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.

| Error Messages |

Methods

Create, Delete, Validate Methods

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

Configuration and Information Methods

mpiAxis**ActualPositionGet**	Get actual position
mpiAxis**ActualPositionGet32**	Gets the lower 32 bits of actual position
mpiAxis**ActualPositionSet**	Set actual position
mpiAxis**ActualVelocity**	Get actual velocity
mpiAxis**ConfigGet**	Get Axis configuration
mpiAxis**ConfigSet**	Set Axis configuration
mpiAxis**CommandPositionGet**	Get command position
mpiAxis**CommandPositionGet32**	Gets the lower 32 bits of command position
mpiAxis**CommandPositionSet**	Set command position
mpiAxis**FlashConfigGet**	Get Axis flash config
mpiAxis**FlashConfigSet**	Set Axis flash config
mpiAxis**FlashOriginGet**	
mpiAxis**FlashOriginSet**	
meiAxis**FrameBufferStatus**	
mpiAxis**OriginGet**	Get Axis origin

mpiAxis**OriginSet**	Set Axis origin
mpiAxis**PositionError**	Get position error of an Axis
mpiAxis**Status**	Get Axis status
mpiAxis**Trajectory**	Get Axis trajectory

Event Methods

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

Memory Methods

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

Relational Methods

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

Data Types

MPIAxis**Config** / MEIAxis**Config**
MPIAxis**EstopModify**
MEIAxis**FrameBufferStatus**
MPIAxis**InPosition**
MPIAxis**Master**
MPIAxis**MasterType**
MPIAxis**Message**

MEI**PreFilter**
MEI**PreFilterForm**

Constants

MEI**PreFilterCoeffsMAX**

MEI**PreFilterCountMAX**

mpiAxisCreate

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisCreate 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	

See Also

mpiAxisDelete | mpiAxisValidate

mpiAxisDelete

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisDelete 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	

See Also

mpiAxisCreate | mpiAxisValidate

mpiAxisValidate

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisValidate 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	

See Also

mpiAxisCreate | mpiAxisDelete

mpiAxisActualPositionGet

Declaration

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

Required Header: stdmpi.h

Description

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

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

Return Values	
MPIMessageOK	

See Also

AxisCommandPositionSet | Using the Origin Variable

mpiAxisActualPositionGet32

Declaration

```
long mpiAxisActualPositionGet32(MPIAxis  axis,
                                double   *actual)
```

Required Header: stdmpi.h
Change History: Added in the 03.04.00

Description

mpiAxisActualPositionGet32 gets the lower 32 bits of the actual position of an Axis (***axis***) and puts it in the location pointed to by ***actual***. The command and actual positions are stored in the controller as 64 bit values (2x 32bit words). Use mpiAxisActualPositionGet(…) to read the full position value. Internally, the MPI performs several reads and operations to transfer the full 64 bit position value. For applications that need optimum performance and if the position range is less than 32 bits, then use mpiAxisActualPositionGet32(…).

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

Return Values	
MPIMessageOK	

See Also

mpiAxisActualPositionGet | AxisCommandPositionSet | Using the Origin Variable

mpiAxisActualPositionSet

Declaration

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

Required Header: stdmpi.h

Description

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

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

Return Values	
MPIMessageOK	

See Also

AxisCommandPositionSet | Using the Origin Variable | Controller Positions

mpiAxisActualVelocity

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisActualVelocity 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	

See Also

mpiAxisConfigGet

Declaration

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

Required Header: stdmpi.h
Change History: Modified in the 03.03.00

Description

mpiAxisConfigGet 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 an Axis object.
***config**	pointer to the MPIAxisConfig structure
***external**	pointer to an external. See remarks below

Return Values	
MPIMessageOK	

Remarks

For XMP and ZMP 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);
}
```

See Also

MPIAxisConfig | mpiAxisConfigSet | MEIAxisConfig

mpiAxisConfigSet

Declaration

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

Required Header: stdmpi.h
Change History: Modified in the 03.03.00

Description

mpiAxisConfigSet 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 an Axis object.
***config**	pointer to the MPIAxisConfig structure
***external**	pointer to an external. See remarks below

Return Values	
MPIMessageOK	

Remarks

For XMP and ZMP 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);
}
```

See Also

mpiAxisConfigGet | MEIAdcConfig | MEIAxisConfig

mpiAxisCommandPositionGet

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisCommandPositionGet 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 an Axis object.
***command**	a pointer to the Axis command position returned by the method

Return Values	
MPIMessageOK	

See Also

mpiAxisCommandPositionSet | Controller Positions

mpiAxisCommandPositionGet32

Declaration

```
long mpiAxisCommandPositionGet32(MPIAxis   axis,
                                 double    *command)
```

Required Header: stdmpi.h
Change History: Added in the 03.04.00

Description

mpiAxisCommandPositionGet32 gets the lower 32 bits of the command position of an Axis (***axis***) and puts it in the location pointed to by ***command***. The command and actual positions are stored in the controller as 64 bit values (2x 32bit words). Use mpiAxisCommandPositionGet(…) to read the full position value. Internally, the MPI performs several reads and operations to transfer the full 64 bit position value. For applications that need optimum performance and if the position range is less than 32 bits, then use mpiAxisCommandPositionGet32(…).

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

Return Values	
MPIMessageOK	

See Also

mpiAxisCommandPositionGet | mpiAxisCommandPositionSet | Controller Positions

mpiAxisCommandPositionSet

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisCommandPositionSet sets the value of the command position of an Axis (***axis***) from ***command***. The motor will servo directly to the new command position the next servo sample after the new command position is set. This change in position will not be gradual or controlled, as it is in an mpiMotionStart(...). Use mpiMotionStart(...) and/or mpiMotionModify(...) for controlled, gradual motion.

mpiAxisCommandPositionSet truncates ***command*** to an integer value before sending the new position to the controller. If a different type of rounding is desired, then it should be implemented by the application prior to calling **AxisCommandPositionSet**.

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.

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

Remarks

Setting the Axis Command Position will cause the axis to jump. See the discussion of the Axis Origin before using the mpiAxisActualPositionSet(...) and mpiAxisCommandPositionSet(...) methods. The origin is not changed when mpiAxisCommandPositionSet(...) is called. The change is made directly to the command position.

Return Values	
MPIMessageOK	
MPIMessageARG_INVALID	if ***command*** lies outside the range of signed 32-bit integers: [-2147483648, 2147483647]
MPIAxisMessageCOMMAND_NOT_SET	

See Also

MEIMotorDisableAction | AxisActualPositionSet | AxisCommandPositionSet | MPIAxisMessage | Controller Positions

mpiAxisFlashConfigGet

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisFlashConfigGet 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 ***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**	a handle to the Flash object
***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	

See Also

MEIFlash | mpiAxisFlashConfigSet | MEIAxisConfig

mpiAxisFlashConfigSet

Declaration

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

Required Header: stdmpi.h

Description

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

Remarks

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	

See Also

MEIFlash | mpiAxisFlashConfigGet | MEIAxisConfig

mpiAxisFlashOriginGet

Declaration

```
long mpiAxisFlashOriginGet(MPIAxis   axis,
                           void      *flash,
                           double    *origin)
```

Required Header: stdmpi.h
Change History: Added in the 03.04.00

Description

mpiAxisFlashOriginGet reads the flash value of the origin for an Axis and writes it into the location pointed to by ***origin***. The ***flash*** origin value is useful for applying an offset value to the controller's actual position with absolute feedback devices.

axis	a handle to the Axis object.
***flash**	***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally. Using MPIHandleVOID is recommended, as it simplifies code. If ***flash*** 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.
***origin**	pointer to the Origin value returned by the method.

Return Values	
MPIMessageOK	

See Also

mpiAxisFlashOriginSet | mpiAxisOriginGet | Using the Origin Variable | Controller Positions

mpiAxisFlashOriginSet

Declaration

```
long mpiAxisFlashOriginSet(MPIAxis   axis,
                           void      *flash,
                           double    origin)
```

Required Header: stdmpi.h
Change History: Added in the 03.04.00

Description

mpiAxisFlashOriginSet writes the flash origin for an Axis using the origin value. The flash origin value is useful for applying an offset value to the controller's actual position with absolute feedback devices.

axis	a handle to the Axis object.
***flash**	***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally. Using MPIHandleVOID is recommended, as it simplifies code. If ***flash*** 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.
origin	a value to which the Axis origin will be set.

Return Values	
MPIMessageOK	

See Also

mpiAxisFlashOriginGet | mpiAxisOriginSet | Using the Origin Variable | Controller Positions

meiAxisFrameBufferStatus

Declaration

```
long  meiAxisFrameBufferStatus(MPIAxis                    axis,
                               MEIAxisFrameBufferStatus   *status);
```

Required Header: stdmpi.h
Change History: Added in the 03.04.00

Description

meiAxisFrameBufferStatus reads an axis's frame buffer status and writes it into the structure pointed to by ***status***.

axis	a handle to the Axis object.
status	a pointer to the frame buffer status structure returned by the method.

Return Values	
MPIMessageOK	
MPIMessageARG_INVALID	
MPIMessageHANDLE_INVALID	

Sample Code

```
void printAxisFrameBufferInfo(MPIAxis axis)
{
   MEIAxisFrameBufferStatus status;
   long returnValue = meiAxisFrameBufferStatus(axis,
                                               &status);
   msgCHECK(returnValue);

   printf("Size of frame buffer: %d\n", status.size);
   printf("Number of remaining frames: %d\n", status.frameCount);
   printf("Number of free frames %d\n",(status.size - status.
frameCount));
}
```

See Also

MEIAxisFrameBufferStatus

mpiAxisOriginGet

Declaration

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

Required Header: stdmpi.h

Description

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

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

Return Values	
MPIMessageOK	

See Also

mpiAxisOriginSet | Using the Origin Variable | Controller Positions

mpiAxisOriginSet

Declaration

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

Required Header: stdmpi.h

Description

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

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

Return Values	
MPIMessageOK	

See Also

mpiAxisOriginGet | Using the Origin Variable | Controller Positions

mpiAxisPositionError

Declaration

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

Required Header: stdmpi.h

Description

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

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

Return Values	
MPIMessageOK	

See Also

mpiAxisCommandPositionGet | mpiAxisActualPositionGet | Controller Positions

mpiAxisStatus

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisStatus 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.

Remarks

external should always be set to NULL.

Return Values	
MPIMessageOK	
MPIMessageARG_INVALID	

See Also

mpiAxisCommandPositionGet | mpiAxisActualPositionGet | Controller Positions

mpiAxisTrajectory

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisTrajectory reads the current velocity and acceleration of ***axis*** and writes it into the structure pointed to by ***trajectory***.

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

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

Remarks

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

Sample Code

```
MPITrajectory trajectory;

    mpiAxisTrajectory(axis, &trajectory);

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

Return Values	
MPIMessageOK	

See Also

mpiMotionStart | mpiMotionModify | MPITrajectory

mpiAxisEventNotifyGet

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisEventNotifyGet 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

Remarks

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	

See Also

MEIEventNotifyData | MEIEventStatusInfo | mpiAxisEventNotifySet

mpiAxisEventNotifySet

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisEventNotifySet 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

Remarks

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

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

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

Return Values	
MPIMessageOK	

See Also

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

mpiAxisEventReset

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisEventReset resets the event(s) that are specified in ***eventMask*** and generated by ***axis***. Your application must call mpiAxisEventReset only after one or more latchable events have occurred.

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

Return Values	
MPIMessageOK	

See Also

mpiControlEventReset | mpiMotionEventReset | mpiMotorEventReset | mpiRecorderEventReset | mpiSequenceEventReset | meiSynqNetEventReset | meiSqNodeEventReset

Event Notification Methods

mpiAxisMemory

Declaration

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

Required Header: stdmpi.h

Description

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

Return Values	
MPIMessageOK	

See Also

mpiAxisMemoryGet | mpiAxisMemorySet

mpiAxisMemoryGet

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisMemoryGet 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	

See Also

mpiAxisMemory | mpiAxisMemorySet

mpiAxisMemorySet

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisMemorySet 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	

See Also

mpiAxisMemory | mpiAxisMemoryGet

mpiAxisControl

Declaration

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

Required Header: stdmpi.h

Description

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

axis	a handle to an Axis object.

Return Values	
MPIHandleVOID	

See Also

mpiAxisFilterMapGet

Declaration

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

Required Header: stdmpi.h

Description

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

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

Remarks

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

Return Values	
MPIMessageOK	

See Also

mpiAxisFilterMapSet

mpiAxisFilterMapSet

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisFilterMapSet 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

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

Return Values	
MPIMessageOK	

See Also

mpiAxisFilterMapGet | MPIObjectMap

mpiAxisMotorMapGet

Declaration

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

Required Header: stdmpi.h

Description

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

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

Remarks

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

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

Return Values	
MPIMessageOK	

See Also

mpiAxisFilterMapGet | MPIObjectMap

mpiAxisNumber

Declaration

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

Required Header: stdmpi.h

Description

mpiAxisNumber 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	

See Also

MPIAxisConfig / MEIAxisConfig

Definition: MPIAxisConfig

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

Change History: Modified in the 03.03.00.

Description

estopModify	See MPIAxisEstopModify.
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.

Definition: MEIAxisConfig

```
typedef struct MEIAxisConfig {
    char                 userLabel[MEIObjectLabelCharMAX+1];
                            /* +1 for NULL terminator */
    long                 *FeedbackDeltaPtr[MEIXmpAxisPosInputs];
    MEIXmpAxisPreFilter  PreFilter;
    MEIXmpAxisGear       Gear;
    MEIXmpAxisGantryType GantryType;
}MEIAxisConfig;
```

Change History: Modified in the 03.04.00. Modified in the 03.03.00.

Description

userLabel - consists of 16 characters that are used to label the axis object for user identification purposes. The userLabel field is NOT used by the controller.

***FeedbackDeltaPtr** - Pointer to the position feedback delta, which is the difference in the feedback position between two sample periods, calculated by the controller.

PreFilter

- 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

GantryType -

```
typedef enum {
   MEIXmpAxisGantryTypeNONE = 0,
   MEIXmpAxisGantryTypeLINEAR = 1,
   MEIXmpAxisGantryTypeTWIST = 2
} MEIXmpAxisGantryType;
```

- **MEIXmpAxisGantryTypeNONE** - is the default. No gantry enabled.
- **MEIXmpAxisGantryTypeLINEAR** - is used to add the axis' two feedback values.
- **MEIXmpAxisGantryTypeTWIST** - is used to subtract the axis' two feedback values.

See Also

mpiAxisConfigGet | mpiAxisConfigSet | MPIAxisInPosition

MPIAxisEstopModify

Definition

```
typedef struct MPIAxisEstopModify {
    float    deceleration;
    float    decelerationJerk;
    float    jerkPercent;
} MPIAxisEstopModify
```

Change History: Added in the 03.03.00

Description

MPIAxisEstopModify is used with mpiAxisConfigGet(...) and mpiAxisConfigSet(...) as part of the MPIAxisConfig structure. This structure can be used to configure the deceleration and jerk applied to a motor when an EStopModify event occurs.

Any of the limits can be configured to generate an EStopModify instead of a standard EStop (a standard EStop stops the motor by stepping down the feedrate until it reaches 0.0).

See mpiMotorEventConfigSet(...) and mpiMotorEventConfigGet(...).

NOTE: Standard firmware uses jerkPercent and does not support decelerationJerk. See the MPITrajectory structure documentation.

deceleration	Specifies the Deceleration applied to the motor when an EStopModify occurs. Units are counts/sec^2.
decelerationJerk	Specifies the Jerk applied to the motor when an EstopModify occurs. Units are counts/sec^3. Jerk moves specified in counts/sec^3 are not supported in the standard firmware. Please contact MEI if this feature is required in your application.
jerkPercent	Specifies the JerkPercent applied to the motor when an EstopModify occurs. Units are in percent. Range is 0.0 to 100.0.

See Also

MPIAxisConfig | mpiMotorEventConfigSet | mpiMotorEventConfigGet | MPIAction

MEIAxisFrameBufferStatus

Definition

```
typedef struct MEIAxisFrameBufferStatus {
    long   size;
    long   frameCount;
} MEIAxisFrameBufferStatus;
```

Change History: Added in the 03.04.00.

Description

MEIAxisFrameBufferStatus provides status information of the frame buffer for a specified axis.

size	The value specifies the size (maximum number of frames) of the frame buffer for the given axis. This value is controlled by mpiControlConfigGet /Set under axisFrameCount. The default size the frame buffer (axisFrameCount) is 128.
frameCount	The value is equal to the number of frames on the controller that still need to be executed. At the default buffer size (128 frames), the range for frameCount is 0 to 127.

See Also

meiAxisFrameBufferStatus | mpiControlConfigGet | mpiControlConfigSet

MPIAxisInPosition

Definition

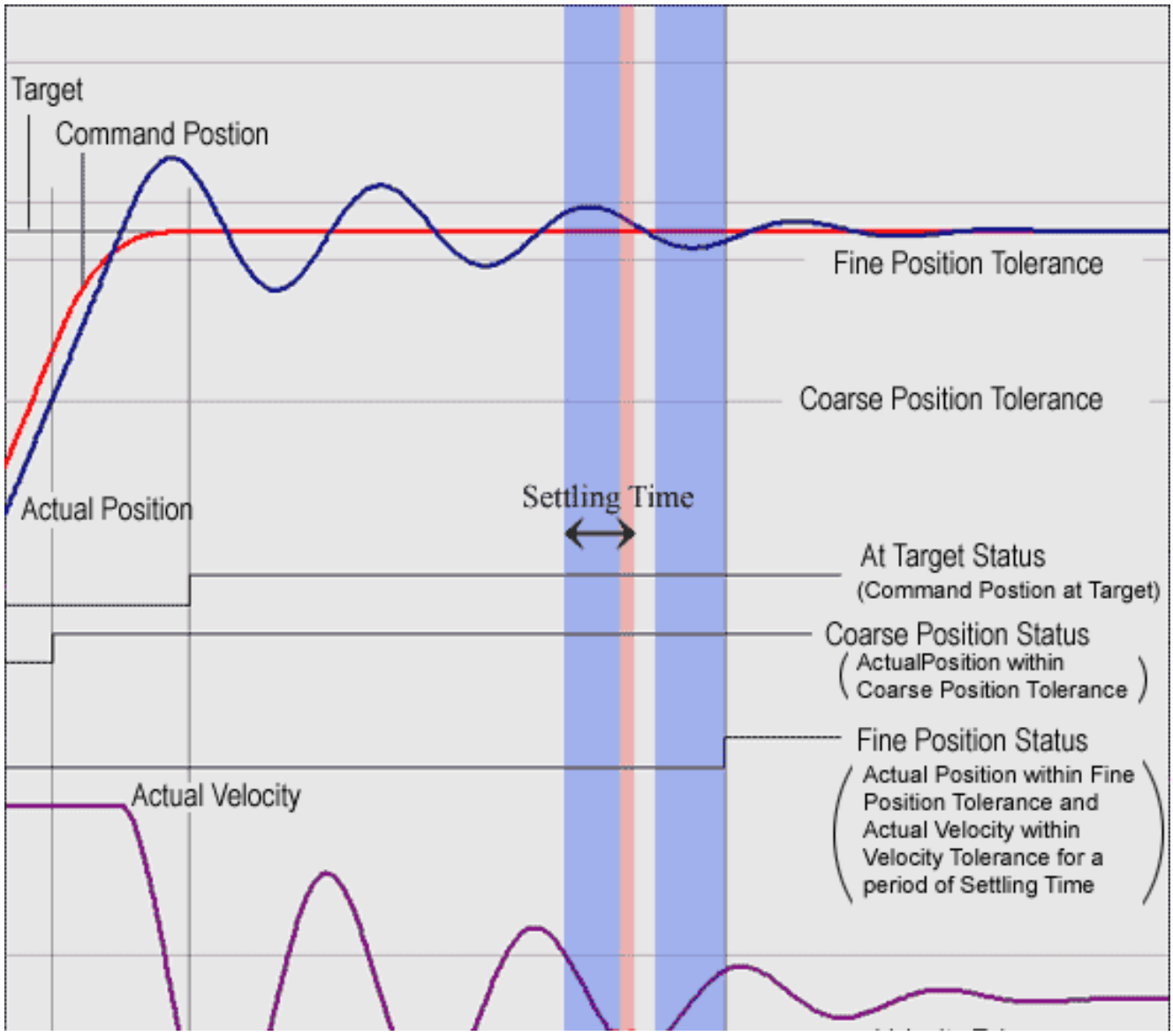
```
typedef struct MPIAxisInPosition {
    struct {
        float   positionFine;
        long    positionCoarse;
        float   velocity;
    } tolerance;
        float   settlingTime;  /* seconds */
        long    settleOnStop;
        long    settleOnEstop
        long    settleOnEstopCmdEqAct;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

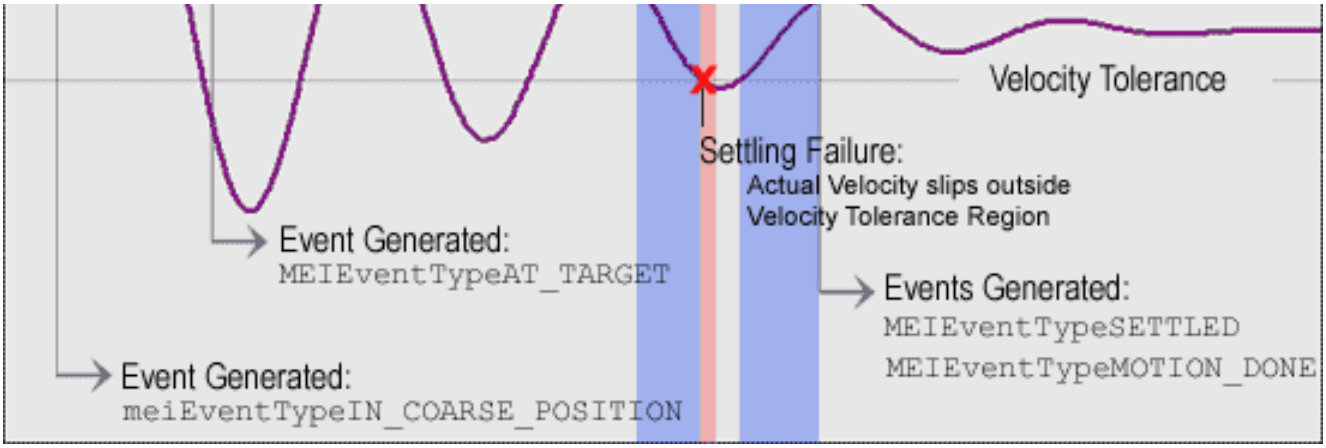
Description

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. **NOTE**: Always enter a Tolerance Velocity value that is a multiple of the controller sample rate. The controller will then receive velocity in counts/ controller sample. 1 Counts/second = (1 counts/second) * (1/sample rate(Hz)) = (1/sample rate) counts/controller sample

	NOTE: Value is truncated to the next smallest integer. **Example**: With a sample rate of 2000Hz, • a Tolerance Velocity value of 500 counts/second = 0.25 = 0 count/ controller sample (after truncation). • a Tolerance Velocity value of 2000 counts/second = 1 count/ controller sample.
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: • The settling criteria are satisfied AND. • The stop duration for the axis' Motion Supervisor has elapsed. • This state can be read with mpiAxisStatus(MPIAxis axis, MPIStatus *status, void *external). The value to look for is (MPIState) status.state. If settleOnStop = FALSE, 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: • The settling criteria are satisfied AND. • The Estop duration for the axis' Motion Supervisor has elapsed. • This state can be read with mpiAxisStatus(MPIAxis axis, MPIStatus *status, void *external). The value to look for is (MPIState) status.state. If settleOnEStop = FALSE, 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 Related Events: How Motion Related Events are Generated

Configuration of IN_POSITION and Done Events after STOP or E_STOP Events

MPIAxisMaster

Definition

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

Description

MPIAxisMaster 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
MPIAxisMasterTypeNONE	Not used	Not used

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

Definition

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

Change History: Modified in the 03.04.00. Modified in the 03.03.00.

Description

MPIAxisMasterType specifies the type of master position source used with cam motions. See also MPIAxisMaster.

Fields	Number	Address
MPIAxisMasterTypeNONE	Not used	Not used
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

See Also

MPIAxisMaster

MPIAxisMessage

Definition

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

Description

MPIAxisMessage 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(...), mpiMotionAxisListGet(...), 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

MEIPreFilter

Definition

```
typedef struct MEIPreFilter {
    long                axisNumber;
    MEIPreFilterForm    form;
    long                length;
    long                coeff[MEIPreFilterCoeffsMAX];
} MEIPreFilter;
```

Change History: Added in the 03.04.00.

Description

PreFilters are used to filter motion trajectories. The command positions generated by the controller firmware during a move are passed through the filter before being used as set points by the control algorithm. PreFilters are useful for removing unwanted frequencies from the motion profile or for smoothing out motion generated by joysticks or other manual input devices.

Two forms of PreFilters are supported: BOXCAR and SHAPING. The BOXCAR filter is a simple averager where the output of the filter is the of average a number of previous command positions. The number of points is determined by the length parameter. For BOXCAR filters the coeff[] array is ignored.

The SHAPING PreFilter passes the trajectory through a special filter type patented by **Convolve, Inc.** ® (www.convolve.com). This filter can greatly enhance the performance of mechanical systems with resonances or flexible hardware. The length and coefficients of the SHAPING filter are generated by Convolve® for the specific system using the filter. See the Convolve website for information about the advantages of Input Shaping®.

axisNumber	The number of the axis to apply the filter.
form	The type of filter (NONE, BOXCAR, or SHAPING).
length	The filter length (number of stages).
coeff	Used only for SHAPING filters. The coefficients are generated by Convolve®, Inc. software (see www.convolve.com).

See Also

MEIPreFilterForm | MEIControlConfig

MEIPreFilterForm

Definition

```
typedef enum {
   MEIPreFilterFormNONE,
   MEIPreFilterFormBOXCAR,
   MEIPreFilterFormSHAPING,

} MEIPreFilterForm;
```

Change History: Added in the 03.04.00.

Description

MEIPreFilterForm specifies the filter types for filtering the command position profile produced by the controller.

MEIPreFilterFormNONE	The PreFilter is disabled and can be used by another axis.
MEIPreFilterFormBOXCAR	The axis' command positions are averaged using a BOXCAR averager. The length of the averager (number of samples to average) can be specified.
MEIPreFilterFormSHAPING	The axis' command positions are filtered using a special resonance elimination filter patented by Convolve®, Inc. See www.convolve.com for more information about SHAPING filters.

See Also

MEIPreFilter

MEIPreFilterCoeffsMAX

Definition

```
#define  MEIPreFilterCoeffsMAX      (MEIXmpMAX_PreCoeffs)
```

Change History: Added in the 03.04.00.

Description

MEIPreFilterCoeffsMAX defines the maximum number of coefficients for a SHAPING filter. (See MEIPreFilter description.)

See Also

MEIPreFilterCountMAX

MEIPreFilterCountMAX

Definition

```
#define  MEIPreFilterCountMAX      (MEIXmpMAX_PreFilters)
```

Change History: Added in the 03.04.00.

Description

MEIPreFilterCountMAX defines the maximum number of axes that can be filtered (with either BOXCAR or SHAPING filters).

See Also

MEIPreFilterCoeffsMAX

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. meiCanFlashConfigGet and meiCanFlashConfigSet 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 | CAN Hardware on the XMP | CAN Analog Values

| Error Messages |

Methods

Create, Delete, Validate Methods

meiCan**Create**	Create Can object
meiCan**Delete**	Delete Can object
meiCan**Validate**	Validate Can object

Configuration and Information Methods

meiCan**ConfigGet**	Get Can's configuration
meiCan**ConfigSet**	Set Can's configuration
meiCan**FlashConfigGet**	Get Can's flash configuration
meiCan**FlashConfigSet**	Set Can's flash configuration
meiCan**Status**	Get status of the CAN controller.
meiCan**Version**	Returns the version information about a controller's CAN system.
meiCan**Command**	Get Can's flash configuration
meiCan**NodeConfigGet**	Return a copy of the current configuration

meiCan**NodeConfigSet**	Update the current configuration that the specified CAN node is using.
meiCan**NodeFlashConfigGet**	Get the flash configuration of the Can node
meiCan**NodeFlashConfigSet**	Set the flash configuration of the Can node
meiCan**NodeStatus**	Get the instantaneous state of the local CAN interface.
meiCan**NodeInfo**	Return the node information after the XMP finishes scanning the network.

I/O Methods

- meiCan**NodeAnalogIn**
- meiCan**NodeAnalogOutGet**
- meiCan**NodeAnalogOutSet**
- meiCan**NodeDigitalIn**
- meiCan**NodeDigitalOutGet**
- meiCan**NodeDigitalOutSet**

Event Methods

meiCan**EventNotifyGet**	Get event mask of events for which host notification has been requested
meiCan**EventNotifySet**	Set event mask of events for which host notification will be requested

Firmware Methods

meiCan**FirmwareDownload**	Downloads firmware to the Can controller
meiCan**FirmwareErase**	Erases firmware on the Can controller
meiCan**FirmwareUpload**	Uploads firmware from the Can controller

Memory Methods

meiCan**Memory**	Get address to Can's memory
meiCan**MemoryGet**	Copy data from Can memory to application memory
meiCan**MemorySet**	Copy data from application memory to Recorder memory

Action Methods

- meiCan**Init**

Relational Methods

- meiCan**Control**
- meiCan**Number**

Data Types

- MEICan**BitRate**
- MEICan**BusState**

MEICan**Callback**
MEICan**Command**
MEICan**CommandType**
MEICan**Config**
MEICan**HealthType**
MEICan**Message**
MEICan**NodeConfig**
MEICan**NodeInfo**
MEICan**NodeInfoProductCode**
MEICan**NodeInfoVendor**
MEICan**NodeStatus**
MEICan**NodeType**
MEICan**NMTState**
MEICan**Status**
MEICan**TransmissionType**
MEICan**Version**

Constants

MEICan**NetworkMAX**

meiCanCreate

Declaration

```
MEICan meiCanCreate(MPIControl  control,
                    long        number);
```

Required Header: stdmei.h
Change History: Modified in the 03.02.00

Description

meiCanCreate 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.
number	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.

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

Sample Code

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

```
MPIControl ControlHandle;
 MEICan CANHandle;
 long Result;

 /* Create, validate and initalise a handle to the controller. */
 ControlHandle = mpiControlCreate( MPIControlTypeDEFAULT, 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 );
```

See Also

mpiCanDelete | mpiCanValidate

meiCanDelete

Declaration

```
long meiCanDelete(MEICan  can);
```

Required Header: stdmei.h

Description

meiCanDelete deletes the specified CAN object.

can	handle to the CAN object to delete.

Return Values	
MPIMessageOK	

Sample Code

```
See meiCanCreate for an example of how to use meiCanDelete.
```

See Also

meiCanCreate | meiCanValidate

meiCanValidate

Declaration

```
long meiCanValidate(MEICan   can);
```

Required Header: stdmei.h

Description

meiCanValidate validates the specified CAN handle.

can	handle to the CAN object

Return Values	
MPIMessageOK	
MPIMessageUNSUPPORTED	

Sample Code

```
See meiCanCreate for an example of how to use meiCanValidate.
```

See Also

meiCanNodeInfo | meiCanNodeStatus

meiCanConfigGet

Declaration

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

Required Header: stdmei.h

Description

meiCanConfigGet 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	

See Also

meiCanConfigSet

meiCanConfigSet

Declaration

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

Required Header: stdmei.h

Description

meiCanConfigSet 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	

See Also

meiCanConfigGet

meiCanFlashConfigGet

Declaration

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

Required Header: stdmei.h

Description

meiCanFlashConfigGet 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	

See Also

meiCanFlashConfigSet

meiCanFlashConfigSet

Declaration

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

Required Header: stdmei.h

Description

meiCanFlashConfigSet updates 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	

See Also

meiCanFlashConfigGet

meiCanStatus

Declaration

```
long meiCanStatus(MEICan          can,
                  MEICanStatus*   status);
```

Required Header: stdmei.h

Description

meiCanStatus gets the instantaneous state of the local CAN interface to the CAN network.

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.

Return Values	
MPIMessageOK	

See Also

meiCanNodeInfo | meiCanNodeStatus

meiCanVersion

Declaration

```
long meiCanVersion(MEICan           can,
                   MEICanVersion*   version);
```

Required Header: stdmei.h

Description

meiCanVersion returns the version of the firmware being used by the CAN controller.

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

Return Values	
MPIMessageOK	

See Also

meiCanCommand

Declaration

```
long meiCanCommand(MEICan            can,
                   MEICanCommand*    command);
```

Required Header: stdmei.h

Description

meiCanCommand allows a set of basic commands to be performed. The ***type*** field of the MEICanCommand structure specifies the type of command to perform.

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.

Return Values	
MPIMessageOK	

See Also

MEICanCommand

meiCanNodeConfigGet

Declaration

```
long meiCanNodeConfigGet(MEICan              can,
                         long                node,
                         MEICanNodeConfig*   nodeConfig);
```

Required Header: stdmei.h

Description

meiCanNodeConfigGet returns a copy of 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 that will be filled in by this function.

Return Values	
MPIMessageOK	

See Also

meiCanNodeConfigSet | meiCanConfigGet | meiCanConfigSet

meiCanNodeConfigSet

Declaration

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

Required Header: stdmei.h

Description

meiCanNodeConfigSet 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	

See Also

meiCanNodeConfigGet | meiCanConfigGet | meiCanConfigSet

meiCanNodeFlashConfigGet

Declaration

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

Required Header: stdmei.h

Description

meiCanNodeFlashConfigGet 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

Return Values	
MPIMessageOK	

See Also

meiCanNodeFlashConfigSet

meiCanNodeFlashConfigSet

Declaration

```
long meiCanNodeFlashConfigSet(MEICan            can,
                              void*             flash,
                              long              node,
                              MEICanNodeConfig* nodeConfig);
```

Required Header: stdmei.h

Description

meiCanNodeFlashConfigSet 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	

See Also

meiCanNodeFlashConfigGet

meiCanNodeStatus

Declaration

```
long meiCanNodeStatus(MEICan              can,
                      long                node,
                      MEICanNodeStatus*   nodeStatus);
```

Required Header: stdmei.h

Description

meiCanNodeStatus gets the instantaneous state of the specified node on the CAN network.

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.

Return Values	
MPIMessageOK	

See Also

meiCanNodeInfo | meiCanStatus

meiCanNodeInfo

Declaration

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

Required Header: stdmei.h

Description

meiCanNodeInfo returns the node information for the specified node on the CAN network that was generated when the XMP/ZMP 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	

See Also

meiCanNodeStatus | meiCanStatus

meiCanNodeAnalogIn

Declaration

```
long meiCanNodeAnalogIn(MEICan        can,
                        long          node,
                        long          channel,
                        long          *state);
```

Required Header: stdmei.h
Change History: Added in the 03.03.00

Description

meiCanNodeAnalogIn gets the current state of an analog input on the specified CAN node.

can	handle to the CAN object
node	the node number of the CANOpen node.
channel	the index of the analog input.
***state**	a pointer to where the current state of the input is written to by this function. See CAN Analog Values.

Return Values	
MPIMessageOK	

Sample Code

The following code shows how to get the state of analog input 3 on node 5.

```
long analog3;
meiCanNodeAnalogIn( can, 5, 3, &analog3 );
```

See Also

meiCanNodeAnalogOutSet | meiCanNodeAnalogOutGet | CAN Analog Values

meiCanNodeAnalogOutGet

Declaration

```
long meiCanNodeAnalogOutGet(MEICan        can,
                            long          node,
                            long          channel,
                            long          *state);
```

Required Header: stdmei.h
Change History: Added in the 03.03.00

Description

meiCanNodeAnalogOutGet gets the current state of an analog output on the specified CAN node.

can	handle to the CAN object
node	the node number of the CANOpen node.
channel	the index of the analog output.
***state**	a pointer to where the current state of the output is written to by this function. See CAN Analog Values.

Return Values	
MPIMessageOK	

Sample Code

The following code shows how to get the state of analog output 3 on node 5.

```
long analog3;
meiCanNodeAnalogOutGet( can, 5, 3, &analog3 );
```

See Also

meiCanNodeAnalogIn | meiCanNodeAnalogOutSet | CAN Analog Values

meiCanNodeAnalogOutSet

Declaration

```
long meiCanNodeAnalogOutSet(MEICan        can,
                            long          node,
                            long          channel,
                            long          state,
                            long          wait);
```

Required Header: stdmei.h
Change History: Added in the 03.03.00

Description

meiCanNodeAnalogOutSet changes the state of an analog output on the specified CAN node.

can	handle to the CAN object
node	the node number of the CANOpen node.
channel	the index of the analog output.
state	the new state of the analog output. See CAN Analog Values.
wait	a Boolean flag that indicates if the new output state is immediately applied or a *wait* is inserted so that any previously set output is transmitted over CAN first before applying the new output state.

Return Values	
MPIMessageOK	

Sample Code

The following code shows how to change the state of analog output 3 on node 5 to the maximum value 7FFFh.

```
meiCanNodeAnalogOutGet( can, 5, 3, 0x7FFF, 1 );
```

See Also

meiCanNodeAnalogIn | meiCanNodeAnalogOutGet | CAN Analog Values

meiCanNodeDigitalIn

Declaration

```
long meiCanNodeDigitalIn(MEICan         can,
                         long           node,
                         long           bitStart,
                         long           bitCount,
                         unsigned long *state);
```

Required Header: stdmei.h
Change History: Added in the 03.03.00

Description

meiCanNodeDigitalIn gets the current state of one or multiple digital inputs on the specified CAN node.

can	handle to the CAN object
node	the node number of the CANOpen node.
bitSmart	the first input bit on the CAN node that will be returned by this function.
bitCount	the number of bits that will be returned by the function.
***state**	the address of the current state of the input(s) that is returned.

Return Values	
MPIMessageOK	

Sample Code

The following code shows how to get the state of controller input 1.

```
unsigned long input3;
meiCanDigitalIn( can, 5, 3, 1, &input3 );
```

See Also

meiCanNodeDigitalOutSet | meiCanNodeDigitalOutGet

meiCanNodeDigitalOutGet

Declaration

```
long meiCanNodeDigitalOutGet(MEICan        can,
                             long          node,
                             long          bitStart,
                             long          bitCount,
                             unsigned long *state);
```

Required Header: stdmei.h
Change History: Added in the 03.03.00

Description

meiCanNodeDigitalOutGet gets the current state of one or multiple digital outputs on the specified CAN node.

can	handle to the CAN object
node	the node number of the CANOpen node.
bitSmart	the first output bit on the CAN node that will be returned by this function.
bitCount	the number of bits that will be returned by the function.
***state**	the address of the current state of the output(s) that is returned.

Return Values	
MPIMessageOK	

Sample Code

The following code shows how to get the state of digital output 3 on node 5.

```
unsigned long output3;
meiCanDigitalOutGet( can, 5, 3, 1, &output3 );
```

See Also

meiCanNodeDigitalOutSet | meiCanNodeDigitalIn

meiCanNodeDigitalOutSet

Declaration

```
long meiCanNodeDigitalOutSet(MEICan        can,
                             long          node,
                             long          bitStart,
                             long          bitCount,
                             unsigned long state,
                             MPI_BOOL      wait);
```

Required Header: stdmei.h
Change History: Added in the 03.03.00

Description

meiCanNodeDigitalOutSet changes the state of one or multiple digital outputs on the specified CAN node.

can	handle to the CAN object
node	the node number of the CANOpen node.
bitSmart	the first output bit on the CAN node that will be returned by this function.
bitCount	the number of bits that will be set by the function.
state	the new state of the outputs on the CANOpen node.
wait	a Boolean flag that indicates if the new output state is immediately applied or a wait is inserted so that any previously set output is transmitted over CAN first before applying the new output state.

Return Values	
MPIMessageOK	

Sample Code

The following code shows how to set the state of digital output 3 on node 5.

```
meiCanDigitalOutSet( can, 5, 3, 1, 1, 1);
```

See Also

meiCanNodeDigitalOutGet | meiCanNodeDigitalIn

meiCanEventNotifyGet

Declaration

```
long meiCanEventNotifyGet(MEICan          can,
                          MPIEventMask    *eventMask,
                          void            *external);
```

Required Header: stdmei.h

Description

meiCanEventNotifyGet 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	

See Also

meiCanNotifySet

meiCanEventNotifySet

Declaration

```
long meiCanEventNotiySet(MEICan          can,
                         MPIEventMask    eventMask,
                         void            *external);
```

Required Header: stdmei.h

Description

meiCanEventNotifySet 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	

See Also

meiCanEventNotifyGet

meiCanFirmwareDownload

Declaration

```
long meiCanFirmwareDownload(MEICan            can,
                            const  char*      filename,
                            MEICanCallback    callback);
```

Required Header: stdmei.h
Change History: Modified in the 03.03.00

Description

meiCanFirmwareDownload 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	

See Also

meiCanFirmwareErase | meiCanFirmwareUpload

meiCanFirmwareErase

Declaration

```
long meiCanFirmwareErase(MEICan   can);
```

Required Header: stdmei.h

Description

meiCanFirmwareErase allows the user to erase the CAN controllers firmware.

can	handle to the CAN object

Return Values	
MPIMessageOK	

See Also

meiCanFirmwareDownload | meiCanFirmwareUpload

meiCanFirmwareUpload

Declaration

```
long meiCanFirmwareUpload(MEICan            can,
                          const  char*      filename,
                          MEICanCallback    callback);
```

Required Header: stdmei.h
Change History: Modified in the 03.03.00

Description

meiCanFirmwareUpload 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	

See Also

meiCanFirmwareErase | meiCanFirmwareDownload

meiCanMemory

Declaration

```
long meiCanMemory(MEICan   can,
                  void**   memory);
```

Required Header: stdmei.h

Description

meiCanMemory 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	

See Also

meiCanMemoryGet | meiCanMemorySet

meiCanMemoryGet

Declaration

```
long meiCanMemoryGet(MEICan        can,
                     void*         dst,
                     const void*   src,
                     long          count);
```

Required Header: stdmei.h
Change History: Modified in the 03.03.00

Description

meiCanMemoryGet 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	

See Also

meiCanMemory | meiCanMemorySet

meiCanMemorySet

Declaration

```
long meiCanMemorySet(MEICan        can,
                     void*         dst,
                     const void*   src,
                     long          count);
```

Required Header: stdmei.h
Change History: Modified in the 03.03.00

Description

meiCanMemorySet 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	

See Also

meiCanMemory | meiCanMemoryGet

meiCanInit

Declaration

```
long meiCanInit(MEICan    can);
```

Required Header: stdmei.h
Change History: Added in the 03.03.00

Description

meiCanInit will reset the CAN network and will not affect the rest of the controller or SynqNet.

can	handle to the CAN object

Return Values	
MPIMessageOK	

See Also

meiCanControl

Declaration

```
MPIControl meiCanControl(MEICan   can);
```

Required Header: stdmei.h
Change History: Added in the 03.02.00

Description

meiCanControl returns a handle to the control object associated with the Can object.

can	a handle to a Can object.

Return Values	
MPIControl	a handle to a control object.
MPIHandleVOID	if the object could not be created

See Also

meiCanCreate | mpiControlCreate

meiCanNumber

Declaration

```
long  meiCanNumber(MEICan   can,
                   long     *number);
```

Required Header: stdmei.h
Change History: Added in the 03.02.00

Description

meiCanNumber reads the index of a Can object and writes it into the contents of a long pointed to by ***number***. Each Can node associated with a controller is indexed by a number (0, 1, 2, etc.).

can	a handle to a Can object.
***number**	a pointer to the index of a Can node.

Return Values	
MPIMessageOK	
MPIMessageARG_INVALID	
MPIMessageHANDLE_INVALID	

See Also

meiCanNodeInfo

MEICanBitRate

Definition

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

Description

MEICanBitRate 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

Definition

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

Description

MEICanBusState 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

CAN Bus State

MEICanCallback

Definition

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

Description

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

See Also

MEICanCommand

Definition

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

Description

MEICanCommand 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

Definition

```
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

MEICanCommandType 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 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

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

MEICanConfig

Definition

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

Description

MEICanConfig 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 minimum time (in milliseconds) that a node on the network will remain silent before transmitting their next event message. See also CAN Transmission Types.

See Also

meiCanConfigGet | meiCanConfigSet | meiCanNodeConfigGet | meiCanNodeConfigSet

MEICanHealthType

Definition

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

Description

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

See Also

MEICanMessage

Definition

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

Change History: Modified in the 03.02.00

Description

MEICanMessage is an enumeration of Can error messages that can be returned by the MPI library.

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.

MEICanMessageCAN_INVALID

The can network number is out of range. This message code is returned by meiCanCreate(...) if the network number is less than zero or greater than or equal to MEICanNetworkMAX.

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

Definition

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

Description

MEICanNodeConfig 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

MEICanTransmissionType | meiCanNodeConfigGet | meiCanNodeConfigSet

MEICanNodeInfo

Definition

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

Change History: Modified in the 03.03.00

Description

MEICanNodeInfo 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. See also 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 0x014F. See also MEICanNodeInfoVendor.
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 0x0204. See also MEICanNodeInfoProductCode.
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

MEICanNodeInfoProductCode

Definition

```
typedef enum {
    MEICanNodeInfoProductCodeUNKNOWN = 0,
    MEICanNodeInfoProductCodeMEI_SLICE_IO = 0x0204
} MEICanNodeInfoProductCode;
```

Change History: Modified in the 03.03.00

Description

MEICanNodeInfoProductCode defines the product codes for the MEI manufactured CANOpen nodes. If the node is not manufactured by MEI then the product code may be any non-zero number. A zero product code (UNKNOWN) indicates that the manufacturer does not support the CANOpen method to read this from the node.

See Also

MEICanNodeInfo | Slice-I/O Hardware

MEICanNodeInfoVendor

Definition

```
typedef enum {
   MEICanNodeInfoVendorUNKNOWN = 0,
   MEICanNodeInfoVendorMEI = 0x014F
} MEICanNodeInfoVendor;
```

Change History: Added in the 03.03.00

Description

MEICanNodeInfoVendor defines some vendor Ids for CANOpen nodes. A zero vendor ID (UNKNOWN) indicates that the manufacturer does not support the CANOpen method to read this from the node.

See Also

MEICanNodeInfo

MEICanNodeStatus

Definition

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

Description

MEICanNodeStatus 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

CAN Node Health

MEICanNodeType

Definition

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

Description

MEICanNodeType 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

CAN Node Health

MEICanNMTState

Definition

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

Description

MEICanNMTState 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

Definition

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

Description

MEICanStatus holds the current status of the XMP's or ZMP's CAN object.

busState	The current bus state of the XMP's or ZMP'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

Definition

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

Description

MEICanTransmissionType enumerates the transmission types a node can use.

For more information, see the CAN Transmission Types section.

See Also

MEICanNodeConfig | meiCanNodeConfigGet | meiCanNodeConfigSet

MEICanVersion

Definition

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

Description

MEICanVersion holds the version information about the XMP's or ZMP'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

MEICanNetworkMAX

Definition

```
#define MEICanNetworkMAX  (1)
```

Change History: Added in the 03.02.00

Description

MEICanNetworkMAX defines the maximum number of Can networks supported by a controller.

See Also

meiCanCreate

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

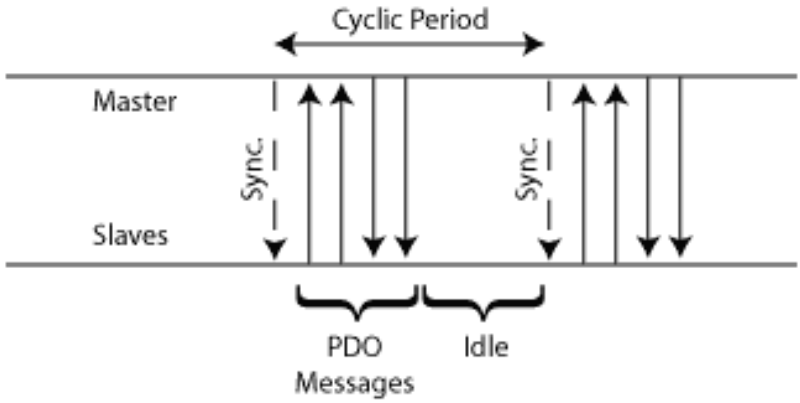
CAN Transmission Types

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

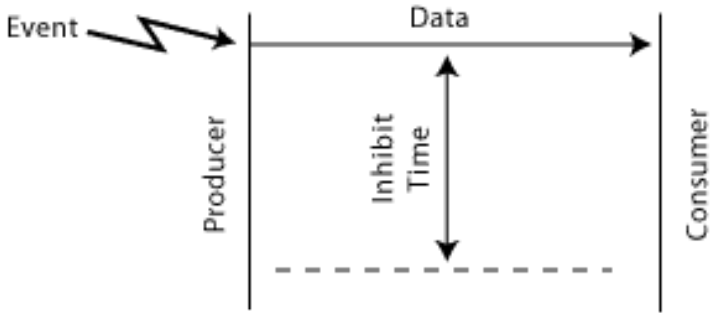
The cyclicPeriod field within the MEICanConfig structure allows the user to specify the period (in milliseconds) that the XMP will use between the successive transmission of synchronization messages. The minimum cyclic period that can be used is dependent upon the chosen bit rate and the number of nodes.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

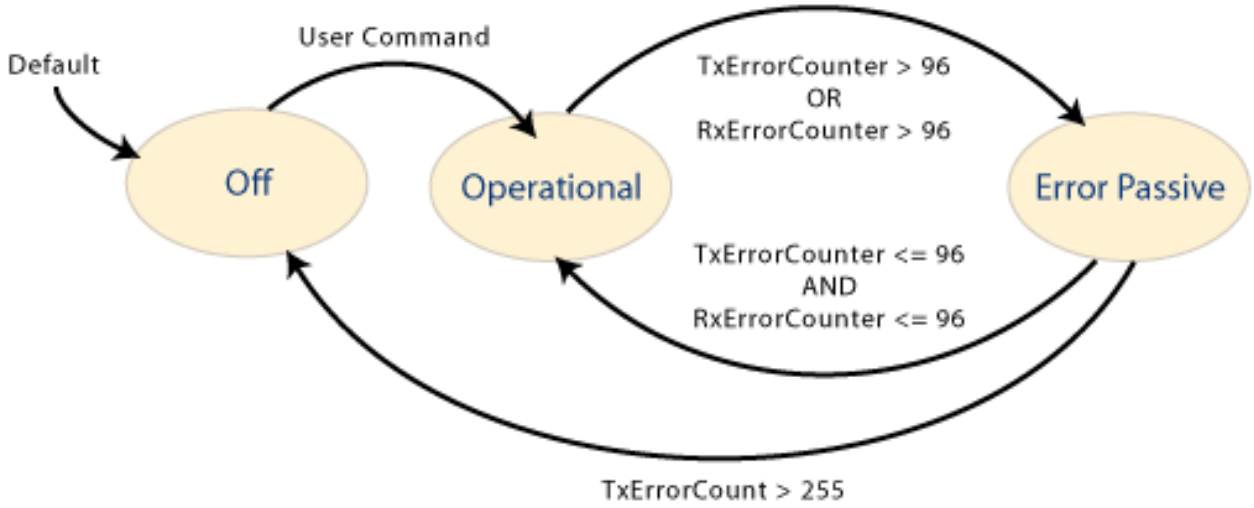
Inhibit Time

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.

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:

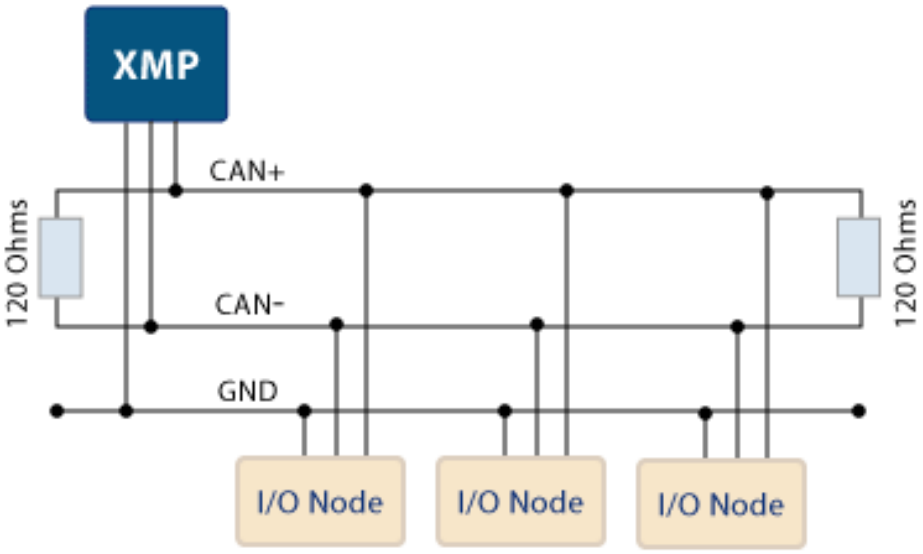


When a node is in the **Operational** state it will participate fully with all communications over the network, as the errors increase the CAN hardware will become **Passive** (detecting errors but not generating error messages), before turning **Off** and isolating the node from the network once the 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 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 that 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

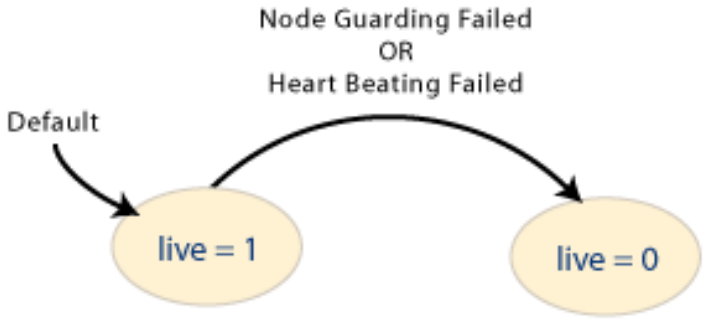
function.

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.

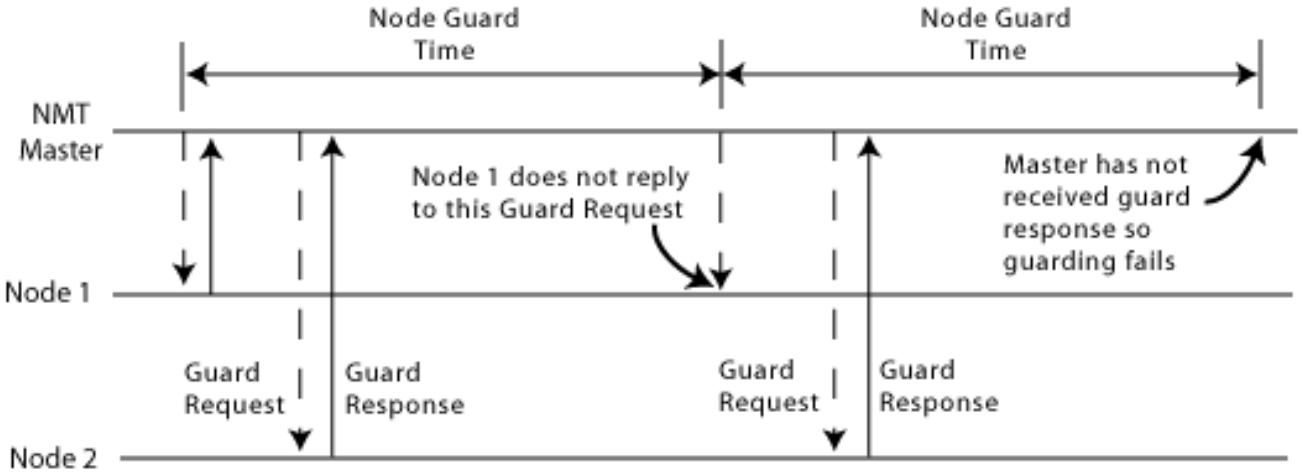
Using these services the XMP can monitor the health of the communications to each of the nodes. The current health of each node is reported in the live field of the MEICANNodeStatus structure.



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

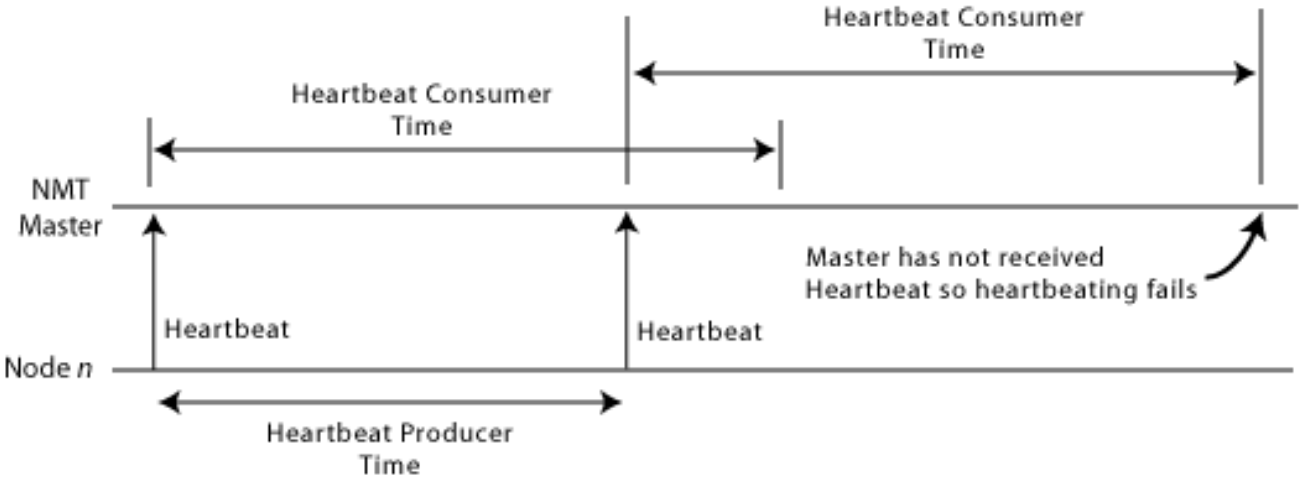
The Node Guarding protocol has the master sending an RTR message to all nodes on the network and checks to see whether a response is received from each of the nodes.



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
Heatbeat Consumer Time	healthPeriod

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

CAN Emergency Messages

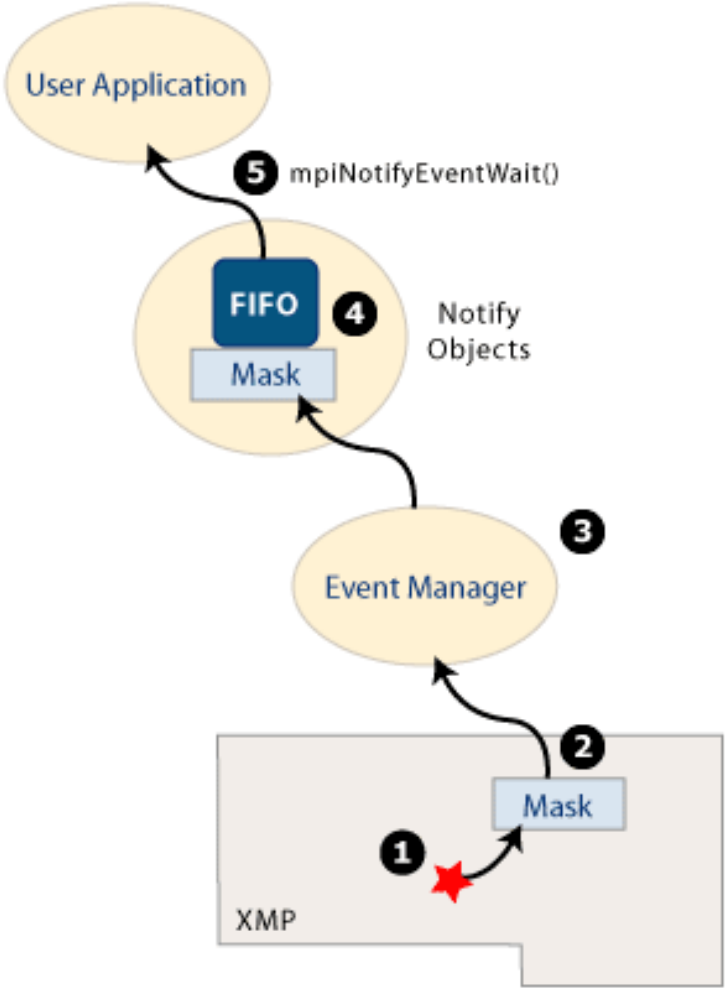
Every type of CANOpen node can transmit an emergency message. These messages are designed to report errors and warnings, as well as fatal problems on a node. The contents of these emergency messages are very dependent upon the node manufacturer and node type. To interpret this data, you will need to 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 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

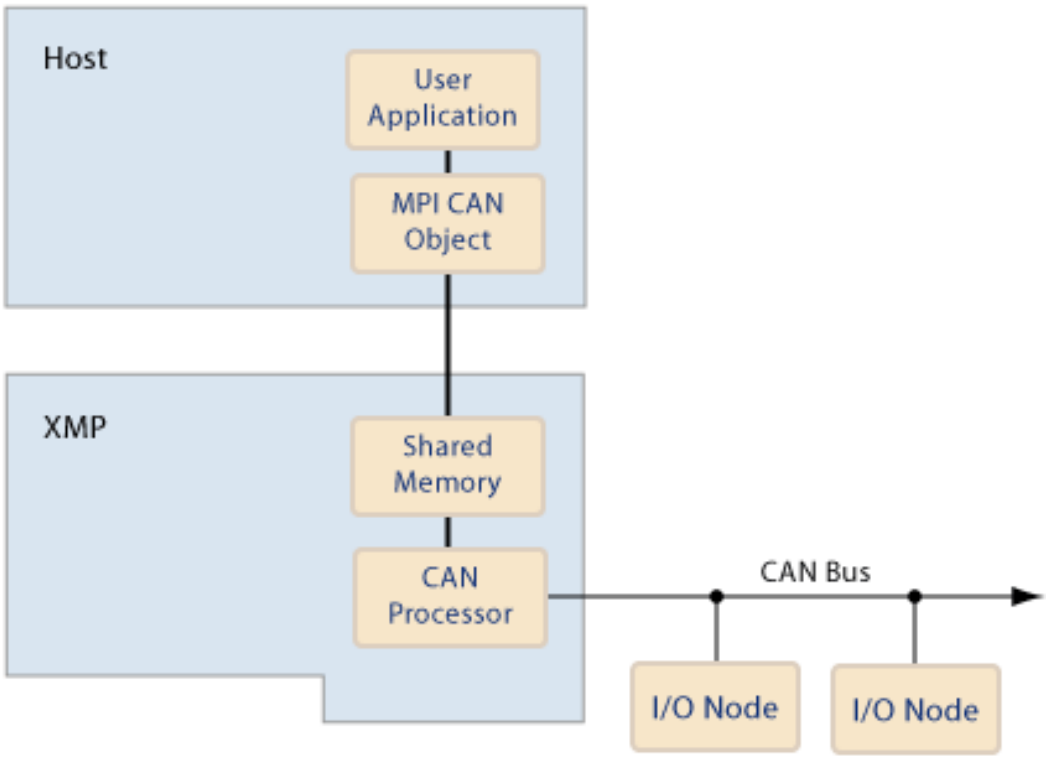
The events above have been appended to the standard MPI event handling scheme in order to provide the user the ability to respond to these events. The diagram below shows an overview of how events are relayed to the user's application.



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).

CAN Hardware on the XMP

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.

Capture Objects

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.

NOTE: Time-based capture will only work correctly if the speed of an axis is less than 344 million counts per second.

For an overview of the Capture Engine, see diagram below.

See Also: Overview of Capture

| Error Messages |

Methods

Create, Delete, Validate Methods

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

Configuration and Information Methods

mpiCapture**ConfigGet**	Get Capture configuration
mpiCapture**ConfigSet**	Set Capture configuration
mpiCapture**Status**	Get status of Capture
mpiCapture**ConfigReset**	

Action Methods

mpiCapture**Arm**	Arm capture object

Memory Methods

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

Relational Methods

mpiCapture**Control**	
mpiCapture**Number**	Get index of Capture (for Control list)

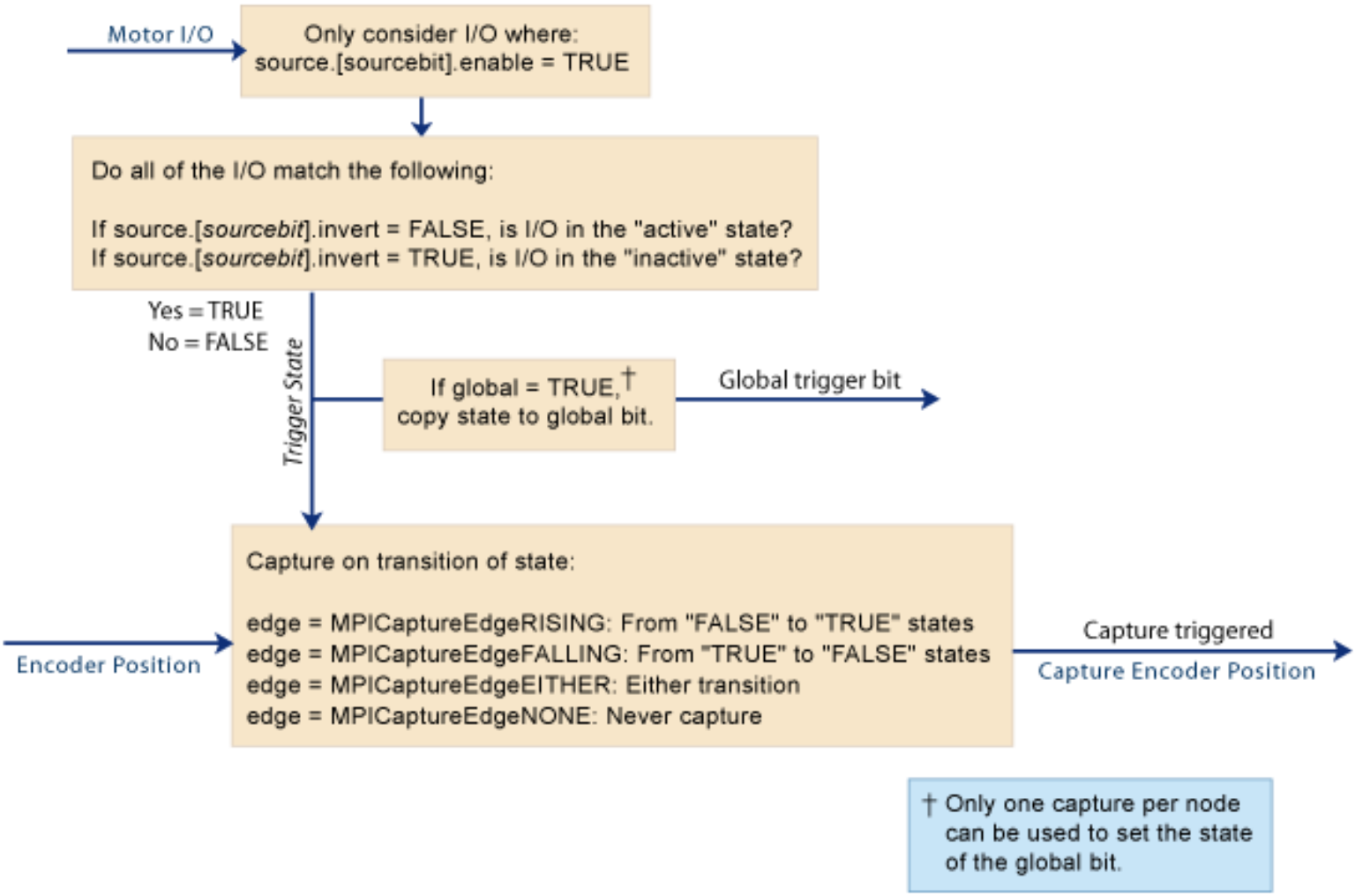
Data Types

MPICapture**Config**
MPICapture**Edge**
MPICapture**Message** / MEICapture**Message**
MPICapture**Source**
MPICapture**State**
MPICapture**Status**
MPICapture**Trigger**
MPICapture**TriggerGlobal**
MPICapture**Type**

Constants

MPICapture**NOT_MAPPED**

Capture Engine Diagram



mpiCaptureCreate

Declaration

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

Required Header: stdmpi.h

Description

mpiCaptureCreate 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 32.

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

mpiCaptureDelete deletes a Capture object and invalidates its handle (***capture***).

CaptureDelete is the equivalent of a C++ destructor.

Return Values	
MPIMessageOK	

See Also

mpiCaptureCreate | mpiCaptureValidate

mpiCaptureValidate

Declaration

```
long mpiCaptureValidate(MPICapture  capture)
```

Required Header: stdmpi.h

Description

mpiCaptureValidate 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	

See Also

mpiCaptureCreate | mpiCaptureDelete

mpiCaptureConfigGet

Declaration

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

Required Header: stdmpi.h

Description

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

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

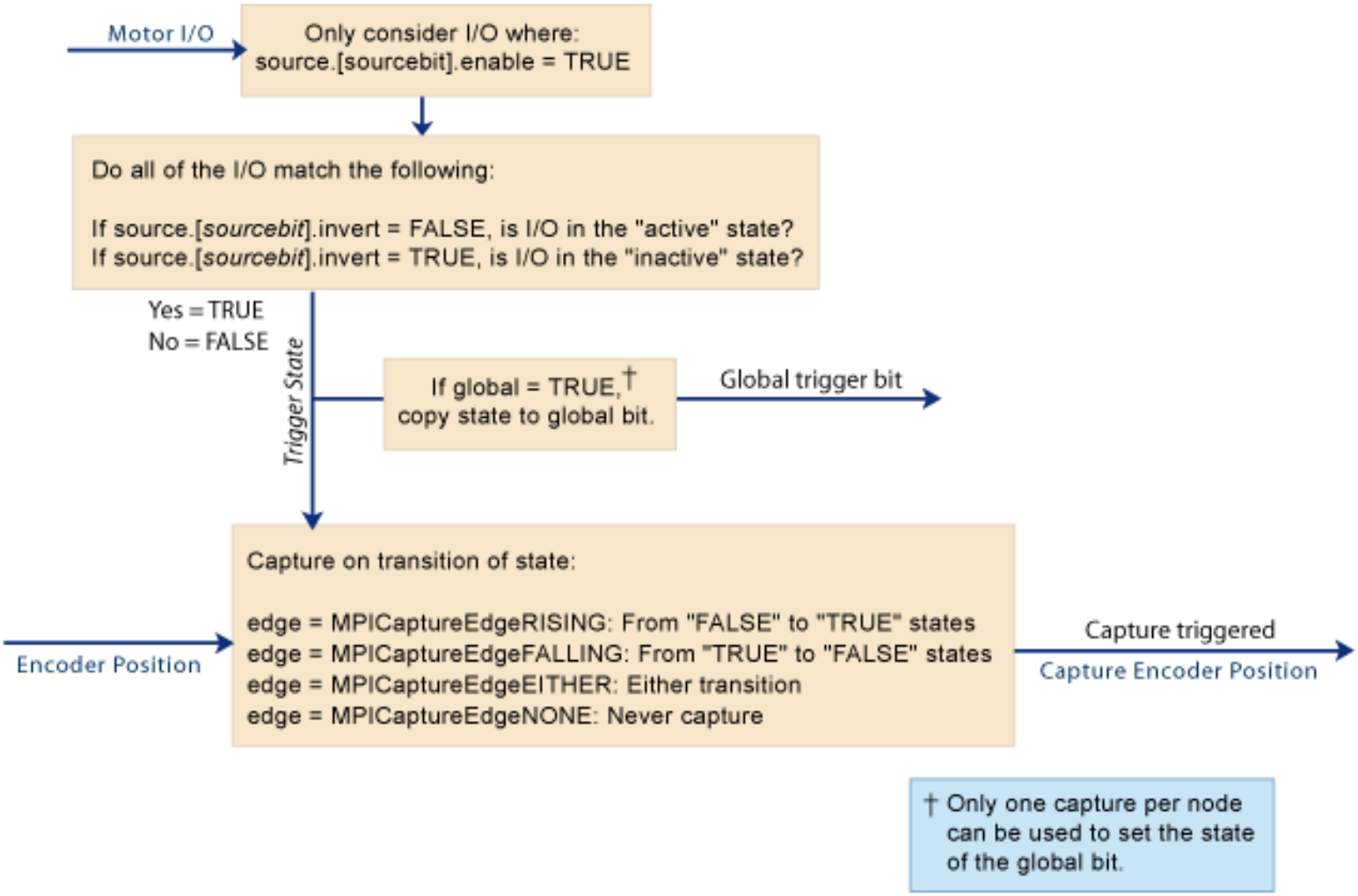
If a capture has been previously configured (non-default), use mpiCaptureConfigReset(...) to return the capture to the default configuration before calling mpiCaptureConfigGet(...) and mpiCaptureConfigSet (...). Or if you do not call mpiCaptureConfigReset(...), make sure that all members of the MPICaptureConfig{...}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

Remarks

***external** should be NULL.

Return Values	
MPIMessageOK	

Capture Engine Diagram



See Also

mpiCaptureConfigSet | mpiCaptureConfigReset

mpiCaptureConfigSet

Declaration

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

Required Header: stdmpi.h

Description

mpiCaptureConfigSet 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 has been previously configured (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(...).

NOTE:
Don't reconfigure the source (trigger) capture resource with different settings via different MPICapture objects. Time-based capture allows you to capture multiple encoder positions using the same trigger. Currently, each motor only has one trigger resource— i.e. one trigger whose trigger state may be configured. If MPICapture object 0 is configured to trigger off of motor 0's index line, and then MPICapture object 1 is configured to trigger off of motor 0's home input, then only the capture trigger for motor 0 will have been reconfigured. Both
MPICapture object 0 and MPICapture object 1 will now trigger off of motor 0's home input.

Remarks

***external** should be NULL.

Return Values	
MPIMessageOK	

See Also

mpiCaptureConfigGet | mpiCaptureConfigReset

mpiCaptureStatus

Declaration

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

Required Header: stdmpi.h

Description

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

Remarks

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

capture	a handle to a Capture object
***status**	a pointer to MPIStatus structure
***external**	a pointer to an implementation-specific structure

Return Values	
MPIMessageOK	
MPIMessageARG_INVALID	

See Also

mpiCaptureConfigReset

Declaration

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

Required Header: stdmpi.h

Description

mpiCaptureConfigGet return the capture object to its unmapped state.

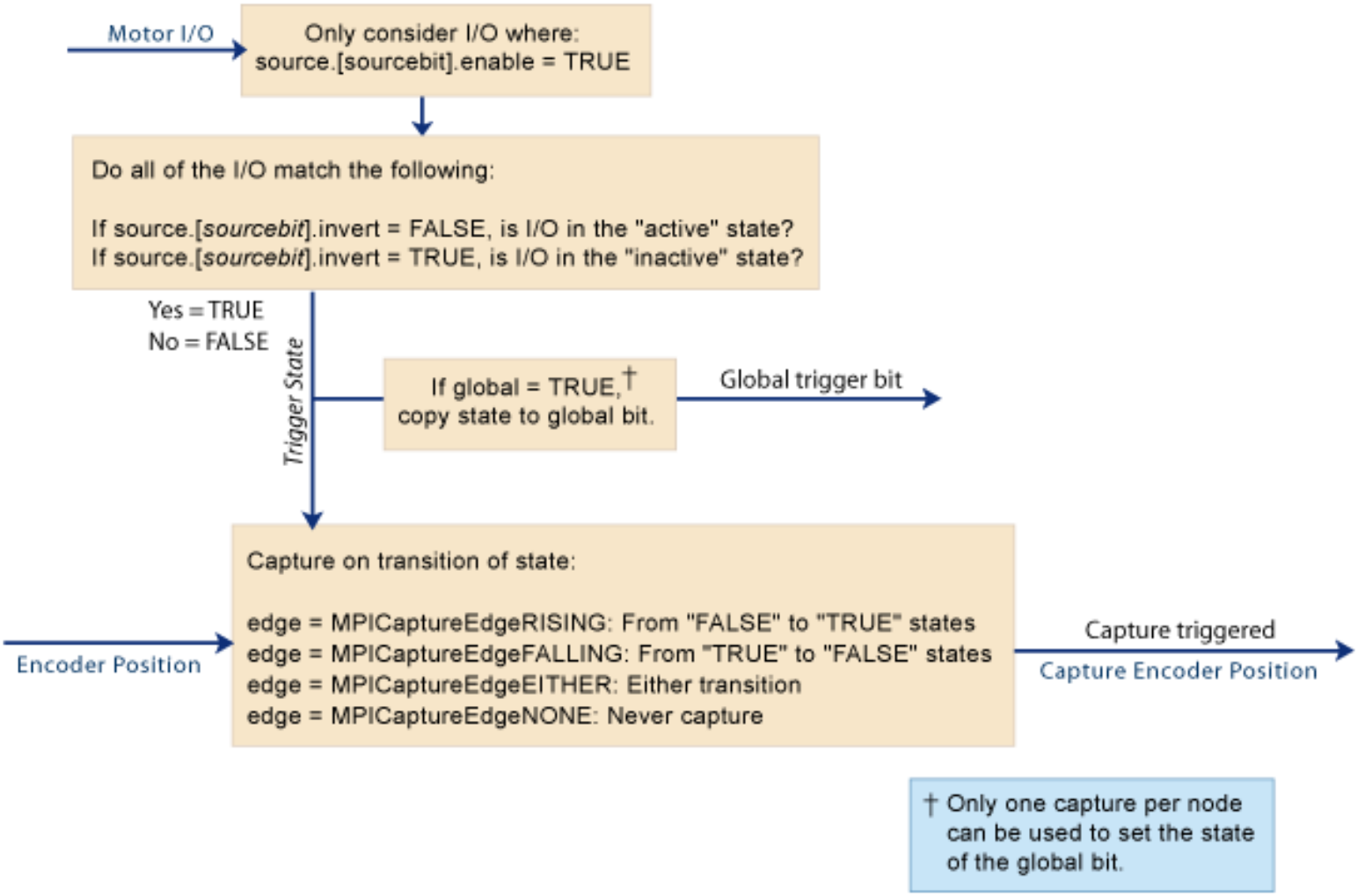
A capture object has no assumed resources, and is unmapped under default conditions. When a capture is first created, its captureMotorNumber and feedbackMotorNumber are unmapped. Once a capture has been configured, the next time that the capture object is created, it will retain the captureMotorNumber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

If a capture has been previously configured (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

Return Values	
MPIMessageOK	

Capture Engine Diagram



See Also

mpiCaptureConfigGet | mpiCaptureConfigSet | MPICaptureConfig

mpiCaptureArm

Declaration

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

Required Header: stdmpi.h
Change History: Modified in the 03.03.00

Description

mpiCaptureArm arms or disarms ***capture***.

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

Return Values	
MPIMessageOK	

See Also

MPICaptureState

mpiCaptureMemory

Declaration

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

Required Header: stdmpi.h

Description

mpiCaptureMemory 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	

See Also

mpiCaptureMemoryGet | mpiCaptureMemorySet

mpiCaptureMemoryGet

Declaration

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

Required Header: stdmpi.h
Change History: Modified in the 03.03.00

Description

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

Return Values	
MPIMessageOK	

See Also

mpiCaptureMemory | mpiCaptureMemorySet

mpiCaptureMemorySet

Declaration

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

Required Header: stdmpi.h
Change History: Modified in the 03.03.00

Description

mpiCaptureMemorySet 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	

See Also

mpiCaptureMemory | mpiCaptureMemoryGet

mpiCaptureControl

Declaration

```
long mpiCaptureControl(MPICapture capture);
```

Required Header: stdmpi.h
Change History: Added in the 03.02.00

Description

mpiCaptureControl returns a handle to the motion controller (Control) with which the Capture is associated.

capture	a handle to a Capture object

Return Values	
MPIControl	Handle to a Control object
MPIHandleVOID	If capture is invalid

See Also

mpiCaptureCreate | mpiControlCreate

mpiCaptureNumber

Declaration

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

Required Header: stdmpi.h

Description

mpiCaptureNumber 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	

See Also

mpiCaptureCreate

MPICaptureConfig

Definition

```
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 postion-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.**

Remarks

Time-based capture will only work correctly if the speed of an axis is less than 344 million counts per second.

See Also

MPICaptureType | mpiCaptureConfigGet | mpiCaptureConfigSet

MPICaptureEdge

Definition

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

Description

MPICaptureEdge 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

Definition: MPICaptureMessage

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

Description

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

MEICaptureMessageMOTOR_INVALID

The capture motor number is not valid. This message code is returned by mpiCaptureConfigSet(...) if the captureMotorNumber does not have node hardware or the value is MPICaptureNOT_MAPPED.

MEICaptureMessageCAPTURE_TYPE_INVALID

The capture type is not valid. This message code is returned by mpiCaptureConfigSet(...) if the type is not one of the values defined by the enum MPICaptureType.

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 MPIMotorEncoder.

See Also

mpiCaptureCreate | mpiControlConfigSet

Definition: 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,
    MEICaptureMessageCAPTURE_ARMED,
} MEICaptureMessage;
```

Change History: Modified in the 03.03.00

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 mpiCaptureCreate(...) 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.

MEICaptureMessageCAPTURE_ARMED

The Capture resource being configured is already armed and cannot be reconfigured until it is disabled or triggered.

See Also

mpiCaptureCreate

MPICaptureSource

Definition

```
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

MPICaptureSource is an enumeration of input trigger sources for a capture.

For dedicated input capture sources, (Home, Index, Limits, etc.) use the enums defined in MPICaptureSource.

For general input capture sources, you will need to look up the node specific input enum that matches the MPICaptureSourceMOTOR_IO values. You can determine the appropriate MPICaptureSourceMOTOR_IO by referencing the appropriate node header file. In the node specific header file (*manufacturer_model.h*), look for the NodeMotorIoConfig (replacing "Node" with the node name). The NodeMotorIoConfig contains the indices for the node specific I/O bits that correspond to the MPICaptureSource enums. Use the MPICaptureSource enum that matches the node specific enum to select the capture trigger source.

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 MPIMotorInfoDedicatedIn.
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.

Example: RMB-10V
Configure MEI RMB-10V capture for trigger on XCVR_C.

From RMBMotorIoConfig in mei_rmb.h:

```
typedef enum {     /* index values for MEIMotorConfigIo[] */
  RMBMotorIoConfigINVALID = -1,

  RMBMotorIoConfigXCVR_A = MEIMotorIoConfigIndex0,
  RMBMotorIoConfigXCVR_B = MEIMotorIoConfigIndex1,
  RMBMotorIoConfigXCVR_C = MEIMotorIoConfigIndex2,
  RMBMotorIoConfigUSER_0_IN = MEIMotorIoConfigIndex6,
  RMBMotorIoConfigUSER_0_OUT = MEIMotorIoConfigIndex7,
  RMBMotorIoConfigUSER_1_IN = MEIMotorIoConfigIndex8,
  RMBMotorIoConfigUSER_1_OUT = MEIMotorIoConfigIndex9,
  RMBMotorIoConfigUSER_2_IN = MEIMotorIoConfigIndex10,
  RMBMotorIoConfigUSER_2_OUT = MEIMotorIoConfigIndex11,

  RMBMotorIoConfigLAST,
  RMBMotorIoConfigFIRST = RMBMotorIoConfigINVALID + 1
} RMBMotorIoConfig;
```

The matching enumeration value for XCVR_C is RMBMotorIoConfigXCVR_C. RMBMotorIoConfigXCVR_C is defined as index 2. Thus, the MPICaptureSource to use is MPICaptureSourceMOTOR_IO_2 (the third MPICaptureSourceMOTOR_IO value).

```
MPICaptureConfig    captureConfig;

/* enable capture source trigger for XCVR_C on mei_rmb */
captureConfig.source[MPICaptureSourceMOTOR_IO_2].enabled = TRUE;
```

If you want to set the capture source for a HOME, simply use the MPICaptureSourceHOME enum.

Example: Trust TA800
To configure the capture source for hall A on a Trust TA800 node, use MPICaptureSourceMOTOR_IO_0 (matches to node specific enum: TA800MotorIoConfigHALL_A). Remember that you will need to look in *trust_ta800.h* (the node module) to find TA800MotorIoConfigHALL_A.

See Also

MPICaptureTrigger

MPICaptureState

Definition

```
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

Definition

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

Description

state	An enumerated value representing the present state of the capture logic
latchedValue	The captured encoder position value. This value is only valid when the state is CAPTURED. The actual position value is the captured encoder position value subtracted by the origin variable. The origin variable can be set/get through mpiAxisOriginSet(...) and mpiAxisOriginGet(...). Recall that encoder positions have no origin adjustment whereas actual positions do have origin adjustment. Please refer to Using the Origin Variable for more information.

Sample Code

```
void displayPosition(MPICapture   capture,
                     MPIAxis      axis)
{
   double             origin;
   MPICaptureStatus   captureStatus;

   /* Check captured position */
   mpiCaptureStatus(capture,
                    &captureStatus,
                    NULL);

   /* Getting origin variable and store it to origin */
   mpiAxisOriginGet(axis, &origin);
```

```
   if (captureStatus.state == MPICaptureStateCAPTURED)
   {

        printf("Latched actual position: %.0lf\n",
                        captureStatus.latchedValue - origin);

        printf("Latched encoder position: %.0lf\n",
                        captureStatus.latchedValue);

    }

}
```

See Also

MPICaptureState

MPICaptureTrigger

Definition

```
typedef struct MPICaptureTrigger {
    MPI_BOOL enabled;
    MPI_BOOL invert;
} MPICaptureTrigger;
```

Change History: Modified in the 03.03.00

Description

The **MPICaptureTrigger** 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

Definition

```
typedef struct MPICaptureTriggerGlobal {
    MPI_BOOL   enabled;
} MPICaptureTriggerGlobal;
```

Change History: Modified in the 03.03.00

Description

The **MPICaptureTriggerGlobal** 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

Definition

```
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.

Remarks

Time-based capture will only work correctly if the speed of an axis is less than 344 million counts per second.

See Also

MPICaptureConfig

MPICaptureNOT_MAPPED

Definition

```
#define MPICaptureNOT_MAPPED (-1)
```

Description

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

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.

| Error Messages |

Methods

Create, Delete, Validate Methods

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

Configuration and Informational Methods

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

Other Methods

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

Data Types

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

meiCommandCreate

Declaration

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

Required Header: stdmei.h

Description

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

CommandCreate is the equivalent of a C++ constructor.

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

See Also

mpiCommandDelete | mpiCommandValidate

mpiCommandDelete

Declaration

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

Required Header: stdmpi.h

Description

mpiCommandDelete deletes a Command object and invalidates its handle (***command***).

CommandDelete is the equivalent of a C++ destructor.

Return Values	
MPIMessageOK	

See Also

mpiCommandCreate | mpiCommandValidate

mpiCommandValidate

Declaration

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

Required Header: stdmpi.h

Description

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

command	a handle to the Command object

Return Values	
MPIMessageOK	

See Also

mpiCommandCreate | mpiCommandValidate

mpiCommandLabel

Declaration

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

Required Header: stdmpi.h

Description

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

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

Return Values	
MPIMessageOK	
pointer	to a Command's (***command***) label (that is in the location pointed to by ***label***)

See Also

mpiCommandCreate

mpiCommandParams

Declaration

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

Required Header: stdmpi.h

Description

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

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

Return Values	
MPIMessageOK	
Command (*command*) parameters	in the structure pointed to by ***params***

See Also

mpiCommandCreate | MPICommandParams

mpiCommandType

Declaration

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

Required Header: stdmpi.h

Description

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

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

Return Values	
MPIMessageOK	
Command (*command*) parameters	in the location pointed to by ***type***

See Also

mpiCommandCreate | MPICommandType

meiCommandAxisListGet

Declaration

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

Required Header: stdmei.h

Description

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

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

Return Values	
MPIMessageOK	

See Also

MPICommand | MPIAxis | MPIMotion

MPICommandAddress

Definition

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

Description

MPICommandAddress 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

Definition

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

Description

MPICommandConstant 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

Definition

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

Description

MPICommandExpr 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

Definition

```
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

Definition

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

Change History: Modified in the 03.03.00

Description

MPICommandMotion 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(...), MPIActionABORT for details.
MPICommandMotionE_STOP	Commands an E-Stop action on the motion supervisor associated with the motion object. See mpiMotionAction(...), MPIActionE_STOP for details.
MPICommandMotionE_STOP_MODIFY	Commands an E-Stop Modify action on the motion supervisor associated with the motion object. See mpiMotionAction(...), MPIActionE_STOP_MODIFY for details.

MPICommandMotionE_STOP_ABORT	Commands an E-Stop, then Abort action on the motion supervisor associated with the motion object. See mpiMotionAction(...), MPIActionE_STOP_ABORT 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(...), MPIActionE_STOP_CMD_EQ_ACT 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

Definition

```
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 divison. Equivalent to the C operator (**/**).
MPICommandOperatorAND	Performs a logical AND. Equivalent to the C operator (&).
MPICommandOperatorOR	Performs a logical OR. Equivalent to the C operator (**|**).
MPICommandOperatorXOR	Performs a logical XOR. Equivalent to the C operator (**^**).

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

See Also

MPICommand | MPICommandExpr | MPICommandParams

MPICommandParams

Definition

```
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

MPICommandParams 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 *goto* statement) if a particular comparison evaluates to TRUE: branch to **label** on **expr** If *label* = NULL, then no more commands will be executed if the comparison evaluates to TRUE. **branch.control** is currently not supported and is reserved for future use.	MPICommandTypeBRANCH MPICommandTypeBRANCH_REF MPICommandTypeBRANCH_FLOAT MPICommandTypeBRANCH_FLOAT_REF
branchEvent	Branch to a particular command (similar to a *goto* statement) if a particular event occurs or has occured: branch to ***label*** on **MPIEventMask**(*handle*) **oper** *mask* If *label* = NULL, then no more commands will be executed if a particular event occurs or has occured.	MPICommandTypeBRANCH_EVENT
branchIO	Branch to a particular command (similar to a goto statement) if a particular i/o state matches a specified condition: branch to *label* on *Io.input* **oper** *mask* If *label* = NULL, then no more commands will be executed if a particular i/o state matches a specified condition.	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: *Io.output* = *Io.output* **oper** *mask*	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 *delay* 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 *expr* **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 *Io.input* **oper** *mask*	MPICommandTypeWAIT_IO

See Also

MPICommand | MPICommandType | mpiCommandCreate | mpiCommandParams

MPICommandType

Definition

```
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

MPICommandType 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(...).

Element	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 occured.	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

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.

See Also:
Configuring the Compensator Objects for Operation
Determining Required Compensator Table Size
Loading the Compensation Table
Setting up an area for 2D Position Compensation

| Error Messages |

Methods

Create, Delete, Validate Methods

mpiCompensator**Create**	Create Compensator object
mpiCompensatorl**Delete**	Delete Compensator object
mpiCompensator**Validate**	Validate Compensator object

Configuration and Information Methods

mpiCompensator**ConfigGet**	Get Compensator configuration
mpiCompensator**ConfigSet**	Set Compensator configuration
meiCompensator**Info**	Get Compensator information
meiCompensator**TableGet**	Get Compensator table
meiCompensator**TableSet**	Set Compensator table

Memory Methods

meiCompensator**Memory**	Set address to be used to access Compensator memory
meiCompensator**MemoryGet**	Get bytes of Compensator memory and place it into application memory
meiCompensator**MemorySet**	Put (set) bytes of application memory into Compensator memory

Relational Methods

meiCompensator**Control**	Return handle of Control object associated with Compensator
meiCompensator**Number**	Get number of Compensator

Data Types

- MPICompensator**Config**
- MPICompensator**Dimension**
- MPICompensator**Info**
- MPICompensator**InputAxis**
- MPICompensator**Message**
- MPICompensator**Range**

Constants

- MPICompensator**DimensionsMAX**

mpiCompensatorCreate

Declaration

```
MPICompensator  mpiCompensatorCreate(MPIControl  control,
                                     long        number);
```

Required Header: stdmpi.h

Description

mpiCompensatorCreate creates a Host Compensator object associated with the compensation object identified by ***number*** located on motion controller ***control***. CompensatorCreate is the equivalent of a C++ constructor.

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.

Return Values	
handle	handle to a Compensator object
MPIHandleVOID	if the object could not be created

See Also

mpiCompensatorDelete | mpiCompensatorValidate | MPIControlConfig

mpiCompensatorDelete

Declaration

```
long  mpiCompensatorDelete(MPICompensator  compensator);
```

Required Header: stdmpi.h

Description

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

CompensatorDelete is the equivalent of a C++ destructor.

Return Values	
MPIMessageOK	

See Also

mpiCompensatorCreate | mpiCompensatorValidate

mpiCompensatorValidate

Declaration

```
long  mpiCompensatorValidate(MPICompensator  compensator);
```

Required Header: stdmpi.h

Description

mpiCompensatorValidate validates the Compensator object (***compensator***) and its handle. Always call mpiCompensatorValidate after creating a new Compensator object.

Return Values	
MPIMessageOK	
MPICompensatorMessageCOMPENSATOR_INVALID	
MPICompensatorMessageNOT_ENABLED	

See Also

mpiCompensatorCreate | mpiCompensatorDelete

mpiCompensatorConfigGet

Declaration

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

Required Header: stdmpi.h

Description

mpiCompensatorConfigGet 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	
MPIMessagePARAM_INVALID	

See Also

mpiCompensatorConfigSet | MEICompensatorConfig

mpiCompensatorConfigSet

Declaration

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

Required Header: stdmpi.h

Description

mpiCompensatorConfigSet 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).

Remarks

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

Return Values	
MPIMessageOK	
MPIMessagePARAM_INVALID	
MPIMessageARG_INVALID	See MPICompensatorConfig.
MPICompensatorMessageDIMENSION_NOT_SUPPORTED	
MPICompensatorMessageAXIS_NOT_ENABLED	
MPICompensatorMessagePOSITION_DELTA_INVALID	
MPICompensatorMessageTABLE_SIZE_ERROR	

See Also

mpiCompensatorConfigGet | MEICompensatorConfig

mpiCompensatorInfo

Declaration

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

Required Header: stdmpi.h

Description

mpiCompensatorInfo 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	
MPICompensatorMessageNOT_CONFIGURED	

See Also

MPICompensatorInfo | MPIControlConfig | mpiCompensatorTableGet | mpiCompensatorTableSet | mpiCompensatorInfo

mpiCompensatorTableGet

Declaration

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

Required Header: stdmpi.h

Description

mpiCompensatorTableGet 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	
MPIMessagePARAM_INVALID	

See Also

mpiCompensatorTableSet | MPICompensatorConfig

mpiCompensatorTableSet

Declaration

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

Required Header: stdmpi.h

Description

mpiCompensatorTableSet 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	
MPIMessagePARAM_INVALID	

See Also

mpiCompensatorTableGet | MPICompensatorConfig

mpiCompensatorMemory

Declaration

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

Required Header: stdmpi.h

Description

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

Return Values	
MPIMessageOK	

See Also

mpiCompensatorMemoryGet | mpiCompensatorMemorySet

mpiCompensatorMemoryGet

Declaration

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

Required Header: stdmpi.h
Change History: Modified in the 03.03.00

Description

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

Return Values	
MPIMessageOK	

See Also

mpiCompensatorMemorySet | mpiCompensatorMemory

mpiCompensatorMemorySet

Declaration

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

Required Header: stdmpi.h
Change History: Modified in the 03.03.00

Description

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

Return Values	
MPIMessageOK	

See Also

mpiCompensatorMemoryGet | mpiCompensatorMemory

mpiCompensatorControl

Declaration

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

Required Header: stdmpi.h

Description

mpiCompensatorControl 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 *compensator* is invalid

See Also

mpiCompensatorCreate | mpiControlCreate

mpiCompensatorNumber

Declaration

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

Required Header: stdmpi.h

Description

mpiCompensatorNumber 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	

See Also

mpiCompensatorCreate

MPICompensatorConfig

Definition

```
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

Definition

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

Description

MPICompensatorDimension an enumeration of valid Compensator dimensions.

MPICompensatorDimensionX	First Compensating Dimension
MPICompensatorDimensionY	Second Compensating Dimension

See Also

MPICompensatorConfig | MPICompensatorInfo

MPICompensatorInfo

Definition

```
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

Definition

```
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.
positionDelta	Spacing between compensation positions on the compensating axis: **positionDelta** must meet some specifications: • positionDelta must be an exact multiple of the range (i.e. ((range. positionMax – range.positionMin) / positionDelta) must be an integer value). • positionDelta must be greater than zero. • positionDelta must be greater than (range.positionMax - range. positionMin).

See Also

MPICompensatorConfig | mpiCompensatorConfigGet | mpiCompensatorConfigSet

MPICompensatorMessage

Definition

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

Description

MPICompensatorMessageCOMPENSATOR_INVALID

The compensator number is not valid. This message code is returned by mpiCompensatorCreate(...) if the compensator number is less than 0 or greater than MPIControlMAX_COMPENSATORS.

MPICompensatorMessageNOT_CONFIGURED

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

MPICompensatorMessageNOT_ENABLED

The compensator is not available on the controller. This message code is returned by mpiCompensatorValidate(…) if the compensator number is not within the range of enabled compensators on the controller. To correct the problem, use MPIControlConfig to configure the compensatorCount to be greater than the required compensator number.

MPICompensatorMessageAXIS_NOT_ENABLED

The axis is not available on the controller. This message coder is returned by mpiCompensatorConfigSet(...) if the axisOutNumber or any of the inputAxis[n].axisNumbers are not within the range of enabled axes on the controller. To correct the problem, use MPIControlConfig to configure the axisCount to be greater than the required axis number.

MPICompensatorMessageTABLE_SIZE_ERROR

The host compenstation table will not fit within the controller's configured compensation table. See Determining Required Compensation Table Size.

MPICompensatorMessagePOSITION_DELTA_INVALID

The positionDelta is either out of range or is not a multiple of the range. This message code is returned by mpiCompensatorConfigSet(…). To correct the problem, check the valid range values for MPICompensatorInputAxis.

MPICompensatorMessageDIMENSION_NOT_SUPPORTED

The dimensionCount is out of range. This message code is returned by mpiCompensatorConfigSet(…). To correct the problem, check the valid range values for MPICompensatorConfig.

See Also

MPICompensatorRange

Definition

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

Change History: Modified in the 03.04.00.

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

MPICompensatorDimensionsMAX

Definition

```
#define MPICompensatorDimensionsMAX (MPICompensatorDimensionLAST)
```

Description

MPICompensatorDimensionMAX defines the maximum number of dimensions supported by the Compensator object's compensation tables. Currently, the maximum dimension value is 2.

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: Points = (positionMax-positionMin) / positionDelta + 1

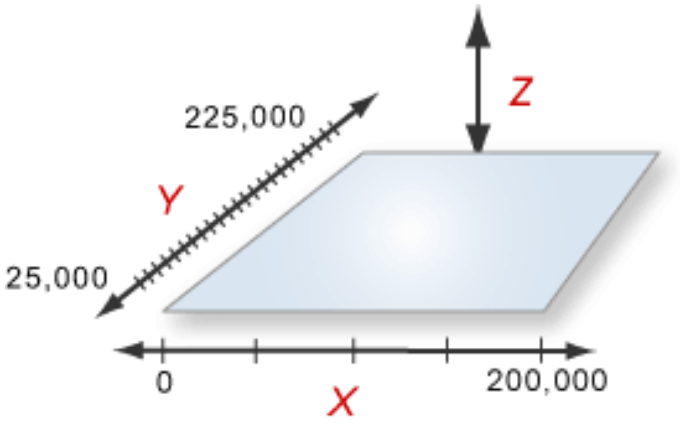
2D Compensation: Points = PointsX * PointsY

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 = 200000, Xdelta = 50000
Ymin = 25000, Ymax = 225000, Ydelta = 10000



For this table our X & Y dimensions are:
X_DIM = (200000-0)/50000 + 1 = 5
Y_DIM = (225000-25000)/10000 +1 = 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.

Configuring the Compensator Objects for Operation

After determining the required compensator table size, we need to configure both the embedded compensation tables on controller and the MPI Compensator object.

We will illustrate how to do this using the X-Y-Z system defined in the Determining Required Compensator Table Size section.

Configuring Controller Compensation Table
From our example in the previous section we have calculated that we need at least a point count of 105 to hold all of our measured compensation points (Acquiring and loading compensation points will be described in in the next section). First we need tell the motion controller to allocate memory space to hold the compensation table. We also need to enable a compensator since compensator objects are disabled on the controller by default. For an example, see the code below.

```
MPIControlConfig config;
long returnValue; returnValue =
    mpiControlConfigGet(control,
                        &config,
                        NULL);

if (returnValue == MPIMessageOK) {
    /* configure first compensator table size so our 2D array will fit */
    config.compensatorCount = 1;
    config.compensatorPointCount[0] = 105;

    /*
     * WARNING: this is a low-level configuration that will
     * reinitialize the controller's dynamic memory buffers!
     * Only preform this operation at system initialization.
    */
     returnValue =
         mpiControlConfigSet(control,
                             &config,
                             NULL);
}
```

The comment above reminds us that calling mpiControlConfigSet(...) will reallocate dynamic memory. Reallocation of dynamic memory affects other objects on the controller, so it should only be done during system initialization and not during the execution of a move.

Configuring the MPI Compensator Object
Continuing with our example, we will now assume that our axis numbers for axis X, Y, and Z are 0, 1, & 2 respectively. If we also assume that the MPI Compensator object has already been created, the code to configure the object would look like the following:

```
if (returnValue == MPIMessageOK) {
       MPICompensatorConfig config;

       returnValue =
              mpiCompensatorConfigGet(compensator,
                                      &config,
                                      NULL);
}

       if (returnValue == MPIMessageOK) {
              config.dimensionCount = 2;

              /* configure first compensating (input) axis */
              config.inputAxis[0].axisNumber = 0;
              config.inputAxis[0].range.positionMin = 0;
              config.inputAxis[0].range.positionMax = 250000;
              config.inputAxis[0].positionDelta = 50000;

              /* configure second compensating (input) axis */
              config.inputAxis[1].axisNumber = 1;
              config.inputAxis[1].range.positionMin = 25000;
              config.inputAxis[1].range.positionMax = 225000;
              config.inputAxis[1].positionDelta = 10000;

              /* configure compensated (out) axis */
              config.outputAxisNumber = 2;

              returnValue =
                  mpiCompensatorConfigSet(compensator,
                                          &config,
                                          NULL); }
```

Once we have the Compensation table allocated and have a configured Compensation object, the last step is to Load the Compensation Table.

Loading the Compensation Table

Next we need to somehow acquire high precision distance measurements (via interferometer, etc.) to the surface at each of the X-Y locations in the compensation area, and store the X and Y offset positions.

Once you've obtained these positions, they will need to be loaded into our previously configured compensation table (See Determining Required Compensator Table Size). Continuing with our original example let's assume that our measurements are as defined by the following table below (taken from the comp.c sample application):

```
long compensatorTable[21][5] =
{
      { 0,    0,    0,     0,     0, },
      { 100,  200,  -200,  -100,  0, },
      { 200,  400,  -400,  -200,  0 },
      { 300,  600,  -600,  -300,  0 },
      { 400,  800,  -800,  -400,  0 },
      { 500,  1000, -1000, -500,  0 },
      { 600,  1200, -1200, -600,  0 },
      { 700,  1400, -1400, -700,  0 },
      { 800,  1600, -1600, -800,  0 },
      { 900,  1800, -1800, -900,  0 },
      { 1000, 2000, -2000, -1000, 0 },
      { 900,  1800, -1800, -900,  0 },
      { 800,  1600, -1600, -800,  0 },
      { 700,  1400, -1400, -700,  0 },
      { 600,  1200, -1200, -600,  0 },
      { 500,  1000, -1000, -500,  0 },
      { 400,  800,  -800,  -400,  0 },
      { 300,  600,  -600,  -300,  0 },
      { 200,  400,  -400,  -200,  0 },
      { 100,  200,  -200,  -100,  0 },
      { 0,    0,    0,      0,    0 },
};
```

Interpreting compensator table above:

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 = 200000, Xdelta = 50000

Ymin = 25000, Ymax = 225000, Ydelta = 10000

Y Values	X Values				
	0	50000	100000	150000	200000
25000	0	0	0	0	0
35000	100	200	-200	-100	0
45000	200	400	-400	-200	0
55000	300	600	-600	-300	0
65000	400	800	-800	-400	0
75000	500	1000	-1000	-500	0
85000	600	1200	-1200	-600	0
95000	700	1400	-1400	-700	0
105000	800	1600	-1600	-800	0
115000	900	1800	-1800	-900	0
125000	1000	2000	-2000	-1000	0
135000	900	1800	-1800	-900	0
145000	800	1600	-1600	-800	0
155000	700	1400	-1400	-700	0
165000	600	1200	-1200	-600	0
175000	500	1000	-1000	-500	0
185000	400	800	-800	-400	0
195000	300	600	-600	-300	0
205000	200	400	-400	-200	0
215000	100	200	-200	-100	0
225000	0	0	0	0	0

To load the above compensation table, execute the following code:

```
returnValue = mpiCompensatorTableSet(compensator,
                                     (long*)compensatorTable);
```

Once the compensation positions are loaded, the compensation will be applied to the Z-axis' position feedback loop every servo cycle.

NOTE: No more interpolated compensation of the Z-axis will occur outside of the defined compensation range. Therefore, the compensation of the Z-axis will remain fixed outside of this range.

Setting up an area for 2D Position Compensation

The XMP has 2D compensation capabilities. The user-supplied compensation table is downloaded into XMP memory. The XMP automatically applies this compensation information to optimize motion profiles in real time.