Methods Breakdown

--Table of Contents--

---Topic---	pg
Object Methods	2
Configuration Methods	3
Memory Methods	7
Status Methods	13
Event Notification Methods	17
List Methods	19
Identity Methods	27

Copyright © 2002
Motion Engineering

Object Methods

Introduction | Common Methods for MPI Objects
mpiObjectCreate(...) | mpiObjectDelete(...) | mpiObjectValidate(...)

Introduction

Object-oriented languages such as C++ and Java provide capabilities such as inheritance and polymorphism. The MPI is written in the C language, and so these more exotic object-oriented features are not available. However, the basic concepts of object-oriented design are in the MPI: modularity, data hiding and no global or static data.

It's similar to C++
If you are already familiar with C++, you will quickly realize that the MPI methods that create and delete objects are patterned after C++ constructors and destructors. An MPI method takes an object as its first argument, much as a C++ method has a "this" pointer. Object data is private and available only by calling object methods. MPI source code is organized into modules, with one object per module. Library state is distributed across application-created objects; there is virtually no state information maintained by the library.

All object methods and all functions return a value indicating whether the call to them succeeded or not. In general, the return value is of type long, with a value of 0 indicating success. The return value can be treated as an MPIMessage, so that you can call the mpiMessage(...) function to obtain a text string that describes the return value. The text string that indicates a successful return value of MPIMessageOK (0) is empty ("").

All MPI objects must implement certain required methods. Note that most MPI objects implement standard configuration and memory methods; many MPI objects implement standard status and list manipulation methods and a few objects implement standard event notification methods. For multi-threaded environments, all objects must implement standard resource allocation timeout methods.

In general, there are few methods that are unique to an object, and those unique methods tend to be those that perform object-specific actions. The following sections describe the standard methods that are implemented by many objects. For specific details about objects and their methods/functions, refer to the breakdown of Objects.

Where used in the rest of this section, the word *Object* may be replaced with the name of an MPI object in order to obtain the name of the methods for that object.

Example	For the term mpiObjectCreate(...), replace Object with Axis to obtain the name of the method that you would use to create Axis objects, mpiAxisCreate(...). For the term MPIObject, replace Object with Axis to obtain the data type of the Axis handle (MPIAxis) returned by the Axis method mpiAxisCreate(...), and also used by other Axis methods, such as mpiAxisValidate(...).

Common Methods for MPI Objects

All MPI objects have four types of common methods:

- methods which create an object
- methods which delete an object
- methods which validate an object
- methods which identify an object

mpi*Object*Create(...)

Use the Create method to create an object and return a handle that uniquely identifies that object. The MPI does not support declaring an object directly, so you must always use the Create method. The Create method is the only method that does not take an object handle as its first argument. The Create method arguments depend on the type of object being created.

If an object cannot be created, the value of the handle returned is MPIHandleVOID, which typically indicates only that no memory is available to hold the object. If memory is available to create the object, but the Create method encounters an error, then the Create method returns a handle, but the object referenced by the handle is not valid. A call to the mpiObjectValidate(...) method returns a value that indicates whether an object is valid, and if it is not valid, why. In general, if you call a method using an object handle that is invalid, the method will fail and return a value indicating the cause of failure.

The MPI supports multiple motion controllers. Typically, an MPI application first creates a Control object that corresponds to the desired motion controller. If the application will use more than one motion controller, it must create a Control object for each motion controller. The arguments to mpiControlCreate(...) associate a Control object with a specific motion controller.

A motion controller provides various types of resources, such as axis, digital-to-analog converter, data recorder, etc. Many of a motion controller's resource types have multiple instances. For example, motion controllers provide multiple axes and DACs.

Most MPI objects are associated with a specific instance of a resource on a specific motion controller. An Axis object, for example, is associated with a single axis on a motion controller. When creating an MPI object that corresponds to a specific motion controller resource, the Create method takes an MPIControl handle as its first argument and a number as the second argument. The number argument identifies the specific motion controller resource to be used.

Example	The number argument of mpiAxisCreate(...) identifies an axis on the motion controller specified by the MPIControl handle argument.

For XMP only	Number arguments, for those MPI objects that take them, start with 0 (not 1). For example, on an 8-axis XMP motion controller, valid axis numbers are 0 - 7.

Some MPI objects maintain a list whose elements are other MPI objects. The Create method for a list object generally takes a third argument that is a handle to an element object. The element object then becomes the first element of the list maintained by the list object. Specifying an element object is optional; if the handle (third argument) has value MPIHandleVOID, the list object will be created with an empty list.

Example	The Motion object maintains an ordered list of Axis objects. This ordered list of Axis objects defines a motion coordinate system. The third argument to mpiMotionCreate(...) is an Axis handle (MPIAxis). If the Axis handle is valid, then the Axis object (that the handle points to) will be made the first axis in the motion coordinate system.

All MPI objects provide a method to return the value of each of its Create method arguments. These methods serve to identify the object. For more detail, see Identity Methods on page 1-47.

mpi*Object*Delete(...)

Use the Delete method to delete an object created by the Create method. After an object has been deleted, the handle to that object can no longer be used.

If a method attempts to use a handle to a deleted object, it returns MPIMessageObject_FREED, but not always. If the memory originally allocated for the deleted object is subsequently allocated for a different purpose, then the method will not return MPIMessageObject_FREED.

Your application must ensure that handles to deleted objects are no longer used. After an object is deleted, your application should ensure that the handle to that object is set to MPIHandleVOID. After that, any calls to methods using this handle will return MPIMessageHANDLE_INVALID.

When your application exits, it should call Delete methods for all objects that have been created by your application. We recommend that your application delete objects in the reverse order in which they were created. Regardless, the Control object should be the last object deleted. The MPI does not provide the ability to automatically delete all of the objects that have been created.

Alternatively, to delete objects, your application may use the C library function atexit(...), which guarantees that the objects will be deleted even if the application terminates abnormally.

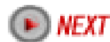
mpi*Object*Validate(...)

Use the Validate method to “validate” an object created by the Create method. Your application should always call the Validate method immediately after calling the Create method. Afterwards, the Validate method can additionally be called at any time in an application.

If the object is valid, the Validate method will return MPIMessageOK. If the handle passed to the Validate method has value MPIHandleVOID, the Validate method returns MPIMessageHANDLE_INVALID. If a method attempts to use a handle to a deleted object, the method returns MPIMessageObject_FREED. (Also see the previous discussion in mpiObjectDelete(...)).

Other possible return values are declared in the message header file(MPI\include\message.h) and in the object’s header file (MPI\include\object.h).

Object Methods | Configuration Methods | Memory Methods | Status Methods | Event Notification Methods | List Methods | Identity Methods |



Copyright © 2002
Motion Engineering

Configuration Methods

Introduction | mpiObjectConfig{} Structure | mpiObjectConfigGet(...) / mpiObjectConfigSet(...)
mpiObjectFlashConfigGet(...) / mpiObjectFlashConfigSet(...)

Introduction

An object's configuration is static information that may be read and written, and an object retains its configuration until an application changes it. To configure an MPI object, you use a Config structure that is specific to each object.

To configure an MPI object, get the current Config from the motion controller by calling the ConfigGet method. Modify the Config structure as desired, and then send it back to the motion controller by calling the ConfigSet method.

If a motion controller has flash memory, you can configure that flash memory using the same Config structure. The MPI implementation handles all of the details for flash memory support; all your application needs is a handle to a flash memory object. If a motion controller does not have flash memory, the flash configuration methods will return MPIMessageUNSUPPORTED.

MPI*Object*Config{} Structure

An object's configuration structure contains all of the configurable items for that object. To configure an object, your application must declare a variable of type MPIObjectConfig{}. Note that some Config structures are very large, so if you declare a Config structure on a small stack, your application might encounter problems. A pointer to a Config structure is passed to the Config methods.

The MPI Config structures should be able to support generic object configurations for a variety of motion controllers. However, the MPI also provides you with a way to perform configuration unique to a particular motion controller. In addition to the MPIObjectConfig{} structure, MPI Config methods can also use an external configuration structure that is defined by the implementation.

For XMP only	The external configuration structures are of type MEIObjectConfig{}.

Example	The MEIAxisConfig{} structure contains XMP-specific configuration parameters that extend the configuration available in MPIAxisConfig{}.

Configuration Methods

Root	**Method**	
ConfigGet	mpiObjectConfigGet(...)	Fill configuration structures with current motion controller configuration
ConfigSet	mpiObjectConfigSet(...)	Change current motion controller configuration

FlashConfigGet	mpiObjectFlashConfigGet(...)	Fill configuration structures with current motion controller Flash configuration
FlashConfigSet	mpiObjectFlashConfigSet(...)	Change current motion controller Flash configuration

mpiObjectConfigGet(...) / mpiObjectConfigSet(...)

The mpiObjectConfig( ) structure is meant to illustrate how objects are configured. Please swap the object you are interested in for the the object to configure. For example, mpiObjectConfigSet( ) shows the naming convention that mpiAxisConfigGet( ) follows.Both ConfigGet and ConfigSet take three arguments:

- the first argument is the object handle (MPIObject)
- the second argument is a pointer to an MPIObjectConfig{ } structure (MPIObjectConfig *)
- the third argument is a pointer to an external implementation-defined configuration structure (void *).The third argument may also be NULL

For XMP only	The external argument is a pointer to an XMP-specific configuration structure of type MEIObjectConfig{ }.

Use the Get method to fill the configuration structures with the current motion controller configuration for the object.
Use the Set method to change the current motion controller configuration for the object. (The Set method uses the configuration structures to do that.)

mpiObjectFlashConfigGet(...) / mpiObjectFlashConfigSet(...)

The FlashConfigGet and FlashConfigSet methods take four arguments:

- The first argument is the object handle (MPIObject)
- The second argument is an implementation-specific handle to Flash memory (void *).
- The second argument can also be NULL.
- The third argument is a pointer to an MPIObjectConfig{ } structure (MPIObjectConfig *)
- The fourth argument is a pointer to an external implementation-defined configuration structure (void *). The fourth argument can also be NULL.

For XMP only	The flash handle is of type MEIFlash (second argument); the external argument (fourth argument) is a pointer to a configuration structure of type MEIObjectConfig{ }.

Use the Get method to fill the configuration structures with the current motion controller Flash configuration for the object.
Use the Set method to change the current motion controller Flash configuration for the object. (The Set method uses the configuration structures to do that.)

The flash configuration is in effect after system start-up and also after resetting the motion controller (by calling the mpiControlReset(...) method).

[Object Methods](#) | **Configuration Methods** | [Memory Methods](#) | [Status Methods](#)
| [Event Notification Methods](#) | [List Methods](#) | [Identity Methods](#) |

Copyright © 2002
Motion Engineering

Memory Methods

Introduction | Accessing Memory Methods | Control Object Memory Methods
mpiObjectMemory(...) | mpiObjectMemoryGet(...) / Set(...)

Introduction

You use Memory methods to access a motion controller's RAM. The MPI imposes no structure on motion controller RAM, so the use of memory methods requires implementation-specific knowledge of the motion controller's memory map.

For MEI/XMP only	The memory map is defined in the xmp.h header file. All accessible memory is defined b#eeeeFFy the MEIXmpData{ } and MEIXmpBufferData{ }structures.

Each object has its own memory methods; only the memory directly associated with the specific object can be accessed using that object's Memory methods.

Example	An Axis object created with axis number 0 may only access memory associated with Axis 0.

Accessing Memory Methods

Memory access via these methods is thread-safe; only one thread at a time can read or write the portion of memory associated with the object. The Get/Set methods will bounds-check the memory address to be accessed, and return an error if the address is not associated with the object.

Control Object Memory Methods

The Control object memory methods are an exception. Using the ControlMemory methods, an application may access all memory on the motion controller at any time, without constraint. The ControlMemory methods are **not** thread-safe. Memory methods for the other objects are generally implemented by locking the section of memory associated with the object, and then calling a ControlMemory method.

It is possible to write a motion application using only a Control object. After creating and initializing the Control object, the address of motion controller memory can be obtained, and you can use the ControlMemoryGet/Set methods to access that memory.

Depending on the type of Control object created, an application can directly access the motion controller memory without using the ControlMemoryGet/Set methods. Such an application would bypass the rest of the MPI library and must implement its own thread safety, as well as deal with how the motion controller firmware operates.

Method	
mpi*Object*Memory(...) mpi*Object*MemoryGet(...) mpi*Object*MemorySet(...)	Return host address of object in memory Read motion controller memory Write motion controller memory

mpiObjectMemory(...)

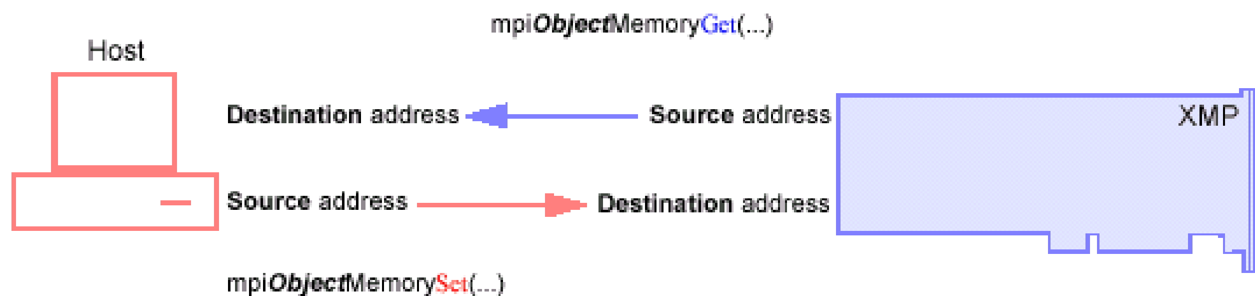
The mpiObjectMemory(...) method returns a host address that maps to the section of motion controller memory associated with the object. mpiObjectMemory(...) uses 2 arguments: the **object handle**, and an **output argument** of type void **. If a call to mpiObjectMemory(...) succeeds, the location pointed to by the output argument is set to the host address.

mpiObjectMemoryGet(...) / mpiObjectMemorySet(...)

These methods read (get) and write (set) motion controller memory. They take 4 arguments:

- the object handle
- a destination address of type void *
- a source address of type void *
- and a length in bytes

The Get method's destination address points to host memory, while the source address points to motion controller memory (based on the address returned by the Memory method for the object). The Set method is the opposite. The Set method's destination address points to motion controller memory and the source address points to host memory.



Copyright © 2002
Motion Engineering

Status Methods

Introduction | MPIStatus{...} Structure | MPI*Object*Status{...} Structure

Introduction

Status is read-only information that changes dynamically. Many objects define a Status structure and a method that uses that structure to return status to the application.

MPIStatus{...} Structure

The MPIStatus{...} structure contains information about the state of an axis or an entire motion. The Axis object and the Motion object return status using the MPIStatus{...} structure. Because the MPIStatus{...} structure is shared among objects, it is defined in the mpidef.h header file instead of being defined in an object header file.

For objects that involve more than 1 axis, the status returned by the motion object indicates the "sum" of the status conditions of the axes in its coordinate system. In general, the status structure gives an indication of whether an axis is moving or idle. The axis structure also indicates whether the axis is idle because of an error.

For MEI/XMP only	The filter object (whose handle is of type MEIFilter) also returns status in MPIStatus{...}.

MPI*Object*Status{...} Structure

Objects that are not Axis and Motion objects return status using an object-specific Status structure defined in the object's header file.

mpi*Object*Status(...)
For Axis and Motion objects, mpiObjectStatus(...) returns the object's status using the object's handle and a pointer to an MPIStatus{...} structure. For objects that are not Axis or Motion objects, mpiObjectStatus(...) returns the object's status using the object's handle and a pointer to an MPIObjectStatus{...} structure.

Object Methods | Configuration Methods | Memory Methods | **Status Methods**
| Event Notification Methods | List Methods | Identity Methods |

Copyright © 2002
Motion Engineering

Event Notification Methods

Introduction | mpiObjectEventNotifyGet(...) | mpiObjectEventNotifySet(...)
mpiObjectEventNotifyReset(...) | Notification Process

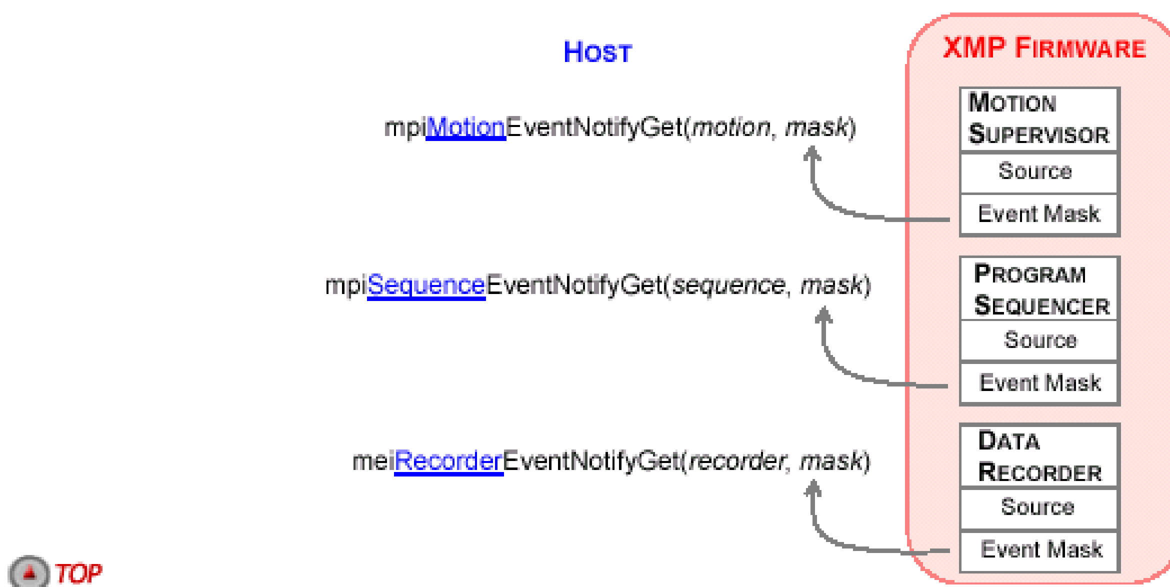
Introduction

The MPI EventMgr is responsible for the collection and distribution of host notifications of firmware events. An application often needs to be notified of events that take place on the motion controller. Events include normal motion completion, motion limits being reached, hardware failure, etc. The EventNotify methods enable your application to request host notification of certain types of events, while ignoring other types of events. Some events are latched, in which case your application must reset the event before the event can be triggered again.

Method	Description
mpiObjectEventNotifyGet(...)	Get event mask used for host notification
mpiObjectEventNotifySet(...)	Set event mask to request host notification of events
mpiObjectEventNotifyReset(...)	Reset event notification

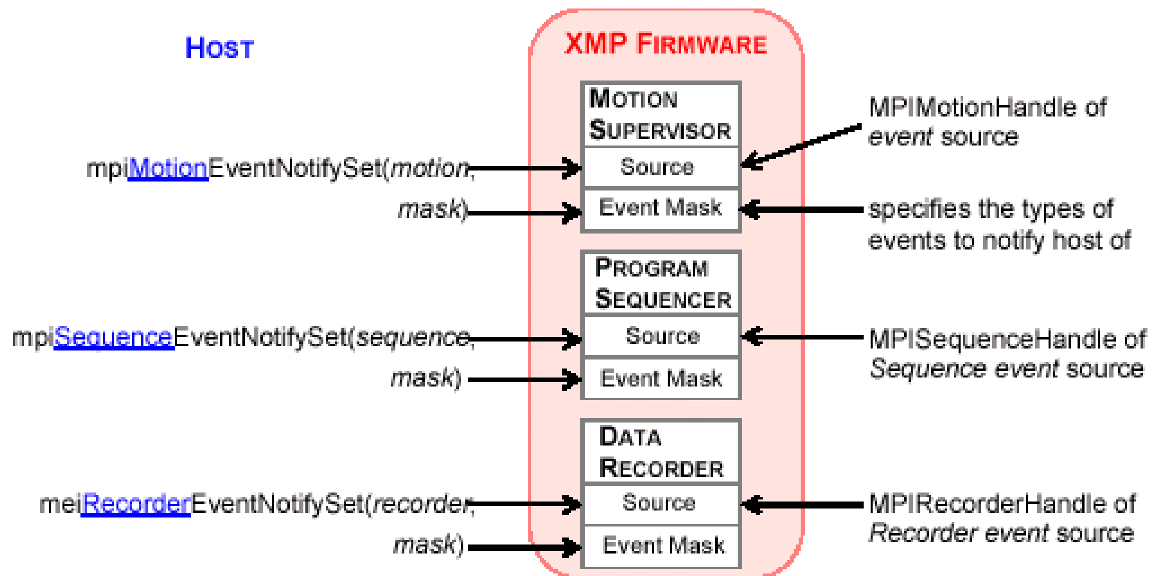
mpiObjectEventNotifyGet(...)

Use EventNotifyGet to return an MPIEventMask, which has a bit set for each type of event that host notification has been requested for, by the object.



mpiObjectEventNotifySet(...)

Use EventNotifySet to request host notification for each type of event specified in the MPIEventMask argument.

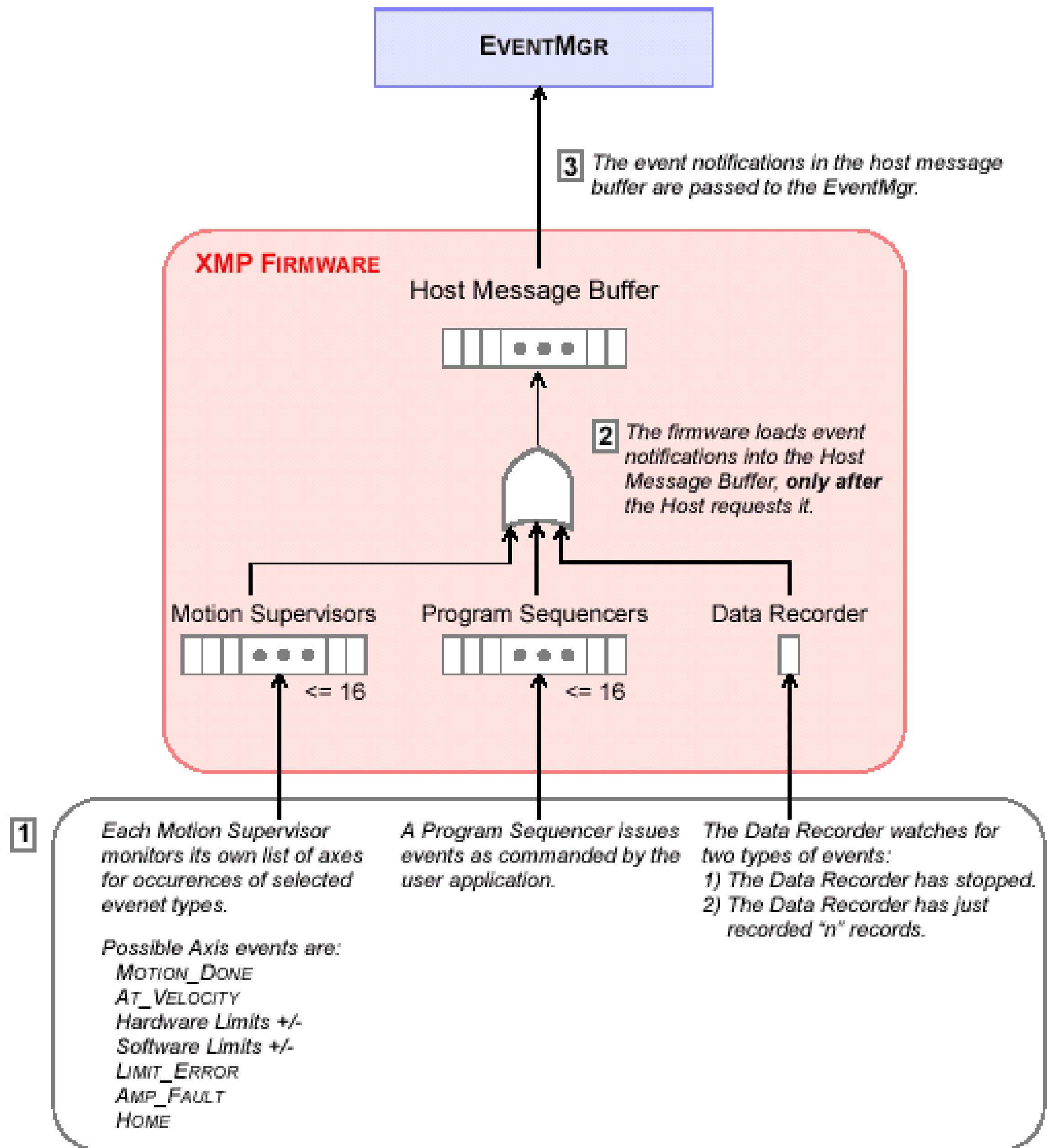


mpiObjectEventNotifyReset(...)

Use EventNotifyReset to reset each type of event specified in the MPIEventMask argument.

Notification Process

Events [that are requested by mpiObjectEventNotifySet(...)] move up through the firmware to the EventMgr.



To collect events, the EventMgr either polls the firmware (using mpiEventMgrService(...)) or is interrupted by the firmware. The EventMgr passes each event to its list of Notify objects, who further qualify events based on the event type and the source of the event, and generate signals and otherwise notify the host application of event occurrences.

EVENTMGR

EVENT
Type
Source

2

The EventMgr passes the event types and object handles to a list of Notify objects.

3

0 • • • n

NOTIFY
Mask
Signal

Notify objects watch for events and sources of events.

NOTIFY
Mask
Signal

A Notify object keeps two types of lists: events and sources (of events).

0 • • • n

EVENT EVENT

FIFO list

0 • • • n

SOURCE SOURCE

A source can be one of three possible objects:
Motion
Sequence
Recorder

4 If there is an Event and Source match (as selected by Mask), then the Notify object issues the Signal. Note that any source matches an empty source list.

mpiEventMgrService( )

1 The event types and object handles in the host message buffer are passed to the EventMgr.

XMP FIRMWARE

Object Methods | Configuration Methods | Memory Methods | Status Methods
| **Event Notification Methods** | List Methods | Identity Methods |

Copyright © 2002
Motion Engineering

List Methods

Introduction | List | Element | Array Methods
List Expansion Methods | List Contraction Methods | List Traversal Methods

Introduction

Several MPI objects maintain a list whose elements are other MPI objects.

Example	A Motion object maintains a list of Axis objects. An EventMgr object maintains 2 lists: a list of Control objects and a list of Notify objects.

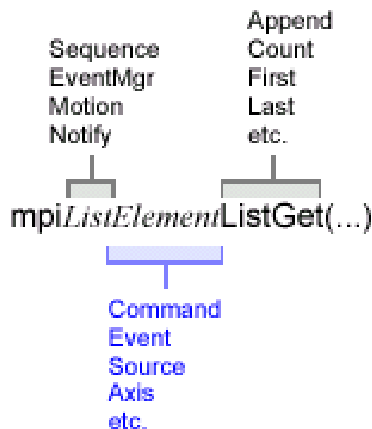
The MPI declares standard list manipulation methods for all such objects. The MPI does not specify how lists are to be implemented.

List

A list is an ordered sequence of elements. Generally, the elements of a list are objects of the same type, but this is not required. A list that contains no elements is empty. A list may be traversed either forwards or backwards from any element that is a member of the list. List manipulation methods are declared for the object that maintains the list.

Element

An element is a member of a list. An element is generally an object. An element is added to or removed from a list using list manipulation methods (declared by the object that maintains the list).



Grouping	List Action	Method	
Array Methods	ListGet ListSet		
List Expansion	Append Insert		
List Contraction	Remove		
List Traversal	 Count First Index Last Next Previous	mpiListElement(...) mpiListElementCount(...) mpiListElementFirst(...) mpiListElementIndex(...) mpiListElementLast(...) mpiListElementNext(...) mpiListElementPrevious(...)	Return handle of indexth element Return number of elements in list Return handle to first element in list Return index number of element in list Return handle to last element in list Return handle to next element in list Return handle to previous element in list

Array Methods

You use Array methods to manipulate lists. Array methods use an element count and an application-resident array of elements. These methods provide a convenient way of dealing with a list as a whole. Note that if you do use array methods for list manipulation, you can also use other list manipulation methods as well.

mpiListElementListGet(MPIList list, long *count, MPIElement *Element)
The ListGet method returns the list (maintained by List) as an array of object handles of type MPIElement. Upon successful return of the ListGet method, the contents of the location pointed to by count will be set to the number of elements in the list, or set to zero (if the list is empty). Your application must also define the object handle array pointed to by Element, and the array must be large enough to hold the current number of list elements.

mpiListElementListSet(MPIList list, long count, MPIElement *Element)
The ListSet method sets the list (maintained by List) from an array of object handles of type MPIElement. Upon successful return of the ListSet method, the list will contain count elements. Your application must also define the object handle array pointed to by Element, and the array must contain at least count list elements. After using the ListSet method, any existing list will be completely replaced by the new list. To make an existing list empty, call the ListSet method with a count of 0 and Element set to NULL.

List Expansion Methods

A list can be expanded by inserting an element at any point, either as the first element of the list, after a specific list element, or as the last element of the list. Lists generally check and verify that an element to be inserted is not already a list element. Other list-specific checks may be made before insertion as well.

mpiListElementAppend(MPIList list, MEIElement Element)
The Append method appends an object (whose handle is Element) to the list maintained by List, making it the last element of the list.

mpiListElementInsert(MPIList list, MEIElement before, MEIElement insert)
The Insert method inserts an object (whose handle is insert) into the list maintained by List, placing it after the list element whose handle is before. If before is MPIHandleVOID, insert becomes the first element of the list.

List Contraction Methods

A list can be contracted (shortened) by removing a list element. Lists check and verify that the element to be removed is actually a list element.

mpiListElementRemove(MPIList list, MEIElement Element)
The Remove method removes an object (whose handle is Element) from the list maintained by List.

List Traversal Methods

To traverse a list means to be able to move through all of the elements on a list. It can also involve determining the number of list elements, searching for a particular list element, finding the index of a list element, and more.

MPIElement mpiListElement(MPIList list, long index)

Return Values	
handle	of the indexth list element of the list maintained by List
MPIHandleVOID	if index is less than zero if index is greater-than-or-equal-to the number of list elements

mpiListElementCount(MPIList list)

Return Values	
number of list elements	on the list maintained by List
-1	if list is invalid

MPIElement mpiListElementFirst(MPIList list)

Return Values	
handle	to the first list element of the list maintained by List
MPIHandleVOID	if list is empty or is not valid if list is not valid

mpiListElementIndex(MPIList list, MPIElement Element)

Return Values	
index	of Element in the list maintained by List
-1	if list is empty or not valid if Element is not a list element of the list maintained by List

MPIElement mpiListElementLast(MPIList list)

Return Values	
handle	to the last list element of the list maintained by List
MPIHandleVOID	if list is empty if list is not valid

MPIElement mpiListElementNext(MPIList list, MPIElement Element)

Return Values	
handle	to the list element immediately after Element in the list maintained by List
MPIHandleVOID	if list is not valid if Element is not a list element if Element is the last list element of the list maintained by List

MPIElement mpiListElementPrevious(MPIList list, MPIElement Element)

Return Values	
handle	to the list element immediately before Element in the list maintained by list
MPIHandleVOID	if list is not valid if Element is not a list element if Element is the first list element of the list maintained by List

Object Methods | Configuration Methods | Memory Methods | Status Methods | Event Notification Methods | **List Methods** | Identity Methods |

Copyright © 2002
Motion Engineering

Identity Methods

Identity methods are used to identify an object, by providing the value of each of the arguments passed to the Create method to create that object. An identity method exists for each Create method argument, and typically is named by that argument.

Identity methods generally have a second output argument that is a pointer to the Create method argument to be returned; the return value of the method will indicate success or failure (as usual). However, identity methods that return an object handle do so directly rather than using an output argument.

Example	The mpiAxisCreate(...) method takes 2 arguments; an MPIControl handle and an axis number. Therefore, the Axis module declares 2 identity methods: mpiAxisControl(...) and mpiAxisNumber(...). The mpiAxisControl(...) method returns the MPIControl handle with which the axis object was created. The mpiAxisNumber(...) method returns a value that indicates success or failure; if the value indicates success, the second argument points to a location that will be set to the axis number.

Object Methods | Configuration Methods | Memory Methods | Status Methods | Event Notification Methods | List Methods | **Identity Methods** |

Return to Software: Method's Main Menu

Copyright © 2002
Motion Engineering