 32 bits of general and dedicated input values. The dedicated input values (bits 0-15) are specified by MEIMotorDedicatedIn . The general purpose input values (bits 16-31) can be specified by MEIMotorIoMask .
output	The output value reflects the 32 bits of general and dedicated output values. The dedicated output values (bits 0-15) are specified by MEIMotorDedicatedOut . The general purpose output values (bits 16-31) can be specified by MEIMotorIoMask .

See Also [MEIMotorDedicatedIn](#) | [MEIMotorDedicatedOut](#) | [MEIMotorIoMask](#)

MEIMotorIoConfig

MEIMotorIoConfig

```
typedef struct MEIMotorIoConfig {  
    MEIMotorIoType      Type;  
} MEIMotorIoConfig;
```

Description The [MotorIoConfig](#) structure specifies the configuration for the motor's digital I/O.

Type	The I/O bit function. See MEIMotorIoType .

See Also [mpiMotorConfigGet](#) | [mpiMotorConfigSet](#)

MEIMotorIoConfigIndex

MEIMotorIoConfigIndex

```
typedef enum MEIMotorIoConfigIndex {
    MEIMotorIoConfigIndex0,
    MEIMotorIoConfigIndex1,
    MEIMotorIoConfigIndex2,
    MEIMotorIoConfigIndex3,
    MEIMotorIoConfigIndex4,
    MEIMotorIoConfigIndex5,
    MEIMotorIoConfigIndex6,
    MEIMotorIoConfigIndex7,
    MEIMotorIoConfigIndex8,
    MEIMotorIoConfigIndex9,
    MEIMotorIoConfigIndex10,
    MEIMotorIoConfigIndex11,
    MEIMotorIoConfigIndex12,
    MEIMotorIoConfigIndex13,
    MEIMotorIoConfigIndex14,
    MEIMotorIoConfigIndex15,
} MEIMotorIoConfigIndex;
```

Description

MotorIoConfigIndex is an enumeration of configurable motor I/O, indexed by a number.

MEIMotorIoConfigIndex0	motor I/O number 0
MEIMotorIoConfigIndex1	motor I/O number 1
MEIMotorIoConfigIndex2	motor I/O number 2
MEIMotorIoConfigIndex3	motor I/O number 3
MEIMotorIoConfigIndex4	motor I/O number 4
MEIMotorIoConfigIndex5	motor I/O number 5
MEIMotorIoConfigIndex6	motor I/O number 6
MEIMotorIoConfigIndex7	motor I/O number 7
MEIMotorIoConfigIndex8	motor I/O number 8
MEIMotorIoConfigIndex9	motor I/O number 9
MEIMotorIoConfigIndex10	motor I/O number 10
MEIMotorIoConfigIndex11	motor I/O number 11
MEIMotorIoConfigIndex12	motor I/O number 12
MEIMotorIoConfigIndex13	motor I/O number 13

MEIMotorIoConfigIndex14	motor I/O number 14
MEIMotorIoConfigIndex15	motor I/O number 15

See Also [MEIMotorIoConfig](#)

MEIMotorIoMask

MEIMotorIoMask

```
typedef enum {
    MEIMotorIoMask0 = MEIXmpMotorIoMaskConfigurable0, /* bit 16 */
    MEIMotorIoMask1 = MEIXmpMotorIoMaskConfigurable1, /* bit 17 */
    MEIMotorIoMask2 = MEIXmpMotorIoMaskConfigurable2, /* bit 18 */
    MEIMotorIoMask3 = MEIXmpMotorIoMaskConfigurable3, /* bit 19 */
    MEIMotorIoMask4 = MEIXmpMotorIoMaskConfigurable4, /* bit 20 */
    MEIMotorIoMask5 = MEIXmpMotorIoMaskConfigurable5, /* bit 21 */
    MEIMotorIoMask6 = MEIXmpMotorIoMaskConfigurable6, /* bit 22 */
    MEIMotorIoMask7 = MEIXmpMotorIoMaskConfigurable7, /* bit 23 */
    MEIMotorIoMask8 = MEIXmpMotorIoMaskConfigurable8, /* bit 24 */
    MEIMotorIoMask9 = MEIXmpMotorIoMaskConfigurable9, /* bit 25 */
    MEIMotorIoMask10 = MEIXmpMotorIoMaskConfigurable10, /* bit 26 */
    MEIMotorIoMask11 = MEIXmpMotorIoMaskConfigurable11, /* bit 27 */
    MEIMotorIoMask12 = MEIXmpMotorIoMaskConfigurable12, /* bit 28 */
    MEIMotorIoMask13 = MEIXmpMotorIoMaskConfigurable13, /* bit 29 */
    MEIMotorIoMask14 = MEIXmpMotorIoMaskConfigurable14, /* bit 30 */
    MEIMotorIoMask15 = MEIXmpMotorIoMaskConfigurable15, /* bit 31 */
} MEIMotorIoMask;
```

Description [MotorIoMask](#) is an enumeration of bit masks for the motor's configurable I/O. The support for configurable I/O is node/drive specific. See the node/drive manufacturer's documentation for details.

MEIMotorIoMask0	motor I/O mask for bit number 0
MEIMotorIoMask1	motor I/O mask for bit number 1
MEIMotorIoMask2	motor I/O mask for bit number 2
MEIMotorIoMask3	motor I/O mask for bit number 3
MEIMotorIoMask4	motor I/O mask for bit number 4
MEIMotorIoMask5	motor I/O mask for bit number 5
MEIMotorIoMask6	motor I/O mask for bit number 6
MEIMotorIoMask7	motor I/O mask for bit number 7
MEIMotorIoMask8	motor I/O mask for bit number 8
MEIMotorIoMask9	motor I/O mask for bit number 9
MEIMotorIoMask10	motor I/O mask for bit number 10
MEIMotorIoMask11	motor I/O mask for bit number 11
MEIMotorIoMask12	motor I/O mask for bit number 12
MEIMotorIoMask13	motor I/O mask for bit number 13

MEIMotorIoMask14	motor I/O mask for bit number 14
MEIMotorIoMask15	motor I/O mask for bit number 15

See Also [mpiMotorIoGet](#) | [mpiMotorIoSet](#) | [MEIMotorIoType](#)

MEIMotorIoType

MEIMotorIoType

```
typedef enum MEIMotorIoType {
    MEIMotorIoTypeOUTPUT,
    MEIMotorIoTypeSOURCE1,
    MEIMotorIoTypeSOURCE2,
    MEIMotorIoTypeSOURCE3,
    MEIMotorIoTypeSOURCE4,
    MEIMotorIoTypeSOURCE5,
    MEIMotorIoTypeSOURCE6,
    MEIMotorIoTypeSOURCE7,
    MEIMotorIoTypeSOURCE8,
    MEIMotorIoTypeSOURCE9,
    MEIMotorIoTypeSOURCE10,
    MEIMotorIoTypeSOURCE11,
    MEIMotorIoTypeSOURCE12,
    MEIMotorIoTypeSOURCE13,
    MEIMotorIoTypeSOURCE14,
    MEIMotorIoTypeSOURCE15,

    MEIMotorIoTypeINPUT = 0x10,
} MEIMotorIoType;
```

Description **MotorIoType** is an enumeration of motor I/O functions. A SynqNet node's motor I/O may support one or more features depending on the node hardware and FPGA. MotorIoType contains the generic enumerations for all nodes. For the node specific enumerations, please see the individual SqNode modules for details.

MEIMotorIoTypeOUTPUT	discrete digital output
MEIMotorIoTypeSOURCE1	motor I/O source number 1 for node-specific feature.
MEIMotorIoTypeSOURCE2	motor I/O source number 2 for node-specific feature.
MEIMotorIoTypeSOURCE3	motor I/O source number 3 for node-specific feature.
MEIMotorIoTypeSOURCE4	motor I/O source number 4 for node-specific feature.
MEIMotorIoTypeSOURCE5	motor I/O source number 5 for node-specific feature.
MEIMotorIoTypeSOURCE6	motor I/O source number 6 for node-specific feature.
MEIMotorIoTypeSOURCE7	motor I/O source number 7 for node-specific feature.
MEIMotorIoTypeSOURCE8	motor I/O source number 8 for node-specific feature.
MEIMotorIoTypeSOURCE9	motor I/O source number 9 for node-specific feature.
MEIMotorIoTypeSOURCE10	motor I/O source number 10 for node-specific feature.
MEIMotorIoTypeSOURCE11	motor I/O source number 11 for node-specific feature.

MEIMotorIoTypeSOURCE12	motor I/O source number 12 for node-specific feature.
MEIMotorIoTypeSOURCE13	motor I/O source number 13 for node-specific feature.
MEIMotorIoTypeSOURCE14	motor I/O source number 14 for node-specific feature.
MEIMotorIoTypeSOURCE15	motor I/O source number 15 for node-specific feature.
MEIMotorIoTypeINPUT	discrete digital input

See Also [mpiMotorConfigGet](#) | [mpiMotorConfigSet](#)

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.

MPIMotorMessageMOTOR_INVALID

The motor number is out of range. This message code is returned by [mpiMotorCreate\(...\)](#) if the motor number is less than zero or greater than or equal to MEIXmpMAX_Motors.

MEIMotorMessage

```
typedef enum {
    MEIMotorMessageMOTOR_NOT_ENABLED,
    MEIMotorMessageSECONDARY_ENCODER_NA,
    MEIMotorMessageHARDWARE_NOT_FOUND,
    MEIMotorMessageSTEPPER_INVALID,
    MEIMotorMessageDISABLE_ACTION_INVALID,
    MEIMotorMessagePULSE_WIDTH_INVALID,
    MEIMotorMessageFEEDBACK_REVERSAL_NA,
    MEIMotorMessageFILTER_DISABLE_NA,
} MEIMotorMessage;
```

Description

MotorMessage is an enumeration of Motor error messages that can be returned by the MPI library.

MEIMotorMessageMOTOR_NOT_ENABLED

The motor number is not active in the controller. This message code is returned by [mpiMotorEventConfig\(...\)](#) if the specified motor is not enabled in the controller. To correct the problem, use [mpiControlConfigSet\(...\)](#) to enable the motor object, by setting the motorCount to greater than the motor number. For example, to enable motor 0 to 3, set motorCount to 4.

MEIMotorMessageSECONDARY_ENCODER_NA

The motor's secondary encoder is not available. This message code is returned by [mpiMotorConfigSet\(...\)](#) or [mpiMotorEventConfigSet\(...\)](#) if the encoder fault trigger is configured for a secondary encoder when the hardware does not support a secondary encoder. To correct the problem, do not select the secondary encoder when configuring the encoder fault conditions.

MEIMotorMessageHARDWARE_NOT_FOUND

The motor object's hardware resource is not available. This message code is returned by [mpiMotorConfigGet\(...\)](#) or [mpiMotorConfigSet\(...\)](#) if the node hardware for the motor is not found. During controller and network initialization the nodes and motor count for each node is discovered and mapped to the controller's motor objects. An application should not configure a motor object if there is no mapped hardware to receive the service commands. To correct this problem, verify that all expected nodes were found. Use [meiSqNodeInfo\(...\)](#) to determine the node topology and motor count per node. Check the node hardware power and network connections.

MEIMotorMessageSTEPPER_INVALID

The motor object stepper configuration is not valid. These message codes are returned by [mpiMotorConfigGet\(...\)](#) or [mpiMotorConfigSet\(...\)](#) if the motor type is configured for stepper while the disable action is configured for command position equals actual position. The disable action feature sets the command position equal to the actual position when the amp enable signal is set to disable. Stepper motor types are driven by a digital pulse, which is triggered by the controller's command position. Do not use disable action set to command equals actual with stepper motor types.

MEIMotorMessageDISABLE_ACTION_INVALID

The motor object stepper configuration is not valid. These message codes are returned by [mpiMotorConfigGet\(...\)](#) or [mpiMotorConfigSet\(...\)](#) if the motor type is configured for stepper while the disable action is configured for command position equals actual position. The disable action feature sets the command position equal to the actual position when the amp enable signal is set to disable. Stepper motor types are driven by a digital pulse, which is triggered by the controller's command position. Do not use disable action set to command equals actual with stepper motor types.

MEIMotorMessagePULSE_WIDTH_INVALID

The motor object stepper pulse width is not valid. The pulse width must be no greater than 1 millisecond and no less than 100 nanoseconds.

MEIMotorMessageFEEDBACK_REVERSAL_NA

Feedback reversal is not applicable or is not supported for the feedback type specified.

MEIMotorMessageFILTER_DISABLE_NA

Disabling of filters is not applicable or is not supported for the feedback type specified.

See Also [mpiMotorCreate](#)

MEIMotorStatus

MEIMotorStatus

```
typedef struct MEIMotorStatus {
    MEIMotorDacStatus      dac;
    MEIMotorFaultMask     faultMask;
    MEIMotorStepperStatus  stepper;
} MEIMotorStatus;
```

Description The **MotorStatus** structure that returns the specific Motor Status registers of the controller.

dac	The status for both Command and Auxiliary DACs.
faultMask	The motor fault bit masks for the SynqNet node drive interface.
stepper	Shows the status of the pulse module. It is used for checking network synchronization with the pulse module. It can also be used to check if a pulse velocity/width error occurred.

See Also [MEIMotorStepperStatus](#)

MEIMotorStatusOutput

MEIMotorStatusOutput

```
typedef struct MEIMotorStatusOutput {
    long          *outPtr;
    MEIXmpIOMask  ioMask[MEIXmpMotorStatusOutputs];
} MEIMotorStatusOutput;
```

Description The **MotorStatusOutput** structure specifies which motor outputs to follow the motor status bits. This feature requires custom controller firmware.

*outPtr	a pointer to controller memory.
ioMask	a bit mask to select the motor output signals.

See Also [mpiMotorConfigGet](#) | [mpiMotorConfigSet](#)

MEIMotorStepper

MEIMotorStepper

```
typedef struct MEIMotorStepper {
    long                loopback;    /* TRUE = FPGA pulse feedback,
                                     FALSE = encoder feedback */

    MEIMotorStepperPulse pulseA;
    MEIMotorStepperPulse pulseB;
    double              pulsewidth; /* output pulse width (sec),
                                     10^(-7) < pulsewidth < 10^(-3) */
} MEIMotorStepper;
```

Description [MotorStepper](#) is a structure used to configure Stepper Motor parameters.

To use pulse outputs, you will need to:

- 1. Configure the motor type for STEPPER.
- 2. See the motor's stepper pulse width. Make sure it meets the drive's requirments and is not smaller than 2x the minimum pulse period.
- 3. Enable the motor's stepper loopback if there is no encoder feedback.
- 4. Configure the motor's stepper pulseA and pulseB types for STEP, DIR, CW, CCW, QUADA, or QUADB.
- 5. Select the motor I/O's config type for pulseA and pulseB. This will route the pulseA/B signals to the node's digital outputs.

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.
pulseA	Structure used to configure Step Pulse A.
pulseB	Structure used to configure Step Pulse B.
pulseWidth	sets width of Step pulse. Valid values range from a minimum width of 0.1 usec to maximum width of 25.5 usec.

See Also [MEIMotorStepperPulse](#) | [MEIMotorStepperPulseType](#)

MEIMotorStepperPulse

MEIMotorStepperPulse

```
typedef struct MEIMotorStepperPulse {
    MEIMotorStepperPulseType    type;
    long                        invert;
} MEIMotorStepperPulse;
```

Description

[MotorStepperPulse](#) is a

To use pulse outputs, you will need to:

- 1. Configure the motor type for STEPPER.
- 2. See the motor's stepper pulse width. Make sure it meets the drive's requirments and is not smaller than 2x the minimum pulse period.
- 3. Enable the motor's stepper loopback if there is no encoder feedback.
- 4. Configure the motor's stepper pulseA and pulseB types for STEP, DIR, CW, CCW, QUADA, or QUADB.
- 5. Select the motor I/O's config type for pulseA and pulseB. This will route the pulseA/B signals to the node's digital outputs.

type	Configures the stepper motor's pulse type.
invert	If set to TRUE the actual Pulse output will be inverted by the FPGA.

See Also

[MEIMotorStepper](#)

MEIMotorStepperPulseType

MEIMotorStepperPulseType

```
typedef enum MEIMotorStepperPulseType {  
    MEIMotorStepperPulseTypeSTEP,  
    MEIMotorStepperPulseTypeDIR,  
  
    MEIMotorStepperPulseTypeCW,  
    MEIMotorStepperPulseTypeCCW,  
  
    MEIMotorStepperPulseTypeQUADA,  
    MEIMotorStepperPulseTypeQUADB,  
  
} MEIMotorStepperPulseType;
```

Description **MotorStepperPulseType** is an enumeration used to specify the pulse output signal type.

MEIMotorStepperPulseTypeSTEP	This will enable the pulse output (either A or B) to generate a step output. Use it together with MEIMotorStepperPulseTypeDIR to provide a complete step interface.
MEIMotorStepperPulseTypeDIR	This will enable the pulse output (either A or B) to output the direction of the move. Use it together with MEIMotorStepperPulseTypeSTEP to provide a complete step interface.
MEIMotorStepperPulseTypeCW	This will enable the pulse output (either A or B) to output a clockwise pulse train. Used together with MEIMotorStepperPulseTypeCCW to provide a complete step interface.
MEIMotorStepperPulseTypeCCW	This will enable the pulse output (either A or B) to output a counter-clockwise pulse train. Use it together with MEIMotorStepperPulseTypeCW to provide a complete step interface.

MEIMotorStepperPulseTypeQUADA	This will enable the pulse output (either A or B) to output a quadrature signal (90 degree phase difference to MEIMotorStepperPulseTypeQUADB). Use it together with MEIMotorStepperPulseTypeQUADB to provide a complete quadrature interface.
MEIMotorStepperPulseTypeQUADB	This will enable the pulse output (either A or B) to output a quadrature signal (90 degree phase difference to MEIMotorStepperPulseTypeQUADA). Use it together with MEIMotorStepperPulseTypeQUADA to provide a complete quadrature interface.

See Also

MEIMotorStepperStatus

MEIMotorStepperStatus

```
typedef struct MEIMotorStepperStatus {
    long    pulseLockLost; /* TRUE if Pulse jitter logic has lost lock,
                           otherwise FALSE */
    long    pulseStatus;   /* TRUE if a Pulse was generated incorrectly,
                           otherwise FALSE */
} MEIMotorStepperStatus;
```

Description **MotorStepperStatus** is a structure used to check for error conditions relating to the pulse module.

pulseLockLost	Indicates that the pulse module lost timing lock with the SynqNet network.
pulseStatus	Indicates that the pulse module generated a new pulse when the previous pulse was not finished. It usually indicates that the pulse width is incorrect for the desired velocity.

See Also [MEIMotorStatus](#) | [MEIMotorStepper](#)

MPIMotorType

MPIMotorType

```
typedef enum {  
    MPIMotorTypeINVALID,  
  
    MPIMotorTypeSERVO,  
    MPIMotorTypeSTEPPER,  
  
} MPIMotorType;
```

Description MPIMotorType is an enumeration of valid Motor Types.

MPIMotorTypeSERVO	Motor configured as Servo
MPIMotorTypeSTEPPER	Motor configured as Stepper

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](#)

MEIMotorAmpMsgMAX

MEIMotorAmpMsgMAX

```
#define MEIMotorAmpMsgMAX      (200) /* max chars per motor */
```

Description

MotorAmpMsgMAX defines the maximum number of characters per amp fault message.

See Also

MEIMotorAmpFaultsMAX

MEIMotorAmpFaultsMAX

```
#define MEIMotorAmpFaultsMAX      (32) /* max faults per motor */
```

Description

MotorAmpFaultsMAX defines the maximum number of amp fault messages per motor.

See Also

MEIMotorAmpWarningsMAX

MEIMotorAmpWarningsMAX

```
#define MEIMotorAmpWarningsMAX      (32) /* max Warnings per motor */
```

Description

MotorAmpFaultsMAX defines the maximum number of amp warning messages per motor.

See Also

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 ($< 1\text{msec}$) noise spikes on the limit input will ignore the noise if 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](#)

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 determined by the number of encoder counts (after quadrature) for one revolution of the motor. The **sign** of the countsPerRev is determined 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 (CW) 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 configured 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).

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];

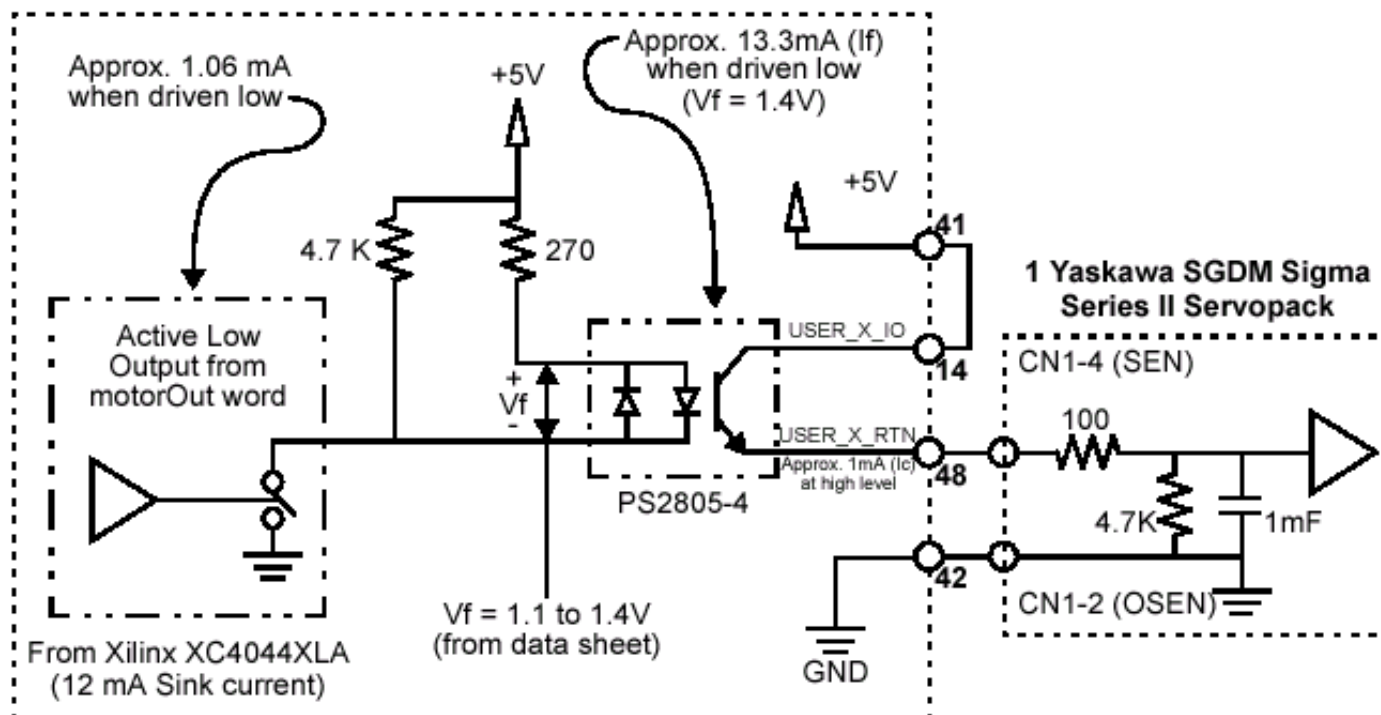
    MEIMotorTransceiver  Transceiver[MEIXmpMotorTransceivers];
    long                 UserOutInvert;    /* Opto Polarity */
    MEIMotorStepper      Stepper;
    long                 EncoderTermination;

    /* Commutation is read-only from field Theta to end*/
    MEIXmpCommutationBlock Commutation;

    MEIXmpLimitData      Limit[MEIXmpLimitLAST];

    MEIMotorFilterInput  FilterInput[MEIXmpMotorFilterInputs];
} MEIMotorConfig;
```

ABS Encoder Support



[Return to Motor Objects](#)

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.

Implementation

Notify objects maintain a FIFO ("First In, First Out") buffer of events that have occurred. Each call to `mpiNotifyEventWait(...)` removes one event from the buffer. If the event buffer is empty, `mpiNotifyEventWait(...)` will wait for an event to be sent to the **Notify** object. This ensures that events will not be missed in cases where multiple events have occurred between calls to `mpiNotifyEventWait(...)`. However, the danger is that an application may not call `mpiNotifyEventWait(...)` for a long time. In this case, the event buffer may grow rather quickly and use a large amount of system memory. In order to prevent this problem from occurring, one should use `mpiNotifyEventMaskSet(...)` to enable and disable event notification at the proper times.

Methods

Create, Delete, Validate Methods

<u>mpiNotifyCreate</u>	Create a Notify object
<u>mpiNotifyDelete</u>	Delete a Notify object
<u>mpiNotifyValidate</u>	Validate a Notify object

Event Methods

<u>mpiNotifyEvent</u>	Check to see if events have occurred
<u>mpiNotifyEventFlush</u>	Flush pending events from event queue
<u>mpiNotifyEventMaskGet</u>	Get event mask
<u>mpiNotifyEventMaskSet</u>	Set event mask
<u>mpiNotifyEventWait</u>	Get next event from queue or wait timeout msec for it to arrive
<u>mpiNotifyEventWake</u>	Wake up thread waiting for notify object

Relational Methods

List Methods for Event Sources

<u>mpiNotifySource</u>	Get the indexth event source in list
<u>mpiNotifySourceAppend</u>	Append an event source to list
<u>mpiNotifySourceCount</u>	Count number of event sources in list
<u>mpiNotifySourceFirst</u>	Get first event source in list
<u>mpiNotifySourceIndex</u>	Get index value for event source in list
<u>mpiNotifySourceInsert</u>	Place event source after source in list
<u>mpiNotifySourceLast</u>	Get last event source in list
<u>mpiNotifySourceListGet</u>	Get list of event sources
<u>mpiNotifySourceListSet</u>	Create a list of event sources
<u>mpiNotifySourceNext</u>	Get next event source after source in list
<u>mpiNotifySourcePrevious</u>	Get the event source before source in list
<u>mpiNotifySourceRemove</u>	Remove event source from list

Data Types

[MPINotifyMessage](#)
[MEINotifyTrace](#)

mpiNotifyCreate

Declaration `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 <i>NotifyDelete</i> 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](#)

Required Header

IF
the type field of **status** matches a bit in the event mask,

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***.

MPIMessageOK	if the Notify object (<i>notify</i>) successfully places the event in a FIFO event queue and signals that an event has been received

file:///D:/pdfs/030100/html/Software-MPI/docs/Notify/Method/evt1.htm [7/23/2004 8:41:51 AM]

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 <i>NotifyEventFlush</i> successfully flushes the pending events from the internal FIFO event queue maintained by the Notify object

See Also

Required Header

Sample Code

Return Values

See Also [mpiNotifyEventMaskSet](#)

Required Header

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***.

```

/*
    Disables event notification and copies the previously used
    event mask to oldMask.  oldMask may then be used to re-enable
    event notification via another call to mpiNotifyEventMaskSet().
*/
void NotifyDisable(MPI_Notify notify, MPI_EventMask* oldMask)
{
    MPI_EventMask newMask;
    long returnValue;

    returnValue = mpi_notify_event_mask_get(notify, oldMask);
    msg_check(returnValue);

    mpi_event_mask_clear(&newMask);

    returnValue = mpi_notify_event_mask_set(notify, &newMask);
    msg_check(returnValue);
}

```

MPIMessageOK	if <i>NotifyEventMaskSet</i> successfully sets the event type(s) for which notification will be accepted by a Notify object

file:///D:/pdfs/030100/html/Software-MPI/docs/Notify/Method/evtmskset1.htm [7/23/2004 8:41:52 AM]

mpiNotifyEventWait

Declaration

long mpiNotifyEventWait([MPINotify](#)
[MPIEventStatus](#)
[MPIWait](#)
notify,
*status,
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](#)

Required Header

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*).

MPIMessageOK	if <i>NotifyEventWake</i> successfully wakes a thread that is waiting for an event notification from a Notify object

file:///D:/pdfs/030100/html/Software-MPI/docs/Notify/Method/evtwk1.htm [7/23/2004 8:41:53 AM]

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	
<i>index</i> th event source	in the event source list maintained by a Notify object (<i>notify</i>)
NULL	if <i>notify</i> is invalid if <i>index</i> is less than 0 if <i>index</i> is greater than or equal to mpiNotifySourceCount (notify)
MPIMessageARG_INVALID	if <i>index</i> is a negative number.
MEIListMessageELEMENT_NOT_FOUND	if <i>index</i> is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	if <i>notify</i> 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 <i>NotifySourceAppend</i> successfully appends source to the list of event <i>sources</i> maintained by a Notify object
MPIMessageHANDLE_INVALID	Either <i>source</i> is NULL or <i>notify</i> 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.

<code>notify</code>	a handle to the Notify object.

Return Values	
number of event sources	in the event source list maintained by a Notify object (<i>notify</i>)
-1	if <i>notify</i> 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 conjunction 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 (<i>notify</i>)
NULL	if <i>notify</i> is invalid or if the event source list is empty
MPIMessageHANDLE_INVALID	if <i>notify</i> is an invalid handle.

See Also [mpiNotifySourceNext](#) | [mpiNotifySourceLast](#)

Required Header

notify	a handle to the Notify object.
source	a pointer with an arbitrary (but non-NULL) value.

index of <i>source</i>	in the event source list maintained by a Notify object (<i>notify</i>)
-1	if <i>notify</i> is invalid if the event source (<i>source</i>) was not found in the event source list

file:///D:/pdfs/030100/html/Software-MPI/docs/Notify/Method/srcinx1.htm [7/23/2004 8:41:54 AM]

mpiNotifySourceInsert

Declaration

```
long mpiNotifySourceInsert(MPINotify notify,  
                           void *source,  
                           void *insert)
```

Required Header

```
stdmpi.h
```

Description	NotifySourceInsert places the event source (pointed to by <i>insert</i>) after a specified event source (<i>source</i>), in the list of event sources that are maintained by a Notify object (<i>notify</i>).

Return Values

MPIMessageOK	if <i>NotifySourceInsert</i> 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 conjunction 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 (<i>notify</i>)
NULL	if <i>notify</i> is invalid if the event source list is empty

See Also [mpiNotifySourcePrevious](#) | [mpiNotifySourceFirst](#)

Required Header

Return Values

See Also [mpiNotifySourceListSet](#) | [NotifySourceAppend](#) | [NotifySourceInsert](#) | [mpiNotifyCreate](#)


```
long mpiNotifySourceListSet(MPI_Notify notify,
                             long sourceCount,
                             void **sourceList)
```

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.

MPIMessageOK	if <i>NotifySourceListSet</i> successfully creates an <i>event source</i> list using the source pointers specified by <i>sourceList</i>

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 conjunction 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 (<i>source</i>) in the event source list maintained by a Notify object (<i>notify</i>)
NULL	if <i>notify</i> is invalid if the event source (<i>source</i>) is the first event source in the event source list
MPIMessageHANDLE_INVALID	Either <i>source</i> is NULL or <i>notify</i> is an invalid handle.

See Also

[mpiNotifySourcePrevious](#)

Required Header

file:///D:/pdfs/030100/html/Software-MPI/docs/Notify/Method/srcpre1.htm [7/23/2004 8:41:55 AM]

Required Header

Return Values

See Also

MPINotifyMessage

MPINotifyMessage

```
typedef enum {

    MPINotifyMessageNOTIFY_INVALID,
    MPINotifyMessageWAIT_IN_PROGRESS,
} MPINotifyMessage;
```

Description

MPINotifyMessageNOTIFY_INVALID

The notify object is not valid. This message code is returned by a notify method if the notify object handle is not valid. This problem can be caused by failed [mpiNotifyCreate\(...\)](#). To prevent this problem, check your notify objects after creation by using [mpiNotifyValidate\(...\)](#).

MPINotifyMessageWAIT_IN_PROGRESS

The notify object is waiting for an event. This message code is returned by [mpiNotifyEventWait\(...\)](#) if the notify object is already waiting for an event in another thread. To prevent this problem, make sure a thread does not share notify objects with other threads.

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

[mpiObjectValidate](#)

Configuration and Information Methods

[mpiObjectModuleId](#)

[mpiObjectTimeoutGet](#)

[mpiObjectTimeoutSet](#)

[meiObjectTraceGet](#)

[meiObjectTraceSet](#)

Data Types

[MPIOBJECTMAP](#)

Macros

[mpiObjectMapAND_ASSIGN](#)

[mpiObjectMapASSIGN](#)

[mpiObjectMapBitCountMAX](#)

[mpiObjectMapBitGET](#)

[mpiObjectMapBitSET](#)

[mpiObjectMapCLEAR](#)

[mpiObjectMapCOMPLEMENT](#)

[mpiObjectMapIS_CLEAR](#)

[mpiObjectMapIS_EQUAL](#)

[mpiObjectMapIS_VALID](#)

[mpiObjectMapMAX](#)

[mpiObjectMapOR_ASSIGN](#)

[meiObjectTraceGET](#)

[meiObjectTraceSET](#)

mpiObjectValidate

Declaration

```
long mpiObjectValidate(MPIHandle handle,  
                       MPIModuleId moduleId)
```

Required Header

Description	The ObjectValidate method verifies that the object (handle) is of the type moduleId.

Return Values

MPIMessageOK	if <i>handle</i> is a handle to a valid object of type <i>moduleId</i> .

See Also

mpiObjectModuleId

```
long mpiObjectModuleId(MPIHandle handle, MPIModuleId *moduleId)
```

Required Header

Description	The ObjectModuleId function writes the MPIModuleId of the object (<i>handle</i>) to the location pointed to by <i>moduleId</i> .

Return Values

MPIMessageOK	if <i>ObjectModuleId</i> successfully writes the MPIModuleId to the location

See Also

Required Header

Return Values

See Also [mpiObjectTimeoutSet](#)

Required Header

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.

Return Values	
MPIMessageOK	if ObjectTimeoutSet successfully sets the timeout value of the object
MPIMessageTIMEOUT	if a timeout occurs before the shared resource becomes available

file:///D:/pdfs/030100/html/Software-MPI/docs/Object/Method/timeoutset1.htm [7/23/2004 10:45:10 AM]

```
long meiObjectTraceGet(MPIHandle handle,
                       MEITraceMask *traceMask)
```

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

MPIMessageOK	if <i>ObjectTraceGet</i> successfully gets the trace mask for an object.

Declaration

```
#define meiObjectTraceGET(object) (((MEIObject)(object))->trace)
```

Description	ObjectTraceGET
	gets the object's global trace mask.

See Also [meiObjectTraceSET](#) | [meiObjectTraceSet](#)

```
Declaration      long meiObjectTraceSet(MPIHandle handle,
                        MEITraceMask traceMask)
```

Description	ObjectTraceSet sets an Object's trace mask to the value specified by <i>traceMask</i> .

handle	a handle to an object
traceMask	the value of an object's trace mask

MPIMessageOK	if <i>ObjectTraceSet</i> successfully sets the trace mask for an object.

Declaration

```
#define meiObjectTraceSET(object,mask) \
        (((MEIObject)(object))->trace = (mask))
```

Required Header

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 (objectMap)
		(sizeof(objectMap) * 8)

Required Header

Description	ObjectMapBitCountMAX calculates the maximum bit count for the specified object <i>objectMap</i> .

See Also

mpiObjectMapBitGET

Declaration

```
#define mpiObjectMapBitGET(objectMap, bit) (((objectMap) & (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

[illegible]

Required Header

Description	ObjectMapBitSET sets object map's specified <i>bit</i> number to the specified <i>value</i> .

See Also [mpiObjectMapBitGET](#)

mpiObjectMapCLEAR

Declaration `#define mpiObjectMapCLEAR(objectMap) ((objectmap) = 0x0)`

Required Header `stdmpi.h`

Description **ObjectMapCLEAR** sets the object *map* to zero.

See Also

mpiObjectMapCOMPLEMENT

Declaration `#define mpiObjectMapCOMPLEMENT(objectMap)`
`((objectMap) = ~(objectMap))`

Required Header

Description	ObjectMapCOMPLEMENT performs bitwise complement to the object map and assigns the result to <i>objectMap</i> .

See Also

mpiObjectMapIS_CLEAR

Declaration `#define mpiObjectMapIS_CLEAR(objectMap) ((objectMap) == 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(objectMap, count) \
                (((count) >= (sizeof(objectMap) * 8)) || \
                (((~((0x1 << (count)) - 1)) & (objectMap)) == 0))
```

Required Header

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(objectMap, count)  
    ((objectMap) = (0x1 << (count)) - 1)
```

Required Header stdmpi.h

Description **ObjectMapMAX** calculates the maximum object ***objectMap*** 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

<u>mpiPathCreate</u>	Create a Path object
<u>mpiPathDelete</u>	Delete a Path object
<u>mpiPathValidate</u>	Validate a Path object

Configuration and Information Methods

<u>mpiPathParamsGet</u>
<u>mpiPathParamsSet</u>

Relational Methods

<u>mpiPathAppend</u>

Action Methods

<u>mpiPathMotionParamsGenerate</u>

Data Types

[MPIPath**Arc**](#)

[MPIPath**ArcCenter**](#)

[MPIPath**ArcEndPoint**](#)

[MPIPath**Attr**](#)

[MPIPath**Direction**](#)

[MPIPath**Element**](#)

[MPIPath**ElementAttributes**](#)

[MPIPath**ElementAttrMask**](#)

[MPIPath**ElementType**](#)

[MPIPath**Line**](#)

[MPIPath**Message**](#)

[MPIPath**Params**](#)

[MPIPath**Point**](#)

Macros

[mpiPath**ElementType**](#)

[mpiPath**ElementAttrMaskBIT**](#)

[mpiPath**ElementATTR**](#)

Constants

[MPIPathPoint**DIMENSION_MAX**](#)

mpiPathCreate

Declaration [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 <i>PathDelete</i> 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

Description	PathParamsGet reads the parameters for a path object and writes them into the structure pointed to by <i>params</i> , and also writes it into the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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 <i>PathParamsGet</i> successfully reads the path parameters.

See Also [mpiPathParamsSet](#)

mpiPathParamsSet

Declaration

long **mpiPathParamsSet**([MPIPath](#)
[MPIPathParams](#)
void

path,
***params,**
***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

[illegible]

Required Header

Description	PathAppend adds an array of path elements pointed to by <i>element</i> to the end of the <i>path</i> stored in the Path object.

Return Values

MPIMessageOK	if <i>PathAppend</i> successfully adds the elements to the path.

See Also [mpiPathCreate](#) | [mpiPathMotionParamsGenerate](#)

Required Header

To create a path, use [`mpiPathCreate\(...\)`](#) to create a path object. Initialize the path parameters with [`mpiPathParamsGet\(...\)`](#) / [`mpiPathParamsSet\(...\)`](#). Then add path elements (line, arc, etc.) with [`mpiPathAppend\(...\)`](#). Before calling `mpiPathMotionParamsGenerate(...)` make sure to specify the [`MPIMotionPoint{...}`](#) values in the [`MPIMotionParams`](#) structure. It is very important to set `point.final = TRUE` or `FALSE` before calling `mpiPathMotionParamsGenerate(...)`.

path	a handle to a Path object
*params	a pointer to a MPIMotionParams structure.

Return Values

See Also [mpiPathCreate](#) | [mpiPathParamsGet](#) | [mpiPathParamsSet](#) | [mpiPathAppend](#) | [mpiMotionStart](#)

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      = -1,

    MPIPathElementAttrRELATIVE     = MPIPathElementAttrLAST - 1,
    MPIPathElementAttrID           = MPIPathElementAttrLAST - 2,
    MPIPathElementAttrVELOCITY     = MPIPathElementAttrLAST - 3,
    MPIPathElementAttrACCEL        = MPIPathElementAttrLAST - 4,
    MPIPathElementAttrTIMESLICE    = MPIPathElementAttrLAST - 5,
    MPIPathElementAttrCOUNT       = MPIPathElementAttrLAST -
MPIPathElementAttrFIRST,
} 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    = -1,  
    MPIPathDirectionCCW = 1,  
} 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;
        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.
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](#)

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. The id is limited to 16-bit resolution by the controller firmware.
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](#)

MPIPathElementAttrMask

MPIPathElementAttrMask

```
typedef enum {
    MPIPathElementAttrMaskRELATIVE,    =
mpiPathElementAttrMaskBIT(MPIPathElementAttrRELATIVE),
    MPIPathElementAttrMaskID,         =
mpiPathElementAttrMaskBIT(MPIPathElementAttrID),
    MPIPathElementAttrMaskVELOCITY,   =
mpiPathElementAttrMaskBIT(MPIPathElementAttrVELOCITY),
    MPIPathElementAttrMaskACCEL,      =
mpiPathElementAttrMaskBIT(MPIPathElementAttrACCEL),
    MPIPathElementAttrMaskTIMESLICE,  =
mpiPathElementAttrMaskBIT(MPIPathElementAttrTIMESLICE),

    MPIPathElementAttrMaskALL          = -1 << MPIPathElementAttrFIRST,
} 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](#)

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

The path object is not valid. This message code is returned by a path method if the path object handle is not valid. This problem can be caused by a failed [mpiPathCreate\(...\)](#). To prevent this problem, check your path objects after creation by using [mpiPathValidate\(...\)](#).

MPIPathMessageILLEGAL_DIMENSION

The path dimensions are not valid. This message code is returned by [mpiPathParamsSet\(...\)](#) or [mpiPathMotionParamsGenerate\(...\)](#) if the path dimension is less than one or greater than or equal to MPIPathPointDIMENSION_MAX. Also, this message code is returned if specific path element types have dimension restrictions. For example, the ARC type is limited to 2 dimensions and the ARC_END_POINT type is limited to 3 dimensions. To correct this problem, select an appropriate dimension for the path element type.

MPIPathMessageILLEGAL_ELEMENT

The path element type is not valid. This message code is returned by [mpiPathAppend\(...\)](#) if the specified path element type is not a member of the [MPIPathElementType](#) enumeration.

MPIPathMessageARC_ILLEGAL_DIMENSION

The path element arc dimension is not valid. This message code is returned by [mpiPathAppend\(...\)](#) if the ARC or ARC_CENTER element is not 2 dimensions. To correct this problem, set the path dimension to 2.

MPIPathMessageHELIX_ILLEGAL_DIMENSION

Not supported.

MPIPathMessageILLEGAL_RADIUS

The path element arc radius is not valid. This message code is returned by [mpiPathAppend\(...\)](#) if the ARC element radius is less than or equal to zero. To correct this problem, set the arc radius to a value greater than zero.

MPIPathMessagePATH_TOO_LONG

The path length is not valid. This message code is returned by [mpiPathMotionParamsGenerate\(...\)](#) if the path length is greater than MAX_PATH_POINTS. To correct the problem, specify a path with fewer points than MAX_PATH_POINTS.

MPIPathMessageILLEGAL_VELOCITY

The path element velocity is not valid. This message code is returned by [mpiPathAppend\(...\)](#) if the specified velocity is less than or equal to zero. To correct this problem, set the element velocity to a value greater than zero.

MPIPathMessageILLEGAL_ACCELERATION

The path element velocity is not valid. This message code is returned by [mpiPathAppend\(...\)](#) if the specified velocity is less than or equal to zero. To correct this problem, set the element velocity to a value greater than zero.

MPIPathMessageILLEGAL_TIMESLICE

The path element time slice is not valid. This message code is returned by [mpiPathAppend\(...\)](#) if the specified time slice is less than or equal to zero. To correct this problem, set the element time slice to a value greater than zero.

MPIPathMessageINVALID_BLENDING

The path element blending is not valid. This message code is returned by [mpiPathMotionParamsGenerate\(...\)](#) if the element blending is set to TRUE and the motion type does not support blending. To correct this problem, either set the element blending to FALSE or select a different 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.
conversion	This value is an $N \times N$ matrix (where N is the number of dimensions in the path motion) that scales and rotates the axes used in the path motion. This is useful when using two axes with different resolutions for each axis. For two axes with different resolution: Set conversion[0][0] to (desired x resolution / actual x resolution) Set conversion [1][1] to (desired y resolution / actual y resolution) Set conversion[0][1] and conversion[1][0] = 0 For a coordinate rotation, where alpha is the rotation of the coordinate system: Set conversion[0][0] and conversion [1][1] = cos(alpha) Set conversion[0][1] = sin(alpha) Set conversion[1][0] = -sin(alpha)

See Also [mpiPathParamsGet](#) | [mpiPathParamsSet](#) | [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](#) | [mpiPathParamsSet](#) | [mpiPathPointDIMENSION_MAX](#)

mpiPathElementType

mpiPathElementType

```
#define mpiPathElementType(type) ((type) & MPIPathElementTypeMASK)
```

Description

PathElementType is a macro that masks off all other bits in type, leaving the path element type.

See Also

[MPIPathElementType](#)

mpiPathElementAttrMaskBIT

mpiPathElementAttrMaskBIT

```
#define mpiPathElementAttrMaskBIT(attr) (0x1 << (attr))
```

Description

PathElementAttrMaskBIT is a macro that converts the path element attribute into the path element attribute mask.

See Also

[MPIPathElementAttrs](#) | [MPIPathElementAttrMask](#)

mpiPathElementATTR

mpiPathElementATTR

```
#define mpiPathElementATTR(type, attr)
                                ((type) |= mpiPathElementAttrMaskBIT\(attr\))
```

Description [PathElementATTR](#) is a macro that turns on the specified path element attribute mask bits in the path element type.

See Also [MPIPathAttr](#) | [MPIPathElementAttrMask](#)

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

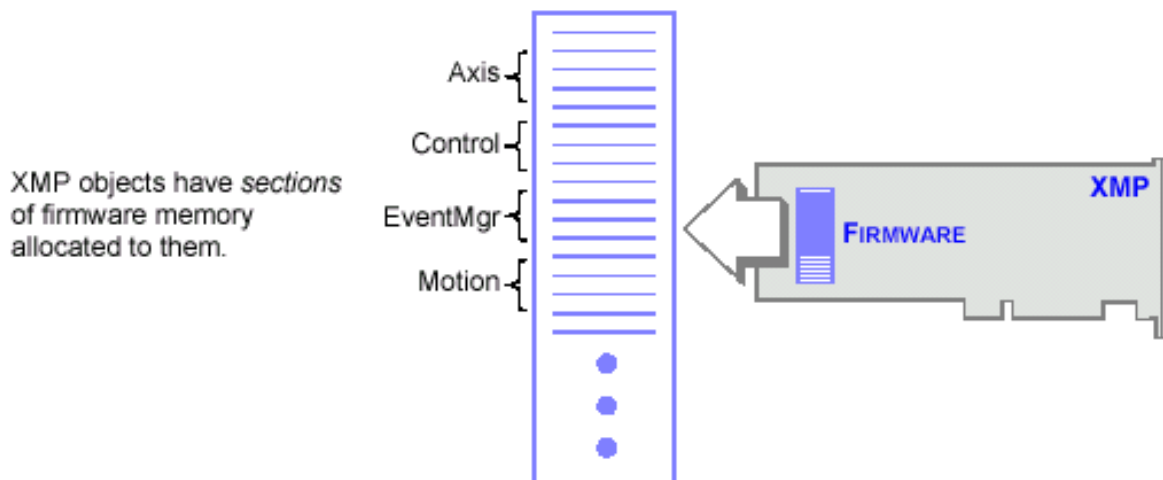
The **Platform** module provides a common interface to platform-specific functionality, such as memory allocation, resource locking, interrupts, signalling, and others.

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. The Platform module was written to provide platform-independent access functions for use by the MPI. Unless your application needs to be written for compatibility with different platforms, MEI encourages the use of OS-specific functions. If an MEIPlatform object handle is required, one should obtain this handle from the MPIControl method [meiControlPlatform\(\)](#).

WARNING!

Do NOT attempt to use the (intentionally undocumented) method, `meiPlatformCreate()`. Using this method will interfere with the inner workings of the MPI.

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

[meiPlatformAlloc](#)

Allocate system memory.

[meiPlatformAssertSet](#)

Set an assertion handling function to be used by the MPI library.

[meiPlatformAtof](#)

Convert a numeric string to a double.

[meiPlatformAtol](#)

Convert a numeric string to a long.

[meiPlatformFileClose](#)

Close a file handle.

[meiPlatformFileOpen](#)

Open a file handle.

[meiPlatformFileRead](#)

Read data from a file handle created by meiPlatformFileOpen.

[meiPlatformFileWrite](#)

Writes data to a file whose handle was created by meiPlatformFileOpen.

[meiPlatformFree](#)

Free system memory.

[meiPlatformKey](#)

Return an input character if an input character is available.

[meiPlatformMemoryToFirmware](#)

Convert a host memory address to a controller memory address.

[meiPlatformMemoryToHost](#)

Covert a controller memory address to a host memory address.

[meiPlatformSleep](#)

Put the current thread to sleep for the number of milliseconds specified.

[meiPlatformProcessId](#)

Return the process identification number of the current process.

[meiPlatformTimerCount](#)

Write to ticks the current timer count.

[meiPlatformTimerFrequency](#)

Write to frequency the timer frequency of the current platform.

[meiPlatformTrace](#)

Display printf(...)-style trace information

[meiPlatformTraceEol](#)

Set the end-of-line (eol) to be used by meiPlatformTrace(...).

[meiPlatformTraceFile](#)

Redirect trace output.

[meiPlatformTraceFunction](#)

Display the trace output.

Data Types

[MEIPlatformBoardType](#)[MEIPlatformFileMode](#)[MEIPlatformMessage](#)

meiPlatformAlloc

Declaration void **meiPlatformAlloc**(long **size**)

Required Header stdmei.h

Description **PlatformAlloc** allocates system memory. meiPlatformAlloc will return NULL upon failure to allocate memory.

size the number of bytes to allocate.

See Also [meiPlatformFree](#)

Required Header

When an assertion occurs, the filename and line number where that assertion happened will be passed to the assertion handling function. If no assertion handling function is set, information about the assertion will be reported to stdout and the C function, `exit()` will be called with an argument of 1.

NOTE: `meiPlatformAssertSet` is only fully implemented for Windows operating systems.

See Also

meiPlatformAtof

Declaration `double meiPlatformAtof(const char *ascii)`

Required Header `stdmei.h`

Description **PlatformAtof** converts a numeric string to a double. This function returns the converted value as a double.

*ascii	string to be converted

Returns	converted the numeric text string <code>ascii</code> to a <i>long</i> and returned it

See Also [meiPlatformAtol](#)

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 <i>long</i> and returned it

See Also [meiPlatformAtof](#)

meiPlatformFileClose

Declaration long **meiPlatformFileClose**(long **file**)

Required Header stdmei.h

Description **PlatformFileClose** closes a file handle. meiPlatformFileClose is a platform independent replacement for the C function fclose().

file the file handle to be closed.

See Also [meiPlatformFileOpen](#)

meiPlatformFileOpen

Declaration

long **meiPlatformFileOpen**(const char *file_name, meiPlatformFileMode mode)

Required Header

stdmei.h

Description

PlatformFileOpen opens a file handle. meiPlatformFileOpen is a platform independent replacement for the C function fopen().

file_name	the name of the file to open.
mode	the access mode used to open the file. Different MEIPlatformFileMode values may be or'ed together to produce the desired mode. For example: MEIPlatformFileModeREAD MEIPlatformFileModeBINARY

Returns

meiPlatformFileOpen returns the newly created file handle.

See Also

[meiPlatformFileClose](#) | [meiPlatformFileRead](#) | [meiPlatformFileWrite](#) | [MEIPlatformFileMode](#)

meiPlatformFileRead

```
Declaration      long  meiPlatformFileRead( long  file,
                                char  *buffer,
                                long  byteCount )
```

Required Header

Description	PlatformFileRead reads data from a file handle created by <code>meiPlatformFileOpen</code> . <code>meiPlatformFileRead</code> is a platform independent replacement for the C function <code>fread()</code> .

file	the handle of the file from which data will be read.
buffer	the storage location where data from the file will be written to.
byteCount	the number of bytes to read from the file.

See Also [meiPlatformFileOpen](#) | [meiPlatformFileWrite](#)

meiPlatformFileWrite

```
Declaration      long  meiPlatformFileWrite(long      file,
                                const char *buffer,
                                long      byteCount)
```

Required Header

Description	PlatformFileWrite writes data to a file whose handle was created by <code>meiPlatformFileOpen</code> . <code>meiPlatformFileWrite</code> is a platform independent replacement for the C function <code>fwrite()</code> .

file	the handle of the file to which data will be written.
buffer	the location of the data to be written to the file.
byteCount	the number of bytes to be written to the file.

See Also [meiPlatformFileOpen](#) | [meiPlatformFileRead](#)

meiPlatformFree

Declaration long **meiPlatformFree**(void ***alloc,**
 long **size)**

Required Header stdmei.h

Description **PlatformFree** frees system memory.

alloc	the address of the memory to free.
size	the number of bytes to free. This must match the size specified for meiPlatformAlloc when alloc was allocated.

See Also [meiPlatformAlloc](#)

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.

NOTE:
meiPlatformKey is not fully implemented for all operating sytems. For example, in VentureCom's RTX Windows Extensions, a keystroke will be simulated after 10 seconds.

<i>If "wait" is</i>	<i>Then</i>
MPIWaitFOREVER (-1)	<i>PlatformKey</i> will wait for an input character forever
MPIWaitPOLL (0)	<i>PlatformKey</i> will return immediately
a value (not -1 or 0)	<i>PlatformKey</i> will wait for an input character for wait milliseconds

Return Values	
-1	if no input character was available
0	<i>PlatformKey</i> 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

meiPlatformMemoryToFirmware

Declaration

long **meiPlatformMemoryToFirmware**([MEIPlatform](#)
void
void

platform,
***host,**
****firmware)**

Required Header

stdmei.h

Description

PlatformMemoryToFirmware converts a host memory address to a controller memory address.

platform	the handle to the controller's platform object. This should be obtained from meiControlPlatform().
host	the host memory address to be converted.
firmware	the location where the controller's memory address will be written.

See Also

[meiPlatformMemoryToHost](#) | [meiControlPlatform](#)

meiPlatformMemoryToHost

Declaration

long **meiPlatformMemoryToHost**([MEIPlatform](#)
void
void

platform,
***firmware,**
****host)**

Required Header

stdmei.h

Description

PlatformMemoryToHost converts a controller memory address to a host memory address.

platform	the handle to the controller's platform object. This should be obtained from meiControlPlatform().
firmware	the controller memory address to be converted.
host	the location where the host memory address will be written.

See Also

[meiPlatformMemoryToFirmware](#) | [meiControlPlatform](#)

meiPlatformSleep

Declaration void **meiPlatformSleep**(long **milliseconds**)

Required Header stdmei.h

Description **PlatformSleep** puts the current thread to sleep for the number of *milliseconds* specified.

NOTE:

Different platforms have different time slice "quanta" (minimum sleep resolution) for threads. For example, Windows NT, 2000, and XP have a quanta of 10ms. For example, even if meiPlatformSleep(2) is specified, the actual sleep period will essentially be equivalent to meiPlatformSleep(10).

milliseconds	the number of milliseconds for which to put the current thread to sleep

Returns

converted the numeric text string ascii to a *long* and returned it

See Also

meiPlatformProcessId

Declaration long **meiPlatformProcessId**(void)

Required Header stdmei.h

Description **PlatformProcessId** returns the process identification number of the current process.

See Also

meiPlatformTimerCount

Declaration long **meiPlatformTimerCount**(long ***ticks**)

Required Header stdmei.h

Description **PlatformTimerCount** reads the host CPU's timer value and writes it into the contents of a long pointed to by ***ticks***. The resolution of the platform timer can be determined with meiPlatformTimerFrequency(...).

*ticks	a pointer to the timer value for the host CPU.

See Also [meiPlatformTimerFrequency](#)

meiPlatformTimerFrequency

Declaration long `meiPlatformTimerFrequency`(long ***frequency**)

Required Header stdmei.h

Description **PlatformTimerFrequency** reads the host CPU's timer frequency and writes it into the contents of a long pointed to by ***frequency***. The platform timer value can be read with `meiPlatformTimerCount(...)`.

*frequency	a pointer to the timer frequency for the host CPU.

See Also [meiPlatformTimerCount](#)

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 <i>PlatformTrace</i> 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 <i>PlatformTraceFile</i> successfully closes any <i>previously opened</i> trace file and redirects trace output to <i>fileName</i>

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 <i>traceFunction</i>	if there is a previous function
NULL	if no <i>traceFunction</i> has been specified (the default trace function is used)

See Also [meiPlatformTrace](#)

MEIPlatformBoardType

MEIPlatformBoardType

```
typedef enum {  
    MEIPlatformBoardTypeUNKNOWN,  
    MEIPlatformBoardTypeXMP,  
    MEIPlatformBoardTypeZMP,  
} MEIPlatformBoardType;
```

Description **PlatformBoardType** is the type of motion controller card that is being used.

MEIPlatformBoardTypeUNKNOWN	Board is not being recognized.
MEIPlatformBoardTypeXMP	An XMP Motion Controller board has been recognized.
MEIPlatformBoardTypeZMP	A ZMP Motion Controller board has been recognized.

See Also

MEIPlatformFileMode

MEIPlatformFileMode

```
typedef enum {
    MEIPlatformFileModeREAD,      /* default */
    MEIPlatformFileModeWRITE,
    MEIPlatformFileModeTEXT,      /* default */
    MEIPlatformFileModeBINARY,
    MEIPlatformFileModeTRUNC,
    MEIPlatformFileModeAPPEND,
} MEIPlatformFileMode;
```

Description **PlatformFileMode** is an enumeration that is used as an argument for methods that open files.

MEIPlatformFileModeREAD	Open a file for read access
MEIPlatformFileModeWRITE	Open a file for write access
MEIPlatformFileModeTEXT	Open a file as text format
MEIPlatformFileModeBINARY	Open a file as binary format
MEIPlatformFileModeTRUNC	Truncate existing file or create for reading and writing
MEIPlatformFileModeAPPEND	Open existing file for appending all writes

See Also [meiPlatformFileOpen](#)

MEIPlatformMessage

MEIPlatformMessage

```
typedef enum {

    MEIPlatformMessagePLATFORM_INVALID,
    MEIPlatformMessageDEVICE_INVALID,
    MEIPlatformMessageDEVICE_ERROR,
    MEIPlatformMessageDEVICE_MAP_ERROR,
} MEIPlatformMessage;
```

Description

MEIPlatformMessagePLATFORM_INVALID

The platform object is not valid. This message code is returned by a platform method if the platform object handle is not valid. Most applications do not use the platform module. The MPI library uses the platform module internally. If an application needs a platform handle, use [meiControlPlatform\(...\)](#). Do NOT create your own platform object with `meiPlatformCreate(...)`.

MEIPlatformMessageDEVICE_INVALID

The platform device driver is not valid. This message code is returned by [mpiControlInit\(...\)](#) or [mpiControlReset\(...\)](#) if the platform device handle is not valid. This message code comes from the lower level routines, `meiPlatformInit(...)` or `meiPlatformDeviceClose(...)`. To correct the problem, make sure the device driver is installed and operational.

MEIPlatformMessageDEVICE_ERROR

The platform device failed. This message code is returned by the platform methods that fail to access a controller via a device driver. It occurs if the specified board type is not a member of the `MEIPlatformBoardType` enumeration. It also occurs if the device driver fails to read/write controller memory or there is an interrupt handling failure. To correct the problem, verify the platform has support for your controller and the device drive is installed and operational. Check for any resource conflicts (memory range, I/O port range, and interrupts) with other devices.

MEIPlatformMessageDEVICE_MAP_ERROR

The platform device memory mapping failed. This message code is returned by [mpiControlInit\(...\)](#) or [mpiControlReset\(...\)](#) if the controller memory could not be mapped to the operating system's memory space. To correct this problem, verify there are no memory resource conflicts. Also, make sure the host computer and operating system have enough free memory for the controller (XMP-Series requires 8 Mbytes).

See Also

Probe Objects

Introduction

A Probe object manages a hardware logic block. The Probe hardware can latch and store up to 16 motor positions or node clock values within one controller sample period. The latches can be triggered by one of the configurable input sources (home, index, +/- limits, or one of the first 16 general purpose motor I/O).

The Probe is used in applications that need multiple data captures within one controller sample period. For applications that do not need multiple latches per sample, the Capture object is a much better solution. For example, a homing routine should use Capture (not Probe). Homing usually requires one or two precise position latches triggered by a home sensor and/or encoder index input to calibrate the position feedback to a physical location. The Probe is useful in high speed scanning applications, where the input can trigger at rates higher than the controller's sample rate. In this case, the Probe can be used with the Data Recorder to collect bursts of scan data. Later, the application can process the data.

Probe data is in raw position feedback counts or SynqNet node clock ticks. Typically, the high-speed Probe data and the sampled position data are collected with the Data Recorder. Later, the data is processed and the probed positions are calculated or interpolated from the time values.

There are two Probe data types: Position and Time. For the position data type, the lower 16 bits of the position counters are latched in the FPGA. This methodology works well for incremental quadrature encoders. For the time data type, the FPGA latches the clock value. The application must read the position from the controller's previous sample and the present sample, to interpolate the probe position. This methodology works well for cyclic feedback data that is digitally transmitted from the drive to the FPGA. Many drives have proprietary serial encoders that decode the encoder position and send the position information to the FPGA once per sample. In these cases, time-based probing is more accurate than position based probing.

For the position data type, the trigger source input and the feedback must be on the same motor.

For the time data type, the trigger source input and the clock must be on the same node. But, since SynqNet nodes are synchronized to the controller's clock, the position values can be interpolated across multiple nodes. Thus, a single probe input used with the Data Recorder can be used to collect/interpolate positions for several axes, across several nodes.

Methods

[mpiProbeConfigGet](#)

[mpiProbeConfigSet](#)

[mpiProbeControl](#)

[mpiProbeCreate](#)

[mpiProbeDelete](#)

[mpiProbeInfo](#)

[mpiProbeParams](#)

[mpiProbeStatus](#)

[mpiProbeValidate](#)

Data Types

[MEIProbeAddress](#)

[MPIProbeConfig](#)

[MPIProbeData](#)

[MPIProbeInfo](#)

[MPIProbeMessage](#) / [MEIProbeMessage](#)

[MPIProbeNumber](#)

[MPIProbeParams](#)

[MPIProbeSource](#)

[MPIProbeStatus](#)

[MEIProbeTrace](#)

[MPIProbeType](#)

Constants

[MPIProbeMaxProbeRegisters](#)

mpiProbeConfigGet

Declaration

```
mpiProbeConfigGet(MPIProbe probe,  
                  MPIProbeConfig *config,  
                  void *external);
```

Required Header

```
stdmei.h
```

Description

ProbeConfigGet reads a Probe's configuration and writes it into the structure pointed to by *config*, and also into the implementation specific structure pointed to by *external* (if external is not NULL).

external is reserved for future functionality and should be set to NULL.

probe	a handle to a Probe object.
*config	a pointer to a Probe configuration structure.
*external	a pointer to an implementation specific structure.

Return Values	
MPIMessageOK	if <i>ProbeConfigGet</i> successfully reads the Probe configuration from the controller.
MPIProbeMessagePROBE_INVALID	if the specified Probe params or configurations are out of range.

See Also

[mpiProbeConfigSet](#)

mpiProbeConfigSet

Declaration

```
mpiProbeConfigSet(MPIProbe probe,  
                  MPIProbeConfig *config,  
                  void *external);
```

Required Header

stdmei.h

Description

ProbeConfigSet sets a Probe's 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).

external is reserved for future functionality and should be set to NULL.

probe	a handle to a Probe object.
*config	a pointer to a Probe configuration structure.
*external	a pointer to an implementation specific structure. <i>external</i> is reserved for future functionality and should be set to NULL.

Return Values

MPIMessageOK	if <i>ProbeConfigSet</i> successfully wrote the Probe configuration to the controller.
MPIProbeMessagePROBE_TYPE_INVALID	if the specified Probe data type is not one of the enumerated values
MPIProbeMessagePROBE_INVALID	if the specified Probe params or configurations are out of range.

See Also

[mpiProbeConfigGet](#)

mpiProbeControl

Declaration `mpiProbeControl(MPIProbe probe);`

Required Header `stdmei.h`

Description `ProbeControl` returns a handle to the Control object with which the Probe object is associated.

<code>probe</code>	a handle to a Probe object.

Return Values

<code>MPIControl</code>	handle to a Control object.
<code>MPIHandleVOID</code>	if Probe is not valid

See Also [mpiProbeCreate](#) | [mpiControlCreate](#)

mpiProbeDelete

Declaration `mpiProbeDelete(MPIProbe probe);`

Required Header `stdmei.h`

Description [ProbeDelete](#) deletes a Probe object and invalidates its handle.

ProbeDelete is the equivalent of a C++ constructor.

probe	a handle to a Probe object.

Remarks

All objects that are created must be deleted in the reverse order to avoid memory leaks.

Return Values

MPIMessageOK	if <i>ProbeDelete</i> successfully deletes a Probe object and invalidates its handle.

See Also [mpiProbeCreate](#) | [mpiProbeValidate](#)

meiProbeInfo

Declaration

`meiProbeInfo(MPIProbe probe,
MEIProbeInfo *info);`

Required Header

`stdmei.h`

Description

[ProbeControl](#) reads a Probe's information and writes it into a structure pointed to by *info*. The *info* structure contains read only data about the probe engine.

probe	a handle to a Probe object.
*info	contains read only data about the probe engine.

Return Values	
MPIMessageOK	

See Also

mpiProbeParams

Declaration

`mpiProbeParams(MPIProbe probe,
MPIProbeParams *params);`

Required Header

`stdmei.h`

Description

ProbeParams reads the parameters of a Probe object and writes it to the structure pointed to by params. The Probe parameters are set with [mpiProbeCreate\(...\)](#).

probe	a handle to a Probe object.
*params	a pointer to a Probe parameter's structure.

Return Values

MPIMessageOK	

See Also

[mpiProbeStatus](#) | [mpiProbeCreate](#)

mpiProbeStatus

Declaration

`mpiProbeStatus(MPIProbe probe, MPIProbeStatus *status, void *external);`

Required Header

`stdmei.h`

Description

ProbeStatus reads status from the Probe and writes it into the structure pointed to by *status* and also writes it into the implementation-specific structure pointed to be *external* (if external is not NULL). The Probe status structure contains data from the probe engine, including digital I/O states and the latched probe values.

external is reserved for future functionality and should be set to NULL.

probe	a handle to a Probe object.
*status	a pointer to a Probe status structure.
*external	a pointer to an implementation specific structure. <i>external</i> is reserved for future functionality and should be set to NULL.

Return Values

MPIMessageOK	

See Also

[meiProbeInfo](#) | [mpiProbeParams](#)

mpiProbeValidate

Declaration `mpiProbeValidate(MPIProbe                      probe);`

Required Header `stdmei.h`

Description **ProbeValidate** validates the Probe object and its handle. ProbeValidate should be called immediately after an object is created.

probe	a handle to a Probe object.

Return Values

MPIMessageOK	if Probe is a handle to a valid object.

See Also [mpiProbeCreate](#) | [mpiProbeDelete](#)

MEIProbeAddress

MEIProbeAddress

```
typedef struct MEIProbeAddress {
    long    *primaryPosition;
    long    *secondaryPosition;
    long    *time;
    long    *status;
    short   *data;
} MEIProbeAddress;
```

Description

The **ProbeAddress** structure contains the controller addresses for the Probe data fields. The addresses in this structure are useful for configuring the Recorder to collect data during probing.

*primaryAddress	a pointer to the address for the Motor's primary position feedback.
*secondaryPosition	a pointer to the address for the Motor's secondary position feedback.
*time	a pointer to the address for the feedback time value.
*status	a pointer to the address for the Probe status
*data	a pointer to the address for the Probe data.

See Also

[MEIProbeInfo](#)

MPIProbeConfig

MPIProbeConfig

```
typedef struct MPIProbeConfig {
    long          enable;          /* TRUE/FALSE */
    MPIProbeSource source;
    MPIProbeData  data;
    long          inputFilter;     /* TRUE/FALSE */
} MPIProbeConfig;
```

Description

The **ProbeConfig** structure specifies the Probe's configuration.

enable	Enables or disables the Probe triggering. A value of TRUE enables the Probe triggering, FALSE disables Probe triggering.
source	An enumerated Probe trigger source input. A Probe can be configured to trigger from one input source.
data	An enumerated Probe data type. A probe can be configured to collect position or time data from a motor's feedback.
inputFilter	Enables or disables the source input filtering. The hardware Probe engine has a 4 state digital filter to eliminate noise on the input signal. When enabled, the Probe will not trigger until the input signal is stable for 4 clock periods. For example, the MEI-RMB clock is 25Mhz. When the inputFilter is enabled, the input signal must transition to and remain at either a high or low state for 160 nanoseconds to trigger the Probe.

See Also

[meiProbeConfigGet](#) | [meiProbeConfigSet](#)

MPIProbeData

MPIProbeData

```
typedef enum MPIProbeData {
    MPIProbeDataPOSITION_PRIMARY,
    MPIProbeDataPOSITION_SECONDARY,
    MPIProbeDataTIME,
} MPIProbeData;
```

Description

ProbeData is an enumeration of Probe data types. This specifies to the probe engine what data to collect when triggered.

MPIProbeDataPOSITION_PRIMARY	Position from the motor's primary feedback
MPIProbeDataPOSITION_SECONDARY	Position from the motor's secondary feedback
MPIProbeDataTIME	Clock value from the node. The clock value can be used to derive the position by interpolating between the positions from the previous sample and the present sample period and dividing by the difference between the probe clock value and the initial clock value.

See Also

[meiProbeConfigGet](#) | [meiProbeConfigSet](#) | [MPIProbeConfig](#)

mpiProbeCreate

Declaration `mpiProbeCreate(MPIProbe control,
MPIProbeParams *params)`

Required Header `stdmei.h`

Description **ProbeCreate** creates a Probe object identified by the params, which is associated with a control object.

ProbeCreate is the equivalent of a C++ constructor.

The probe params structure specifies the probe type, a probe object identification number, and an index to a hardware probe engine. See [MPIProbeParams](#) for details.

control	a handle to a Probe object.
*params	a pointer to a probe parameter's structure.

Return Values

handle	handle to a Probe object.
MPIHandleVOID	if the object could not be created.

See Also [mpiProbeDelete](#) | [mpiProbeValidate](#) | [MPIProbeParams](#)

MEIProbeInfo

MEIProbeInfo

```
typedef struct MEIProbeInfo {  
    MEIProbeAddress    address;  
} MEIProbeInfo;
```

Description

The **ProbeInfo** structure contains read only information about the Probe.

address	a structure containing controller addresses for useful data during probing.

See Also

[meiProbeInfo](#) | [meiProbeStatus](#)

MPIProbeMessage / MEIProbeMessage

MPIProbeMessage

```
typedef enum {
    MPIProbeMessageNODE_INVALID,
    MPIProbeMessagePROBE_TYPE_INVALID,
    MPIProbeMessagePROBE_INVALID,
} MPIProbeMessage;
```

Description

MPISequenceMessageNODE_INVALID

The SynqNet node number is not available on the network. This message code is returned by MPI methods that fail a service command transaction due to the specified node number being greater than or equal to the total number of nodes discovered during network initialization. To correct this problem, check the discovered node count with [meiSynqNetInfo\(...\)](#). If the node count is not what you expected, check your network wiring, node condition, and re-initialize the network with [mpiControlReset\(...\)](#).

MPISequenceMessagePROBE_TYPE_INVALID

The Probe data type is not valid. This message is returned by [mpiProbeConfigSet\(...\)](#) if the specified Probe data type is not one of the enumerated values.

MPISequenceMessagePROBE_INVALID

The Probe is not valid. This message is returned by [mpiProbeConfigGet\(...\)](#) or [mpiProbeConfigSet\(...\)](#) if the specified Probe params or configurations are out of range. To correct this problem, check your Probe params and config structures.

See Also

MEIProbeMessage

```
typedef enum {
    MEIProbeMessageFIRST = MPIProbeMessageLAST,
    MEIProbeMessageLAST
} MEIProbeMessage;
```

Description

ProbeMessage is mainly a placeholder.

See Also

MPIProbeNumber

MPIProbeNumber

```
typedef union MPIProbeNumber {
    long      motor;
    long      node;
} MPIProbeNumber;
```

Description

The **ProbeNumber** union specifies the identification number for the Probe. The Probe type determines the identification number definition.

motor	An index to identify the motor that is associated with the Probe. Must be specified for MPIProbeTypeMOTOR Probe types.
node	An index to identify the node that is associated with the Probe. Must be specified for MPIProbeTypeIO Probe types.

See Also

[MPIProbeType](#) | [MPIProbeParams](#)

MPIProbeParams

MPIProbeParams

```
typedef struct MPIProbeParams {
    MPIProbeType    type;
    MPIProbeNumber  number;
    long            probeIndex;
} MPIProbeParams;
```

Description

The **ProbeParams** structure specifies the parameters for a Probe engine.

type	An enumerated Probe type. Determines the Probe number definition.
number	An union to identify the object number that is associated with the Probe. The Probe type defines the number.
probeIndex	An index to specify the Probe engine. The hardware may support one or more Probe engines per Motor or Node. The number of Probe engines supported by the hardware can be read with meiMotorInfo(...) and meiSqNodeInfo(...) .

See Also

[mpiProbeParams](#) | [mpiProbeCreate](#) | [MEIMotorInfo](#) | [MEISqNodeInfo](#)

MPIProbeSource

MPIProbeSource

```
typedef enum MPIProbeSource {
    MPIProbeSourceHOME,
    MPIProbeSourceINDEX,
    MPIProbeSourceLIMIT_HW_NEG,
    MPIProbeSourceLIMIT_HW_POS,
    MPIProbeSourceINDEX_SECONDARY,
    MPIProbeSourceMOTOR_IO_0,
    MPIProbeSourceMOTOR_IO_1,
    MPIProbeSourceMOTOR_IO_2,
    MPIProbeSourceMOTOR_IO_3,
    MPIProbeSourceMOTOR_IO_4,
    MPIProbeSourceMOTOR_IO_5,
    MPIProbeSourceMOTOR_IO_6,
    MPIProbeSourceMOTOR_IO_7,
    MPIProbeSourceMOTOR_IO_8,
    MPIProbeSourceMOTOR_IO_9,
    MPIProbeSourceMOTOR_IO_10,
    MPIProbeSourceMOTOR_IO_11,
    MPIProbeSourceMOTOR_IO_12,
    MPIProbeSourceMOTOR_IO_13,
    MPIProbeSourceMOTOR_IO_14,
    MPIProbeSourceMOTOR_IO_15,
} MPIProbeSource;
```

Description **ProbeSource** is an enumeration of input trigger sources for a Probe.

MPIProbeSourceHOME	Home input in the dedicated I/O
MPIProbeSourceINDEX	Index input from the primary encoder in the dedicated I/O
MPIProbeSourceLIMIT_HW_NEG	Hardware Negative Limit input in the dedicated I/O
MPIProbeSourceLIMIT_HW_POS	Hardware Positive Limit input in the dedicated I/O
MPIProbeSourceINDEX_SECONDARY	Index input from the secondary encoder in the dedicated I/O
MPIProbeSourceMOTOR_IO_0	Bit number 0 in the Motor's configurable I/O

MPIProbeStatus

MPIProbeStatus

```
typedef struct MPIProbeStatus {
    long    maxRegisters;
    long    index;
    long    io;
    long    regs[MPIMaxProbeRegisters];
} MPIProbeStatus;
```

Description

The **ProbeStatus** structure specifies the present condition of the Probe engine.

maxRegisters	The maximum number of Probe data registers that are available. This value is determined by the hardware Probe engine capability and the number of data registers that are initialized in the SynqNet packets.
index	The Probe engine index. Depending on the Probe type, the Probe is either associated with a motor or node object. Each Probe can have 1 or more hardware Probe engines. The probe engine is identified by its index.
io	The input state values for the Probe data. Each bit represents a triggered Probe data value in the regs array. If a bit is active, then a corresponding Probe data value was stored in the regs array. For example, if io = 0x3, then regs[0] and regs[1] have valid Probe data.
regs	An array of captured Probe data. The maximum number of data fields is specified by maxRegisters. Each element of the array that contains valid Probe data is flagged by a bit in the I/Ostatus.

See Also

[mpiProbeStatus](#) | [mpiProbeParams](#) | [MPIProbeParams](#)

MPIProbeSourceMOTOR_IO_1	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_2	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_3	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_4	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_5	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_6	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_7	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_8	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_9	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_10	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_11	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_12	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_13	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_14	Bit number 0 in the Motor's configurable I/O
MPIProbeSourceMOTOR_IO_15	Bit number 0 in the Motor's configurable I/O

See Also [MPIProbeConfig](#) | [MEIMotorIoMask](#)

MEIProbeTrace

MEIProbeTrace

```
typedef enum {  
    MEIProbeTraceFIRST = MEITraceLAST << 1,  
    MEIProbeTraceLAST  = MEIProbeTraceFIRST << 15  
} MEIProbeTrace;
```

Description **ProbeTrace** is an enumeration of probe object trace bits to enable debug tracing.

See Also

MEIRecorderTrigger

MEIRecorderTrigger

```
typedef struct MEIRecorderTrigger {
    MEIRecorderTriggerType    type;
    union {
        MEIRecorderTriggerUser user;
    } attributes;
} MEIRecorderTrigger;
```

Description

The **RecorderTrigger** structure specifies the configurations for a data recorder trigger.

type	The data recorder trigger type. See the MEIRecorderTriggerType enumeration.
user	The configurations for a user specified trigger type. See MEIRecorderTriggerUser .

See Also

[MEIRecorderTrigger](#) | [mpiRecorderConfigGet](#) | [mpiRecorderConfigSet](#)

MEIRecorderTriggerIndex

MEIRecorderTriggerIndex

```
typedef enum MEIRecorderTriggerIndex {  
    MEIRecorderTriggerIndexSTART,  
    MEIRecorderTriggerIndexSTOP,  
} MEIRecorderTriggerIndex;
```

Description **RecorderTriggerIndex** is an enumeration of indices to a data recorder's trigger logic.

MEIRecorderTriggerIndexSTART	Index to a data recorder's start trigger.
MEIRecorderTriggerIndexSTOP	Index to a data recorder's stop trigger.

See Also [MEIRecorderConfig](#) | [mpiRecorderConfigGet](#) | [mpiRecorderConfigSet](#)

MPIRecorderMessage

MPIRecorderMessage

```
typedef enum {

    MPIRecorderMessageRECORDER_INVALID,
    MPIRecorderMessageSTARTED,
    MPIRecorderMessageSTOPPED,
    MPIRecorderMessageNOT_CONFIGURED,
    MPIRecorderMessageNO_RECORDERS_AVAIL,
    MPIRecorderMessageNOT_ENABLED,
    MPIRecorderMessageRUNNING,
} MPIRecorderMessage;
```

Description

MPIRecorderMessageRECORDER_INVALID

The recorder object is not valid. This message code is returned by a recorder method if the recorder object handle is not valid. This problem can be caused by a failed [mpiRecorderCreate\(...\)](#). To prevent this problem, check your recorder objects after creation by using [mpiRecorderValidate\(...\)](#).

MPIRecorderMessageSTARTED

The data recorder is already running. This message code is returned by [mpiRecorderStart\(...\)](#) if the data recorder has already been started. If this is a problem, call [mpiRecorderStop\(...\)](#) to stop the data recorder or wait for the recorder to collect the number of specified records and stop.

MPIRecorderMessageSTOPPED

The data recorder is not running. This message code is returned by [mpiRecorderStop\(...\)](#) if the data recorder has already been stopped. If this is a problem, call [mpiRecorderStart\(...\)](#) to start the data recorder.

MPIRecorderMessageNOT_CONFIGURED

The data recorder has not been configured. This message code is returned by [mpiRecorderRecordGet\(...\)](#) if the data address count has not been configured. To correct this problem, configure the data recorder with [mpiRecorderConfigSet\(...\)](#).

MPIRecorderMessageNO_RECORDERS_AVAIL

Returned when a recorder number of -1 is specified and all enabled recorders have been previously reserved by [mpiRecorderCreate\(...\)](#) method calls. Reserved recorders are released by calling [mpiRecorderDelete\(...\)](#), however, it is possible for a fatal error to occur in your application in which case [mpiRecorderDelete\(...\)](#) may not be called. To override a reserved recorder number, explicitly specify the recorder number (i.e. a number other than -1) when calling [mpiRecorderCreate\(...\)](#).

MPIRecorderMessageNOT_ENABLED

An attempt was made to create a recorder that is not enabled on the controller. Recorder objects can be enabled on the controller by calling [mpiControlConfigSet\(...\)](#).

MPIRecorderMessageRUNNING

An attempt was made to call [mpiRecorderConfigSet\(...\)](#) while the recorder was running.

See Also [mpiRecorderCreate](#) | [mpiRecorderValidate](#)

MPIRecorderRecordPoint

MPIRecorderRecordPoint

```
typedef long MPIRecorderRecordPoint;
```

Description

MPIRecorderRecordPoint	represents one recorder record. This will correspond to the value of one XMP address.

See Also

MEIRecorderTrace

MEIRecorderTrace

```
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

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 your support engineer. We recommend that you do **NOT** implement complex Sequences on your own.

Commands are implemented using [MPICommand](#) objects. Information about the different types of commands can be found on [MPICommandType](#) and [MPICommandParams](#). Sample applications for using sequencers can be found in the Sample Applications section. Search for application names starting with **seq**. **Seqkill.c** is a good place to start.

Methods

Create, Delete, Validate Methods

mpiSequenceCreate	Create Sequence object
mpiSequenceDelete	Delete Sequence object
mpiSequenceValidate	Validate Sequence object

Configuration and Information Methods

mpiSequenceConfigGet	Get sequence config
mpiSequenceConfigSet	Set sequence config
mpiSequenceFlashConfigGet	Get sequence flash config
mpiSequenceFlashConfigSet	Set sequence flash config
mpiSequencePageSize	Set pageSize to number of command slots used by sequence
mpiSequenceStatus	Return sequence status

Event Methods

mpiSequenceEventNotifyGet	Select an event mask for host notification of events
mpiSequenceEventNotifySet	Enable host notification of sequence events
mpiSequenceEventReset	Reset sequence events

Action Methods

[mpiSequenceCompile](#)

[mpiSequenceLoad](#)

[mpiSequenceResume](#)

[mpiSequenceStart](#)

[mpiSequenceStep](#)

[mpiSequenceStop](#)

Load sequence commands into firmware

Resume execution of sequence

Start execution of sequence

Execute count steps of a stopped sequence

Stop sequence

Memory Methods

[mpiSequenceMemory](#)

[mpiSequenceMemoryGet](#)

[mpiSequenceMemorySet](#)

Set address used to access sequence memory

Get bytes of sequence memory and put into application memory

Put (set) bytes of application memory into sequence memory

Relational Methods

[mpiSequenceControl](#)

[mpiSequenceNumber](#)

Get handle to Control

Get index number of sequence

List Methods for Event Sources

[mpiSequenceCommand](#)

[mpiSequenceCommandAppend](#)

[mpiSequenceCommandCount](#)

[mpiSequenceCommandFirst](#)

[mpiSequenceCommandIndex](#)

[mpiSequenceCommandInsert](#)

[mpiSequenceCommandLast](#)

[mpiSequenceCommandListGet](#)

[mpiSequenceCommandListSet](#)

[mpiSequenceCommandNext](#)

[mpiSequenceCommandPrevious](#)

[mpiSequenceCommandRemove](#)

Return handle to indexed command of sequence

Append command to sequence

Count the number of commands in sequence

Return handle to first command in sequence

Return the index of a command in sequence

Insert command into sequence

Return handle of last command in sequence

Get list of commands in sequence

Set list of commands in sequence

Get handle to next command in list

Get handle to previous command in list

Remove command from list

Data Types

[MPISequenceConfig](#) / [MEISequenceConfig](#)

[MPISequenceMessage](#)

[MPISequenceState](#)

[MPISequenceStatus](#)

[MEISequenceTrace](#)

mpiSequenceCreate

Declaration `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	<i>SequenceCreate</i> 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	<i>SequenceCreate</i> 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 <i>SequenceDelete</i> 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

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 <i>SequenceConfigGet</i> 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

Description	SequenceConfigSet sets the configuration of a Sequence (<i>sequence</i>) using data from the structure pointed to by <i>config</i> , and also using data from the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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 <i>SequenceConfigSet</i> 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 <i>SequenceFlashConfigGet</i> successfully writes the Sequence’s flash configuration to the structure(s)

See Also [mpiSequenceFlashConfigSet](#)

mpiSequenceFlashConfigSet

[illegible]

Required Header

Description	SequenceFlashConfigSet sets a Sequence's (<i>sequence</i>) flash configuration using data from the structure pointed to by <i>config</i> , and also using data from the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> 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 <i>SequenceFlashConfigSet</i> successfully sets the Sequence's flash configuration using data from the structure(s)

See Also [MEISequenceConfig](#) | [mpiSequenceFlashConfigGet](#)

mpiSequencePageSize

```
long mpiSequencePageSize(MPISequence sequence,  
                          long *pageSize)
```

Required Header

Description	SequencePageSize writes the <i>number</i> of command slots that are available to a Sequence (<i>sequence</i> , on its associated motion controller) to the contents of <i>pageSize</i> .

Return Values

MPIMessageOK	if <i>SequencePageSize</i> successfully writes the number of command slots (available to the Sequence) to the contents of <i>pageSize</i>

See Also

mpiSequenceStatus

Declaration

```
long mpiSequenceStatus(MPISequence sequence,  
                       MPISequenceStatus *status,  
                       void *external)
```

Required Header

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).

sequence	a handle to a Sequence object
*status	a pointer to Sequence's status structure
*external	a pointer to an implementation-specific structure

XMP Only

external should always be set to NULL.

Return Values

MPIMessageOK	if <i>SequenceStatus</i> successfully returns the Sequence's status and writes the status to the structure(s)
MPIMessageARG_INVALID	if the <i>status</i> pointer is NULL.

See Also [MPISequenceStatus](#)

Required Header

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).

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

Return Values

MPIMessageOK	if <i>SequenceEventNotifyGet</i> 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 <i>SequenceEventNotifySet</i> successfully requests host notification of the events in the event mask(s)

See Also

[MPIEventMaskEXTERNAL](#) | [MEIEventMask](#) | [mpiSequenceEventNotifyGet](#)

mpiSequenceEventReset

[illegible]

Required Header

Description	SequenceEventReset resets the event(s) that are specified in <i>eventMask</i> and generated by a Sequence (<i>sequence</i>). Your application should not call SequenceEventReset <i>until</i> one or more latchable events have occurred.

Return Values

MPIMessageOK	if <i>SequenceEventReset</i> successfully resets the event(s) that are specified in <i>eventMask</i> 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 <i>SequenceCompile</i> 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 mpiEventManagerService(...) (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
<i>command</i> is MPIHandleVOID	<i>SequenceLoad</i> loads Commands starting with the first Command of the Sequence
<i>start</i> is not FALSE	<i>SequenceLoad</i> starts the sequence after the commands are loaded

Return Values	
MPIMessageOK	if <i>SequenceLoad</i> successfully loads the firmware command slots of a Sequence

See Also

[mpiSequenceStart](#) | [mpiEventManagerService](#) | [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 <i>SequenceResume</i> successfully resumes a Sequence from the point where the Sequence has stopped

See Also

mpiSequenceStart

Declaration

```
long mpiSequenceStart(MPISequence sequence,  
                     MPICommand command)
```

Required Header

Description	SequenceStart begins the execution of a Sequence (<i>sequence</i>), starting with the Command (<i>command</i>). If <i>command</i> is MPIHandleVOID, execution starts with the first command of the Sequence.

Return Values

MPIMessageOK	if <i>SequenceStart</i> successfully begins the execution of a Command Sequence

See Also [mpiSequenceStop](#)

mpiSequenceStep

```
long mpiSequenceStep(MPISequence sequence,
                    long count)
```

Required Header

Description	SequenceStep executes <i>count</i> steps (Commands) of a stopped Sequence (<i>sequence</i>). After executing the Commands, the Sequence will be in the MPISequenceStateSTOPPED state.

Return Values

MPIMessageOK	if <i>SequenceStep</i> successfully executes <i>count</i> 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 <i>SequenceStop</i> successfully stops a Sequence (while it is executing)

See Also [mpiSequenceStart](#)

mpiSequenceMemory

```
long mpiSequenceMemory(MPISequence sequence,  
void **memory)
```

Required Header

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 <i>SequenceMemory</i> 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

Description	SequenceMemoryGet copies <i>count</i> bytes of a Sequence's (<i>sequence</i>) memory (starting at address <i>src</i>) to application memory (starting at address <i>dst</i>).

Return Values

MPIMessageOK	if <i>SequenceMemoryGet</i> 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

Description	<i>SequenceMemorySet</i> copies count bytes of application memory (starting at address src) to a Sequence's (sequence) memory (starting at address dst).

Return Values

MPIMessageOK	if <i>SequenceMemorySet</i> successfully copies <i>count</i> bytes of application memory to a Sequence object's memory

See Also [mpiSequenceMemory](#) | [mpiSequenceMemoryGet](#)

mpiSequenceControl

Declaration [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 <i>sequence</i> is invalid

See Also [mpiSequenceCreate](#) | [mpiControlCreate](#)

mpiSequenceNumber

```
long mpiSequenceNumber(MPISequence sequence,  
                      long *number)
```

Required Header

Description	SequenceNumber writes the index of a Sequence (<i>sequence</i> , on the motion controller that the Sequence object is associated with) to the contents of <i>number</i> .

Return Values

MPIMessageOK	if <i>SequenceNumber</i> successfully writes the Sequence's index to the contents of <i>number</i>

See Also

mpiSequenceCommand

Declaration `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 <i>index</i> th Command of a Sequence (<i>sequence</i>)
MPIHandleVOID	if <i>sequence</i> is invalid if <i>index</i> is less than 0 if <i>index</i> is greater than or equal to <code>mpiSequenceCount(sequence)</code>
MPIMessageARG_INVALID	if <i>index</i> is a negative number.
MEIListMessageELEMENT_NOT_FOUND	if <i>index</i> is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	if <i>sequence</i> is an invalid handle.

See Also

mpiSequenceCommandAppend

Declaration

```
long mpiSequenceCommandAppend(MPISequence sequence,  
                              MPICommand  command)
```

Required Header

Description	SequenceCommandAppend appends a Command (<i>command</i>) to a Sequence (<i>sequence</i>).

sequence	a handle to the Sequence object.
command	a handle to a Command object.

Return Values

MPIMessageOK	if <i>SequenceCommandAppend</i> successfully appends a Command to a Sequence
MPIMessageHANDLE_INVALID	Either <i>sequence</i> or <i>command</i> 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 (<i>sequence</i>)
-1	if <i>sequence</i> is invalid
0	if <i>sequence</i> is empty

See Also

mpiSequenceCommandFirst

Declaration [MPICommand](#) **mpiSequenceCommandFirst**([MPISequence](#) **sequence**)

Required Header stdmpi.h

Description **SequenceCommandFirst** returns the first element in the list. This function can be used in conjunction 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 (<i>sequence</i>)
MPIHandleVOID	if <i>sequence</i> is invalid if <i>sequence</i> is empty
MPIMessageHANDLE_INVALID	if <i>sequence</i> is an invalid handle.

See Also [mpiSequenceCommandNext](#) | [mpiSequenceCommandLast](#)

Required Header

sequence	a handle to the Sequence object.
command	a handle to a Command object.

index	of a Command (<i>command</i>) in a Sequence (<i>sequence</i>)
-1	if <i>sequence</i> is invalid if the Command (<i>command</i>) was not found in the Sequence (<i>sequence</i>)

file:///D:/pdfs/030100/html/Software-MPI/docs/Sequence/Method/cmdinx1.htm [7/23/2004 11:27:26 AM]

mpiSequenceCommandInsert

Declaration

```
long mpiSequenceCommandInsert( MPISequence sequence,  
                               MPICommand command,  
                               MPICommand insert )
```

Required Header

Method	Description
SequenceCommandInsert	inserts a Command (<i>insert</i>) in a Sequence (<i>sequence</i>) just after the specified Command (<i>command</i>).

Return Values

MPIMessageOK	if <i>SequenceCommandInsert</i> successfully inserts the Command (<i>insert</i>) in a Sequence following the specified Command (<i>command</i>)

See Also

mpiSequenceCommandLast

Declaration [MPICommand](#) **mpiSequenceCommandLast**([MPISequence](#) **sequence**)

Required Header `stdmpi.h`

Description [SequenceCommandLast](#) returns the last element in the list. This function can be used in conjunction 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 (<i>sequence</i>)
MPIHandleVOID	if <i>sequence</i> is invalid if <i>sequence</i> is empty
MPIMessageHANDLE_INVALID	if <i>sequence</i> is an invalid handle.

See Also [mpiSequenceCommandFirst](#) | [mpiSequenceCommandPrevious](#) | [mpiSequenceCommandNext](#)

mpiSequenceCommandListGet

[illegible]

Required Header

```
stdmpi.h
```

Description	SequenceCommandListGet gets the Commands in a Sequence (<i>sequence</i>). <i>SequenceCommandListGet</i> writes the number of Commands [in a Sequence (<i>sequence</i>)] 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 <i>SequenceCommandListGet</i> successfully gets the list of Commands in a Sequence

See Also [mpiSequenceCommandListSet](#)

mpiSequenceCommandListSet

Declaration	long mpiSequenceCommandListSet (MPISequence	sequence,
	long	commandCount,
	MPICommand	*commandList)

Required Header

Description	SequenceCommandListSet creates a Sequence (<i>sequence</i>) of <i>commandCount</i> Commands using the Command handles specified by <i>commandList</i> . 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 <i>SequenceCommandListGet</i> successfully creates a Sequence of Commands using the Command handles specified by <i>commandList</i>

See Also [mpiSequenceCommandListGet](#)

mpiSequenceCommandNext

[illegible]

Required Header

Description	SequenceCommandNext returns the next element following "command" on the list. This function can be used in conjunction with <code>mpiSequenceCommandFirst()</code> 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 (<i>command</i>) in a Sequence (<i>sequence</i>)
MPIHandleVOID	if <i>sequence</i> is invalid if <i>command</i> is the last command in a Sequence (<i>sequence</i>)
MPIMessageHANDLE_INVALID	Either <i>sequence</i> or <i>command</i> is an invalid handle.

See Also [mpiSequenceCommandFirst](#) | [mpiSequenceCommandPrevious](#)

mpiSequenceCommandPrevious

Declaration [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 conjunction 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

[illegible]

Required Header

Description	SequenceCommandRemove removes a Command (<i>command</i>) from a Sequence (<i>sequence</i>).

Return Values

MPIMessageOK	if <i>SequenceCommandRemove</i> 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

The sequence number is out of range. This message code is returned by [mpiSequenceCreate\(...\)](#) if the sequence number is less than zero or greater than or equal to MEIXmpMAX_PSs. This message code is also returned if the specified sequence number is not active in the controller. To correct this problem, use [mpiControlConfigSet\(...\)](#) to enable the sequence object, by setting the sequenceCount to greater than the sequence number. For example, to enable sequence 0 to 3, set sequenceCount to 4. This message code is returned by [mpiSequenceLoad\(...\)](#) if the sequence buffer size and the sequence page size are not equal. This indicates an internal MPI Library problem.

MPISequenceMessageCOMMAND_COUNT

The sequence command count is out of range. This message code is returned by [mpiSequenceStart\(...\)](#) or [meiSequenceCompile\(...\)](#) if the sequence command count is less than or equal to zero. To correct this problem, set the command count to a value greater than zero.

MPISequenceMessageCOMMAND_NOT_FOUND

The sequence command is not found. This message code is returned by [mpiSequenceLoad\(...\)](#), [mpiSequenceStart\(...\)](#), or [meiSequenceCompile\(...\)](#) if the specified command is not a member of the sequence. To correct this problem, specify a command that is a member of the sequence.

MPISequenceMessageSTARTED

The program sequencer is already running. This message code is returned by [mpiSequenceResume\(...\)](#), [mpiSequenceStart\(...\)](#), or [mpiSequenceStep\(...\)](#) if the program sequencer has already been started. If this is a problem, call [mpiSequenceStop\(...\)](#) to stop the program sequencer or monitor the sequence status and wait for the state to equal STOPPED.

MPISequenceMessageSTOPPED

The program sequencer is not running. This message code is returned by [mpiSequenceStop\(...\)](#) if the program sequencer has already been stopped. If this is a problem, call [mpiSequenceStart\(...\)](#) to start the program sequencer.

See Also

MPISequenceState

MPISequenceState

```
typedef enum {
    MPISequenceStateSTOPPED = 0,
    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 **MEISequenceTrace** sets tracing on for the mpiSequenceLoad() method.

See Also [MPISequence](#) | [MEITrace](#) | [mpiSequenceLoad](#)

Recorder Objects

Introduction

A **Recorder** object provides a mechanism to collect and buffer any data in the controller's memory. After a recorder is configured and started, the controller copies the data from the specified addresses to a local buffer every "N" samples. Later, the host can collect the data by polling or via interrupt-based events.

The controller supports up to 32 data recorders, which can collect data from up to a total of 32 addresses. The buffers can be dynamically allocated. A larger data recorder buffer may be required for higher sample rates, slow host computers, when running via client/server, or when a large number of data fields are being recorded.

A recorder can be started or stopped from the host application or from the controller by configuring a data recorder trigger. When the trigger conditions are met, the controller will automatically start or stop a data recorder. This is very useful for logging relevant variables during the period preceding a fault or error. Normally, the recorder stops collecting data when the buffer is full. It can also be configured to continuously collect data, overwriting the previous data until it is commanded to stop. This is useful for trapping a recent history of controller data.

When using data recorders, make sure to enable enough recorder objects and buffer memory with `mpiControlConfigSet(...)`. Then, configure the recorders with `mpiRecorderRecordConfig(...)` or `mpiRecorderConfigSet(...)`, and start recording with `mpiRecorderStart(...)`. Data can then be collected with `mpiRecorderRecordGet(...)`.

| [Buffer Size](#) |

Methods

Create, Delete, Validate Methods

<u>mpiRecorderCreate</u>	Create Recorder object
<u>mpiRecorderDelete</u>	Delete Recorder object
<u>mpiRecorderValidate</u>	Validate Recorder object

Configuration and Information Methods

<u>mpiRecorderConfigGet</u>	Get Recorder's configuration
<u>mpiRecorderConfigSet</u>	Set Recorder's configuration
<u>mpiRecorderRecordConfig</u>	Configure type of data record that Recorder will capture

[mpiRecorderStatus](#)

Get status of Recorder

Event Methods

[mpiRecorderEventNotifyGet](#)

Get event mask of events for which host notification has been requested

[mpiRecorderEventNotifySet](#)

Set event mask of events for which host notification will be requested

[mpiRecorderEventReset](#)

Reset the events specified in event mask that are generated by Recorder

Action Methods

[mpiRecorderRecordGet](#)

Get data records from Recorder

[mpiRecorderStart](#)

Start recording data records using Recorder

[mpiRecorderStop](#)

Stop recording data records using Recorder

Memory Methods

[mpiRecorderMemory](#)

Get address to Recorder's memory

[mpiRecorderMemoryGet](#)

Copy data from Recorder memory to application memory

[mpiRecorderMemorySet](#)

Copy data from application memory to Recorder memory

Relational Methods

[mpiRecorderControl](#)

Return handle of Control object associated with Recorder

Data Types

[MPIRecorderConfig](#) / [MEIRecorderConfig](#)[MPIRecorderMessage](#)[MPIRecorderRecord](#) / [MEIRecorderRecord](#)[MEIRecorderRecordAxis](#)[MEIRecorderRecordFilter](#)[MPIRecorderRecordPoint](#)[MPIRecorderRecordType](#) / [MEIRecorderRecordType](#)[MPIRecorderStatus](#)[MEIRecorderTrace](#)[MEIRecorderTrigger](#)[MEIRecorderTriggerCondition](#)[MEIRecorderTriggerIndex](#)[MEIRecorderTriggerType](#)[MEIRecorderTriggerUser](#)

Constants

[MPIRecorderADDRESS_COUNT_MAX](#)

[MEIRecorderMAX_AXIS_RECORDS](#)

[MEIRecorderMAX_FILTER_RECORDS](#)

mpiRecorderCreate

Declaration `MPIRecorder mpiRecorderCreate(MPIControl control,
long number);`

Required Header `stdmpi.h`

Description [RecorderCreate](#) creates a Recorder object identified by **number**, which is associated with a control object.

RecorderCreate is the equivalent of a C++ constructor.

The recorder number specifies which recorder to create. The valid range for the **number** parameter is 0 to the controller's recordCount -1. See [MPIControlConfig{...}](#) for details. If the recorder is not enabled or is already in use (another process has called [mpiRecorderCreate\(...\)](#) with the same number parameter), [mpiRecorderCreate\(...\)](#) will return an invalid handle causing subsequent [mpiRecorderValidate\(...\)](#) calls to fail.

control	a handle to a Control object.
number	An index to the controller's data recorder. If (-1) is specified, the next available recorder object handle will be returned. The valid range is from -1 (next available recorder) to the controller's recordCount - 1. When using (-1), make sure to delete the recorder object to free it for other applications. If the recorder object is not freed, it will not be accessible to another application until the controller is reset.

Return Values	
handle	to a Recorder object
MPIHandleVOID	if the Recorder object could not be created

See Also [mpiRecorderDelete](#) | [mpiRecorderValidate](#) | [MPIControlConfig](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

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 *RecorderDelete* 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. RecorderValidate should be called immediately after an object is created.

recorder	a handle to a Recorder object

Return Values	
MPIMessageOK	if Recorder is a handle to a valid object.
MPIRecorderMessageNOT_ENABLED	if the specified recorder number has not been enabled in the controller.
MPIRecorderMessageNO_RECORDERS_AVAIL	if the specified recorder number is (-1) and there are no more recorders available on the controller.

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

[MPIRecorderConfig](#) | [mpiRecorderConfigSet](#)

Required Header

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).

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

MPIMessageOK

See Also [MEIRecorderConfig](#) | [mpiRecorderConfigGet](#)

mpiRecorderRecordConfig

Declaration

long [mpiRecorderRecordConfig](#)([MPIRecorder](#)
[MPIRecorderRecordType](#)
long
void

recorder,
type,
count,
*handle)

Required Header

stdmpi.h

Description

[RecorderRecordConfig](#) configures the type (*type*) of record that a Recorder (*recorder*) will capture.

If "type" is	Then
MPIRecorderRecordTypePOINT	<i>count</i> data points will be recorded, and <i>handle</i> points to an array of <i>count</i> controller addresses
MEIRecorderRecordTypeAXIS	<i>count</i> records of type MPIRecorderRecordAxis{ } will be recorded, and <i>handle</i> points to an array of <i>count</i> Axis handles
MEIRecorderRecordTypeFILTER	<i>count</i> records of type MPIRecorderRecordFilter{ } will be recorded, and <i>handle</i> points to an array of <i>count</i> Filter handles

Return Values	
MPIMessageOK	if <i>RecorderRecordConfig</i> successfully configures the type of record that the Recorder will capture

See Also

[MPIRecorderRecordAxis](#) | [MPIRecorderRecordFilter](#)

Required Header

XMP Only

Return Values

See Also [MPIRecorderStatus](#)

Required Header

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).

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 <i>RecorderEventNotifyGet</i> successfully writes the event mask to the structure(s)

See Also [MEIEventNotifyData](#) | [MEIEventStatusInfo](#) | [mpiRecorderEventNotifySet](#)

Required Header

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).

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

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

See Also [MEIEventMaskRECORDER](#) | [MEIEventNotifyData](#) | [MEIEventStatusInfo](#)
[mpiRecorderEventNotifyGet](#)

mpiRecorderEventReset

[illegible]

Required Header

Description	RecorderEventReset resets the event(s) specified in <i>eventMask</i> and generated by a Recorder (<i>recorder</i>). Your application should call <i>RecorderEventReset</i> only after one or more latchable events have occurred.

Return Values

MPIMessageOK	if <i>RecorderEventReset</i> successfully resets the event(s) that are specified in <i>eventMask</i> and generated by a Recorder

See Also

Required Header

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 <i>RecorderRecordGet</i> successfully gets the data records

See Also [mpiRecorderRecordConfig](#)

mpiRecorderStart

Declaration

```
long mpiRecorderStart(MPIRecorder recorder,
                      long count); /* -1 => continuous,
                                   >0 => # of records */
```

Required Header stdmpi.h

Description **RecorderStart** commands the controller to begin recording data records. Before starting a recorder, it must be configured with `mpiRecorderRecordConfig(...)` or `mpiRecordConfigGet/Set(...)`.

recorder	a handle to a Recorder object
count	The number of data records to record. If (-1) is specified, the data recorder will continuously record until the buffer is full. If the host is retrieving data from the buffer faster than the controller can fill the buffer, the controller will continuously copy data to the buffer. The valid range is from -1 (continuous recording) to the maximum number of records available in the data recorder buffer.

Return Values

MPIMessageOK	if the data recorder successfully begins to record data.
MPIRecorderMessageSTARTED	if the data recorder is already running.

See Also [mpiRecorderRecordConfig](#) | [mpiRecorderStop](#) | [mpiRecorderConfigGet](#) | [mpiRecorderConfigSet](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

mpiRecorderStop

Declaration long [mpiRecorderStop](#)([MPIRecorder](#) recorder)

Required Header stdmpi.h

Description [RecorderStop](#) instructs a Recorder (*recorder*) to stop recording data records.

Sample Code

```
/*
    Look for the warning code when the recorder is already stopped.
    This is usually not considered a bad thing (error).
*/
returnValue = mpiRecorderStop(recorder);
if(returnValue == MPIRecorderMessageSTOPPED)
{
    returnValue = MPIMessageOK;
}
msgCHECK(returnValue);
```

Return Values	
MPIMessageOK	if <i>RecorderStop</i> successfully stops recording data records
MPIRecorderMessageSTOPPED	This means the the recorder was already stopped when <i>mpiRecorderStop</i> was called. This is a warning, not an error. This can be ignored if the user does not have some reason for why the recorder must be running at this point.

See Also [mpiRecorderStart](#)

Required Header

Return Values

if *RecorderMemory* successfully writes the Recorder's memory address to the contents of *memory*

file:///D:/pdfs/030100/html/Software-MPI/docs/Recorder/Method/mem1.htm [7/27/2004 2:00:15 PM]

mpiRecorderMemoryGet

```

Declaration
long mpiRecorderMemoryGet(MPIRecorder recorder,
void *dst,
void *src,
long count)

```

Required Header

Description	RecorderMemoryGet copies <i>count</i> bytes of a Recorder's (<i>recorder</i>) memory (starting at address <i>src</i>) to application memory (starting at address <i>dst</i>).

Return Values

MPIMessageOK	if <i>RecorderMemoryGet</i> 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

Description	RecorderMemorySet copies <i>count</i> bytes of application memory (starting at address <i>src</i>) to a Recorder's (<i>recorder</i>) memory (starting at address <i>dst</i>).

Return Values

MPIMessageOK

if *RecorderMemorySet* successfully copies data from application memory to Recorder memory

See Also [mpiRecorderMemory](#) | [mpiRecorderMemoryGet](#)

mpiRecorderControl

Declaration [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        highCount;        /* >0 => record count to trigger high buffer */
    long        bufferWrap;       /* TRUE/FALSE */

    long        addressCount;     /* number of data point addresses in address[] */
    void        *address[MPIRecorderADDRESS_COUNT_MAX];
} MPIRecorderConfig;
```

Description

RecorderConfig structure specifies the configurations for a data recorder. It configures the sampling period, the buffer high event level, whether the buffering should wrap around, and a list of controller addresses to record.

period	The number of controller samples between successive data recorder acquisitions. A value of zero or one means the data recorder will acquire data every sample. A value of 2 means every other sample, 3 means every 3rd sample, etc. The valid range is 0 to 32767.
highCount	The number of buffered records until a MPIEventTypeRECORDER_HIGH status/event is generated. The valid range is 1 to the recorder buffer size configured by mpiControlConfigSet(...) .
bufferWrap	Data recorder buffer rollover. A value of TRUE enables the buffer rollover, FALSE (default) disables the buffer rollover. When the bufferWrap is disabled, the controller will stop collecting data when the buffer is full. When bufferWrap is enabled, the controller will continuously collect data after the buffer is full, overwriting any previously collected data. The bufferWrap should be enabled if your application only wants to retrieve the last buffer of data after the data recorder is stopped. Most applications should set the bufferWrap to FALSE.
addressCount	The number of controller addresses in the address array.
*address	An array of controller memory addresses to be recorded.

MEIRecorderConfig

```
typedef struct MEIRecorderConfig {
    MEIRecorderTrigger trigger[MPIRecorderTriggerIndexLAST];
} MEIRecorderConfig;
```

Description

The **RecorderConfig** structure specifies the configurations for the controller's data recorder triggers.

A data recorder can be started or stopped from the host application with `mpiRecorderStart/Stop(...)` or from the controller by configuring a data recorder trigger. When the trigger conditions are met, the controller will automatically start or stop a data recorder.

trigger	An array of data recorder trigger configuration structures.

See Also

[mpiRecorderConfigGet](#) | [mpiRecorderConfigSet](#) | [mpiRecorderStart](#) | [mpiRecorderStop](#)

MPIRecorderMessage

MPIRecorderMessage

```
typedef enum {

    MPIRecorderMessageRECORDER_INVALID,
    MPIRecorderMessageSTARTED,
    MPIRecorderMessageSTOPPED,
    MPIRecorderMessageNOT_CONFIGURED,
    MPIRecorderMessageNO_RECORDERS_AVAIL,
    MPIRecorderMessageNOT_ENABLED,
    MPIRecorderMessageRUNNING,
} MPIRecorderMessage;
```

Description

MPIRecorderMessageRECORDER_INVALID

The recorder object is not valid. This message code is returned by a recorder method if the recorder object handle is not valid. This problem can be caused by a failed [mpiRecorderCreate\(...\)](#). To prevent this problem, check your recorder objects after creation by using [mpiRecorderValidate\(...\)](#).

MPIRecorderMessageSTARTED

The data recorder is already running. This message code is returned by [mpiRecorderStart\(...\)](#) if the data recorder has already been started. If this is a problem, call [mpiRecorderStop\(...\)](#) to stop the data recorder or wait for the recorder to collect the number of specified records and stop.

MPIRecorderMessageSTOPPED

The data recorder is not running. This message code is returned by [mpiRecorderStop\(...\)](#) if the data recorder has already been stopped. If this is a problem, call [mpiRecorderStart\(...\)](#) to start the data recorder.

MPIRecorderMessageNOT_CONFIGURED

The data recorder has not been configured. This message code is returned by [mpiRecorderRecordGet\(...\)](#) if the data address count has not been configured. To correct this problem, configure the data recorder with [mpiRecorderConfigSet\(...\)](#).

MPIRecorderMessageNO_RECORDERS_AVAIL

Returned when a recorder number of -1 is specified and all enabled recorders have been previously reserved by [mpiRecorderCreate\(...\)](#) method calls. Reserved recorders are released by calling [mpiRecorderDelete\(...\)](#), however, it is possible for a fatal error to occur in your application in which case [mpiRecorderDelete\(...\)](#) may not be called. To override a reserved recorder number, explicitly specify the recorder number (i.e. a number other than -1) when calling [mpiRecorderCreate\(...\)](#).

MPIRecorderMessageNOT_ENABLED

An attempt was made to create a recorder that is not enabled on the controller. Recorder objects can be enabled on the controller by calling [mpiControlConfigSet\(...\)](#).

MPIRecorderMessageRUNNING

An attempt was made to call [mpiRecorderConfigSet\(...\)](#) while the recorder was running.

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 [MPIRecorderConfig](#)

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 a non-zero value (-1), otherwise enabled will equal 0.
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](#)

MEIRecorderTrace

MEIRecorderTrace

```
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

MEIRecorderTrigger

MEIRecorderTrigger

```
typedef struct MEIRecorderTrigger {
    MEIRecorderTriggerType    type;
    union {
        MEIRecorderTriggerUser user;
    } attributes;
} MEIRecorderTrigger;
```

Description

The **RecorderTrigger** structure specifies the configurations for a data recorder trigger.

type	The data recorder trigger type. See the MEIRecorderTriggerType enumeration.
user	The configurations for a user specified trigger type. See MEIRecorderTriggerUser .

See Also

[MEIRecorderTrigger](#) | [mpiRecorderConfigGet](#) | [mpiRecorderConfigSet](#)

MEIRecorderTriggerCondition

MEIRecorderTriggerCondition

```
typedef enum MEIRecorderTriggerCondition {  
    MEIRecorderTriggerConditionMATCH,  
    MEIRecorderTriggerConditionCHANGE,  
} MEIRecorderTriggerCondition;
```

Description

RecorderTriggerCondition is an enumeration of a data recorder's trigger conditions. The mask and pattern fields referred to are from the [MEIRecorderTriggerUser](#) structure.

MEIRecorderTriggerTriggerMATCH	Triggers when the value at the specified address ANDed with the mask is equal to the specified pattern .
MEIRecorderTriggerTriggerCHANGE	Triggers when the value at the specified address ANDed with the mask changes. The pattern field is only used to set the initial bit pattern used to determine if a change occurs.

See Also

[MEIRecorderTriggerUser](#) | [mpiRecorderConfigGet](#) | [mpiRecorderConfigSet](#)

MEIRecorderTriggerIndex

MEIRecorderTriggerIndex

```
typedef enum MEIRecorderTriggerIndex {  
    MEIRecorderTriggerIndexSTART,  
    MEIRecorderTriggerIndexSTOP,  
} MEIRecorderTriggerIndex;
```

Description **RecorderTriggerIndex** is an enumeration of indices to a data recorder's trigger logic.

MEIRecorderTriggerIndexSTART	Index to a data recorder's start trigger.
MEIRecorderTriggerIndexSTOP	Index to a data recorder's stop trigger.

See Also [MEIRecorderConfig](#) | [mpiRecorderConfigGet](#) | [mpiRecorderConfigSet](#)

MEIRecorderTriggerType

MEIRecorderTriggerType

```
typedef enum MEIRecorderTriggerType {  
    MEIRecorderTriggerTypeDISABLED,  
    MEIRecorderTriggerTypeUSER,  
} MEIRecorderTriggerType;
```

Description **RecorderTriggerType** is an enumeration of a data recorder's trigger logic types.

MEIRecorderTriggerTypeDISABLED	The data recorder trigger is not enabled.
MEIRecorderTriggerTypeUSER	The data recorder trigger is user configurable. See the MEIRecorderTriggerUser{...} structure for details.

See Also [MEIRecorderTrigger](#) | [MEIRecorderTriggerUser](#) | [mpiRecorderConfigGet](#) | [mpiRecorderConfigSet](#)

MEIRecorderTriggerUser

MEIRecorderTriggerUser

```
typedef struct  MEIRecorderTriggerUser {
    MEIRecorderTriggerCondition    condition;
    long                        *addr;
    unsigned long               mask;
    unsigned long               pattern;
    unsigned long               count;
} MEIRecorderTriggerUser;
```

Description

The [RecorderTriggerCondition](#) structure specifies the configurations for a user specified data recorder trigger.

condition	The logic that determines how to evaluate the addr, mask, and pattern. See the MEIRecorderTriggerCondition enumeration.
*addr	A pointer to a controller address.
mask	A bit mask ANDed with the value at the controller address.
pattern	A bit pattern compared to the masked value at the controller address.
count	The number of records to collect when the recorder is triggered. This is valid for both start and stop triggers. The valid range is 0 to the recorder buffer size configured by mpiControlConfigSet(...). When used for the start trigger, the valid values range from -1 (continuous recording) to the maximum number of records available in the data recorder buffer. When used for the stop trigger, <i>count</i> records will be collected after the trigger has triggered.

See Also

[MEIRecorderTrigger](#) | [mpiRecorderConfigGet](#) | [mpiRecorderConfigSet](#)

MPIRecorderADDRESS_COUNT_MAX

MPIRecorderADDRESS_COUNT_MAX

```
#define MPIRecorderADDRESS_COUNT_MAX      (32)
```

Description

RecorderADDRESS_COUNT_MAX defines the maximum number of addresses the Recorder object supports.

See Also [MPIRecorderConfig](#)

MEIRecorderMAX_AXIS_RECORDS

MEIRecorderMAX_AXIS_RECORDS

```
#define MEIRecorderMAX_AXIS_RECORDS (8)
```

Description

RecorderMAX_AXIS_RECORDS defines the maximum number of MEIRecorderRecordAxis records that can be recorded by a single recorder at any one time.

See Also [MEIRecorderRecordAxis](#) | [mpiRecorderRecordConfig](#)

MEIRecorderMAX_FILTER_RECORDS

MEIRecorderMAX_FILTER_RECORDS

```
#define MEIRecorderMAX_FILTER_RECORDS (8)
```

Description

RecorderMAX_FILTER_RECORDS defines the maximum number of MEIRecorderRecordFilter records that can be recorded by a single recorder at any one time.

See Also [MEIRecorderRecordFilter](#) | [mpiRecorderRecordConfig](#)

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 Buffers*.

[Return to Recorder Object's page](#)

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 your support engineer. We recommend that you do **NOT** implement complex Sequences on your own.

Commands are implemented using [MPICommand](#) objects. Information about the different types of commands can be found on [MPICommandType](#) and [MPICommandParams](#). Sample applications for using sequencers can be found in the Sample Applications section. Search for application names starting with **seq**. **Seqkill.c** is a good place to start.

Methods

Create, Delete, Validate Methods

mpiSequenceCreate	Create Sequence object
mpiSequenceDelete	Delete Sequence object
mpiSequenceValidate	Validate Sequence object

Configuration and Information Methods

mpiSequenceConfigGet	Get sequence config
mpiSequenceConfigSet	Set sequence config
mpiSequenceFlashConfigGet	Get sequence flash config
mpiSequenceFlashConfigSet	Set sequence flash config
mpiSequencePageSize	Set pageSize to number of command slots used by sequence
mpiSequenceStatus	Return sequence status

Event Methods

mpiSequenceEventNotifyGet	Select an event mask for host notification of events
mpiSequenceEventNotifySet	Enable host notification of sequence events
mpiSequenceEventReset	Reset sequence events

Action Methods

[mpiSequenceCompile](#)

[mpiSequenceLoad](#)

[mpiSequenceResume](#)

[mpiSequenceStart](#)

[mpiSequenceStep](#)

[mpiSequenceStop](#)

Load sequence commands into firmware

Resume execution of sequence

Start execution of sequence

Execute count steps of a stopped sequence

Stop sequence

Memory Methods

[mpiSequenceMemory](#)

[mpiSequenceMemoryGet](#)

[mpiSequenceMemorySet](#)

Set address used to access sequence memory

Get bytes of sequence memory and put into application memory

Put (set) bytes of application memory into sequence memory

Relational Methods

[mpiSequenceControl](#)

[mpiSequenceNumber](#)

List Methods for Event Sources

[mpiSequenceCommand](#)

[mpiSequenceCommandAppend](#)

[mpiSequenceCommandCount](#)

[mpiSequenceCommandFirst](#)

[mpiSequenceCommandIndex](#)

[mpiSequenceCommandInsert](#)

[mpiSequenceCommandLast](#)

[mpiSequenceCommandListGet](#)

[mpiSequenceCommandListSet](#)

[mpiSequenceCommandNext](#)

[mpiSequenceCommandPrevious](#)

[mpiSequenceCommandRemove](#)

Get handle to Control

Get index number of sequence

Return handle to indexed command of sequence

Append command to sequence

Count the number of commands in sequence

Return handle to first command in sequence

Return the index of a command in sequence

Insert command into sequence

Return handle of last command in sequence

Get list of commands in sequence

Set list of commands in sequence

Get handle to next command in list

Get handle to previous command in list

Remove command from list

Data Types

[MPISequenceConfig](#) / [MEISequenceConfig](#)

[MPISequenceMessage](#)

[MPISequenceState](#)

[MPISequenceStatus](#)

[MEISequenceTrace](#)

See Also

[MPICommand](#)

[MPICommandType](#)

[MPICommandParams](#)

[seqKill.c](#) (sample application)

mpiSequenceCreate

Declaration	<code>MPISequence mpiSequenceCreate(MPIControl control,</code>
	<code>long number,</code>
	<code>long pageSize)</code>

Required Header

Description	SequenceCreate
	SequenceCreate creates a Sequence object associated with the program sequencer identified by <i>number</i> located on motion controller (control). SequenceCreate is the equivalent of a C++ constructor.

<i>If</i>	<i>Then</i>
number is -1	<i>SequenceCreate</i> selects the next unused program sequencer. If this is the first use of the program sequencer, then <i>SequenceCreate</i> will attempt to allocate <code>pageSize</code> firmware command slots.
pageSize is -1	<i>SequenceCreate</i> will allocate all remaining firmware command slots, which may prevent any more <i>Sequence</i> 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 <i>SequenceDelete</i> 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	<pre>long mpiSequenceConfigGet(MPISequence sequence, MPISequenceConfig *config, void *external)</pre>
Required Header	stdmpi.h
Description	SequenceConfigGet gets the configuration of a Sequence object (<i>sequence</i>) and writes it in the structure pointed to by <i>config</i>, and also writes it into the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> is not NULL). The Sequence's configuration information in <i>external</i> is in addition to the Sequence's configuration information in <i>config</i>, i.e, the configuration information in <i>config</i> and in <i>external</i> is not the same information. Note that <i>config</i> or <i>external</i> can be NULL (but not both NULL).
XMP Only	<i>external</i> either points to a structure of type MEISequenceConfig{ } or is NULL.
Return Values	
MPIMessageOK	if <i>SequenceConfigGet</i> 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 <i>SequenceConfigSet</i> 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 <i>SequenceFlashConfigGet</i> 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 <i>SequenceFlashConfigSet</i> 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

Description	SequencePageSize writes the <i>number</i> of command slots that are available to a Sequence (<i>sequence</i> , on its associated motion controller) to the contents of <i>pageSize</i> .

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

Description	SequenceStatus returns the status of a Sequence (<i>sequence</i>), and writes it into the structure pointed to by <i>status</i> , and also writes it into the implementation-specific structure pointed to by <i>external</i> (if <i>external</i> is not NULL).

sequence	a handle to a Sequence object
*status	a pointer to Sequence's status structure
*external	a pointer to an implementation-specific structure

XMP Only

external should always be set to NULL.

Return Values

MPIMessageOK	if <i>SequenceStatus</i> successfully returns the Sequence's status and writes the status to the structure(s)
MPIMessageARG_INVALID	if the <i>status</i> pointer is NULL.

See Also [MPISequenceStatus](#)

Required Header

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).

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

Return Values

MPIMessageOK	if <i>SequenceEventNotifyGet</i> successfully writes the event mask to the structure(s)

See Also [MEIEventMask](#) | [mpiSequenceEventNotifySet](#)

mpiSequenceEventReset

[illegible]

Required Header

Description	SequenceEventReset resets the event(s) that are specified in <i>eventMask</i> and generated by a Sequence (<i>sequence</i>). Your application should not call SequenceEventReset <i>until</i> one or more latchable events have occurred.

Return Values

MPIMessageOK	if <i>SequenceEventReset</i> successfully resets the event(s) that are specified in <i>eventMask</i> 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 <i>SequenceCompile</i> 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 `mpiEventManagerService(...)` (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
<i>command</i> is MPIHandleVOID	<i>SequenceLoad</i> loads Commands starting with the first Command of the Sequence
<i>start</i> is not FALSE	<i>SequenceLoad</i> starts the sequence after the commands are loaded

Return Values	
MPIMessageOK	if <i>SequenceLoad</i> successfully loads the firmware command slots of a Sequence

See Also

[mpiSequenceStart](#) | [mpiEventManagerService](#) | [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 <i>SequenceResume</i> successfully resumes a Sequence from the point where the Sequence has stopped

See Also

mpiSequenceStart

Declaration

```
long mpiSequenceStart(MPISequence sequence,  
                     MPICommand command)
```

Required Header

Description	SequenceStart begins the execution of a Sequence (<i>sequence</i>), starting with the Command (<i>command</i>). If <i>command</i> is MPIHandleVOID, execution starts with the first command of the Sequence.

Return Values

MPIMessageOK	if <i>SequenceStart</i> successfully begins the execution of a Command Sequence

See Also [mpiSequenceStop](#)

mpiSequenceStep

```
long mpiSequenceStep(MPISequence sequence,
                    long count)
```

Required Header

Description	SequenceStep executes <i>count</i> steps (Commands) of a stopped Sequence (<i>sequence</i>). After executing the Commands, the Sequence will be in the MPISequenceStateSTOPPED state.

Return Values

MPIMessageOK	if <i>SequenceStep</i> successfully executes <i>count</i> 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 <i>SequenceStop</i> successfully stops a Sequence (while it is executing)

See Also [mpiSequenceStart](#)

mpiSequenceMemory

```
long mpiSequenceMemory(MPISequence sequence,  
void **memory)
```

Required Header

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 <i>SequenceMemory</i> 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 <i>SequenceMemoryGet</i> 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

Description	<i>SequenceMemorySet</i> copies count bytes of application memory (starting at address src) to a Sequence's (sequence) memory (starting at address dst).

Return Values

MPIMessageOK	if <i>SequenceMemorySet</i> successfully copies <i>count</i> bytes of application memory to a Sequence object's memory

See Also [mpiSequenceMemory](#) | [mpiSequenceMemoryGet](#)

mpiSequenceControl

Declaration [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 <i>sequence</i> is invalid

See Also [mpiSequenceCreate](#) | [mpiControlCreate](#)

mpiSequenceNumber

```
long mpiSequenceNumber(MPISequence sequence,  
                      long *number)
```

Required Header

Description	SequenceNumber writes the index of a Sequence (<i>sequence</i> , on the motion controller that the Sequence object is associated with) to the contents of <i>number</i> .

Return Values

MPIMessageOK	if <i>SequenceNumber</i> successfully writes the Sequence's index to the contents of <i>number</i>

See Also

mpiSequenceCommand

Declaration `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 <i>index</i> th Command of a Sequence (<i>sequence</i>)
MPIHandleVOID	if <i>sequence</i> is invalid if <i>index</i> is less than 0 if <i>index</i> is greater than or equal to <code>mpiSequenceCount(sequence)</code>
MPIMessageARG_INVALID	if <i>index</i> is a negative number.
MEIListMessageELEMENT_NOT_FOUND	if <i>index</i> is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	if <i>sequence</i> is an invalid handle.

See Also

mpiSequenceCommandAppend

Declaration

```
long mpiSequenceCommandAppend(MPISequence sequence,  
                              MPICommand  command)
```

Required Header

Description	SequenceCommandAppend appends a Command (<i>command</i>) to a Sequence (<i>sequence</i>).

sequence	a handle to the Sequence object.
command	a handle to a Command object.

Return Values

MPIMessageOK	if <i>SequenceCommandAppend</i> successfully appends a Command to a Sequence
MPIMessageHANDLE_INVALID	Either <i>sequence</i> or <i>command</i> 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 (<i>sequence</i>)
-1	if <i>sequence</i> is invalid
0	if <i>sequence</i> is empty

See Also

mpiSequenceCommandFirst

Declaration [MPICommand](#) **mpiSequenceCommandFirst**([MPISequence](#) **sequence**)

Required Header stdmpi.h

Description **SequenceCommandFirst** returns the first element in the list. This function can be used in conjunction 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 (<i>sequence</i>)
MPIHandleVOID	if <i>sequence</i> is invalid if <i>sequence</i> is empty
MPIMessageHANDLE_INVALID	if <i>sequence</i> is an invalid handle.

See Also [mpiSequenceCommandNext](#) | [mpiSequenceCommandLast](#)

Required Header

sequence	a handle to the Sequence object.
command	a handle to a Command object.

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)

file:///D:/pdfs/030100/html/Software-MPI/docs/Sequence/Method/cmdinx1.htm [7/27/2004 12:26:14 PM]

mpiSequenceCommandInsert

[illegible]

Required Header

Description	SequenceCommandInsert inserts a Command (<i>insert</i>) in a Sequence (<i>sequence</i>) just after the specified Command (<i>command</i>).

Return Values

MPIMessageOK	if <i>SequenceCommandInsert</i> successfully inserts the Command (<i>insert</i>) in a Sequence following the specified Command (<i>command</i>)

See Also

mpiSequenceCommandLast

Declaration [MPICommand](#) **mpiSequenceCommandLast**([MPISequence](#) **sequence**)

Required Header `stdmpi.h`

Description **SequenceCommandLast** returns the last element in the list. This function can be used in conjunction 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 (<i>sequence</i>)
MPIHandleVOID	if <i>sequence</i> is invalid if <i>sequence</i> is empty
MPIMessageHANDLE_INVALID	if <i>sequence</i> is an invalid handle.

See Also [mpiSequenceCommandFirst](#) | [mpiSequenceCommandPrevious](#) | [mpiSequenceCommandNext](#)

mpiSequenceCommandListGet

[illegible]

Required Header

```
stdmpi.h
```

Description	SequenceCommandListGet gets the Commands in a Sequence (<i>sequence</i>). <i>SequenceCommandListGet</i> writes the number of Commands [in a Sequence (<i>sequence</i>)] 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 <i>SequenceCommandListGet</i> successfully gets the list of Commands in a Sequence

See Also [mpiSequenceCommandListSet](#)

mpiSequenceCommandListSet

Declaration	long	mpiSequenceCommandListSet	(MPISequence	sequence,
			long	commandCount,
			MPICommand	*commandList)

Required Header

Description	SequenceCommandListSet creates a Sequence (<i>sequence</i>) of <i>commandCount</i> Commands using the Command handles specified by <i>commandList</i> . 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 <i>SequenceCommandListGet</i> successfully creates a Sequence of Commands using the Command handles specified by <i>commandList</i>

See Also [mpiSequenceCommandListGet](#)

mpiSequenceCommandNext

[illegible]

Required Header

Description	SequenceCommandNext returns the next element following "command" on the list. This function can be used in conjunction with <code>mpiSequenceCommandFirst()</code> 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 (<i>command</i>) in a Sequence (<i>sequence</i>)
MPIHandleVOID	if <i>sequence</i> is invalid if <i>command</i> is the last command in a Sequence (<i>sequence</i>)
MPIMessageHANDLE_INVALID	Either <i>sequence</i> or <i>command</i> is an invalid handle.

See Also [mpiSequenceCommandFirst](#) | [mpiSequenceCommandPrevious](#)

mpiSequenceCommandPrevious

Declaration [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 conjunction 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

Description	SequenceCommandRemove removes a Command (<i>command</i>) from a Sequence (<i>sequence</i>).

Return Values

MPIMessageOK	if <i>SequenceCommandRemove</i> 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

The sequence number is out of range. This message code is returned by [mpiSequenceCreate\(...\)](#) if the sequence number is less than zero or greater than or equal to MEIXmpMAX_PSs. This message code is also returned if the specified sequence number is not active in the controller. To correct this problem, use [mpiControlConfigSet\(...\)](#) to enable the sequence object, by setting the sequenceCount to greater than the sequence number. For example, to enable sequence 0 to 3, set sequenceCount to 4. This message code is returned by [mpiSequenceLoad\(...\)](#) if the sequence buffer size and the sequence page size are not equal. This indicates an internal MPI Library problem.

MPISequenceMessageCOMMAND_COUNT

The sequence command count is out of range. This message code is returned by [mpiSequenceStart\(...\)](#) or [meiSequenceCompile\(...\)](#) if the sequence command count is less than or equal to zero. To correct this problem, set the command count to a value greater than zero.

MPISequenceMessageCOMMAND_NOT_FOUND

The sequence command is not found. This message code is returned by [mpiSequenceLoad\(...\)](#), [mpiSequenceStart\(...\)](#), or [meiSequenceCompile\(...\)](#) if the specified command is not a member of the sequence. To correct this problem, specify a command that is a member of the sequence.

MPISequenceMessageSTARTED

The program sequencer is already running. This message code is returned by [mpiSequenceResume\(...\)](#), [mpiSequenceStart\(...\)](#), or [mpiSequenceStep\(...\)](#) if the program sequencer has already been started. If this is a problem, call [mpiSequenceStop\(...\)](#) to stop the program sequencer or monitor the sequence status and wait for the state to equal STOPPED.

MPISequenceMessageSTOPPED

The program sequencer is not running. This message code is returned by [mpiSequenceStop\(...\)](#) if the program sequencer has already been stopped. If this is a problem, call [mpiSequenceStart\(...\)](#) to start the program sequencer.

See Also

MPISequenceState

MPISequenceState

```
typedef enum {  
    MPISequenceStateSTOPPED = 0,  
    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 <code>mpiSequenceStop</code> will stop the sequencer, and the sequencer state will be <code>MPISequenceStateSTOPPED</code> .
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 <code>mpiSequenceStart</code> .

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 **MEISequenceTrace** sets tracing on for the mpiSequenceLoad() method.

See Also [MPISequence](#) | [MEITrace](#) | [mpiSequenceLoad](#)

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(...)**.

NOTE: The Service object is not part of the standard MPI. In order to use the Service Object, the file, *apputil.h* needs to be included by your code and the apputil library needs to be linked to your application.

Methods

Create, Delete, Validate Methods

[ServiceCreate](#)

Create a Service for EventMgr and the threads necessary for it to run.

[ServiceDelete](#)

Stop all threads belonging to the Service and deletes the Service.

Configuration and Information Methods

[ServiceEnable](#)

Enable or disables the Service.

file:///D:/pdfs/030100/html/Software-MPI/docs/Service/Method/create1.htm [7/23/2004 11:44:22 AM]

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 <i>serviceDelete</i> 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 <i>service</i> .
TRUE	serviceEnable will enable <i>service</i> .

Return Values	
handle	to a Service object
MPIHandleVOID	if the Service could not be created

See Also

SqNode Objects

Introduction

A **SqNode** object manages a single SynqNet network node connected to a SynqNet network. It represents the physical network node. It contains information about the node, as well as its status and configuration. It provides read/write access to the node via network cyclic data and service commands. It also provides an interface to any drives connected to the node.

During network initialization, the SynqNet nodes are discovered and mapped to the SynqNet object. The number of motors per SqNode is determined and mapped to the controller's motor objects. Each node connected to a controller is assigned a number (0, 1, 2, etc) in the order it is discovered. The node number is used to index the SqNode objects.

Methods

Create, Delete, Validate Methods

[meiSqNodeCreate](#)

[meiSqNodeDelete](#)

[meiSqNodeValidate](#)

Configuration and Information Methods

[meiSqNodeCommand](#)

[meiSqNodeConfigGet](#)

[meiSqNodeConfigSet](#)

[meiSqNodeFlashConfigGet](#)

[meiSqNodeFlashConfigSet](#)

[meiSqNodeFpgaDefaultFileName](#)

[meiSqNodeInfo](#)

[meiSqNodeStatus](#)

[meiSqNodeStatusClear](#)

[meiSqNodeUserDataGet](#)

[meiSqNodeUserDataSet](#)

Drive Interface Methods

[meiSqNodeDriveConfigGet](#)

[meiSqNodeDriveConfigSet](#)

[meiSqNodeDriveInfo](#)

[meiSqNodeDriveMapParamCount](#)

[meiSqNodeDriveMapParamList](#)

[meiSqNodeDriveMapConfigCount](#)

[meiSqNodeDriveMapConfigList](#)

[meiSqNodeDriveMapParamFileGet](#)

[meiSqNodeDriveMapParamFileSet](#)

[meiSqNodeDriveMonitor](#)

[meiSqNodeDriveMonitorConfigGet](#)

[meiSqNodeDriveMonitorConfigSet](#)

[meiSqNodeDriveParamCalculate](#)

[meiSqNodeDriveParamClear](#)

[meiSqNodeDriveParamGet](#)

[meiSqNodeDriveParamListGet](#)

[meiSqNodeDriveParamListSet](#)

[meiSqNodeDriveParamReload](#)

[meiSqNodeDriveParamRestore](#)

[meiSqNodeDriveParamSet](#)

[meiSqNodeDriveParamStore](#)

I/O Methods

[meiSqNodeAnalogInput](#)

[meiSqNodeDigitalIn](#)

[meiSqNodeDigitalInBit](#)

[meiSqNodeDigitalOutBitGet](#)

[meiSqNodeDigitalOutBitSet](#)

[meiSqNodeDigitalOutGet](#)

[meiSqNodeDigitalOutSet](#)

Action Methods

[meiSqNodeDownload](#)

[meiSqNodeFlashErase](#)

[meiSqNodeFpgaFileNameVerify](#)

[meiSqNodeVerify](#)

Event Methods

[meiSqNodeEventNotifyGet](#)

[meiSqNodeEventNotifySet](#)

[meiSqNodeEventReset](#)

Memory Methods

[meiSqNodeMemory](#)

[meiSqNodeMemoryGet](#)[meiSqNodeMemorySet](#)

Relational Methods

[meiSqNodeControl](#)[meiSqNodeNumber](#)

Data Types

[MEISqNodeCallback](#)[MEISqNodeChannel](#)[MEISqNodeCmdType](#)[MEISqNodeCmdHeader](#)[MEISqNodeCommand](#)[MEISqNodeConfig](#)[MEISqNodeConfigIoAbort](#)[MEISqNodeConfigAlarm](#)[MEISqNodeConfigPacketError](#)[MEISqNodeConfigTrigger](#)[MEISqNodeConfigUserFault](#)[MEISqNodeDataSize](#)[MEISqNodeDownloadParams](#)[MEISqNodeDriveInfo](#)[MEISqNodeDriveMonitor](#)[MEISqNodeDriveMonitorConfig](#)[MEISqNodeDriveMonitorData](#)[MEISqNodeDriveMonitorDataType](#)[MEISqNodeDriveParamCallback](#)[MEISqNodeDriveParamCallbackType](#)[MEISqNodeFeedbackSecondary](#)[MEISqNodeFileName](#)[MEISqNodeFpgaType](#)[MEISqNodeInfo](#)[MEISqNodeInfoId](#)[MEISqNodeInfoIo](#)[MEISqNodeInfoFpga](#)[MEISqNodeInfoNetwork](#)[MEISqNodeMemory](#)[MEISqNodeMessage](#)

[MEISqNodeMonitorValue](#)

[MEISqNodeMonitorValueIndex](#)

[MEISqNodeResponse](#)

[MEISqNodeStatus](#)

[MEISqNodeStatusCrcError](#)

[MEISqNodeStatusPacketError](#)

[MEISqNodeUserData](#)

Constants

[MEISqNodeID_CHAR_MAX](#)

[MEISqNodeFILENAME_MAX](#)

[MEISqNodeManufacturerDATA_CHAR_MAX](#)

[MEISqNodeMaxFEEDBACK_SECONDARY](#)

[MEISqNodeMaxMOTORS](#)

[MEISqNodeNOT_AVAILABLE](#)

[MEISqNodeSTATUS_NOT_AVAILABLE](#)

[MEISqNodeUserData_CHAR_MAX](#)

[MEIDriveMapParamMAX_STRING_LENGTH](#)

[MEIFPGARINCONREV](#)

[MEIFpgaSqMACVersionDEFAULT](#)

[MEIFpgaSqMACVersionMIN](#)

[MEIFpgaSqMACVersionMAX](#)

[MEIFpgaSqNodeVersionDEFAULT](#)

[MEIFpgaSqNodeVersionMIN](#)

[MEIFpgaSqNodeVersionMAX](#)

Required Header

```
stdmei.h
```

SqNodeCreate is the equivalent of a C++ constructor.

Return Values

See Also [meiSqNodeDelete](#) | [meiSqNodeValidate](#)

meiSqNodeDelete

Declaration long [meiSqNodeDelete](#)([MEISqNode](#) **node**);

Required Header stdmei.h

Description [SqNodeDelete](#) deletes a SqNode object and invalidates its handle.

SqNodeDelete is the equivalent of a C++ destructor.

node	a handle of the SqNode object to delete in the reverse order to avoid memory leaks.

Return Values

MPIMessageOK	if <i>SqNode</i> successfully deleted the object.

See Also [meiSqNodeCreate](#) | [meiSqNodeValidate](#)

meiSqNodeValidate

Declaration long **meiSqNodeValidate**([MEISqNode](#) **node**);

Required Header stdmei.h

Description **SqNodeValidate** validates a SqNode object and its handle.

SqNodeValidate is the equivalent of a C++ constructor.

node	a handle to SqNode object.

Return Values

MPIMessageOK	if <i>SqNode</i> is a handle to a valid object.

See Also [meiSqNodeCreate](#) | [meiSqNodeDelete](#)

meiSqNodeCommand

Declaration

long **meiSqNodeCommand**([MEISqNode](#) node,
[MEISqNodeCommand](#) *command,
[MEISqNodeResponse](#) *response);

Required Header

stdmei.h

Description

SqNodeCommand sends a service command to a SynqNet node using the data from the structure pointed to by command and writes the response into the structure pointed to by response. Service commands occur across the SynqNet network through a service channel. In SYNQ mode there is one service channel for each node. In ASYNQ mode there is one service channel for all nodes. The controller sends the command and waits for a response using a 4 state handshake. In SYNQ mode the service command data is sent through the cyclic packets, but due to the handshaking, it is not considered a cyclic operation.

For SynqNet nodes that have drive memory interfaces, service commands can be sent to drives. Also, the service commands supports access to drive data memory, program memory, I/O memory, and direct commands. Please see the `MEISqNodeCommand{...}` and `MEISqNodeResponse{...}` structures for more information. And be sure to consult the drive's header file in the `(C:\MEI)\XMP\sqNodeLib\include` directory, as well as, the drive manufacturer's manual for valid drive addresses.

node	a handle to a SynqNet node object
*command	a pointer to a SynqNet node command structure
*response	a pointer to a SynqNet node response structure

Return Values	
MPIMessageOK	if <i>SqNodeCommand</i> successfully sends a service command and receives a response.
MPIMessageARG_INVALID	if the command pointer is NULL or the response pointer is NULL
MEISynqNetMessageRESPONSE_TIMEOUT	if the node does not respond to the service command.
MEISynqNetMessageREADY_TIMEOUT	if the node is busy and does not acknowledge the service command.

See Also

[MEISqNodeCommand](#) | [MEISqNodeResponse](#) | [MEISqNodeCmdHeader](#) | [MEISqNodeCmdType](#) | [MEISqNodeDataSize](#) | [MEISqNodeMemory](#) | [MEISqNodeChannel](#)

meiSqNodeConfigGet

```
Declaration      long meiSqNodeConfigGet(MEISqNode node,  
                  MEISqNodeConfig *config);
```

Required Header

Description	SqNodeConfigGet reads a SynqNet node's (<i>node</i>) configuration and writes it into the structure pointed to by <i>config</i> .

node	a handle to a SynqNet node object.
*config	a pointer to a SynqNet node config structure.

Return Values

MPIMessageOK	if <i>SqNodeConfigGet</i> successfully reads the node configuration.

See Also [meiSqNodeInfo](#) | [meiSqNodeDriveConfigGet](#)

meiSqNodeConfigSet

```
Declaration      long  meiSqNodeConfigSet(MEISqNode      node,  
                   MEISqNodeConfig *config);
```

Required Header

```
stdmei.h
```

Description	SqNodeConfigSet writes a SynqNet node's (<i>node</i>) configuration using data from the structure pointed to by <i>config</i> .

node	a handle to a SynqNet node object
*config	a pointer to a SynqNet node config structure

Return Values

MPIMessageOK	if <i>SqNodeConfigGet</i> successfully writes the node configuration.
MEIMessageARG_INVALID	if the upStreamError or downStreamError fault/fail limits are less than 0 or greater than 255.
MEISqNodeMessageFEEDBACK_MAP_INVALID	if secondary encoder (n) is not mappable to the motor on the node specified by MEISqNodeFeedbackSecondary[n].motorIndex..

See Also [meiSqNodeInfo](#) | [meiSqNodeDriveConfigSet](#) | [meiSynqNetFlashTopologySave](#)

Required Header

node	a handle to a SynqNet node object.
*flash	<i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally.
*config	a pointer to a SynqNet node config structure.

MPIMessageOK if *SqNodeFlashConfigGet* successfully reads a SynqNet node's flash configuration and writes it into the structure pointed to by *config*.

file:///D:/pdfs/030100/html/Software-MPI/docs/sqNode/Method/flacfgget2.htm [7/26/2004 9:11:59 AM]

meiSqNodeFlashConfigSet

Declaration

long meiSqNodeFlashConfigSet(MEISqNode node,
 void *flash,
 MEISqNodeConfig *config);

Required Header

stdmei.h

Description

SqNodeFlashConfigSet sets a SynqNet Node (*node*) flash configuration using data from the structure pointed to by *config*.

NOTE: The network topology must first be saved before changing node config values in Flash memory. These values will also be cleared when network topology is cleared using [meiSynqNetFlashTopologyClear\(...\)](#).

node	a handle to a SynqNet node object.
*flash	<i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally. If <i>flash</i> is a valid MEIFlash handle, then the MEIFlash object cache will be updated, but the actual write to controller flash will not occur. Use meiFlashMemoryFromFileType(...) to prompt the actual write to <i>flash</i> .
*config	a pointer to a SynqNet node config structure.

Return Values	
MPIMessageOK	if <i>SqNodeFlashConfigSet</i> successfully sets a SynqNet Node (<i>node</i>) flash configuration using data from the structure pointed to by <i>config</i> .
MEISqNodeMessageFEEDBACK_MAP_INVAILD	given secondary encoder (<i>n</i>) is not mappable to the motor on the node specified by MEISqNodeFeedbackSecondary[n].motorIndex.
MPIMessageARG_INVALID	If the upStreamError or downStreamError fault/fail limits are less than 0 or greater than 255.
MEIFlashMessageNETWORK_TOPOLOGY_ERROR	if a valid flash handle was supplied and SynqNet topology had not yet been saved to flash using meiSynqNetFlashTopologySave(...) .

meiSqNodeFpgaDefaultFileName

Declaration

```
long meiSqNodeFpgaDefaultFileName(MEISqNode sqNode, MEISqNodeFileName *fileName);
```

Required Header stdmei.h

Description [SqNodeFpgaFileDefaultFileName](#) provides the default image filename for an sqNode.

sqNode	handle to a SqNode object.
*fileName	a pointer to a structure that has space allocated to hold an FPGA filename.

Return Values	
MPIMessageOK	if <i>SqNodeFpgaDefaultFileName</i> successfully returns the name of an FPGA image file.
MPIMessageARG_INVALID	if the pointer to filename is NULL

See Also

Required Header

```
stdmei.h
```

The RMB-10V, RMB-10V2 and some Trust nodes support analog inputs. MPI support has been added to support the reading of node-based analog inputs. The number of analog inputs a node supports can be determined with `meiSqNodeInfo(...)`. An analog input value can be read with [`meiSqNodeAnalogIn\(...\)`](#). The analog to digital converted value is scaled from -1.0 to +1.0, where +1.0 is a full-scale positive voltage. The input range of the ADC is hardware-specific.

node	a handle to a SynqNet node object.
*info	a pointer to a drive specific information structure.

Return Values

MPIMessageOK	if <i>SqNodeInfo</i> successfully reads the node information.
MPIMessageARG_INVALID	if the info pointer is NULL.

See Also [meiSqNodeDriveInfo](#) | [meiSqNodeConfigGet](#) | [meiSqNodeDriveConfigGet](#) | [meiSqNodeAnalogIn](#)

Required Header

```
stdmei.h
```

NOTE: This data requires service commands to access the data on the node. As a result, it may take up to 9 servo cycles to read the data. At the default sample rate of 2kHz, this would translate to 4.5ms.

Return Values	
MPIMessageOK	if <i>SynqNetStatus</i> successfully reads the node status and writes it into the structure.
MPIMessageARG_INVALID	if the status pointer is NULL.

file:///D:/pdfs/030100/html/Software-MPI/docs/sqNode/Method/sts2.htm [7/26/2004 9:12:01 AM]

meiSqNodeStatusClear

Declaration long **meiSqNodeConfigSet**([MEISqNode](#) **node**);

Required Header stdmei.h

Description **SqNodeStatusClear** clears node CRC errors on all ports, clears node Packet errors, clears node ioAbort state, and resets SqNode events.

node	a handle to a SynqNet node object

Return Values

MPIMessageOK	if <i>SqNodeStatusClear</i> successfully ...

See Also [MEIEventTypeSQNODE](#)

meiSqNodeUserDataGet

```
Declaration      long meiSqNodeUserDataGet(MEISqNode node,  
                   MEISqNodeUserData *data);
```

Required Header

Description	SqNodeUserDataGet reads the user data from the node.

node	a handle to a SynqNet node object.
*data	a pointer to a MEISqNodeUserData structure, allocated on the host.

Return Values

MPIMessageOK	if <i>SqNodeUserDataGet</i> successfully gets the user data from the node.

See Also [meiSqNodeUserDataSet](#)

meiSqNodeUserDataSet

```
Declaration      long  meiSqNodeUserDataSet(MEISqNode      node,  
                  MEISqNodeUserData  *data);
```

Required Header

Description	SqNodeUserDataSet writes the user data to the node.

node	a handle to a SynqNet node object.
*data	a pointer to a MEISqNodeUserData structure, allocated on the host.

Return Values

MPIMessageOK	if <i>SqNodeUserDataSet</i> successfully sets the user data to the node.

See Also [meiSqNodeUserDataGet](#) | [MEISqNodeUserData](#)

meiSqNodeDriveConfigGet

Declaration

```
long meiSqNodeDriveConfigGet(MEISqNode node,
                             long driveIndex, /* relative to the node */
                             void *config); /* node specific */
```

Required Header stdmei.h

Description [SqNodeDriveConfigGet](#) reads a SynqNet node's drive configuration and writes it into a drive specific structure pointed to by *config*. SynqNet nodes may support one or more drive interfaces. The drive configuration can be read if the drive interface hardware supports a communication channel to the drive processor. The drive interface(s) for a SynqNet node are indexed by a number (0, 1, 2, etc.).

The drive configuration structure is drive specific. The SqNodeLib includes the drive specific structures and methods. Please see the drive's header file in the (C:\MEI)\XMP\sqNodeLib\include directory, as well as, the drive manufacturer's documentation for details. Use meiSqNodeInfo(...), to determine if the SynqNet node supports a drive interface and it's type.

node	a handle to a SynqNet node object
driveIndex	an index to a drive interface on a SynqNet node. The first drive interface is 0, the second is 1, etc.
*config	a pointer to a drive specific configuration structure.

Return Values

MPIMessageOK	if <i>SqNodeDriveConfigGet</i> successfully reads the drive configuration.

See Also [meiSqNodeInfo](#) | [meiSqNodeDriveInfo](#) | [meiSqNodeConfigGet](#) |

See Also

[meiSqNodeFlashConfigGet](#) | [Flash Objects](#) | [meiSynqNetFlashTopologySave](#)

meiSqNodeDriveConfigSet

Declaration

```
long meiSqNodeDriveConfigSet(MEISqNode node,
                             long driveIndex, /* relative to the node */
                             void *config); /* node specific */
```

Required Header stdmei.h

Description [SqNodeDriveConfigSet](#) writes a SynqNet node's drive configuration from a drive specific structure pointed to by *config*. SynqNet nodes may support one or more drive interfaces. The drive configuration can be written if the drive interface hardware supports a communication channel to the drive processor. The drive interface(s) for a SynqNet node are indexed by a number (0, 1, 2, etc.).

The drive configuration structure is drive specific. The SqNodeLib includes the drive specific structures and methods. Please see the drive manufacturer's documentation for details. Use meiSqNodeInfo(...), to determine if the SynqNet node supports a drive interface and its type.

node	a handle to a SynqNet node object
driveIndex	an index to a drive interface on a SynqNet node. The first drive interface is 0, the second is 1, etc.
*config	a pointer to a drive specific configuration structure.

Return Values

MPIMessageOK	if <i>SqNodeDriveConfigSet</i> successfully writes the drive configuration.

See Also [meiSqNodeInfo](#) | [meiSqNodeDriveInfo](#) | [meiSqNodeConfigSet](#) |

meiSqNodeDriveInfo

Declaration

```
long meiSqNodeDriveInfo(MEISqNode node,
                        long driveIndex, /* relative to the node */
                        MEISqNodeDriveInfo *info,
                        void *external); /* node specific */
```

Required Header stdmei.h

Description **SqNodeDriveInfo** reads a SynqNet node's drive information and writes it into a drive specific structure pointed to by *info*. The drive info structure contains read only data. SynqNet nodes may support one or more drive interfaces. The drive information can be read if the drive interface hardware supports a communication channel to the drive processor. The drive interface(s) for a SynqNet node are indexed by a number (0, 1, 2, etc.).

The drive information structure is drive specific. The SqNodeLib includes the drive specific structures and methods. Please see the drive's header file for the drive specific information structures, as well as, the drive manufacturer's documentation for details. All supported drive header files are located in the (C:\MEI)\XMP\sqNodeLib\include directory. Use meiSqNodeInfo(...), to determine if the SynqNet node supports a drive interface and it's type.

node	a handle to a SynqNet node object.
driveIndex	an index to a drive interface on a SynqNet node. The first drive interface is 0, the second is 1, etc.
*info	a pointer to a structure that contains general drive information.
*external	a pointer to a drive specific information structure. See the appropriate drive vendor *.h for definition. (NOTE: it can be NULL)

Return Values

MPIMessageOK if *SqNodeDriveInfo* successfully reads the drive information.

See Also [meiSqNodeInfo](#) | [meiSqNodeConfigGet](#) | [meiSqNodeDriveConfigGet](#) |

meiSqNodeDriveMapParamCount

Declaration

```
long meiSqNodeDriveMapParamCount (MEISqNode sqNode,
                                   MEIDriveMap driveMap,
                                   long driveIndex,
                                   long *paramsCount);
```

Required Header stdmei.h

Description **SqNodeDriveMapParamCount** scans the drive map file for a drive entry that matches this node on the network. If an entry is found, then this function returns the number of drive parameters that need to be preserved for the configuration of the drive.

This function is normally used with the meiSqNodeDriveMapParamList function. First, this function is called in order to get the size of the drive parameter list. Then the user can use this size to allocate enough memory to hold the complete parameter list before calling meiSqNodeDriveMapParamList to fill in the list.

sqNode	a handle to a SynqNet node object.
driveMap	a handle to a DriveMap object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
*paramsCount	pointer to the variable that will be set by this function.

Return Values

MPIMessageOK	if meiSqNodeDriveMapParamCount successfully scans the drive map file for a drive entry that matches this node on the network and returns the number of drive parameters that need to be preserved for the configuration of the drive.

See Also [meiSqNodeDriveMapParamList](#)

meiSqNodeDriveMapParamList

Declaration

```
long meiSqNodeDriveMapParamList (MEISqNode
                                MEIDriveMap
                                long
                                long
                                MEIDriveParamInfo
                                sqNode,
                                driveMap,
                                driveIndex,
                                paramsCount,
                                *driveParamInfo);
```

Required Header stdmei.h

Description **SqNodeDriveMapParamList** scans the drive map file for an entry that matches the node on the network. If a drive entry is found, this function writes the drive parameter information about each of the drive parameters to the *driveParamInfo* list.

This function is normally used with the meiSqNodeDriveMapParamCount function. The meiSqNodeDriveMapParamCount function is called first to get the size of the parameter list, the user can then use this size to allocate enough memory to hold the complete parameter list before calling this function to fill in the parameter list.

sqNode	a handle to a SynqNet node object.
driveMap	a handle to a DriveMap object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
paramsCount	the number of drive parameter information records that can be written to the <i>driveParamInfo</i> list.
*driveParamInfo	pointer to the list of drive parameter information records that will be filled in by this function.

Return Values

MPIMessageOK	if <i>meiSqNodeDriveMapParamList</i> successfully scans the drive map file for an entry that matches the node on the network and writes the information about each of the drive parameters to the <i>driveParamInfo</i> list.

See Also [meiSqNodeDriveMapParamCount](#)

meiSqNodeDriveMapConfigCount

Declaration

```
long meiSqNodeDriveMapConfigCount (MEISqNode sqNode,
                                   MEIDriveMap driveMap,
                                   long driveIndex,
                                   long *configCount);
```

Required Header stdmei.h

Description **SqNodeDriveMapScanConfigCount** scans the drive map file for a drive entry that matches this node on the network. If an entry is found, this function returns the number of drive parameters that need to be preserved for the configuration of the drive.

This function is normally used with the meiSqNodeDriveMapConfigList function. This function is first called to get the size of the drive configuration list. Then the user can use this size to allocate enough memory to hold the complete configuration list before calling meiSqNodeDriveMapConfigList to fill in the list.

sqNode	a handle to a SynqNet node object.
driveMap	a handle to a DriveMap object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
*configCount	pointer to the variable that will be set by this function.

Return Values

MPIMessageOK	if meiSqNodeDriveMapConfigCount successfully scans the drive map file for a drive entry that matches this node on the network and returns the number of drive parameters that need to be preserved for the configuration of the drive.

See Also [meiSqNodeDriveMapConfigList](#)

meiSqNodeDriveMapConfigList

Declaration

```
long meiSqNodeDriveMapConfigList (MEISqNode sqNode,
                                  MEIDriveMap driveMap,
                                  long driveIndex,
                                  long configCount,
                                  long *configList);
```

Required Header stdmei.h

Description **SqNodeDriveMapScanConfigList** scans the drive map file for a drive entry that matches this node on the network. If an entry is found, this function returns the list of drive parameters that need to be preserved for the configuration of the drive.

This function is normally used with the meiSqNodeDriveMapConfigCount function. The meiSqNodeDriveMapConfigCount function is first called to get the size of the drive configuration list. Then the user can use this size to allocate enough memory to hold the complete configuration list before calling this function to fill in the list.

sqNode	a handle to a SynqNet node object.
driveMap	a handle to a DriveMap object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
configCount	the number of drive parameter information records that can be written to the <i>configList</i> list.
*configList	pointer to the list of drive parameters that make up the drive configuration that will be filled in by this function.

Return Values

MPIMessageOK	if <i>meiSqNodeDriveMapConfigList</i> successfully scans the drive map file for a drive entry that matches this node on the network and returns the list of drive parameters that need to be preserved for the configuration of the drive.

See Also [meiSqNodeDriveMapConfigCount](#)

meiSqNodeDriveMapParamFileGet

Declaration

```
long meiSqNodeDriveMapParamFileGet(MEISqNode sqNode,  
                                     MEIDriveMap driveMap,  
                                     long driveIndex,  
                                     char *driveConfigFilename,  
                                     long append);
```

Required Header stdmei.h

Description [SqNodeDriveMapParamFileGet](#) saves the current set of drive parameters in the drive to the *driveConfigFilename*.

sqNode	a handle to the SynqNet node object.
driveMap	a handle to the DriveMap object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
*driveConfigFilename	the name of the file that holds the stored drive configuration file.
append	1 = The new data is appended to the existing drive configuration file if it exists. 0 = A new drive configuration file is created to hold the drive parameters. If a file already exists, it will be overwritten.

Return Values

MPIMessageOK	if <i>SqNodeDriveMapParamFileGet</i> successfully saves the current set of drive parameters in the drive to the <i>driveConfigFilename</i> .

See Also [meiSqNodeDriveMapParamFileSet](#)

meiSqNodeDriveMapParamFileSet

Declaration

```
long meiSqNodeDriveMapParamFileSet(MEISqNode sqNode,  
                                   MEIDriveMap driveMap,  
                                   long driveIndex,  
                                   char *driveConfigFilename,  
                                   MEISqNodeDriveParamCallback callback,  
                                   long warning);
```

Required Header stdmei.h

Description [SqNodeDriveMapParamFileSet](#) loads the drive parameters stored in the file named *driveConfigFilename* into the drive.

sqNode	a handle to the SynqNet node object.
driveMap	a handle to the DriveMap object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
*driveConfigFilename	the name of the file that holds the stored drive configuration file.
callback	A callback function that this function calls to indicate if the function is changing the value of a drive parameter or setting a new drive parameter that has failed. Passing NULL for this parameter will disable the callback feature.
warning	0 = if setting a drive parameter fails, this function will fail immediately. 1 = if setting a drive parameter fails, then the function will continue with the remaining drive parameters and generate a warning by calling the callback function.

Return Values

MPIMessageOK	if <i>SqNodeDriveMapParamFileSet</i> successfully loads the drive parameters stored in the file named <i>driveConfigFilename</i> into the drive.

See Also [meiSqNodeDriveMapParamFileGet](#)

meiSqNodeDriveMonitor

Declaration

```
long meiSqNodeDriveMonitor(MEISqNode node,
                           long driveIndex, /* relative to the node
*/
                           MEISqNodeMonitorValue *value);
```

Required Header stdmei.h

Description [SqNodeDriveMonitor](#) reads the monitor fields from the drive and writes them into the structure pointed to by *value*.

node	a handle to a SynqNet node object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
*value	pointer to a structure of monitor values

Return Values

MPIMessageOK	if <i>SqNodeDriveMonitor</i> successfully reads the monitor information.

See Also [MEISqNodeMonitorValue](#)

meiSqNodeDriveMonitorConfigGet

Declaration

```
long meiSqNodeDriveMonitorConfigGet(MEISqNode node,  
                                     long driveIndex, /* relative to the node  
*/  
                                     MEISqNodeDriveMonitorConfig *config);
```

Required Header stdmei.h

Description **SqNodeDriveMonitorConfigGet** reads a SynqNet node's drive monitor configuration and writes it into a structure pointed to by **config**. SynqNet nodes may support one or more drive interfaces. The drive monitor configuration can be read if the drive interface hardware supports a communication channel to the drive processor. The drive interface(s) for a SynqNet node are indexed by a number (0, 1, 2, etc.).

The SynqNet network packets have some extra fields that can be configured to read drive data every sample. Each monitor field is 32 bits. SynqNet nodes with drive interfaces that support drive monitoring can be configured to transmit the data. The drive manufacturer determines what data is available for monitoring. The monitor data can be specified by a predetermined index or memory address. Please see the drive's header file for the drive specific configuration structures, as well as, the drive manufacturer's documentation for details. All supported drive header files are located in the (C:\MEI)\XMP\sqNodeLib\include directory.

node	a handle to a SynqNet node object.
driveIndex	an index to a drive interface on a SynqNet node. The first drive interface is 0, the second is 1, etc.
*config	a pointer to a drive monitor configuration structure.

Return Values

MPIMessageOK if *SqNodeDriveMonitorConfigGet* successfully reads the drive monitor configuration.

See Also [meiSqNodeInfo](#) | [meiSqNodeDriveInfo](#)

meiSqNodeDriveMonitorConfigSet

Declaration

```
long meiSqNodeDriveMonitorConfigSet(MEISqNode node,
                                     long driveIndex, /* relative to the node */
                                     MEISqNodeDriveMonitorConfig *config);
```

Required Header stdmei.h

Description [SqNodeDriveMonitorConfigSet](#) writes a SynqNet node's drive monitor configuration from a structure pointed to by *config*. SynqNet nodes may support one or more drive interfaces. The drive monitor configuration can be written if the drive interface hardware supports a communication channel to the drive processor. The drive interface(s) for a SynqNet node are indexed by a number (0, 1, 2, etc.).

The SynqNet network packets have some extra fields that can be configured to read drive data every sample. Each monitor field is 32 bits. SynqNet nodes with drive interfaces that support drive monitoring can be configured to transmit the data. The drive manufacturer determines what data is available for monitoring. The monitor data can be specified by a predetermined index or memory address. Please see the drive's header file for the drive specific configuration structures, as well as, the drive manufacturer's documentation for details. All supported drive header files are located in the (C:\MEI)\XMP\sqNodeLib\include directory.

node	a handle to a SynqNet node object.
driveIndex	an index to a drive interface on a SynqNet node. The first drive interface is 0, the second is 1, etc.
*config	a pointer to a drive monitor configuration structure.

Return Values

MPIMessageOK if *SqNodeDriveMonitorConfigSet* successfully writes the drive monitor configuration.

See Also [meiSqNodeInfo](#) | [meiSqNodeDriveInfo](#)

Required Header

```
stdmei.h
```

Return Values

See Also

Required Header

NOTE: This function may not be supported by all drives. The default set of drive parameters may be different between different drive types and different drive manufactures.

Return Values

See Also [meiSqNodeDriveParamReload](#)

meiSqNodeDriveParamGet

Declaration

```
long meiSqNodeDriveParamGet (MEISqNode node,
                             long driveIndex,
                             long param,
                             MEIDriveMapParamType paramType,
                             MEIDriveMapParamValue *value);
```

Required Header stdmei.h

Description **SqNodeDriveParamGet** reads a drive parameter from the drive and fills in the appropriate field of the union pointed to by **value**. The **paramType** defines the type of data that is read from the drive and also defines which field will be used in the **value** union.

node	a handle to the SynqNet node object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
param	an index for the drive parameter that is being accessed.
paramType	the type of the data read from the drive and which field will be used in the <i>value</i> union.
*value	a pointer to the union that will be filled in.

Return Values

MPIMessageOK	if <i>SqNodeDriveParamGet</i> successfully reads a drive parameter from the drive and fills in the appropriate field of the union pointed to by value .

See Also [meiSqNodeDriveParamSet](#)

meiSqNodeDriveParamListGet

Declaration

```
long meiSqNodeDriveParamListGet( MEISqNode node,  
                                long driveIndex,  
                                long size,  
                                long *paramList,  
                                MEIDriveMapParamType *paramTypes,  
                                MEIDriveMapParamValue *paramValues);
```

Required Header stdmei.h

Description [SqNodeDriveParamListGet](#) reads a series of drive parameters from the drive and fills in the appropriate fields of the unions pointed to by *paramValues*. The *paramTypes* defines the type of data that is read from the drive and also defines which fields in the *paramValues* unions are going to be used.

node	a handle to the SynqNet node object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
size	the number of drive parameters to be read.
*paramList	a pointer to a list of the drive parameter indexes that are being accessed.
*paramTypes	a pointer to a list of drive parameter types to be read from the drive and which field in the paramValues union is going to be used.
*paramValues	a pointer to a list of unions that will be filled in by this function.

Return Values

MPIMessageOK	if <i>SqNodeDriveParamListGet</i> successfully reads a series of drive parameters from the drive and fills in the appropriate fields of the unions pointed to by <i>paramValues</i> .

See Also [meiSqNodeDriveParamListSet](#)

meiSqNodeDriveParamListSet

Declaration

```
long meiSqNodeDriveParamListSet(MEISqNode node,  
                                long driveIndex,  
                                long size,  
                                long *paramList,  
                                MEIDriveMapParamType *paramTypes,  
                                MEIDriveMapParamValue *paramValues);
```

Required Header stdmei.h

Description [SqNodeDriveParamListSet](#) writes a series of drive parameters pointed to by *value* to the drive. The *paramTypes* defines each type of data item that is written to the drive and also defines which field in the *paramValues* unions are going to be used.

node	a handle to the SynqNet node object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
size	the number of drive parameters to be written.
*paramList	a pointer to a list of drive parameter types to be written to the drive and which field in the paramValues union is going to be used.
*paramTypes	a pointer to a list of the drive parameter indexes that are being accessed.
*paramValues	a pointer to a list of unions that will be written to the drive.

Return Values

MPIMessageOK	if <i>SqNodeDriveParamListSet</i> successfully writes a series of drive parameters pointed to by <i>value</i> to the drive.

See Also [meiSqNodeDriveParamListGet](#)

meiSqNodeDriveParamReload

Declaration	long	meiSqNodeDriveParamReload	(<u>MEISqNode</u>	sqNode,
			long	driveIndex);

Required Header

Description	SqNodeDriveParamReload overwrites the current set of drive parameters with the set from the non-volatile storage on the drive. These new drive parameters will take effect immediately.

sqNode	a handle to a SynqNet node object
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.

Return Values

MPIMessageOK	if <i>SqNodeDriveParamReload</i> successfully overwrites the current set of drive parameters with the set from the non-volatile storage on the drive.

See Also [meiSqNodeDriveParamClear](#)

Required Header

```
stdmei.h
```

sqNode	a handle to a SynqNet node object
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.

MPIMessageOK	if <i>SqNodeDriveParamRestore</i> successfully loads the default set of drive parameters into current set of drive parameters on the drive.

file:///D:/pdfs/030100/html/Software-MPI/docs/sqNode/Method/drvparrestore2.htm [7/26/2004 9:12:09 AM]

meiSqNodeDriveParamSet

Declaration

```
long meiSqNodeDriveParamSet(MEISqNode node,  
                             long driveIndex,  
                             long param,  
                             MEIDriveMapParamType paramType,  
                             MEIDriveMapParamValue *value);
```

Required Header stdmei.h

Description [SqNodeDriveParamSet](#) writes the drive parameter that is pointed to by *value* to the drive. The *paramType* defines the type of data that is written to the drive and also defines which field will be used in the *value* union.

node	a handle to the SynqNet node object.
driveIndex	an index to the drive (0, 1, 2, etc), relative to the node.
param	an index for the drive parameter that is being accessed.
paramType	the type of data being written to the drive and which field will be used in the <i>value</i> union.
*value	pointer to the union that will be written to the drive.

Return Values

MPIMessageOK	if <i>SqNodeDriveParamSet</i> successfully writes the drive parameter that is pointed to by <i>value</i> to the drive.

See Also [meiSqNodeDriveParamGet](#)

Required Header

```
stdmei.h
```

NOTE: This function may not be supported by all drives.

Return Values

See Also [meiSqNodeDriveParamRestore](#)

meiSqNodeAnalogInput

Declaration

long meiSqNodeAnalogInput(MEISqNode
 long node,
 double channel,
 *state);

Required Header

stdmei.h

Description

SqNodeAnalogInput gets the latest sample of the analog input channel on a SynqNet node. The analog value is between +1 and -1.

The RMB-10V, RMB-10V2 and some Trust nodes support analog inputs. MPI support has been added to support the reading of node-based analog inputs. The number of analog inputs a node supports can be determined with [meiSqNodeInfo\(...\)](#). An analog input value can be read with [meiSqNodeAnalogIn\(...\)](#). The analog to digital converted value is scaled from -1.0 to +1.0, where +1.0 is a full-scale positive voltage. The input range of the ADC is hardware-specific.

node	a handle to a SynqNet node object.
channel	the number of the analog input channel on this node.
*state	the magnitude to the analog input.

Return Values	
MPIMessageOK	if <i>SqNodeAnalogInput</i> successfully gets the latest sample of the analog input channel on a SynqNet node.

See Also

meiSqNodeDigitalIn

Declaration

```
long meiSqNodeDigitalIn(MEISqNode node,  
                        long bitCount,  
                        long *state);
```

Required Header

stdmei.h

Description

[SqNodeDigitalIn](#) gets the state of a set of digital inputs on a SynqNet node. The *bitCount* argument defines how may input bits are to be copied into the memory pointed to by the *state* argument. You can find out the maximum number of digital input bits on each node by using the [meiSqNodeInfo](#) function.

For example:

```
long inputs[2];  
meiSqNodeDigitalIn( node0, 32, inputs );
```

node	a handle to a SynqNet node object.
bitCount	how many bits are to be copied.
*state	pointer to where the inputs state is copied.

Return Values	
MPIMessageOK	if <i>SqNodeDigitalIn</i> successfully gets the state of all the digital inputs on a SynqNet node.

See Also

meiSqNodeDigitalInBit

Declaration

long

[meiSqNodeDigitalInBit](#)

[\(MEISqNode](#)

long
long

node,
bit,
***state);**

Required Header

stdmei.h

Description

The [SqNodeDigitalInBit](#) function gets the state of a digital input bit on a SynqNet node.

node	a handle to a SynqNet node object.
bit	the number of the input bit on the node.
*state	The current state of the input on the node. This will either be one or zero.

Return Values

MPIMessageOK

if *SqNodeDigitalInBit* successfully gets the state of a digital input bit on a SynqNet node.

See Also [meiSqNodeDigitalOutBitGet](#) | [meiSqNodeDigitalOutBitSet](#)

meiSqNodeDigitalOutBitGet

Declaration

long meiSqNodeDigitalOutBitGet(MEISqNode
long bit,
long *state);

Required Header

stdmei.h

Description

SqNodeDigitalOutBitGet gets the state of a digital output bit on a SynqNet node.

node	a handle to a SynqNet node object
bit	the number of the output bit on the node.
*state	the current state of the output on the node. This will either be one or zero.

Return Values	
MPIMessageOK	if SqNodeDigitalOutBitGet successfully gets the state of a digital output bit on a SynqNet node.

See Also

[meiSqNodeDigitalOutBitSet](#)

meiSqNodeDigitalOutBitSet

Declaration

long meiSqNodeDigitalOutBitSet([MEISqNode](#)
long
long node,
bit,
state);

Required Header

stdmei.h

Description

[SqNodeDigitalOutBitSet](#) sets the state of a digital output bit on a SynqNet node.

node	a handle to a SynqNet node object
bit	the number of the output bit on the node.
state	the desired state of the output on the node. This should either be one or zero.

Return Values	
MPIMessageOK	if <i>SqNodeDigitalOutBitSet</i> successfully sets the state of a digital output bit on a SynqNet node.

See Also

[meiSqNodeDigitalOutBitGet](#)

meiSqNodeDigitalOutGet

Declaration

```
long meiSqNodeDigitalOutGet( MEISqNode node,
                             long bitCount,
                             long *state);
```

Required Header

stdmei.h

Description

[SqNodeDigitalOutSet](#) gets the current state of a set of digital outputs on a SynqNet node. The *bitCount* argument defines how may output bits are to be copied into the memory pointed to by the *state* argument. You can find out the maximum number of digital output bits on each node using the [meiSqNodeInfo](#) function.

For example:

```
long outputs[2];
meiSqNodeDigitalOutGet( node0, 32, outputs );
```

node	a handle to a SynqNet node object.
bitCount	how many bits are to be copied.
*state	points to where the data is held in the user's program.

Return Values	
MPIMessageOK	if <i>SqNodeDigitalOutGet</i> successfully gets the state of all the digital outputs on a SynqNet node.

See Also

[meiSqNodeDigitalOutSet](#)

meiSqNodeDigitalOutSet

Declaration

```
long meiSqNodeDigitalOutSet( MEISqNode node,
                             long bitCount,
                             long *state);
```

Required Header

stdmei.h

Description

[SqNodeDigitalOutSet](#) sets a set of digital outputs on a SynqNet node. The *bitCount* argument defines how may output bits are to be set. The memory holding the new output states are pointed to by the *state* argument. You can find out the maximum number of digital output bits on each node using the [meiSqNodeInfo](#) function.

For example:

```
long outputs[2];
outputs[0] = 0x0100;
outputs[1] = 0x0080;
meiSqNodeDigitalOutSet( node0, 32, outputs );
```

node	a handle to a SynqNet node object.
bitCount	how many bits are to be set.
*state	pointer to where the desired output bits are held.

Return Values	
MPIMessageOK	if <i>SqNodeDigitalOutSet</i> successfully sets the state of all the digital outputs on a SynqNet node.

See Also

[meiSqNodeDigitalOutGet](#)

Required Header

```
stdmei.h
```

The binary download process requires a significant amount of time, probably between 5 to 30 seconds, depending on the node/drive type and file size. A callback function pointer is provided in the MEISqNodeDownloadParams structure for the calling application to monitor the download progress.

node	a handle to a SynqNet node object
*params	a pointer to the download parameters structure.

Return Values

MPIMessageOK	if <i>SqNodeNumber</i> successfully reads the network number

See Also [meiSqNodeInfo](#) | [meiSynqNetInfo](#) | [MEISqNodeDownloadParams](#) | [MEISqNodeChannel](#) | [MEISqNodeCallback](#)

meiSqNodeFlashErase

Declaration long **meiSqNodeFlashErase**([MEISqNode](#) **sqNode**);

Required Header stdmei.h

Description **SqNodeFlashErase** brings the SynqNet network down to discovery mode, sends a service command down to the node that erases its runtime flash, and leaves the network down. The next time the network is brought up to Synq mode the node will be running off its boot image.

node	a handle to a SynqNet node object

Return Values

MPIMessageOK	if <i>SqNodeFlashErase</i> successfully brings the SynqNet network down to discovery mode, sends a service command down to the node that erases its runtime flash, and leaves the network down.

See Also

meiSqNodeFpgaFileNameVerify

Declaration

```
long meiSqNodeFpgaFileNameVerify(MEISqNode sqNode, char* fileName);
```

Required Header

stdmei.h

Description

[meiSqNodeFpgaFileNameVerify](#) verifies that the filename provided is compatible with a given sqNode.

sqNode	a handle to an SqNode object.
fileName	a pointer to a string containing the name of an SqNode image file.

Return Values

MPIMessageOK	If the FPGA image file is compatible with the specified SqNode.
MEISqNodeMessageFILE_NODE_MISMATCH	If the FPGA image file is NOT compatible with the specified SqNode.

See Also

meiSqNodeVerify

Declaration long **meiSqNodeVerify**([MEISqNode](#) node, [MEISqNodeDownloadParams](#) *params);

Required Header

Function	Description
SqNodeVerify	verifies that the runtime image on a sqNode matches the data contained in a provided image file.

node	a handle to SqNode object.
*params	a pointer to parameters used in the verify routine.

Return Values

MPIMessageOK	If the <i>meisqNodeVerify</i> successfully verifies that there is a matching runtime image.
MEISqNodeMessageVERIFY_FAIL	If the file provided does not match the runtime image on the node.

See Also

meiSqNodeEventNotifyGet

Declaration

long **meiSqNodeEventNotifyGet**([MEISqNode](#)
 [MPIEventMask](#)
 void **node**,
 ***eventMask**,
 ***external**);

Required Header

stdmei.h

Description

SqNodeEventNotifyGet reads the event mask (that specifies the event types 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).

Use the event mask macros `mpiEventMaskGET(...)`, `mpiEventMaskBitGET(...)`, etc. to decode the eventMask.

The event notification data in external is in addition to the event notification data in eventMask. If external is NULL, the event notification data will not be copied to the external pointer.

XMP Only

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

node	a handle to a SynqNet node object
*eventMask	pointer to an event mask, whose bits are defined by the MPI/MEIEventType enumerations.
*external	pointer to external

Return Values

MPIMessageOK	if <i>SqNodeEventNotifyGet</i> successfully reads the event mask
MPIMessageARG_INVALID	if the eventMask pointer is NULL

See Also

[MPI/MEIEventType](#) | [MEIEventNotifyData](#) | [MEIEventStatusInfo](#)

Required Header `stdmei.h`

Use the event mask macros `meiEventMaskSQNODE(...)`, `mpiEventMaskSET(...)`, `mpiEventMaskBitSET(...)`, `mpiEventMaskCLEAR(...)`, etc. to create the `eventMask`.

The event notification data in `external` is in addition to the event notification data in `eventMask`. If `external` is `NULL`, the event notification data will not be copied to the `external` pointer.

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

node	a handle to a SynqNet node object
eventMask	pointer to an event mask, whose bits are defined by the MPI/MEIEventType enumerations.
*external	pointer to external

MPIMessageOK	if <i>SynqNetEventNotifySet</i> successfully writes the event mask
MPIMessageARG_INVALID	if the eventMask pointer is NULL

file:///D:/pdfs/030100/html/Software-MPI/docs/sqNode/Method/evtnfyset2.htm [7/26/2004 9:12:13 AM]

Required Header

```
stdmei.h
```

See [MEIEventType](#).

sqNode	a handle to a SynqNet node object
eventMask	pointer to an event mask, whose bits are defined by the MPI/MEIEventType enumerations.

MPIMessageOK	if <i>SynqNetEventReset</i> successfully writes the event mask
MPIMessageARG_INVALID	if the eventMask pointer is NULL

file:///D:/pdfs/030100/html/Software-MPI/docs/sqNode/Method/evtrst2.htm [7/26/2004 9:12:13 AM]

meiSqNodeMemory

```
Declaration      long  meiSqNodeMemory(MEISqNode    node,  
                  void      **memory);
```

Required Header

Description	SqNodeMemory writes an address (that can be used to access SqNode memory) to the contents of memory. This address (or an address calculated from it) can be passed as the src argument to <code>mpiSqNodeMemoryGet(...)</code> or the dst argument to <code>mpiSqNodeMemorySet(...)</code>.

node	a handle to a SynqNet node object
**memory	a pointer to an SqNode memory address.

Return Values

MPIMessageOK	if <i>SqNodeMemory</i> successfully writes the SqNode memory address to the contents of memory.

See Also [meiSqNodeMemoryGet](#) | [meiSqNodeMemorySet](#)

meiSqNodeMemoryGet

Declaration

long meiSqNodeMemoryGet (MEISqNode node,
 void *dst,
 void *src,
 long count);

Required Header

stdmei.h

Description

SqNodeMemoryGet reads count bytes of an SqNode's memory, starting from address *src* and writes it to application memory, starting at address *dst*.

node	a handle to a SynqNet node object
*dst	pointer to the destination address in application memory
*src	pointer to the source address in SqNode memory
count	number of bytes to copy

Return Values	
MPIMessageOK	if <i>SqNodeMemoryGet</i> successfully copies data from SqNode memory to application memory.

See Also

[meiSqNodeMemory](#) | [meiSqNodeMemorySet](#)

meiSqNodeMemorySet

```

Declaration
long meiSqNodeMemorySet(MEISqNode node,
void *dst,
void *src,
long count);

```

Required Header `stdmei.h`

Description	SqNodeMemorySet reads count bytes of application memory, starting from address <i>src</i> and writes it to an SqNode's memory, starting at address <i>dst</i> .

node	a handle to a SynqNet node object
*dst	pointer to the destination address in SqNode memory
*src	pointer to the source address in application memory
count	number of bytes to copy

Return Values

MPIMessageOK	if <i>SqNodeMemorySet</i> successfully copies data from application memory to <i>SqNode</i> memory.

See Also [meiSqNodeMemory](#) | [meiSqNodeMemoryGet](#)

meiSqNodeControl

Declaration `meiSqNodeControl(MEISqNode node);`

Required Header `stdmei.h`

Description [SqNodeControl](#) returns a handle to the control object associated with the SqNode object.

node	a handle to a SynqNet node object

Return Values

MPIControl	a handle to a control object
MPIHandleVOID	if node is not valid

See Also [meiSqNodeCreate](#) | [mpiControlCreate](#)

meiSqNodeNumber

```
long meiSqNodeNumber(MEISqNode node,  
                    long *number);
```

Required Header

Description	SqNodeNumber reads the index of a SynqNet <i>node</i> and writes it into the contents of a long pointed to by <i>number</i> . Each SqNode associated with a controller is indexed by a identification number (0, 1, 2, etc.).

node	a handle to a SynqNet node object
*number	a pointer to the index of a SynqNet node.

Return Values

MPIMessageOK	if <i>SqNodeNumber</i> successfully reads the network number

See Also [meiSynqNetInfo](#) | [meiSynqNetNumber](#)

MEISqNodeCallback

MEISqNodeCallback

```
typedef long (*MEISqNodeCallback)(long percentage);
```

Description

SqNodeCallback is a pointer to a function, which can be used to monitor the `meiSqNodeDownload(...)` progress.

percentage	The portion (from 0% to 100%) of memory download that has been completed.

See Also

[meiSqNodeDownload](#)

MEISqNodeChannel

MEISqNodeChannel

```
typedef enum MEISqNodeChannel {
    MEISqNodeChannelDRIVE0,
    MEISqNodeChannelDRIVE1,
    MEISqNodeChannelDRIVE2,
    MEISqNodeChannelDRIVE3,
    MEISqNodeChannelDRIVE4,
    MEISqNodeChannelDRIVE5,
    MEISqNodeChannelDRIVE6,
    MEISqNodeChannelDRIVE7,
    MEISqNodeChannelNODE,
} MEISqNodeChannel;
```

Description

SqNodeChannel is an enumeration of communication interfaces to a node. All SynqNet nodes support a single NODE channel to the network interface device. SynqNet nodes may support one or more drive channels to a drive processor. DRIVE channels are indexed by an enumeration (DRIVE0, DRIVE1, DRIVE2, etc.).

MEISqNodeChannelDRIVE0	interface to drive number 0
MEISqNodeChannelDRIVE1	interface to drive number 1
MEISqNodeChannelDRIVE2	interface to drive number 2
MEISqNodeChannelDRIVE3	interface to drive number 3
MEISqNodeChannelDRIVE4	interface to drive number 4
MEISqNodeChannelDRIVE5	interface to drive number 5
MEISqNodeChannelDRIVE6	interface to drive number 6
MEISqNodeChannelDRIVE7	interface to drive number 7
MEISqNodeChannelNODE	interface to the node device

See Also [meiSqNodeCommand](#) | [meiSqNodeDownload](#)

MEISqNodeCmdType

MEISqNodeCmdType

```
typedef enum MEISqNodeCmdType {  
    MEISqNodeCmdTypeREAD,  
    MEISqNodeCmdTypeWRITE,  
} MEISqNodeCmdType;
```

Description [SqNodeCmdType](#) is an enumeration of service command types to send to a node or drive.

MEISqNodeCmdTypeREAD	read data
MEISqNodeCmdTypeWRITE	write data

See Also [meiSqNodeCommand](#) | [MEISqNodeCmdHeader](#)

MEISqNodeCmdHeader

MEISqNodeCmdHeader

```
typedef struct MEISqNodeCmdHeader {
    MEISqNodeChannel    channel;    /* internal node destination */
    MEISqNodeMemory    memory;
    MEISqNodeDataSize  size;
    MEISqNodeCmdType   type;        /* read/write command */
} MEISqNodeCmdHeader;
```

Description

The **SqNodeCmdHeader** structure specifies the service command communication interface to the device, the memory region on the device to access, the data size, and type.

channel	Communication interface to a device. See MEISqNodeChannel .
memory	The memory region to access. See MEISqNodeMemory .
size	The length of data to send or receive. See MEISqNodeDataSize .
type	The service command action (read or write). See MEISqNodeCmdType .

See Also [meiSqNodeCommand](#) | [MEISqNodeCommand](#)

MEISqNodeCommand

MEISqNodeCommand

```
typedef struct MEISqNodeCommand {
    MEISqNodeCmdHeader header;
    unsigned long address; /* command resgisters */
    unsigned long data; /* command data */
} MEISqNodeCommand;
```

Description

The **SqNodeCommand** structure specifies the service command. It includes a header structure (channel, memory, size, and type), a destination address, and the data.

header	A structure that specifies the channel, memory region, and data size. See MEISqNodeCmdHeader .
address	A memory location to read or write the data.
data	The command data to send.

See Also [meiSqNodeCommand](#) | [MEISqNodeResponse](#)

MEISqNodeConfig

MEISqNodeConfig

```
typedef struct MEISqNodeConfig {
    MEISqNodeConfigAlarm           nodeAlarm;
    MEISqNodeConfigIoAbort        ioAbort;
    MEISqNodeConfigPacketError    upStreamError;
    MEISqNodeConfigPacketError    downStreamError;
    MEISqNodeConfigUserFault      userFault;
    MEISqNodeFeedbackSecondary    feedbackSecondary
                                [MEISqNodeMaxFEEDBACK\_SECONDARY];
} MEISqNodeConfig;
```

Description The **SqNodeConfig** structure specifies the SynqNet node configurations.

nodeAlarm	A structure to configure a SynqNet node's trigger conditions for the Node Alarm output bit. The node alarm circuit is node specific, but is intended to notify users when the node has a problem. The nodeAlarm occurs on an ioAbort, DedicatedInAMP_FAULT (one per motor/drive) or an FPGA fails to operate with run-time code. See MEISqNodeConfigIoAbort and MEISqNodeConfigNodeAlarm for the trigger configurations.
ioAbort	A structure to configure a SynqNet node's trigger conditions for an I/O Abort action. When an ioAbort is triggered, the SynqNet node's outputs are disabled (set to the power-on condition). See MEISqNodeConfigIoAbort for the trigger configurations.
upStreamError	A structure used to configure the fault and failure limits for the upstream SynqNet packets. The controller keeps track of how many bad packets are received from the Node and performs the appropriate actions when the fault and fail limits are reached. See MEISqNodeConfigPacketError for appropriate ranges and resulting actions.
downStreamError	A structure used to configure the fault and failure limits for the downstream SynqNet packets. The node keeps track of how many bad packets are received from the controller and performs the appropriate actions when the fault and fail limits are reached. See MEISqNodeConfigPacketError for appropriate ranges and resulting actions.
userFault	A structure to configure the trigger conditions for a SynqNet node user fault. When a user fault is triggered, a node ioAbort and/or an action on each motor will occur. See MEISqNodeConfigUserFault for the trigger configurations.

feedbackSecondary

A structure to configure the secondary encoder resources on the node. See [MEISqNodeFeedbackSecondary](#) for more information.

See Also

[meiSqNodeConfigGet](#) | [meiSqNodeConfigSet](#) | [MEISqNodeConfigPacketError](#)

MEISqNodeConfigIoAbort

MEISqNodeConfigIoAbort

```
typedef struct MEISqNodeConfigIoAbort {
    MEISqNodeConfigTrigger    synqLost;        /* communication error */
    MEISqNodeConfigTrigger    nodeDisable;     /* external input */
    MEISqNodeConfigTrigger    powerFault;      /* analog power failure */
    long                      userFault;       /* TRUE = user fault causes ioabort */
} MEISqNodeConfigIoAbort;
```

Description

The **SqNodeConfigIoAbort** structure specifies the SynqNet node configurations to generate an I/O Abort action. When an ioAbort is triggered, the SynqNet node's outputs are disabled (set to the power-on condition). The IoAbort is triggered when any one or more of the following enabled configurations occur:

synqLost	Occurs when a SynqNet node drops out of SYNQ (cyclic) mode to SYNQ_LOST mode. See MEISqNodeConfigTrigger .
nodeDisable	An input bit to the SynqNet node. The node disable circuit is node specific, but is intended to shutdown the node via the IoAbort. See MEISqNodeConfigTrigger .
powerFault	An input bit to the SynqNet node. The power fault circuit is node specific, but is usually connected to an analog power monitor. Typically, when the DAC power or other analog component power is either too high or drops below a threshold, the power fault is triggered. Please see the node/drive manufacturer's documentation for details. See MEISqNodeConfigTrigger .
userFault	A user configurable trigger condition. A value of TRUE enables the trigger, FALSE disables the trigger.

See Also

[meiSqNodeConfigGet](#) | [meiSqNodeConfigSet](#)

MEISqNodeConfigAlarm

MEISqNodeConfigAlarm

```
typedef struct MEISqNodeConfigAlarm {
    unsigned long    mask;                /* One bit per drive/motor. Triggered by
                                           the MEIMotorDedicatedInAMP_FAULT input. */
    long             notCyclicEnable;     /* allow nodeAlarm to be asserted when the
node                                     is not in cyclic mode */
    long             ioAbortEnable;       /* allow ioAbort to assert nodeAlarm */
} MEISqNodeConfigAlarm;
```

Description

The [SqNodeConfigAlarm](#) structure specifies the input trigger for the SynqNet node alarm output. The input triggers are the MEIMotorDedicatedInAMP_FAULT bits for each motor/drive interface.

mask	Each bit in the mask represents a motor or drive interface. For example, a value of 0x3 will trigger the node alarm output when either motor 0's OR motor 1's MEIMotorDedicatedInAMP_FAULT bit is TRUE.
notCyclicEnable	This Boolean variable is used to specify whether or not a node can receive an alarm when it is not in cyclic mode. TRUE = node alarm can be asserted in any mode. FALSE = node alarm can only be asserted in cyclic mode.
ioAbortEnable	This Boolean variable is used to specify the effect an I/O abort will have on the node alarm output. TRUE = an I/O abort will trigger a node alarm. FALSE = an I/O abort will not necessarily trigger a node alarm.

See Also

[meiSqNodeConfigGet](#) | [meiSqNodeConfigSet](#) | [MEIMotorDedicatedIn](#)

MEISqNodeConfigPacketError

MEISqNodeConfigPacketError

```
typedef struct MEISqNodeConfigPacketError {
    long faultLimit;    /* 0 - 255 */
    long failLimit;    /* 0 - 255 */
} MEISqNodeConfigPacketError;
```

Description The **SqNodeConfigPacketError** structure specifies the limit conditions for SynqNet node packet rate errors.

faultLimit	Packet error rate limit to generate a fault. When the faultLimit is reached, the node will attempt to recover by switching the port used for data transmission. Valid range is 0 to 255. The value saturates at 255.
failLimit	Packet error rate limit to generate a failure. When the failLimit is reached, the node will drop to the SYNQ_LOST state and disable its outputs. Valid range is 0 to 255. The value saturates at 255.

See Also [meiSqNodeConfigGet](#) | [meiSqNodeConfigSet](#)

MEISqNodeConfigTrigger

MEISqNodeConfigTrigger

```
typedef struct MEISqNodeConfigTrigger {
    long    enable;
    long    invert;
} MEISqNodeConfigTrigger;
```

Description

The [SqNodeConfigTrigger](#) structure specifies trigger configurations.

enable	Enables or disables the trigger. A value of TRUE enables the trigger, FALSE disables the trigger.
invert	Normal or inverted trigger polarity. A value of FALSE indicates normal polarity, TRUE indicates inverted polarity.

See Also

[MEISqNodeConfigIoAbort](#)

MEISqNodeConfigUserFault

MEISqNodeConfigUserFault

```
typedef struct MEISqNodeConfigUserFault {
    long          *addr;      /* firmware addr */
    unsigned long  mask;
    unsigned long  pattern;
} MEISqNodeConfigUserFault;
```

Description

The [SqNodeConfigUserFault](#) structure specifies the trigger conditions for a user defined input. The trigger condition can be configured for any controller address. When the masked value at the specified addr matches the pattern, the user fault is active. The user fault triggers a SynqNet node IoAbort if the userFault flag in MEISqNodeConfigIoAbort{...} is enabled. The user fault also triggers an action for all the motors associated with the node. The userFaultAction is specified in the MEIMotorConfig{...} structure.

*addr	A pointer to a controller address.
mask	A bit mask ANDed with the value at the controller address.
pattern	A bit pattern compared to the masked value at the controller address. When the masked value equals the pattern, the user trigger is TRUE.

See Also

[MEISqNodeConfigIoAbort](#) | [MEIMotorConfig](#) | [meiSqNodeConfigGet](#) | [meiSqNodeConfigSet](#)

MEISqNodeDataSize

MEISqNodeDataSize

```
typedef enum MEISqNodeDataSize { /* read/write data width */
    MEISqNodeDataSize8BIT,
    MEISqNodeDataSize16BIT,
    MEISqNodeDataSize24BIT,
    MEISqNodeDataSize32BIT,
} MEISqNodeDataSize
```

Description **SqNodeDataSize** is an enumeration of service command data lengths. The data length is in units of bits.

MEISqNodeDataSize8BIT	8 bit data length
MEISqNodeDataSize16BIT	16 bit data length
MEISqNodeDataSize24BIT	24 bit data length
MEISqNodeDataSize32BIT	32 bit data length

See Also [meiSqNodeCommand](#) | [MEISqNodeCmdHeader](#)

MEISqNodeDownloadParams

MEISqNodeDownloadParams

```
typedef struct MEISqNodeDownloadParams {
    char                *filename;
    MEISqNodeChannel    channel;
    MEISqNodeCallback   callback;
} MEISqNodeDownloadParams;
```

Description

The **SqNodeDownloadParams** structure specifies the parameters for downloading a binary image to a SynqNet node.

*filename	A pointer to a file name. The file contains a header and binary code/data. Files are node/drive specific. Please see the Node Binary Files: Product Table or the drive manufacturer's documentation for the drive binary files.
channel	A communication interface to a node's logic device or drive processor. See MEISqNodeChannel .
callback	A pointer to a callback function, to monitor the download progress. See MEISqNodeCallback .

See Also

[meiSqNodeDownload](#)

MEISqNodeDriveInfo

MEISqNodeDriveInfo

```
typedef struct MEISqNodeDriveInfo {  
    char    firmwareVersion[MEISqNodeDriveParamMAX_STRING_LENGTH];  
} MEISqNodeDriveInfo;
```

Description **SqNodeDriveInfo** is a structure containing information about a specified drive.

firmwareVersion	A string containing drive firmware version information that is retrieved from the Drive Processor on the Node.

See Also

MEISqNodeDriveMonitor

MEISqNodeDriveMonitor

```
typedef struct MEISqNodeDriveMonitor {
    MEISqNodeDriveMonitorDataType type;
    MEISqNodeDriveMonitorData data;
} MEISqNodeDriveMonitor;
```

Description The **SqNodeDriveMonitor** structure specifies the data to be placed in the monitor field by the drive.

type	The drive data is selected by its type. See MEISqNodeDriveMonitorDataType .
data	The location of the drive data. See MEISqNodeDriveMonitorData .

See Also [MEISqNodeMonitorValue](#) | [meiSqNodeDriveMonitorConfigGet](#) | [meiSqNodeDriveMonitorConfigSet](#)

MEISqNodeDriveMonitorConfig

MEISqNodeDriveMonitorConfig

```
typedef struct MEISqNodeDriveMonitorConfig {
    MEISqNodeDriveMonitor    monitorA;
    MEISqNodeDriveMonitor    monitorB;
    MEISqNodeDriveMonitor    monitorC;
}MEISqNodeDriveMonitorConfig;
```

Description The [SqNodeDriveMonitorConfig](#) structure specifies the configuration for the drive monitor fields.

monitorA	configuration for drive monitor A
monitorB	configuration for drive monitor B
monitorC	configuration for drive monitor C

See Also [MEISqNodeDriveMonitor](#) | [meiSqNodeDriveMonitorConfigGet](#) | [meiSqNodeDriveMonitorConfigSet](#) |

MEISqNodeDriveMonitorData

MEISqNodeDriveMonitorData

```
typedef union {
    long    index;      /* the values for these parameters are drive specific    */
    long    address;    /* and can be found in the appropriate drive modules    */
} MEISqNodeDriveMonitorData;
```

Description **SqNodeDriveMonitorData** specifies the location of the monitor data. Drive data can be specified by either an index or an address. The location is drive specific. Please see the drive manufacturer's documentation.

index	A drive specific value to select a monitor data field from a table.
address	A drive specific memory address to select the monitor data.

See Also [meiSqNodeDriveMonitorConfigGet](#) | [meiSqNodeDriveMonitorConfigSet](#)

MEISqNodeDriveMonitorDataType

MEISqNodeDriveMonitorDataType

```
typedef enum MEISqNodeDriveMonitorDataType {  
    MEISqNodeDriveMonitorDataTypeINDEX,  
    MEISqNodeDriveMonitorDataTypeADDRESS,  
} MEISqNodeDriveMonitorDataType;
```

Description [SqNodeDriveMonitorDataType](#) is an enumeration of monitor data selection types.

MEISqNodeDriveMonitorDataTypeINDEX	Select monitor data using an index to a table.
MEISqNodeDriveMonitorDataTypeADDRESS	Select monitor data using an address.

See Also [meiSqNodeDriveMonitorConfigGet](#) | [meiSqNodeDriveMonitorConfigSet](#)

MEISqNodeDriveParamCallback

MEISqNodeDriveParamCallback

```
typedef void (*MEISqNodeDriveParamCallback)
(MEISqNodeDriveParamCallbackType type,
char *name,
long number,
char *value);
```

Description

In the [SqNodeDriveParamCallback](#) structure, the function's pointer type defines a function that can be passed to the `meiSqNodeDriveParamFileSet` function. The `meiSqNodeDriveParamFileSet` function will call this type of function to report progress or warnings. A NULL value for the callback pointer will disable the callback feature.

type	the type of event that caused the callback function to be called.
name	name of the drive parameter.
number	drive parameter index.
value	the value of the drive parameter.

See Also [meiSqNodeDriveParamFileSet](#)

MEISqNodeDriveParamCallbackType

MEISqNodeDriveParamCallbackType

```
typedef enum {
    MEISqNodeDriveParamCallbackTypeCHANGED,
    MEISqNodeDriveParamCallbackTypeSET_FAILED,
} MEISqNodeDriveParamCallbackType;
```

Description

The [SqNodeDriveParamCallbackType](#) enumeration is used by the MEISqNodeDriveParamCallback function to describe the type of event that caused the callback function to be called.

MEISqNodeDriveParamCallbackTypeCHANGED	This indicates that the new drive parameter value is different to the current parameter value.
MEISqNodeDriveParamCallbackTypeSET_FAILED	Setting this drive parameter failed.

See Also

[MEISqNodeDriveParamCallback](#)

MEISqNodeFeedbackSecondary

MEISqNodeFeedbackSecondary

```
typedef struct MEISqNodeFeedbackSecondary {  
    long    motorIndex;  
} MEISqNodeFeedbackSecondary;
```

Description

The **SqNodeFeedbackSecondary** structure allows for configuration of the secondary feedback resources on a SynqNet node.

motorIndex	Indicates motorIndex on the node to which the secondary feedback resource is mapped. This value is MEISqNodeNOT_AVAILABLE if the secondary feedback resource does not exist on the node hardware

See Also

[MEISqNodeConfig](#)

MEISqNodeFileName

MEISqNodeFileName

```
typedef struct MEISqNodeFileName{  
    char fileName[MEISqNodeFILENAME\_MAX];  
}MEISqNodeFileName;
```

Description **FileName** is used in methods that retrieve filenames from the MPI.

fileName	String containing the name of an SqNode image file.

See Also

MEISqNodeFpgaType

MEISqNodeFpgaType

```
typedef enum MEISqNodeFpgaType {  
    MEISqNodeFpgaTypeBOOT,  
    MEISqNodeFpgaTypeRUN_TIME,  
} MEISqNodeFpgaType
```

Description `sqNodeFpgaType` is an enumeration of FPGA types.

MEISqNodeFpgaTypeBOOT	The FPGA is operating with a boot image. The boot image only supports basic SynqNet communication. Use <code>meiSqNodeDownload(...)</code> to download the runtime image to the SynqNet node.
MEISqNodeFpgaTypeRUN_TIME	The FPGA is operating with a runtime image.

See Also [meiSqNodeInfo](#) | [MEISqNodeInfoFpga](#) | [meiSqNodeDownload](#)

MEISqNodeInfo

MEISqNodeInfo

```
typedef struct MEISqNodeInfo {
    long          motorCount;
    long          driveCount;
    long          motorOffset;
    long          feedbackSecondaryCount;
    MEISqNodeInfoId id;
    MEISqNodeInfoFpga fpga;
    MEISqNodeInfoNetwork network;
    MEISqNodeInfoIo io;
} MEISqNodeInfo;
```

Description

The [SqNodeInfo](#) structure contains static data stored for the SynqNet node. The motor objects are indexed sequentially across all the SynqNet nodes associated with each network. Each motor on a controller has a unique number.

motorCount	The number of motors that the SynqNet node supports.
driveCount	The number of drives interfaces that the SynqNet node supports.
motorOffset	The starting number for the first motor on the SynqNet node.
feedbackSecondaryCount	The number of auxillary feedbacks on the node.
id	A structure that contains identification data for the SynqNet node. See MEISqNodeInfoId .
fpga	A structure that contains identification data for the SynqNet node FPGA. See MEISqNodeInfoFpga .
network	A structure that contains network interface information for the SynqNet node. See MEISqNodeInfoNetwork .
io	A structure that returns how many of each type of node I/O this node supports.

See Also

[meiSqNodeInfo](#)

MEISqNodeInfoId

MEISqNodeInfoId

```
typedef struct MEISqNodeInfoId {
    unsigned long    nodeType;    /* product/mfg code */
    char             *nodeName;    /* product/mfg string */
    unsigned long    option;        /* product option code*/
    unsigned long    switchId;      /* rotary switch id */
    unsigned long    unique;         /* unique id code */

    long             exactMatch; /* TRUE/FALSE */

    char             serialNumber[MEISqNodeID\_CHAR\_MAX];
    char             modelName[MEISqNodeID\_CHAR\_MAX];
    char             manufacturerData[MEISqNodeManufacturerDATA\_CHAR\_MAX];
} MEISqNodeInfoId;
```

Description

The [SqNodeInfoId](#) structure contains identification data for the SynqNet node.

All nodes by all manufacturers will have **nodeType** and **unique** numbers that should generate a unique identification for each node on the SynqNet network.. Although some node manufacturers may opt to leave the **serialNumber** and **modelName** fields blank, you can still identify and distinguish a node by comparing the **nodeType** and **unique** numbers. The **nodeType** number is also represented by a unique text string *nodeName*.

nodeType	A 32 bit value that identifies the node hardware. The upper 16 bits represent the manufacturer of the SynqNet node hardware. Each manufacturer has a unique value. The lower 16 bits represent the SynqNet node product type. The SynqNet node manufacturer determines a unique value to track a product series. Typically, the node type value is displayed in hex.
*nodeName	A string that represents the SynqNet nodeType. The nodeName string matches the name of the SqNodeLib node specific header file.
option	The product option code within a product series.
switchId	If a node/drive have an physical address switch on its faceplate, switchId will contain the value to which the switch is set. If an ID switch is not supported by a node, this value will be set to -1 (0xFFFFFFFF).

MEISqNodeInfoIo

MEISqNodeInfoIo

```
typedef struct MEISqNodeInfoIo {
    long    digitalInCount;
    long    digitalOutCount;
    long    analogInCount;
    long    analogOutCount;
} MEISqNodeInfoIo;
```

Description

The [SqNodeInfoIo](#) structure lists the number of digital and analog inputs that are supported by a SynqNet node.

digitalInCount	The number of digital inputs on a SynqNet node.
digitalOutCount	The number of digital outputs on a SynqNet node.
analogInCount	The number of analog inputs on a SynqNet node.
analogOutCount	The number of analog outputs on a SynqNet node.

See Also

[meiSqNodeInfo](#)

MEISqNodeInfoFpga

MEISqNodeInfoFpga

```
typedef struct MEISqNodeInfoFpga {
    MEISqNodeFpgaType    type;
    unsigned long        vendorDevice;
    unsigned long        version;
    long                 defaultVersion; /* TRUE/FALSE */
} MEISqNodeInfoFpga;
```

Description The [SqNodeInfoFpga](#) structure contains identification data for the SynqNet node FPGA.

type	The FPGA type. See MEISqNodeFpgaType .
vendorDevice	A 32 bit value that identifies the FPGA image. The upper 16 bits represent the manufacturer of the SynqNet node network interface device. Each manufacturer has a unique vendor value. The lower 16 bits represent the SynqNet node network interface component. The device is typically an FPGA (could be an ASIC). If the device is an FPGA, the vendorDevice information is stored in the FPGA binary image. Each device for a particular vendor has a unique device value. Typically, the vendorDevice value is displayed in hexadecimal format.
version	A 32 bit value that represents the revision of the device. The upper 16 bits represent the SynqNet network interface revision. The lower 16 bits represent the device revision. Typically, the version value is displayed in hexadecimal format.
defaultVersion	Indicates if the default version of the SqNode FPGA image is loaded on this node. The defaultVersion defines the version of the SynqNet node FPGA image that was built and tested with the current version of the MPI.

See Also [meiSqNodeInfo](#) | [MEISqNodeInfoId](#) | [MPI/SynqNet FPGA Compatibility Check](#)

unique	A 32 bit value that identifies the node. It is an unsigned long. The SynqNet node manufacturer determines this unique value to track a single product. This is useful to determine when individual nodes of the same type are switched or replaced on a SynqNet network. NOTE: It is possible for a manufacturer to use the same unique identification number for two nodes of different models. The combination of SqNode.Name (or nodeType) and SqNode.UniqueId will be unique for any given code.
exactMatch	A string that tells you if the node is running under a matched or unmatched classification. The value of meiSqNodeInfo.id.exactMatch is TRUE when all ID components have been matched to a supported configuration. The value is FALSE when running with a default (unmatched) configuration.
serialNumber	A string that represents the SynqNet node serial number. For a given node type, the serial number is unique. The SynqNet node manufacturer determines the serial number to track an individual unit.
modelNumber	A string that represents the SynqNet node model number. The SynqNet node manufacturer determines the model number.
manufacturerData	A string containing Manufacturer-specific data which is stored on the node at time of production.

See Also [meiSqNodeInfo](#) | [MEISqNodeInfoFpga](#)

MEISqNodeInfoNetwork

MEISqNodeInfoNetwork

```
typedef struct MEISqNodeInfoNetwork {
    long    number;
    long    inPorts;
    long    outPorts;
} MEISqNodeInfoNetwork;
```

Description

The [SqNodeInfoNetwork](#) structure contains information about the SynqNet node's network interface.

number	An index to a SynqNet network associated with a controller.
inPorts	The number of SynqNet IN port network interfaces.
outPorts	The number of SynqNet OUT port network interfaces.

Remarks

The labeling convention for IN and OUT ports is for convenience. The hardware ports are identical. During SynqNet initialization, the node are discovered based on the OUT to IN port connections.

See Also

[meiSqNodeInfo](#)

MEISqNodeMemory

MEISqNodeMemory

```
typedef enum MEISqNodeMemory {
    MEISqNodeMemoryDATA,      /* node/drive processor RAM */
    MEISqNodeMemoryPROGRAM,   /* drive processor program memory */
    MEISqNodeMemoryIO,        /* drive I/O memory */
    MEISqNodeMemoryDRIVE,     /* direct command to drive */
} MEISqNodeMemory;
```

Description **SqNodeMemory** is an enumeration of drive region types to access with a service command.

MEISqNodeMemoryDATA	node/drive processor data memory
MEISqNodeMemoryPROGRAM	drive processor program memory
MEISqNodeMemoryIO	drive I/O memory
MEISqNodeMemoryDRIVE	direct command to drive processor

See Also [MEISqNodeCmdHeader](#) | [meiSqNodeCommand](#)

MEISqNodeMessage

MEISqNodeMessage

```
typedef enum {
    MEISqNodeMessageINVALID,
    MEISqNodeMessageCONFIG_NETWORK_MISMATCH,
    MEISqNodeMessageMAP_CONFIG_MISMATCH,
    MEISqNodeMessageNOT_IN_CONFIG_FILE,
    MEISqNodeMessageCONFIG_FILE_FORMAT_INVALID,

    MEISqNodeMessageRESPONSE_TIMEOUT,
    MEISqNodeMessageREADY_TIMEOUT,
    MEISqNodeMessageSRVC_ERROR,
    MEISqNodeMessageSRVC_UNSUPPORTED,
    MEISqNodeMessageSRVC_CHANNEL_INVALID,

    MEISqNodeMessageCMD_NOT_SUPPORTED,
    MEISqNodeMessageDISCOVERY_FAILURE,
    MEISqNodeMessageDISPATCH_ERROR,
    MEISqNodeMessageINIT_FAILURE,
    MEISqNodeMessageINTERFACE_ERROR1,
    MEISqNodeMessageFILE_NODE_MISMATCH,
    MEISqNodeMessageFILE_INVALID,
    MEISqNodeMessageINVALID_HEADER,
    MEISqNodeMessageDOWNLOAD_FAIL,
    MEISqNodeMessageVERIFY_FAIL,
    MEISqNodeMessageDOWNLOAD_NOT_SUPPORTED,
    MEISqNodeMessageVERIFY_NOT_SUPPORTED,
    MEISqNodeMessageBOOT_ROM_INVALID,
    MEISqNodeMessageINVALID_TABLE,
    MEISqNodeMessageINVALID_STR_LEN,
    MEISqNodeMessageFEEDBACK_MAP_INVALID,
} MEISqNodeMessage;
```

Description

SqNodeMessage is an enumeration of SynqNet node error messages that can be returned by the MPI library.

MEISqNodeMessageINVALID

The SqNode type is out of range. This message code is returned by SynqNet node methods if the node type is not a member of the SQNodeLibNodeType enumeration.

MEISqNodeMessageCONFIG_NETWORK_MISMATCH

The type of map file specified in meiSqNodeDriveParamFileSet does not match the type of drive found on the network.

MEISqNodeMessageMAP_CONFIG_MISMATCH

The parameter name or number specified in meiSqNodeDriveMapParamFileSet was not valid for the specified drive.

MEISqNodeMessageNOT_IN_CONFIG_FILE

The parameter name or number specified in meiSqNodeDriveMapParamFileSet was not found.

MEISqNodeMessageCONFIG_FILE_FORMAT_INVALID

A file with an incorrect format was used in meiSqNodeDriveMapParamFileSet.

MEISqNodeMessageRESPONSE_TIMEOUT

Currently unused and is reserved for future use.

MEISqNodeMessageREADY_TIMEOUT

Currently unused and is reserved for future use.

MEISqNodeMessageSRVC_ERROR

Currently unused and is reserved for future use.

MEISqNodeMessageSRVC_UNSUPPORTED

Currently unused and is reserved for future use.

MEISqNodeMessageSRVC_CHANNEL_INVALID

Invalid service channel specified. See [MEISqNodeCmdHeader](#).

MEISqNodeMessageCMD_NOT_SUPPORTED

The service command is not supported by the node.

MEISqNodeMessageDISCOVERY_FAILURE

Unable to discover node resources.

MEISqNodeMessageDISPATCH_ERROR

Is the default error code returned when a node specific routine has failed. Check the node FPGA version to verify whether or not it is correct.

MEISqNodeMessageINIT_FAILURE

A node specific initialization routine was unable to successfully complete its routine. Verify that the node FPGA is the default version for your MPI version. See [MEISqNodeInfoFpga.defaultVersion](#).

MEISqNodeMessageINTERFACE_ERROR1

This is an outdated node, which does not support the current discovery routine.

MEISqNodeMessageFILE_NODE_MISMATCH

Node type does not match the file provided for download.

MEISqNodeMessageFILE_INVALID

The file provided for download was not found or was corrupted.

MEISqNodeMessageINVALID_HEADER

The header information in the download image is invalid. Please verify firmware file to be correct and retry download. If firmware file is correct please contact firmware manufacturer.

MEISqNodeMessageDOWNLOAD_FAIL

Node firmware download failed. Verify that the firmware file is correct and retry the download.

NOTE: A network reset may be required.

MEISqNodeMessageVERIFY_FAIL

The node FPGA firmware does not match the FPGA image file.

MEISqNodeMessageDOWNLOAD_NOT_SUPPORTED

The downloading of the node firmware (FPGA) image is not supported for this node.

MEISqNodeMessageVERIFY_NOT_SUPPORTED

The Node specified for verification does not support the upload of the FPGA image. Therefore, the image cannot be verified.

MEISqNodeMessageBOOT_ROM_INVALID

The SqNode Boot Rom identification or version is not recognized by the MPI.

MEISqNodeMessageINVALID_TABLE

Invalid resource table in node module. This is a fatal error within the MPI. Please verify MPI and node FPGA versions to be correct and then contact MEI's Technical Support.

MEISqNodeMessageINVALID_STR_LEN

An attempt to write information to the node has failed due to an invalid string length.

MEISqNodeMessageFEEDBACK_MAP_INVALID

Returned from MEISqNodeConfigSet(...) when the given secondary encoder (n) is not mappable to the motor on the node specified by MEISqNodeFeedbackSecondary[n].motorIndex.

See Also [meiSqNodeDriveMapParamFileSet](#) | [meiSqNodeConfigSet](#)

MEISqNodeMonitorValue

MEISqNodeMonitorValue

```
typedef struct MEISqNodeMonitorValue {
    long    count;
    long    monitor[MEISqNodeMonitorValueIndexLAST];
} MEISqNodeMonitorValue;
```

Description The [SqNodeMonitorValue](#) structure contains the data for the monitor fields read by the `meiSqNodeDriveMonitor(...)` method.

count	The number of monitor fields read. This specifies the size of the monitor array.
monitor	An array of monitor data fields. Each field is indexed by the MEISqNodeMonitorValueIndex enumeration.

See Also [meiSqNodeDriveMonitor](#) | [meiSqNodeDriveMonitorConfigGet](#) | [meiSqNodeDriveMonitorConfigSet](#)

MEISqNodeMonitorValueIndex

MEISqNodeMonitorValueIndex

```
typedef enum MEISqNodeMonitorValueIndex {  
    MEISqNodeMonitorValueIndexA,  
    MEISqNodeMonitorValueIndexB,  
    MEISqNodeMonitorValueIndexC,  
    MEISqNodeMonitorValueIndexD,  
} MEISqNodeMonitorValueIndex;
```

Description [SqNodeMonitorValueIndex](#) is an enumeration of indices to node monitor values.

MEISqNodeMonitorValueIndexA	Index to node monitor value A.
MEISqNodeMonitorValueIndexB	Index to node monitor value B.
MEISqNodeMonitorValueIndexC	Index to node monitor value C.
MEISqNodeMonitorValueIndexD	Index to node monitor value D.

See Also [meiSqNodeDriveMonitor](#) | [meiSqNodeDriveMonitorConfigGet](#) | [meiSqNodeDriveMonitorConfigSet](#)

MEISqNodeResponse

MEISqNodeResponse

```
typedef struct MEISqNodeResponse {
    unsigned long    data;    /* response data */
} MEISqNodeResponse;
```

Description

The **SqNodeResponse** structure contains the service command response data.

data	The response information from a service command. The data field is only valid for MEISqNodeCmdTypeREAD command types.

See Also

[meiSqNodeCommand](#)

MEISqNodeStatus

MEISqNodeStatus

```
typedef struct MEISqNodeStatus {
    MEISqNodeStatusPacketError    upStreamError;
    MEISqNodeStatusPacketError    downStreamError;
    MEISqNodeStatusCrcError      crcError;
    MPIEventMask                  eventMask;
} MEISqNodeStatus;
```

Description The [SqNodeStatus](#) structure contains error counters and the *eventMask* for a SynqNet node.

upStreamError	The rate and count of bad synqNet messages received by the controller from the Node. See MEISqNodeStatusPacketError .
downStreamError	The rate and count of bad synqNet messages received by the Node from the controller. See MEISqNodeStatusPacketError .
crcError	Counters for the CRC errors. See MEISqNodeStatusCrcError .
eventMask	Array that defines the event mask bits. The array is defined as: typedef MPIEventMaskeLEMENT_TYPE MPIEEventMask[MPIEventMaskeLEMENTS] The bits are defined by the MPI/MEIEventType enumerations.

See Also [meiSqNodeStatus](#) | [meiSynqNetStatus](#) | [MEISqNodeConfig](#)

MEISqNodeStatusCrcError

MEISqNodeStatusCrcError

```
typedef struct MEISqNodeStatusCrcError {
    long    port[MEINetworkPortLAST];
} MEISqNodeStatusCrcError;
```

Description

The **SqNodeStatusCrcError** structure contains CRC error counters for each network port. The CRC error counters are helpful for diagnosing data integrity problems. The counter increments for any CRC error on any packet received at that port (whether the packet is addressed to the node or not). The CRC error counters are cleared during network initialization.

port	An array of CRC error counters. Each network port has one CRC error counter. The valid range is 0 to 255. The value saturates at 255.

See Also

[meiSqNodeStatus](#) | [MEINetworkPort](#)

MEISqNodeStatusPacketError

MEISqNodeStatusPacketError

```
typedef struct MEISqNodeStatusPacketError {
    long    rate;
    long    count;
} MEISqNodeStatusPacketError;
```

Description

The [SqNodeStatusPacketError](#) structure contains packet error counters and rate counters. Each SynqNet node has a packet error counter and a packet error rate counter. Packets addressed to a node are checked for integrity.

The packet error counters are used to monitor long-term data integrity. These counters do not trigger any fault or fail actions. The packet error counter is incremented once for each missing or invalid packet. Typically, an application will periodically read the packet error counters and store the values in a log.

The packet error rate counters are used to trigger fault recovery and/or failure shutdown. The packet error rate counter is incremented for each missing or invalid packet and is decremented for 16 consecutive valid packets. Thus, the packet error rate counters can detect large errors over short periods of time or small errors over long periods of time.

rate	The packet error rate counter. The valid range is 0 to 255. The value saturates at 255.
count	The packet error counter. The valid range is 0 to 255. The value saturates at 255.

See Also

[meiSqNodeStatus](#) | [MEISqNodeConfigPacketError](#)

MEISqNodeUserData

MEISqNodeUserData

```
typedef struct MEISqNodeUserData {
    char    data[MEISqNodeUserData_CHAR_MAX];
}MEISqNodeUserData ;
```

Description

The **SqNodeUserData** structure is used to store the user information that is located on the SqNode.

data	User information on the SqNode used for storing SqNode identification or any other useful data. Programmable string to be used by a customer to store identification-specific information. This data is not used by the MPI and is stored in the SqNode's EEPROM.

See Also

[meiSqNodeUserDataGet](#) | [meiSqNodeUserDataSet](#)

MEISqNodeID_CHAR_MAX

MEISqNodeID_CHAR_MAX

```
#define MEISqNodeID_CHAR_MAX (30)
```

Description

SqNodeID_CHAR_MAX defines the maximum length (number of characters) in a sqNode identification string.

See Also

MEISqNodeFILENAME_MAX

MEISqNodeFILENAME_MAX

```
#define MEISqNodeeeFILENAME_MAX (18)
```

Description

FILENAME_MAX defines the maximum size allowed for SqNode filenames.

See Also

MEISqNodeManufacturerDATA_CHAR_MAX

MEISqNodeManufacturerDATA_CHAR_MAX

```
#define MEISqNodeManufacturerDATA_CHAR_MAX (0x10)
```

Description

SqNodeManufacturerDATA_CHAR_MAX defines the maximum number of characters stored in the Manufacturer's Data field.

See Also

MEISqNodeMaxFEEDBACK_SECONDARY

MEISqNodeMaxFEEDBACK_SECONDARY

```
#define MEISqNodeMaxFEEDBACK_SECONDARY (MEISqNodeMaxMOTORS)
```

Description

SqNodeMaxFEEDBACK_SECONDARY defines the maximum number of secondary feedback devices per SynqNet node.

See Also

[MEISqNodeConfig](#) | [MEISqNodeFeedbackSecondary](#)

MEISqNodeMaxMOTORS

MEISqNodeMaxMOTORS

```
#define MEISqNodeMaxMOTORS (MEIXmpMotorsPerBlock)
```

Description

SqNodeMaxMOTORS defines the maximum number of motor objects supported on a single SynqNet node. This define should be used instead of the MEIXmpMotorsPerBlock definition in xmp.h. It is recommended that applications avoid programming to defines or structures in xmp.h

See Also [SqNode Objects](#)

MEISqNodeNOT_AVAILABLE

MEISqNodeNOT_AVAILABLE

```
#define MEISqNodeNOT_AVAILABLE (-1)
```

Description

SqNodeNOT_AVAILABLE defines a possible value for
MEISqNodeConfig.feedbackSecondary[n].motorIndex.

If motorIndex = MEISqNodeNOT_AVAILABLE, then a secondary feedback device does not
exist on the hardware.

See Also [MEISqNodeConfig](#) | [MEISqNodeFeedbackSecondary](#)

MEISqNodeSTATUS_NOT_AVAILABLE

MEISqNodeSTATUS_NOT_AVAILABLE

```
#define  MEISqNodeSTATUS_NOT_AVAILABLE  (-1)
```

Description

With exception to MEISqNodeStatus.eventMaskValue, the **SqNodeSTATUS_NOT_AVAILABLE** value is assigned to all Node status variables when the Status is not available as a result of lost communication with the node.

See Also [MEISqNodeStatus](#)

MEISqNodeUserData_CHAR_MAX

MEISqNodeUserData_CHAR_MAX

```
#define MEISqNodeUserData_CHAR_MAX (0x10)
```

Description

SqNodeUserData_CHAR_MAX defines the maximum number of characters in the User defined string, stored on the SqNode.

See Also

MEIDriveMapParamMAX_STRING_LENGTH

MEIDriveMapParamMAX_STRING_LENGTH

```
#define  MEIDriveMapParamMAX_STRING_LENGTH  (256)
```

Description

The **DriveMapParamMAX_STRING_LENGTH** macro defines the maximum length of a string that can be read from or written to a drive parameter.

See Also

MEIFPGARINCONREV

MPIFPGARINCONREV

```
#define MEIFPGARINCONREV      (0x0221)
```

Description

MEIFPGARINCONREV defines the version of the SynqNet controller FPGA image that was built and tested with the current version of the MPI.

See Also

MEIFpgaSqMACVersionDEFAULT

MEIFpgaSqMACVersionDEFAULT

```
#define MEIFpgaSqMACVersionDEFAULT (0x020E)
```

Description

MEIFpgaMACVersionDEFAULT defines the version of the SqMAC FPGA image that was built and tested with the current version of the MPI. The sqMAC FPGA image is built into the SynqNet node FPGA image. This version applies to all node types.

See Also

[MEIFpgaSqNodeVersionDEFAULT](#) | [MPI/SynqNet FPGA Compatibility Check](#)

MEIFpgaSqMACVersionMIN

MEIFpgaSqMACVersionMIN

```
#define MEIFpgaSqMACVersionMIN (0x0207)
```

Description

MEIFpgaMACVersionMIN defines the minimum version of the SqMAC FPGA image that is compatible with the current version of the MPI. The sqMAC FPGA image is built into the SynqNet node FPGA image.

This version applies to all node types.

See Also

[MEIFpgaSqNodeVersionDEFAULT](#) | [MPI/SynqNet FPGA Compatibility Check](#)

MEIFpgaSqMACVersionMAX

MEIFpgaSqMACVersionMAX

```
#define MEIFpgaSqMACVersionMAX      (0x02FF)
```

Description

MEIFpgaMACVersionMAX defines the maximum version of the SqMAC FPGA image that is compatible with the current version of the MPI. The sqMAC FPGA image is built into the SynqNet node FPGA image.

This version applies to all node types.

See Also

[MEIFpgaSqNodeVersionDEFAULT](#) | [MPI/SynqNet FPGA Compatibility Check](#)

MEIFpgaSqNodeVersionDEFAULT

MEIFpgaSqNodeVersionDEFAULT

```
#define MEIFpgaSqNodeVersionDEFAULT      (0x0311)
```

Description

MEIFpgaSqNodeVersionDEFAULT defines the version of the SynqNet node FPGA image that was built and tested with the current version of the MPI. This is the recommended version to have loaded on all SynqNet nodes.

This version may not apply to all node types.

See Also

[MEIFpgaSqNodeVersionMIN](#) | [MEIFpgaSqNodeVersionMAX](#) | [MPI/SynqNet FPGA Compatibility Check](#)

MEIFpgaSqNodeVersionMIN

MEIFpgaSqNodeVersionMIN

```
#define MEIFpgaSqNodeVersionMIN      (0x0303)
```

Description

MEIFpgaSqNodeVersionMIN defines the minimum version of the SynqNet node FPGA image that is compatible with the current version of the MPI.

This version may not apply to all node types.

See Also

[MEIFpgaSqNodeVersionMAX](#) | [MEIFpgaSqNodeVersionDEFAULT](#) | [MPI/SynqNet FPGA Compatibility Check](#)

MEIFpgaSqNodeVersionMAX

MEIFpgaSqNodeVersionMAX

```
#define MEIFpgaSqNodeVersionMAX      (0x03FF)
```

Description

MEIFpgaSqNodeVersionMAX defines the maximum version of the SynqNet node FPGA image that is compatible with the current version of the MPI.

This version may not apply to all node types.

See Also

[MEIFpgaSqNodeVersionMIN](#) | [MEIFpgaSqNodeVersionDEFAULT](#) | [MPI/SynqNet FPGA Compatibility Check](#)

SynqNet Objects

Introduction

A **SynqNet** object manages a single SynqNet network connected to a motion controller. It represents the physical network. It contains information about the network state, number of nodes, and network status.

A SynqNet network can have one or more [SqNode](#) objects associated with it and each SqNode object can have one or more motors and/or drive interfaces. The SynqNet network provides read/write access to each node. During network initialization, the SynqNet nodes are discovered and mapped to the SynqNet object. The number of motors per SqNode object are determined and mapped to the controller's motor objects.

Methods

Create, Delete, Validate Methods

[meiSynqNetCreate](#)

[meiSynqNetDelete](#)

[meiSynqNetValidate](#)

Configuration and Information Methods

[meiSynqNetCableNumToNodePort](#)

[meiSynqNetConfigGet](#)

[meiSynqNetConfigSet](#)

[meiSynqNetFlashConfigGet](#)

[meiSynqNetFlashConfigSet](#)

[meiSynqNetIdleCableListGet](#)

[meiSynqNetIdleCableStatus](#)

[meiSynqNetInfo](#)

[meiSynqNetPacketFlashConfigGet](#)

[meiSynqNetPacketFlashConfigSet](#)

[meiSynqNetPortToCableNum](#)

[meiSynqNetPacketConfigGet](#)

[meiSynqNetPacketConfigSet](#)

[meiSynqNetStatus](#)

[meiSynqNetTiming](#)

Action Methods

[meiSynqNetFlashTopologyClear](#)

[meiSynqNetFlashTopologySave](#)

[meiSynqNetInit](#)

[meiSynqNetShutdown](#)

Event Methods

[meiSynqNetEventNotifyGet](#)

[meiSynqNetEventNotifySet](#)

[meiSynqNetEventReset](#)

Relational Methods

[meiSynqNetControl](#)

[meiSynqNetNumber](#)

Data Types

[MEISynqNetCableList](#)

[MEISynqNetCableLength](#)

[MEISynqNetCableStatus](#)

[MEISynqNetConfig](#)

[MEISynqNetFailedNodeMask](#)

[MEISynqNetInfo](#)

[MEISynqNetMessage](#)

[MEISynqNetRecoveryMode](#)

[MEISynqNetResourceCommand](#)

[MEISynqNetResourceIoBits](#)

[MEISynqNetResourceMonitor](#)

[MEISynqNetPacketCfg](#)

[MEISynqNetPacketCfgEncoder](#)

[MEISynqNetPacketCfgIo](#)

[MEISynqNetPacketCfgMotor](#)

[MEISynqNetPacketCfgNode](#)

[MEISynqNetPacketCfgProbe](#)

[MEISynqNetResourceCfgProbeDepth](#)

[MEISynqNetState](#)

[MEISynqNetStatus](#)

[MEISynqNetStatusCrcError](#)

[MEISynqNetTiming](#)

[MEISynqNetTrace](#)

Constants

[MEISynqNetMaxCableHOP_COUNT](#)

[MEISynqNetMaxMotorFEEDBACK_PRIMARY_COUNT](#)

[MEISynqNetMaxMotorFEEDBACK_SECONDARY_COUNT](#)

[MEISynqNetMaxMotorCAPTURE_COUNT](#)

[MEISynqNetMaxMotorCOMPARE_COUNT](#)

[MEISynqNetMaxMotorENCODER_COUNT](#)

[MEISynqNetMaxMotorPULSE_ENGINE_COUNT](#)

[MEISynqNetMaxMotors](#)

[MEISynqNetMaxNodeMOTORS](#)

[MEISynqNetMaxNODE_COUNT](#)

[MEISynqNetNodeMaskELEMENTS](#)

[MEISynqNetPacketCfgIo](#)

[MEISynqNetPacketCfgMotor](#)

[MEISynqNetPacketCfgNode](#)

Required Header

```
stdmei.h
```

SynqNetCreate is the equivalent of a C++ constructor.

Return Values	
handle	to a SynqNet object. After creating a SynqNet object it must be validated using <code>meiSynqNetValidate()</code> .
MPIHandleVOID	if the object could not be created

file:///D:/pdfs/030100/html/Software-MPI/docs/Syngnet/Method/create2.htm [7/26/2004 10:53:41 AM]

meiSynqNetDelete

Declaration long **meiSynqNetDelete**([MEISynqNet](#) **synqNet**) ;

Required Header stdmei.h

Description **SynqNetDelete** deletes a SynqNet object and invalidates its handle.

SynqNetDelete is the equivalent of a C++ constructor.

synqNet	a handle to a SynqNet object.

Remarks

All objects that are created must be deleted in the reverse order to avoid memory leaks.

Return Values

MPIMessageOK	if <i>SynqNetDelete</i> successfully deleted the object.

See Also [meiSynqNetCreate](#) | [meiSynqNetValidate](#)

meiSynqNetValidate

Declaration long **meiSynqNetValidate**([MEISynqNet](#) **synqNet**);

Required Header stdmei.h

Description **SynqNetValidate** validates the SynqNet object and its handle. SynqNetValidate should be called immediately after an object is created.

synqNet	a handle to SynqNet object

Return Values

MPIMessageOK	if SynqNet is a handle to a valid object.
MEISynqNetMessageINTERFACE_NOT_FOUND	if the controller does not support a SynqNet interface.

See Also [meiSynqNetCreate](#) | [meiSynqNetDelete](#)

Required Header

Network connections can be identified by a cable number OR a node number and port. For simple network topologies, it is easier to identify network connections by a cable number. For complex network topologies, it is easier to identify network connections by a node number and port.

synqNet	a handle to the SynqNet object
cableNumber	the number of the cable
*nodeNumber	a pointer to a node number
*port	a pointer to an enumerated port value

Return Values

MPIMessageOK	if <i>SynqNetCableNumToNodePort</i> successfully converts a cable number on a SynqNet network into a node number and port.

See Also [meiSynqNetNodePortToCableNum](#) | [MEISynqNetCableList](#)

meiSynqNetConfigGet

Declaration

```
long meiSynqNetConfigGet(MEISynqNet synqNet,  
                        MEISynqNetConfig *config);
```

Required Header `stdmei.h`

Description	SynqNetConfigGet reads a SynqNet network (<i>synqNet</i>) configuration and writes it into the structure pointed to by <i>config</i> .

synqNet	a handle to a SynqNet object
*config	a pointer to a SynqNet config structure.

Return Values

MPIMessageOK	if <i>meiSynqNetConfigGet</i> successfully reads a SynqNet network's configuration and writes it into the structure pointed to by <i>config</i> .

See Also [meiSynqNetInfo](#) | [meiSqNodeConfigGet](#) | [meiSynqNetConfigSet](#)

meiSynqNetConfigSet

Declaration

long **meiSynqNetConfigSet**([MEISynqNet](#) **synqNet**,
[MEISynqNetConfig](#) ***config**);

Required Header

stdmei.h

Description

SynqNetConfigSet writes a SynqNet network (*synqNet*) configuration from the structure pointed to by *config*.

synqNet	a handle to a SynqNet object
*config	a pointer to a SynqNet config structure.

Return Values	
MPIMessageOK	if <i>meiSynqNetConfigSet</i> successfully writes a SynqNet network's configuration from the structure pointed to by <i>config</i> .
MPIMessageARG_INVALID	if config is NULL
MPIMessagePARAM_INVALID	if an invalid recoveryMode was specified or a non-zero cable length value was specified for a non-existent cable.
MPISynqNetMessageRING_ONLY	only a value of MEISynqNetRecoveryModeDISABLED for recoveryMode is allowed on a string network.

See Also

[meiSynqNetInfo](#) | [meiSqNodeConfigSet](#) | [meiSynqNetConfigGet](#)

meiSynqNetFlashConfigGet

Declaration

```
long meiSynqNetFlashConfigGet(MEISynqNet synqNet,
                              void *flash,
                              MEISynqNetConfig *config);
```

Required Header stdmei.h

Description [SynqNetFlashConfigGet](#) gets a SynqNet object's flash configuration and writes it in the structure pointed to by *config*..

synqNet	a handle to a SynqNet object
*config	pointer to a locally instantiated MEISynqNetConfig structure.
*flash	<i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if <i>meiSynqNetIdleFlashConfigGet</i> successfully writes the Motor's flash configuration to the structure(s).

See Also [meiSynqNetFlashConfigSet](#) | [MEISynqNetConfig](#) | [MEIFlash](#)

meiSynqNetFlashConfigSet

Declaration

```
long meiSynqNetFlashConfigSet(MEISynqNet synqNet,  
                             MEISynqNetConfig *flash,  
                             MEIFlash *config);
```

Required Header stdmei.h

Description [SynqNetFlashConfigSet](#) gets a SynqNet object's flash configuration and writes it in the structure pointed to by *config*.

NOTE: The network topology must first be saved before changing MEISynqNetConfig.cableLength[n] values in Flash memory. These values will also be cleared when network topology is cleared using meiSynqNetFlashTopologyClear(...).

synqNet	a handle to a SynqNet object
*config	pointer to a locally instantiated MEISynqNetConfig structure that has been initialized by performing a call to meiSynqNetFlashConfigGet(...) or meiSynqNetConfigGet(...) .
*flash	<i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally. If <i>flash</i> is a valid MEIFlash handle, then the MEIFlash object cache will be updated but the actual write to controller flash will not occur. Use meiFlashMemoryFromFileType(...) to prompt the actual write to flash.

Return Values

MPIMessageOK	if <i>meiSynqNetIdleFlashConfigSet</i> successfully sets the Motor's flash configuration using data from the structure(s).
MPIMessageARG_INVALID	if config is NULL.
MPIMessagePARAM_INVALID	if an invalid recoveryMode was specified or a non-zero cable length value was specified for a non-existent cable.

MEISynqNetMessageRING_ONLY	only a value of MEISynqNetRecoveryModeDISABLED for recoveryMode is allowed on a string network.
MEIFlashMessageNETWORK_TOPOLOGY_ERROR	if topology has not yet been saved to flash using meiSynqNetFlashTopologyClear(...)

See Also [MEISynqNetConfig](#) | [MEIFlash](#) | [meiSynqNetFlashConfigGet](#) | [meiSynqNetConfigGet](#) |
[meiSynqNetFlashTopologySave](#) | [meiSynqNetFlashTopologyClear](#)

meiSynqNetIdleCableListGet

Declaration

```
long meiSynqNetIdleCableListGet(MEISynqNet synqNet,
                                MEISynqNetCableList *idleCable);
```

Required Header stdmei.h

Description **SynqNetIdleLinkGet** reads a SynqNet network's list of idle cables and writes them into the structure pointed to by **idleCable**. An idle cable has no data traffic. It is the redundant network connection available in ring topologies only. A single ring topology has only one idle cable.

After network initialization, the cable from the last node to the controller is idle by default. If a network fault occurs and the network is configured for fault recovery, the network traffic will be redirected around the faulty connection, through the idle cable, and the faulty connection will become the new idle cable. To test the idle cable, use the meiSynqNetIdleCableStatus(...) method.

synqNet	a handle to a SynqNet object
*idleCable	a pointer to a SynqNet cable list structure. The cable list structure contains the number of idle cables and their identifying numbers.

Return Values

MPIMessageOK	if meiSynqNetIdleCableListGet successfully reads a SynqNet network's list of idle cables and writes them into the structure pointed to by idleCable.

See Also [meiSynqNetIdleCableStatus](#) | [SynqNet Topologies](#)

meiSynqNetIdleCableStatus

Declaration

```
long meiSynqNetIdleCableStatus(MEISynqNet
                                long
                                MEISynqNetCableStatus
                                synqNet,
                                cableNumber,
                                *cableStatus);
```

Required Header stdmei.h

Description [SynqNetIdleCableStatus](#) reads an idle cable's status and writes it into the structure pointed to by ***cableStatus***. Normally, the idle cable has no data traffic. SynqNetIdleCableStatus sends a special test packet across the specified idle cable and then waits for a valid response, then it sends another test packet in the opposite direction and waits for valid response.

The idle cable number for a network can be found using `meiSynqNetIdleCableListGet(...)`. SynqNetIdleCableStatus is not allowed for non-idle cables or when the network is recovering from a fault. During fault recovery (SynqNetState = SYNQ_RECOVERING), the network traffic is redirected around the faulty connection and the idle cable is reassigned. After recovery is complete (SynqNetState = SYNQ), the new idle cable can be tested with SynqNetIdleCableStatus. Use `meiSynqNetStatus(...)` to determine the SynqNet state.

synqNet	a handle to a SynqNet object
cableNumber	the number of the cable to be tested
*cableStatus	a pointer to a SynqNet cable status enumerated value.

Return Values

MPIMessageOK	if <i>meiSynqNetIdleCableStatus</i> successfully reads an idle cable's status and writes it into the structure pointed to by <i>cableStatus</i> .

See Also [meiSynqNetIdleCableListGet](#) | [MEISynqNetState](#)

Required Header `stdmei.h`

synqNet	a handle to a SynqNet object
*info	pointer to a SynqNet network information structure

MPIMessageOK	if <i>SynqNetInfo</i> successfully reads the network information and writes it into the info structure.
MPIMessageARG_INVALID	if the info pointer is NULL.

file:///D:/pdfs/030100/html/Software-MPI/docs/Synqnet/Method/inf2.htm [7/26/2004 10:53:43 AM]

meiSynqNetPacketFlashConfigGet

Declaration

```
long meiSynqNetPacketFlashConfigGet (MEISynqNet synqNet, void *flash, MEISynqNetPacketCfg *config);
```

Required Header stdmei.h

Description **SynqNetPacketFlashConfigGet** is currently unsupported and is reserved for future use.

synqNet	a handle to a SynqNet object
*flash	<i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally. If <i>flash</i> is a valid MEIFlash handle, then the MEIFlash object cache will be updated, but the actual write to controller flash will not occur. Use meiFlashMemoryFromFileType(...) to prompt the actual write to flash.
*config	pointer to configuration structure defined by MEISynqNetPacketCfg .

Return Values

MPIMessageUNSUPPORTED	is cuurrently not supported.

See Also [meiSynqNetPacketFlashConfigSet](#) | [MEISynqNetPacketCfgNode](#)

meiSynqNetPacketFlashConfigSet

Declaration

```
long meiSynqNetPacketFlashConfigSet(MEISynqNet synqNet,  
                                     void *flash,  
                                     MEISynqNetPacketCfg *config);
```

Required Header stdmei.h

Description [SynqNetPacketFlashConfigSet](#) is currently unsupported and is reserved for future use.

synqNet	a handle to a SynqNet object
*flash	<i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally. If <i>flash</i> is a valid MEIFlash handle, then the MEIFlash object cache will be updated but the actual write to controller flash will not occur. Use meiFlashMemoryFromFileType() to prompt the actual write to flash
*config	pointer to configuration structure defined by MEISynqNetPacketCfg.

Return Values

MPIMessageUNSUPPORTED	currently not supported.

See Also [meiSynqNetPacketFlashConfigGet](#) | [MEISynqNetPacketCfgNode](#)

meiSynqNetNodePortToCableNum

Declaration

```
long meiSynqNetNodePortToCableNum(MEISynqNet synqNet,  
                                   long      nodeNumber,  
                                   MEINetworkPort port,  
                                   long      *cableNumber);
```

Required Header stdmei.h

Description

[SynqNetNodePortToCableNum](#) converts a node number and port on a SynqNet network into a cable number. It reads the cable number and writes it into a long pointed to by ***cableNumber***.

Network connections can be identified by a cable number OR a node number and port. For simple network topologies, it is easier to identify network connections by a cable number. For complex network topologies, it is easier to identify network connections by a node number and port.

synqNet	a handle to the SynqNet object
nodeNumber	the number of the node
port	an enumerated port value
*cableNumber	a pointer to a cable number

Return Values	
MPIMessageOK	if <i>SynqNetPortToCableNum</i> successfully converts a node number and port on a SynqNet network into a cable number.

See Also [MEISynqNetCableList](#)

meiSynqNetPacketConfigGet

Declaration

```
long meiSynqNetPacketConfigGet (MEISynqNet synqNet,
                                MEISynqNetPacketCfg *config);
```

Required Header stdmei.h

Description **SynqNetPacketConfigGet** reads the current network packet configuration for all nodes found on the network to the location pointed to by *config*.

This method is useful for viewing the current network packed data being sent across the network. It is used in conjunction with SynqNetPacketConfigSet. This method is also useful for optimizing network traffic (bandwith).

Only configurable packet data fields are configured by this method. Fixed packet fields are not application configurable.

synqNet	a handle to a SynqNet object
*config	pointer to configuration structure defined by MEISynqNetPacketCfg.

Return Values	
MPIMessageOK	if <i>SynqNetPacketConfigGet</i> successfully reads the network packet configuration.
MPIMessageARG_INVALID	if the config pointer is NULL.
MPIMessageUNSUPPORTED	An sqNodeLib module is missing or it contains a corrupt a resource table.

See Also [meiSynqNetPacketConfigSet](#) | [MEISynqNetPacketCfgNode](#)

meiSynqNetPacketConfigSet

Declaration

```
long meiSynqNetPacketConfigSet(MEISynqNet synqNet,
                               MEISynqNetPacketCfg *config);
```

Required Header

stdmei.h

Description

SynqNetPacketConfigSet sets the network packet configuration to the configuration defined in the location pointed to by *config*.

WARNING: meiSynqNetPacketConfigSet(...) is a low-level network routine that will clear other controller object configurations and reset the SynqNet network. This method should be executed in your application before configuring any other MPI objects.

This method is useful for optimizing network traffic (bandwidth).

Only configurable packet data fields are configured by this method. Fixed packet fields are not application configurable.

synqNet	a handle to a SynqNet object
*config	pointer to configuration structure defined by MEISynqNetPacketCfg.

Return Values

MPIMessageOK	if <i>SynqNetPacketConfigSet</i> successfully writes the network packet configuration.
MPIMessageARG_INVALID	if the config pointer is NULL.
MEISynqNetMessageINCOMPLETE_MOTOR	minimum packet requirements for a motor have not been met. Be sure that your motor has at least one command field and one encoder.
MEISynqNetMessageINVALID_AUX_ENC_COUNT	too many secondary encoders have been configured on a node, or there are more secondary encoders than configured/enabled motors.
MEISynqNetMessageINVALID_MOTOR_COUNT	more motors configured than supported by the node.
MEISynqNetMessageINVALID_COMMAND_CFG	an unsupported command value has been selected, or it is not supported by the node.
MEISynqNetMessageINVALID_ENCODER_COUNT	feedback count exceeds MEISynqNetMaxEncoderFEEDBACK_COUNT, or exceeds the count supported by the node.
MEISynqNetMessageINVALID_CAPTURE_COUNT	capture count value is out of range, capture count exceeds capture resources supported by the node, or capture count is non-zero when encoder is disabled (feedbackCount == 0x0).

meiSynqNetStatus

Declaration

long **meiSynqNetStatus**([MEISynqNet](#)
[MEISynqNetStatus](#)**synqNet,**
***status**);

Required Header

stdmei.h

Description

SynqNetStatus reads status from the network associated with the SynqNet object and writes it into the structure pointed to by status. The SynqNet status structure contains network operation state and error counters. This data is updated every controller sample.

synqNet	a handle to a SynqNet object
*status	pointer to a SynqNet status structure

Return Values	
MPIMessageOK	if <i>meiSynqNetStatus</i> successfully gets the network status and writes it into the structure
MPIMessageARG_INVALID	if the status pointer is NULL

See Also

[meiSqNodeStatus](#) | [meiSynqNetInfo](#)

MEISynqNetMessageINVALID_COMPARE_COUNT	compare count value out of range, compare count exceeds compare resources supported by the node, or compare count is non-zero when encoder is disabled (feedbackCount == 0x0).
MEISynqNetMessageINVALID_INPUT_COUNT	ioInput count is out of range or it exceeds the count supported by the node.
MEISynqNetMessageINVALID_OUTPUT_COUNT	ioOutput count is out of range or it exceeds the count supported by the node.
MEISynqNetMessageINVALID_MONITOR_CFG	monitorConfig is out of range or it exceeds the count supported by the node.
MPIMessageUNSUPPORTED	an sqNodeLib module is missing or it contains a corrupt a resource table.

See Also [meiSynqNetMessageConfigGet](#) | [MEISynqNetMessageCfgNode](#) | [MEISynqNetMessageCfg](#)

Required Header

```
stdmei.h
```

synqNet	handle to a valid SynqNet object.
*timing	a pointer to a MEISynqNetTiming structure.

MPIMessageOK	if <i>meiSynqNetTiming</i> successfully retrieves the network timing values.
MPIMessageARG_INVALID	if *timing equals NULL.

file:///D:/pdfs/030100/html/Software-MPI/docs/Synqnet/Method/timing2.htm [7/26/2004 10:53:48 AM]

Required Header

```
stdmei.h
```

Executing this method will disable the controller from comparing the discovered network topology with the topology saved in flash memory. But, during network initialization, the MPI Library still compares the topology stored in the controller's dynamic memory with the discovered topology. In both cases, the default service commands are sent to the nodes if the network initialization transitions to SYNQ (cyclic) mode.

The meiSynqNetFlashTopologyClear method will:

1. Shutdown the SynqNet network.
2. Reset SqNode and Motor configurations to their defaults. See [MPI Object Configurations that use Service Commands](#).
3. Clear network topology information from Flash and Dynamic memory.
4. Initialize the SynqNet network.

synqNet	a handle to an MEISynqNet object whose network topology is to be cleared.
flash	is either a valid MEIFlash handle or MPIHandleVOID. If flash is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if <i>meiSynqNetFlashTopologyClear</i> successfully clears the networks topology information. <i>flash</i> is either an MEIFlash handle or MPIHandleVOID. If <i>flash</i> is MPIHandleVOID, an MEIFlash object will be created and deleted internally.
MPISynqNetMessageTOPOLOGY_CLEAR	if topology information has already been cleared or if it has not yet been saved to flash.

See Also

[Save/Clear Topology to Flash](#) | [meiSynqNetFlashTopologySave](#)

Required Header

```
stdmei.h
```

After a power-on or network shutdown/init, the controller/MPI Library will compare the discovered network topology with the topology saved in flash memory. If the topology information matches, the controller will automatically transition the network to SYNQ (cyclic) mode and configured service commands are sent to the nodes.

The meiSynqNetFlashTopologySave method will:

1. Shutdown the SynqNet network.
2. Reset SqNode and Motor configurations to their defaults. See [MPI Object Configurations that use Service Commands](#).
3. Save network topology information from Flash memory.
4. Initialize the SynqNet network.

synqNet	a handle to an MEISynqNet object whose network topology is to be saved.
flash	is either a valid MEIFlash handle or MPIHandleVOID. If flash is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

Return Values

MPIMessageOK	if <i>meiSynqNetFlashTopologySave</i> successfully saves the current SynqNet topology to the controller's flash memory.
MPISynqNetMessageTOPOLOGY_SAVED	if topology information has already been saved to flash. You must clear network topology before saving another network topology to flash.

See Also [Save/Clear Topology to Flash](#) | [meiSynqNetFlashTopologyClear](#)

meiSynqNetInit

Declaration long **meiSynqNetInit**([MEISynqNet](#) **synqNet**) ;

Required Header stdmei.h

Description **SynqNetInit** initializes a SynqNet network. This method performs the same network initialization that is automatically done with mpiControlInit(...).

synqNet	a handle to a SynqNet object

Return Values

MPIMessageOK	if <i>SynqNetInit</i> successfully initializes the network
MPIMessageARG_INVALID	if the <i>eventMask</i> pointer is NULL.

See Also [meiSynqNetInfo](#) | [meiSynqNetStatus](#)

meiSynqNetShutdown

Declaration long **meiSynqNetShutdown**([MEISynqNet](#) **synqNet**) ;

Required Header stdmei.h

Description **SynqNetEventShutdown** disables a SynqNet network by shutting off cyclic data transmission from the controller to the nodes. This will cause the network state to transition to MEISynqNetStateDISCOVERY and nodes will go into a SynqLost state and disable their outputs.

synqNet	a handle to a SynqNet object

Return Values

MPIMessageOK	if <i>meiSynqNetShutdown</i> successfully disables a SynqNet network by shutting off cyclic data transmission from the controller to the nodes.

See Also [meiSynqNetInit](#) | [mpiControlReset](#) | [mpiControlInit](#) | [meiSynqNetStatus](#)

meiSynqNetEventNotifyGet

Declaration

long meiSynqNetEventNotifyGet([MEISynqNet](#)
[MPIEventMask](#)
void
synqNet,
*eventMask,
*external);

Required Header

stdmei.h

Description

SynqNetEventNotifyGet reads the event mask (that specifies the event types 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).

Use the event mask macros mpiEventMaskGET(...), mpiEventMaskBitGET(...), etc. to decode the eventMask.

The event notification data in external is in addition to the event notification data in eventMask. If either eventMask is NULL or external is NULL (not both), then the event notification data will not be copied to the NULL pointer.

synqNet	a handle to a SynqNet object
*eventMask	pointer to an event mask, whose bits are defined by the MPI/MEIEventType enumerations.
*external	pointer to external

XMP Only

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

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

Return Values	
MPIMessageOK	if meiSynqNetEventNotifyGet successfully reads the event mask
MPIMessageARG_INVALID	if the eventMask pointer is NULL.

See Also

[MEI/MPIEventType](#) | [MEIEventNotifyData](#) | [MEIEventStatusInfo](#)

meiSynqNetEventNotifySet

Declaration

long meiSynqNetEventNotifySet([MEISynqNet](#)
 [MPIEventMask](#)
 void **synqNet,**
 eventMask,
 ***external**);

Required Header

stdmei.h

Description

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

Use the event mask macros `meiEventMaskSYNQNET(...)`, `mpiEventMaskSET(...)`, `mpiEventMaskBitSET(...)`, `mpiEventMaskCLEAR(...)`, etc. to create the `eventMask`.

The event notification data in **external** is in addition to the event notification data in **eventMask**. If either **eventMask** is NULL or **external** is NULL (not both), then the event notification data will not be copied to the NULL pointer.

synqNet	a handle to a SynqNet object
eventMask	pointer to an event mask, whose bits are defined by the MPI/MEIEventType enumerations.
*external	pointer to external

XMP Only

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

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

Return Values

MPIMessageOK	if <i>SynqNetEventNotifySet</i> successfully writes the event mask
MPIMessageARG_INVALID	if the eventMask pointer is NULL

See Also

[MEI/MPIEventType](#) | [MEIEventNotifyData](#) | [MEIEventStatusInfo](#)

meiSynqNetEventReset

Declaration

long [meiSynqNetEventReset](#)([MEISynqNet](#)
[MEIEventMask](#) **synqNet**,
eventMask);

Required Header

stdmei.h

Description

[SynqNetEventReset](#) resets the status/event bits that are specified in the eventMask and generated by the synqNet object. After a SynqNet event occurs, SynqNetEventReset should be called to reset the latched status/event bits so the specified event(s) can be generated again.

synqNet	a handle to a SynqNet object
eventMask	An array that defines the event mask bits. The array is defined as: <pre>typedef MPIEventMaskELEMENT_TYPE MPIEventMask[MPIEventMaskeLEMENTS]</pre> The bits are defined by the MPI/MEIEventType enumerations.

Return Values

MPIMessageOK

if *meiSynqNetEventReset* successfully resets the status/event bits that are specified in the eventMask and generated by the synqNet object.

See Also

[meiSynqNetStatus](#) | [meiSqNodeEventReset](#) | [meiSqNodeStatus](#)

meiSynqNetControl

Declaration `meiSynqNetControl(MEISynqNet    synqNet ) ;`

Required Header `stdmei.h`

Description `SynqNetControl` returns a handle to the control object associated with the *SynqNet* object.

<code>synqNet</code>	a handle to a SynqNet object

Return Values

<code>MPIControl</code>	a handle to a control object.
<code>MPIHandleVOID</code>	if synqNet is not valid.

See Also [meiSynqNetCreate](#) | [mpiControlCreate](#)

meiSynqNetNumber

```
long meiSynqNetNumber(MEISynqNet synqNet,  
long *number);
```

Required Header

Description	SynqNetNumber reads the index of a SynqNet network and writes it into the contents of a long pointed to by number. Each SynqNet network associated with a controller is indexed by an identification number (0, 1, 2, etc.).

synqNet	a handle to a SynqNet object.
*number	a pointer to the index of a SynqNet network.

Return Values

MPIMessageOK	if <i>SynqNetNumber</i> successfully reads the network number.

See Also [meiSynqNetInfo](#)

MEISynqNetCableList

MEISynqNetCableList

```
typedef struct MEISynqNetCableList {
    long    count;
    long    cableNumber[MEISynqNetCableHOP_COUNT];
} MEISynqNetCableList;
```

Description

The **SynqNetCableList** structure contains the number of cables in the SynqNet network and their identifying numbers. The idle cables can be identified with the `meiSynqNetIdleCableListGet(...)` method.

A cable can also be identified by the node number and port. Use [meiSynqNetCableNumToNodePort\(...\)](#) to translate the cable number to a node number and a port.

count	The number of cables and number of valid elements in the cableNumber[] array.. Range is 0 to MEISynqNetCableHOP_COUNT.
cableNumber	The cables identified by their number. Cables are numbered in the order they are discovered during network initialization.

See Also

[meiSynqNetIdleCableListGet](#) | [meiSynqNetIdleCableStatus](#) | [meiSynqNetCableNumToNodePort](#)

MEISynqNetCableLength

MEISynqNetCableLength

```
typedef struct MEISynqNetCableLength { /* lengths in meters */
    long minimum;
    long nominal;
    long maximum;
} MEISynqNetCableLength;
```

Description

The [SynqNetCableLength](#) structure contains the nominal (average), minimum and maximum length in meters for a given network cable.

To disable the cable length checking feature, set the **minimum**, **nominal**, and **maximum** values to zero.

minimum	Minimum cable length in meters. A value less than zero or greater than the nominal value is invalid.
nominal	Nominal cable length in meters. A value less than zero is invalid.
maximum	Maximum cable length in meters. A value less than the nominal value is invalid.

See Also

[MEISynqNetConfig](#) | [meiSynqNetConfigSet](#) | [meiSynqNetConfigSet](#) | [meiSynqNetInfo](#)

MEISynqNetCableStatus

MEISynqNetCableStatus

```
typedef enum MEISynqNetCableStatus {  
    MEISynqNetCableStatusGOOD,  
    MEISynqNetCableStatusBAD_UPSTREAM,  
    MEISynqNetCableStatusBAD_DOWNSTREAM,  
    MEISynqNetCableStatusBAD,  
} MEISynqNetCableStatus;
```

Description

SynqNetCableStatus is an enumeration of a network connection's operating condition. Data transmission is verified in both directions. If the network hardware does not support separate upstream and downstream data path verification, it will report either GOOD or BAD.

MEISynqNetCableStatusGOOD	Network communication across the cable is working properly in both directions.
MEISynqNetCableStatusBAD_UPSTREAM	Network communication across the cable failed in the upstream direction (from node to controller).
MEISynqNetCableStatusBAD_DOWNSTREAM	Network communication across the cable failed in the downstream direction (from the controller to the node).
MEISynqNetCableStatusBAD	Network communication across the cable failed in both directions.

See Also

[meiSynqNetIdleCableStatus](#) | [meiSynqNetIdleCableListGet](#)

MEISynqNetConfig

MEISynqNetConfig

```
typedef struct MEISynqNetConfig {
    MEISynqNetRecoveryMode    recoveryMode;
    MEISynqNetCableLength    cableLength[MEISynqNetCableHOP\_COUNT];
} MEISynqNetConfig;
```

Description

The [SynqNetConfig](#) structure contains configurations for the network's fault recovery mode and the network's cable lengths. The SynqNet configuration can be read with `meiSynqNetConfigGet(...)` and can be written with `meiSynqNetConfigSet(...)`.

recoveryMode	A enumerated value representing the network's fault recovery response mode.
cableLength	An array of cable lengths. Range for cable lengths is 0 to 100 meters. This structure is provided to allow the user application to specify minimum, maximum, and nominal values to use network scheduling equations. Setting a minimum and maximum value will also enable cable length checking at network initialization time. If measured cable lengths fall outside the range defined by the minimum and maximum values, network initialization will fail with a return value of <code>MEISynqNetMessageCABLE_LENGTH_MISMATCH</code>. By default, the values in this structure are zero. Most applications will not need to modify these defaults. NOTE: The network topology must be saved before changing these values in Flash memory. These values will also be cleared when network topology is cleared using <code>meiSynqNetFlashTopologyClear(...)</code>

See Also

[meiSynqNetConfigGet](#) | [meiSynqNetConfigSet](#) | [meiSynqNetInfo](#) | [meiSynqNetFlashConfigGet](#) | [meiSynqNetFlashConfigSet](#) | [meiSynqNetFlashTopologyClear](#)

MEISynqNetFailedNodeMask

MEISynqNetFailedNodeMask

```
#define MEISynqNetNodeMaskELEMENTS (1)
typedef long MEISynqNetFailedNodeMask [MEISynqNetNodeMaskELEMENTS];
```

Description

SynqNetFailedNodeMask is an array of longs with a length of MEISynqNetNodeMaskELEMENTS. Each bit in the failed node mask represents a failed node (0x1 = node 0, 0x2 = node 2, 0x4 = node 3, etc.). A node failure occurs when the packet error rate counters exceed the packet error rate fail limit.

See Also

[MEISynqNetStatus](#) | [meiSynqNetStatus](#) | [meiSqNodeStatus](#) | [MEISqNodeConfig](#)

MEISynqNetInfo

MEISynqNetInfo

```
typedef struct MEISynqNetInfo {
    MEINetworkType      networkType;
    long                nodeCount;
    long                nodeOffset;
    MEISynqNetCableLength cableLength[MEISynqNetCableHOP\_COUNT];
                        /* measured length */
} MEISynqNetInfo;
```

Description The [SynqNetInfo](#) structure contains static data that is determined during network initialization. It identifies the network type, number of nodes on the network, starting number (offset) of the first node, and an rough estimate of measured cable lengths.

networkType	contains currently discovered network topology type.
nodeCount	Contains the number of nodes currently discovered on the network.
nodeOffset	Starting node number for nodes on this network. Nodes are currently numbered sequentially across all networks on a controller.
cableLength	An array of measured network cable lengths. These values are estimated values. At network discovery, where network topology has not been saved to flash, the controller will send network packets to measure each cable length and automatically fill out these values for each cable found. These values are estimated values and are the values used in the network scheduling calculations.

See Also [meiSynqNetInfo](#) | [MEISynqNetConfig](#)

MEISynqNetMessage

MEISynqNetMessage

```
typedef enum {
    MEISynqNetMessageSYNQNET_INVALID,
    MEISynqNetMessageMAX_NODE_ERROR,
    MEISynqNetMessageSTATE_ERROR,

    MEISynqNetMessageCOMM_ERROR,
    MEISynqNetMessageCOMM_ERROR_CRC,
    MEISynqNetMessageCOMM_ERROR_RX,
    MEISynqNetMessageCOMM_ERROR_RX_LEN,
    MEISynqNetMessageCOMM_ERROR_RX_FIFO,
    MEISynqNetMessageCOMM_ERROR_RX_DRIBBLE,
    MEISynqNetMessageCOMM_ERROR_RX_CRC,

    MEISynqNetMessageINTERFACE_NOT_FOUND,
    MEISynqNetMessageTOPOLOGY_MISMATCH,
    MEISynqNetMessageTOPOLOGY_MISMATCH_FLASH,

    MEISynqNetMessageRESET_REQ_TIMEOUT,
    MEISynqNetMessageRESET_ACK_TIMEOUT,
    MEISynqNetMessageDISCOVERY_TIMEOUT,
    MEISynqNetMessageNO_NODES_FOUND,

    MEISynqNetMessageNO_TIMING_DATA_AVAIL,

    MEISynqNetMessageINTERNAL_BUFFER_OVERFLOW,
    MEISynqNetMessageINVALID_MOTOR_COUNT,
    MEISynqNetMessageINVALID_AUX_ENC_COUNT,
    MEISynqNetMessageINCOMPLETE_MOTOR,
    MEISynqNetMessageINVALID_COMMAND_CFG,
    MEISynqNetMessageINVALID_ENCODER_COUNT,
    MEISynqNetMessageINVALID_CAPTURE_COUNT,
    MEISynqNetMessageINVALID_COMPARE_COUNT,
    MEISynqNetMessageINVALID_INPUT_COUNT,
    MEISynqNetMessageINVALID_OUTPUT_COUNT,
    MEISynqNetMessageINVALID_MONITOR_CFG,
    MEISynqNetMessageINVALID_ANALOG_IN_COUNT,

    MEISynqNetMessageLINK_NOT_IDLE,
    MEISynqNetMessageIDLE_LINK_UNKNOWN,
    MEISynqNetMessageRING_ONLY,
    MEISynqNetMessageRECOVERING,

    MEISynqNetMessageCABLE_LENGTH_UNSUPPORTED,
    MEISynqNetMessageCABLE_LENGTH_MISMATCH,
    MEISynqNetMessageCABLE_LENGTH_INVALID_NOMINAL,
    MEISynqNetMessageCABLE_LENGTH_INVALID_MIN,
    MEISynqNetMessageCABLE_LENGTH_INVALID_MAX,
```

```

    MEISynqNetMessageNODE_FPGA_VERSION,
    MEISynqNetMessageMAX_NODE_ERROR,
    MEISynqNetMessagePLL_ERROR,
    MEISynqNetMessageNODE_INIT_FAIL,

    MEISynqNetMessageTOPOLOGY_CLEAR,
    MEISynqNetMessageTOPOLOGY_SAVED,

    MEISynqNetMessageNODE_MAC_VERSION,
    MEISynqNetMessageADC_SAMPLE_FAILURE,

    MEISynqNetMessageSCHEDULING_ERROR,
    MEISynqNetMessageINVALID_PROBE_CFG,
    MEISynqNetMessageINVALID_PROBE_DEPTH,
} MEISynqNetMessage;

```

Description

SynqNetMessage is an enumeration of SynqNet error messages that can be returned by the MPI library.

MEISynqNetMessageSYNQNET_INVALID

The SynqNet number is out of range. This message code is returned by [meiSynqNetCreate\(...\)](#) if the SynqNet network number is less than zero or greater than or equal to MEIXmpMaxSynqNets.

MEISynqNetMessageMAX_NODE_ERROR

The SynqNet node number is out of range. This message code is returned by a SynqNet method if the specified node number is greater than or equal to MEIXmpMaxSynqNetBlocks.

MEISynqNetMessageSTATE_ERROR

The SynqNet network state is not valid. This message code is returned by any method that initializes a SynqNet network if the controller's network state is not a member of the MEIXmpSynqNetState or MEIXmpSynqNetInternalState enumeration. The most commonly used methods that initialize the SynqNet network are: [mpiControlInit\(...\)](#), [mpiControlReset\(...\)](#) and [meiSynqNetInit\(...\)](#). This message code indicates a failure in the controller's initialization sequence. To correct this problem, call [mpiControlReset\(...\)](#).

MEISynqNetMessageCOMM_ERROR

The SynqNet network communication failed. This message code is returned by MPI methods that fail a service command transaction due to a network shutdown. This message indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageCOMM_ERROR_CRC

The SynqNet network communication failed due to excessive CRC errors. This message code is returned by MPI methods that fail a service command transaction due to a network shutdown. This message indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageCOMM_ERROR_RX

The SynqNet network communication failed due to a Rincon receive error. This message code is returned by MPI methods that fail a service command transaction due to a network shutdown. This message indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageCOMM_ERROR_RX_LEN

The SynqNet network communication failed due to a Rincon receive length error. This message code is returned by MPI methods that fail a service command transaction due to a network shutdown. This message indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageCOMM_ERROR_RX_FIFO

The SynqNet network communication failed due to a Rincon receive buffer error. This message code is returned by MPI methods that fail a service command transaction due to a network shutdown. This message indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageCOMM_ERROR_RX_DRIBBLE

The SynqNet network communication failed due to a Rincon receive dribble error. This message code is returned by MPI methods that fail a service command transaction due to a network shutdown. This message indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageCOMM_ERROR_RX_CRC

The SynqNet network communication failed due to a Rincon receive CRC error. This message code is returned by MPI methods that fail a service command transaction due to a network shutdown. This message indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageINTERFACE_NOT_FOUND

The controller does not support a SynqNet interface. This message code is returned by [meiSynqNetValidate\(...\)](#), [meiSynqNetFlashTopologySave\(...\)](#), [meiSynqNetFlashTopologyClear\(...\)](#), [mpiControlInit\(...\)](#), and [mpiControlReset\(...\)](#) if the controller does not have a SynqNet hardware interface. To correct this problem, use a controller that supports SynqNet.

MEISynqNetMessageTOPOLOGY_MISMATCH

The network topology does not match the expected network topology. This message code is returned by [mpiControlInit\(...\)](#) or [meiSynqNetInit\(...\)](#) if the discovered network topology does not match the controller's expected network topology (stored in dynamic memory). During the first network initialization the controller stores node identification information (manufacturer, product, and unique values) into its dynamic memory. This message code indicates the number of nodes, the node order, or types of nodes have changed since the initial network initialization. To correct this problem, either change the network topology to the original configuration or clear the controller's memory with [mpiControlReset\(...\)](#).

MEISynqNetMessageTOPOLOGY_MISMATCH_FLASH

The network topology does not match the expected network topology. This message code is returned by [mpiControlInit\(...\)](#) or [meiSynqNetInit\(...\)](#) if the discovered network topology does not match the controller's expected network topology (stored in flash memory). During the first network initialization the controller stores node identification information (manufacturer, product, and unique values) into its dynamic memory. Later, when [meiSynqNetFlashTopologySave\(...\)](#) is called, the topology information is stored into flash memory. This message indicates the number of nodes, the node order, or types of nodes have changed since the topology information was stored in flash. To correct this problem, either change the network topology to the saved configuration or clear the controller's flash topology with [meiSynqNetFlashTopologyClear\(...\)](#).

MEISynqNetMessageRESET_REQ_TIMEOUT

The network reset request packet exceeded the timeout. This message code is returned by [mpiControlInit\(...\)](#), [mpiControlReset\(...\)](#), or [meiSynqNetInit\(...\)](#) if the reset request packet fails to traverse the network in the allotted time. This message code indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageRESET_ACK_TIMEOUT

The network reset complete packet exceeded the timeout. This message code is returned by [mpiControlInit\(...\)](#), [mpiControlInit\(...\)](#), or [meiSynqNetInit\(...\)](#) if the reset complete packet fails to traverse the network in the allotted time. This message code indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageDISCOVERY_TIMEOUT

The network topology discovery exceeded the timeout. This message code is returned by [mpiControlInit\(...\)](#), [mpiControlInit\(...\)](#), or [meiSynqNetInit\(...\)](#) if the controller failed to discover the network topology in the allotted time. This message code indicates a node or network cable failure. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageNO_NODES_FOUND

The controller did not find network nodes. This message code is returned by [mpiControlInit\(...\)](#), [meiSynqNetPacketConfigSet\(...\)](#), or [meiSynqNetInit\(...\)](#) if the controller failed to discover any nodes during network initialization. This message code indicates the first node has failed or the network connection from the controller to the first node is faulty. To correct this problem, check your network wiring and node condition.

MEISynqNetMessageNO_TIMING_DATA_AVAIL

The corresponding SynqNet node module does not contain timing data, so the network cannot be initialized. Contact MEI or drive manufacturer for an updated node module.

MEISynqNetMessageINTERNAL_BUFFER_OVERFLOW

The controller's SynqNet buffer size was exceeded. This message code is returned by [mpiControlInit\(...\)](#), [mpiControlReset\(...\)](#), or [meiControlInit\(...\)](#) if the controller's SynqNet buffer could not be allocated due to overflow. This message code indicates the controller does not have enough memory to initialize the network topology. To correct the problem, either reduce the network nodes or use a different controller model.

MEISynqNetMessageINVALID_MOTOR_COUNT

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains more motors than supported by the node.

MEISynqNetMessageINVALID_AUX_ENC_COUNT

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains more secondary encoders than supported by the node.

MEISynqNetMessageINCOMPLETE_MOTOR

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains a motor configuration that is missing feedback or command fields.

MEISynqNetMessageINVALID_COMMAND_CFG

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains a different command configuration than is supported by the node.

MEISynqNetMessageINVALID_ENCODER_COUNT

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains more encoders than supported by the node.

MEISynqNetMessageINVALID_CAPTURE_COUNT

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains a larger capture count than is supported by the node.

MEISynqNetMessageINVALID_COMPARE_COUNT

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains a larger compare count than is supported by the node.

MEISynqNetMessageINVALID_INPUT_COUNT

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains a larger ioInput count than is supported by the node.

MEISynqNetMessageINVALID_OUTPUT_COUNT

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains a larger ioOutput count than is supported by the node.

MEISynqNetMessageINVALID_MONITOR_CFG

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains more monitor fields than the node can support.

MEISynqNetMessageINVALID_ANALOG_IN_COUNT

This message is returned by [meiSynqNetPacketConfigSet\(...\)](#), when the packet configuration contains a larger analogIn count than is supported by the node.

MEISynqNetMessageLINK_NOT_IDLE

This message is returned by [meiSynqNetIdleCableStatus\(...\)](#) when the cable number supplied is not the idle link. The status check can only be run on an idle cable. See [meiSynqNetIdleCableListGet\(...\)](#).

MEISynqNetMessageIDLE_LINK_UNKNOWN

This message is returned by [meiSynqNetIdleCableListGet\(...\)](#) when an idle cable number in a ring topology cannot be determined. This is due to one or more failed nodes on a ring topology. Be sure to check the status of the nodes.

MEISynqNetMessageRECOVERING

The network is recovering from a fault condition. This message is returned by [meiSynqNetIdleCableStatus\(...\)](#) during network fault recovery, because the network traffic is being redirected around the faulted cable and a new idle cable is in the process of being assigned. If you receive this message, wait for recovery to complete and then check the identity of the idle cable with [meiSynqNetIdleCableListGet\(...\)](#).

MEISynqNetMessageRING_ONLY

This message is returned by [meiSynqNetIdleCableListGet\(...\)](#) and [meiSynqNetIdleCableStatus\(...\)](#) because idle cables exist on ring topologies only. It is also returned by [meiSynqNetConfigSet\(...\)](#) when attempting to enable SynqNet recovery mode, which is only supported for ring topologies.

MEISynqNetMessageCABLE_LENGTH_UNSUPPORTED

This message is returned when a cable on the network is too long.

MEISynqNetMessageCABLE_LENGTH_MISMATCH

This message is returned at network initialization when topology has been saved to flash and a cable length is detected that lies outside the `MEISynqNetConfig.cableLength[n].minimum` and `maximum`. Check cable length (if necessary) and save new cable length values to flash.

MEISynqNetMessageCABLE_LENGTH_INVALID_NOMINAL

The nominal cable length is too long.

MEISynqNetMessageCABLE_LENGTH_INVALID_MIN

The minimum cable length is too long or exceeds nominal value.

MEISynqNetMessageCABLE_LENGTH_INVALID_MAX

The maximum cable length is too long or is less than nominal value.

MEISynqNetMessageNODE_FPGA_VERSION

The node resource FPGA image is out of date. This is only a warning message. Your network will still reach cyclic (SYNQ) mode, but certain node resources may not be available or may not operate as expected. Only ignore this warning if you are aware of FPGA changes between this version and the version built with the MPI or use the `sqNodeFlash.exe` utility to load the most current FPGA version to the node.

MEISynqNetMessageMAX_NODE_ERROR

The number of motors on the network exceeds the number set by [MEISynqNetMaxMOTORS](#).

MEISynqNetMessagePLL_ERROR

The node PLL is unable to lock with drive.

MEISynqNetMessageNODE_INIT_FAIL

A node specific initialization routine was unable to complete successfully. Verify node FPGA is the default version for your MPI version. See [MEISqNodeInfoFpga.defaultVersion](#).

MEISynqNetMessageTOPOLOGY_CLEAR

An attempt to clear the saved network topology was made when no network topology has been saved.

MEISynqNetMessageTOPOLOGY_SAVED

An attempt to save network topology was made when the network topology has already been saved. Clear the network topology before attempting to save another topology to flash.

MEISynqNetMessageNODE_MAC_VERSION

The node MAC FPGA image is out of date. This is an error message. Your network will probably never reach cyclic (SYNQ) mode. Use the sqNodeFlash.exe utility to load the most current FPGA version to the node.

MEISynqNetMessageADC_SAMPLE_FAILURE

A check to verify that all analog inputs can be sampled during the given controller sample rate has failed. Either reduce the number of analog inputs on your network (see [meiSynqNetMessageConfigGet](#) / [meiSynqNetMessageConfigSet](#)) or decrease your controller sample rate (see [mpiControlConfigGet](#) / [mpiControlConfigSet](#)).

MEISynqNetMessageSCHEDULING_ERROR

This is a generic return value to indicate that a problem has occurred while calculating the network scheduling. For more specific information about the error, run your application with timing assignment tracing turned on, using the trace mask:

0x00100000 SynqNet: Display timing assignments

MEISynqNetMessageINVALID_PROBE_CFG

This message is returned by [meiSynqNetMessageConfigSet\(...\)](#), when the [MEISynqNetMessageCfgProbe](#).count contains a larger count than is supported by the node.

MEISynqNetMessageINVALID_PROBE_DEPTH

This message is returned by [meiSynqNetMessageConfigSet\(...\)](#), when the [MEISynqNetMessageCfgProbe](#).depth contains a larger count than is supported by the node.

See Also

MEISynqNetRecoveryMode

MEISynqNetRecoveryMode

```
typedef enum MEISynqNetRecoveryMode {  
    MEISynqNetRecoveryModeDISABLED,  
    MEISynqNetRecoveryModeSINGLE_SHOT,  
    MEISynqNetRecoveryModeAUTO_ARM,  
} MEISynqNetRecoveryMode;
```

Description

SynqNetRecoveryMode is an enumeration of a network's fault recovery mode. Networks with ring topologies can be configured to recover from a fault condition. A fault condition occurs when a packet error rate counter exceeds its packet error rate fault limit. If fault recovery is enabled and a fault occurs, the node hardware and/or controller will detect the location of the fault and switch the direction of network traffic for nodes downstream from the faulted connection. During fault recovery, the network state is MEISynqNetStateSYNQ_RECOVERING. After the upstream and downstream packet error rate counters decrement to zero, the recovery is completed, and the network state returns to MEISynqNetStateSYNQ.

When fault recovery occurs, the controller will generate a MEIEventTypeSYNQNET_RECOVERY status/event. An application should notify the user about the fault and fix the broken cable/hardware as soon as possible. An application can determine the location of the fault using meiSynqNetIdleCableListGet(...). A network with a ring topology has one idle cable, which has no data traffic. The operation of the idle cable can be tested with meiSynqNetIdleCableStatus(...).

WARNING: If the idle cable is broken and a second fault occurs, then fault recovery will not be able to fully recover from the fault. SynqNet will try to recover, but some nodes will be stranded. Presently, SynqNet does not support an event to notify the application if an idle cable fails. In the meantime, an application can periodically poll the idle cable with meiSynqNetIdleCableStatus(...) to test the cable.

MEISynqNetRecoveryModeDISABLED	The network will not attempt to recover from a fault condition. This is the default mode for string topologies.
MEISynqNetRecoveryModeSINGLE_SHOT	The network will only attempt to recover from a fault condition one time. A second fault will be ignored.
MEISynqNetRecoveryModeAUTO_ARM	The network will attempt to recover from a fault condition. After the fault recovery is complete, the network will automatically be re-armed to respond to another fault condition. This is the default mode for ring topologies.

See Also

[meiSynqNetConfigGet](#) | [meiSynqNetConfigSet](#) | [meiSynqNetStatus](#) | [meiSynqNetInfo](#) | [meiSynqNetIdleCableListGet](#) | [meiSynqNetIdleCableStatus](#)

MEISynqNetResourceCommand

MEISynqNetResourceCommand

```
typedef enum MEISynqNetResourceCommand {
    MEISynqNetResourceCommandINVALID,
    MEISynqNetResourceCommandNONE,
    MEISynqNetResourceCommandDAC_ONE,
    MEISynqNetResourceCommandDAC_TWO,
    MEISynqNetResourceCommandDEMAND_A,
    MEISynqNetResourceCommandDEMAND_ABC,
} MEISynqNetResourceCommand;
```

Description

The [SynqNetResourceCommand](#) enumeration is a member of the MEISynqNetPacketCfgMotor configuration structure. The values are used to configure the number of 32-bit command data fields sent to drive a motor. Valid values range from greater or equal to MEISynqNetResourceCommandFIRST and less than MEISynqNetResourceCommandLast, but may be further limited by the available resources on the node.

NOTE: A motor is considered incomplete if it has a command value of NONE.

MEISynqNetResourceCommandINVALID	Non-vaild command value was detected.
MEISynqNetResourceCommandNONE	No command data will be sent to this motor. NOTE: this disables the motor and will require the application to remove all other resources for this motor.
MEISynqNetResourceCommandDAC_ONE	Configures a 32bit data field of DAC data to besent to the motor.
MEISynqNetResourceCommandDAC_TWO	Configures two 32bit data fields of DAC data to besent to the motor.
MEISynqNetResourceCommandDEMAND_A	Configures one 16bit data field of Demand data to be sent to the motor.
MEISynqNetResourceCommandDEMAND_ABC	Configures three 16bit data fields of Demand data to be sent to the motor.

See Also

[meiSynqNetPacketConfigGet](#) | [meiSynqNetPacketConfigSet](#) | [MEISynqNetPacketCfgMotor](#) | [MEISynqNetPacketCfg](#) | [MEISynqNetPacketCfgNode](#)

MEISynqNetResourceIoBits

MEISynqNetResourceIoBits

```
typedef enum MEISynqNetResourceIoBits {
    MEISynqNetResourceIoBitsINVALID,
    MEISynqNetResourceIoBitsNONE,
    MEISynqNetResourceIoBits16,
    MEISynqNetResourceIoBits48,
} MEISynqNetResourceIoBits;
```

Description

The [SynqNetResourceIoBits](#) enumeration is a member of the MEISynqNetPacketCfgMotor configuration structure. The values are used to configure the number of I/O data fields sent to a motor. Valid values range from greater or equal to MEISynqNetResourceIoBitsFIRST and less than MEISynqNetResourceIoBitsLAST, but may be further limited by the available resources on the node. This is a generic enumeration that is used to configure the resource count of different types of motor and node I/O.

NOTE: (for Motor I/O only) Enabled motors always contain one fixed 16bit data field of dedicated I/O. This is not application configurable.

MEISynqNetResourceIoBitsINVALID	Non-valid bit count was detected.
MEISynqNetResourceIoBitsNONE	No I/O bits will be sent/received.
MEISynqNetResourceIoBits16	One data field of 16 I/O bits will be sent/received.
MEISynqNetResourceIoBits48	One data field of 48 I/O bits will be sent/received.

See Also

[meiSynqNetPacketConfigGet](#) | [meiSynqNetPacketConfigSet](#) | [MEISynqNetPacketCfgMotor](#) | [MEISynqNetPacketCfg](#) | [MEISynqNetPacketCfgNode](#) | [mpiMotorIoGet](#) | [mpiMotorIoSet](#)

MEISynqNetResourceMonitor

MEISynqNetResourceMonitor

```
typedef enum MEISynqNetResourceMonitor {  
    MEISynqNetResourceMonitorINVALID,  
    MEISynqNetResourceMonitorNONE,  
    MEISynqNetResourceMonitorAB,  
    MEISynqNetResourceMonitorABCD,  
} MEISynqNetResourceMonitor;
```

Description

The [SynqNetResourceMonitor](#) enumeration is a member of the MEISynqNetPacketCfgMotor configuration structure. The values are used to configure the number of Monitor data fields received from a motor. Valid values range from greater or equal to MEISynqNetResourceMonitorFIRST and less than MEISynqNetResourceMonitorLAST, but may be further limited by the available resources on the node.

MEISynqNetResourceMonitorINVALID	Non-valid bit count was detected.
MEISynqNetResourceMonitorNONE	No monitor data fields will be sent/received for this motor.
MEISynqNetResourceMonitorAB	Two 16bit fields of monitor data will be sent/received for this motor.
MEISynqNetResourceMonitorABCD	Four 16bit fields of monitor data will be sent/received for this motor.

See Also

MEISynqNetPacketCfg

MEISynqNetPacketCfg

```
typedef struct MEISynqNetPacketCfg {  
    MEISynqNetPacketCfgNode node[MEISynqNetMaxNODE_COUNT];  
} MEISynqNetPacketCfg;
```

Description

The [SynqNetPacketCfg](#) structure is used as a parameter to the meiSynqNetPacketConfigGet/Set methods. It contains the network packet configuration for all nodes found on the network. Only configurable packet data is represented in this structure.

NOTE: Fixed packet fields are NOT application configurable.

node	is an array of configurable packet structures for node data. Array indices 0 through MEISynqNetInfo.nodeCount are valid, with a maximum number of motors per node being defined by MEISynqNetMaxNODE_COUNT.

See Also

[meiSynqNetPacketConfigSet](#) | [meiSynqNetPacketConfigGet](#) | [MEISynqNetPacketCfgNode](#) | [MEISynqNetPacketCfgMotor](#) | [MEISynqNetPacketCfgEncoder](#)

MEISynqNetPacketCfgEncoder

MEISynqNetPacketCfgEncoder

```
typedef struct MEISynqNetPacketCfgEncoder {
    long feedbackCount;    /* per encoder */
    long captureCount;     /* per encoder */
    long compareCount;     /* per encoder */
} MEISynqNetPacketCfgEncoder;
```

Description

The **SynqNetPacketCfgEncoder** structure is a member of the MEISynqNetPacketCfgMotor configuration structure. It contains the network packet configuration for a single encoder found on a particular motor. Only configurable packet data is represented in this structure. Fixed packet fields are not application configurable.

Setting all structure members to zero (0x0) values will effectively disable the encoder on the motor. Disabling an encoder will NOT affect encoder numbering, however, encoders with higher index numbers must be disabled before using encoders with lower index numbers.

NOTE: A motor is considered incomplete if it has no encoders enabled.

feedbackCount	configures the number of 32-bit feedback data fields to be sent for an encoder. Valid numbers are zero thru MEISynqNetMaxEncoderFEEDBACK_COUNT , but may be further limited by the available resources on the node. NOTE: Setting this count to a value of zero (0x0) will affectively disable the encoder. All other encoder resources must also be set to NONE or zero (0x0).
captureCount	configures the number of 32-bit capture data fields to be sent for an encoder. Valid numbers are zero thru MEISynqNetMaxMotorCAPTURE_COUNT, but may be further limited by the available resources on the node. If feedbackCount is zero (0x0), then this value must also be zero. NOTE: all capture/compare resources on a motor rely on 2 fixed 32bit data fields for command and status information. These fixed fields are currently not configurable.

compareCount	configures the number of 32-bit compare data fields to be sent for an encoder. Valid numbers are zero thru MEISynqNetMaxMotorCOMPARE_COUNT, but may be further limited by the available resources on the node. If feedbackCount is zero (0x0), then this value must also be zero. NOTE: all capture/compare resources on a motor rely on 2 fixed 32bit data fields for command and status information. These fixed fields are currently not configurable.

See Also [meiSynqNetPacketConfigGet](#) | [meiSynqNetPacketConfigSet](#) | [MEISynqNetPacketCfgMotor](#) | [MEISynqNetPacketCfg](#) | [MEISynqNetPacketCfgNode](#)

MEISynqNetPacketCfgIo

MEISynqNetPacketCfgIo

```
typedef struct MEISynqNetPacketCfgIo {
    long    digitalInCount;
    long    digitalOutCount;
    long    analogInCount;
    long    analogOutCount;
} MEISynqNetPacketCfgIo;
```

Description The [SynqNetPacketCfgIo](#) structure configures the network data packets that contain general purpose I/O.

digitalInCount	Selects the number of 32bit words of digital input data to receive over the SynqNet network.
digitalOutCount	Selects the number of 32bit words of digital output data to send over the SynqNet network.
analogInCount	Selects the number of 32bit words of analog input data to receive over the SynqNet network.
analogOutCount	Selects the number of 32bit words of analog output data to send over the SynqNet network.

See Also [meiSynqNetPacketConfigSet](#) | [meiSynqNetPacketConfigGet](#) | [MEISynqNetPacketCfgNode](#)

MEISynqNetPacketCfgMotor

MEISynqNetPacketCfgMotor

```
typedef struct MEISynqNetPacketCfgMotor {
    MEISynqNetResourceCommand    command;
    long                          pulseEngineCount;
    long                          feedbackPrimaryCount;
    long                          captureCount;
    long                          compareCount;
    MEISynqNetResourceIoBits      ioInput;
    MEISynqNetResourceIoBits      ioOutput;
    MEISynqNetResourceMonitor     monitor;
    MEISynqNetPacketCfgProbe      probe;
} MEISynqNetPacketCfgMotor;
```

Description

The **SynqNetPacketCfgMotor** structure is a member of the MEISynqNetPacketCfgNode configuration structure. It contains the network packet configuration for a single motor found on a particular node. Only configurable packet data is represented in this structure. Fixed packet fields are not application configurable.

NOTE: Setting all structure members to a NONE or Zero (0x0) value will effectively disable the motor on the network. This will cause the controller to renumber the subsequent motors on the network.

command	Selects the command data type and count. Please see MEISynqNetResourceCommand .
pulseEngineCount	Configures the number of pulse engine to be enable for a motor. Valid numbers are zero thru MEISynqNetMaxMotorPULSE_ENGINE_COUNT, but may be further limited by the available resources on the node. Each pulse engine requires a 64-bit upstream data field and a 32-bit downstream data field.
feedbackPrimaryCount	Configures the number of 32-bit feedback data fields to be sent for a motor. Valid numbers are zero thru MEISynqNetMaxEncoderFEEDBACK_COUNT, but may be further limited by the available resources on the node.

captureCount	Configures the number of 32-bit capture data fields to be sent for a motor. Valid numbers are zero thru MEISynqNetMaxMotorCAPTURE_COUNT, but may be further limited by the available resources on the node. If feedbackCount is zero (0x0), then this value must also be zero. NOTE: All capture/compare resources on a motor rely on two fixed 32bit data fields for command and status information. These fixed fields are currently not configurable.
compareCount	Configures the number of 32-bit compare data fields to be sent for a motor. Valid numbers are zero thru MEISynqNetMaxMotorCOMPARE_COUNT, but may be further limited by the available resources on the node. NOTE: all capture/compare resources on a motor rely on two fixed 32bit data fields for command and status information. These fixed fields are currently not configurable.
ioInput	Selects the number of input bit data received from this motor. Please see MEISynqNetResourceIoBits .
ioOutput	Selects the number of output bit data sent to this motor. Please see MEISynqNetResourceIoBits .
monitor	Selects the number of montior data fields received from this motor. Please see MEISynqNetResourceMonitor .
probe	A structure that specifies the probe count and register depth.

See Also [meiSynqNetPacketConfigGet](#) | [meiSynqNetPacketConfigSet](#) | [MEISynqNetPacketCfg](#) | [MEISynqNetPacketCfgNode](#) | [MEISynqNetResourceCommand](#) | [MEISynqNetResourceIoBits](#) | [MEISynqNetResourceMonitor](#) | [MEISynqNetPacketCfgProbe](#)

MEISynqNetPacketCfgNode

MEISynqNetPacketCfgNode

```
typedef struct MEISynqNetPacketCfgNode {
    MEISynqNetPacketCfgMotor    motor[MEISynqNetMaxNodeMOTORS]
    MEISynqNetPacketCfgIo      io;
    long                       feedbackSecondaryCount;
} MEISynqNetPacketCfgNode;
```

Description

The [SynqNetPacketCfgNode](#) structure is a member of the MEISynqNetPacketCfg configuration structure. It contains the network packet configuration for all motors found on a particular node. Only configurable packet data is represented in this structure. Fixed packet fields are not application configurable.

motor	An array of configurable packet structures for motor data. Array indices 0 through MEISqNodeInfo.motorCount are valid, with a maximum number of motors per node being defined by MEISynqNetMaxNodeMOTORS .
io	Configures the network data packets general purpose node I/O.
feedbackSecondaryCount	Configures the number of 32-bit secondary feedback data fields to be sent for a node. Valid numbers are zero thru MEISynqNetMaxMotorFEEDBACK_SECONDARY_COUNT , but may be further limited by the available resources on the node.

See Also

[meiSynqNetPacketConfigGet](#) | [meiSynqNetPacketConfigSet](#) | [MEISynqNetPacketCfg](#) |

MEISynqNetPacketCfgProbe

MEISynqNetPacketCfgProbe

```
typedef struct MEISynqNetPacketCfgProbe {
    long
    MEISynqNetResourceProbeDepth
} MEISynqNetPacketCfgNode;
count;
depth[MEISynqNetMaxMotorPROBE_COUNT];
```

Description

The [SynqNetPacketCfgProbe](#) structure specifies the network packet configuration for all the Probes found on a particular motor. Only configurable packet data is represented in this structure. Fixed packet fields are not application configurable.

count	the number of Probe engines per motor. Each Probe engine requires cyclic packet data for the status fields.
depth	an array of enumerated values representing the Probe data depth. Each probe engine can support up to 16 register fields for the probe data. The length of the array is specified by the Probe count.

See Also

[MEISynqNetPacketCfgMotor](#) | [MEISynqNetResourceProbeDepth](#) | [MPIProbeStatus](#)

MEISynqNetResourceCfgProbeDepth

MEISynqNetResourceCfgProbeDepth

```
typedef enum MEISynqNetResourceProbeDepth {
    MEISynqNetResourceProbeDepthNONE,
    MEISynqNetResourceProbeDepthTWO,
    MEISynqNetResourceProbeDepthFOUR,
    MEISynqNetResourceProbeDepthEIGHT,
    MEISynqNetResourceProbeDepthSIXTEEN,
} MEISynqNetResourceProbeDepth;
```

Description

The [SynqNetPacketCfgProbeDepth](#) is an enumeration of the possible Probe register depths. Each Probe register is 16 bits. The packet data size is 32 bits. Each Probe engine can support up to 16 register fields for the Probe data.

MEISynqNetResourceProbeDepthNONE	zero Probe data registers will be transmitted upstream
MEISynqNetResourceProbeDepthTWO	2 Probe data registers will be transmitted upstream
MEISynqNetResourceProbeDepthFOUR	4 Probe data registers will be transmitted upstream
MEISynqNetResourceProbeDepthEIGHT	8 Probe data registers will be transmitted upstream
MEISynqNetResourceProbeDepthSIXTEEN	16 Probe data registers will be transmitted upstream

See Also

MEISynqNetState

MEISynqNetState

```
typedef enum MEISynqNetState {
    MEISynqNetStateINVALID,
    MEISynqNetStateDISCOVERY,
    MEISynqNetStateASYNQ,
    MEISynqNetStateSYNQ,
    MEISynqNetStateSYNQ_RECOVERING,
} MEISynqNetState;
```

Description

SynqNetState is an enumeration of the SynqNet network states. Each state shows the present operation mode for the network. Network data traffic occurs in two modes: asynchronous and synchronous.

During **asynchronous** communication, the MPI or controller sends one packet to one node and the node responds with one packet. There are no timing restrictions. Asynchronous communication is used during network initialization, reset, discovery, and node binary download.

During **synchronous** communication, the controller sends packets to the nodes and the nodes respond with packets. All data traffic is scheduled in fixed timeslots to avoid collisions. Packet data updates are cyclic and synchronized to the controller's sample rate. The network state can be read with `meiSynqNetStatus(...)`.

MEISynqNetStateDISCOVERY	This is the initial state before the network is initialized. During this phase, the controller checks the network integrity, determines the network topology, resets the nodes, identifies, initializes, and addresses the nodes. Data packets are sent/received asynchronously.
MEISynqNetStateASYNQ	This state is used for off-line operations, like node binary download. Data packets are sent/received asynchronously.
MEISynqNetStateSYNQ	This is the normal operating state. Data packets are sent/received synchronously.

MEISynqNetStateSYNQ_RECOVERING

During this state, a network fault condition was detected and the network is being reconfigured to route data packets around the fault. After the reconfiguration is complete and the packet error rate counters have decremented to zero, the network will return to the SYNQ state.

See Also [MEISynqNetStatus](#) | [meiSynqNetStatus](#)

MEISynqNetStatus

MEISynqNetStatus

```
typedef struct MEISynqNetStatus {
    MEISynqNetState           state;
    long                    topologySaved; /* TRUE/FALSE */
    MEISynqNetStatusCrcError crcError;
    MPIEventMask             eventMask;
    MEISynqNetFailedNodeMask failedNodeMask;
} MEISynqNetStatus;
```

Description

The [SynqNetStatus](#) structure contains network state information, CRC counters, eventMask, and the failedNodeMask for a SynqNet network. The network status can be read with the meiSynqNetStatus(...) method.

state	Present operation mode for the network.
topologySaved	Contains the status of the network topology. FALSE means the topology is dynamically discovered at initialization. TRUE means that the topology is verified against an expected topology at initialization. See Saving Current Topology To Flash for more information.
crcError	CRC error counters for the controller's network ports. The CRC value increments when received data is corrupt. Range is from 0 to 255. Counter saturates at 255.
eventMask	An array that defines the status for the event mask bits. The array is defined as: <pre>typedef MPIEventMaskELEMENT_TYPE MPIEventMask[MPIEventMaskELEMENTS]</pre> The bits are defined by the MPI/MEIEventType enumerations.
failedNodeMask	Array that defines the failed node mask bits. Each bit represents a failed node (0x1 = node 0, 0x2 = node 2, 0x4 = node 3, etc.). The array is defined as: <pre>#define MEISynqNetNodeMaskELEMENTS (1) typedef long MEISynqNetFailedNodeMask[MEISynqNetNodeMaskeLEMENTS];</pre>

See Also

[meiSynqNetStatus](#) | [meiSqNodeStatus](#) | [MEISqNodeStatus](#)

MEISynqNetStatusCrcError

MEISynqNetStatusCrcError

```
typedef struct MEISynqNetStatusCrcError {  
    long port[MEINetworkPortLAST];  
} MEISynqNetStatusCrcError;
```

Description The [SynqNetStatusCrcError](#) structure contains the CRC values for each network port on the controller. This structure is read via the `meiSynqNetStatus(...)` method.

port	an array of CRC values, one for each network port.

See Also [MEINetworkPort](#) | [MEISynqNetStatus](#) | [meiSynqNetStatus](#) | [CRC Error Counters](#)

MEISynqNetTiming

MEISynqNetTiming

```
typedef struct MEISynqNetTiming {
    double controllerFreq;      /* kHz */
    double controllerPeriod;    /* uS */

    long   txTime;               /* % */
    double calculationLimit;    /* uS */

    double driveUpdateFreq;     /* kHz */
    double driveUpdatePeriod;  /* uS */

    long   bandwidthUsage;      /* % */

    struct {
        double calculationTime; /* uS */
        double calculationSlack; /* uS */
        double demandLatency;   /* uS */
        double feedbackLatency; /* uS */

        double total;           /* uS */
        double latencySlack;    /* uS */
    } controlLatency;

    struct {
        double synq;            /* uS - synq packet + spacing */
        double demand;          /* uS - sum demand packets + spacing */
        double control;         /* uS - sum control packets + spacing */

        double total;           /* uS */
    } downstream;

    struct {
        double feedback;        /* uS - sum feedback packets + spacing */
        double status;          /* uS - sum status packets + spacing */

        double total;           /* uS */
    } upstream;
} MEISynqNetTiming;
```

Description

The **SynqNetTiming** structure contains static data that is determined during network initialization. It identifies timing statistics about the current network. Most values are just reported based on network timing calculations. The values that can be configured by the MPI are stated below as configurable.

controllerFreq	The sample rate calculation of the SynqNet controller board, in kHz. It is also the rate at which SynqNet packets are sent. The controllerFreq must be configured so that the driveUpdateFreq is an integer multiple of the controllerFreq. This value is configurable. See MPIControlConfig.
controllerPeriod	The sample period calculation of the SynqNet controller board, in uS. Derived from controllerFreq.
txTime	The scheduled time for SynqNet packets to be sent, in % (of the controllerPeriod). Defaults to 75%. Must be set to a value greater than the foreground calculation time, and less than 100%. This value is configurable. See MEIControlConfig.
calculationLimit	The maximum allowed foreground calculation time, in uS. Derived from txTime * controllerPeriod.
driveUpdateFreq	The cyclic communication rate of a drive processor on a SynqNet node, in kHz. Typically fixed to 16 kHz, but may vary depending on drive type. May be configurable on some drives. This rate often matches the drive PWM rate.
driveUpdatePeriod	The cyclic communication period of a drive processor on a SynqNet node, in uS. Derived from driveUpdateFreq.
bandwidthUsage	The amount of SynqNet bandwidth used, in %, for this SynqNet configuration. The actual packet payload configured is divided by the maximum available SynqNet bandwidth, for both upstream and downstream packets. The greater of the two is reported.
calculationTime	The foreground calculation time of the controller, in uS.
calculationSlack	The available slack time between the foreground calculation time, and the scheduled txTime, in uS.

demandLatency	The time to send SynqNet demand data downstream from controller to nodes, in uS. Does not include drive demand delays.
feedbackLatency	The time to send SynqNet feedback data upstream from nodes to controller, in uS. Rounded up to make the total SynqNet latency an integer multiple of the driveUpdatePeriod. Does not include drive feedback delays.
latencySlack	The slack time from the configured feedbackLatency (which is rounded up to an integer multiple of the driveUpdatePeriod), to the minimum possible upstream delay, in uS.
controlLatency	The overall SynqNet system control latency, in uS. Always rounded up to an integer multiple of driveUpdatePeriod. Control latency begins when position feedback is sampled on the node, includes upstream packet delays, controller foreground calculation, downstream packet delays, and ends when demands are strobed on the nodes. Note drive feedback and demand delays are NOT included. Control latency can be broken down into the four components listed below: feedbackLatency, calculationTime, calculationSlack, and demandLatency.
synq	Downstream SYNQ packet payload, in uS. Includes spacing between packets.
demand	Downstream DEMAND packet payload, in uS. Includes spacing between packets.
control	Downstream CONTROL packet payload, in uS. Includes spacing between packets.
downstream	Total downstream (controller to nodes) packet payload, in uS. Includes spacing between packets. Breaks down into the following three packet types.
feedback	Upstream FEEDBACK packet payload, in uS. Includes spacing between packets.
status	Upstream STATUS packet payload, in uS. Includes spacing between packets.
upstream	Total upstream (nodes to controller) packet payload, in uS. Includes spacing between packets. Breaks down into the following two packet types.

See Also [SynqNet Timing Values](#)

MEISynqNetTrace

MEISynqNetTrace

```
typedef enum {
    MEISynqNetTraceDYNA_ALLOC = MEISynqNetTraceFIRST << 0,
    MEISynqNetTraceDYNA_FREE = MEISynqNetTraceFIRST << 1,
    MEISynqNetTraceINIT = MEISynqNetTraceFIRST << 2,
    MEISynqNetTraceRESET = MEISynqNetTraceFIRST << 3,
    MEISynqNetTraceTIMING = MEISynqNetTraceFIRST << 4,
    MEISynqNetTraceSERVICE_CMD = MEISynqNetTraceFIRST << 5,
    MEISynqNetTraceFLOWCTRL = MEISynqNetTraceFIRST << 6,
    MEISynqNetTraceMAP = MEISynqNetTraceFIRST << 7,
    MEISynqNetTraceUNMAP = MEISynqNetTraceFIRST << 8,
} MEISynqNetTrace;
```

Description **SynqNetTrace** is an enumeration of SynqNet trace bits that can be used to enable/disable library trace statement output for the SynqNet object.

MEISynqNetTraceDYNA_ALLOC	Enables trace statements for controller external memory allocation during SynqNet initialization.
MEISynqNetTraceDYNA_FREE	Enables trace statements for controller external memory de-allocation.
MEISynqNetTraceINIT	Enables trace statements for SynqNet network initialization.
MEISynqNetTraceRESET	Enables trace statements for SynqNet network reset.
MEISynqNetTraceTIMING	Enables trace statements for SynqNet network timing calculations.
MEISynqNetTraceSERVICE_CMD	Enables trace statements for SynqNet service command transactions between the controller and SynqNet nodes.
MEISynqNetTraceFLOWCTRL	This trace bit enables trace statements for the SynqNet service command handshake between the controller and SynqNet nodes.
MEISynqNetTraceMAP	This trace bit enables trace statements for the mapping of firmware pointers to the dynamic controller memory.
MEISynqNetTraceUNMAP	This trace bit enables trace statements for the unmapping of firmware pointers from the dynamic controller memory.

See Also

[Trace.exe utility](#)

MEISynqNetCableHOP_COUNT

MEISynqNetCableHOP_COUNT

```
#define MEISynqNetCableHOP_COUNT (MEISynqNetMaxNODE_COUNT + 1)
```

Description

SynqNetMaxCableHOP_COUNT is the maximum number of cables for a SynqNet network. Presently, string and ring topologies are supported. The maximum number of cables is based on the maximum number of nodes, plus one return cable for ring topologies.

See Also

[MEISynqNetCableList](#) | [MEISynqNetConfig](#)

MEISynqNetMaxMotorFEEDBACK_PRIMARY_COUNT

MEISynqNetMaxMotorFEEDBACK_PRIMARY_COUNT

```
#define MEISynqNetMaxMotorFEEDBACK_PRIMARY_COUNT (1)
```

Description

SynqNetMaxMotorFEEDBACK_PRIMARY_COUNT defines the maximum number of primary feedback resources per motor.

NOTE: Feedback count may be further limited by the available resources on the node.

See Also

MEISynqNetMaxMotorFEEDBACK_SECONDARY_COUNT

MEISynqNetMaxMotorFEEDBACK_SECONDARY_COUNT

```
#define MEISynqNetMaxMotorFEEDBACK_SECONDARY_COUNT (MEISqNodeMaxFEEDBACK_SECONDARY)
```

Description

SynqNetMaxMotorFEEDBACK_SECONDARY_COUNT defines the maximum number of secondary feedback resources per motor.

NOTE: Feedback count may be further limited by the available resources on the node.

See Also

MEISynqNetMaxMotorCAPTURE_COUNT

MEISynqNetMaxMotorCAPTURE_COUNT

```
#define MEISynqNetMaxMotorCAPTURE_COUNT (MEIXmpMaxCapturesPerMotor) /* 2 */
```

Description

SynqNetMaxMotorCAPTURE_COUNT defines the maximum number of capture resources per motor.

NOTE: Capture count may be further limited by the available resources on the node.

This define should be used instead of the MEIXmpMaxCapturesPerMotor definition in xmp.h. It is recommended that applications avoid programming to defines or structures in xmp.h

See Also

MEISynqNetMaxMotorCOMPARE_COUNT

MEISynqNetMaxMotorCOMPARE_COUNT

```
#define MEISynqNetMaxMotorCOMPARE_COUNT (MEIXmpMaxCapturesPerMotor) /* 2 */
```

Description

SynqNetMaxMotorCOMPARE_COUNT defines the maximum number of compare resources per motor.

NOTE: Compare count may be further limited by the available resources on the node.

This define should be used instead of the MEIXmpMaxComparesPerMotor definition in xmp.h. It is recommended that applications avoid programming to defines or structures in xmp.h

See Also

MEISynqNetMaxMotorENCODER_COUNT

MEISynqNetMaxMotorENCODER_COUNT

```
#define MEISynqNetMaxMotorENCODER_COUNT (MEIXmpMotorEncoders) /* 2 */
```

Description

SynqNetMaxMotorENCODER_COUNT defines the maximum number of encoder resources per motor.

NOTE: The encoder count may be further limited by the available resources on the node.

This define should be used instead of the MEIXmpMotorEncoders definition in xmp.h. It is recommended that applications avoid programming to defines or structures in xmp.h

See Also

MEISynqNetMaxMotorPULSE_ENGINE_COUNT

MEISynqNetMaxMotorPULSE_ENGINE_COUNT

```
#define MEISynqNetMaxMotorPULSE_ENGINE_COUNT (1)
```

Description **SynqNetMaxMotorPULSE_ENGINE_COUNT** defines the number of pulse engines per motor.

See Also

MEISynqNetMaxMOTORS

MEISynqNetMaxMOTORS

```
#define MEISynqNetMaxMOTORS (MEIXmpMAX_Motors) /* 32 */
```

Description

SynqNetMaxMOTORS defines the maximum number of motors supported on a single SynqNet network.

See Also

MEISynqNetMaxNodeMOTORS

MEISynqNetMaxNodeMOTORS

```
#define MEISynqNetMaxNodeMOTORS (MEISqNodeMaxMOTORS)
```

Description

SynqNetMaxNodeMOTORS defines the maximum number of motor objects supported on a single SynqNet node.

See Also

MEISynqNetMaxNODE_COUNT

MEISynqNetMaxNODE_COUNT

```
#define MEISynqNetMaxNODE_COUNT (MEIXmpMaxSynqNetBlocks) /* 32 */
```

Description

SynqNetMaxNODE_COUNT defines the maximum number of nodes supported on a single SynqNet network. This define should be used instead of the MEIXmpMaxSynqNetBlocks definition in xmp.h. It is recommended that applications avoid programming to defines or structures in xmp.h

See Also

MEISynqNetNodeMaskELEMENTS

MEISynqNetNodeMaskELEMENTS

```
#define MEISynqNetNodeMaskELEMENTS (1)
typedef long MEISynqNetFailedNodeMask [MEISynqNetNodeMaskELEMENTS];
```

Description [SynqNetNodeMaskELEMENTS](#) defines the number of data elements in an MEISynqNetFailedNodeMask.

See Also [MEISynqNetFailedNodeMask](#)

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 **XMPI\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.

NOTE: **Debug** and **DebugSingle** are the only MPI library configurations that will produce trace output.

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.

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

meiTraceEol	Set the end-of-line character to be used by Trace
meiTraceFile	Send trace output to a file
meiTraceFunction	sets function used to display a trace buffer
meiTraceGet	Get global trace mask
meiTraceMaskBits	Convert the trace mask into an array of trace bits.
meiTraceMsg	Convert the message identification value into a string.
meiTraceMsgFunction	Set a module's trace message function.
meiTraceSet	Set global trace mask

Data Types

[MEITrace](#)
[MEITraceFunction](#)
[MEITraceMask](#)
[MEITraceMsgFunction](#)

Constants

[MEITraceMaskGLOBAL](#)

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 <i>TraceFile</i> 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](#)

meiTraceMaskBits

Declaration long **meiTraceMaskBits**(long **mask**,
 long ***bitCount**,
 long ***bit**);

Required Header stdmei.h

Description **TraceMaskBits** converts the trace mask into an array of trace bits and the length of the array.

mask	A bit mask of enumerated trace bits.
*bitCount	A pointer to a long containing the number of trace bits enabled in the mask. This value is also the length of the bit array.
*bit	A pointer to an array of longs containing the enumerated trace bits. Each array member contains one trace bit enumerated value.

Returns

See Also [MEITrace](#) | [meiTraceGet](#) | [meiTraceSet](#)

meiTraceMsg

Declaration `const char * meiTraceMsg( long messageId, char *messageText );`

Required Header `stdmei.h`

Description **TraceMsg** converts the message identification value into a string pointed to by *messageText*.

messageId	a message identification value.
*messageText	a pointer to a character string containing the text for the messageId.

Returns

See Also [meiTraceMsgFunction](#)

meiTraceMsgFunction

Declaration

long **meiTraceMsgFunction**([MPIModuleId](#) **moduleId**,
[MEITraceMsgFunction](#) **function**);

Required Header

stdmei.h

Description

TraceMsgFunction sets a module's trace message function.

moduleId	an enumerated module identification value
function	a pointer to a trace message function.

Returns

See Also

meiTraceSet

Declaration [MEITraceMask](#) **meiTraceSet**([MEITraceMask](#) mask)

Required Header stdmei.h

Description [TraceSet](#) sets the global trace mask to *mask*.

<i>If "traceMask" is</i>	<i>Then</i>
MEITraceALL	all global categories of trace will be enabled
MEITraceNONE	all categories of trace will be disabled

Returns	the value of the previous <i>global trace mask</i>

See Also [meiTraceGet](#)

MEITrace

MEITrace

```
typedef enum {
    MEITraceNONE = 0,
    MEITraceFIRST = 0x0001,
    MEITraceFUNCTION_ENTRY = (int) MEITraceFIRST << 0,
    MEITraceFUNCTION_RETURN = (int) MEITraceFIRST << 1,
    MEITraceMEMORY_ALLOC = (int) MEITraceFIRST << 2,
    MEITraceMEMORY_FREE = (int) MEITraceFIRST << 3,
    MEITraceMEMORY_GET = (int) MEITraceFIRST << 4,
    MEITraceMEMORY_SET = (int) MEITraceFIRST << 5,
    MEITraceVALIDATE = (int) MEITraceFIRST << 6,
    MEITraceLOCK_GIVE = (int) MEITraceFIRST << 7,
    MEITraceLOCK_TAKE = (int) MEITraceFIRST << 8,
    MEITraceEVENT = (int) MEITraceFIRST << 9,

    MEITraceALL = (int) ((MEITraceLAST << 1) - 1)
} MEITrace;
```

Description **Trace** is an enumeration of generic trace bits that can be used to enable/disable library trace statement output for objects throughout the MPI.

MEITraceFUNCTION_ENTRY	Trace the entry into all methods.
MEITraceFUNCTION_RETURN	Trace the return from all methods.
MEITraceMEMORY_ALLOC	Enables trace statements for all host memory allocations.
MEITraceMEMORY_FREE	Enables trace statements for all host memory de-allocations.
MEITraceMEMORY_GET	Enables trace statements for all controller memory reads.
MEITraceMEMORY_SET	Enables trace statements for all controller memory writes.
MEITraceVALIDATE	Enables trace statements for all function parameter validations.
MEITraceLOCK_GIVE	Enables trace statements for all IPC lock releases.
MEITraceLOCK_TAKE	Enables trace statements for all IPC lock takes.
MEITraceEVENT	Enables trace statements for all MPI Events.

See Also [Trace Object](#) | [Trace.exe utility](#)

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](#)

MEITraceMsgFunction

MEITraceMsgFunction

```
typedef long(*MEITraceMsgFunction)(const char *buffer);
```

Description

TraceMsgFunction is the type definition for the callback function used by `meiTrace(...)`. A default callback function is provided internally to all MPI/MEI modules, but an application can also be written to override it and provide a custom message function instead.

See Also

[meiTraceMsgFunction](#) | [MEITrace](#)

MEITraceMaskGLOBAL

MEITraceMaskGLOBAL

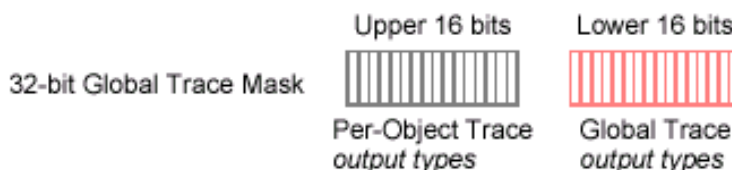
```
extern MEITraceMask MEITraceMaskGLOBAL;
```

Description **TraceMaskGlobal** is a non-object specific MPI Trace mask variable used for library wide Trace bits.

See Also [MEITraceMask](#)

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](#)

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](#)

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.

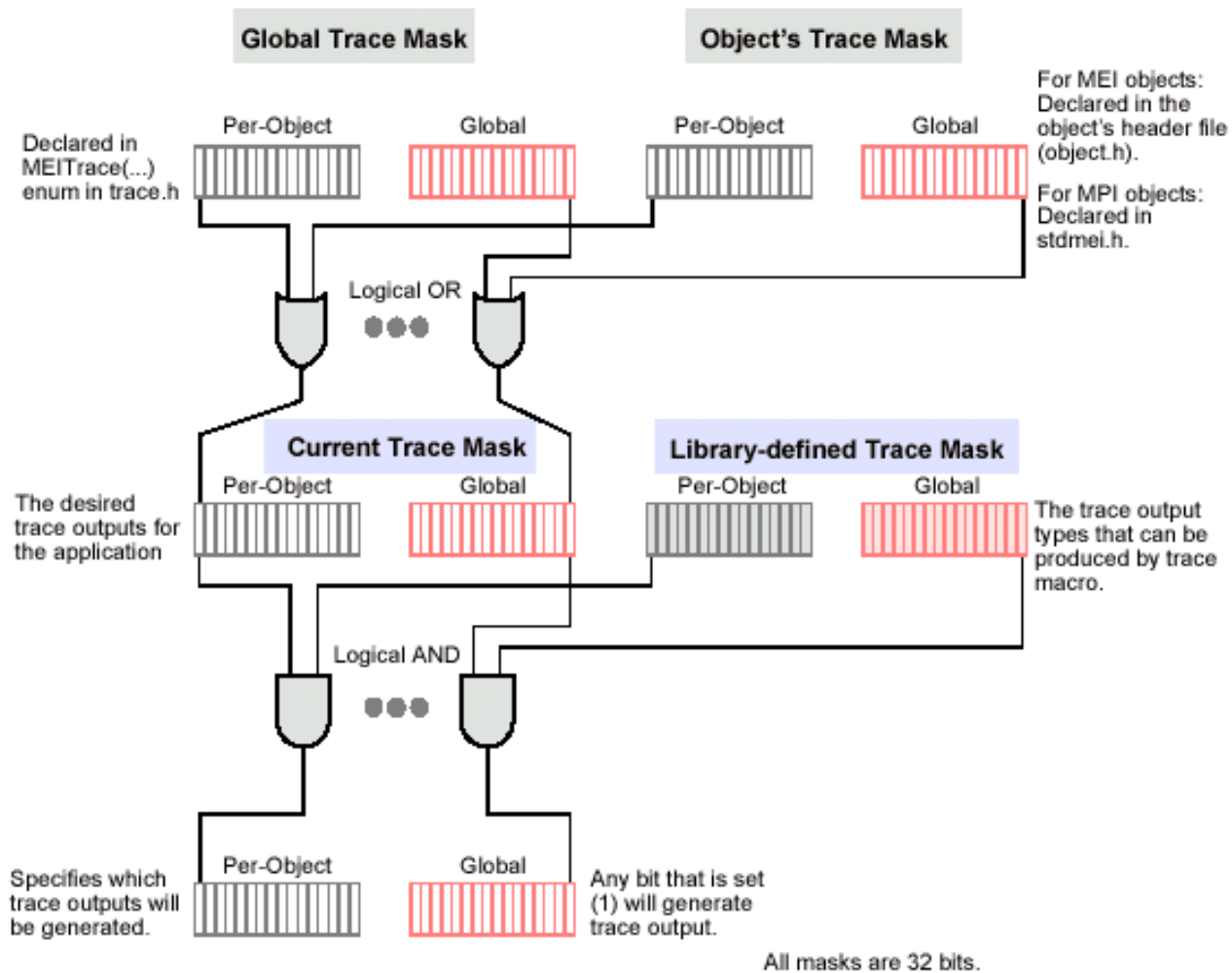
An object will produce trace output for a trace category when the logical **OR** of the **global trace mask** and the **object's trace mask** has the bit set that corresponds to the trace category.

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 **ORed** 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 **ANDed** 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](#)

Xmp Module

Introduction

The **Xmp** module provides the low-level interface between the controller and the MPI library. It defines the shared memory access between the controller's processor and the host CPU. It also contains several hardware constants and maximum values for resources. ALL data transactions between the MPI and controller are defined by this module.

The MPI provides a layer between the application code and the controller. It protects the application from *xmp.h* changes, hides controller complexity, handles semaphore locking, performs data validation and range checking, plus many other features. Normally, an application does not need the Xmp module. An application should ALWAYS use the MPI to access the controller. In some cases, the MPI may not have methods/structures to support a controller feature. For example, a custom feature may require support for direct access to the MPI, but it has not been yet been developed. In these cases, an application can use *xmp.h* and *mpiControlMemoryGet/Set(...)* to directly access the controller.

WARNING!

The *xmp.h* file is version dependent. Make sure to ONLY use the *xmp.h* that was included with the MPI and controller firmware software package. Using mismatched *xmp.h* defines can cause unexpected and potentially dangerous behavior!

Be aware that the *xmp.h* is always changing. It is an internal file, used by the controller firmware and MPI. It is optimized for memory allocation and performance for the controller's processor. As new features are developed and improved, the *xmp.h* is modified. If you use *xmp.h* defines in your application code, make sure to check for changes when upgrading to new software releases.

Data Types

[MEIXmpSwitchType](#)

MEIXmpSwitchType

MEIXmpSwitchType

```
typedef enum {
    MEIXmpSwitchTypeNONE,
    MEIXmpSwitchTypeMOTION_ONLY,
    MEIXmpSwitchTypeWINDOW,
    MEIXmpSwitchTypeUSER,
} MEIXmpSwitchType;
```

Description

XmpSwitchType is an enumeration for gain scheduling that determines the gain scheduling mode. Only MEIXmpSwitchTypeNONE and MEIXmpSwitchTypeMOTION_ONLY are available in standard firmware types.

Gain Scheduling is a feature that switches filter gains for the acceleration, deceleration, constant velocity, and idle states of motion. The post filters are not affected by gain scheduling. Standard algorithms are used with gain scheduling (PID, PIV). To change the gain scheduling type from *none* (uses only the gains in gain table index 0), use [MEIFilterConfig](#). GainSwitchType is set with [mpiFilterConfigSet\(...\)](#).

When setting filter gain parameters using [mpiFilterGainGet\(...\)](#) and [mpiFilterGainSet\(...\)](#), use the gain index value to write to a gain index of your choosing.

MEIXmpSwitchTypeNONE	Default value in factory default firmware. This mode uses mpiFilterGainIndexSet() and mpiFilterGainIndexGet() to manipulate the gain index manually, if desired.
MEIXmpSwitchTypeMOTION_ONLY	Switch gains based on controller's switching algorithm.

MEIFilterGainIndex (go to [MEIFilterGainIndex](#))

MEIFilterGainIndexNO_MOTION	When command velocity = 0
MEIFilterGainIndexACCEL	When command acceleration > 0
MEIFilterGainIndexDECEL	When command acceleration < 0
MEIFilterGainIndexVELOCITY	When command velocity > 0 and command acceleration = 0 The firmware automatically takes care of this switching. Be aware when checking the gain index, that the firmware can change the gain index in real time.

Description

Gain Scheduling is a feature that switches filter gains for the acceleration, deceleration, constant velocity, and idle states of motion. The post filters are not affected by gain scheduling. Standard algorithms are used with gain scheduling (PID, PIV). To change the gain scheduling type from NONE (uses only the gains in gain table index 0), use [MEIFilterConfig](#).GainSwitchType, which is set with [mpiFilterConfigSet\(...\)](#).

Use [mpiFilterConfigSet\(...\)](#) to change MEIFilterConfig.GainSwitchType to one of the MEIXmpSwitchType enumerations to change the gain scheduling mode.

See Also

[MPIFilterConfig](#) | [mpiFilterConfigGet](#) | [mpiFilterConfigSet](#) | [MEIFilterGainIndex](#) | [mpiFilterGainIndexSet](#) | [mpiFilterGainIndexGet](#) | [mpiFilterGainGet](#) | [mpiFilterGainSet](#)