MPI LIBRARY BASICS

--Table of Contents--

Introduction to the MPI Library

The MPI is a C language object-oriented interface that makes it easy for you to develop motion control applications that run on many different types of platforms. Programs written using the MPI range from simple single-task, single-controller applications to complex multi-tasking applications using multiple motion controllers. The MPI hides platform-specific and firmware implementation details while providing a rich set of functions to control motion at any desired level. As a system designer, you have complete flexibility to choose the level of control best suited for your application. The MPI library executes on the host computer, while the firmware executes on the XMP controller.

The XMP's firmware components are managed by host software using the MPI (Motion Programming Interface). The MPI provides direct memory access to all XMP firmware components, delivering much more performance than competing systems that use command-based controllers or ASCII interfaces. Designed for multitasking environments, the MPI can access one or more motion controllers at maximum bus bandwidth, while efficiently handling controller interrupts.

Debugging under the MPI environment is simple if you use the configurable trace and debug features. The trace feature enables you to follow the progress of an executing program by observing the stream of messages produced whenever a library function is entered (displaying calling parameters) and whenever a library function is exited (displaying the return value).

You also have the ability to validate all library function parameters, and to stop execution whenever an error occurs, displaying the source file name and line number where the error occurred. You can configure the MPI for optimal trace and debug support during the design and test phase, and then reconfigure the MPI for optimal performance (without the debugging support).

Introduction | Motion Concepts | General Definitions | Naming Conventions | Object Descriptions

Motion Concepts

Introduction | Maps & Links | Motor Links | Filter Links
Axis Links | *see* Object Relationships

Introduction

The XMP software architecture is composed of separate modular objects that can be associated in a variety of ways to create custom systems. For example, a gantry or robot using **more** than one motor for a physical axis may be configured to allow the application to treat the axis as a single entity, simplifying the geometry. This flexibility does not come without a price, however, since configuration of complex systems involves a fairly large number of parameters. To overcome the need to configure simple "standard" systems, the XMP is pre-configured with a factory default configuration which can be used for most simple systems.

To understand how to create a custom XMP configuration, you must understand the component objects involved. Custom configurations are created using four types of objects: Motors, Filters, Axes, and Motion Supervisors (called Motion objects). To simplify implementation of the XMP architecture, we assumed that the association of Motors, Filters, and Axis would be more or less static (created during initialization), while association of Axes with Motion Supervisors would be more dynamic (created or modified during runtime).

TOP

Maps & Links

The relationships between the controller objects are determined by Maps and Links associated with the objects. The Maps are used by the MPI to define the object relationships and usually have the form of bitmaps. The following MPI functions are used to read and write these Maps.

mpiMotorAxisMapGet(...)	
mpiMotorFilterMapGet(...)	mpiMotorFilterMapSet(...)
mpiFilterAxisMapGet(...)	mpiFilterAxisMapSet(...)
mpiFilterMotorMapGet(...)	mpiFilterMotorMapSet(...)
mpiAxisMotorMapGet(...)	
mpiAxisFilterMapGet(...)	mpiAxisFilterMapSet(...)

Note that there are no functions to write any Motor-Axis or Axis-Motor maps. This is because these relationships are defined by the maps of the associated filters.

The links are used by the controller's objects to determine the sources for data generated by other objects. These links generally have the form of pointers that reference the source of the data or indexes that identify the sourcing object.

TOP

Motor Links

There can be up to 2 filter outputs used in calculating the output level of the Motor command. The following expression is evaluated each sample to determine this output:

Used to determine the combination of filter outputs that will be used to control the torque or velocity of a motor.

MEIMotorConfig{}. FilterInput[]

The output level of Motor is calculated on each sample.

Output =
motorConfig.FilterInput[0].InputCoeff * (Filter[motorConfig.FilterInput[0].FilterNumber]. Control Output))
+ motorConfig.FilterInput[1].InputCoeff * (Filter[motorConfig.FilterInput[1]. FilterNumber].ControlOutput));

Used to determine the position value for commutation

MEIMotorConfig{}. Commutation.OpenLoopPointer

MEIMotorConfig{}. Commutation.ClosedLoopPointer

TOP

Filter Links

The Filter object uses the difference between command and actual positions to calculate a control output (control algorithm). Both the command and actual positions used in these calculations can be defined as a linear combination of positions from 2 different sources.

MEIFilterConfig{}.Axis[] determine the sources and weight factors for these position inputs. On each sample, the following equations are used to determine the position error and command position used by the Filter:

These two equations determine the actual and command positions used by the Filter.

Position_Error = Axis[0].Ptr -> PositionError * Axis[0].Coeff +
Axis[1].Ptr -> PositionError * Axis[1].Coeff;

Command_Pos = Axis[0].Ptr -> CommandPosition * Axis[0].Coeff +
Axis[1].Ptr -> CommandPosition * Axis[1].Coeff;

The command velocity and acceleration used in feedforward calculations is also determined from MEIFilterConfig{}.AxisInput[], using the following equations:

Used in feedforward calculations.

Command_vel = Axis[0].Ptr -> CommandVelocity * Axis[0].Coeff;
Axis[1].Ptr -> CommandVelocity * Axis[1].Coeff;

Command_acc = Command_vel - Old_Command_vel;

Old_command_vel = Command_vel;

TOP

Axis Links

The actual position of an axis can be calculated as a linear combination of position values from two separate sources. On each sample, the following equation is used to compute the Axis' actual position:

MEIAxisConfig{}. APos[]

Used to compute the actual position of an Axis.

```
Actual_pos = (*axisConfig.Apos[0].Ptr) * axisConfig.Apos[0].Coeff
           + (*axisConfig.Apos[1].Ptr) * axisConfig.Apos[1].Coeff;
```

TOP

Introduction | Motion Concepts | General Definitions | Naming Conventions | Object Descriptions

NEXT

General Definitions

MPI | Platform | MPI Installation Directory | Object | Handle | Method | Module

MPI

The **M**otion **P**rogramming **I**nterface (**MPI**) defines a set of object-oriented, C-language functions and data types that you can use to develop motion control applications. The MPI is designed to be independent of the motion controller hardware used, as well as the *platform* (operating system & compiler) on which the motion control application is built. To support the MPI, a platform must be 32-bit (e.g. Windows NT). Most 32-bit platforms are capable of multi-threading and multi-tasking, and the MPI has built-in features that simplify the design and development of these types of applications.

For XMP only	Instructions for the implementation of MPI in Motion Engineering's XMP motion controller are flagged with the notice "For XMP only."

To add custom features, applications often make use of extensions to the MPI, which limits portability.

Platform

A platform is the combination of the host computer, operating system, and C compiler. A platform must be 32-bit, i.e., the operating system must be able to execute 32-bit programs.

For XMP only	The MPI/XMP has been ported to the WindowsNT/Microsoft Visual C++ platform. This platform is referred to as XMP-WinNT. If you are interested in using the MPI/XMP with other operating systems and C compilers, please contact MEI.

TOP

MPI Installation Directory

The MPI installation directory is the location into which the MPI software has been installed. The installation procedure suggests a default location (C:\MEI), but you can choose to install it wherever you desire. It is highly recommended that you use the default location. The installation directory contains an MPI directory:

MPI Subdirectories

Directory \ MPI	Contains
\doc	Documentation – User and Reference Guides
\include	MPI header files

For XMP-WinNT only	The default installation directory is C:\MEI, and it contains a release note named YYYYMMDD.pdf, where YYYY represents the year, MM the month, and DD the day of the release. The installation directory also contains an XMP directory, which has various subdirectories.

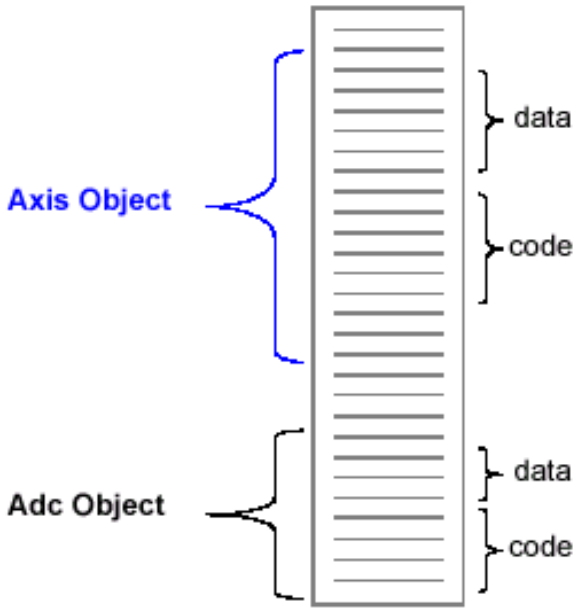
XMP Subdirectories

Directory: C:\MEI\XMP	**Contains**
\app	Sample applications
\bin	XMP firmware
\bin\WinNT	Windows NT utilities (from util)
\doc	Documentation - XMP Harware and Software Reference
\include	XMP header files
\lib	XMP libraries
\util	XMP utility source

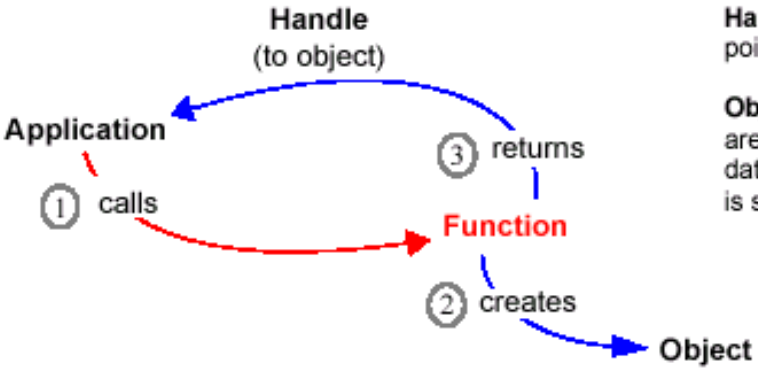
TOP

Object

An object is an instance of a data structure. The data structure is typically not directly accessible by an application.



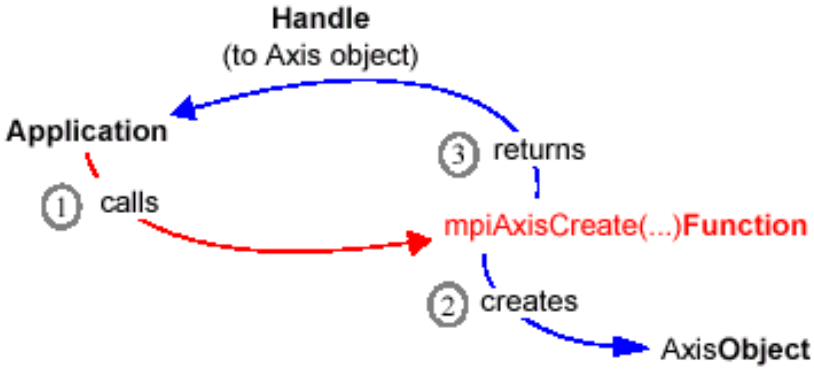
When an application calls a **function** (also called a **method**) to create an object, the function returns a handle to the object. The object handle is then passed as the first argument to all other functions (methods) that are associated with that object. These functions provide indirect access to the object's data structure. When it is finished using an object, an application simply calls another function to delete it.



Handle: a pointer to the pointer to an object.

Object: a temporary area in memory where data and related code is stored.

Example	The MPI defines an Axis object. An Axis object is created by the mpiAxisCreate(...) function, which returns a handle of type MPIAxis. The mpiAxisStatus(...) function takes an Axis handle as its first argument and returns the status of the hardware axis with which the Axis object is associated. An Axis object is deleted by calling the mpiAxisDelete(...) function.



TOP

Handle

A handle is returned by the function that creates an object, and the handle uniquely identifies that object. A handle is simply a pointer to a pointer to an object. The MPI defines the type **MPIHandle** to be a generic handle. The symbol **MPIHandleVOID** is used to represent an invalid handle. The MPI functions that create and delete objects typically allocate and free memory from a heap; in this case a handle is generally a pointer to the allocated object data structure. However, an MPI implementation could define a handle to be a pointer into a static array of data structures, or an index into such an array. Regardless, **an application MUST NOT do anything with a handle other than to pass it as an argument to other functions.** In cases where a handle is a pointer, the pointer cannot be dereferenced, because the object data structure to which the handle points is not directly available to the application.

Example	The MPI defines an Axis handle to be of type MPIAxis. An axis handle named axisHandle can be declared as follows: MPIAxis axisHandle;

TOP

Method

A function associated with an object is called a method. The first argument of a method is always the handle of the object with which it is associated. A function that is not associated with an object is called a function. **Throughout the MPI documentation, the term *function* refers to methods and functions, whereas the term *method* only refers to methods.**

Example	The following functions are all considered to be methods: mpiAxisCreate(...), mpiAxisStatus(...), mpiAxisDelete(...). Except for mpiAxisCreate(...), these functions all take an MPIAxis handle as their first argument.

```
MPIAxis axis;
axis = mpiAxisCreate(…);
mpiAxisStatus(axis, …);
mpiAxisDelete(axis);
```

TOP

Module

A **module** is a source file (module.c) whose interface is declared in a header file (module.h). A module contains code symbols (methods, functions, macros) and data symbols (data, data types and constants) that together implement an object. However, a module may also contain functions that have a logical connection without being associated with an object.

MPI Module	Header
Adc	adc.h
Axis	axis.h
Capture	capture.h
Command	command.h
Control	control.h
Event	event.h
EventMgr	eventmgr.h
Filter	filter.h
Idn	idn.h
IdnList	idnlist.h
Message	message.h
Motion	motion.h
Motor	motor.h
MpiDef	mpidef.h
Node	node.h
Notify	notify.h
Object	object.h
Recorder	recorder.h
Sequence	sequence.h

XMP Module	Header
Client	client.h
Element	element.h
Firmware	firmware.h
Flash	flash.h
List	list.h
Map	map.h
MeiDef	meidef.h
Packet	packet.h
Platform	platform.h
Remove	remove.h
RipTide	riptide.h
Server	server.h
Trace	trace.h
XMP	xmp.h

Utility Module	Header
msgCHECK	msg.h
Service	service.h

Sercos	sercos.h	Log	log.h
StdMpi	stdmpi.h	LogFunc	logFunc.h

TOP

Example

The axis module consists of a source file called **axis.c** and a header file called **axis.h**. The axis.c source file contains the definitions of all Axis code and data symbols, in particular the Axis object data structure. The axis.h header file contains the declarations for the code and data symbols in axis.c.

Introduction | Motion Concepts | General Definitions | Naming Conventions | Object Descriptions

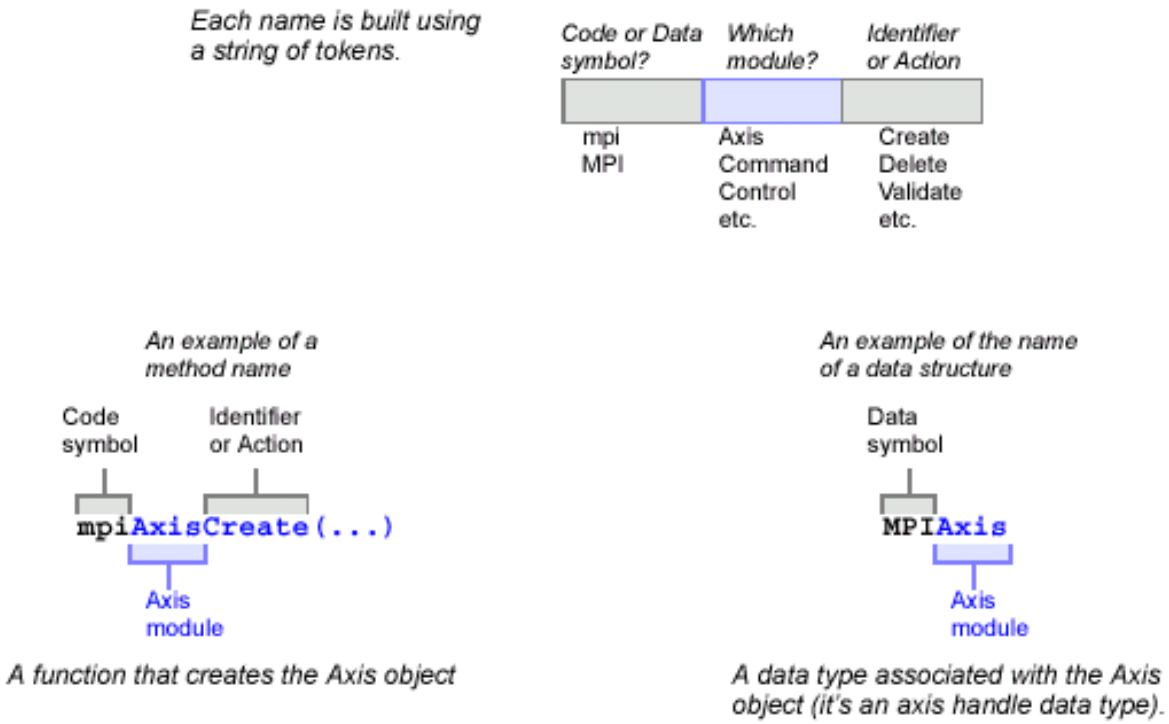
NEXT

Naming Conventions

Uniqueness | Symbol Declaration & Definition | Symbol Naming | Data Symbols | Code Symbols

Uniqueness

We have tried to make the names of all MPI code and data symbols unique, so that the names don't conflict with symbols already defined in the supported operating systems, and in C language and third-party libraries. The MPI naming convention enables you to look at a symbol and know whether the symbol refers to code or data. The MPI uses "MPI" as the initial token of data symbols, while code symbols use an initial token of "mpi".



Example	The axis handle data type is called MPIAxis. The method that creates an axis object is called mpiAxisCreate(...).

MPI/XMP: MEI-specific additions and extensions to the MPI are named similarly with an initial token of "MEI" indicating a data symbol and "mei" indicating a code symbol.

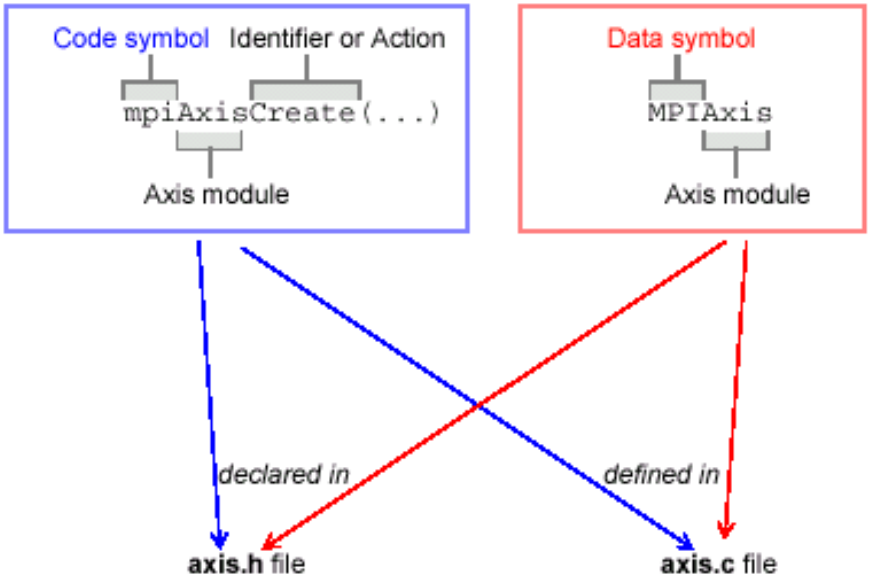
Example	The MEIAxisConfig{} data structure contains XMP-specific axis configuration parameters that extend the MPI-defined axis configuration parameters. The meiFilterCreate(...) function creates a Filter object, returning a handle of type MEIFilter. The Filter object is an addition to the MPI.

Symbol Declaration & Definition

MPI symbols are declared in header files and defined in source files. To determine the file in which an MPI symbol is declared and defined, look at the second token of the symbol name: the second token indicates the module where the symbol is declared and defined. Also, to maximize the portability of the code, the MPI uses a DOS-like 8.3 file naming convention. Therefore, the second token of any symbol name will not exceed 8 characters, nor will the names of modules and objects.

For example, consider a symbol named MPIObjectXyz. The symbol is a data symbol (MPI) declared in object.h and defined in object.c. Similarly, a symbol named mpiObjectAbc is a code symbol (mpi) declared in object.h and defined in object.c.

Example	The MPIAxis data type and the mpiAxisCreate(...) method are declared in the header file axis.h. The MEIServer data type and meiServerDelete(...) method are declared in header file server.h. The code and data symbols for the Axis object are defined in axis.c; for the Server object, they are defined in server.c.



TOP

Symbol Naming

The previous sections use the concept of a token when discussing symbol names, where a token is a word or abbreviation contained in a symbol name. The name of an MPI symbol consists of a left-to-right sequence of tokens (with no spaces in between). An uppercase letter indicates the beginning of a token, except the first token of a symbol, which may be either upper (MPI) or lower case (mpi). The MPI uses underscores in symbol names only when it is necessary to separate tokens that are all uppercase letters.

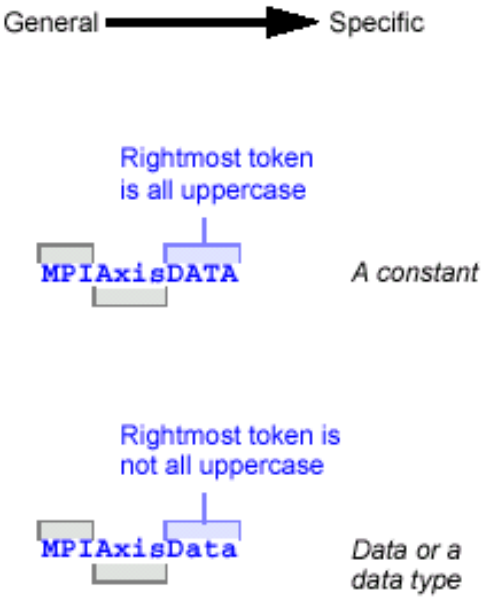
The first token of a symbol (MPI or mpi) guarantees uniqueness and indicates whether the symbol is a data symbol (MPI) or a code symbol (mpi) . The second token indicates the module that declares and defines the symbol. The remaining tokens of the symbol name vary, depending on whether the symbol is code or data.

TOP

Data Symbols

Tokens in data symbols are arranged from left-to-right in a hierarchical fashion, with the leftmost token being general and the rightmost token being specific. If the data symbol names are listed alphabetically, this scheme results in an listing that groups related data symbols together.

If the rightmost token of a data symbol is all uppercase, then that data symbol is a constant. If the rightmost token of a data symbol is not all uppercase, then that data symbol is either data or a data type. Because the MPI contains virtually no global data, there is little need to distinguish between data and data type.



Consider the MPIMotorHomeIndex enumeration:

```
typedef enum {
    MPIMotorHomeIndexINVALID = -1,

    MPIMotorHomeIndexHOME_HI_INDEX,
    MPIMotorHomeIndexINDEX_ONLY,
    MPIMotorHomeIndexHOME_LO_INDEX,
    MPIMotorHomeIndexHOME_ONLY,

    MPIMotorHomeIndexLAST,
    MPIMotorHomeIndexFIRST = MPIMotorHomeIndexINVALID + 1
} MPIMotorHomeIndex;
```

The third and fourth tokens (Home, Index) are the same for all symbols. The enum name (MPIMotorHomeIndex) is incorporated in the name of all of the enum members, whose last token is all uppercase (FIRST, HOME_HI_INDEX, etc.) to indicate that they are constants (instead of a data type). It is standard coding practice to use enums rather than #defines to declare constants, which allows for greater control of the name space and better type-checking. The use of FIRST and LAST members in an enum is also standard coding practice, so that functions can bounds-check enum values.

TOP

Code Symbols

Tokens in code symbols indicate the data type and order of the arguments to the function. The last token in a code symbol is often the action that the function performs, or the data type that the function returns. If the code symbol names are listed alphabetically, this scheme results in a listing that groups related code symbols together.

If the rightmost token of a code symbol is all uppercase, then that code symbol is a macro.

If the rightmost token of a code symbol is not all uppercase, then that code symbol is either a function or a method.

Consider mpiAxisConfigGet(...) and mpiAxisConfigSet(...), which are 2 methods used to configure an Axis. Because these functions are methods, the first argument to them is an Axis handle. The second argument is a pointer to a structure of type MPIAxisConfig{}. The last token of the method name indicates the action to take; Get fills the supplied structure with Axis configuration and Set configures the Axis from the supplied structure.

The mpiAxisControl(...) method returns the MPIControl handle with which the Axis was created, while meiElementNEXT(...) is a macro that returns the MEIElement following the specified Element.

TOP

Object Descriptions

Control Objects

Think of a Control object as a “Controller” object, which is typically a single circuit board residing in a PC or an embedded system. A Control object manages that motion controller board.

Every application creates a single Control object per board (XMP controller).

A Control object can read and write device memory using I/O port, memory-mapped or device driver methods. All communication with motion controller firmware is handled by a Control object.

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.
Objects taht are created with a single Control

Object	Relationship to a Controller
Adc	many-to-one, must reside on that controller
Axis	many-to-one
Filter	one-to-one **or** many-to-one
Motion	many-to-one **or** many-to-many
Motor	many-to-one
Recorder	one-to-one
Sequence	many-to-one

TOP

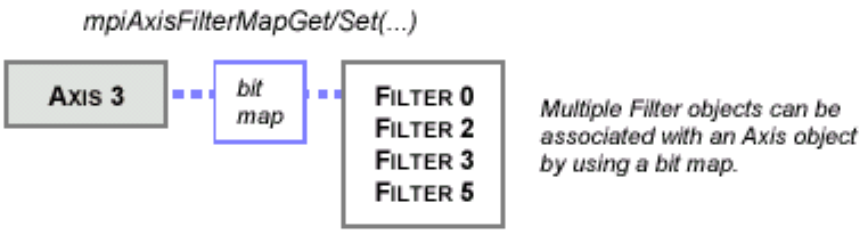
Axis Object

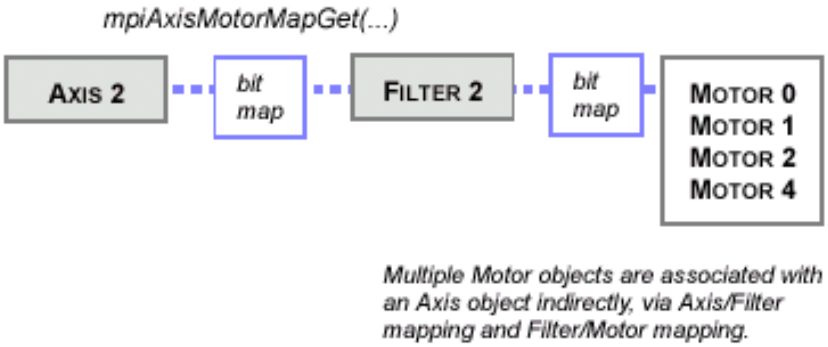
An Axis object is associated with a single physical axis on a motion controller, and corresponds to a geometric axis used for calculation of a path of motion. An Axis may be controlled by one or more Motion objects.

The concept of an Axis is a “geometric” idea, but the main purpose of an Axis object is to generate the desired path (trajectory calculations, i.e., to generate command positions) on every sample, using the path-planning data provided by a Motion Supervisor. An Axis object is mostly a computational block.

The Filter and Motor objects ensure that the command path (calculated by Axis) is followed, and that the signals get to the right motors.

Axis Objects Are Mapped to Filters and Motors





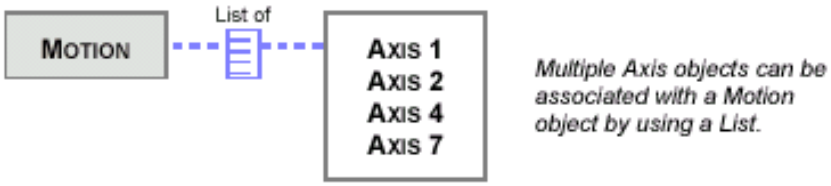
TOP

Motion Objects

Think of a Motion object as really a “Motion Supervisor” object.

The Motion (Supervisor) object corresponds to a coordinate system or collection of axes. The primary function of the Motion Supervisor is to provide data in a synchronized manner to the Axis objects for use in path creation.

A second important function of the Motion Supervisor is to monitor the status of all of the Axes under its control (and all of the Motors, and Filters associated with these Axes), so that motion can be stopped or resumed in a controlled manner, especially in the event of errors. The Motion Supervisor is the primary interface for a your MPI application with respect to motion. A Motion object maintains an ordered list of Axis objects, which specify the coordinate system for all motions to be performed with that Motion object. When a motion is started, the type of motion is specified (trapezoidal, S-curve, parabolic, etc.) along with type-specific motion parameters. A Motion can be started directly (by calling a Motion method), or started when the Motion is associated with a Command that is called by a Sequence (that is executing). An Axis object may be controlled by more than one Motion object, but only one of those Motion objects may be active at a time.



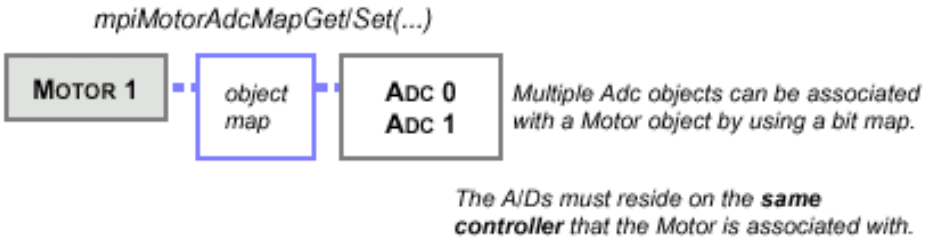
TOP

Motor Objects

The Motor object corresponds to a physical motor used to create motion. The primary function of the Motor object is simply to provide an interface to the physical hardware associated with the physical motor. The Motor object could practically be called the “I/O” object (only the User I/O is handled outside of the Motor object). Essentially, the Motor object is the controller’s interface to the outside world.

The data of the Motor object contains the state of encoders, limit switches, home sensors, amplifier control and status signals (amplifier enable, fault, step and direction), and general purpose digital and analog I/O. Secondary functions of the motor object include commutation control, limit checking, and position capture and compare control.

To perform sinusoidal commutation, the Motor takes the outputs from the Filter object. The Motor object also implements scale interpolation.

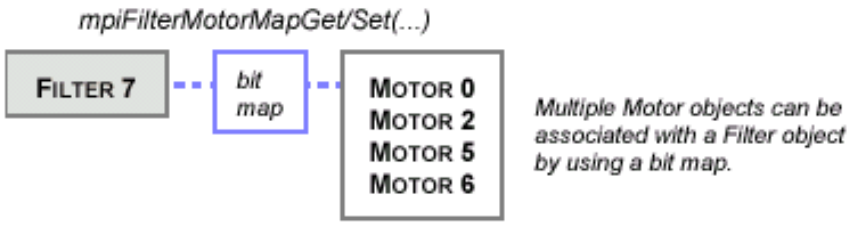


TOP

Filter Objects

The Filter object is concerned with the controller’s control loop, i.e., what should the controlled output be (usually an input to a Dac) based on the position error? The Filter is primarily a computational block, taking command positions and actual positions and computing errors.

The Filter object calculates the output (usually the D/A output level) that controlls a physical motor or motors, using data (command positions) calculated by the Axis object. PID, PIV and Biquad filter calculations are all parts of the Filter object.



TOP

Event Objects

An Event object contains information about an asynchronous event that has occurred. You typically obtain an Event object from the EventMgr object.

TOP

Notify Objects

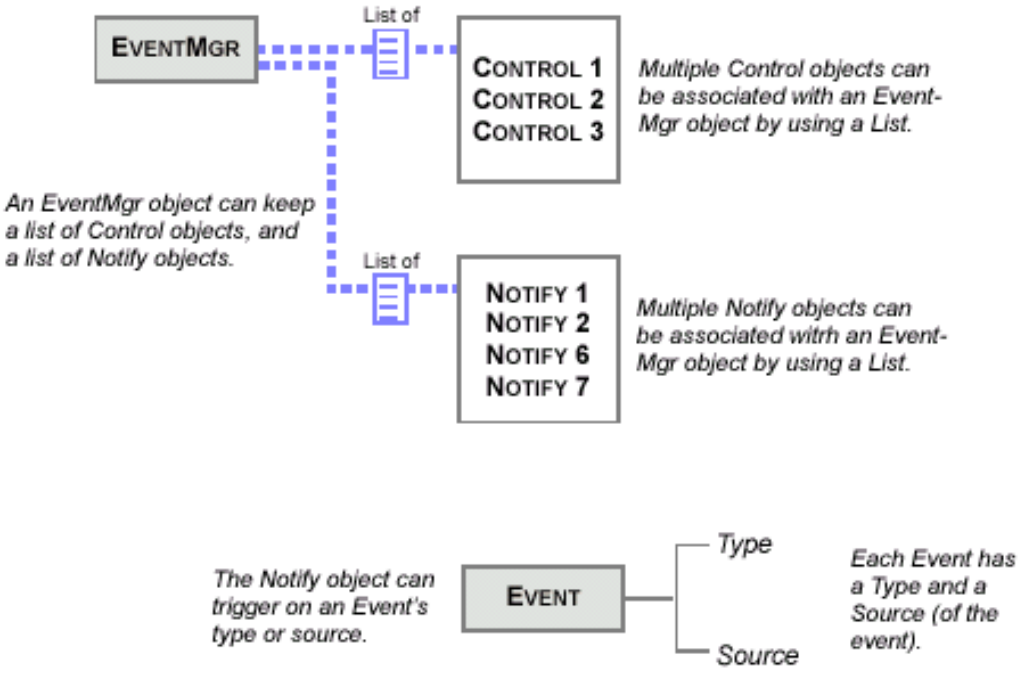
A Notify object is used by a thread to wait for event notification.You can configure a Notify object to wait and look for specific events and for specific event sources.

TOP

Event Manager (EventMgr) Objects

An EventMgr:

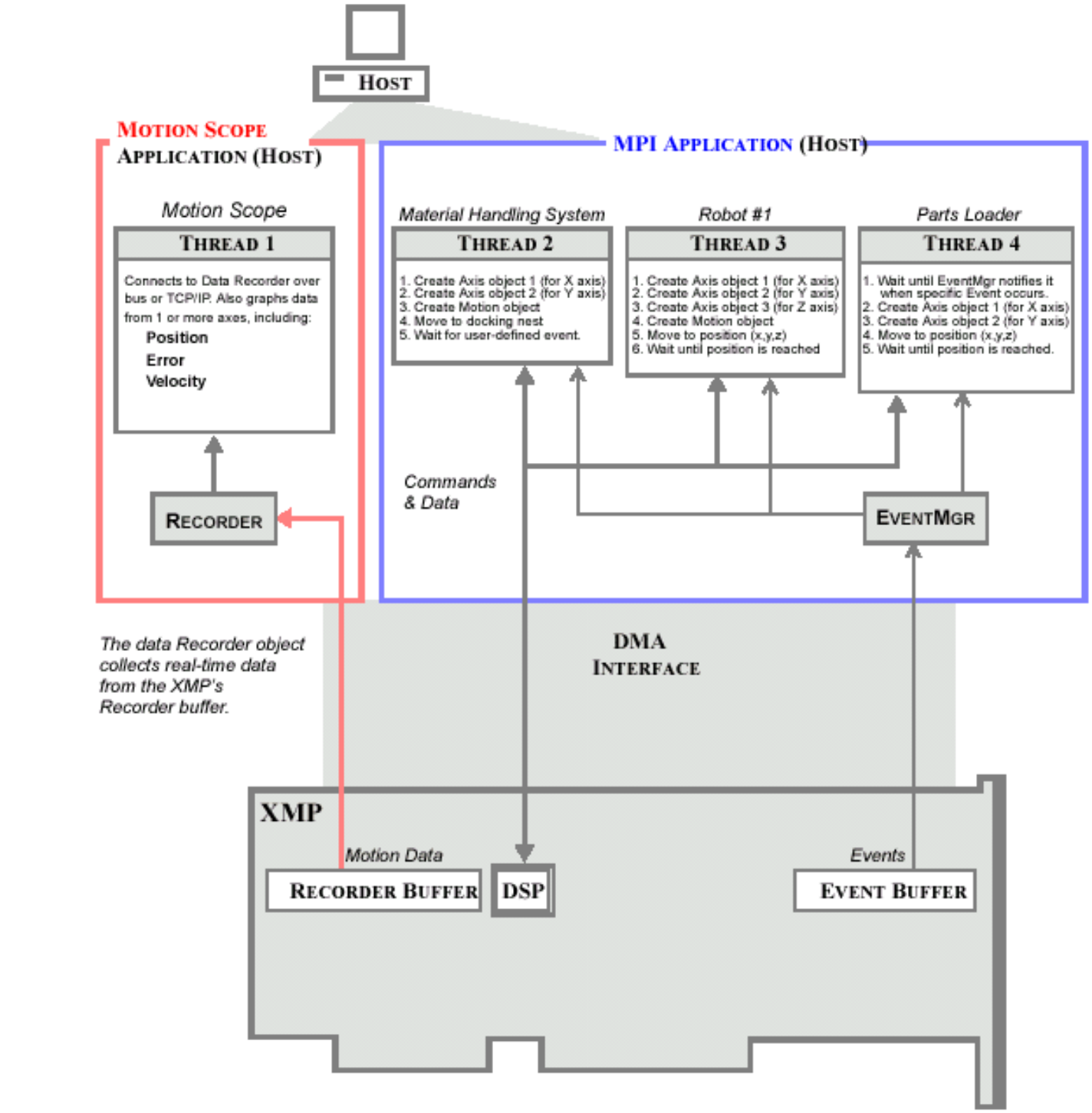
- obtains asynchronous events from the Control object(s) that the EventMgr is associated with
- generates Event objects for enabled event sources
- awakens any threads that are waiting for events



TOP

Recorder Objects

You typically use the Recorder object to periodically record motion data. Note that you can only have one Recorder object per Control object (controller).



TOP

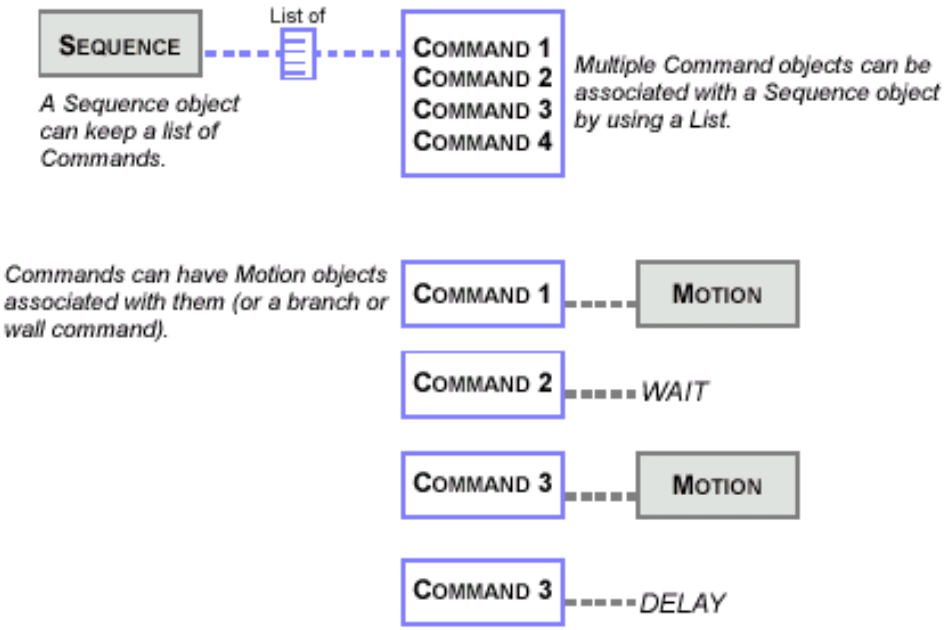
Sequence Objects

A motion application can issue individual motion commands, or can create a series of motion commands (that are executed in sequence by the controller). Essentially, you can use the Sequence object to download commands that are executed by the XMP controller, and not executed by the host.

Using the MPI, you can create a sequence of commands (a Sequence object), using high-level motion (e.g., trapezoidal motion profile on axes 2 & 4), using low level work (e.g., write 0x00043433 into memory address 0x2000).

A typical Sequence might be

1. Start a motion
2. Wait 60 msec
3. Turn on a specific I/O bit
4. Wait for motion to finish
5. Wait for another I/O bit
6. Start a new motion



A Sequence object is always executed starting with the first command. After completing the first command, subsequent commands can initiate a new motion, set values in firmware memory, execute a delay for a specified period, branch to a different command in the sequence, wait for a condition to be met, and so on.

TOP

Command Objects

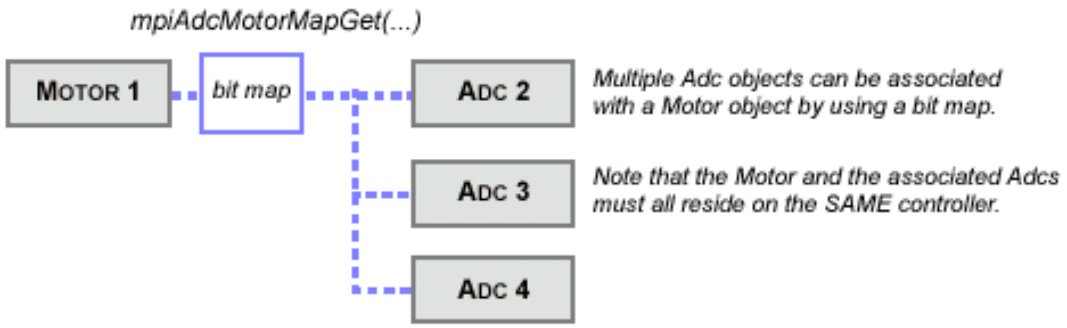
A Command object specifies a single action that is executed by a Sequence, such as motion, conditional branch, computation, time delay, wait for condition, etc. Any Command object that specifies motion must have a Motion object associated with it.

TOP

Adc Objects

Adc objects manage the A/D converters on a controller (Control). Typically the A/Ds can be programmed to perform conversions on different channels. On every timer interrupt, the A/D performs a single conversion, which takes 12 microseconds to complete.

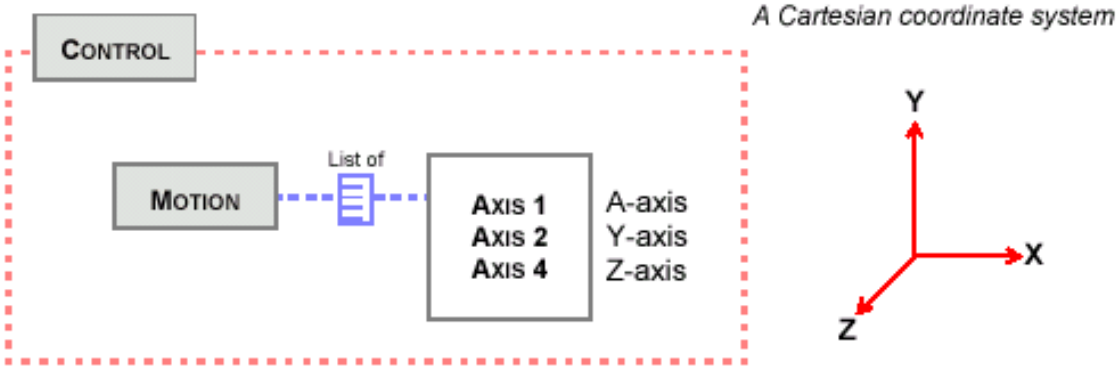
A 16-Axis XMP controller has 8 A/D channels available for your application's use. Each Adc can be correlated to a Motor object.
Note: For the XMP, no more than 2 Adc objects can be associated with a single Motor.



TOP

Coordinate Systems

To create a coordinate system for a motion application, you simply associate a list of Axis objects with a Motion object. The list of Axis objects and the order of those Axis objects on the list essentially define the coordinate system.



TOP

Introduction | Motion Concepts | General Definitions | Naming Conventions | Object Descriptions

Return to Software's Main Menu