SOFTWARE TOPICS

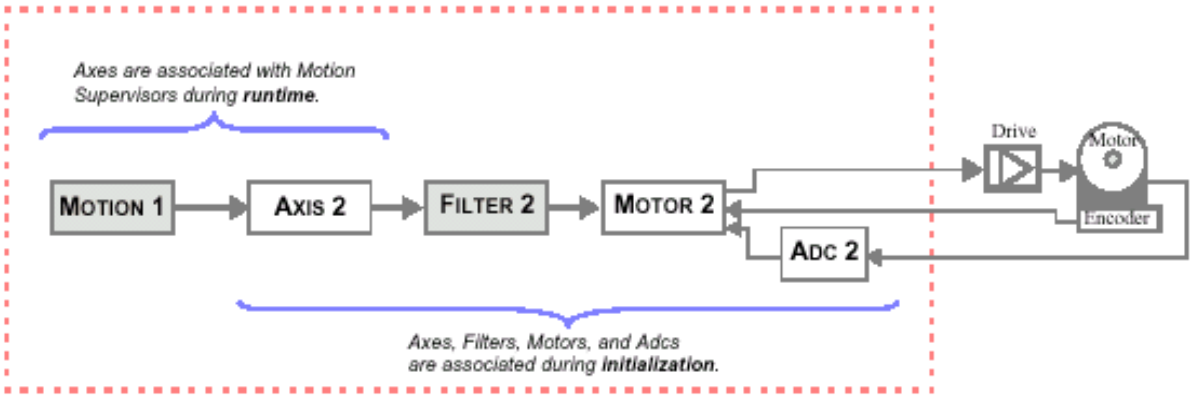
--Table of Contents--

Objects Relationships

Introduction | Object List | Bitmap | Array of Object Numbers
Using Lists, Bitmaps, & Arrays | Application-MPI-Controller Interface | SERCOS Interface

Introduction

To create a motion application, you create objects, link them in relationships (associate them), and provide the desired parameters for motion.



There are three ways that you can associate one object with another object:

- Using an object List
- Using bitmap
- Using an array of object numbers

Objects using bitmaps or an array of numbers must all be present physically on the same controller, while objects on an object list can come from different controllers (such as objects on the EventMgr control list).

Three Ways to Associate Objects with Objects

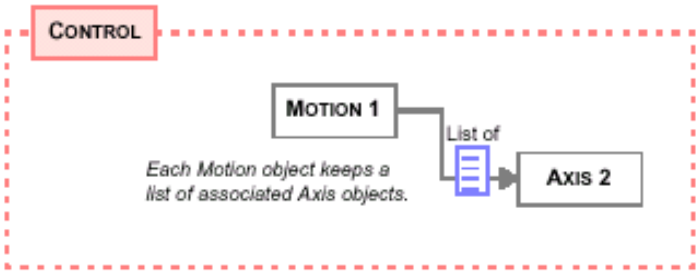
Object		When to Use
List	an ordered list	**Use a List** when the order of the objects to be associated is important, OR when some of the objects reside on different controllers.
Bitmap	an un-ordered list	**Use a bitmap** when the order of the objects is not important, AND the objects must all reside on the same controller (same Control object).
Array of object numbers	an ordered list	**Use an array of object numbers** when the order of the objects is important, AND the objects must all reside on the same controller (same Control object).

TOP

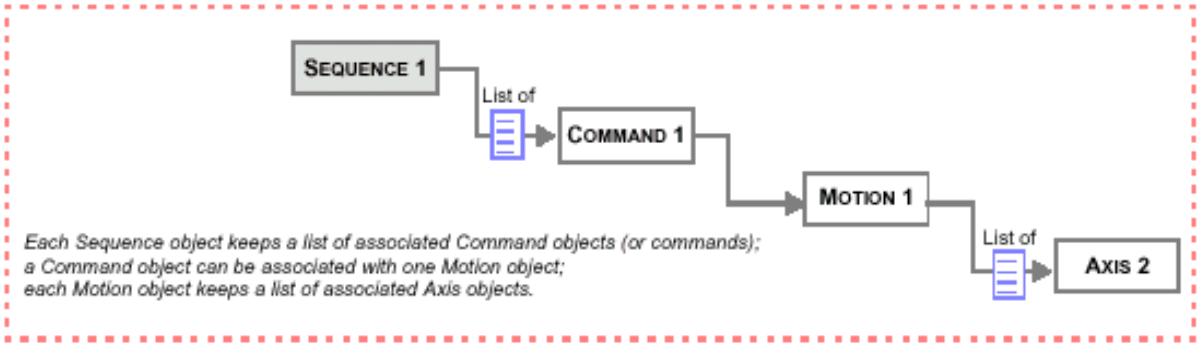
Object List

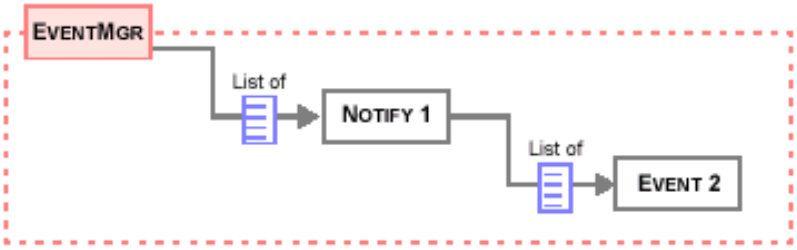
Use an object list to associate objects when ordering is required, such as the Axis list maintained by a Motion object. For example, when using a Motion object, a list of Axes 0, 1, 2 is not the same as a list of Axes 0, 2, 1. Such a distinction is not available when using an object map. Refer to the List methods for Motion, Notify, Sequence and EventMgr objects.

This object:	Can have object lists of:
Motion	Axes
Notify	Events
EventMgr	Controls Notify objects
Sequence	Commands



OR

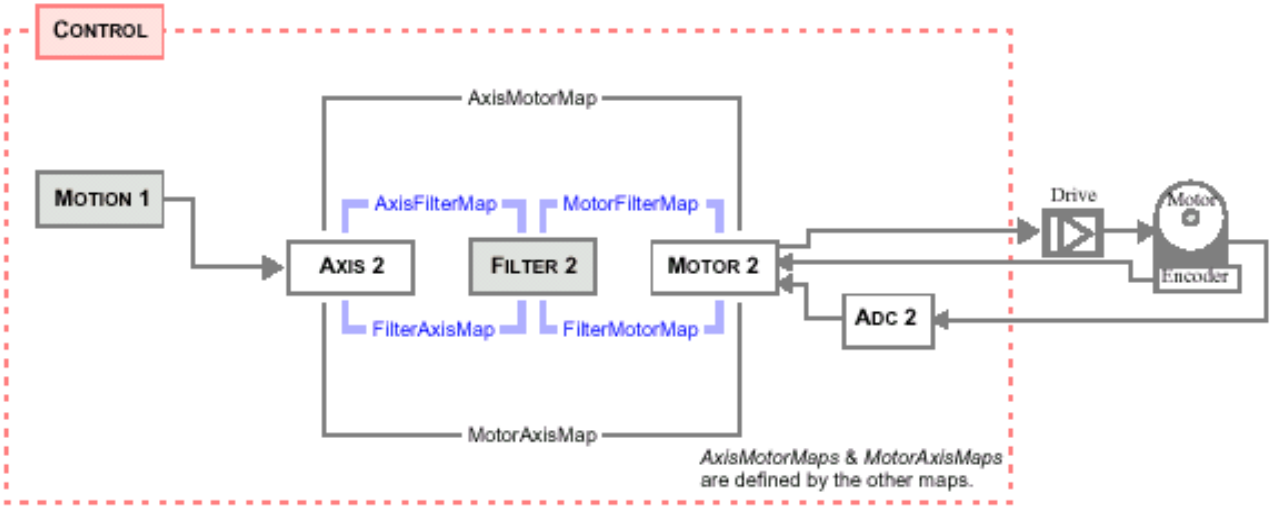




TOP

Bitmap

An object map is a bitmap, where each numbered bit represents the presence or absence of the correspondingly numbered object. You can order the objects in the bitmap from numeric low-to-high or high-to-low, but there is no other meaningful capability for ordering objects with an object map.



Another consideration with object maps is that all objects specified in the map and the object (that those objects are being associated with) must be resident on the same controller (Control object).

Note: Although the MPI allows an Axis to be associated with a Motion without regard to the controller (Control), the XMP implementation of the MPI does not allow this.

Object maps tend to be used to associate low-level objects that:

- must all reside on the same controller (Control)
- and the order in which the objects are ordered does not matter

Note that the objects specified in the bitmap may be associated with another object, without those objects having to be created and deleted. A goal of the MPI has been to minimize the need to create and delete objects, especially objects that are just used for configuration. For example, an MPI application that uses the default configuration doesn't have to create Filter or Motor objects, yet the application can still configure an Axis, by calling mpiAxisFilterMapSet(...), where the Filters are specified by the bitmap.

Object Maps for Objects

This object:	Has associated object maps of:
Filter	Axes Motors
Axis	Filters Motors
Motor	Filters Axes Adcs

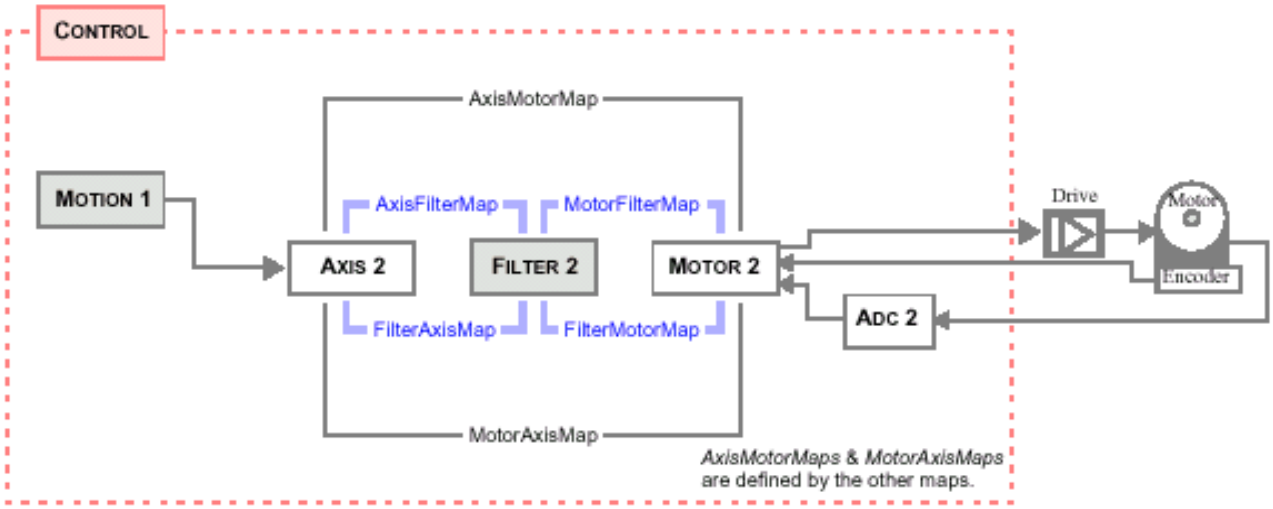
TOP

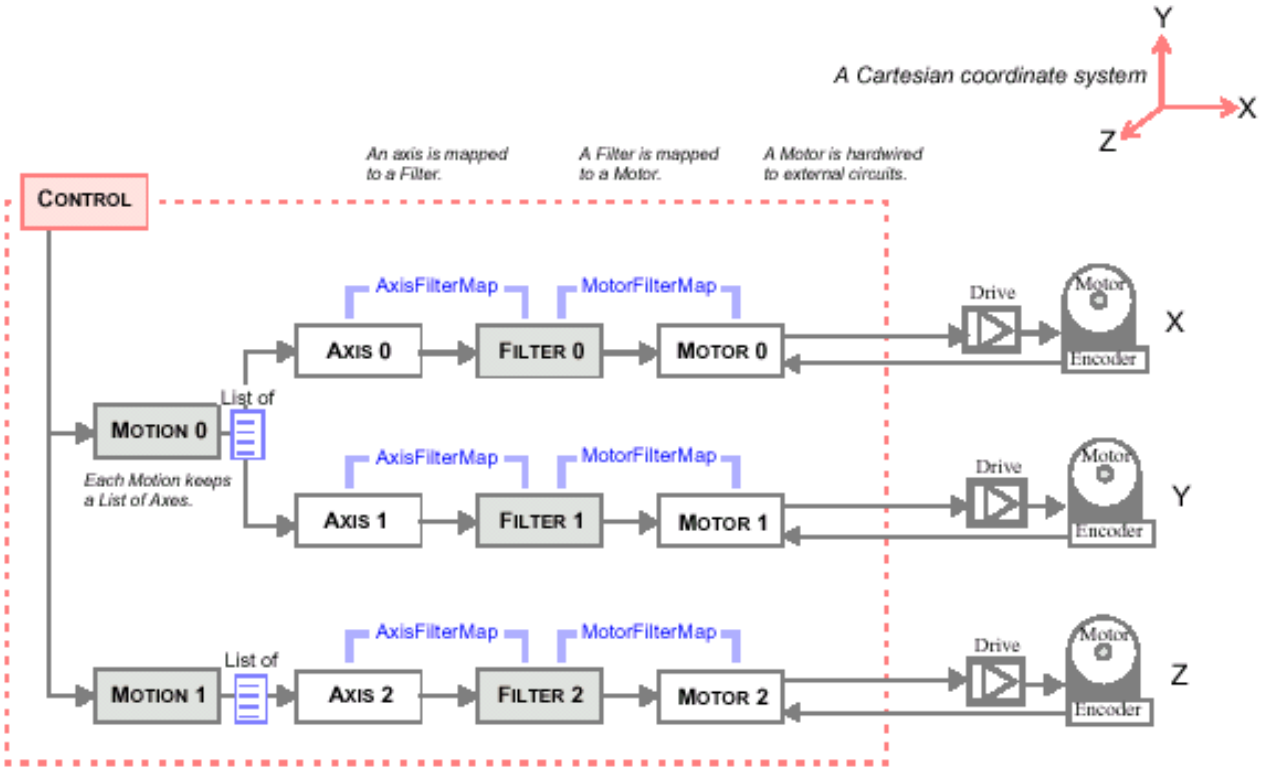
Array of Object Numbers

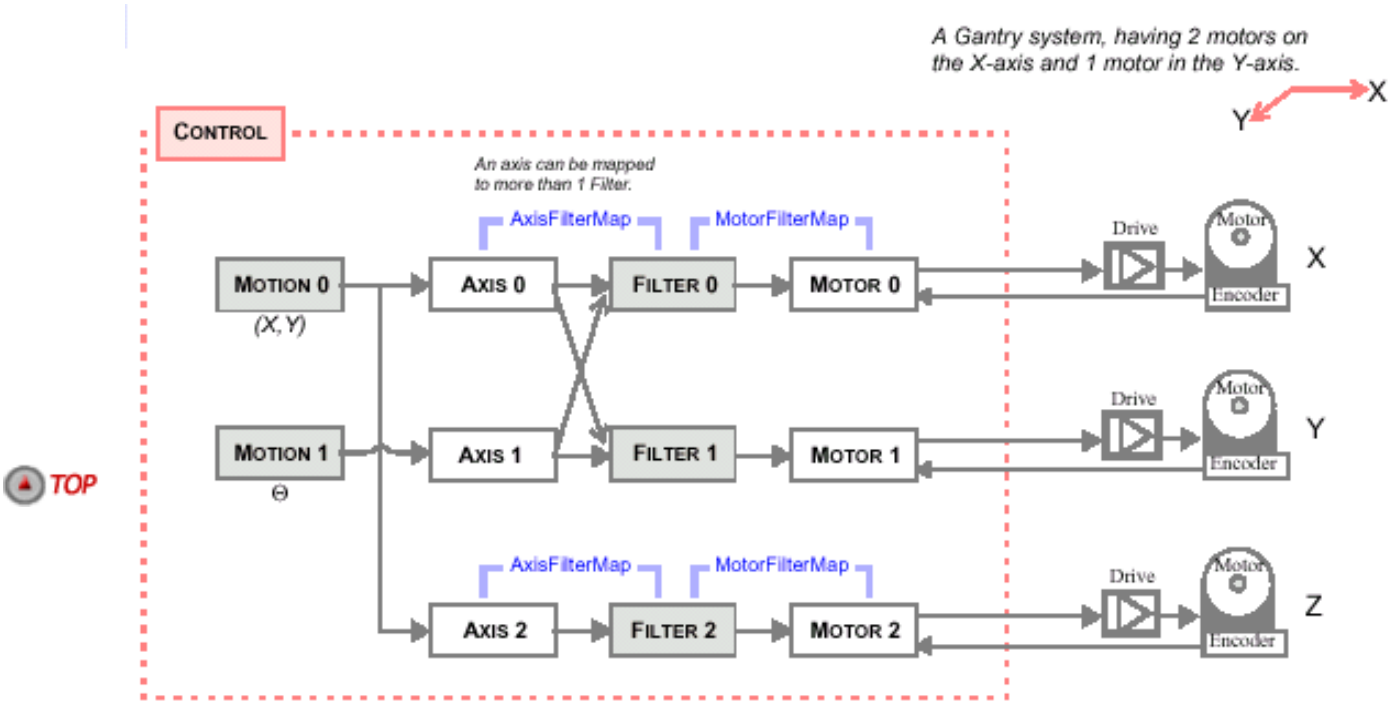
The MPI provides a third means of associating objects, by using object numbers. Object numbers should be used when ordering is required, and all the resources arelocated on the same controller.

TOP

Using List, Bitmaps, & Arrays

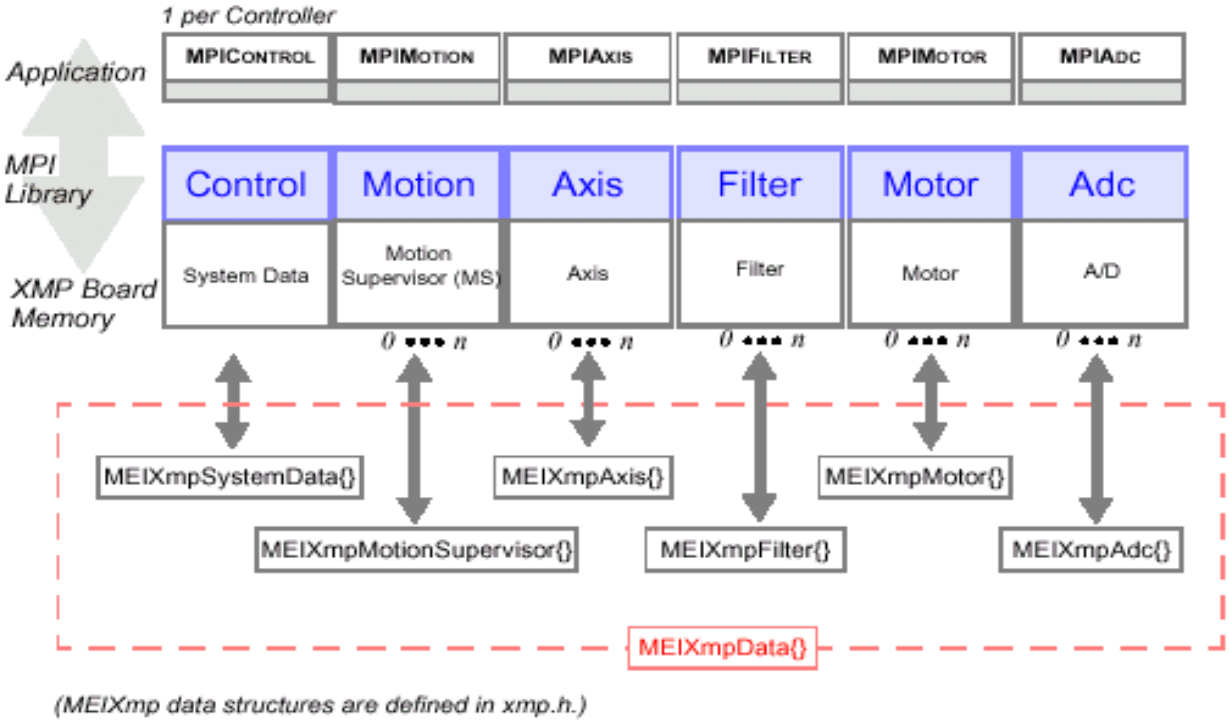




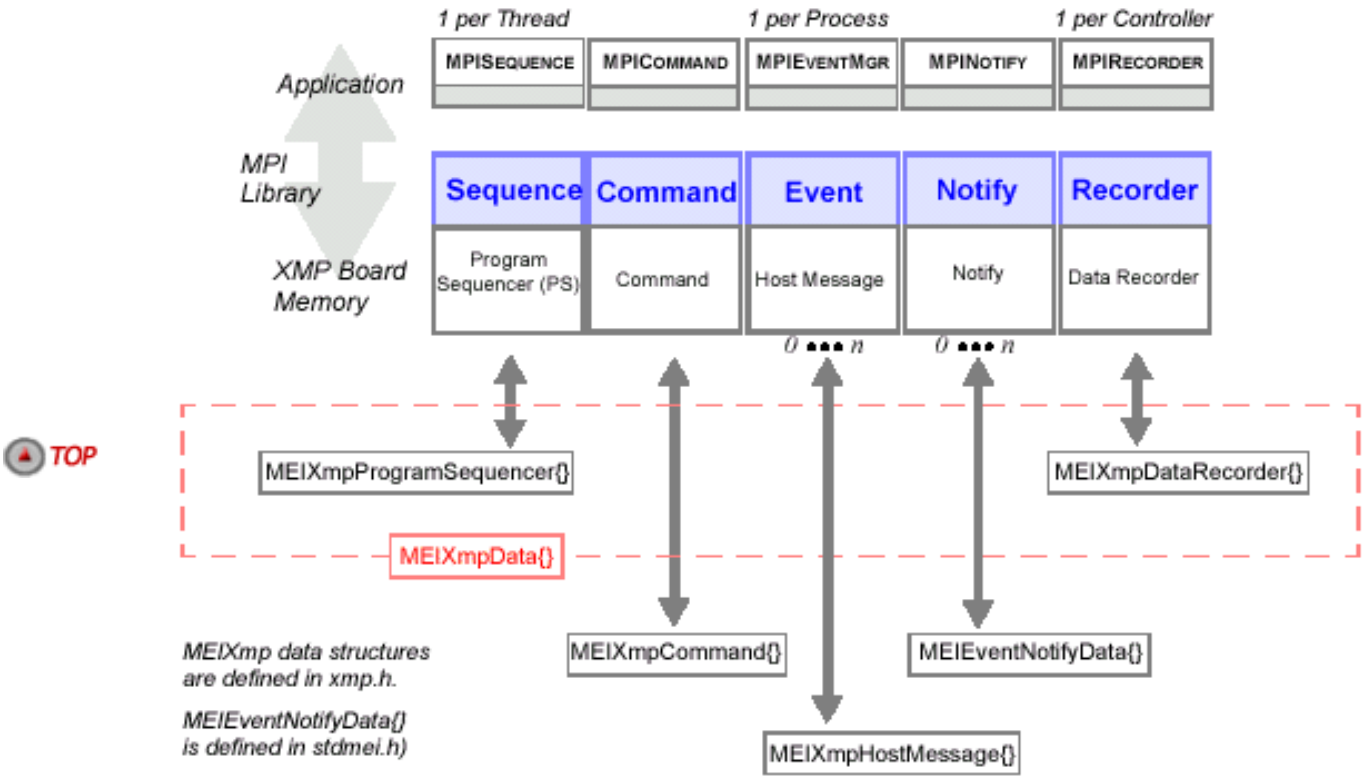


TOP

Application-MPI-Controller Interface



(MEIXmp data structures are defined in xmp.h.)



TOP

SERCOS Interface

The standard XMP controller uses an analog interface. The analog interface requires many wires to make the connections between amplifiers and I/O. The XMP/Sercos controller uses a fiber optic interface coupled with a deterministic serial communications protocol to take the place of the wires needed with the analog XMP.

SERCOS allows the XMP controller to transmit more than just an analog reference signal to an amplifier. SERCOS allows the XMP controller to transmit position, velocity, torque, I/O, tuning parameters, and other configuration parameters to an amplifier. Because the data is transmitted to an amplifier in digital form, the amplifier will contain an on-board processor. Besides parsing the command data and sending feedback, the on board processor gives the drive many capabilities. SERCOS drives have the ability to close a position, velocity, and/or torque loop, execute internal homing procedures, as well as latch position based on digital inputs.

Despite the fact that the XMP/SERCOS controller has a different physical interface than the standard XMP controller, the same MPI is used for both controllers. The complexity of the XMP/SERCOS controller's interface has been hidden by the MPI and by the firmware running on the XMP. This means that the same development tools can be used for both controllers. The XMP/SERCOS controller requires that an initialization procedure be executed before commands can be sent to amplifiers. Once the initialization procedure has successfully executed, the XMP/SERCOS controller acts the same as the standard analog XMP controller.

NEXT

TOP

Host-Based and Controller-Based Motion

Host-Based Motion | Controller-Based Motion | Sequence

Host-Based Motion

Basic Motion

NOTE: The terms “Control object” and “Controller” designate the same thing: a Control object.

To perform a simple motion using the XMP controller requires as few as eight basic steps:

1. Create a Control object
2. Initialize the Control object
3. Create an Axis Object
4. Create a Motion Object
5. Append an Axis to a Motion object’s list (of axes)
6. Start the Motion Check the Status of the Motion
7. Delete the Objects

In more detail,

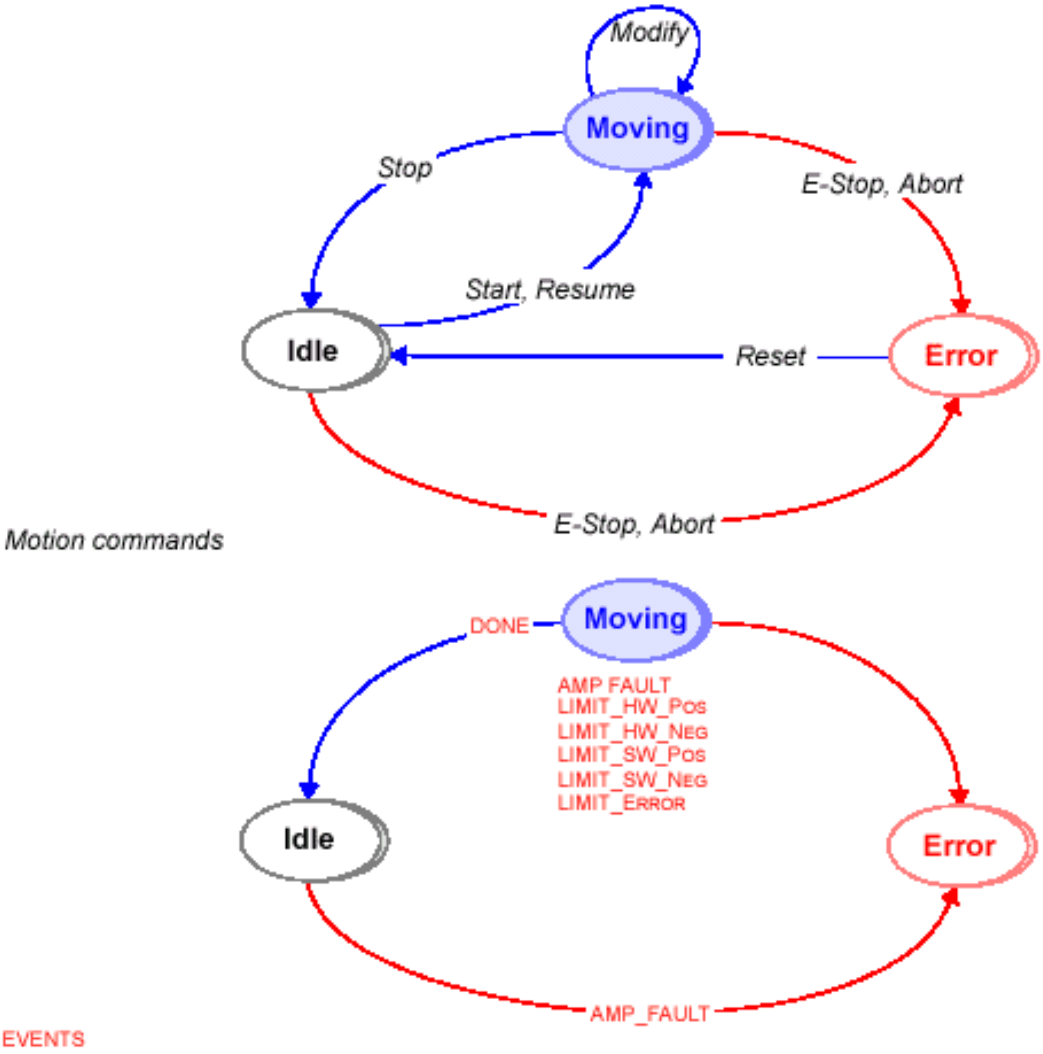
1. **Create a Control Object:** Use mpiControlCreate(...) to create a Control object. You should only create one and only one Control object per board per process. The Control object controls all of the operating system resources for the actual board. You must create a Control object before creating any other objects.
2. **Initialize the Control Object:** Use mpiControlInit(...) to initialize a Control object. mpiControlInit(...) establishes contact with the device driver, and maps the XMP’s memory space to the application’s memory space. An application can call mpiControlInit(...) more than once, but it isn’t necessary.
3. **Create an Axis Object:** Use mpiAxisCreate(...) to create an Axis object, by passing a Control object handle and Axis number to it.
4. **Create a Motion Object:** Use mpiMotionCreate(...) to create a Motion object, by passing a Control object handle and a Motion Supervisor (number) to it. Each motion object maintains a list of axes, which defines a coordinate system (specified by the number and order of the axes).
5. **Add an Axis to a Motion object’s list (of axes):** Use mpiMotionAxisAppend(...) to associate an Axis with a Motion. The association is done by appending the Axis to a Motion object’s list of axes. Note that the Axis must be located on the same controller (board) as the Motion object.
6. **Start the Motion:** Use mpiMotionStart(...) to start a Motion.
7. **Check the Status of the Motion:** Use mpiMotionStatus(...) to determine if the current motion has completed yet.
8. **Delete the Objects:** Use mpiObjectDelete(...) methods to delete the objects. We recommend deleting objects in the reverse order of creation. For this example, you would perform the object deletions in the order: Motion first, then Axis, then Control. Note that for any application, the Control object must be deleted last.

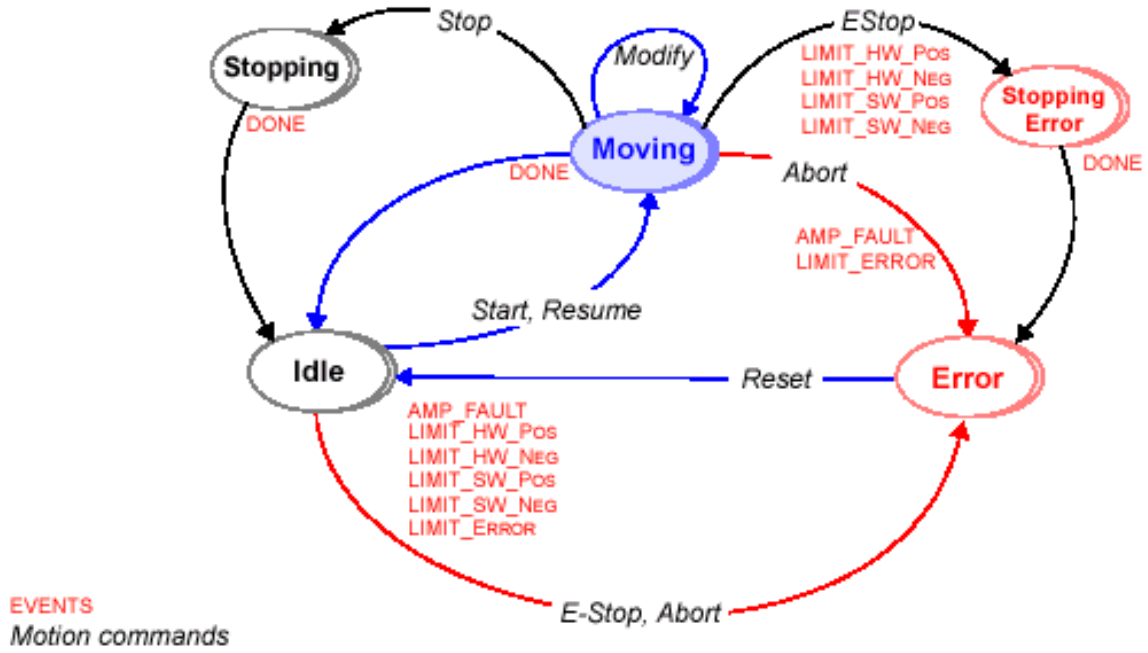
TOP

Controller-Based Motion

Motion Command

The XMP controller's firmware uses a state machine for controller-based motion. The motion state machine has three states (Moving, Idle, Error). Transitions between states occur after a command (Start, Resume, Modify, Stop, E-Stop, Abort, Reset) or after an event (Done, Amp_Fault, Limits).





MPIEvents are generated by the XMP controller. MPIEvents (axis events) cause MPIActions to be generated. The MPIActions in the table are the default values and can be changed. Valid MPIActions for events are None, Stop, EStop, Abort.

MPIEventType	MPIAction (Default)	Event Trigger
AMP_FAULT	ABORT	Polarity (I/O bit Hi/Lo)
HOME	STOP	Polarity (I/O bit Hi/Lo)
LIMIT_HW_POS	ESTOP	Polarity (I/O bit Hi/Lo)
LIMIT_HW_NEG	ESTOP	Polarity (I/O bit Hi/Lo)
LIMIT_SW_POS	ESTOP	Position
LIMIT_SW_NEG	ESTOP	Position
LIMIT_ERROR	ABORT	Error [= absolute value (command position - actual position)]

MPIActions can come from the host or the firmware. For safety, there are 4 guaranteed restrictions on who can generate certain MPIActions:

- The firmware can never issue Start. (**Never** 1 in table)
- The firmware can never issue Resume. (**Never** 1 in table)
- The firmware can never issue Reset. (**Never** 1 in table)
- The host can never issue Done. (**Never** 2 in table)

1 = Start/Resume/Reset can never be issued from the firmware; Start/Resume/Reset can only be issued from the host.
2 = Done can never be issued from the host; Done can only be issued from the firmware.

TOP

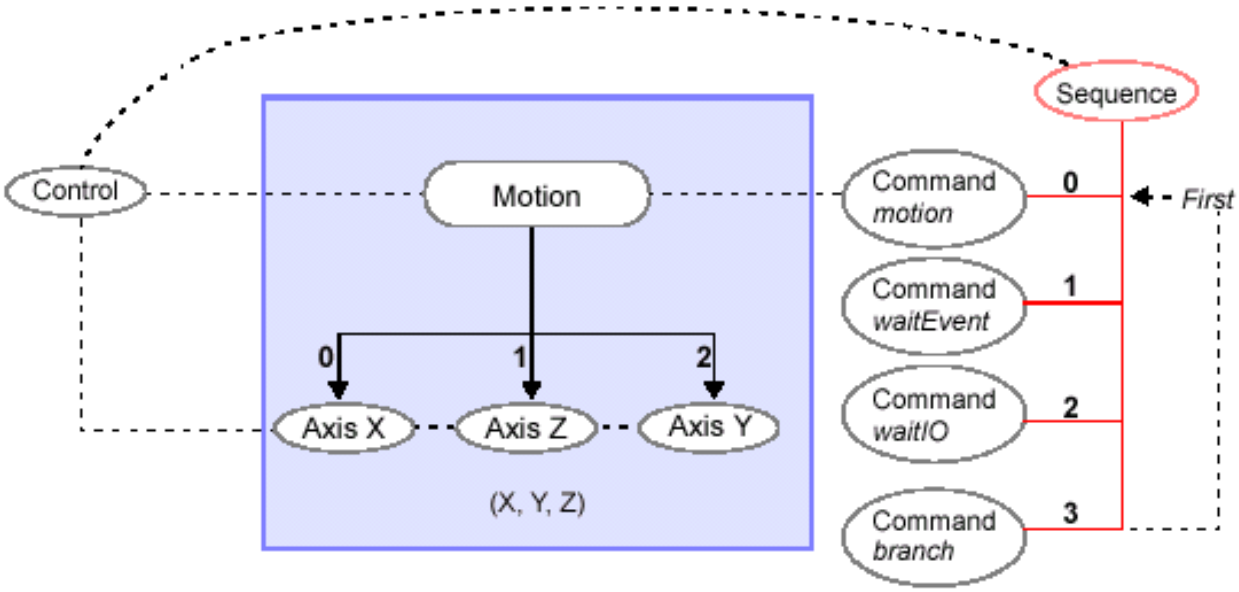
MPIAction	Originating from Host	Originating from XMP Firmware
Start	mpiMotionStart(...)	**Never** 1
Resume	mpiMotionResume(...)	**Never** 1
Reset	mpiMotionReset(...)	**Never** 1
Stop	mpiMotionStop(...)	Event
E_Stop	mpiMotionEStop(...)	Event
Abort	mpiMotionAbort(...)	Event
Done	**Never** 2	If Settled and In_Position

1 = Start/Resume/Reset can never be issued from the firmware; Start/Resume/Reset can only be issued from the host.
2 = Done can never be issued from the host; Done can only be issued from the firmware.

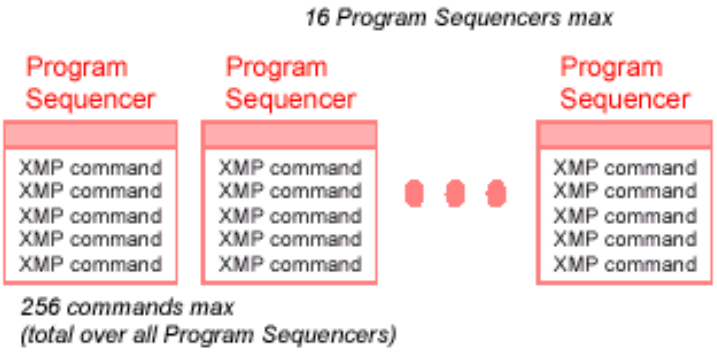
Sequence

A Sequence consists of a linked list of Commands. Execution of a Sequence can be started, stopped, or its status queried. Execution of a Sequence begins with the first Command in the sequence and proceeds sequentially, unless otherwise commanded (e.g., a branch command). Execution of a Sequence ends when the last Command in the sequence is executed, or when the execution is stopped by the host.

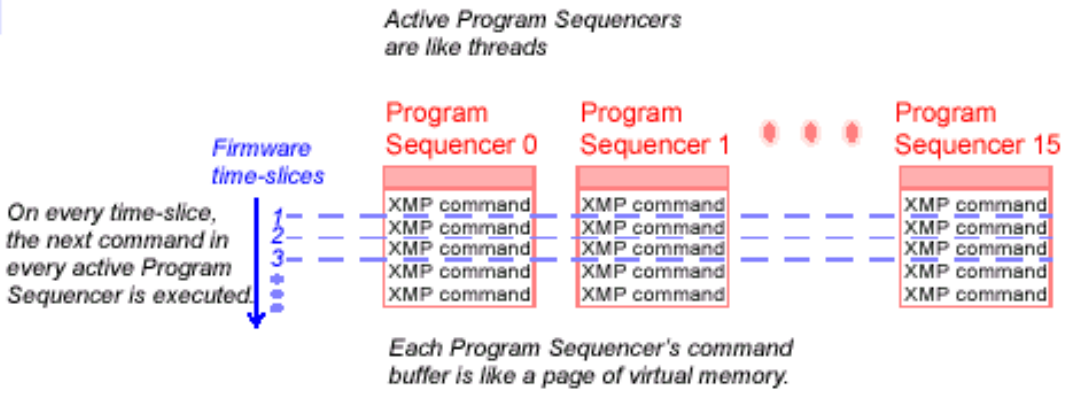
An XMP can be controlled directly by the host or controlled by commands that the host downloads (for independent execution by the XMP firmware). Downloaded commands are managed by a firmware object known as a Program Sequencer (PS).



For the XMP, there are currently 16 Program Sequencers and a single command buffer that holds 256 commands. Each Sequence is created with a maximum command count, but the total command count for all Program Sequencers cannot exceed 256 commands.



The XMP commands can be thought of as a machine language for a virtual RISC processor (Reduced Instruction Set Computer) implemented by the XMP firmware. Similarly, a Program Sequencer can be thought of as an operating system thread, with the PS command buffer considered as a page of virtual memory for the thread. Each PS maintains the location within its command buffer of the next command to be executed, i.e., a program counter. The XMP firmware time-slices the PS "threads" by executing a command in every active PS, on every firmware cycle. When the next command to be executed is not located in the PS command buffer, a host interrupt is generated (i.e., a page fault occurs). The host is then expected to fill the PS command buffer with the next batch of commands. This method allows command sequences to be of any length.



The Motion Programming Interface (MPI) provides a layer of abstraction on top of the XMP Program Sequencer. The MPI Sequence object maintains a list of MPI Command objects. Commands exist to perform motion, compute/load/store data, branch-on/wait-for conditions (arithmetic, logical, I/O bits, events), delay, generate user-defined events, etc. There are a variety of command types, each with type-specific command parameters. The high-level language defined by these commands makes use of MPI objects, and allows for simple expressions with operands (that can be specified by value or by reference). A command can have a text label so that the command itself can be the target of a branch command.

When you create a Sequence, you specify the desired XMP PS command buffer size. Specifying a PS command buffer size of -1 will cause all remaining command buffer space to be allocated. After creating a Sequence, you typically create and append Command objects to the Sequence, and a Sequence can have any number of Command objects appended to it. When the Sequence is started, the Commands are "compiled" into XMP commands and loaded into the PS command buffer for execution.

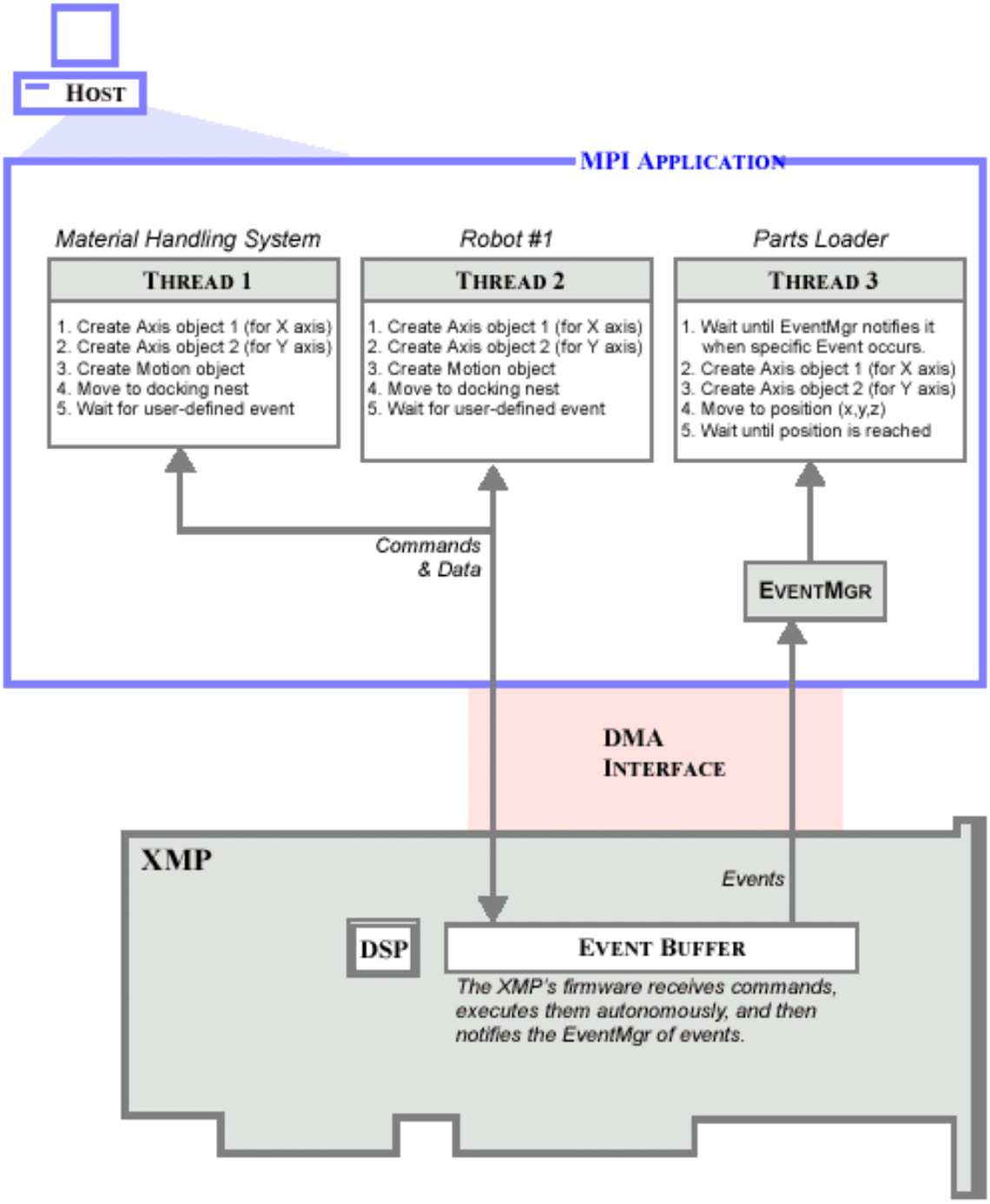
TOP

When the XMP firmware does not find the next command to be executed in the command buffer, the firmware generates a host interrupt. This interrupt is handled by the MPI EventMgr, which then calls the Sequence to load the next batch of commands down to the Program Sequencer; your application does not have to do anything.

XMP Multitasking

Introduction

The MPI provides multitasking capabilities in the host, while the XMP can execute commands and receive data without host intervention.

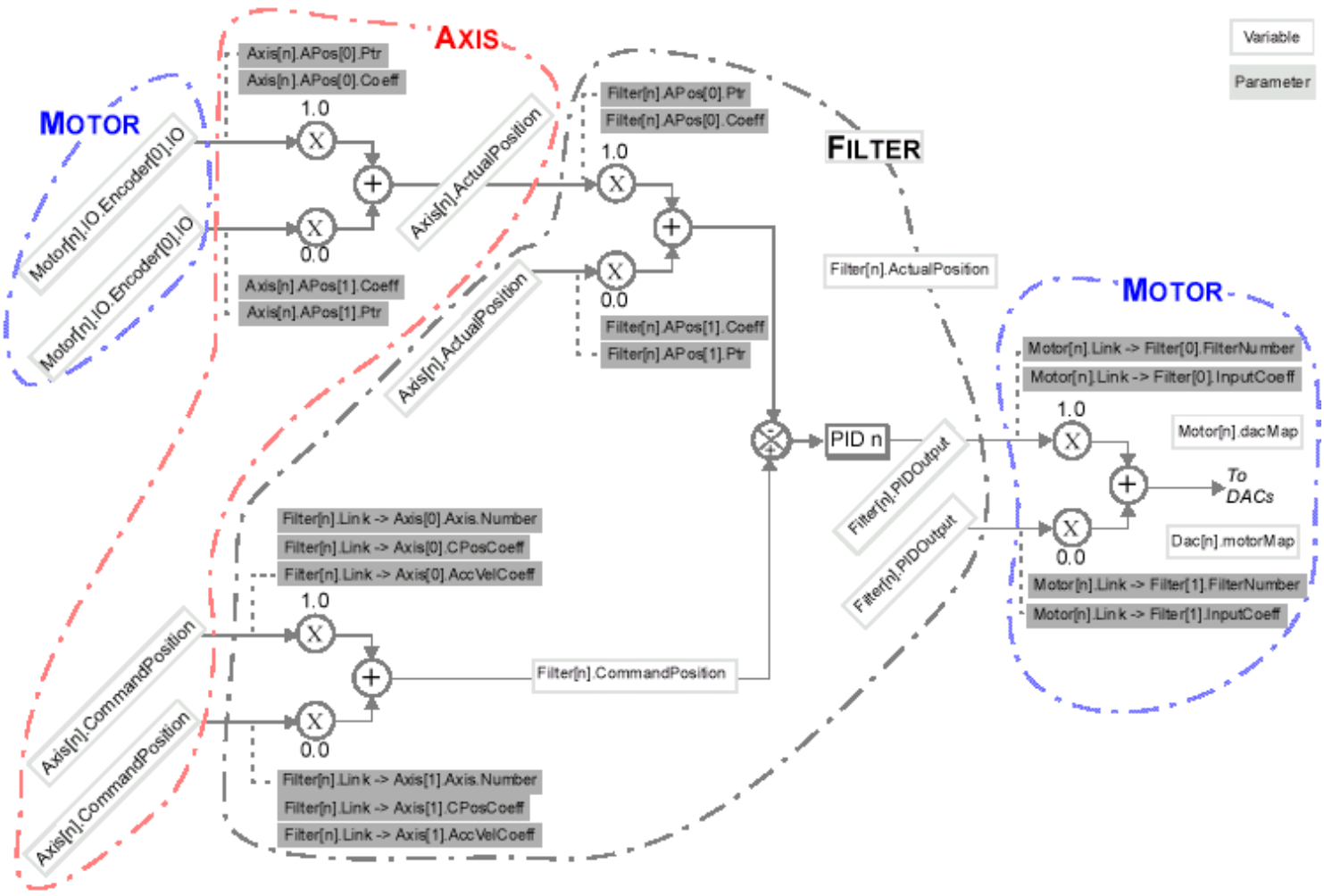


NEXT

TOP

XMP Configuration

For a print-friendly version of the diagram below, click here.



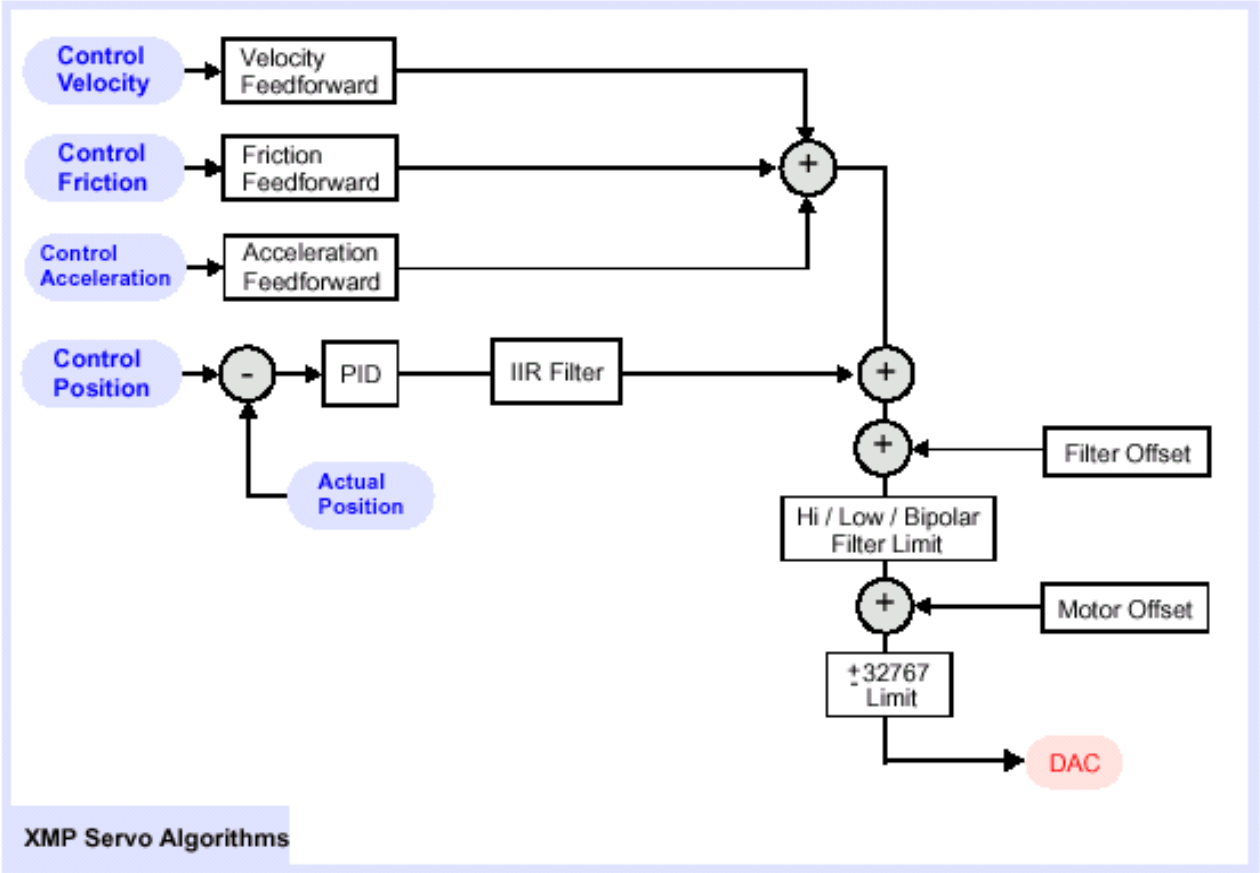
NEXT

TOP

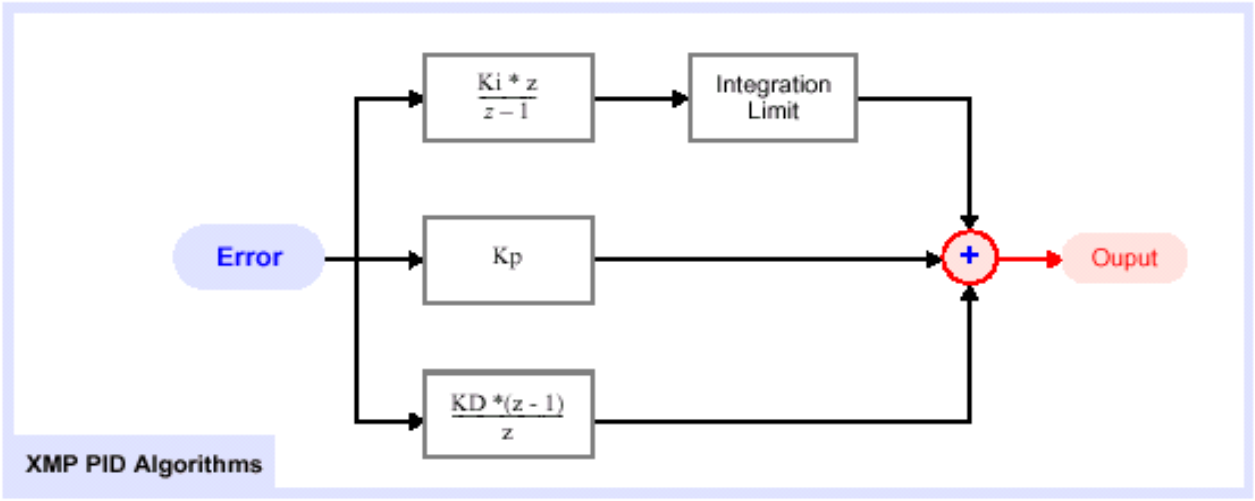
XMP Servo Algorithms

PID-Based | PIV-Based

PID-Based

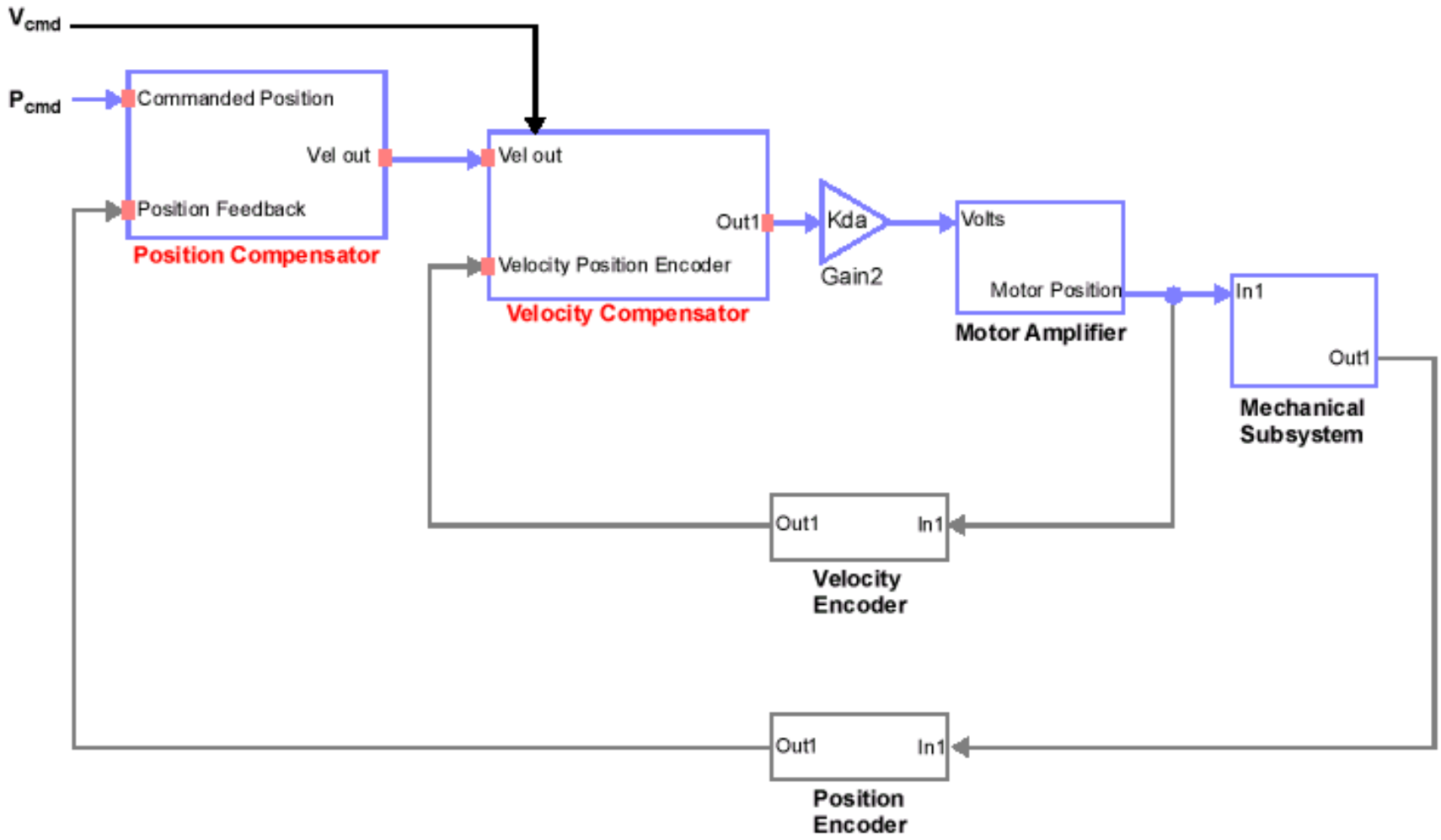


TOP



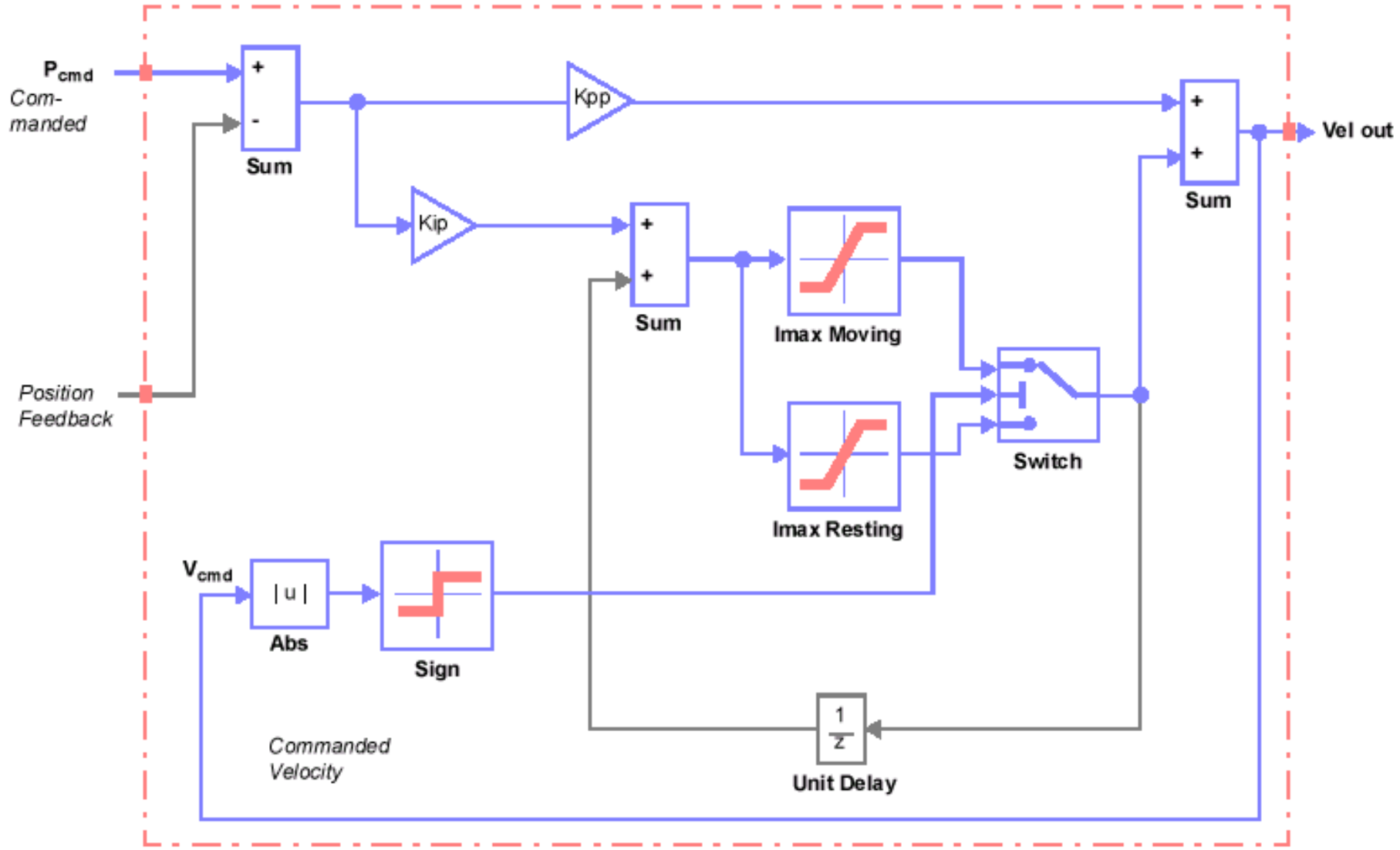
PIV-Based

PIV: Block Diagram



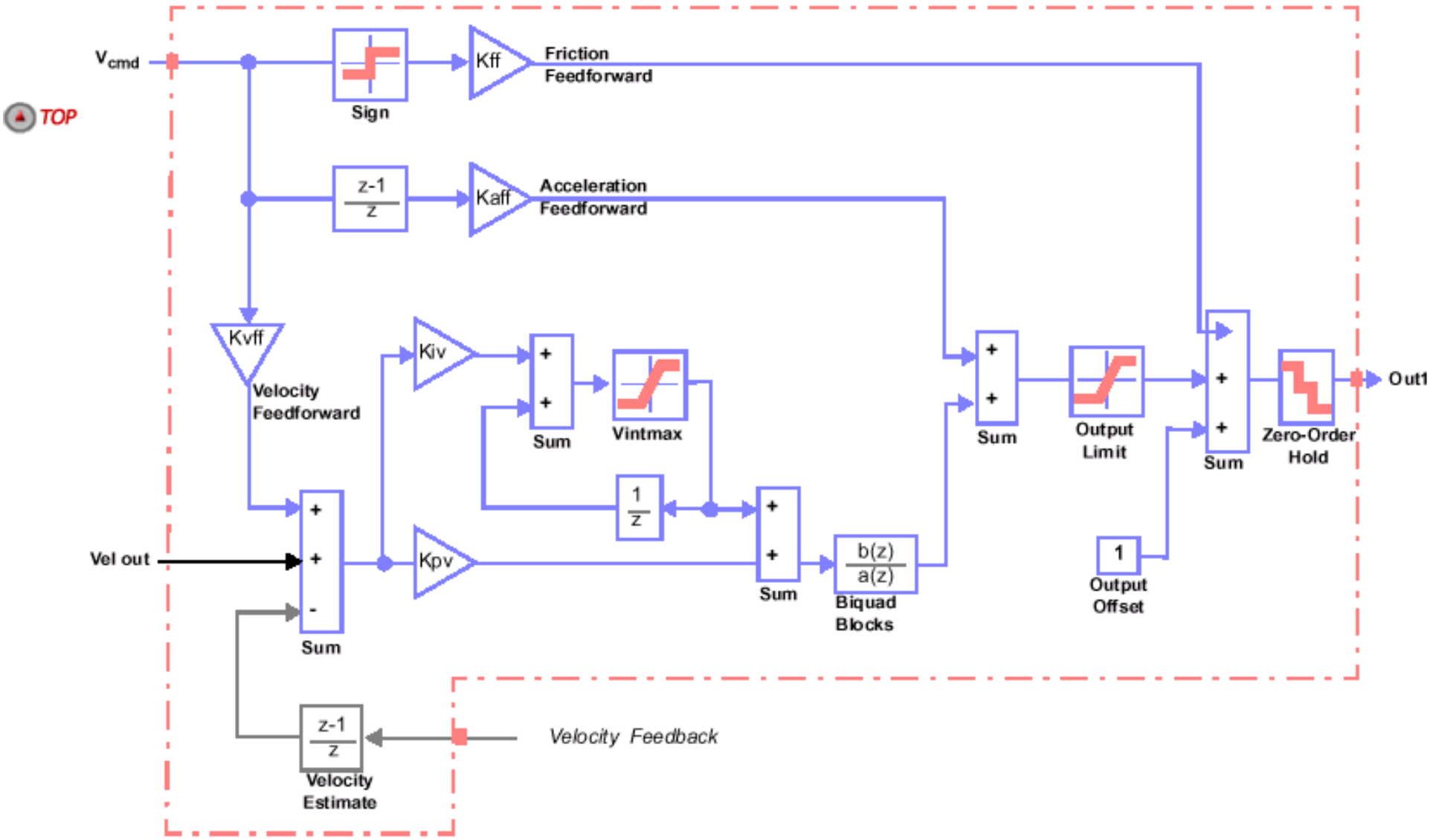
For a print-friendly version of the above diagram, click here.

PIV: Position Compensation



For a print-friendly version of the above diagram, click here.

PIV: Velocity Compensation



For a print-friendly version of the above diagram, click here.

NEXT

Project / Makefile Settings: Symbol Definitions

MPI_DECL1/MPI_DECL2 | MEI_ASSERT | MEI_PLATFORM

Introduction

This section describes compile-time symbol definitions used by the MPI library.

MPI_DECL1/MPI_DECL2

These symbols are used in library header files to handle compiler-specific language extensions when declaring external functions and data; in particular, those extensions used when making Win32 dynamic link libraries (DLLs). MPI_DECL1 precedes the return type of a function declaration; MPI_DECL2 follows the return type and precedes the function name:

```
MPI_DECL1 long MPI_DECL2 mpiModuleFunction();
```

When building an application or the MPI library for Win32 platforms (Windows 95/NT), the following symbol definitions should be used:

```
MPI_DECL1=__declspec(dllexport)
MPI_DECL2=__stdcall
```

MEI_ASSERT

When defined, MEI_ASSERT will cause calls to the macro **mei**ASSERT(***expression***) to be compiled into the library. Otherwise, these calls will be discarded by the preprocessor.

The **meiASSERT(...)** macro is used by the library and is available to applications. It is a valuable debugging tool that can be used to catch programming errors at their source and prevent them from spreading. If the argument to the macro is TRUE (i.e. non-zero), execution proceeds normally. Otherwise, an error message is displayed; the message contains the name of the file and the line number of the **meiASSERT(...)** call. The application exits after displaying the message.

The MEI_ASSERT symbol should generally be defined except for when library size must be as small as possible and application execution speed must be as fast as possible.

MEI_PLATFORM

The MEI_PLATFORM symbols are used to indicate the platform for which the library and applications are built. Each platform has its own unique symbol.

Currently supported platforms:

MEI_PLATFORM_WIN95	Microsoft Windows 95
MEI_PLATFORM_WINNT	Microsoft Windows NT
MEI_PLATFORM_WINRTSS	VenturComm WinNT RealTime SubSystem

Currently supported platforms; however, platform- and version-specific (contact MEI for details):

MEI_PLATFORM_LYNXOS	Real-Time Systems LynxOS
MEI_PLATFORM_QNX	Quantum Software Systems
QNX MEI_PLATFORM_VXWORKS	WindRiver Systems VxWorks

Potentially supported platforms:

MEI_PLATFORM_DOS	Microsoft DOS
MEI_PLATFORM_IRMX	Intel iRMX
MEI_PLATFORM_OS9	Microware OS-9
MEI_PLATFORM_PSOS	Integrated Systems pSOS
MEI_PLATFORM_VRTX	Microtec Research
VRTX MEI_PLATFORM_WIN31	Microsoft Windows 3.1

Note: Existence of an MEI_PLATFORM symbol for a potentially supported platform does not imply intent to support that platform.

TOP

NEXT

About Homing

Capturing position is central to the concept of homing (even for Sercos XMPs). **The HOME event is triggered by the Capture Status instead of the Home Input (Motor[].IO.DedicatedIN.IO)**. Also note that each motor has its own default Capture Register (see the separate tip about capture numbering).

Currently the **only** configurable capture parameter (**mpiCaptureConfigSet(...)**) is captureConfig.trigger. CaptureConfig.trigger indicates which inputs are used for capturing (trigger.mask) and what polarity these inputs must be (trigger.pattern) to trigger a capture (HOME) event.

For non-Sercos XMPs, you can use 7 inputs to capture position: Home, Index, Positive and Negative Overtravel, and the three Transceiver inputs. These 7 inputs can be combined in a wide variety of ways, but the most useful are the *Home* and *Index* combination. Some designers also like to use Overtravel and Index combinations, and there is no problem doing this for non-Sercos drives.

For Sercos drives the selection is more limited. Only Index truly captures position, but all combinations of Home and Index can be used to cause a HOME event.

After a Capture object has been configured, capturing is controlled by **mpiCaptureArm(...)**. To put the capture latch in the armed state, call **mpiCaptureArm(...)** with a TRUE arm parameter.

Actual position capturing will then occur if the capture criteria (defined by trigger.mask and trigger.pattern) is **true** (for at least 100 nsec) *following a period* where the criteria is **false** (for at least 100 nsec). To disable capturing, call **mpiCaptureArm(...)** with a FALSE arm parameter.

Note that capture will not occur (for non-Sercos motors) if the capture criteria is already TRUE when **mpiCaptureArm(...)** is called. Capture will only occur if the criteria goes FALSE, and then returns to TRUE.

TOP

NEXT

Using the Origin Variable

In servo control applications, the Command and Actual Positions for an Axis are generally used for position error calculations. On each sample, the XMP controller reads the new Actual Positions from the feedback devices and calculates the new Command Positions.

To determine the relationship between Command and Actual Positions for an Axis and physical positions on the machine, you can use the Origin variable (which can be set by the host).

If it is important to set the Command or Actual Position to an exact value, you should use the mpiAxisOriginSet(...) function. Calling **mpiAxisOriginSet(...)** will modify both Command and Actual Positions by the same amount.

The Origin variable can be set in two different ways depending on the desired result. For example, assume the Command and Actual Positions are 10,000 and 10,024 counts respectively and that the axis is not moving.

Setting the Origin variable to 10,000 will result in Command Position = 0 and Actual Position = 24. Setting the Origin variable to 10,024 will result in Command Position = -24 and Actual Position = 0.

This second method is the most frequently used method for setting the new Origin after homing. Refer to the next table.

Position and Velocity criteria for generating motion completion events

If Command Position =	If Actual Position	And you set Origin Variable	Then Command Position =	And Actual Position =
10,000	10,024	10,000	0	24
10,000	10,024	10,024	-24	0

Cautions about motion in general:

1. During motion, you should never call the **mpiAxisOriginSet(...)** and **mpiAxis[Actual|Command|Position]Set(...)** functions. If you do call one of these functions, the XMP controller will immediately act on the new positions, causing the move to complete at a different point than the point specified in **mpiMotionStart(...)**.
2. Setting a new Command Position using **mpiAxisCommandPositionSet(...)** should be done with caution, because the XMP controller will immediately try to servo to the new position. If the new and old command positions differ by a large amount the axis may fault (limit error) or jump to the new position at very high speed.
3. The effect of **mpiAxisOriginSet(...)** and **mpiAxis[Actual|Command]PositionSet(...)** calls will not be seen by the host until the next controller sample following the call.

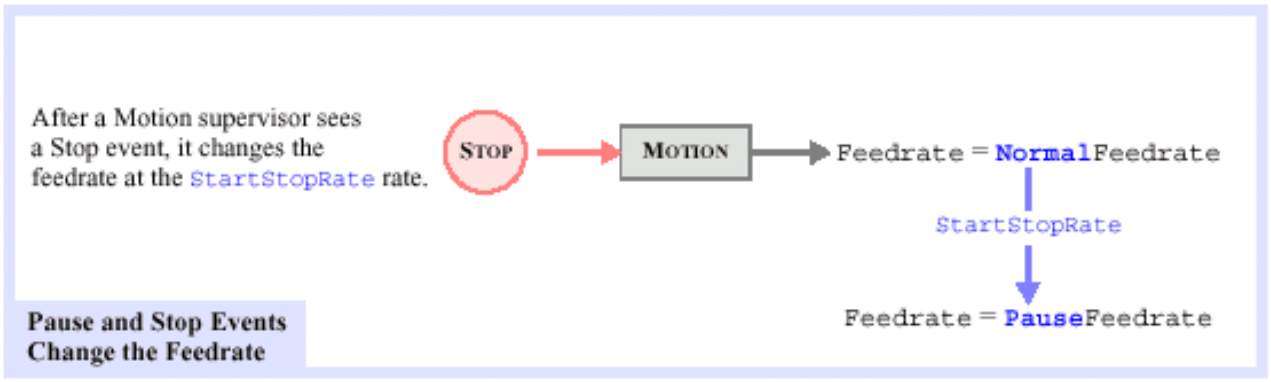
   For example, if you call **mpiAxisCommandPositionGet(...)** immediately after calling **mpiAxisOriginSet(...)** or **mpiAxisCommandPositionSet(...)**, **mpiAxisCommandPositionGet(...)** will probably return the old position values.

NEXT

TOP

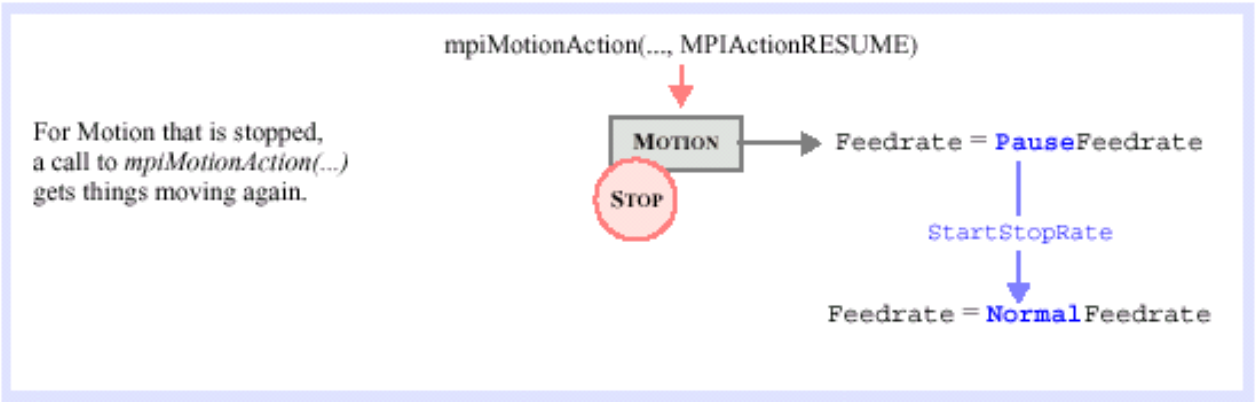
How STOP Events work

Resuming Motion after STOP Events



When a motion supervisor sees a STOP event (generated by anything in the domains of motion supervisor's axes), the Feedrate is changed from NormalFeedrate to PauseFeedrate (MS[n].PauseFeedrate in the firmware). To set the PauseFeedrate , use the method **mpiControlMemorySet(...)**.

How quickly the NormalFeedrate changes to PauseFeedrate is determined by the value of StartStopRate (which is set by motionConfig.decelTime.stop). Normally, PauseFeedrate is **0.0**, which causes the motion to stop.



When the motion is re-started (using **mpiMotionAction**(...,MPIActionRESUME) for STOP events), the Feedrate is changed back to the NormalFeedrate at the StartStopRate rate.

How Motion is Restarted after STOP Events

If Motion was stopped by	Then Motion resumes after
STOP Event	**mpiMotionAction**(...MPIActionRESUME) is called
PAUSE Event	**mpiMotionAction**(...MPIActionRESUME) is called
	the limit (bit) is cleared

TOP

Note that STOP events are latched by the firmware, while PAUSE events aren't. This means that if you configure a user limit to trigger a STOP event when an input (like Xcvr) is set (= 1), the Motion will go from NormalFeedrate to PauseFeedrate when that bit is set, and the Motion will stay at the PauseFeedrate until **mpiMotionAction**(...,MPIActionRESUME) is called.

If a limit is configured for a PAUSE event, the Motion will resume automatically after the bit is cleared (**mpiMotionAction**(...,MPIActionRESUME) is not needed).

NEXT

Compiling Program Sequencer Commands

An MPICommand will “compile” into one or more MEIXmpCommand{}s, each of which takes up a slot in the MEIXmpCommandBuffer{}. In general, an MPICommand will compile into a single MEIXmpCommand{}, but an MPICommand of type MPICommandTypeMOTION [with a motionCommand of MPICommandMotionSTART (i.e. **mpiMotionStart**(...))] will require several MEIXmpCommand{}s.

How many sequencer commands an MPI sequence command compiles to depends on the number of axes and number of positions in the move. The next table shows how many xmp sequencer commands it takes to do the equivalent of an **mpiMotionStart(...)**.

Number of Sequencer Commands to be equivalent to mpiMotionStart(...)

Number of sequencer commands required	To do this:
axisCount +	One MEIXmpCommand{} per axis to write the axis number to MEIXmpLinkBuffer{}.MSLink[].Axis[].AxisNumber
1+	1 + One MEIXmpCommand{} to write axisCount to MEIXmpLinkBuffer{}.MSLink[].Axes
1+	One MEIXmpCommand{} to write the MEIXmpMotionType{} to MS[].Mode.
(((axisCount*pointCount) + 3) / 4) +	One MEIXmpCommand{} for every four MEIXmpPoint{}s written to PointBuffer.Point[]
axisCount +	One MEIXmpCommand{} per axis to load the MEIXmpPoint(s)
1	One MEIXmpCommand{} to start the motion

NEXT

TOP

Running Multiple Applications with the XMP

There are two situations that often cause problems when you are executing multiple applications at the same time:

1. **If one application resets the XMP controller while other applications are busy using the XMP**, this is analogous to pulling the rug out beneath the feet of the other applications. The other applications are not aware that all states have been lost (controller reset), and thus these other applications get lost themselves.

   If you run two XMP applications at the same time (Motion Console and VM3, or Motion Console and your application), and if one application resets the board while the second application is busy accessing memory (like VM3), then the interface between the host and DSP can break down. To resolve this type of problem, you should close both applications, restart one application, and then reset the XMP.
2. If you modify the **default configuration** of objects on the XMP, and then run another application which wants a **different configuration** of the **same objects** (program sequencers, motion supervisors, axes, etc), and you don't manage this, you can get into trouble.You have to coordinate how simultaneously exectuing applications manage the XMP resources.
3. Simultaneously, using DLLs, applications, and Motion Console from **different MEI releases** can and will cause problems.

TOP

NEXT

Perform a Point-to-Point Coordinated Move

To generate a point-to-point coordinated move, you can use **mpiMotionStart(...)** with either a "trapezoidal" or "S-Curve" type motion. By scaling the Velocities, Accels, and Decels by the ratio of the distances, you can get a point-to-point coordinated move. The coordinated moves will all start and end at the same time.

It doesn't matter what the Jerk Percent parameter value is, because the calculated move time is the same for both trapezoidal and S-Curve moves. For example, specifying a vector Vel, Accel, Decel, and a distance X, Y, Z:

$$\text{total distance} = \sqrt{X^2 + Y^2 + Z^2}$$

then,

$$Vx = \left(\frac{X}{\text{total distance}}\right) \bullet \text{Vel} \qquad Vy = \left(\frac{Y}{\text{total distance}}\right) \bullet \text{Vel}$$

$$Ax = \left(\frac{X}{\text{total distance}}\right) \bullet \text{Accel} \qquad Ay = \left(\frac{Y}{\text{total distance}}\right) \bullet \text{Accel}$$

$$Vz = \left(\frac{Z}{\text{total distance}}\right) \bullet \text{Vel}$$

$$Az = \left(\frac{Z}{\text{total distance}}\right) \bullet \text{Accel}$$

NEXT

TOP

About the PID PIDOutputOffset Parameter

The **PIDoutput Offset** parameter is a constant value added to the PID calculation before sending the DAC value to the DAC.

	Reason	More
1	To compensate for the input offset errors in a “conventional” amplifier	For situations when the amplifier performs the commutation. Individual DAC offsets can also be used to zero out-of-spec amplifier offsets.
2	If a static DC torque (force) on the system is needed	Typically used with a gravity-loaded (vertical) axis that has a simple current amplifier controlling the motor
3	For testing the DAC	If the axis is aborted or all PID gains are zeroed, you can use the *PIDOutputOffset* parameter to set the DAC voltage directly. (3277 = 1Volt, 6553 = 2Volts, etc.) You can only use the *PIDOutputOffset* parameter if the controller is not configured for motor commutation. If the controller is configured for commutation, use the Commutation DAC Offset to test the DAC.
4	For using the DAC as a general purpose analog output	**In some cases**, the easiest way for the host to generate an analog signal is to set all PID gains to zero and use the *PIDOutputOffset* parameter to set the desired voltage level. Alternatively, you can use the Commutation DAC Offset for the same purpose, which has the advantage that the voltage levels can be individually set for the two DACs.

TOP

NEXT

How Motion Completion Events are Generated

There are three types of generated events related to motion completion: DONE, AT_VELOCITY, and IN_POSITION_COARSE. Motion completion for a position move is indicated by a DONE event, while motion completion for a velocity move is indicated by an AT_VELOCITY event.

AT_VELOCITY is not generated for position moves. For both position moves and velocity moves, the MPI uses a settling time window to determine when to generate a motion completed event (to indicate that motion has finished on an axis).

For position moves, the MPI uses the settling time window to determine when an axis has reached the final target location and is stable.

For velocity moves, the MPI uses the settling time window to determine when an axis has reached the final velocity. This works because during ringing, the velocity is highest when the position error is lowest. (Velocity and position error are inversely-related.)

Position and Velocity criteria for generating motion completion events

For a	A "motion completed" event is generated when	Criteria
Position move	ABS(position error) < inPosition.tolerance.positionFine	position
Position move	ABS(velocity error) < inPosition.tolerance.velocity	velocity
Velocity move	ABS(velocity error) < inPosition.tolerance.velocity	velocity

where:

position error = (command position - actual position)
velocity error = (command velocity - actual velocity)

For Position Moves

A DONE event is generated after a position move only after both the position and velocity criteria have been met for the period specified by the settling time. To determine if motion has completed, position moves use both position and velocity criteria.

The default (factory) value for inPosition.tolerance.velocity is 20,000,000 counts/sec. We included the ability to additionally use the velocity criteria for position moves because some users prefer to decide that an axis is “settled” only after the position error and actual velocity are both low enough.

For Velocity Moves

An AT_VELOCITY event is generated for a velocity move only after the velocity criteria has been met for the period specified by the *settling time*. To determine if motion has completed, velocity moves use only the velocity criteria (they don’t use the position criteria).

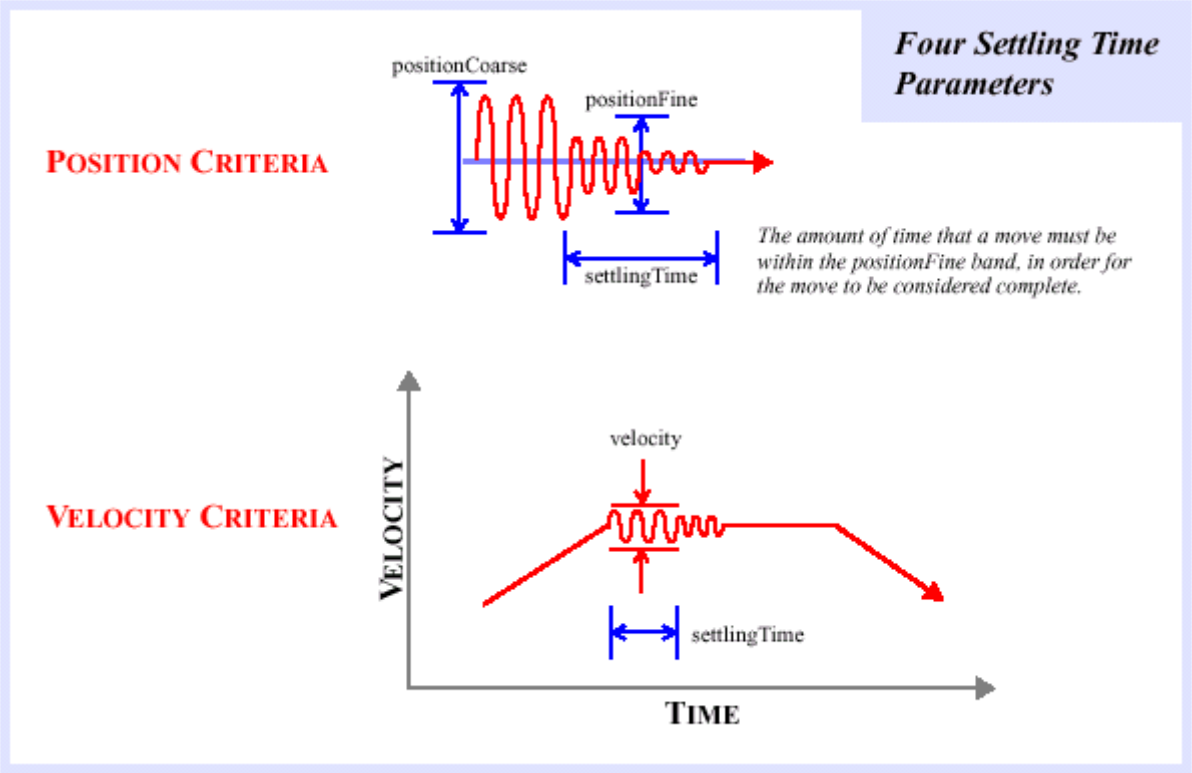
Note that the *motion completed* event generated this time is an AT_VELOCITY event rather than a DONE event. For velocity moves, DONE events will only occur after *mpiMotionAction*(..., MPIActionSTOP) calls.

You can also use the velocity criteria to determine when ringing has stopped after an open loop commutation move. To do this, the position criteria is effectively disabled by setting the position tolerance to a very large value (e.g., 1.0E10). After that, the DONE event following a move is determined entirely by the velocity tolerance and settling time, and, therefore, the event will be generated only after the ringing has stopped.

Be aware that setting inPosition.tolerance.velocity to 0 does not disable the velocity criteria. For interpolated scales, setting inPosition.tolerance.velocity to 0 makes it very difficult to meet the criteria of 0 counts of motion for settlingTime seconds.

Settling Time Parameters

Four Settling Time Parameters



For position moves, the generation of DONE events depends on three parameters:

MPIAxisConfig.inPosition.settlingTime
MPIAxisConfig.inPosition.positionFine
MPIAxisConfig.inPosition.velocity

DONE events are only generated when command = target or after STOP, ESTOP or ABORT events.

For velocity moves, the generation of AT_VELOCITY events depends on two parameters:

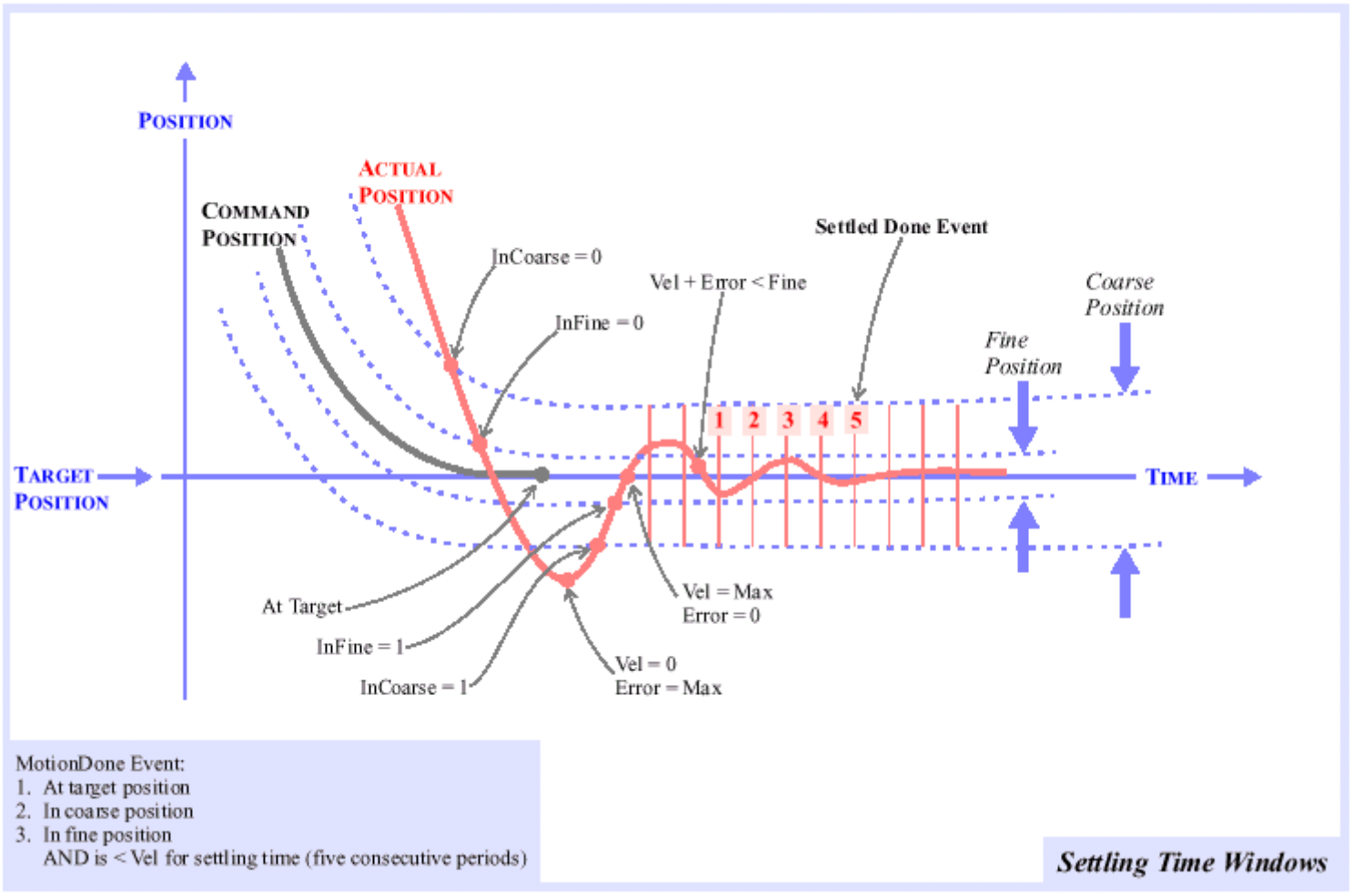
MPIAxisConfig.inPosition.settlingTime
MPIAxisConfig.inPosition.velocity

The **settlingTime** value specifies the amount of time (seconds) that a motion must be within the positionFine band, in order for the motion to be considered complete (finished).

The **positionFine** parameter is a tolerance (encoder counts), where we know a motion is complete if the motion settles within this tolerance.

The **velocity** parameter specifies a band around the target velocity for velocity moves.

A fourth parameter (MPIAxisConfig.inPosition.positionCoarse) specifies a band around the final position. Optionally, an interrupt can be triggered when the motion is in the *positionFine* band, but not necessarily complete (finished). In this case, the interrupt becomes an early warning of motion that is about to finish (i.e., mostly finished).



TOP

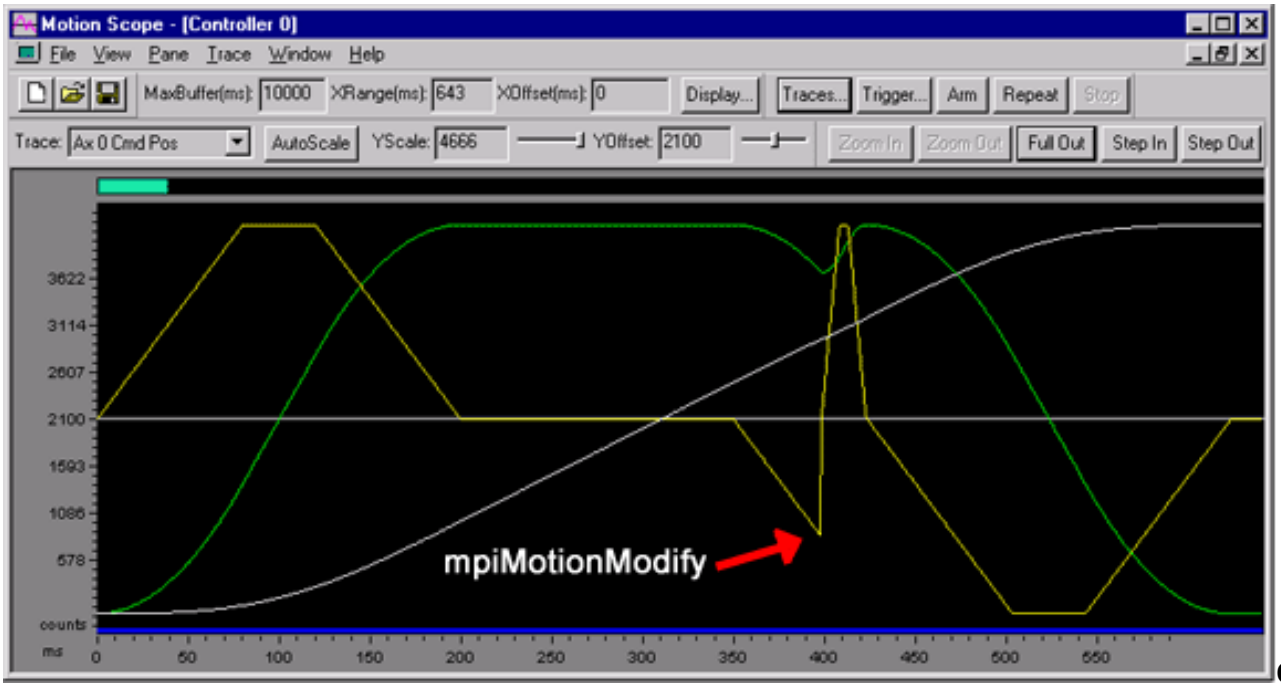
NEXT

S-Curve Jerk Algorithm and Attributes

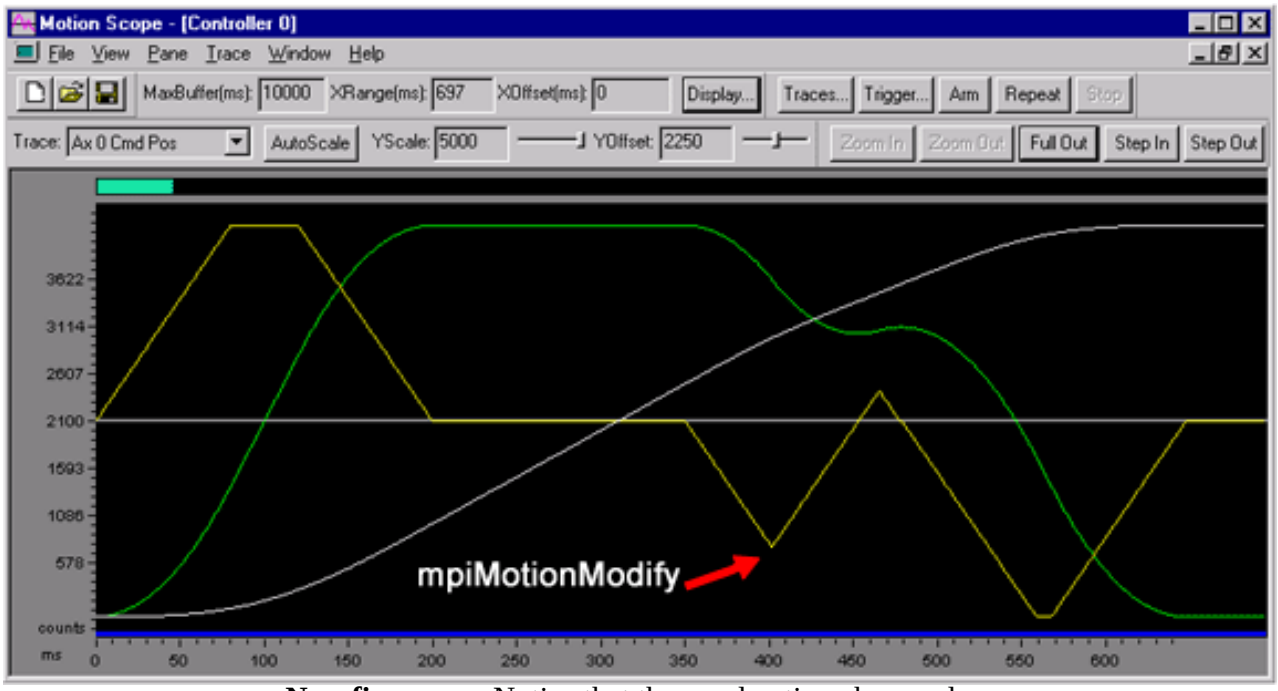
Algorithm | Attributes

Algorithm

A new move type, MPIMotionTypeS_CURVE_JERK, has been added to support a jerk-specified profile. This replaces the old jerkPercent algorithm. Two added features that the new S-Curve Jerk algorithm provide are the ability to call a motion modify at any time during a path move and the freedom to change jerk, acceleration, and maximum velocity independently. None of these values will be exceeded in the resulting motion. The new S-Curve Jerk algorithm will be ideal for making final adjustments to a move as it draws closer to its final target and for making smoother transitions from one motion to the next.



Old firmware: Notice that the acceleration is assumed to be zero and that there is a sudden change in the velocity as a result.



New firmware: Notice that the acceleration changes less abruptly and that the velocity profile is much smoother.

Safe parameters for jerk values should range from a minimum of amax* amax / vmax (amax is just reached when accelerating from 0 to vmax) and a maximum of amax / sample period (amax is reached in one sample period). In the new firmware, changes to the jerk will also change the time needed to complete a motion. For example, a large value of jerk will have a shorter time, but increase the “jerkiness” of the motion (see fig 1). Conversely, a small value of jerk will have a longer time, but a much smoother motion (see fig 2).

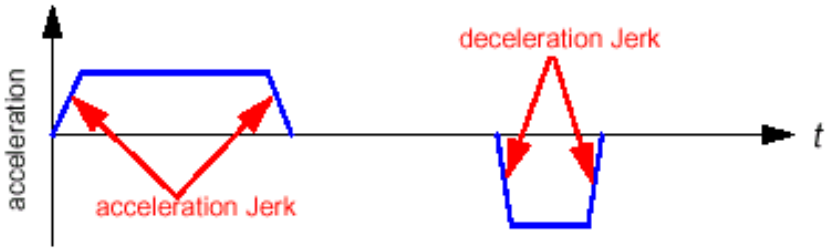


Fig 1. Acceleration profile with **larger** value of jerk.

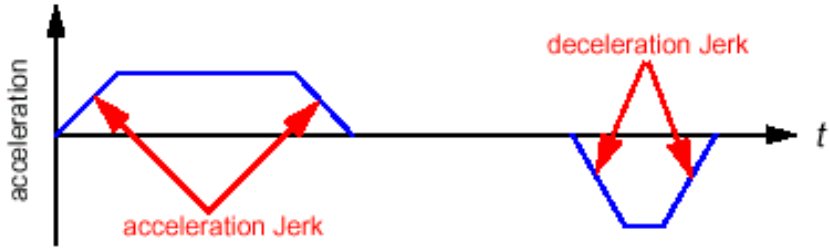


Fig 2. Acceleration profile with **smaller** value of jerk.

TOP

Two new parameters, accelerationJerk and decelerationJerk have been added to the MPITrajectory{…} structure. When they are non-zero, the acceleration profile uses the specified jerk and acceleration to ramp an axis(es) to constant velocity and then decelerate to a stop. If accelerationJerk or decelerationJerk is zero, an illegal parameter error is returned.

For the move type, MPIMotionTypeS_CURVE, the MPI calculates an appropriate jerk value based on the specified velocity, acceleration, and jerkPercent. The jerk value is computed according to the following formula:

jerk = amax * amax / (vmax * jp * sp (1 - jp * sp))

jp = jerkPercent
sp = sample period

If jerkPercent is zero, the jerk value is computed so that the maximum acceleration is reached in one sample period. With the previous S-Curve algorithm, the time for a move would not change as jerkPercent value was varied. This is also true for this S-Curve algorithm, as long as the move reaches maximum velocity. In short moves, where maximum velocity is not reached, setting jerkPercent to be small will result in a quicker move than if you were to set jerkPercent to be large.

WARNING! The same jerkPercent values may cause different profiles than the previous S-Curve algorithm.

Attributes

The new S_Curve algorithm behaves similarly to the previous algorithm, except for its attributes.

MPIMotionAttrMaskDELAY can now be used with any start motion, but never with motion modify.

MPIMotionAttrMaskAPPEND can be used with any motion, as long as it is not preceded by a motion that had a final velocity.

MEIMotionAttrMaskNO_REVERSAL returns a MPIMotionMessagePROFILE_ERROR if the given specifications would result in a move with a reversal init, thereby preventing the move from being executed.

MPIMotionTypeTRAPEZOIDAL, MPIMotionTypeS_CURVE, and MPIMotionTypeS_CURVE_JERK

MPIMotionAttrMaskRELATIVE, when used with MPIMotionTypeTRAPEZOIDAL, MPIMotionTypeS_CURVE and MPIMotionTypeS_CURVE_JERK means that the final position is relative to the beginning position of the motion.

MEIMotionAttrMaskFINAL_VEL can be used with MPIMotionTypeTRAPEZOIDAL, MPIMotionTypeS_CURVE and MPIMotionTypeS_CURVE_JERK, but should be used with caution as it may not be possible for the controller to compute a trajectory to meet these criteria, which would cause a MPIMotionMessagePROFILE_ERROR to be returned, and the move to be ignored.

Multi-Axis Motion

Neither MPIMotionAttrMaskSYNC_START nor MPIMotionAttrMaskSYNC_END
If neither MPIMotionAttrMaskSYNC_START nor MPIMotionAttrMaskSYNC_END are specified, a single MPITrajectory{...} may be specified for the resultant motion of multiple axes on one motion supervisor. The motion of each axis will be synchronized with the others on the motion supervisor. The maximum velocity, acceleration, deceleration, and jerk values of the first MPITrajectory structure will be used for the global vector parameters. It will ignore any other values supplied. This cannot be used with MEIMotionAttrMaskFINAL_VEL.

MPIMotionAttrMaskSYNC_START or MPIMotionAttrMaskSYNC_END but not both

If MPIMotionAttrMaskSYNC_START or MPIMotionAttrMaskSYNC_END (but not both) is specified, each axis will move as fast as possible and either start together, or stop together. If motion is point-to-point and more than one axis on the motion supervisor has a final velocity, MPIMotionAttrMaskSYNC_START or MPIMotionAttrMaskSYNC_END must be used. MPIMotionAttrMaskSYNC_END cannot be used with motion modify.

Both MPIMotionAttrMaskSYNC_START and MPIMotionAttrMaskSYNC_END
With MPIMotionAttrMaskSYNC_START and MPIMotionAttrMaskSYNC_END, the motion for each axis will be scaled so that the motion of all axes will end at approximately the same time. The time for this motion is based on the time for the longest motion, so that the limits are not exceeded. The axes will be scaled to start and stop together, but the scaling may not be exact. Both MPIMotionAttrMaskSYNC_START and MPIMotionAttrMaskSYNC_END cannot be used together with MEIMotionAttrMaskFINAL_VEL.

MPIMotionTypeVELOCITY

MPIMotionTypeVELOCITY moves allow a final velocity to be specified without a final point.

MPIMotionAttrMaskSYNC_START and/or MPIMotionAttrMaskSYNC_END
Neither is supported for this motion type. MPIMotionAttrMaskSYNC_END cannot be used with motion modify.

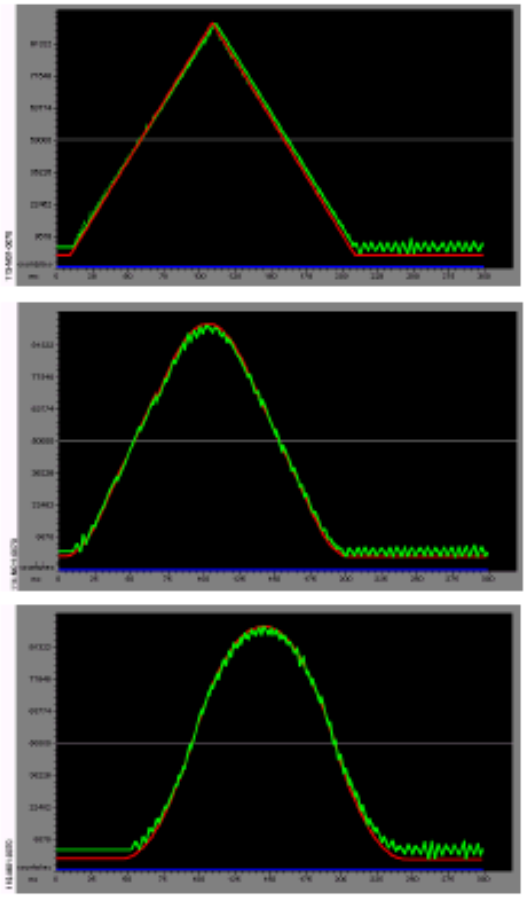
MEIMotionAttrMaskFINAL_VEL
MEIMotionAttrMaskFINAL_VEL is not supported for this motion type.

MPIMotionAttrMaskRELATIVE
MPIMotionAttrMaskRELATIVE, when used with MPIMotionTypeVELOCITY or MPIMotionTypeVELOCITY_JERK, means that the final velocity is relative to the velocity at the start of the motion.

S-Curve Motion Profiles: Command Acceleration vs. Peak Acceleration

The XMP's DSP uses a formula to convert the "commanded acceleration" for an S-Curve move to a "peak acceleration." The DSP calculates the trapezoidal profile first, then adjusts the accel/decel portions of the move based on the "Jerk Percent" parameter, while keeping the move time at a constant value. For example, a 0% jerk would look like a trapezoidal velocity profile (no jerk portion), and a 100% jerk would have no constant accel/decel (all jerks). See the table below.

Changes in Jerk Percent parameter change the acceleration-deceleration curve characteristics



Jerk Percent = 0

Jerk Percent = 50

Jerk Percent = 100

The conversion formula is:

$$\text{max\_accel} = \frac{\text{accel}}{1 - (\text{jerk\_percent} \bullet .005)}$$

where,

max_accel is the maximum acceleration (or decel) for a point to point profile accel is the specified acceleration (or decel) from the application code *jerk_percent* is the specified jerk percentage (0 to 100.0) from the application code.

NEXT

TOP

Default XMP-Series Controller Configuration

Overview

The XMP-Series controller's Flash memory is now pre-loaded at the factory with base firmware. This firmware contains a minimal amount of code to boot the DSP and allows the MPI to identify the XMP-Series controller. To operate the controller you will need to download the binary code included in this MPI software distribution. Please see the sections below for a complete description about software binary management and instructions to download firmware to your controller.

TOP

Introduction

System designers need to give careful consideration to the software configuration management procedures for their machines. To ensure machine consistency and quality, a process for software installation, configuration, version control, and verification must be implemented. Addressing this issue early in the development cycle will significantly reduce confusion and mistakes. Failure to implement some basic procedures can cause unknown machine configurations, costly field repairs/upgrades, mysterious intermittent problems, broken equipment, and possible injury.

The MPI and XMP-Series controllers contain several features to make configuration management easy. Please take the time to understand and implement these features before you begin development.

TOP

Software Components

The MEI software distribution contains several software components, which need to be loaded onto your machine and controller. The MPI DLL, header files, import libraries, device driver, utility programs, controller binaries, sample code, etc. These are all loaded onto your hard drive by the InstallShield distribution. Additionally, the controller contains on-board flash memory to store DSP code, FPGA code, and configuration information. You will also need to load the appropriate DSP (.bin) and FPGA (.fpg) code into your controller's flash memory. The code is loaded into the DSP and FPGA(s) during power-on or when the controller is reset.

TOP

Version Control

Each software component has its own version number. These components have been tested together at the factory for interoperability.

The software version numbers have the following format:

Motion Console	NN.NN.nn	Major, Medium, minor
Motion Scope	NN.NN.nn	Major, Medium, minor
MPI DLL	YYYYMMDD.b...r	Year, Month, Day, branch,...rev
XMP Firmware	NNNnn	Major, minor
FPGA Code	RRR	Revision

The software has automated compatibility checking. If there is a compatibility problem between software components, an error code will be returned.

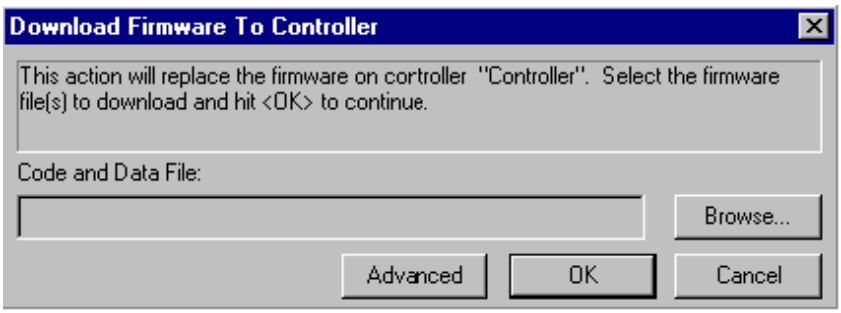
If an application (Motion Console and Motion Scope are applications too) and MPI DLL are NOT compatible, the error message "Control: application not compatible with MPI DLL" will be returned. To correct this problem, you can recompile your application with the appropriate MPI import library OR install the proper MPI DLL.

If the MPI DLL and firmware versions are NOT compatible, the error message, ***Control: firmware version mismatch*** will be returned. The user or application must download the appropriate firmware to correct the problem. The DLL and firmware versions can be determined with the version.exe utility, Motion Console, or application code.

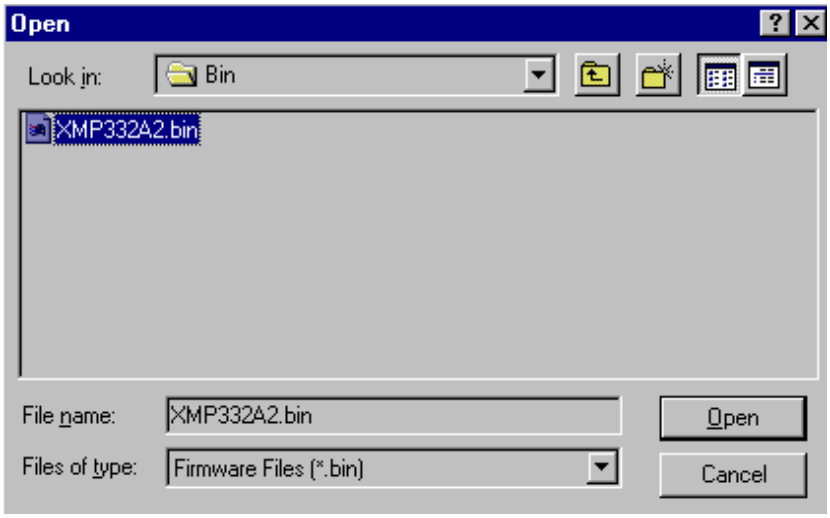
If the controller flash memory has NOT been configured, the error message **Control: no firmware found (factory default)** will be returned. To correct this problem, the user or application must download the appropriate XMP firmware and FPGA images. Firmware and FPGA images can be downloaded from Motion Console, flash.exe, or application code.

For example, if Motion Console detects that the controller is in the factory default configuration (no firmware) or if the MPI DLL is not compatible with the firmware, it will prompt the user to download firmware:

Click **Yes**. Motion Console will then prompt you for the firmware file:

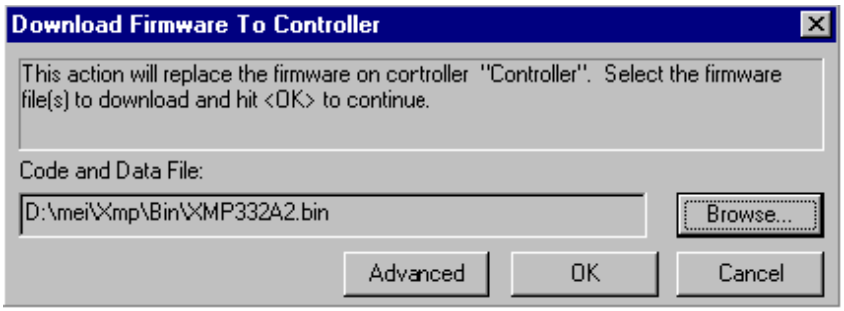


Click **Browse** and select the appropriate firmware file (.bin). The firmware is stored in the mei\xmp\bin directory (by default):



TOP

Select the proper file and click **Open**.



Automated Software Configuration

During the development stage of your machine, several different versions of MPI DLLs and/or XMP firmware might be used. You may want to upgrade to new releases in order to take advantage of new features. MEI may provide custom features or bug patches for previous releases.

During the production stage of your machine, you will want to guarantee software configuration consistency. MEI recommends using the InstallShield release package, a third party installer program, or batch scripts to install your application code and MEI's software onto your machine.

You will also want to guarantee that the controller's flash memory configuration is consistent. To load the flash, you could automatically download firmware (.bin) and FPGA (.fpg) code during software installation OR during your application initialization. For example, the flash.exe program could be executed from an installer or batch script during software installation. Included in the XMP\Apps directory, is a sample program called **initFlsh.c** that demonstrates how to read the controller's firmware/FPGA versions, check if they match the desired configuration, and download the correct versions (if needed).

Also, you can create your own custom firmware file by saving configurations to flash, and uploading the firmware file to your hard drive. The firmware contains a **userVersion** field, so you can keep track of your custom configured files. Using this technique, the firmware can be configured with Motion Console, uploaded to a file (myfirm.bin) and downloaded to future machines using Motion Console, flash.exe or your application.

TOP

FAQs

Why do I need to download firmware to my controller?
Only the machine developer knows which firmware version and configuration works with their application. By downloading firmware directly, you have complete control over your development and release versions.

Why can't MEI download firmware to my controller at the factory?
MEI can download your firmware, but it is expensive and causes several problems:

1. You would need a custom part number for each controller with a different firmware image. Even if you use the same controller hardware in several machines, each version would need to be ordered, purchased, tracked, and stocked separately.

2. Changing a firmware image (version or configuration) would require a new part number. This causes transition problems between "old" and "new" parts.

3. Repairs and replacements are much more complicated.

4. Field upgrades are not possible. Controllers must be returned to the factory to receive new firmware and a new part number.

What if I need to upgrade software in the field?
If you configure the flash memory as part of your application or installation, then it is very easy to upgrade software and/or firmware in the field. If you do not, then you'll need to manually update the flash memory.

Can I modify the FPGA (.fpg) files?
TOP
No. These files are binary and do not contain any configuration data.

NEXT

Axis Tolerances and Related Events

MEIXmpStatusIN_FINE_POSITION | MEIXmpStatusIN_COARSE_POSITION
MEIXmpStatusAT_TARGET | MEIXmpStatusAT_VELOCITY
MEIXmpStatusDONE

MEIXmpStatusIN_FINE_POSITION

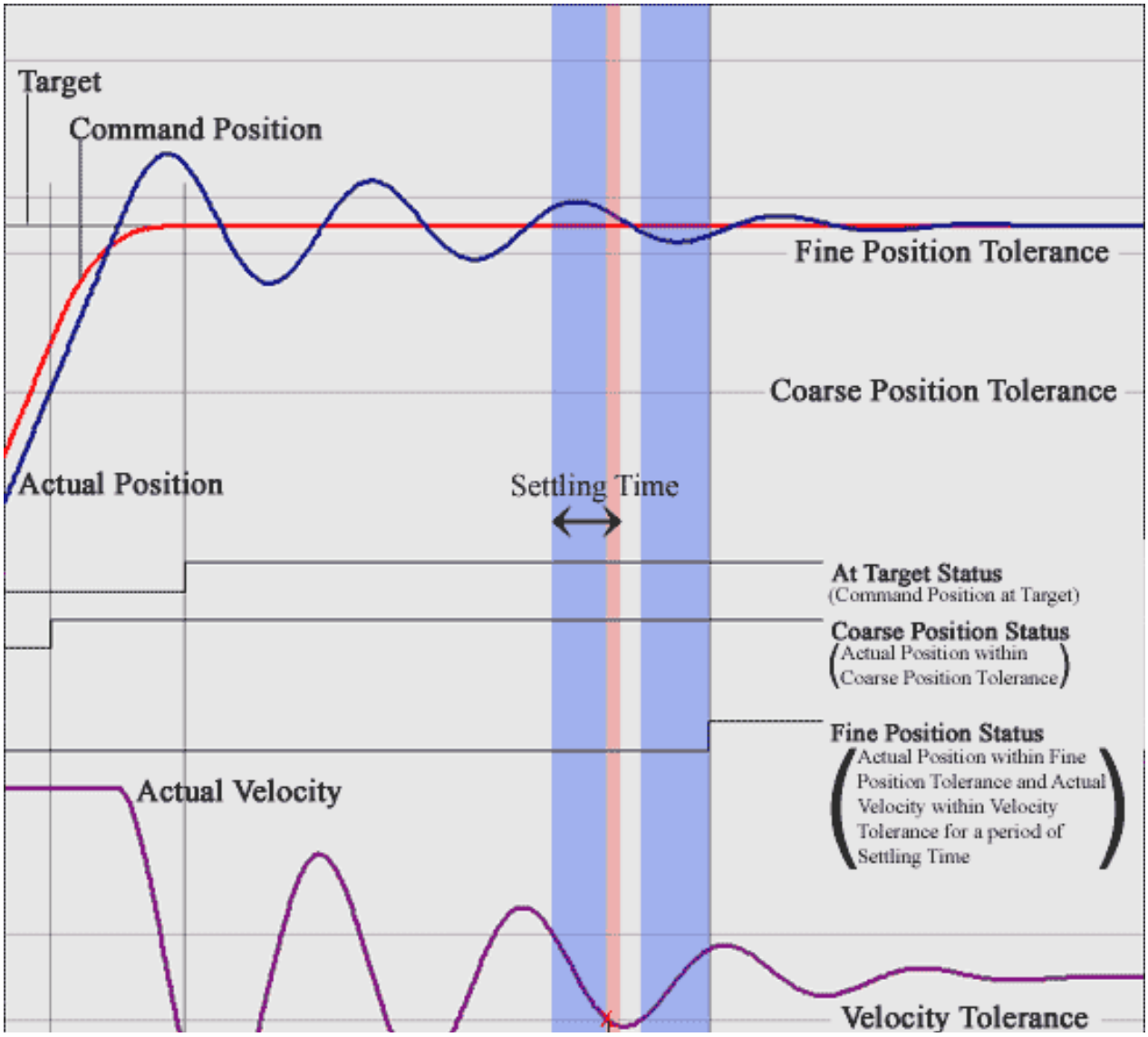
During Normal Conditions (no Stop, E-stop, or Abort)

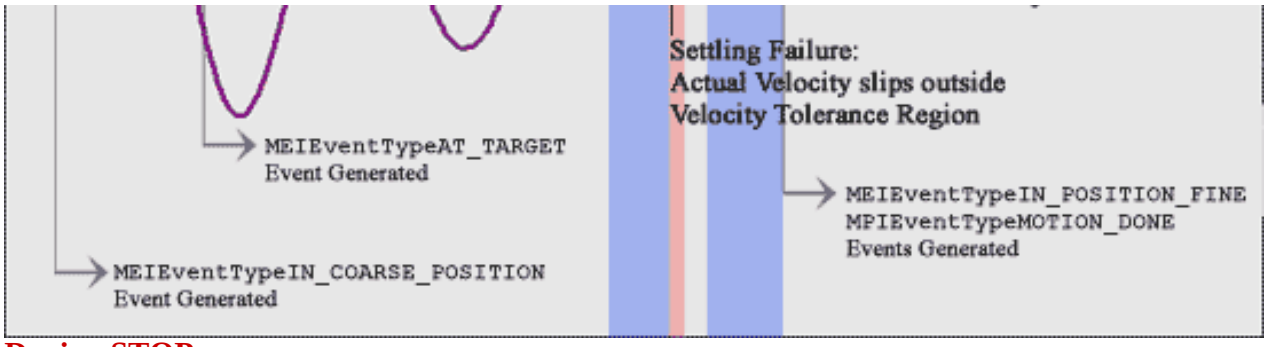
Between mpiMotionStart(...) and AT_TARGET, IN_FINE_POSITION is **FALSE**.

After AT_TARGET, the evaluation of the settling criteria begins. When the settling criteria has been satisfied, IN_FINE_POSITION is **TRUE**.

The settling criteria are:

1. The absolute value of the position error is less than or equal to the fine position tolerance.
2. The absolute value of the velocity is less than or equal to the velocity tolerance.
3. Both of the above criteria has been satisfied for the settling time. Whenever either criteria 1 or 2 is not satisfied, IN_FINE_POSITION is cleared and the settling timer is reset.





During STOP

If settleOnStop is **FALSE**, IN_FINE_POSITION is **FALSE**.

If settleOnStop is **TRUE** and Feedrate != 0, IN_FINE_POSITION is **FALSE**.

If settleOnStop is **TRUE** and Feedrate = = 0, evaluation of the settling criteria begins.

After the **settling criteria** has been satisfied, IN_FINE_POSITION is **TRUE**.

During E-STOP

If settleOnEstop is **FALSE**, IN_FINE_POSITION is **FALSE**.

If settleOnEstop is **TRUE** and Feedrate != 0, IN_FINE_POSITION is **FALSE**.

If settleOnEstop is **TRUE** and Feedrate = = 0, evaluation of the settling criteria begins.

After the settling criteria has been satisfied, IN_FINE_POSITION is **TRUE**.

If both E-STOP and STOPoccur simultaneously

If both settleOnStop and settleOnEstop are **FALSE**, IN_FINE_POSITION is **FALSE**.

If either settleOnStop or settleOnEstop are **TRUE** and Feedrate != 0, IN_FINE_POSITION is **FALSE.**

If either settleOnStop or settleOnEstop are **TRUE** and Feedrate = = 0, evaluation of the settling criteria begins.

After the settling criteria has been satisfied IN_FINE_POSITION is **TRUE**.

Note	If STOP and E-STOP occur simultaneously, the XMP applies priority to the E-STOP. Both status bits will be true, but the motion will decelerate at the E-STOP rate.

During ABORT

IN_FINE_POSITION is **FALSE**.

If both E-STOP and ABORT occur simultaneously

During an E-Stop_Abort condition, IN_FINE_POSITION is **FALSE** (even if settleOnEstop is **TRUE**).

After mpiMotionAction(..., MPIActionRESET)

If IN_FINE_POSITION status is **TRUE** before RESET, the status will be unaffected (no event generated).

If IN_FINE_POSITION status is **FALSE**, the effect of RESET depends on the cause: If IN_FINE_POSITION is **FALSE** because:

- the position error exceeds the tolerance, IN_FINE_POSITION will remain **FALSE** (no event generated).

- the position error is within the tolerance but STOP is **TRUE** and settleOnStop is **FALSE**, IN_FINE_POSITION will become **TRUE** at settlingTime after the RESET call (event will be generated).
- the position error is within the tolerance but E_STOP is **TRUE** and settleOnEstop is FALSE, IN_FINE_POSITION will become **TRUE** at settlingTime after the RESET (event will be generated).
- the position error has been within the tolerance for settlingTime but ABORT is **TRUE**, IN_FINE_POSITION will become **TRUE** immediately after the RESET (event will be generated).

TOP

MEIXmpStatusIN_COARSE_POSITION

During Normal Conditions (no Stop, E-stop, or Abort)

The IN_COARSE_POSITION status bit can be set only during motion (defined from the return of mpiMotionStart(...) until MEIXmpStatusDONE is set for the following motion types:

- S-Curve (final velocity = 0)
- Trap
- PT (after final point has been specified)
- PVT (after final point has been specified)

For these motion types IN_COARSE_POSITION is **TRUE** whenever the absolute distance to the target (measured from the actual position to the final position) is less than or equal to the coarse position tolerance. If the target position is changed by mpiMotionModify(...) calls, IN_COARSE_POSITION may be set and cleared more than once (causing multiple events) during a single move.

IN_COARSE_POSITION is always **FALSE** for velocity moves (or moves using the FINAL_VEL attribute where the final velocity is non-zero).

During STOP, E-STOP, or ABORT

IN_COARSE_POSITION is **FALSE**.

After mpiMotionAction(..., MPIActionRESET)

IN_COARSE_POSITION is **FALSE**.

TOP

MEIXmpStatusAT_TARGET

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:

During Normal Conditions (no Stop, E-stop, or Abort)

The AT_TARGET status bit can be set only during motion (defined from the return of mpiMotionStart(...) until MEIXmpStatusDONE is set for the following motion types:

- S-Curve (final velocity = 0)
- Trap
- PT (after final point has been specified)
- PVT (after final point has been specified)

For these motion types AT_TARGET is **TRUE** whenever the command position is equal to to the target (final) position. If the target position is changed by mpiMotionModify(...) calls, AT_TARGET may be set and cleared more than once (causing multiple events) during a single move.

AT_TARGET is always **FALSE** for velocity moves (or moves using the FINAL_VEL attribute where the final velocity is non-zero).

During STOP, E-STOP, or ABORT

AT_TARGET is always **FALSE**.

After mpiMotionAction(..., MPIActionRESET)

AT_TARGET is always **FALSE**.

TOP

MEIXmpStatusAT_VELOCITY

During Normal Conditions (no Stop, E-stop, or Abort)

The AT_VELOCITY status bit can only be set during the constant velocity portions for the following move types:

- Velocity
- S-Curve (final velocity = 0)

The velocity settling criteria are continuously evaluated during the constant velocity portion of the motion. Once the criteria has been satisfied, the AT_VELOCITY bit is set to **TRUE**. AT_VELOCITY is **FALSE** during any non-constant velocity portions of the motion.

The velocity settling criteria are:

1. The absolute value of the velocity error is less than or equal to the velocity tolerance.
2. Criteria 1 has been satisfied for the settling time. Whenever Criteria 1 is not satisfied, AT_VELOCITY is set to **FALSE** and the settling timer is reset.

After Reset, Stop, E-stop, or Abort

After a Reset, Stop, E-Stop, or Abort, AT_VELOCITY is always **FALSE**.

TOP

MEIXmpStatusDONE

This status bit is set to **TRUE** whenever IN_FINE_POSITION is **TRUE**. It can be cleared only by mpiMotionStart(...).

After mpiMotionAction(..., MPIActionRESET)
DONE is always **TRUE**.

Return to Software's Main Menu