

# Control Objects

## Introduction

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

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

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

Are you using TCP/IP and Sockets? If yes, [click here](#).

| [Error Messages](#) |

## Methods

### Create, Delete, Validate Methods

<a href="#">mpiControlCreate</a>	Create Control object
<a href="#">mpiControlDelete</a>	Delete Control object
<a href="#">mpiControlValidate</a>	Validate Control object

### Configuration and Information Methods

<a href="#">mpiControlAddress</a>	Get original address of Control object (when it was created)
<a href="#">mpiControlConfigGet</a>	Get Control config
<a href="#">mpiControlConfigSet</a>	Set Control config
<a href="#">mpiControlDigitalIn</a>	Read the current input state
<a href="#">mpiControlDigitalOutGet</a>	Read the current output state
<a href="#">mpiControlDigitalOutSet</a>	Change the state of output bit
<a href="#">meiControlExtMemAvail</a>	Gets the amount of external memory available
<a href="#">mpiControlFlashConfigGet</a>	Get Control flash config
<a href="#">mpiControlFlashConfigSet</a>	Set Control flash config
<a href="#">meiControlFPGADefaultGet</a>	Creates a default FPGA filename
<a href="#">meiControlFPGAFileOverride</a>	Replace the default FPGA file
<a href="#">meiControlGateGet</a>	Get the closed state (TRUE or FALSE)

<a href="#"><u>meiControlGateSet</u></a>	Set the closed state (TRUE or FALSE)
<a href="#"><u>meiControlInfo</u></a>	Retrieve information about an MEI motion controller
<a href="#"><u>meiControlSampleCounter</u></a>	Write the number of servo cycles (samples)
<a href="#"><u>meiControlSampleRate</u></a>	Write current sample rate (Hz) of the controller's processor
<a href="#"><u>meiControlSamplestoSeconds</u></a>	Converts samples to seconds
<a href="#"><u>meiControlSampleWait</u></a>	Wait for count samples
<a href="#"><u>meiControlSecondstoSamples</u></a>	Converts seconds to samples
<a href="#"><u>mpiControlStatus</u></a>	Get Control's status
<a href="#"><u>mpiControlType</u></a>	Get type of Control object (used to create Command object)

## I/O Methods

<a href="#"><u>mpiControlDigitalIn</u></a>	Reads the current input state
<a href="#"><u>mpiControlDigitalOutGet</u></a>	Read the current output state
<a href="#"><u>mpiControlDigitalOutSet</u></a>	Change the state of output bit

## Event Methods

<a href="#"><u>mpiControlEventNotifyGet</u></a>	Set the control event that will cause the firmware to generate an interrupt
<a href="#"><u>mpiControlEventNotifySet</u></a>	Configure firmware to generate an interrupt
<a href="#"><u>mpiControlEventReset</u></a>	Reset the events

## Memory Methods

<a href="#"><u>mpiControlMemory</u></a>	Get address of Control memory
<a href="#"><u>mpiControlMemoryAlloc</u></a>	Allocate bytes of firmware memory
<a href="#"><u>mpiControlMemoryCount</u></a>	Get number of bytes available in firmware
<a href="#"><u>mpiControlMemoryFree</u></a>	Free bytes of firmware memory
<a href="#"><u>mpiControlMemoryGet</u></a>	Copy count bytes of Control memory to application memory
<a href="#"><u>mpiControlMemorySet</u></a>	Copy count bytes of application memory to Control memory
<a href="#"><u>meiControlMemoryToFile</u></a>	

## Relational Methods

<a href="#"><u>meiControlPlatform</u></a>	Return a handle to the Platform object
---	--

## Action Methods

<a href="#"><u>mpiControlCycleWait</u></a>	Wait for Control to execute count cycles
<a href="#"><u>mpiControlInit</u></a>	Initialize Control object
<a href="#"><u>mpiControlInterruptEnable</u></a>	Enable interrupts to Control object
<a href="#"><u>mpiControlInterruptWait</u></a>	Wait for controller interrupt
<a href="#"><u>mpiControlInterruptWake</u></a>	Wake all threads waiting for controller interrupt
<a href="#"><u>meiControlRecorderCancel</u></a>	Cancel the reservation for an abandoned recorder.
<a href="#"><u>meiControlRecorderStatus</u></a>	Read the recorder's status.

[mpiControlReset](#)[meiControlSampleWait](#)[meiControlVersionMismatchOverride](#)

Reset controller hardware

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

## Data Types

[MPIControlAddress](#)[MPIControlConfig](#) / [MEIControlConfig](#)[MPIControlFanStatusFlag](#)[MPIControlFanStatusMask](#)[MEIControlFPGA](#)[MEIControlInfo](#)[MEIControlInfoDriver](#)[MEIControlInfoFirmware](#)[MEIControlInfoFirmwareZMP](#)[MEIControlInfoIo](#)[MEIControlInfoIoDigitalIn](#)[MEIControlInfoIoDigitalOut](#)[MEIControlInfoHardware](#)[MEIControlInfoMpi](#)[MEIControlInfoPld](#)[MEIControlInfoRincon](#)[MEIControlInput](#)[MEIControlIoBit](#)[MPIControlMessage](#) / [MEIControlMessage](#)[MPIControlMemoryType](#)[MEIControlOutput](#)[MPIControlStatus](#)[MEIControlTrace](#)[MPIControlType](#)

## Constants

[MPIControlInMAX](#)[MPIControlOutMAX](#)[MPIControlMAX\\_AXES](#)[MPIControlMAX\\_COMPENSATORS](#)

[MPIControlMAX\\_RECORDERS](#)

[MPIControlMIN\\_AXIS\\_FRAME\\_COUNT](#)

[MPIControlMIN\\_SAMPLE\\_RATE](#)

[MEIControlSTRING\\_MAX](#)

## Macros

[mpiControlFanStatusMaskBIT](#)

# mpiControlMemory

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

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

### Return Values

[MPIMessageOK](#)

## Sample Code

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

long returnValue, tempBuffer;

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

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

```
tempBuffer++;

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

## See Also

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

# mpiControlInterruptEnable

## Declaration

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

**Required Header:** stdmpi.h

## Description

If "enable" is **TRUE**, then **mpiControlInterruptEnable** enables interrupts from the motion controller.

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

### Return Values

[MPIMessageOK](#)

## See Also

[mpiControlInterruptWait](#) | [mpiControlInterruptWake](#)

# mpiControlCreate

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

The type parameter determines the form of the address parameter:

<i>If the "type" parameter is</i>	<i>Then the form of the "address" parameter is</i>
MPIControlTypeDEFAULT	implementation-specific
MPIControlTypeMAPPED	MPIControlAddress.mapped
MPIControlTypeDEVICE	MPIControlAddress.device
MPIControlTypeCLIENT	MPIControlAddress.client

## Remarks

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

If <b>MPIControlType</b> is	And <b>MPIControlAddress</b> is	Then the <b>Board Number</b> is	And the <b>"address" parameter</b> to be used is
DEFAULT	Null address	0 address.number	default address parameter default address parameter
DEVICE	Null address	0 address.number	default device driver address.type.device (if address.type.device is Null, then default device driver)
CLIENT	address	specified by server	address.type.client ( <b>NOTE:</b> address.number should be set to zero)

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

### Return Values

<b>handle</b>	to a Control object
<b>MPIHandleVOID</b>	if the object could not be created

## Sample Code

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

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

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

A simple case where board #1 is desired is:

```
{
    MPIControl control;
    MPIControlAddress address;

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

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

## See Also

[MPIControl](#) | [MPIControlAddress](#) | [MPIControlType](#) | [mpiControlValidate](#)  
[mpiControlInit](#) | [mpiControlDelete](#)

# mpiControlValidate

## Declaration

```
long mpiControlValidate(MPIControl control);
```

**Required Header:** stdmpi.h

## Description

**mpiControlValidate** validates the control object and its handle.

### Return Values

[MPIMessageOK](#)

## See Also

[mpiControlCreate](#) | [mpiControlDelete](#)

# mpiControlInit

## Declaration

```
long mpiControlInit(MPIControl control)
```

**Required Header:** stdmpi.h

## Description

**mpiControlInit** initializes the control object. ControlInit must be called after mpiControlCreate(...) and before any other MPI calls in your application. ControlInit establishes communication with the motion controller hardware and initializes any SynqNet networks connected to the controller. Controller communication can occur through direct memory access, device driver, or remote via client/server.

<b>control</b>	a handle to the Control object
----------------	--------------------------------

### Return Values

[MPIMessageOK](#)

[MPIControlMessageLIBRARY\\_VERSION](#)

[MPIControlMessageADDRESS\\_INVALID](#)

[MPIControlMessageCONTROL\\_INVALID](#)

[MPIControlMessageTYPE\\_INVALID](#)

[MPIControlMessageCONTROL\\_NUMBER\\_INVALID](#)

[MEIControlMessageFIRMWARE\\_INVALID](#)

[MEIControlMessageSYNQNET\\_STATE](#)

[MEIPacketMessageADDRESS\\_INVALID](#)

[MEIPlatformMessageDEVICE\\_INVALID](#)

[MEIPlatformMessageDEVICE\\_MAP\\_ERROR](#)

[MEISynqNetMessageSTATE\\_ERROR](#)

## Sample Code

```
MPIControl    control; /* motion controller object handle */

control =
    mpiControlCreate(MPIControlTypeDEFAULT, NULL);
mpiControlValidate(control);

/* Initialize motion controller */
returnValue =
    mpiControlInit(control);
```

## See Also

[mpiControlCreate](#) | [mpiControlDelete](#)

# mpiControlDelete

## Declaration

```
long mpiControlDelete(MPIControl control);
```

**Required Header:** stdmpi.h

## Description

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

Return Values	
<a href="#">MPIMessageOK</a>	

## See Also

[mpiControlCreate](#) | [mpiControlValidate](#)

# mpiControlAddress

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

Return Values	
<a href="#">MPIMessageOK</a>	

## See Also

# mpiControlConfigGet

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

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

## Remarks

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

### Return Values

[MPIMessageOK](#)

## Sample Code

```
/*
   Write a value to element index of the user buffer.
   Make sure to save topology to flash before doing this.
*/
void write2UserBuffer(MPIControl control, long value, long index)
{
    MPIControlConfig config;
    MEIControlConfig external;
    long returnValue;

    if((index < MEIXmpUserDataSize) && (index >= 0))
    {
        /* Make sure to save topology to flash before doing this */
        returnValue = mpiControlConfigGet(control,
```

```
        &config,  
        &external);  
    msgCHECK(returnValue);  
  
    external.UserBuffer.Data[index] = value;  
  
    returnValue = mpiControlConfigSet(control,  
        &config,  
        &external);  
    msgCHECK(returnValue);  
    }  
}
```

## See Also

[mpiControlConfigSet](#) | [MEIControlConfig](#) | [Dynamic Allocation of External Memory Buffers](#)

# mpiControlConfigSet

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

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

**WARNING:** mpiControlConfigSet(...) is a controller-wide routine that will clear other controller object configurations and potentially force a reset of the SynqNet network. This method should be executed in your application before configuring any other MPI objects. For information regarding which configurations force a reset of the SynqNet network, please see the Warning message for [MPIControlConfig](#) / [MEIControlConfig](#).

## Remarks

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

### Return Values

[MPIMessageOK](#)

## Sample Code

```
/*  
    Write a value to element index of the user buffer.  
    Make sure to save topology to flash before doing this.  
*/  
void write2UserBuffer(MPIControl control, long value, long index)  
{  
    MPIControlConfig config;  
    MEIControlConfig external;  

```

## See Also

[mpiControlConfigGet](#) | [MEIControlConfig](#) | [Dynamic Allocation of External Memory Buffers](#)

# mpiControlDigitalIn

## Declaration

```
long mpiControlDigitalIn(MPIControl control,
                        long bitStart,
                        long bitCount,
                        unsigned long *state);
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.03.00

## Description

**mpiControlDigitalIn** reads the current input state of one or more controller inputs.

<b>control</b>	a handle to the Control object
<b>bitStart</b>	the first controller input bit that will be returned by the function.
<b>bitCount</b>	the number of controller bits that will be returned by the function.
<b>*state</b>	the address of the current state of the inputs that is returned.

### Return Values

[MPIMessageOK](#)

## Sample Code

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

```
unsigned long input1;
mpiControlDigitalIn( control, 1, 1, &input1 );
```

## See Also

[Controller I/O](#) | [mpiControlDigitalOutSet](#) | [mpiControlDigitalOutGet](#)

# mpiControlDigitalOutSet

## Declaration

```
long  mpiControlDigitalOutSet(MPIControl    control,
                              long           bitStart,
                              long           bitCount,
                              unsigned long  state,
                              MPI_BOOL      wait);
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.03.00

## Description

**mpiControlDigitalOutSet** function changes the state of one or more general purpose bits.

<b>control</b>	a handle to the Control object
<b>bitStart</b>	the first controller output bit that will be returned by the function.
<b>bitCount</b>	the number of controller output bits that will be returned by the function.
<b>state</b>	the new state of the general purpose bits that will be returned.
<b>wait</b>	a Boolean flag indicating if the new output state is applied immediately or a wait is inserted so that any previously set output is transmitted to the controller hardware before applying the new output state.

### Return Values

[MPIMessageOK](#)

## Sample Code

The next piece of code shows how to set three controller outputs (2, 3 and 4):

```
mpiControlDigitalOutSet( control, 2, 3, 7 );
```

## See Also

# mpiControlDigitalOutGet

## Declaration

```
long  mpiControlDigitalOutGet(MPIControl    control,
                              long           bitStart,
                              long           bitCount,
                              unsigned long *state);
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.03.00

## Description

**mpiControlDigitalOutGet** function reads the current output state of one or more controller output bits.

<b>control</b>	a handle to the Control object
<b>bitStart</b>	the first controller output bit that will be returned by the function.
<b>bitCount</b>	the number of controller output bits that will be returned by the function.
<b>*state</b>	the address of the current state of the controller output bits that will be returned.

### Return Values

[MPIMessageOK](#)

## Sample Code

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

```
unsigned long output1;
mpiControlDigitalOutGet( control, 1, 1, &output1 );
```

## See Also

[Controller I/O](#) | [mpiControlDigitalOutSet](#) | [mpiControlDigitalIn](#)

# meiControlExtMemAvail

## Declaration

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

**Required Header:** stdmei.h

## Description

**meiControlExtMemAvail** gets the amount of external memory available on an XMP-Series controller. It puts the number of words (8 bit) in the location pointed to by size. Since the XMP is a 32 bit controller, the number of 32 bit words available is equal to the value of size divided by 4. The value of size is useful for setting things that use the external memory, such as the Recorder.

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

<b>Return Values</b>	
<a href="#">MPIMessageOK</a>	

## Sample Code

### Example:

```
/* Prints the size of the available external memory size */
void printExternalMemorySize(MPIControl control)
{
    long returnValue;
    long size;

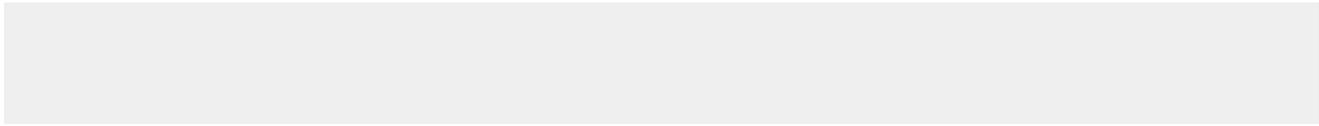
    returnValue = meiControlExtMemAvail(control, &size);

    msgCHECK(returnValue);

    printf("size %d (8 bit), %d (32 bit)", size, size / 4);
}
```

### Output:

```
C:\out\extmemavail\Debug>extmemavail
size 238008 (8 bit), 59502 (32 bit)
```



## See Also

[MPIControlConfig](#)

# mpiControlFlashConfigGet

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

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

## Remarks

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

### Return Values

[MPIMessageOK](#)

## Sample Code

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

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

        external.UserBuffer.Data[index] = value;

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

## See Also

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

# mpiControlFlashConfigSet

## Declaration

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

Required Header: stdmpi.h

## Description

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

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

## Remarks

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

### Return Values

[MPIMessageOK](#)

## Sample Code

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

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

    external.UserBuffer.Data[index] = value;

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

## See Also

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

# meiControlFPGADefaultGet

## Declaration

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

**Required Header:** stdmei.h

## Description

**meiControlFPGADefaultGet** creates a default FPGA filename based on the *socketInfo*.

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

### Return Values

[MPIMessageOK](#)

## See Also

# meiControlFPGADefaultOverride

## Declaration

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

**Required Header:** stdmei.h

## Description

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

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

### Return Values

[MPIMessageOK](#)

## See Also

# meiControlGateGet

## Declaration

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

**Required Header:** stdmei.h

**Change History:** Modified in the 03.03.00

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

[meiControlGateSet](#)

# meiControlGateSet

## Declaration

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

**Required Header:** stdmei.h

**Change History:** Modified in the 03.03.00

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

[meiControlGateGet](#)

# meiControlInfo

## Declaration

```
long meiControlInfo(MPIControl control,  
                   MEIControlInfo *info);
```

**Required Header:** stdmei.h

## Description

**meiControlInfo** retrieves information about an MEI motion controller.

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

<b>Return Values</b>	
<a href="#">MPIMessageOK</a>	
<b>MPIHandleVOID</b>	if <i>control</i> is invalid

## See Also

# meiControlSampleCounter

## Declaration

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

**Required Header:** stdmei.h

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

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

# meiControlSecondsToSamples

## Declaration

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

**Required Header:** stdmei.h

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

[meiControlSampletoSeconds](#) | [meiControlSampleCounter](#) | [meiControlSampleWait](#)

# meiControlSamplesToSeconds

## Declaration

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

**Required Header:** stdmei.h

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

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

# meiControlSampleWait

## Declaration

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

Required Header: stdmei.h

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

[meiControlSampletoSeconds](#) | [meiControlSecondstoSamples](#) | [meiControlSampleCounter](#)

# meiControlSampleRate

## Declaration

```
long meiControlSampleRate(MPIControl control,
                          double *sampleRate)
```

**Required Header:** stdmei.h

## Description

**meiControlSampleRate** writes the current sample rate (Hz) of the controller's processor to the address pointed to by **sampleRate**. This is the same value returned in `MPIControlConfig.sampleRate` after `mpiControlConfigGet()` has been performed, but is also provided as this separate method to avoid the extra processing overhead of `mpiControlConfigGet`.

<b>control</b>	a handle to the Control object
<b>*sampleRate</b>	pointer to a double where the current sample rate will be stored.

### Return Values

[MPIMessageOK](#)

[MEIControlMessageFIRMWARE\\_VERSION\\_NONE](#)

[MEIControlMessageFIRMWARE\\_VERSION](#)

## See Also

[mpiControlConfigGet](#) | [MPIControlConfig](#) | [MEIControlMessage](#)

# mpiControlStatus

## Structure

```
long mpiControlStatus(MPIControl      control ,
                    MPIControlStatus *status ,
                    void                *external ) ;
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.02.00

## Description

**mpiControlStatus** gets a Control's (**control**) status and writes it to the structure pointed to by **status**, and also writes it into the implementation-specific structure pointed to by **external** (if **external** is not NULL).

<b>control</b>	a handle to a Control object.
<b>*status</b>	a pointer to MPIControlStatus structure.
<b>*external</b>	a pointer to an implementation-specific structure.

### Return Values

[MPIMessageOK](#)

[MPIMessageARG\\_INVALID](#)

## See Also

[MPIControl](#) | [MPIControlStatus](#)

# mpiControlType

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

# mpiControlEventNotifyGet

## Declaration

```
long mpiControlEventNotifyGet(MPIControl control,
                              MPIEventMask *eventMask,
                              void *external);
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.02.00

## Description

**mpiControEventNotifyGet** fills in the **eventMask** with the data indicating which control events will cause the firmware to generate an interrupt. If **external** is not NULL (it should be a pointer to a user supplied MEIEventNotifyData structure), then the function will fill out the structure with data from the firmware's control object.

<b>control</b>	a handle to the Control object
<b>*eventMask</b>	pointer to MPIEventMask structure.
<b>*external</b>	pointer to MEIEventNotifyData structure or NULL.

<b>Return Values</b>	
<a href="#">MPIMessageOK</a>	
<a href="#">MPIMessageARG_INVALID</a>	

## See Also

[mpiControlEventNotifySet](#) | [MEIEventNotifyData](#)

# mpiControlEventNotifySet

## Declaration

```
long mpiControlEventNotifySet(MPIControl control,
                              MPIEventMask eventMask,
                              void *external);
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.02.00

## Description

**mpiControEventNotifySet** configures the firmware to generate interrupts based on the control events indicated in the **eventMask**. If **external** is not NULL (it should be a pointer to a user supplied MEIEventNotifyData structure), then the data in the structure is written to the firmware's control object.

<b>control</b>	a handle to the Control object
<b>eventMask</b>	MPIEventMask structure.
<b>*external</b>	pointer to MEIEventNotifyData structure or NULL.

Return Values	
<a href="#">MPIMessageOK</a>	
<a href="#">MPIMessageARG_INVALID</a>	

## See Also

[mpiControlEventNotifyGet](#) | [MEIEventNotifyData](#)

# mpiControlEventReset

## Declaration

```
long mpiControlEventReset(MPIControl control,
                          MPIEventMask eventMask);
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.02.00

## Description

**mpiControlEventReset** resets (clears) the events indicated in the **eventMask** from the firmware's **control** object. Once cleared, the events can cause the firmware to generate an interrupt.

<b>control</b>	a handle to the Control object
<b>eventMask</b>	<a href="#">MPIEventMask</a> structure.

Return Values	
<a href="#">MPIMessageOK</a>	
<a href="#">MPIMessageARG_INVALID</a>	

## See Also

[mpiControlEventNotifyGet](#) | [mpiControlEventNotifySet](#) | [MPIEventMask](#) | [mpiMotionEventReset](#) | [mpiMotorEventReset](#) | [mpiRecorderEventReset](#) | [mpiSequenceEventReset](#) | [meiSynqNetEventReset](#) | [meiSqNodeEventReset](#) | [mpiAxisEventReset](#)

# mpiControlMemoryGet

## Declaration

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

**Required Header:** stdmpi.h

**Change History:** Modified in the 03.03.00

## Description

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

### Return Values

[MPIMessageOK](#)

## Sample Code

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

long returnValue, tempBuffer;

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

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

tempBuffer++;

returnValue = mpiControlMemorySet(control,
    &buffer->UserBuffer.Data[0],
```

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

## See Also

[mpiControlMemorySet](#) | [mpiControlMemory](#) | [mpiControlMemoryAlloc](#) |  
[mpiControlMemoryCount](#) | [mpiControlMemoryFree](#)

# mpiControlMemorySet

## Declaration

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

**Required Header:** stdmpi.h

**Change History:** Modified in the 03.03.00

## Description

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

### Return Values

[MPIMessageOK](#)

## Sample Code

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

long returnValue, tempBuffer;

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

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

tempBuffer++;

returnValue = mpiControlMemorySet(control,
    &buffer->UserBuffer.Data[0],
```

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

## See Also

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

# mpiControlMemoryAlloc

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

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

# mpiControlMemoryCount

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

# mpiControlMemoryFree

## Declaration

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

**Required Header:** stdmpi.h

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

[mpiControlMemoryGet](#) | [mpiControlMemorySet](#) | [mpiControlMemoryAlloc](#) |  
[mpiControlMemoryCount](#) | [mpiControlMemory](#)

# meiControlMemoryToFile

## Declaration

```
long  meiControlMemoryToFile(MPIControl    control ,
                             const char    *fileName );
```

**Required Header:** stdmpi.h

**Change History:** Modified in the 03.03.00

## Description

**mpiControlMemoryToFile** creates a file with a copy of the current controller memory. The contents of this file may then later be viewed using the [VM3 utility](#). This is often useful for helping troubleshoot a problem that is difficult to understand.

<b>control</b>	a handle to the Control object.
<b>*filename</b>	The name of the file to be created which will hold the contents of controller memory.

### Return Values

[MPIMessageOK](#)

## Sample Code

```
long result =
    meiControlMemoryToFile(control, "controllerMemory.mem");
msgCHECK(result);
```

## See Also

# meiControlPlatform

## Declaration

```
MEIPlatform meiControlPlatform(MPIControl control)
```

**Required Header:** stdmei.h

## Description

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

<b>control</b>	a handle to the Control object
----------------	--------------------------------

### Return Values

<b>MPIPlatform</b>	handle to a Platform object
--------------------	-----------------------------

<b>MPIHandleVOID</b>	if <i>control</i> is invalid
----------------------	------------------------------

## See Also

[mpiControlCreate](#)

# meiControlCycleWait

## Declaration

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

**Required Header:** stdmei.h

## Description

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

Return Values	
<a href="#">MPIMessageOK</a>	

## See Also

# mpiControlInterruptWait

## Declaration

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

**Required Header:** stdmpi.h

**Change History:** Modified in the 03.03.00

## Description

**mpiControlInterruptWait** waits for an interrupt from the motion controller if interrupts are enabled. After the `ControlInterruptWait` method returns, if the location pointed to by *interrupted* contains **TRUE**, then an interrupt has occurred. After the `ControlInterruptWait` method returns, if the location pointed to by *interrupted* contains **FALSE**, then no interrupt has occurred, and the return of `ControlInterruptWait` was caused either by a call to **mpiControlInterruptWake(...)**.

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

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

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

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

### Return Values

[MPIMessageOK](#)

[MPIMessageTIMEOUT](#)

## See Also

[mpiControlInterruptWake](#) | [mpiControlInterruptEnable](#)

# mpiControlInterruptWake

## Declaration

```
long mpiControlInterruptWake(MPIControl control)
```

**Required Header:** stdmpi.h

## Description

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

### Return Values

[MPIMessageOK](#)

## See Also

[mpiControlInterruptWait](#) | [mpiControlInteruptEnable](#)

# meiControlRecorderCancel

## Declaration

```
long meiControlRecorderCancel(MPIControl control,
                              long recorderNumber);
```

**Required Header:** stdmei.h

**Change History:** Added in the 03.03.00

## Description

**meiControlRecorderCancel** provides a way to cancel the reservation for an abandoned recorder.

It is possible for a fatal error to occur in your application where `mpiRecorderDelete(...)` is not called, which will leave your recorders abandoned.

An abandoned recorder number cannot be reused until the recorder's reservation is canceled or the reservation is explicitly overwritten by specifying the recorder number (i.e. a number other than -1) when calling `mpiRecorderCreate(...)`.

Use `meiControlRecorderStatus(...)` to make sure you have no reason to believe a recorder is being used before canceling the recorders reservation.

<b>control</b>	a handle to the Control object.
<b>recorderNumber</b>	the index of the abandoned recorder object.

### Return Values

[MPIMessageOK](#)

[MPIMessageARG\\_INVALID](#)

If *recorderNumber* is less than 0 or greater than `MPIRecorderRECORDERS_MAX`

## See Also

[mpiRecorderStatus](#) | [mpiRecorderCreate](#) | [mpiRecorderDelete](#) | [meiControlRecorderStatus](#)

[recorderinuse.c](#)

# meiControlRecorderStatus

## Declaration

```
long meiControlRecorderStatus(MPIControl control,
                              long recorderNumber,
                              MPIRecorderStatus *status);
```

**Required Header:** stdmei.h

**Change History:** Added in the 03.03.00

## Description

**meiControlRecorderStatus** allows the recorder status to be read without actually creating a recorder object.

This is useful to help determine whether or not a recorder is abandoned. An abandoned recorder will usually not be running, yet will still be reserved. Another hint that a recorder is abandoned, is an enabled recorder that is full. This sort of behavior is what happens when a program crashes that has a recorder operating.

<b>control</b>	a handle to the Control object.
<b>recorderNumber</b>	the index of the recorder object.
<b>*status</b>	a pointer to the recorder's status structure.

### Return Values

[MPIMessageOK](#)

[MPIMessageARG\\_INVALID](#)

If *recorderNumber* is less than 0, greater than `MPIRecorderRECORDERS_MAX`, or if *\*status* == NULL.

## See Also

[meiControlRecorderCancel](#) | [mpiRecorderDelete](#) | [mpiRecorderCreate](#)

[recorderinuse.c](#)

# mpiControlReset

## Declaration

```
long mpiControlReset(MPIControl control)
```

**Required Header:** stdmpi.h

## Description

**mpiControlReset** resets the motion controller (*control*) board.

### Return Values

[MPIMessageOK](#)

## See Also

# mpiControlVersionMismatchOverride

## Declaration

```
long meiControlVersionMismatchOverride(MPIControl control);
```

**Required Header:** stdmei.h

## Description

**mpiControlValidate** overrides the version mismatch between the MPI and the controller.

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

### Return Values

[MPIMessageOK](#)

## See Also

# mpiControlIoSet

## Declaration

```
long mpiControlIoSet(MPIControl control,  
                    MPIControlIo *io);
```

**Required Header:** stdmpi.h

## Description

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

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

### Return Values

[MPIMessageOK](#)

## See Also

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

# MPIControlAddress

## Definition

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

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

**Change History:** Modified in the 03.03.00

## Description

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

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

## See Also

[MPIControl](#) | [MPIControlType](#) | [mpiControlCreate](#)

# MPIControlConfig / MEIControlConfig

## Definition: MPIControlConfig

```
typedef struct MPIControlConfig {
    long    axisCount;
    long    axisFrameCount[MPIControlMAX_AXES];
    long    captureCount;
    long    compareCount;
    long    compensatorCount;
    long    compensatorPointCount[MPIControlMAX_COMPENSATORS];
    long    cmdDacCount;
    long    auxDacCount;
    long    filterCount;
    long    motionCount;
    long    motorCount;
    long    recorderCount;
    long    recordCount[MPIControlMAX_RECORDERS];
    long    sequenceCount;
    long    userVersion;
    long    sampleRate;
} MPIControlConfig;
```

## Description

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

### **WARNING!**

`mpiControlConfigSet(...)` should ONLY be called during application initialization and NOT during motion. If the `sampleRate` or `TxTime` is changed, the SynqNet network will be shutdown and re-initialized with the new `sampleRate` and/or `TxTime`. If the `axisCount`, `axisFrameCount[]`, `compensatorCount`, `compensatorPointCount[]`, `recorderCount`, or `recordCount[]` is changed, then the controller's dynamic memory will be cleared and re-allocated with the new configuration. During the re-allocation, compensators, recorders, and axes are not available for application use.

<b>axisCount</b>	Number of axis objects enabled for the controller. The controller's axis object handles the trajectory calculations for command position. For simple systems, set the <i>axisCount</i> equal to the <i>motorCount</i> .
<b>axisFrameCount[]</b>	An array containing the number of frames for each axis frame buffer. Each frame is the size of MEIXmpFrame{ }. The controller's frame buffers are dynamically allocated by changing the axisFrameCount[]. A larger frame buffer may be required for long multi-point or cam motion profiles. Axes mapped to the same motion object <b>MUST</b> have the same frame buffer size. The default and minimum axis frame buffer size is 128. The valid range is from 16 to the available memory. Use <a href="#">meiControlExtMemAvail(...)</a> to determine the controller's available memory. Be sure to leave some free memory for potential future features.
<b>captureCount</b>	Number of capture objects enabled for the controller. The controller supports up to 32 captures. The controller's capture object manages the hardware resources to latch a motor's position feedback, triggered by a motor's input.
<b>compareCount</b>	Number of compare objects enabled for the controller. The controller's compare object manages the hardware resources to trigger a motor's output, triggered by a comparison between the motor's feedback and a pre-loaded position value.
<b>compensatorCount</b>	This value defines the number of enabled compensators.
<b>compensatorPointCount</b>	<p>The number of points in the compensation table for each compensator. Compensator tables get allocated on a per-compensator basis. Each compensator can have a different compensation table size as specified by the compensatorPointCount[n] value. See <a href="#">Determining Required Compensator Table Size</a> for more information.</p> <p>An array of the number of points in the compensation table for each compensator. Each point is 32bits. The controller's compensation tables are dynamically allocated by changing the <i>compensatorPointCount</i>. When using compensator objects, see <a href="#">Determining Required Compensator Table Size</a> for more information on a proper value for the point count.</p>
<b>cmdDacCount</b>	Number of command DACs (digital to analog converter) enabled for the controller. The controller's cmdDac transmits and scales a torque demand value to a SynqNet servo drive or a physical DAC circuit. There is one cmdDac per motor. Normally, the cmdDacCount should be equal to the motorCount.

<b>auxDacCount</b>	Number of auxilliary DACs (digital to analog converter) enabled for the controller. The controller's auxDac transmits and scales a torque demand value to a SynqNet servo drive or a physical DAC circuit. Auxilliary DACs can be used for sinusoidal motor commutation, where the cmdDac and auxDac provide the commutation phases. Or, auxilliary DACs can be used for general purpose analog outputs. There is one auxDac per motor.
<b>filterCount</b>	Number of filter objects enabled for a controller. The filter object handles the closed-loop servo calculations to control the motor. For simple systems, set the filterCount equal to the motorCount.
<b>motionCount</b>	Number of motion supervisor obejcts enabled for a controller. The controller's motion supervisor handles coordination of motion and events for an axis or group of axes. For simple systems, set the motionCount equal to the axisCount.
<b>motorCount</b>	Number of motor objects enabled for a controller. The controller's motor object handles the interface to the servo or stepper drive, dedicated I/O and general purpose motor related I/O. For simple systems, the motorCount should equal the number of physical motors connected to the controller (either directly or via SynqNet).
<b>recorderCount</b>	Number of data recorder objects enabled for a controller. The controller's recorder object handles collecting and buffering any data in controller memory. The enabled data recorders can collect up to a total of 32 addresses each sample. The valid range for the recordCount is 0 to 32.
<b>recordCount</b>	An array of the number of records for each data recorder buffer. Each data record is 32 bits. The controller's data recorder buffers can be dynamically allocated by changing the recordCount. A larger data recorder buffer may be required for higher sample rates, slow host computers, when running via client/server, or when a large number of data fields are being recorded. The valid range is 0 to the available memory. Use <a href="#">meiControlExtMemAvail(...)</a> to determine the controller's available external memory. meiControlExtMemAvail() measures the available memory in 8 bit bytes, so divide the size by 4 to get the number of 32 bit words that the record buffer can be increased by.
<b>sequenceCount</b>	Number of sequence objects enabled for the controller. The controller's sequence object executes and manages a sequence of pre-compiled controller commands.
<b>userVersion</b>	A 32 bit user defined field. The userVersion can be used to mark a firmware image with an identifier. This is useful if multiple controller firmware images are saved to a file.

**sampleRate**

Number of controller foreground update cycles per second. For SynqNet controllers, this is also the cyclic update rate for the SynqNet network. During the controller's foreground cycle, the axis trajectories are calculated, the filters (closed-loop servo control) are calculated, motion is coordinated, the SynqNet data buffers are updated, and other time critical operations are performed. The default sample rate is 2000 (period = 500 microseconds). The minimum sampleRate for SynqNet systems is 1000 (period = 1 millisecond). The maximum is dependent on the controller hardware and processing load.

There are several factors that must be considered to find an appropriate sampleRate for a system. The servo performance, the motion profile accuracy, the SynqNet network cyclic rate, the SynqNet drive update rates, controller background cycle update rate, and controller/application performance.

For SynqNet systems, select a sampleRate that is a common multiple of the SynqNet drives connected to the network. For example, if the drive update rate is 8kHz, then appropriate controller sample rates are: 16000, 8000, 5333, 4000, 3200, 2667, 2286, 2000, 1778, 1600, 1455, 1333, 1231, 1067, and 1000

**Definition: MEIControlConfig**

```
typedef struct MEIControlConfig {
    char                userLabel[MEIObjectLabelCharMAX+1];
                        /* +1 for NULL terminator */
    long                preFilterCount;
    long                TxTime;
    long                syncInterruptPeriod;
    MEIXmpPreFilter     PreFilter[MEIXmpMAX\_PreFilters];
    MEIXmpUserBuffer    UserBuffer;
} MEIControlConfig;
```

**Change History:** Modified in the 03.03.00. Modified in the 03.02.00

**Description**

<b>userLabel</b>	This value consists of 16 characters and is used to label the control object for user identification purposes. The userLabel field is NOT used by the controller.
<b>preFilterCount</b>	This value defines the number of enabled pre-filters.
<b>TxTime</b>	This value determines the controller's transmit time for the SynqNet data. The units are a percentage of the sample period. The default is 75%. Smaller TxTime values will reduce the latency between when the controller receives the data, calculates the outputs, and transmits the data. If the TxTime is too small, the data will be sent before the controller updates the buffer, which will cause a TX_FAILURE event.
<b>syncInterruptPeriod</b>	samples/interrupt. Configures the controller to send a hardware interrupt to host computer every $n$ controller samples.  0 = disabled, 1 = every sample, 2 = every other sample, etc...
<b>PreFilter</b>	This array defines the configuration for each pre-filter.
<b>UserBuffer</b>	This structure defines the controller's user buffer. This is used for custom features that require a controller data buffer.

## Sample Code

```

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

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

    external.UserBuffer.Data[index] = value;

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

```

```
        msgCHECK(returnValue);  
    }  
}
```

## See Also

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

# MPIControlFanStatusFlag

## Definition

```
typedef enum {
    MPIControlFanStatusFlagSTATUS_NOT_AVAILABLE, /* 0 */
    MPIControlFanStatusFlagFAN_OK,             /* 1 */
    MPIControlFanStatusFlagFAN_ERROR,          /* 2 */
    MPIControlFanStatusFlagOVER_TEMP_LIMIT,    /* 3 */
} MPIControlFanStatusFlag;
```

**Change History:** Added in the 03.02.00

## Description

**MPIControlFanStatusFlag** is an enumeration of fan status bit for use in the MPIControlFanStatusMask. The status bits represent the present status condition(s) for the fan controller on a given Control object.

<b>MPIControlFanStatusFlagSTATUS_NOT_AVAILABLE</b>	Value specifies that the fan status is not available for your controller.
<b>MPIControlFanStatusFlagFAN_OK</b>	Value specifies that the fan is fine.
<b>MPIControlFanStatusFlagFAN_ERROR</b>	Value specifies there is a fan error.
<b>MPIControlFanStatusFlagOVER_TEMP_LIMIT</b>	Value specifies there is an over temperature error.

## See Also

[MPIControl](#) | [MPIControlFanStatus](#) | [MPIControlFanStatusMask](#)

# MPIControlFanStatusMask

## Definition

```
typedef enum {
    MPIControlFanStatusMaskNONE = 0x0,
    MPIControlFanStatusMaskSTATUS_NOT_AVAILABLE =
        mpiControlFanStatusMaskBIT(MPIControlFanStatusFlagSTATUS_NOT_AVAILABLE),
        /* 0x00000001 */
    MPIControlFanStatusMaskFAN_OK =
        mpiControlFanStatusMaskBIT(MPIControlFanStatusFlagFAN_OK),
        /* 0x00000002 */
    MPIControlFanStatusMaskFAN_ERROR =
        mpiControlFanStatusMaskBIT(MPIControlFanStatusFlagFAN_ERROR),
        /* 0x00000004 */
    MPIControlFanStatusMaskOVER_TEMP_LIMIT =
        mpiControlFanStatusMaskBIT(MPIControlFanStatusFlagOVER_TEMP_LIMIT),
        /* 0x00000008 */
    MPIControlFanStatusMaskALL =
        mpiControlFanStatusMaskBIT(MPIControlFanStatusFlagLAST) - 1
        /* 0x0000000F */
} MPIControlFanStatusMask;
```

**Change History:** Added in the 03.02.00

## Description

**MPIControlFanStatusMask** is an enumeration of bit masks for the MPIControlFanStatusFlags. The status masks represent the present condition for a Control object.

<b>MPIControlFanStatusMaskNONE</b>	Bit mask containing none of the ControlStatusFlags set.
<b>MPIControlFanStatusMaskSTATUS_NOT_AVAILABLE</b>	Fan status is not available or supported by hardware.
<b>MPIControlFanStatusMaskFAN_OK</b>	Fan status is supported and there are no fan errors or temperature over limits.

**MPIControlFanStatusMaskFAN\_ERROR**

The fan or on-board fan controller has failed. This error indicates a serious problem with the fan or fan controller. This provides an early warning of a possible future over temperature error. If this error occurs, then the fan hardware should be examined and serviced by MEI. Please contact MEI for details.

The cause of a FAN\_ERROR is hardware dependent.

**For ZMP-Series using an ADM1030 fan controller, possible causes are:**

ALARM\_SPEED,  
FAN\_FAULT, and/or  
REMOTE\_DIODE\_ERROR Flags are set.

Please refer to the ADM1030 specifications for more information.

**MPIControlFanStatusMaskOVER\_TEMP\_LIMIT**

The temperature limit has been exceeded. This error indicates the controller processor is too hot. If the controller is operated at excessive temperatures, unknown behavior can result. MEI recommends that the application should be shutdown and the controller should be examined. Excessive temperature could be caused by insufficient air flow or by an improperly operating fan.

The cause of an OVER\_TEMP\_LIMIT is hardware dependent.

**For ZMP-Series using an ADM1030 fan controller, possible causes are:**

REMOTE\_TEMP\_HIGH,  
LOCAL\_TEMP\_HIGH, and/or  
OVER\_TEMP\_LIMIT Flags are set.

Please refer to the ADM1030 specifications for more information.

**MPIControlFanStatusMaskALL**

Bit mask containing all of the ControlStatusFlags set.

**See Also**

# MEIControlFPGA

## Definition

```
typedef struct MEIControlFPGA {  
    char FileName[MEIFlashFileMaxChars]  
} MEIControlFPGA;
```

**Change History:** Modified in the 03.02.00

## Description

**MEIControlFPGA** is a structure containing a **FileName** character array. The character array is used to define which FPGA file is to be loaded on the controller. This is usually used internally by the MPI.

<b>Filename</b>	character array
-----------------	-----------------

## See Also

[meiControlFPGADefaultGet](#) | [meiControlFPGAFileOverride](#)

# MEIControlInfo

## Definition

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

**Change History:** Modified in the 03.03.00

## Description

**MEIControlInfo** contains the information about the motion controller being used.

<b>mpi</b>	Information about the MPI software located on the host computer.
<b>firmware</b>	Information about the Firmware running on the controller.
<b>pld</b>	Information about the PLD located in the controller.
<b>rincon</b>	Information about the Rincon FPGA located on the controller.
<b>hardware</b>	Production information about the hardware stored in the controller.
<b>driver</b>	Information about the Driver, running on the host, used to interface with the controller.
<b>io</b>	Information about the I/O directly connected to the controller.

## See Also

# MEIControlInfoDriver

## Definition

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

## Description

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

<b>version</b>	The version of the Driver the host uses to interface with the controller.
----------------	---

## See Also

# MEIControlInfoFirmware

## Definition

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

**Change History:** Modified in the 03.02.00

## Description

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

<b>version</b>	The major version number for the controller's firmware. To be compatible with the MPI library, this number must match the fwVersion in the <a href="#">MEIControlInfoMpi</a> structure.
<b>option</b>	The firmware option number. Special or custom firmware is given a unique option number. An application or user can identify optional firmware from this value.
<b>revision</b>	The minor version number for the controller's firmware. Indicates a minor change or bug fix to the firmware code.
<b>subRevision</b>	The micro version value for the controller's firmware. Indicates a very minor change or bug fix to the firmware code.
<b>branchId</b>	Identifies an intermediate branch software revision. The branch value is represented as a hex number between 0x00000000 and 0xFFFFFFFF. Each digit represents an instance of a branch (0x1 to 0xF). A single digit represents a single branch from a specific version, two digits represent a branch of a branch, three digits represent a branch of a branch of a branch, etc.
<b>zmp</b>	ZMP-only information. Contains versions and revision info for boot0 and zboot code.

## See Also

# MEIControlInfoFirmwareZMP

## Definition

```
typedef struct MEIControlInfoFirmwareZMP {  
    long    boot0Version;  
    long    boot0Revision;  
    long    zbootVersion;  
    long    zbootRevision;  
} MEIControlInfoFirmwareZMP;
```

**Change History:** Added in the 03.02.00

## Description

**MEIControlInfoFirmwareZMP** is a structure containing version information about the boot0 and zboot code on a ZMP. Boot0 is the bootstrap code and should rarely need updating (updating is done at MEI). Zboot is the initialization code and will get updated every time the firmware is loaded.

**NOTE:** This information is displayed by the [Version](#) utility.

<b>boot0Version</b>	Version of boot0 code.
<b>boot0Revision</b>	Revision of boot0 code.
<b>zbootVersion</b>	Version of zboot code.
<b>zbootRevision</b>	Revision of zboot code.

## See Also

# MEIControlInfoHardware

## Definition

```
typedef struct MEIControlInfoHardware {  
    char    modelName[MEIControlSTRING\_MAX];  
    char    serialNumber[MEIControlSTRING\_MAX];  
    char    type[MEIControlSTRING\_MAX];  
} MEIControlInfoHardware;
```

## Description

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

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

## See Also

# MEIControlInfoIo

## Definition

```
typedef struct MEIControlInfoIo {  
    MEIControlInfoIoDigitalIn    digitalIn[MPIControlInMAX+1];  
    MEIControlInfoIoDigitalOut  digitalOut[MPIControlOutMAX+1];  
} MEIControlInfoIo;
```

**Change History:** Added in the 03.03.00

## Description

**MEIControlInfoIo** contains information about the I/O directly connected to the controller.

<b>digitalIn</b>	information about the digital inputs.
<b>digitalOut</b>	information about the digital outputs.

## See Also

[Controller I/O](#) | [meiControlInfo](#) | [MEIControlInfo](#)

# MEIControlInfoIoDigitalIn

## Definition

```
typedef struct MEIControlInfoIoDigitalIn {  
    MPI_BOOL    supported;  
    const char  *name;  
} MEIControlInfoIoDigitalIn;
```

**Change History:** Added in the 03.03.00

## Description

**MEIControlInfoIoDigitalIn** contains information about the digital inputs supported by this controller.

<b>supported</b>	a Boolean flag indicating whether or not the input is supported.
<b>*name</b>	a string that gives a name for this input.

## See Also

[Controller I/O](#) | [meiControlInfo](#) | [MEIControlInfoIo](#)

# MEIControlInfoIoDigitalOut

## Definition

```
typedef struct MEIControlInfoIoDigitalOut {  
    MPI_BOOL    supported;  
    const char  *name;  
} MEIControlInfoIoDigitalOut;
```

**Change History:** Added in the 03.03.00

## Description

**MEIControlInfoIoDigitalOut** contains information about the digital outputs supported by this controller.

<b>supported</b>	a Boolean flag indicating whether or not the output is supported.
<b>*name</b>	a string that gives a name for this output.

## See Also

[Controller I/O](#) | [meiControlInfo](#) | [MEIControlInfoIo](#)

[Controller I/O](#) | [mpiControlDigitalIn](#) | [mpiControlDigitalOutGet](#)

# MEIControlInfoMpi

## Definition

```
typedef struct MEIControlInfoMpi {
    char        version[MEIControlSTRING_MAX+1];
                /* +1 for null termination character */
    long        fwVersion;
    long        fwOption;
} MEIControlInfoMpi;
```

**Change History:** Modified in the 03.03.00

## Description

**MEIControlInfoMpi** is a structure that contains read-only version information for the MPI.

<b>version</b>	A string representing the version of the MPI. The version of the MPI is broken down by date, branch, and revision (MPIVersion.branch.revision). For ex: 20021220.1.2 means MPI version 20021220, branch 1, revision 2.
<b>fwVersion</b>	The firmware version information that the current version of the MPI will work with. A new field has been added to the XMP's firmware to identify and differentiate between intermediate branch software revisions. The branch value is represented as a hex number between 0x00000000 and 0xFFFFFFFF. Each digit represents an instance of a branch (0x1 to 0xF). A single digit represents a single branch from a specific version, two digits represent a branch of a branch, three digits represent a branch of a branch of a branch, etc.
<b>fwOption</b>	The firmware option number. Special or custom firmware is given a unique option number. An MPI library that requires optional firmware will have a value that must match the firmware's option number.

## See Also

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

# MEIControlInfoPld

## Definition

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

## Description

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

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

## See Also

[MEIControlInfo](#)

# MEIControlInfoRincon

## Definition

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

## Description

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

<b>version</b>	This is a 16-bit value in the hardware. The version string for the Rincon image on the controller.
<b>package</b>	<p>This is a 16-bit value in the hardware. The package string identification for the Rincon. The package string is a coded value that describes the Rincon image build type and target component.</p> <p>Existing types are:</p> <ul style="list-style-type: none"> <li><b>9201</b> - Rincon for XMP, XC2S100, PQ208 package</li> <li><b>9601</b> - Rincon for XMP, XC2S100, FG256 package</li> <li><b>A102</b> - RinconZ for ZMP, XC2S300E, FT256 package</li> <li><b>A301</b> - RinconZ for ZMP, XC3S200, FT256 package</li> </ul> <p>The package and version data can be used to create the FPGA filename. For example, 221_9201.fpg is Rincon type 9201, version 221.</p>

## See Also

[MEIControlInfo](#)

# MEIControlInput

## Definition

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

## Description

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

## See Also

[MEIControlOutput](#)

# MEIControlIoBit

## Definition

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

**Change History:** Modified in the 03.02.00

## Description

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

<b>MEIControlIoBitUSER_0_IN</b>	controller's local input, bit number 0
<b>MEIControlIoBitUSER_1_IN</b>	controller's local input, bit number 1
<b>MEIControlIoBitUSER_2_IN</b>	controller's local input, bit number 2
<b>MEIControlIoBitUSER_3_IN</b>	controller's local input, bit number 3
<b>MEIControlIoBitUSER_4_IN</b>	controller's local input, bit number 4
<b>MEIControlIoBitUSER_5_IN</b>	controller's local input, bit number 5
<b>MEIControlIoBitXESTOP</b>	controller's local input, External Emergency Stop Input.  <b>NOTE:</b> The XESTOP bit does not have any special functionality. The bit number and name were kept for backwards compatibility.
<b>MEIControlIoBitUSER_0_OUT</b>	controller's local output, bit number 0

<b>MEIControlIoBitUSER_1_OUT</b>	controller's local output, bit number 1
<b>MEIControlIoBitUSER_2_OUT</b>	controller's local output, bit number 2
<b>MEIControlIoBitUSER_3_OUT</b>	controller's local output, bit number 3
<b>MEIControlIoBitUSER_4_OUT</b>	controller's local output, bit number 4
<b>MEIControlIoBitUSER_5_OUT</b>	controller's local output, bit number 5

## See Also

[meiControlIoBitGet](#)

# MPIControlMemoryType

## Definition

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

## Description

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

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

## See Also

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

# MPIControlMessage / MEIControlMessage

## Definition: MPIControlMessage

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

## Description

**MPIControlMessage** is an enumeration of Motor error messages that can be returned by the MPI library.

### MPIControlMessageLIBRARY\_VERSION

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

### MPIControlMessageADDRESS\_INVALID

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

#### **MPIControlMessageCONTROL\_INVALID**

Currently not supported.

#### **MPIControlMessageCONTROL\_NUMBER\_INVALID**

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

#### **MPIControlMessageTYPE\_INVALID**

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

#### **MPIControlMessageINTERRUPTS\_DISABLED**

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

#### **MPIControlMessageEXTERNAL\_MEMORY\_OVERFLOW**

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

#### **MPIControlMessageADC\_COUNT\_INVALID**

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

#### **MPIControlMessageAXIS\_COUNT\_INVALID**

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

#### **MPIControlMessageAXIS\_FRAME\_COUNT\_INVALID**

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

#### **MPIControlMessageCAPTURE\_COUNT\_INVALID**

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

**MPIControlMessageCOMPARE\_COUNT\_INVALID**

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

**MPIControlMessageCMDDAC\_COUNT\_INVALID**

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

**MPIControlMessageAUXDAC\_COUNT\_INVALID**

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

**MPIControlMessageFILTER\_COUNT\_INVALID**

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

**MPIControlMessageMOTION\_COUNT\_INVALID**

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

**MPIControlMessageMOTOR\_COUNT\_INVALID**

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

**MPIControlMessageSAMPLE\_RATE\_TO\_LOW**

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

**MPIControlMessageRECORDER\_COUNT\_INVALID**

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

**MPIControlMessageCOMPENSATOR\_COUNT\_INVALID**

The compensator count is not valid. This message code is returned by [mpiControlConfigSet\(...\)](#) if the number of compensators is greater than MPIControlMAX\_COMPENSATORS.

**MPIControlMessageAXIS\_RUNNING**

Attempting to configure the control object while axes are running. It is recommended that all configuration of the control object occur prior to commanding motion.

**MPIControlMessageRECORDER\_RUNNING**

Attempting to configure the control object while a recorder is running. It is recommended that all configuration of the control object occur prior to operation of any recorder objects.

## Definition: MEIControlMessage

```
typedef enum {
    MEIControlMessageFIRMWARE_INVALID,
    MEIControlMessageFIRMWARE_VERSION_NONE,
    MEIControlMessageFIRMWARE_VERSION,
    MEIControlMessageFPGA_SOCKETS,
    MEIControlMessageBAD_FPGA_SOCKET_DATA,
    MEIControlMessageNO_FPGA_SOCKET,
    MEIControlMessageINVALID_BLOCK_COUNT,
    MEIControlMessageSYNQNET_OBJECTS,
    MEIControlMessageSYNQNET_STATE,
    MEIControlMessageIO_BIT_INVALID,
} MEIControlMessage;
```

## Description

**MEIControlMessage** is an enumeration of Control error messages that can be returned by the MPI library.

### MEIControlMessageFIRMWARE\_INVALID

The controller firmware is not valid. This message code is returned by [mpiControlInit\(...\)](#) if the MPI library does not recognize the controller signature. After power-up or reset, the controller loads the firmware from flash memory. When the firmware executes, it writes a signature value into external memory. If [mpiControlInit\(...\)](#) does not recognize the signature, then the firmware did not execute properly. To correct this problem, download firmware and verify the controller hardware is working properly.

### MEIControlMessageFIRMWARE\_VERSION\_NONE

The controller firmware version is zero. This message code is returned by control methods do not find a firmware version. This indicates the firmware did not execute at controller power-up or reset. To correct this problem, download firmware and verify the controller hardware is working properly.

### MEIControlMessageFIRMWARE\_VERSION

The controller firmware version does not match the software version. This message code is returned by control methods if the firmware version is not compatible with the MPI library. To correct this problem, either download compatible firmware or install a compatible MPI run-tim library.

### MEIControlMessageFPGA\_SOCKETS

The maximum number of FPGA socket types has been exceeded. This message code is returned by [meiFlashMemoryFromFile\(...\)](#) if the controller has more FPGA types than the controller has flash memory space to support them.

**MEIControlMessageBAD\_FPGA\_SOCKET\_DATA**

Currently not supported.

**MEIControlMessageNO\_FPGA\_SOCKET**

The FPGA socket type does not exist. This message code is returned by [meiFlashMemoryFromFile\(...\)](#) if the controller does not support the FPGA type that was specified in the FPGA image file. To correct this problem, use a different FPGA image that is compatible with the controller.

**MEIControlMessageINVALID\_BLOCK\_COUNT**

Currently not supported.

**MEIControlMessageSYNQNET\_OBJECTS**

Currently not supported.

**MEIControlMessageSYNQNET\_STATE**

The controller's SynqNet state is not expected. This message code is returned by [mpiControlInit\(...\)](#), [mpiControlReset\(...\)](#) and [mpiControlConfigSet\(...\)](#) if the SynqNet network initialization fails to reach the SYNQ state. To correct this problem, check your node hardware and network connections.

**MEIControlMessageIO\_BIT\_INVALID**

The controller I/O bit is not valid. This message code is returned by [meiControlIoGet\(...\)](#) and [meiControlIoSet\(...\)](#) if the controller I/O bit is not a member of the [MEIControlIoBit](#) enumeration.

**See Also**

# MEIControlOutput

## Definition

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

## Description

**MEIControlOutput** is an enumeration of a controller's local digital output bit masks. Each mask represents a discrete output.

## See Also

[MEIControlInput](#)

# MPIControlStatus

## Definition

```
typedef struct MPIControlStatus {
    MPIEventMask          eventMask;
    MPIControlFanStatusMask fanStatus;
} MPIControlStatus;
```

**Change History:** Added in the 03.02.00

## Description

**MPIControlStatus** is an MPI structure that is used to describe the current state of the controller's object. The XMP-Series controllers do not have fans and therefore do not support fanStatus. The ZMP-Series controllers have an optional processor cooling fan and support fanStatus.

<b>eventMask</b>	Array that defines the event mask bits. The controller event bits are defined in the MEIEventType enumeration.
<b>fanStatus</b>	Value is an enumeration of bit masks for the MPIControlFanStatusFlags. The status mask represents the present condition of the fan controller.

## See Also

[MPIControlFanStatusFlags](#) | [MPIControl](#) | [MPIControlStatus](#) |  
[MPIControlFanStatusMask](#) | [MEIEventType](#) | [meiEventMaskCONTROL](#)

# MEIControlTrace

## Definition

```
typedef enum {  
    MEIControlTraceDYNA_ALLOC = MEIControlTraceFIRST << 0,  
} MEIControlTrace;
```

## Description

**MEIControlTrace** is an enumeration of control object trace bits to enable debug tracing.

<b>MEIControlTraceDYNA_ALLOC</b>	This trace bit enables tracing for calls that dynamically allocate controller memory.
----------------------------------	---

## See Also

# MPIControlType

## Definition

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

## Description

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

## See Also

[MPIControl](#) | [mpiControlCreate](#) | [mpiControlType](#)

# MPIControlMAX\_AXES

## Definition

```
#define MPIControlMAX_AXES (32)
```

## Description

**MPIControlMAX\_AXES** defines the maximum number of axes available on one controller.

## See Also

[MPIAxis](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

# MPIControlMAX\_COMPENSATORS

## Definition

```
#define MPIControlMAX_COMPENSATORS (8)
```

## Description

**MPIControlMAX\_COMPENSATORS** defines the maximum number of compensator objects available on one controller.

## See Also

[MPICompensator](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

# MPIControlMAX\_RECORDERS

## Definition

```
#define MPIControlMAX_RECORDERS (32)
```

## Description

**MPIControlMAX\_RECORDERS** defines the maximum number of recorder objects available on one controller.

## See Also

# MEIControlSTRING\_MAX

## Definition

```
#define MEIControlSTRING_MAX (128)
```

**Change History:** Modified in the 03.03.00

## Description

**MEIControlSTRING\_MAX** defines the maximum number of characters in MEIControlInfo strings.

## See Also

[MEIControlInfo](#) | [MEIControlInfoHardware](#)

# MPIControlInMAX

## Definition

```
#define MPIControlInMAX 6
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.03.00

## Description

**MPIControlInMAX** defines the maximum number of digital inputs a controller could support. The `meiControlInfo(...)` function returns details about the actual inputs that the current controller supports.

## See Also

[MPIControlOutMAX](#) | [meiControlInfo](#)

# MPIControlOutMAX

## Definition

```
#define MPIControlOutMAX 5
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.03.00

## Description

**MPIControlOutMAX** defines the maximum number of digital outputs a controller could support. The `meiControlInfo(...)` function returns details about the actual outputs that the current controller supports.

## See Also

[MPIControlInMAX](#) | [meiControlInfo](#)

# MPIControlMIN\_AXIS\_FRAME\_COUNT

## Definition

```
#define MPIControlMIN_AXIS_FRAME_COUNT (128)
```

**Required Header:** `stdmpi.h`

## Description

**MPIControlMIN\_AXIS\_FRAME\_COUNT** defines the minimum allowed value for which `MPIControlConfig.axisFrameCount` can be set.

## See Also

[MPIControlConfig](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

# MPIControlMIN\_SAMPLE\_RATE

## Definition

```
#define MPIControlMIN_SAMPLE_RATE (1000)
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.03.00

## Description

**MPIControlMIN\_SAMPLE\_RATE** defines the minimum allowed value for which MPIControlConfig.sampleRate can be set.

## See Also

[MPIControlConfig](#) | [mpiControlConfigGet](#) | [mpiControlConfigSet](#)

# mpiControlFanStatusMaskBIT

## Declaration

```
#define mpiControlFanStatusMaskBIT(flag) (int)(0x1 << (flag))
```

**Required Header:** stdmpi.h

**Change History:** Added in the 03.02.00

## Description

**mpiControlFanStatusMaskBIT** is a utility macro to convert MPIControlFanStatusFlag defines to MPIControlFanStatusMask values.

An application should use the MPIControlFanStatusMask values instead of this macro.

## See Also

[MPIControlStatus](#) | [MPIControlFanStatusFlag](#) | [MPIControlFanStatusMask](#)

# TCP/IP and Sockets for Control Objects

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

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

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

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

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

[Return to Control Objects page](#)

# Dynamic Allocation of External Memory Buffers

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

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

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

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

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

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

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

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