Control Objects

Introduction

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

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

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

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

Methods

Create, Delete, Validate Methods

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

Configuration and Information Methods

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

Memory Methods

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

Action Methods

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

Relational Methods

meiControl**Platform**

Data Types

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

mpiControlCreate

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

Required Header stdmpi.h

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

The type parameter determines the form of the address parameter:

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

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

About MPIControlTypes:

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

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

Sample Code

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

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

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

A simple case where board #1 is desired is:

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

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

Return Values

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

See Also

MPIControl | MPIControlAddress | MPIControlType | mpiControlValidate
mpiControlInit | mpiControlDelete

mpiControlDelete

Declaration

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

Required Header

stdmpi.h

Description

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

Return Values

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

See Also

mpiControlCreate | mpiControlValidate

mpiControlValidate

Declaration `long mpiControlValidate(MPIControl control)`

Required Header stdmpi.h

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

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

See Also mpiControlCreate | mpiControlDelete

mpiControlAddress

Declaration

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

Required Header

stdmpi.h

Description

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

Return Values

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

See Also

mpiControlConfigGet

Declaration

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

Required Header

stdmpi.h

Description

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

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

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

Return Values

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

See Also

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

mpiControlConfigSet

Declaration

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

Required Header

stdmpi.h

Description

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

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

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

Return Values

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

See Also

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

meiControlExtMemAvail

Declaration

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

Required Header

stdmei.h

Description

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

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

Return Values

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

See Also

MPIControlConfig

mpiControlFlashConfigGet

Declaration

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

Required Header stdmpi.h

Description

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

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

XMP Only

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

Return Values

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

See Also MEIFlash | mpiControlFlashConfigSet | | MEIControlConfig

mpiControlFlashConfigSet

Declaration

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

Required Header

stdmpi.h

Description

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

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

XMP Only

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

Return Values

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

See Also

MEIFlash | mpiControlFlashConfigGet | | MEIControlConfig

meiControlGateGet

Declaration

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

Required Header

stdmei.h

Description

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

Return Values

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

See Also

meiControlGateSet

meiControlGateSet

Declaration

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

Required Header

stdmei.h

Description

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

Return Values

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

See Also

meiControlGateGet

meiControlSampleCounter

Declaration

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

Required Header stdmei.h

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

Return Values

MPIMessageOK	if the sample counter could be read

See Also meiControlSecondstoSamples | meiControlSamplestoSeconds | meiControlSampleWait

meiControlSamplesToSeconds

Declaration

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

Required Header stdmei.h

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

Return Values

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

See Also meiControlSecondstoSamples | meiControlSampleCounter

meiControlSecondstoSamples

Declaration

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

Required Header stdmei.h

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

Return Values

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

See Also meiControlSamplestoSeconds | meiControlSampleCounter | meiControlSampleWait

mpiControlType

Declaration

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

Required Header

stdmpi.h

Description

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

Return Values

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

See Also

meiControlVersionGet

Declaration

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

Required Header stdmei.h

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

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

See Also meiControlVersionSet

meiControlVersionSet

Declaration

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

Required Header

stdmei.h

Description

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

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

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

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

Return Values

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

See Also

meiControlVersionGet

mpiControlMemory

Declaration

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

Required Header

stdmpi.h

Description

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

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

Return Values

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

See Also

mpiControlMemoryGet | mpiControlMemorySet | mpiControlMemoryAlloc | mpiControlMemoryCount | mpiControlMemoryFree

mpiControlMemoryAlloc

Declaration

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

Required Header stdmpi.h

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

Return Values

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

See Also mpiControlMemoryGet | mpiControlMemorySet | mpiControlMemory | mpiControlMemoryCount | mpiControlMemoryFree

mpiControlMemoryCount

Declaration

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

Required Header control.h

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

Return Values

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

See Also

mpiControlMemoryFree

Declaration

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

Required Header stdmpi.h

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

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

See Also mpiControlMemoryGet | mpiControlMemorySet | mpiControlMemoryAlloc | mpiControlMemoryCount | mpiControlMemory

mpiControlMemoryGet

Declaration

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

Required Header

stdmpi.h

Description

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

Return Values

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

See Also

mpiControlMemorySet | mpiControlMemory | mpiControlMemoryAlloc | mpiControlMemoryCount | mpiControlMemoryFree

mpiControlMemorySet

Declaration

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

Required Header

stdmpi.h

Description

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

Return Values

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

See Also

mpiControlMemoryGet | mpiControlMemory | mpiControlMemoryAlloc | mpiControlMemoryCount | mpiControlMemoryFree

meiControlCycleWait

Declaration

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

Required Header

stdmei.h

Description

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

Return Values

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

See Also

mpiControlInit

Declaration `long mpiControlInit(MPIControl control)`

Required Header stdmpi.h

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

Return Values

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

See Also mpiControlDelete

mpiControlInterruptEnable

Declaration

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

Required Header

stdmpi.h

Description

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

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

Return Values

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

See Also

mpiControlInteruptWait | mpiControlInteruptWake

mpiControlInterruptWait

Declaration

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

Required Header

stdmpi.h

Description

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

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

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

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

Return Values

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

See Also

mpiControlInterruptWake | mpiControlInteruptEnable

mpiControlInterruptWake

Declaration `long mpiControlInterruptWake(MPIControl control)`

Required Header stdmpi.h

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

Return Values

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

See Also mpiControlInterruptWait | mpiControlInteruptEnable

mpiControlReset

Declaration

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

Required Header

stdmpi.h

Description

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

Return Values

MPIMessageOK	if *ControlReset* successfully resets the motion controller board

See Also

meiControlSampleWait

Declaration

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

Required Header stdmpi.h

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

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

See Also meiControlSamplestoSeconds | meiControlSecondstoSamples | meiControlSampleCounter |

meiControlPlatform

Declaration

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

Required Header

stdmei.h

Description

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

control	a handle to the Control object

Return Values

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

See Also

mpiControlCreate

MPIControlAddress

MPIControlAddress

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

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

Description

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

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

See Also

MPIControl | MPIControlType | mpiControlCreate

MPIControlConfig / MEIControlConfig

MPIControlConfig

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

} MPIControlConfig;
```

Description

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

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

Description

ControlConfig is the structure that contains the controller configuration information.

MEIControlConfig

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

Description

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

See Also

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

MPIControlIo

MPIControlIo

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

Description

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

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

See Also

MEIControlOutput | MEIControlInput | mpiControlIoGet | mpiControlIoSet

MEIControlInput

MEIControlInput

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

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

See Also MEIControlOutput | MPIControlIo | mpiControlIoGet | mpiControlIoSet

MPIControlMessage / MEIControlMessage

MPIControlMessage

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

Description

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

MEIControlMessage

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

Description

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

See Also

MEIControlOutput

MEIControlOutput

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

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

See Also MEIControlOutput | MPIControlIo | mpiControlIoGet | mpiControlIoSet

MPIControlType

MPIControlType

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

Description

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

See Also

MPIControl | mpiControlCreate | mpiControlType

MEIControlVersion

MEIControlVersion

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

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

    struct {
        long    version;        /* hardware version */


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

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

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

Description

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

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

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

See Also

MPIControl

Dynamic Allocation of External Memory Buffers

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

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

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

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

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

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

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

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

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

Return to Control Objects page

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