

# NI-Motion™ Help

July 2007, 372134B-01

This help file provides information about the NI-Motion driver software, including background, configuration, and programming information. The purpose of this help file is to provide a basic understanding of the NI-Motion driver software, and provide programming steps and examples to help you develop NI-Motion applications.

This help file is intended for experienced LabVIEW, C/C++, or other developers. Code instructions and examples assume a working knowledge of the given programming language. This help file also assumes a general knowledge of motion control terminology and development requirements.

This help file pertains to all NI motion controllers that use the NI-Motion driver software.

For more information about this help file, refer to the following topics:

**Using Help** 

**Related Documentation** 

**Glossary** 

**Important Information** 

**Technical Support and Professional Services** 

To comment on National Instruments documentation, refer to the <u>National</u> <u>Instruments Web site</u>.

© 2007 National Instruments Corporation. All rights reserved.

## **Related Documentation**

Most motion control manuals also are available as PDFs. You must have Adobe Reader with Search and Accessibility 5.0.5 or later installed to view the PDFs. Refer to the <u>Adobe Systems Incorporated Web site</u> at www.adobe.com to download Adobe Reader. Refer to the <u>National Instruments Product Manuals Library</u> at ni.com/manuals for updated documentation resources.

The following documents contain information that you may find helpful as you use this help file:

- <u>Getting Started with NI-Motion for NI 73xx Motion Controllers</u>— Provides installation instructions and general information about the NI-Motion product.
- Getting Started: NI SoftMotion Controller for Copley Controls CANopen Drives—Contains information about getting started with the NI SoftMotion Controller for CANopen.
- NI-Motion VI Help—Contains LabVIEW VI reference files and provides details about each VI, including VI descriptions, lists of control and input terminals, usage, illustrations, and error codes.
- NI-Motion Function Help—Contains function reference files for C and Visual Basic and provides details about each function, including a description of the function, a list of the function parameters, illustrations, and error codes.
- Measurement & Automation Explorer Help for Motion—Provides information about using MAX to configure the NI 73xx motion controller as well as some advanced conceptual information about topics such as Bode analysis and control loop parameters.
- NI 73xx user manuals—Describes the electrical and mechanical aspects of the NI 73xx motion controller, and contains information about installing and operating the device.
- NI-Motion Readme—Contains system requirements, installation instructions, descriptions of any changes made to the software, information about new features in the release, and information about late-breaking known issues that are not documented in other NI-Motion documents.
- NI Developer Zone (NIDZ)—Visit the <u>NI Developer Zone</u>, at ni.com/zone, for example programs, tutorials, technical

presentations, the Instrument Driver Network, product advisors, application notes, and a discussion forum where you can share ideas, questions, and source code with motion developers around the world.

 Motion Hardware Advisor—Visit the <u>National Instruments Motion</u> <u>Hardware Advisor</u> at ni.com/devzone/advisors/motion/ to select motors and stages appropriate to the motion control application.

In addition to the NI Developer Zone, you can find NI-Motion C/C++ and Visual Basic programming examples in the NI-  $\,$ 

Motion\FlexMotion\Examples folder where you installed NI-Motion. The default directory is Program Files\National Instruments\NI-Motion for Windows XP/2000 and the 32-bit version of Windows Vista and Program Files (x86)\National Instruments\NI-Motion for the 64-bit version of Windows Vista.

You can find LabVIEW example programs under examples\Motion in the directory where you installed LabVIEW. You can find LabWindows™/CVI™ examples under samples\Motion in the directory where you installed LabWindows/CVI.

You can find the NI-Motion C/C++ and LabVIEW example code referenced in this manual in the NI-Motion\Documentation\Examples\NI-Motion User Manual folder where you installed NI-Motion.

# **Using Help**

**Conventions** 

**Navigating Help** 

Searching Help

Printing Help File Topics

## **Conventions**

bold

This help file uses the following formatting and typographical conventions:

< > Angle brackets that contain numbers separated by an ellipsis represent a range of values associated with a bit or signal name—for example, AO <0..3>.

[] Square brackets enclose optional items—for example, [response].

The » symbol leads you through nested menu items and dialog box options to a final action. The sequence File»Page Setup»Options directs you to pull down the File menu, select the Page Setup item, and select Options from the last dialog box.

This icon denotes a tip, which alerts you to advisory information.

This icon denotes a note, which alerts you to important information.

This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.

Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

dark red Text in this color denotes a caution.

<u>green</u> Underlined text in this color denotes a link to a help topic, help file, or Web address.

italic Italic text denotes variables, emphasis, cross-references, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.

monospace Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories,

programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.

bold

monospace Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

italic

monospace Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

# **Navigating Help (Windows Only)**

To navigate this help file, use the **Contents**, **Index**, and **Search** tabs to the left of this window or use the following toolbar buttons located above the tabs:

- **Hide**—Hides the navigation pane from view.
- Locate—Locates the currently displayed topic in the Contents tab, allowing you to view related topics.
- **Back**—Displays the previously viewed topic.
- **Forward**—Displays the topic you viewed before clicking the **Back** button.
- **Options**—Displays a list of commands and viewing options for the help file.

# **Searching Help (Windows Only)**

Use the **Search** tab to the left of this window to locate content in this help file. If you want to search for words in a certain order, such as "related documentation," add quotation marks around the search words as shown in the example. Searching for terms on the **Search** tab allows you to quickly locate specific information and information in topics that are not included on the **Contents** tab.

#### **Wildcards**

You also can search using asterisk (\*) or question mark (?) wildcards. Use the asterisk wildcard to return topics that contain a certain string. For example, a search for "prog\*" lists topics that contain the words "program," "programmatically," "progress," and so on.

Use the question mark wildcard as a substitute for a single character in a search term. For example, "?ext" lists topics that contain the words "next," "text," and so on.



**Note** Wildcard searching will not work on Simplified Chinese, Traditional Chinese, Japanese, and Korean systems.

## **Nested Expressions**

Use nested expressions to combine searches to further refine a search. You can use Boolean expressions and wildcards in a nested expression. For example, "example AND (program OR VI)" lists topics that contain "example program" or "example VI." You cannot nest expressions more than five levels.

## **Boolean Expressions**

Click the **•** button to add Boolean expressions to a search. The following Boolean operators are available:

- AND (default)—Returns topics that contain both search terms. You
  do not need to specify this operator unless you are using nested
  expressions.
- **OR**—Returns topics that contain either the first or second term.
- **NOT**—Returns topics that contain the first term without the second term.
- **NEAR**—Returns topics that contain both terms within eight words of each other.

## **Search Options**

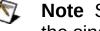
Use the following checkboxes on the **Search** tab to customize a search:

- **Search previous results**—Narrows the results from a search that returned too many topics. You must remove the checkmark from this checkbox to search all topics.
- Match similar words—Broadens a search to return topics that contain words similar to the search terms. For example, a search for "program" lists topics that include the words "programs," "programming," and so on.
- **Search titles only**—Searches only in the titles of topics.

# **Printing Help File Topics (Windows Only)**

Complete the following steps to print an entire book from the **Contents** tab:

- 1. Right-click the book.
- 2. Select **Print** from the shortcut menu to display the **Print Topics** dialog box.
- 3. Select the **Print the selected heading and all subtopics** option.



Note Select Print the selected topic if you want to print the single topic you have selected in the **Contents** tab.

4. Click the **OK** button.

# **Printing PDF Documents**

This help file may contain links to PDF documents. To print PDF documents, click the print button located on the Adobe Acrobat Viewer toolbar.

# Introduction

This section provides basic information about the NI-Motion driver software, motion control setup, and specific task-based instructions for creating motion control applications using the LabVIEW and C/C++ application development environments.

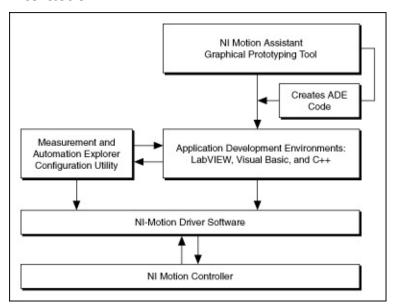
## **About NI-Motion**

NI-Motion is the driver software for National Instruments 73xx motion controllers and the NI SoftMotion Controller. You can use NI-Motion to create motion control applications using the included library of LabVIEW VIs and C/C++ functions.

National Instruments also offers the NI Motion Assistant and NI-Motion development tools for Visual Basic.

#### **NI-Motion Architecture**

The NI-Motion driver software architecture is based on the interaction between the NI motion controllers and a host computer. This interaction includes the hardware and software interface and the physical and functional architecture of the NI motion controllers. The following block diagram demonstrates the NI motion control hardware and software interaction.



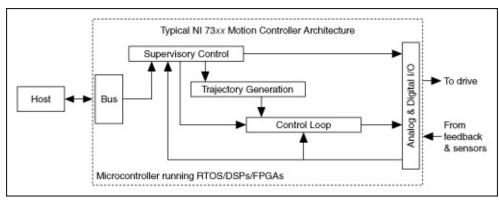


**Note** The NI Motion Controller block is not applicable to the NI SoftMotion Controller.

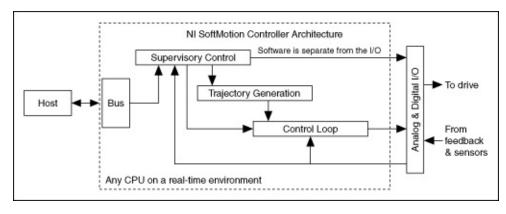
## **NI Motion Controller Architecture**

Functionally, the architecture of the NI 73xx motion controllers and the NI SoftMotion Controller is generally divided into four components: supervisory control, trajectory generator, control loop, and motion I/O. For the NI SoftMotion Controller, the motion I/O component is separate from the controller. The following figures demonstrate how the components of the 73xx and NI SoftMotion Controller interact.

## Typical NI 73xx Motion Controller Functional Architecture



#### **NI SoftMotion Controller Functional Architecture**

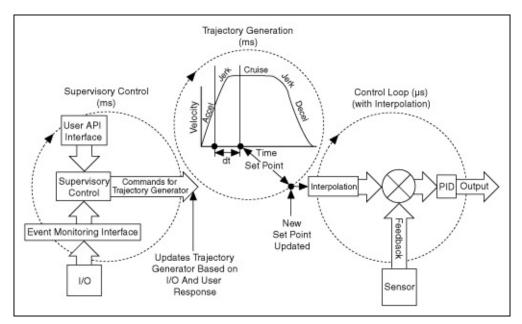


# **NI Motion Controller Functional Components**

The following list describes how each component of the 73xx controllers and the NI SoftMotion Controller functions:

- **Supervisory control**—Performs all the command sequencing and coordination required to carry out the specified operation
  - System initialization, which includes homing to a zero position
  - Event handling, which includes electronic gearing, triggering outputs based on position, updating profiles based on user defined events, and so on
  - Fault Detection, which includes stopping moves on a limit switch encounter, safe system reaction to emergency stop or drive faults, watchdog, and so on
- Trajectory generator—Provides path planning based on the profile specified by the user
- Control loop—Performs fast, closed-loop control with simultaneous position, velocity, and trajectory maintenance on one or more axes

The control loop handles closing the position/velocity loop based on feedback, and it defines the response and stability of the system. For stepper systems, the control loop is replaced with a step generation component. To enable the control loop to execute faster than the trajectory generation, an interpolation component, or spline engine, the control loop interpolates between setpoints calculated by the trajectory generator. The following figure illustrates the functional architecture of NI motion controllers.



• Motion I/O—Analog and digital I/O that sends and receives signals from the rest of the motion control system. Typically, the analog output is used as a command signal for the drive, and the digital I/O is used for quadrature encoder signals as feedback from the motor. The motion I/O performs position breakpoint and high speed capture. Also, the supervisory control uses the motion I/O to achieve certain required functionality, such as reacting to limit switches and creating the movement modes needed to initialize the system.

## NI 73xx Controller Architecture

NI 73xx controllers use a dual-processor architecture. The two processors, a central processing unit (CPU) and a digital signal processor (DSP), form the backbone of the NI motion controller.

The controller CPU is a 32-bit micro-controller running an embedded real time, multitasking operating system. This CPU offers the performance and determinism needed to solve most complex motion applications. The CPU performs command execution, host synchronization, I/O reaction, and system supervision.

The DSP has the primary responsibility of fast closed-loop control with simultaneous position, velocity, and trajectory maintenance on multiple axes. The DSP also closes the position and velocity loops, and directly commands the torque to the drive or amplifier.

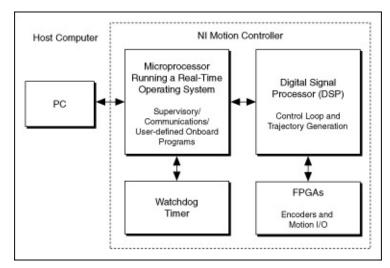
Motion I/O occurs in hardware on an FPGA and consists of limit/home switch detection, position breakpoint, and high-speed capture. This ensures very low latencies in the range of hundreds of nanoseconds for breakpoints and high-speed captures. Refer to the <a href="Synchronization">Synchronization</a> section for information about breakpoints and high-speed capture.

The motion controller processor is monitored by a watchdog timer, which is hardware that can be used to automatically detect software anomalies and reset the processor if any occur. The watchdog timer checks for proper processor operation. If the firmware on the motion controller is unable to process functions within 62 ms, the watchdog timer resets the motion controller and disallows further communications until you explicitly reset the motion controller. This ensures the real-time operation of the motion control system. The following functions may take longer than 62 ms to process:

- Save Defaults
- Reset Defaults
- Enable Auto Start
- Object Memory Management
- Clear Buffer
- End Storage

These functions are marked as non-real-time functions. Refer to the NI-Motion Function Help or the NI-Motion VI Help for more information.

The following block diagram illustrates the physical architecture of the NI motion controller hardware.





**Tip** Because the NI SoftMotion Controller is not a hardware device, information about its architecture is not covered in this topic. Refer to NI SoftMotion Controller Architecture for information about the functional architecture specific to the NI SoftMotion Controller.

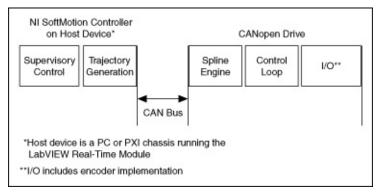
## NI SoftMotion Controller Architecture

The NI-Motion architecture for the NI SoftMotion Controller uses standard PC-based platforms and open standards to connect intelligent drives to a real-time host. In this architecture, the software components of the motion controller run on a real-time host and all I/O is implemented in the drives. This separation of I/O from the motion controller software components helps to lower system cost and improve reliability by improving connectivity. The CANopen standard is used to connect these components.

When you use the NI SoftMotion Controller with a CANopen device, you can daisy chain up to 15 drives together and connect them to the real-time host. The real-time Process Data Objects (PDOs) defined by the CANopen protocol are used to transfer data between the drives and host.

All I/O required by the motion controller is implemented by CANopen drives that support the Device Profile 402 for Motion Control. Currently, the NI SoftMotion Controller supports only CANopen drives from Copley Controls Corp. When used with CANopen devices, the Supervisory Control and Trajectory Generation components of the NI SoftMotion Controller execute in a real-time environment that is running LabVIEW Real-Time Module (ETS).

If your motion control system uses 8 axes or fewer, the supervisory control and trajectory generation loops execute every 10 milliseconds. If your motion control system uses more than 8 axes, the supervisory control and trajectory generation loops execute every 20 milliseconds. When you use the NI SoftMotion Controller with a CANopen drive, the drive implements the control loop and interpolation, as shown in the following figure.



In this configuration, the I/O and the control loop execute on the

CANopen drive. The NI SoftMotion Controller uses an NI-CAN device to communicate to the CAN bus.

# NI SoftMotion Controller Communication Watchdog

The supervisory control in the NI SoftMotion Controller continuously monitors all communication with the drives connected to the host. If any drive fails to update its data in the host loop update period, the axis corresponding to that drive is disabled and the communication watchdog status bit, which is returned by the Read Per Axis Status VI or function, is set to TRUE. Similarly, all drives connected to the NI SoftMotion Controller are configured to go into a fault state if the data from the NI SoftMotion Controller is not updated every host loop update period on the drives.

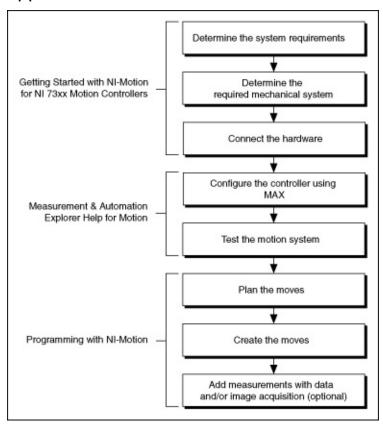
The communication watchdog functionality ensures that the NI SoftMotion Controller operates in real time.



**Tip** To get an axis or axes out of the communication watchdog state, reset the NI SoftMotion Controller.

# **Creating NI-Motion Applications**

The following figure describes the steps for creating an application with NI-Motion, and describes the generic steps required to design a motion application.



For information about interaction with other I/O, such as a National Instruments data and/or image acquisition devices, refer to <a href="Synchronization">Synchronization</a>.

# **Programming with NI-Motion**

You can use the C/C++ functions and LabVIEW VIs, included with NI-Motion, to configure and execute motion control applications. This section covers the NI-Motion algorithms you need to use all the features of NI-Motion.

Each task discussion uses the same structure. First, a generic algorithm flow chart shows how the component pieces relate to each other. Then, the task discussion details any aspects of creating the task that are specific to LabVIEW or C/C++ programming, complete with diagrams and code examples.



**Note** The LabVIEW block diagrams and C/C++ code examples are designed to illustrate concepts, and do not contain all the logic or safety features necessary for most functional applications.

Refer to the NI-Motion Function Help or the NI-Motion VI Help for detailed information about specific functions or VIs.

# What You Need to Know about Moves

This section discusses the concepts necessary for programming motion control including <u>move profiles</u>, <u>move types</u>, velocity and acceleration values, and timing loops.

## **Move Profiles**

The basic function of a motion controller is to make moves. The trajectory generator takes in the type of move and the move constraints and generates points, or instantaneous positions, in real time. Then, the trajectory generator feeds the points to the control loop.

The control loop converts each instantaneous position to a voltage or to step-and-direction signals, depending on the type of motor you are using.

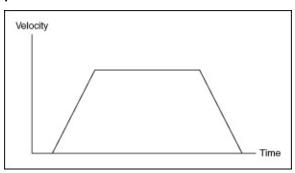
Move constraints are the maximum velocity, acceleration, deceleration, and jerk that the system can handle. The trajectory generator creates a velocity profile based on these move constraint values.

There are two types of profiles that can be generated while making the move: trapezoidal and s-curve.

# **Trapezoidal**

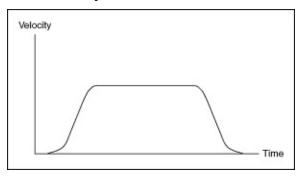
When you use a trapezoidal profile, the axes accelerate at the acceleration value you specify, and then cruise at the maximum velocity you load. Based on the type of move and the distance being covered, it may be impossible to reach the maximum velocity you set.

The velocity of the axis, or axes, in a coordinate space never exceeds the maximum velocity loaded. The axes decelerate to a stop at their final position, as shown.



## **S-Curve**

The acceleration and deceleration portions of an s-curve motion profile are smooth, resulting in less abrupt transitions, as shown in the following figure. This limits the jerk in the motion control system, but increases cycle time. The value by which the profile is smoothed is called the maximum jerk or s-curve value.



## **Basic Moves**

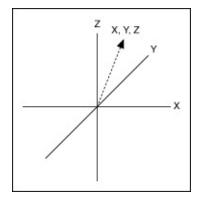
There are four basic move types:

- Reference Move—Initializes the axes to a known physical reference such as a home switch or encoder index
- **Straight–Line Move**—Moves from point A to point B in a straight line. The move can be based on position or velocity
- Arc Move—Moves from point A to point B in an arc or helix
- **Contoured Move**—A user-defined move; you generate the trajectory, and the points loaded into the motion controller are splined to create a smooth profile

The motion controller uses the specified move constraints, along with the move data, such as end position or radius and travel angle, to create a motion profile in all the moves except the contoured moves. Contoured moves ignore the move constraints and follow the points you have defined.

# **Coordinate Space**

With the exception of the arc move, you can execute all basic moves on either a single axis or on a coordinate space. A coordinate space is a logical grouping of axes. If you are performing a move that uses more than one axis, you must specify a coordinate space made up of the axes the move will use, as shown in the following figure.





**Note** Arc moves always execute on a coordinate space.

Use the Configure Vector Space VI or <u>function</u> to configure a coordinate space. This function creates a logical mapping of axes and treats the axes as part of a coordinate space. The function then executes the move generated by the trajectory generator on the vector, and treats all the move constraints as vector values.

## **Multi-Starts versus Coordinate Spaces**

Coordinate spaces always start and end the motion of all axes simultaneously. You can use multi-starts to create a similar effect without grouping axes into coordinate spaces. Using a multi-start automatically starts all axes virtually simultaneously. To simultaneously end the moves, you must calculate the move constraints to end travel at the same time. In coordinate spaces, these values are calculated automatically.

# **Trajectory Parameters**

Use trajectory parameters to control the moves you program in NI-Motion.

All trajectory parameters for servo axes are expressed in terms of quadrature encoder counts. Parameters for open-loop and closed-loop stepper axes are expressed in steps. For servo axes, the encoder resolution, which is expressed in counts per revolution, determines the ultimate positional resolution of the axis.

For stepper axes, the number of steps per revolution depends upon the type of stepper drive and motor you are using. For example, a stepper motor with 1.8°/step (200 steps/revolution) used in conjunction with a 10X microstep drive has an effective resolution of 2,000 steps per revolution. Resolution on closed-loop stepper axes is limited to the steps per revolution or encoder counts per revolution, whichever value is more coarse.

Floating-point versus fixed-point parameter representation and time base are two additional factors that affect the way trajectory parameters are loaded to the NI motion controller as compared to how they are used by the trajectory generators.

#### **Floating-Point versus Fixed-Point Parameters**

On NI 73xx motion controllers, you can load some trajectory parameters as either floating-point or fixed-point values, but the internal representation on the NI motion controller is always fixed-point. You must consider this functionality when working with onboard variables, inputs, and return vectors. This functionality also has a small effect on parameter range and resolution.

The NI SoftMotion Controller uses a 64-bit floating point trajectory generator. The ranges for all move constraints are the full 64-bit range, which includes maximum velocity, maximum acceleration, maximum deceleration, maximum acceleration jerk, maximum deceleration jerk, and velocity override percentage. All arc parameters that use floating point also support the full 64-bit floating point range.

#### NI 73xx Move Constraint Values

The following sections define the minimum and maximum values for move constraints as well as additional information about the resolution of the velocity override value and arc angle. Velocity and acceleration values are loaded in counts/s, RPM, RPS/s, steps/s, and so on, which are all functions of seconds or minutes. However, the trajectory generator updates target position at the Trajectory Update Rate, which is programmable with the Enable Axes VI or function. This means that the range for these parameters depends on the update rate selected.

NI 73xx Velocity in RPM

NI 73xx Velocity in Counts/s or Steps/s

NI 73xx Acceleration in Counts/s<sup>2</sup>

NI 73xx Acceleration in RPS/s

NI 73xx Velocity Override in Percent

NI 73xx Arc Angles in Degrees

## NI 73xx Velocity in RPM

Velocity values in RPM are converted to an internal 16.16 fixed-point format in units of counts (steps) per sample period (update period) before being used by the trajectory generator. NI-Motion can control velocity to 1/65,536 of a count or step per sample.

The following table shows the minimum and maximum velocity in counts/min. Use the formula shown in the *Calculation Based on Units* column to determine the counts/min value to RPM.

Update Rate	MIN	MAX	Calculation Based on Units
62.5 µs		For servo motors, the maximum counts/min is 1.2 billion independent of the update rate.	$\pm RPM_{max} =$ $MAX \times 1/r$ where $r =$ counts/revolution
125 μs	7.324219 counts/min		
187.5 μs	4.882813 counts/min		
250 μs	3.662109 counts/min		
312.5 μs	2.929688 counts/min	For stepper motors, the maximum counts/min value is dependent on the controller:  • 480 million counts/min for NI 7350  • 240 million counts/min for NI 7330/7340/7390	±RPM <sub>min</sub> = MIN×1/r where r = counts/revolution
375 μs	2.441406 counts/min		
437.5 μs	2.092634 counts/min		
500 μs	1.831055 counts/min		

You can calculate this minimum velocity increment in RPM with the following formula:

minimum RPM = 
$$Vmin \times (\frac{1}{T_s}) \times 60 \times (\frac{1}{R})$$

where

Vmin = 1/65,536 counts/sample or steps/sample

Ts = sample period in seconds per sample

60 = number of seconds in a minute

R =counts or steps per revolution

or

minimum RPM = MIN  $\times \frac{1}{R}$ 

You also can calculate the minimum velocity using the formula shown in the *Calculation Based on Units* column.

For a typical servo axis with 2,000 counts per revolution operating at a 250  $\mu$ s update rate, the minimum RPM increment is

$$\frac{1}{65,536} \times \frac{1}{250\mu s} \times 60 \times \frac{1}{2000} = 0.00183105 \text{ RPM}$$

or

$$3.662109 \times \frac{1}{2000} = 0.00183105 \text{ RPM}$$

You can calculate the maximum velocity in RPM with the following equation:

maximum RPM =  $Vmax \times 60 \times \frac{1}{R}$ 

where

Vmax = 20 MHz for servos

8 MHz for steppers on a NI 7350 controller

4 MHz for steppers on a NI 7330, NI 7340, or NI 7390 motion controller

*R* = counts/steps per revolution

and is constrained by acceleration/deceleration according to the following equation:

velocity ≤ (65,536 × deceleration) acceleration

where velocity is in counts/sample and acceleration and deceleration are in counts/sample<sup>2</sup>.

From the example, the maximum RPM is

$$(20 \times 10^6) \times 60 \times \frac{1}{2000} = 600,000 \text{ RPM}$$

RPM values stored in onboard variables are in double-precision IEEE format (f64).

## NI 73xx Velocity in Counts/s or Steps/s

Velocity values in counts/s or steps/s are also converted to the internal 16.16 fixed-point format in units of counts or steps per sample (update) period before being used by the trajectory generator. Although the motion controller can control velocity to 1/65,536 of a count or step per sample, it is impossible to load a value that small with the Load Velocity VI or function, as shown in the following formula:

Velocity in counts or step/s = Vmin  $\times \frac{1}{T_s}$ 

where:

Vmin is 1/65,536 counts/sample or steps/sample and

*T*s is the sample period in seconds per sample.

Even at the fastest update rate,  $Ts = 62.5 \times 10^6$ 

$$\left(\frac{1}{65,536}\right) \times \left(\frac{1}{62.5 \times 10^{-6}}\right) = 0.244 \text{ counts or steps/s}$$

Load Velocity takes an integer input with a minimum value of 1 count/s or step/s. You cannot load fractional values. If you need to load a velocity slower than one count or step per second, use the Load Velocity in RPM VI or function.

You can calculate the maximum velocity with the following equation:

maximum velocity = Vmax

where

Vmax = 20 MHz for servos

8 MHz for steppers on a NI 7350 controller

4 MHz for steppers on a NI 7330, NI 7340, or NI 7390 motion controller and is constrained by acceleration/deceleration according to the following equation:

velocity ≤ (65,536 × deceleration) acceleration

where velocity is in counts/sample and acceleration and deceleration are in counts/sample<sup>2</sup>.

# NI 73xx Acceleration in Counts/s<sup>2</sup>

Acceleration and deceleration values are converted to an internal 16.16 fixed-point format in units of counts/s<sup>2</sup> before being used by the trajectory generator.

The following table shows the minimum and maximum acceleration update rates in counts/sec<sup>2</sup>.

Update Rate	MAX	MIN	Calculation Based on Units
62.5 μs	2,048,000,000 counts/sec <sup>2</sup>	3906 counts/sec <sup>2</sup>	Accel <sub>max</sub> = MAX
125 μs	2,048,000,000 counts/sec <sup>2</sup>	977 counts/sec <sup>2</sup>	
187.5 μs	910,222,222 counts/sec <sup>2</sup>	434 counts/sec <sup>2</sup>	
250 μs	512,000,000 counts/sec <sup>2</sup>	244 counts/sec <sup>2</sup>	
312.5 µs	327,680,000 counts/sec <sup>2</sup>	156 counts/sec <sup>2</sup>	Accel <sub>min</sub> = MIN
375 μs	227,555,556 counts/sec <sup>2</sup>	109 counts/sec <sup>2</sup>	
437.5 μs	167,183,673 counts/sec <sup>2</sup>	80 counts/sec <sup>2</sup>	
500 μs	128,000,000 counts/sec <sup>2</sup>	61 counts/sec <sup>2</sup>	

You can calculate the minimum acceleration increment with the following formula:

minimum acceleration/deceleration =  $Amin \times (\frac{1}{Ts})^2$  where

Amin = 1/65,536 counts/sample<sup>2</sup> or steps/sample<sup>2</sup>

Ts = sample period in seconds per sample

For a typical servo axis with 2,000 counts per revolution operating at the 250  $\mu$ s update rate, calculate the minimum acceleration/deceleration increment using the following equation:

$$(\frac{1}{65,536}) \times (\frac{1}{250\mu s})^2 = 244 \text{ counts/second}^2$$

You can calculate the maximum acceleration/deceleration using the following equation:

maximum acceleration/deceleration =  $Amax \times (\frac{1}{T_s})^2$  where

 $Amax = 32 \text{ counts/sample}^2$ 

Ts =sample period in seconds per sample

and is constrained according to the following equations:

acceleration 256 ≤ deceleration

deceleration 65536 ≤ acceleration

### NI 73xx Acceleration in RPS/s

Acceleration and deceleration values in RPS/s are converted to an internal 16.16 fixed-point format in units of counts/sample<sup>2</sup> or steps/sample<sup>2</sup> before being used by the trajectory generator.

The following table shows the minimum and maximum acceleration update rates in counts/sec<sup>2</sup>.

Update Rate	MAX	MIN	Calculation Based on Units
62.5 μs	2,048,000,000 counts/sec <sup>2</sup>	3906 counts/sec <sup>2</sup>	$\pm RPS/s_{max} = MAX \times 1/r$ where $r =$ counts/revolution
125 μs	2,048,000,000 counts/sec <sup>2</sup>	977 counts/sec <sup>2</sup>	
187.5 μs	910,222,222 counts/sec <sup>2</sup>	434 counts/sec <sup>2</sup>	
250 μs	512,000,000 counts/sec <sup>2</sup>	244 counts/sec <sup>2</sup>	
312.5 μs	327,680,000 counts/sec <sup>2</sup>	156 counts/sec <sup>2</sup>	$\pm RPS/s_{min} = MIN \times 1/r$ where $r =$ counts/revolution
375 μs	227,555,556 counts/sec <sup>2</sup>	109 counts/sec <sup>2</sup>	
437.5 μs	167,183,673 counts/sec <sup>2</sup>	80 counts/sec <sup>2</sup>	
500 μs	128,000,000 counts/sec <sup>2</sup>	61 counts/sec <sup>2</sup>	

You can calculate the minimum acceleration increment in RPS/s with the following formula:

RPS/sec = Amin × 
$$(\frac{1}{T_s})^2$$
 ×  $\frac{1}{R}$  where

Amin = 1/65,536 counts/sample<sup>2</sup> or steps/sample<sup>2</sup>

Ts =sample period in seconds per sample

R = counts or steps per revolution

or

RPM/s = MIN 
$$\times \frac{1}{p}$$

For a typical servo axis with 2,000 counts or steps per revolution operating at the 250  $\mu$ s update rate, calculate the minimum RPS/s increment using the following equation:

$$\left(\frac{1}{65,536}\right) \times \left(\frac{1}{250\mu s}\right)^2 \times \frac{1}{2000} = 0.122070 \text{ RPS/s}$$

or

$$244 \times \frac{1}{2000} = 0.122$$

You can calculate the maximum RPS/s using the following equation:

maximum RPS/s = 
$$Amax \times (\frac{1}{Ts})^2 \times (\frac{1}{R})$$

where

 $Amax = 32 \text{ counts/sample}^2$ 

Ts = sample period in seconds per sample

R = counts or steps per revolution

and is constrained according to the following equations:

acceleration ≤ 256 × deceleration

deceleration ≤ 65536 × acceleration

or

$$MAX \times \frac{1}{r}$$

For a typical servo axis with 2,000 counts or steps per revolution operating at the 250  $\mu$ s update rate, calculate the maximum RPS/s increment using the following equation:

$$32 \times (\frac{1}{250\mu s})^2 \times \frac{1}{2000} = 256,000 \text{ RPS/s}$$

RPS/s values stored in onboard variables are in double-precision IEEE format (f64).

## NI 73xx Velocity Override in Percent

The Load Velocity Override VI or <u>function</u> takes a single-precision floating-point (f32) data value from 0 to 150%, but velocity override is internally implemented as a velocity scale factor of 0 to 384 with an implicit fixed denominator of 256. NI-Motion uses the velocity override to increase the speed of the calculation for the sake of calculation speed—the division is a shift right by eight bits. The resolution for velocity override is therefore limited to 1/256, or about 0.39%.



**Note** The conversion from floating-point to fixed-point is performed on the host computer, not on the motion controller. To load velocity override from an onboard variable, you must use the integer representation of 0 to 384, where 384 corresponds to 150%.



**Note** If the distance of the move is too small, it may not be possible to reach the commanded maximum move constraints. In such instances, NI-Motion adjusts the move constraints lower to reach the commanded position.

## NI 73xx Arc Angles in Degrees

The Load Circular Arc, Load Helical Arc, and Load Spherical Arc VIs/functions take angle parameters in degrees as double-precision floating-point values. These values are converted to an internal 16.16 fixed-point representation where the integer part corresponds to multiples of 45° (for example, 360° is represented as 0x0008 0000).

Use the following formula to convert from floating-point to fixed point:

$$\frac{\text{Angle in degrees}}{45^{\circ}} = Q + R$$

where:

Q is the quotient, the integer multiple of 45°

R is the remainder.

Angle in 16.16 format = 
$$Q.(\frac{R}{45^{\circ}} \times 65,536)$$

For example, 94.7° is represented in 16.16 format as follows:

Angle in 16.16 format = 
$$2.\left(\frac{4.7^{\circ}}{45^{\circ}} \times 65,536\right)$$

The minimum angular increment is therefore

$$\frac{1}{65,536} \times 45^{\circ} = 0.000687^{\circ}$$



**Note** The conversion from floating-point to fixed-point is performed on the host computer, not on the NI motion controller. To load arc VIs from onboard variables, you must use the 16.16 fixed-point representation for all angles.

#### **NI 73xx Arc Move Limitations**

The following are limitations to the velocity and acceleration of arc moves.



**Note** Round the result of these equations to the nearest integer.

Arc moves must meet the requirements of the following equations or an NIMC\_invalidVelocityError is generated:

$$V \times P \times 4 < R$$

and

$$65,536 \ge \frac{V \times I \times 83,443}{R} \ge 16$$

where

*V* = Velocity in counts/s

P = PID sample rate in seconds

I = Coarse Arc Points Interval Period (10 ms or 20 ms) in seconds, configured in  $\underline{\text{MAX}}$ 

R = Radius in counts

Arc moves must meet the requirements of the following equations or an NIMC\_invalidAccelerationError is generated:

$$A \times P \times 4 < R$$

and

$$65,536 \ge \frac{A \times I^2 \times 83,443}{R} \ge 1$$

where

 $A = Acceleration/deceleration in counts/s^2$ 

P = PID sample rate in seconds

I = Coarse Arc Points Interval Period (10 ms or 20 ms) in seconds, configured in  $\underline{\text{MAX}}$ 

R = Radius in counts

# **Timing Loops**

National Instruments recommends that you use the loop timings discussed in the following sections.

#### **Status Display**

When you are displaying status information to the user, such as position, move status, or velocity, an update rate faster than 60 ms has no value. In fact, there is no need to update a display any faster than 22 Hz because the human eye can detect flicker only at refresh rates slower than 22 Hz.

However, you might see flicker in monitors at around 60 Hz, because of interference with artificial light from light bulbs that run on a 60 Hz AC signal. The recommended standard is 60 ms because one might need multiple function calls within one loop to acquire all the necessary data.

#### **Graphing Data**

When acquiring data for graphing or tracking purposes, a 10 ms update time suits most applications. MAX, for example, updates its motion graphs every 10 ms. This update time equates to 100 samples every second and provides enough resolution for typical applications. Consider how accurate the graph display is when choosing the timing for the loop.

#### **Event Polling**

Use a polling interval of 5 ms when polling for a time-critical event that must occur before the program continues. This interval is fast enough to satisfy most time-critical polling needs, although certain high-speed applications may require a faster interval. Consider the allowable response time when choosing a polling interval.

For example, to synchronize the motion with the acquisition in an application where a user places an object under the scan area and clicks a **Scan** button, you create periodic breakpoints every 10 counts to trigger a data acquisition over RTSI. In this example, the loop needs only to read the position and wait for the move to complete before ending the scan. Although the program polls for an event (move complete), no action is being triggered by the move complete. Because there is no need for instantaneous action, there is no need to update the position any faster than 60 ms, and 60 ms is acceptable for monitoring the move complete status as well.

# **Straight-Line Moves**

A straight-line move executes the shortest move between two points and can use one, two, or three axes. Straight-line moves are either <u>position-based</u> or <u>velocity-based</u>.

### **Position-Based Straight-Line Moves**

Position-based straight-line moves use the specified target position to generate the move trajectory. For example, if the motor is currently at position zero, and the target position is 100, a position-based move creates a trajectory that moves 100 counts (steps).

The controller requires the following information to move to another position in a straight line:

- **Start position**—Current position, normally held over from a previous move or initialized to zero
- **End position**—Also known as the target position, or where you want to move to
- Move constraints—Maximum velocity, maximum acceleration, maximum deceleration, and maximum jerk



**Tip** When you are using the NI SoftMotion Controller, you can load separate acceleration and deceleration jerk values.

The motion controller uses the given information to create a trajectory that never exceeds the move constraints and that moves an axis or axes to the end position you specify. The controller generates the trajectory in real time, so you can change any of the parameters while the axes are moving.

#### **Straight-Line Move Algorithm**

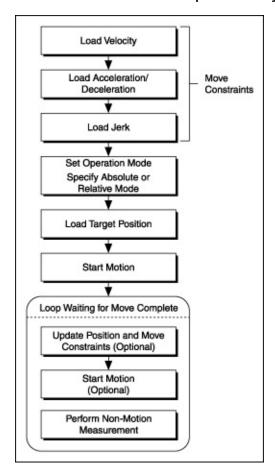
The straight-line move algorithm includes the following procedures:

- Load target position—Specifies the end position
- Load the move constraints—Loads the velocity, acceleration, deceleration, and jerk values
- Start motion—Starts the move

The start position is always the current position of the axis or axes. You can load the end position as either an absolute position to move to or as a position relative to the starting position. Although you can update any parameter while the move is in progress, the new parameter is used only after a subsequent Start or Blend Move.



**Tip** You must load the move constraints only if they are different from what was previously loaded.



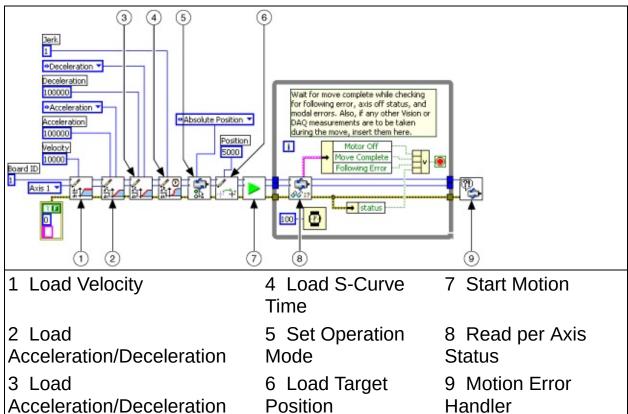
# **Straight-Line Move LabVIEW Diagram**

The following block diagrams demonstrate using NI-Motion to complete a 1D and 2D straight-line move, respectively.



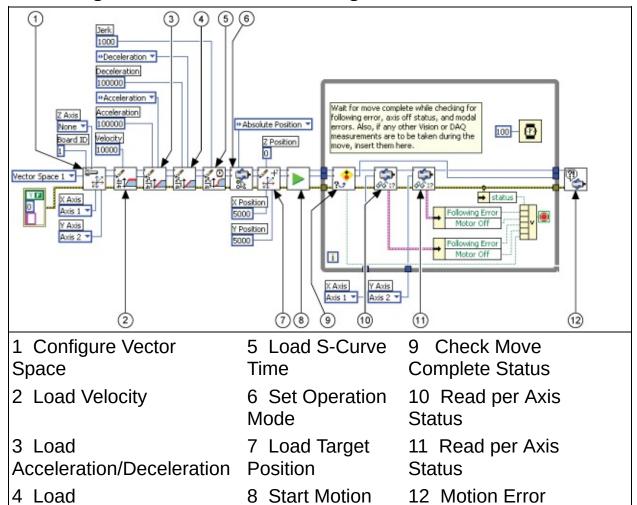
**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.

#### 1D Straight-Line Move Block Diagram



#### 2D Straight-Line Move Block Diagram

Acceleration/Deceleration



Handler

# **Straight-Line Move C/C++ Code**



**Tip** The following example code is not necessarily complete, and may not compile if copied exactly. Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for files that are complete and compile as is.

# 1D Straight-Line Move Code

# **2D Straight-Line Move Code**

## **Velocity-Based Straight-Line Moves**

Some motion applications require moves that travel in a straight line for a specific amount of time at a given speed. This type of move is known as *velocity profiling* or *jogging*.

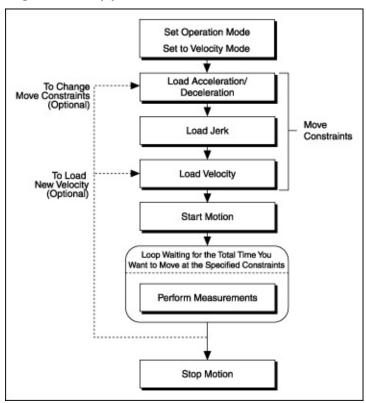
You can use a motion control application to move a motor at a given speed for a specific time, and then change the speed without stopping the axis. The sign of the loaded velocity specifies the direction of motion. Positive velocity implies forward motion and negative velocity implies reverse motion.



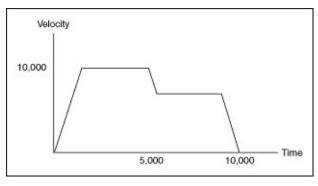
**Tip** You can change the move constraints during a velocity move.

#### **Velocity-Based Straight-Line Move Algorithm**

The following figure shows a generic velocity-based straight-line move algorithm applicable to both C/C++ and LabVIEW code.



Loading a second velocity and executing the Start Motion VI or function causes the motion controller to accelerate or decelerate to the newly loaded velocity using the acceleration or deceleration parameters last loaded. The axis decelerates to a stop using the Stop Motion VI or function. The velocity profile created in the example code is shown in the following figure.



# Velocity-Based Straight-Line Move LabVIEW Diagram

The following block diagram demonstrates using NI-Motion to complete a velocity-based straight-line move.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.

1 Load Velocity	5 Set Operation Mode	9 Start Motion
2 Load Acceleration/Deceleration	6 Start Motion	10 Read per Axis Status
3 Load Acceleration/Deceleration	7 Read per Axis Status	11 Stop Motion
4 Load S-Curve Time	8 Load Velocity	12 Motion Error Handler

## Velocity-Based Straight-Line Move C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
   u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
i32 moveTime1;  // Time for the 1st segment
i32 moveTime2;  // Time for the 2nd segment
i32 initialTime:
   i32 initialTime;
   i32 currentTime;
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 3;
   // Set the axis number
   axis = 1;
   // Move time for the first segment
   moveTime1 = 5000; //milliseconds
   // Move time for the second segment
   moveTime2 = 10000; //milliseconds
   // First segment
   // Set the velocity for the move (in counts/sec)
   err = flex load velocity(boardID, axis, 10000, 0xFF);
   CheckError;
   // Set the acceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, axis, NIMC_ACCELERATION, 10
   CheckError;
   // Set the deceleration for the move (in counts/sec^2)
```

```
err = flex_load_acceleration(boardID, axis, NIMC_DECELERATION, 10
CheckError;
// Set the jerk (s-curve value) for the move (in sample periods)
err = flex_load_scurve_time(boardID, axis, 100, 0xFF);
CheckError;
// Set the operation mode to velocity
err = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
CheckError;
// Start the move
err = flex_start(boardID, axis, 0);
CheckError;
// Wait for the time for first segment
initialTime = timeGetTime();
do
{
  // Check the move complete status/following error/axis off sta
  err = flex read axis status rtn(boardID, axis, &axisStatus);
  CheckError;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
   // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG)
      err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
  }
  // Get the current time and check if time is over for the firs
  currentTime = timeGetTime();
   if((currentTime - initialTime) >= moveTime1) break;
  Sleep (50); // Check every 50 ms
}while (!(axisStatus & NIMC_MOVE_COMPLETE_BIT) && !(axisStatus &
//-----
// Second segment
// Set the velocity for the move (in counts/sec)
err = flex_load_velocity(boardID, axis, 6568, 0xFF);
CheckError;
// Start the move - to update the velocity
```

```
err = flex_start(boardID, axis, 0);
CheckError;
// Wait for the time for second segment
initialTime = timeGetTime();
do
{
  // Check the move complete status/following error/axis off sta
  err = flex_read_axis_status_rtn(boardID, axis, &axisStatus);
  CheckError;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG)
   {
     err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
  }
  // Get the current time and check if time is over for the seco
  currentTime = timeGetTime();
  if((currentTime - initialTime) >= moveTime2) break;
  Sleep (50); // Check every 50 ms
}while (!(axisStatus & NIMC_MOVE_COMPLETE_BIT) && !(axisStatus &
// Decelerate the axis to a stop
err = flex_stop_motion(boardID, axis, NIMC_DECEL_STOP, 0);
CheckError;
return; // Exit the Application
// Error Handling
//
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG){
  do{
     // Get the command ID, resource and the error code of the m
     // error from the error stack on the board
     flex read error msq rtn(boardID,&commandID,&resourceID,&err
     nimcDisplayError(errorCode,commandID,resourceID);
```

```
// Read the Communication Status Register
flex_read_csr_rtn(boardID,&csr);

}while(csr & NIMC_MODAL_ERROR_MSG);
}
else    // Display regular error
    nimcDisplayError(err,0,0);
return;    // Exit the Application
}
```

## **Velocity Profiling Using Velocity Override**

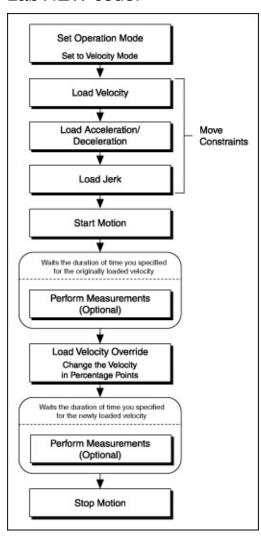
You also can use the Load Velocity Override VI or <u>function</u> to shift from one velocity to another while executing moves. When you use this function, you indicate the new velocity in terms of a percentage of the originally loaded velocity instead of explicitly stating the velocity you want to change to.

For example, 120 percent of an original velocity of 10,000 changes the velocity to 12,000.

The transition between velocities follows all other move constraints.

#### **Velocity Profiling Using Velocity Override Algorithm**

The following figure shows a velocity-based straight-line move algorithm showing the use of velocity override applicable to both C/C++ and LabVIEW code.

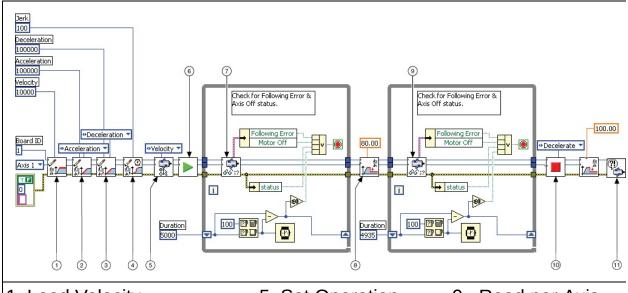


# Velocity Profiling Using Velocity Override LabVIEW Diagram

The following block diagram demonstrates using NI-Motion to do velocity profiling using Velocity Override.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



1 Load Velocity	5 Set Operation Mode	9 Read per Axis Status
2 Load Acceleration/Deceleration	6 Start Motion	10 Stop Motion
3 Load Acceleration/Deceleration	7 Read per Axis Status	11 Motion Error Handler
4 Load S-Curve Time	8 Load Velocity Override	

# **Velocity Profiling Using Velocity Override C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
i32 moveTime1;  // Time for the 1st segment
i32 moveTime2;  // Time for the 2nd segment
   i32 initialTime;
   i32 currentTime;
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 3:
   // Set the axis number
   axis = 1;
   // Move time for the first segment
   moveTime1 = 5000; //milliseconds
   // Move time for the second segment
   moveTime2 = 10000; //milliseconds
   //-----
   // First seament
   //-----
   // Set the velocity for the move (in counts/sec)
   err = flex_load_velocity(boardID, axis, 10000, 0xFF);
   CheckError:
   // Set the acceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, axis, NIMC_ACCELERATION, 10
```

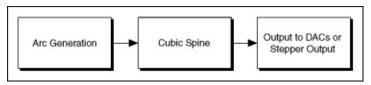
```
CheckError;
// Set the deceleration for the move (in counts/sec^2)
err = flex_load_acceleration(boardID, axis, NIMC_DECELERATION, 10
CheckError;
// Set the jerk (s-curve value) for the move (in sample periods)
err = flex_load_scurve_time(boardID, axis, 100, 0xFF);
CheckError;
// Set the operation mode to velocity
err = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
CheckError;
// Start the move
err = flex_start(boardID, axis, 0);
CheckError;
// Wait for the time for first segment
initialTime = timeGetTime();
do
{
  // Check the move complete status/following error/axis off sta
  err = flex read axis status rtn(boardID, axis, &axisStatus);
  CheckError:
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC MODAL ERROR MSG)
  {
     err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
  }
  // Get the current time and check if time is over for the firs
  currentTime = timeGetTime();
  if((currentTime - initialTime) >= moveTime1) break;
  Sleep (50); // Check every 50 ms
}while (!(axisStatus & NIMC_MOVE_COMPLETE_BIT) && !(axisStatus &
//-----
// Second segment
//-----
// Change the velocity to 80% of the initially loaded value
err = flex load velocity override(boardID, axis, 80, 0xFF);
```

```
CheckError;
// Wait for the time for second segment
initialTime = timeGetTime();
do
{
  // Check the move complete status/following error/axis off sta
  err = flex_read_axis_status_rtn(boardID, axis, &axisStatus);
  CheckError;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG)
     err = csr & NIMC MODAL ERROR MSG;
     CheckError;
  }
  // Get the current time and check if time is over for the seco
  currentTime = timeGetTime();
  if((currentTime - initialTime) >= moveTime2) break;
  Sleep (50); // Check every 50 ms
}while (!(axisStatus & NIMC MOVE COMPLETE BIT) && !(axisStatus &
// Decelerate the axis to a stop
err = flex stop motion(boardID, axis, NIMC DECEL STOP, 0);
CheckError;
// Reset velocity override back to 100%
err = flex_load_velocity_override(boardID, axis, 100, 0xFF);
CheckError;
return; // Exit the Application
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC MODAL ERROR MSG){
  do{
```

#### **Arc Moves**

An arc move causes a coordinate space of axes to move on a <u>circular</u>, <u>spherical</u>, or <u>helical</u> path. You can move two-dimensional vector spaces in a circle only on a 2D plane. You can move a 3D vector space on a spherical or helical path.

Each arc generated by the motion controller passes through a cubic spline algorithm that ensures the smoothest arc. This also ensures negligible chordal error, which is error caused when two points on the surface of the arc join with each other using a straight line. A cubic spline algorithm generates multiple points between every two points of the arc, ensuring smooth motion, minimum jerk, and maximum accuracy at all times. The data path is shown in the following figure.





**Note** Arc moves must meet the requirements of the equations in NI 73xx Arc Move Limitations or an NIMC\_invalidVelocityError is generated.

#### **Circular Arcs**

A circular arc defines an arc in the XY plane of a 2D or 3D coordinate space. The arc is specified by a radius, starting angle, and travel angle. Also, like all coordinate space moves, the arc uses the values of move constraints—maximum velocity, maximum acceleration, and maximum deceleration.



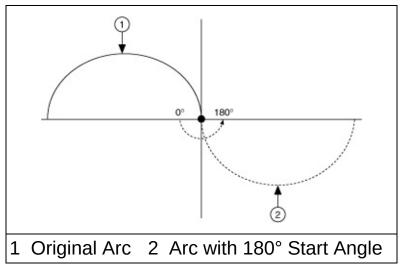
**Tip** For the NI SoftMotion Controller, the arc generation also uses acceleration jerk and deceleration jerk while calculating the arc move.



**Note** When you use an NI 73xx motion controller to move a motor in an arc, you can use only trapezoidal profiles. You do not use jerk to calculate the profile for arc moves.

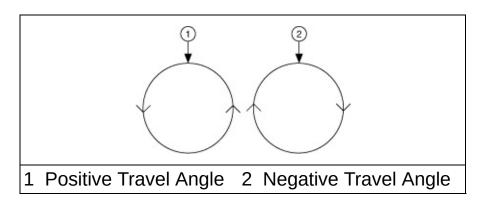
To move axes in a circular arc, the motion controller needs the following information:

- Radius—Specifies the distance from the center of the arc to its edge
- **Start Angle**—Orients the arc on its plane using the starting point as an axis to spin around. Because the starting point for a new arc is fixed based on the current position, moving its center around the starting point alters the orientation of a new arc. For example, the following figure shows the effect of changing the start angle from 0° to 180°.



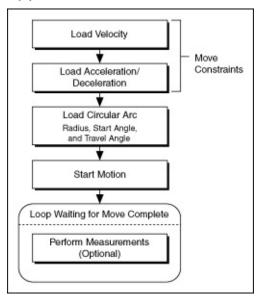
• Travel Angle—Indicates how far the arc travels in a 360° circle.

For example, a travel angle of 90° executes a quarter-circle, a travel angle of 360° creates a full circle, and a travel angle of 720° creates two full circles. A positive travel angle always creates counterclockwise circular motion. A negative travel angle reverses the direction to create clockwise circular motion, as shown in the following figure.



## **Circular Arc Move Algorithm**

The following figure shows a generic circular arc move algorithm applicable to both C/C++ and LabVIEW code.

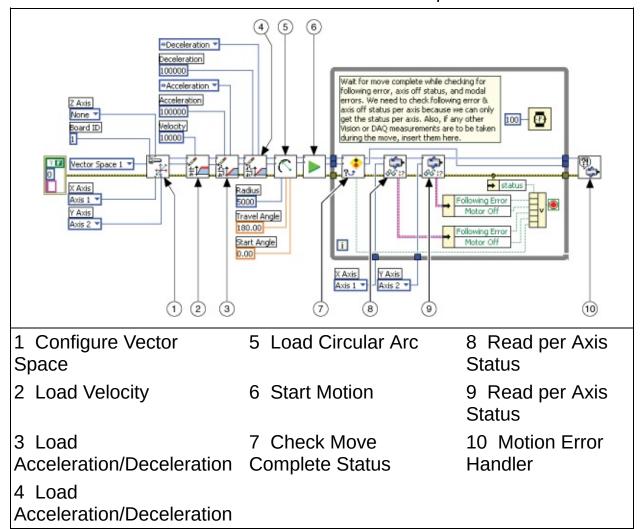


## **Circular Arc Move LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to complete a circular arc move.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



#### Circular Arc Move C/C++ Code

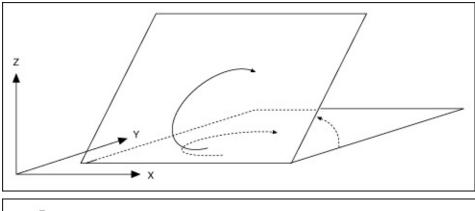
The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

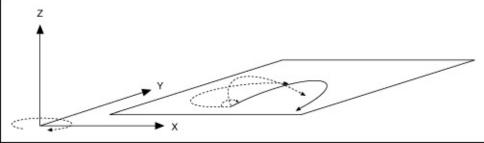
```
// Main function
void main(void)
   u8 boardID;  // Board identification number
u8 vectorSpace;  // Vector space number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
   u16 status;
   u16 moveComplete;
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 1;
   // Set the axis number
   vectorSpace = NIMC_VECTOR_SPACE1;
   // Configure a 2D Vector Space comprising of axes 1 and 2
   err = flex_config_vect_spc(boardID, vectorSpace, 1, 2, 0);
   CheckError;
   // Set the velocity for the move (in counts/sec)
   err = flex_load_velocity(boardID, vectorSpace, 10000, 0xFF);
   CheckError;
   // Set the acceleration for the move (in counts/sec^2)
   err = flex load acceleration(boardID, vectorSpace, NIMC ACCELERAT
   CheckError;
   // Set the deceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, vectorSpace, NIMC_DECELERAT
   CheckError;
```

```
// Load Spherical Arc
err = flex load circular arc (boardID, vectorSpace, 5000/*radius*
CheckError;
// Start the move
err = flex_start(boardID, vectorSpace, 0);
CheckError;
do
{
  axisStatus = 0;
  //Check the move complete status
  err = flex check move complete status(boardID, vectorSpace, 0,
  CheckError;
  // Check the following error/axis off status for axis 1
  err = flex read axis status rtn(boardID, 1, &status);
  CheckError;
  axisStatus |= status;
  // Check the following error/axis off status for axis 2
  err = flex_read_axis_status_rtn(boardID, 2, &status);
  CheckError;
  axisStatus |= status;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
   // Check the modal errors
  if (csr & NIMC MODAL ERROR MSG)
     err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
   }
}while (!moveComplete && !(axisStatus & NIMC FOLLOWING ERROR BIT)
return; // Exit the Application
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC MODAL ERROR MSG){
  do{
```

## **Spherical Arcs**

A 3D spherical arc defines a 2D circular arc in the X'Y' plane of a coordinate system that is transformed by rotation in pitch, as shown in the first figure, and yaw, as shown in the second figure, from the normal 3D coordinate space (XYZ).



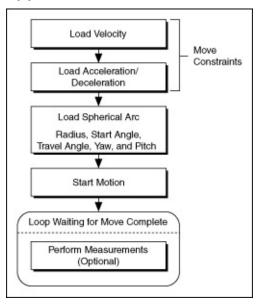


In the transformed X'Y'Z' space, the 3D arc is reduced to a simpler 2D arc. The 3D arc is defined as a 2D circular arc in the X'Y' plane of a transformed vector space X'Y'Z'. This transformed vector space, X'Y'Z', is defined in orientation only, with no absolute position offset. Its orientation is relative to the XYZ vector space, and is defined in terms of pitch and yaw angles. When rotating through the pitch angle, the Y and Y' axes stay aligned with each other while the X'Z' plane rotates around them. When rotating through the yaw angle, the Y' axis never leaves the original XY plane, as the newly-defined X'Y'Z' vector space rotates around the original Z-axis.

The radius, start angle, and travel angle parameters also apply to a spherical arc that defines the arc in two dimensions.

## **Spherical Arc Move Algorithm**

The following figure shows a generic spherical arc move algorithm applicable to both C/C++ and LabVIEW code.

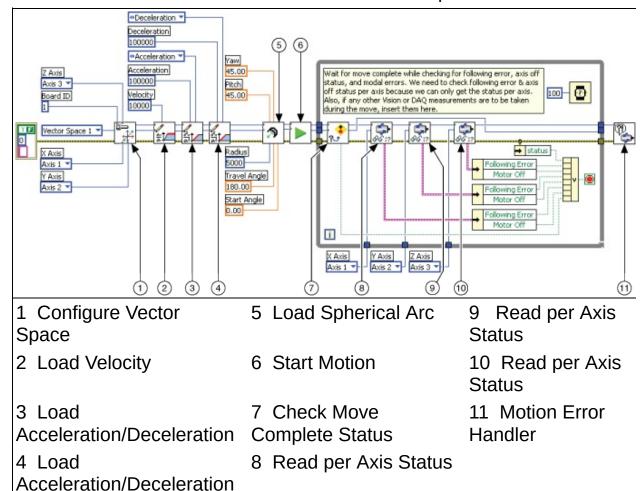


## **Spherical Arc Move LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to complete a spherical arc move.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



## **Spherical Arc Move C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

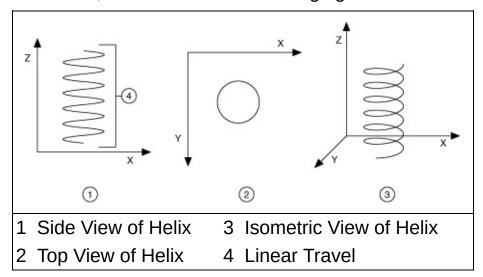
```
// Main function
void main(void)
   u8 boardID;  // Board identification number
u8 vectorSpace;  // Vector space number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
   u16 status;
   u16 moveComplete;
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 1;
   // Set the axis number
   vectorSpace = NIMC_VECTOR_SPACE1;
   // Configure a 3D Vector Space comprising of axes 1, 2 and 3
   err = flex_config_vect_spc(boardID, vectorSpace, 1, 2, 3);
   CheckError;
   // Set the velocity for the move (in counts/sec)
   err = flex_load_velocity(boardID, vectorSpace, 10000, 0xFF);
   CheckError;
   // Set the acceleration for the move (in counts/sec^2)
   err = flex load acceleration(boardID, vectorSpace, NIMC ACCELERAT
   CheckError;
   // Set the deceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, vectorSpace, NIMC_DECELERAT
   CheckError;
```

```
// Load Spherical Arc
err = flex_load_spherical_arc (boardID, vectorSpace, 5000/*radius
        45.0/*planeYaw*/, 0.0/*startAngle*/, 180.0/*travelAngle*
CheckError;
// Start the move
err = flex start(boardID, vectorSpace, 0);
CheckError;
do
{
  axisStatus = 0;
  // Check the move complete status
  err = flex_check_move_complete_status(boardID, vectorSpace, 0,
  CheckError;
  // Check the following error/axis off status for axis 1
  err = flex read axis status rtn(boardID, 1, &status);
  CheckError;
  axisStatus |= status;
  // Check the following error/axis off status for axis 2
  err = flex read axis status rtn(boardID, 2, &status);
  CheckError;
  axisStatus |= status;
  // Check the following error/axis off status for axis 3
  err = flex read axis status rtn(boardID, 3, &status);
  CheckError;
  axisStatus |= status;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC MODAL ERROR MSG)
     err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
   }
}while (!moveComplete && !(axisStatus & NIMC_FOLLOWING_ERROR_BIT)
            // Exit the Application
return;
// Error Handling
```

```
//
   nimcHandleError; //NIMCCATCHTHIS:
   // Check to see if there were any Modal Errors
   if (csr & NIMC_MODAL_ERROR_MSG){
      do{
         // Get the command ID, resource and the error code of the m
         // error from the error stack on the board
         flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
         nimcDisplayError(errorCode, commandID, resourceID);
         // Read the Communication Status Register
         flex_read_csr_rtn(boardID, &csr);
      }while(csr & NIMC_MODAL_ERROR_MSG);
   }
   else
              // Display regular error
      nimcDisplayError(err, 0, 0);
   return; // Exit the Application
}
```

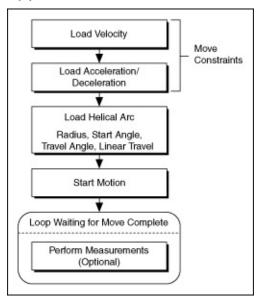
#### **Helical Arcs**

A helical arc defines an arc in a 3D coordinate space that consists of a circle in the XY plane and synchronized linear travel in the Z-axis. The arc is specified by a radius, start angle, travel angle, and Z-axis linear travel. Linear travel is the linear distance traversed by the helical arc on the Z-axis, as shown in the following figure.



## **Helical Arc Move Algorithm**

The following figure shows a generic helical arc move algorithm applicable to both C/C++ and LabVIEW code.

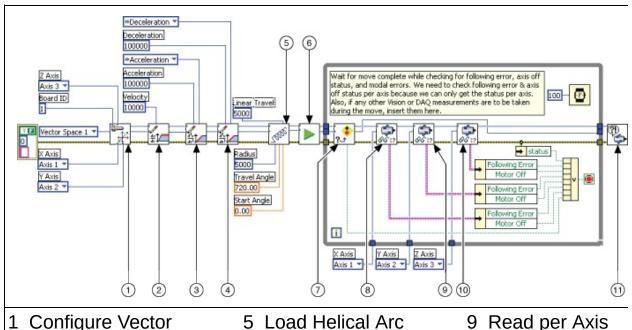


## **Helical Arc Move LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to complete a helical arc move.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



1 Configure Vector 5 Load Helical Arc 9 Read per Axis Space Status

2 Load Velocity 6 Start Motion 10 Read per Axis Status

3 Load 7 Check Move Complete 11 Motion Error Acceleration/Deceleration Status Handler

4 Load 8 Read per Axis Status

Acceleration/Deceleration

#### Helical Arc Move C/C++ Code

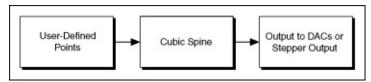
The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void){
   u8 boardID;  // Board identification number
u8 vectorSpace;  // Vector space number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
   u16 status;
   u16 moveComplete;
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 1;
   // Set the axis number
   vectorSpace = NIMC VECTOR SPACE1;
   // Configure a 3D Vector Space comprising of axes 1, 2 and 3
   err = flex_config_vect_spc(boardID, vectorSpace, 1, 2, 3);
   CheckError;
   // Set the velocity for the move (in counts/sec)
   err = flex load velocity(boardID, vectorSpace, 10000, 0xFF);
   CheckError;
   // Set the acceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, vectorSpace, NIMC_ACCELERAT
   CheckError;
   // Set the deceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, vectorSpace, NIMC_DECELERAT
   CheckError;
   // Load Helical Arc
   err = flex_load_helical_arc (boardID, vectorSpace, 5000/*radius*/
```

```
720.0/*travelAngle*/, 5000 /*linear
CheckError;
// Start the move
err = flex_start(boardID, vectorSpace, 0);
CheckError;
do
  axisStatus = 0;
  // Check the move complete status
  err = flex_check_move_complete_status(boardID, vectorSpace, 0,
  CheckError;
  // Check the following error/axis off status for axis 1
  err = flex_read_axis_status_rtn(boardID, 1, &status);
  CheckError;
  axisStatus |= status;
  // Check the following error/axis off status for axis 2
  err = flex_read_axis_status_rtn(boardID, 2, &status);
  CheckError;
  axisStatus |= status;
  // Check the following error/axis off status for axis 3
  err = flex read axis status rtn(boardID, 3, &status);
  CheckError;
   axisStatus |= status;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG)
     err = csr & NIMC MODAL ERROR MSG;
     CheckError;
   }
}while (!moveComplete && !(axisStatus & NIMC_FOLLOWING_ERROR_BIT)
return; // Exit the Application
// Error Handling
//
nimcHandleError; //NIMCCATCHTHIS:
```

#### **Contoured Moves**

A contoured move moves an axis or a coordinate space of axes in a pattern that you define. The trajectory generator on the motion controller is not used during a contoured move. The controller takes position data in the form of an array, and splines the data before outputting it to the DACs or stepper outputs, as shown in the following figure.

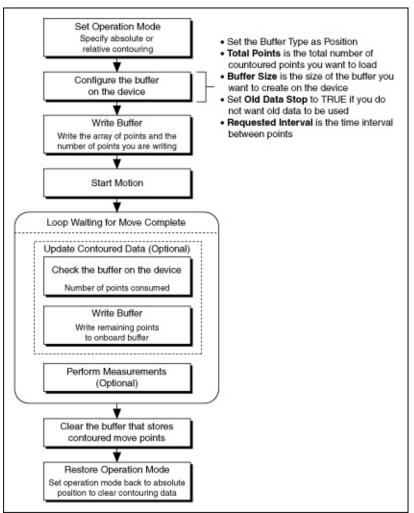


Contoured moves are useful when you want to generate a trajectory that cannot be constructed from straight lines and arcs. To ensure that the motion is smooth with minimum jerk, the motion controller creates intermediate points using a cubic spline algorithm.

The move constraints commonly used to limit other types of moves, such as maximum velocity, maximum acceleration, maximum deceleration, and maximum jerk, have no effect on contoured moves. However, the NI Motion Assistant prototyping tool can remap a user-defined trajectory based on specified move constraints, preserving move characteristics and move geometry.

#### **Contoured Move Algorithm**

The following figure shows a generic contoured move algorithm applicable to both C/C++ and LabVIEW code.



All contoured moves are relative, meaning motion starts from the position of the axis or axes at the time the contouring move starts. This behavior is similar to the way arc moves work. Depending on the operation mode you use, you can load absolute positions in the array or relative positions, which imply incremental position differences between contouring points.

#### **Absolute versus Relative Contouring**

All positions in a contouring buffer are relative to the current position when starting. There is an assumed 0 point that the firmware adds to the front of the buffer of points. For example, if the contour buffer is [10, 20, 30, 40], the positions are [0, 10, 20, 30, 40] in the firmware.

When a contour move starts it takes a snap shot of the current position according to the following equation:

StartPosition = currentPosition.

The start position is added to each point in the buffer to get the actual position to move through according to the following equation:

Point = StartPosition + bufferPosition[*n*].

If the current position is 100, and the buffer is [10, 20, 30, 40], the contour move follows these points: [100, 110, 120, 130, 140].

The difference between absolute contouring and relative contouring is how the points in the buffer are treated. The previous example was of an absolute contour move. A relative contour move treats the points as deltas according to the following formula:

Point[n] = Point[n-1] + bufferPosition[n]

For a relative contour move that starts at position 100 and includes a buffer with the following values: [10, 20, 30, 40], the points the contour move follows are [100, 110, 130, 160, 200].

For contoured moves, no two consecutive points can differ by more than  $2^{15}$  1. For absolute position mode, the first position in the array passed to the controller must be less than  $2^{15}$  1, and any two consecutive points must be less than  $2^{15}$  1. For relative position mode, no point passed to the controller can be greater than  $2^{15}$  1.

#### **Absolute versus Relative Contouring Example**

If an axis starts at position 0 and uses either of the following sets of contouring points, the axis ends up at position 28. If the axis starts at position 10, it ends up at position 38 in both cases.

1	3	6	10	14	18	22	25	27	28

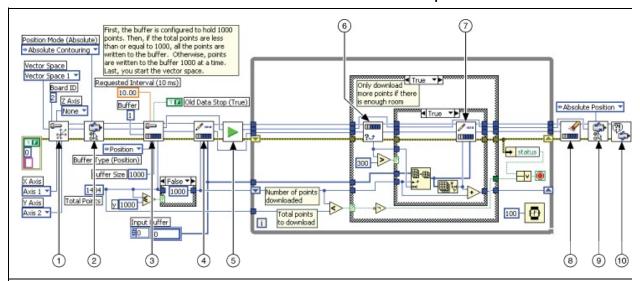
## 1 2 3 4 4 4 4 4 3 2 1

## **Contoured Move LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to complete a contoured move.

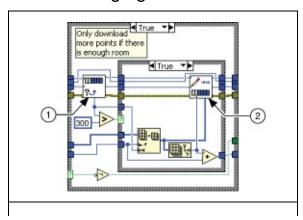


**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



1 Configure Vector 9 Clear Buffer 5 Start Motion Space 2 Set Operation 10 Set Operation 6 Read per Axis Status Mode Mode 3 Configure Buffer 7 Read per Axis Status 11 Motion Error Handler 8 Check Move Complete 4 Write Buffer Status

The following figure demonstrates the TRUE case of the example.



1 Check Buffer 2 Write Buffer

#### Contoured Move C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
    i32 i;
    i32 points[1994] = NIMC_SPIRAL_ARRAY; // Array of 2D points t
    132 points[1994] = NIMC_SPIRAL_ARRAY; // Array of 2D points t
u32 numPoints = 1994; // Total number of points to contour
i32 bufferSize = 1000; // The size of the buffer to allocate
f64 actualInterval; // The interval the controller can re
i32* downloadData = NULL; // The temporary array that is create
u32 currentDataPoint = 0; // Indicates the next point in the pc
i32 backlog; // Indicates the available space to a
u16 bufferState; // Indicates the state of the onboard
u32 pointsDone; // Indicates the number of points tha
u32 dataCopied = 0; // Keeps tack of the points copied
    // Variables for modal error handling
    u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
    // Set the board ID
    boardID = 1;
    // Set the axis number
    vectorSpace = NIMC_VECTOR_SPACE1;
    // Configure a 2D Vector Space comprising of axes 1 and 2
    err = flex_config_vect_spc(boardID, vectorSpace, 1, 2, 0);
    CheckError;
    // Set the operation mode to absolute position
    err = flex_set_op_mode(boardID, vectorSpace, NIMC_ABSOLUTE_CONTOU
    CheckError;
```

```
// Configure buffer on motion controller memory (RAM)
// Note requested time interval is hardcoded to 10 milliseconds
err = flex_configure_buffer(boardID, 1 /*buffer number*/, vectorS
CheckError;
// Send the first 1000 points of the data
downloadData = malloc(sizeof(i32)*bufferSize);
for(i=0;i<bufferSize;i++){</pre>
   downloadData[i] = points[i];
   currentDataPoint++;
}
err = flex_write_buffer(boardID, 1/*buffer number*/, bufferSize,
free(downloadData);
downloadData = NULL;
CheckError;
// Start Motion
err = flex_start(boardID, vectorSpace, 0);
CheckError;
for(;;){
   axisStatus = 0;
   // Check for available space and download remaining points eve
   Sleep(50);
   // Check to see if we have more points to download
   if(currentDataPoint < numPoints){</pre>
      err = flex check buffer_rtn(boardID, 1/*buffer number*/, &b
      CheckError;
      if(backlog >= 300){
         downloadData = malloc(sizeof(i32)*backlog);
         dataCopied = 0;
         for(i=0;i<backlog;i++){</pre>
            if(currentDataPoint > numPoints) break;
            downloadData[i] = points[currentDataPoint];
            currentDataPoint++;
            dataCopied++;
         err = flex_write_buffer (boardID, 1 /*buffer number*/, d
         free(downloadData);
         downloadData = NULL;
         CheckError;
      }
   }
```

```
// Check the move complete status
  err = flex check move complete status(boardID, vectorSpace, 0,
  CheckError;
   if(moveComplete) break;
  // Check for axis off status/following error or any modal erro
       Read the communication status register and check the moda
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG){
     err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
  }
  // Check the motor off status on all the axes or axis
  err = flex read axis status rtn(boardID, 1, &status);
  CheckError;
  axisStatus |= status;
  err = flex read axis status rtn(boardID, 2, &status);
  CheckError;
  axisStatus |= status;
  if( (axisStatus & NIMC_FOLLOWING_ERROR_BIT) || (axisStatus & N
     break; // Break out of the for loop as an axis was killed
  }
}
// Set the mode back to absolute mode to get the controller out o
// mode
err = flex_set_op_mode(boardID, vectorSpace, NIMC_ABSOLUTE_POSITI
CheckError;
// Free the buffer allocated on the controller memory
err = flex_clear_buffer(boardID, 1/*buffer number*/);
CheckError:
return; // Exit the Application
// Error Handling
//
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC MODAL ERROR MSG){
```

```
do{
    // Get the command ID, resource and the error code of the m
    // error from the error stack on the board
    flex_read_error_msg_rtn(boardID,&commandID,&resourceID,&err
        nimcDisplayError(errorCode,commandID,resourceID);

    // Read the Communication Status Register
    flex_read_csr_rtn(boardID,&csr);

} while(csr & NIMC_MODAL_ERROR_MSG);
} else    // Display regular error
    nimcDisplayError(err,0,0);
    return;    // Exit the Application
}
```

#### **Reference Moves**

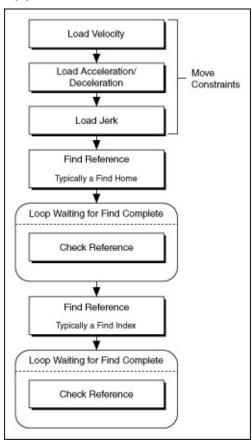
Use reference moves to move the axes to a known starting location and orientation. Reference functions include the Find Reference VI or function, the Check Reference VI or function, the Wait Reference VI or function, the Read Reference Status VI or function, the Load Reference Parameters VI or function, and the Get Reference Parameters VI or function.

Use Check Reference to determine if the Find Reference operation is complete. This function is often placed in a loop that continues until the status for the Find Reference operation is shown to be complete. You can use Wait Reference if there is no need to monitor the status of the Find Reference.

Use a Find Reference move to initiate a search operation to find a reference position. Available search operations include <a href="https://www.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new.nobe.new

## **Reference Move Algorithm**

The following figure shows a generic reference move algorithm applicable to both C/C++ and LabVIEW code.



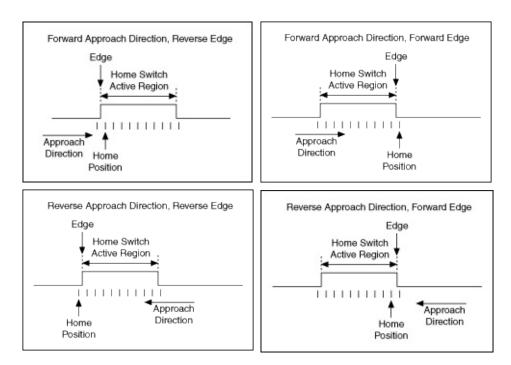
### **Find Home**

Setting the Find Reference search type to Home initiates a search sequence for a home input or other sensor hooked up to the home input. Attributes you can specify include the initial search direction, the edge (forward or reverse) of the home signal to stop on, and the desired travel direction when you approach the specified home edge.

The search is performed at the maximum find velocity specified by Max Find Velocity or Max Find RPM. The velocity of the axis is set to this value during the find operation. The velocity is then restored to its previous value after the search completes.

When the initial search direction is forward, the axis starts moving in the forward direction. If the specified home signal transition is detected, the find home sequence continues based on the other control bits. If the forward limit switch is encountered before the home input, the axis automatically reverses direction and continues searching for the home input. Then, if the reverse limit is encountered before the home input, the sequence stops and the Home Found status is False. If a home input exists, finding it is guaranteed. A similar search sequence is followed when the initial search direction is reverse, except the search starts in the reverse direction.

You can configure the find home sequence to detect either the rising or falling edge of the home signal, as shown in the following figures. You also can set the active state of the limit and home inputs with the Set Limit Input Polarity VI or function and Set Home Input Polarity VI or function, respectively. After the home input is found, motion approaches the home edge from the specified direction. If necessary, the axis travels past the home edge and reverses direction to approach it from the programmed direction. To smoothly approach the edge, this portion of the sequence is executed at a slower velocity using the **Approach Velocity Percent** value set with Load Reference Parameter VI or function or in MAX. This approach direction feature is used to minimize the effects of motion system windup, backlash, and/or home sensor hysteresis.



The forward edge is the edge of the home switch that has a position higher than that of the reverse edge.

The Wait Reference VI or function reports the success of a Find Home. Wait Reference also can be used to query if the find operation is complete by using a timeout of 0. You also can monitor this status with the Read Reference Status VI or function and Read per Axis Status VI or function. When the home input is found, the Home Found status is set to True. If the home sequence fails to locate the specified edge of the home signal, the Home Found status is False.



**Note** National Instruments recommends not connecting the home input to either the forward or reverse limits. This might cause inappropriate behavior. Instead use Find Forward or Find Reverse. Consider using Find Center for stages if your goal is to center the system to the center of travel.

If you execute a Find Home on systems without a home input, the sequence always terminates at the limit switch opposite to the initial search direction. The Home Found status is false and the edge and approach direction features are not applicable. Consider using <a href="Find">Find</a> <a href="Forward and Reverse Limits">Forward and Reverse Limits</a> instead.

If Smart Enable is disabled you must enable both limits and the home input prior to executing the Find Home. If any of the limit or home inputs

are disabled on the axis, the Find Home does not start and a modal error is generated. Also, after a Find Home sequence completes, the home input must be disabled because it is no longer required, assuming you do not want to stop on it the next time the system moves past it.

With the Smart Enable feature enabled both limits and the home input are automatically enabled prior to initiating the search. After the search completes, the limits and the home input are returned to their previous states. If you use Smart Enable, National Instruments recommends setting the default in MAX to enable the forward and reverse limits and disable the home input.

After the home input is found, an offset move is performed, if specified. Offset Move specifies the amount and direction of the offset. This offset move is performed at the maximum find velocity specified by Max Find Velocity or Max Find RPM. By default Offset Move is 0, resulting in no offset move.



**Note** If you are doing an offset move after a Find Home, verify that you set the correct Edge to Stop On using the Load Reference Parameter VI or function so that the offset move does not cross the home switch.

After the offset move is performed, the position is reset if Reset Position? is true. The position is set to the values indicated by Reset Primary Position and Reset Secondary Position. This procedure establishes a repeatable reference position that is as accurate as the home edge location.

In closed-loop encoder based systems, you can perform a <u>Find Index</u> after a Find Home to overcome any errors in the home edge location. The Find Index operation allows you to get a fixed encoder position relative to your home input.

#### **Example 1**

You want to find the forward edge of the home input on axis 2 and approach it from the reverse direction. You also want to perform an offset of 500 counts in the forward direction and then reset the position to 0. In addition, you want to begin searching in the reverse direction. First, the settings in MAX must correspond to the following:

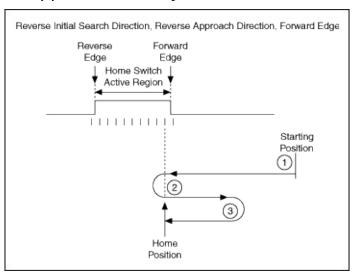
Final Search Edge Smart Reset Offset Reset R

Direction	Direction		Enable	Position?		Primary Position	
True	True	False	True	True	500	0	0
(Reverse)	(Reverse)	(Forward)					

Then, to start the search, call Find Reference with the following parameters:

```
axisOrVectorSpace = Axis 2 (using LabVIEW)/NIMC_AXIS2
(using C)
searchType = Find Home (0)
axisOrVSMap = Unused
```

As shown in the following figure, this Find Home sequence searches for the home input at the Max Find Velocity from the reverse direction (1). When the home input is found, the axis decelerates and repositions to the desired approach direction (2). Then, the forward edge is approached at the Approach Velocity in the reverse direction (3).



#### **Example 2**

You want to find the forward edge of the home input on axis 2 and approach it from the forward direction. You also want to perform an offset of 500 counts in the reverse direction and then reset the position to 0. In addition, you want to begin searching in the reverse direction. First, the settings in MAX must correspond to the following:

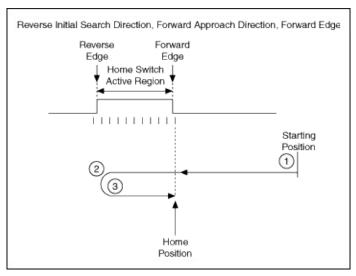
Final	Search	Edge	<b>Smart</b>	Reset	Offset	Reset	R
Direction	Direction		<b>Enable</b>	Position?		<b>Primary</b>	S
						<b>Position</b>	P

False True False True —500 0 (Forward) (Reverse) (Forward)

Then, to start the search, call Find Reference with the following parameters:

```
axisOrVectorSpace = Axis 2 (using LabVIEW)/NIMC_AXIS2
(using C)
searchType = Home (0)
axisOrVSMap = Unused
```

As shown in the following figure, this Find Home sequence searches for the home input from the reverse direction (1). When the home input is found, the axis decelerates (2). Then the forward edge is approached at the Approach Velocity in the forward direction (3).



#### **Example 3**

You want to find the reverse edge of the home input on axis 2 and approach it from the reverse direction. In addition, you want to begin searching in the reverse direction. First, the settings in MAX must correspond to the following:

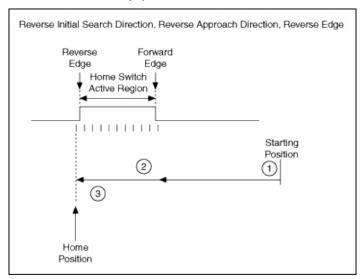
	Search Direction	•	Reset Position?		Reset Primary Position	S
True (Reverse)		True (Reverse)	 False	0	0	0

Then, to start the search, call Find Reference with the following

#### parameters:

```
axisOrVectorSpace = Axis 2 (using LabVIEW)/NIMC_AXIS2
(using C)
searchType = Home (0)
axisOrVSMap = Unused
```

As shown in the following figure, this Find Home sequence searches for the home input from the reverse direction (1). When the forward edge of the home input is found, the axis decelerates to the Approach Velocity (2). Then the reverse edge is approached at the Approach Velocity in the reverse direction (3).



#### **Example 4**

You want to find the reverse edge of the home input on axis 2 and approach it from the forward direction. In addition, you want to begin searching in the reverse direction. First, the settings in MAX must correspond to the following:

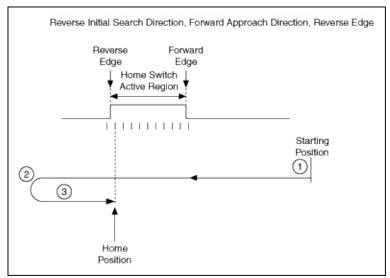
Search Direction	•	Reset Position?		Reset Primary Position	S
 	True (Reverse)	 False	0	0	0

Then, to start the search, call Find Reference with the following parameters:

axisOrVectorSpace = Axis 2 (using LabVIEW)/NIMC\_AXIS2

(using C)
searchType = Home (0)
axisOrVSMap = Unused

As shown in the following figure, this Find Home sequence searches for the home input from the reverse direction (1). When the reverse edge of the home input is found, the axis decelerates (2). Then the reverse edge is approached at the Approach Velocity in the forward direction (3).





**Note** When the initial search direction is forward, the Find Home behavior is the reverse of the examples shown.

### **Find Index**

Setting the Find Reference search type to Index initiates a search sequence to find the index (marker) signal of the feedback encoder. Attributes you can specify include the initial search direction, approach velocity percent, and search distance.

The encoder index signal is accurate to one quadrature count and provides a much more repeatable reference than using just a limit or home input edge. Find Index is typically executed after a Find Home, Find Center, or Find Limit operation. With this procedure, the home input need only be accurate enough to repeatedly locate the axis within the same encoder revolution or index period. Then you can use Find Index to find a unique instance of the index.



**Note** Find Index is only available on axes with incremental encoder feedback.

When performing a Find Index operation after another find, you might notice inconsistent operation if the index (marker) signal is near to the edge of the home input or limit. In this case, a Find Index operation can sometimes be off by one revolution. To solve this, National Instruments recommends performing an offset move on the previous find to move away from the index (marker) signal, and then perform the Find Index operation.

The search sequence is performed in the specified direction at a fixed low velocity of ¼ RPS unless the maximum find velocity is slower. To guarantee finding the index (if one exists), the length of the move is automatically set to slightly greater than one encoder revolution. If you previously performed a Find Forward Limit or a Find Reverse Limit National Instruments recommends you set the search direction of the Find Index to a direction opposite to the direction of travel into the limit.

Find Index uses the **Approach Velocity Percent** value set with the Load Reference Parameter VI or <u>function</u> or in MAX throughout the entire search sequence.



**Note** If the velocity of the Find Index operation is very high, it is possible the search distance is reported complete before index found reported complete. This results in Index Found status reported as false. To avoid this you can either increase the search

distance to allow more time to report the index found and/or decrease the search speed to slow down the search.

In normal operation, Find Index uses high-speed capture circuitry to detect the index. The axis typically overshoots by an amount that varies depending on the velocity, then comes back to the index. In most cases this is not noticeable.



**Note** If you are using a linear encoder or if your counts/steps per revolution is set to a value other than the actual counts/steps per revolution, use the **Search Distance** parameter set with Load Reference Parameter or in MAX to load a custom search distance for a Find Index move.

The Wait Reference VI or function reports the success of a Find Index. Wait Reference also can be used to query if the find operation is complete using a timeout of 0. You also can monitor this status with the Read per Axis Status VI or function. When the index is found the Index Found status is set to True. If the index is not found during the search revolution, the axis comes to a stop and indicates the failure by setting the Index Found status to False. Missing the index is possible for a number of reasons including an incorrectly connected encoder or an incorrect value for counts per revolution. Refer to the motion controller documentation for more information about encoder connections and index phasing.

You can only execute a Find Index on properly configured axes that are presently stopped or killed. Attempting to execute a Find Index while the axis is in motion generates a modal error. Refer to the *Errors and Error Handling* section of the NI-Motion VI Help or NI-Motion Function Help for more information about errors.

If Smart Enable is disabled National Instruments recommends you disable the home input before performing a Find Index. If you previously performed a Find Home, depending on the settings, you might encounter the home input during a Find Index.

With the Smart Enable feature enabled both limits are enabled and the home input is disabled prior to initiating the search. After the search completes the limits are returned to their previous states. If you use Smart Enable, National Instruments recommends setting the default in MAX to enable the forward and reverse limits and disable the home input.

After the index (marker) signal is found, an offset move is performed. Offset Move specifies the amount and direction of the offset. This offset move is performed at the maximum find velocity specified by Max Find Velocity or Max Find RPM. By default Offset Move is 0, resulting in no offset move.

After the offset move is performed, the position is reset if **Enable Reset Position** is true. The position is set to the values indicated by **Primary Reset Position** and **Secondary Reset Position**.

### **Find Forward and Reverse Limits**

Setting the Find Reference search type to Forward Limit initiates a search sequence to find the forward limit. Setting the search type to Reverse Limit initiates a search sequence to find the reverse limit. Attributes you can specify include the Initial Search Direction and the Final Approach Direction (for when the search encounters each of the limits).

The search is performed at the maximum find velocity specified by Max Find Velocity or Max Find RPM. The velocity of the axis is set to this value during the find operation. The velocity is then restored to its previous value after the search completes.

When the search operation initiates the axis starts moving in the direction of the limit. When the axis encounters the limit in that direction the search completes and the Wait Reference VI or function returns a true value.

A fine-tuning operation is performed when the limit is reached. This provides for a more accurate search. To smoothly approach the edge, this portion of the sequence is executed at a fixed low velocity of 1/4 RPS unless the maximum find velocity is slower. This approach direction feature is used to minimize the effects of motion system windup, backlash, and/or home sensor hysteresis. If the final approach direction is the opposite to the direction of travel into the limit (true), the axis then slowly backs off the limit until the limit becomes inactive again. If the final approach direction is the same as the direction of travel into limit (false), then the axis slowly backs off the limit and re-approaches it. You also can set the active state of the limit with the Set Limit Input Polarity  $\[ \bigvee \]$  or function.

If Smart Enable is disabled National Instruments recommends you disable the home input before performing a Find Forward Limit or Find Reverse Limit. If the home input is enabled and encountered during a Find Center, the search stops at the home input and generates a modal error.

With the Smart Enable feature enabled the limit you are searching for is enabled, the opposite limit is left as-is, and home input is disabled prior to initiating the search. After the search completes, the limits are returned to their previous states. If you use Smart Enable, National Instruments

recommends setting the default in MAX to enable the forward and reverse limits and disable the home input.

After the limit is located, an offset move is performed. Offset Move specifies the amount and direction of the offset. This offset move is performed at the maximum find velocity specified by Max Find Velocity or Max Find RPM. Usually the Offset Move is negative for a Find Forward Limit and positive for a Find Reverse Limit. By default Offset Move is 0, resulting in no offset move.

After the offset move is performed, the position is reset if Reset Position? is true. The position is set to the values indicated by Reset Primary Position and Reset Secondary Position.

### **Find Center**

Setting the Find Reference search type to Center initiates a search sequence positioning the axis in the middle of the forward and reverse limits. Attributes you can specify include the Initial Search Direction and the Final Approach Direction (for when the search encounters each of the limits).

The search is performed at the maximum find velocity specified by Max Find Velocity or Max Find RPM. The velocity of the axis is set to this value during the find operation. The velocity is then restored to its previous value after the search completes.

When the search direction is forward, the axis starts moving in the forward direction. When the limit in that direction is encountered, the direction is reversed to find the reverse limit. After both limits are reached a center position can be calculated from the recorded positions. The axis then proceeds to the calculated center. When the center is found the Wait Reference VI or function returns a true value. A similar search sequence starting in the reverse direction is followed when the initial search direction is reverse.

At the end of each limit of travel, a fine-tuning operation is performed. This provides for a more accurate search. To smoothly approach the edge, this portion of the sequence is executed at a fixed low velocity of 1/4 RPS unless the maximum find velocity is slower. This approach direction feature is used to minimize the effects of motion system windup, backlash, and/or home sensor hysteresis. If the final approach direction is opposite to the direction of travel into the limit (true), the axis slowly backs off the limit until the limit becomes inactive again. If the final approach direction is the same as the direction of travel into limit (false), the axis slowly backs off the limit and re-approaches it. You also can set the active state of the limits in MAX or with the Set Limit Input Polarity VI or function.

If Smart Enable is disabled National Instruments recommends you disable the home input before performing a Find Center. If the home input is enabled and encountered during a Find Center, the search stops at the home input and generates a modal error.

With the Smart Enable feature enabled both limits are enabled and the home input is disabled prior to initiating the search. After the search completes the limits are returned to their previous states. If you use Smart Enable, National Instruments recommends setting the default in MAX to enable the forward and reverse limits and disable the home input.

After the center is located, an offset move is performed. Offset Move specifies the amount and direction of the offset. This offset move is performed at the maximum find velocity specified by Max Find Velocity or Max Find RPM. By default Offset Move is 0, resulting in no offset move.

After the offset move is performed, the position is reset if Reset Position? is true. The position is set to the values indicated by Reset Primary Position and Reset Secondary Position.

### **Run Sequence**

Setting the Find Reference search type to Run Sequence initiates a predefined search across multiple axes and multiple find types. The sequence run is specified in MAX. You also can use the Load Reference Parameter VI or function to programmatically set the Sequence Order for a given axis and search type.

The Run Sequence option allows you to pre-configure a search sequence. That sequence can then be run with a single function. The sequence gets saved when you save user defaults to Flash (ROM).

To specify a run sequence programmatically, you can use Load Reference Parameter to set the Sequence Order for a given axis and search type. The Sequence Order refers to the phase a particular search is in. The default Sequence Order 0 indicates that a particular search type is not included in the sequence.



**Note** Only one search type on a given axis can have the same Sequence Order number. If you set the Find Home Sequence Order of 2, then set the Find Index Sequence Order to 2, Find Home Sequence Order resets to 0 and Find Index Sequence Order is 2. No error is reported if this is the case.

When specifying Sequence Orders you must be sure not to skip over a number, all phases must be contiguous. Any finds specified after the gap is not executed. For example, if you specify a Find Center Sequence Order of 1 and a Find Index Sequence Order of 3, only the Find Center is executed.

### **Example**

Suppose you have a stage configured with its x-axis tied to axis 2, its y-axis tied to axis 3, and another device attached to the stage tied to axis 1. If you want to make sure you center the device attached to the stage before you centered the stage you might set up a run sequence similar to the following:

	Phase 1	Wait	Phase 2	Wait	Phase 3
Axis 1	Find Center		Find Index		
Axis 2			Find Center		Find Index

Axis 3		Find Center	Find Index
--------	--	-------------	------------

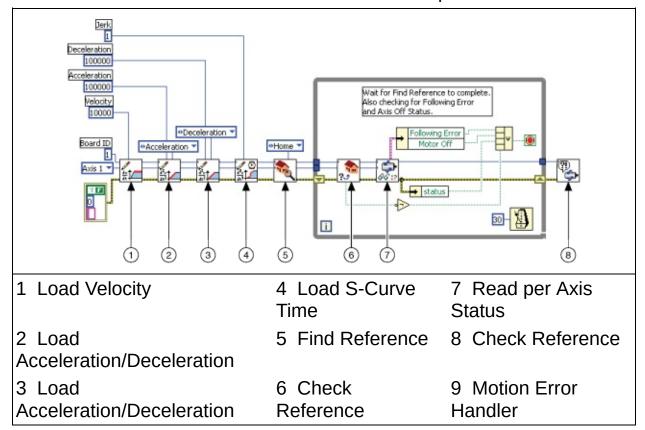
The sequence is specified as different phases. The next phase does not start until all find operations in a given phase are complete. In this example, Phase 1 is the centering of axis 1; Phase 2 consists of finding the index of axis 1, and centering axes 2 and 3; Phase 3 consists of finding the index of axes 2 and 3. For this example you would set Find Center Sequence Order on Axis 1 to 1, Find Index Sequence Order on Axis 1 to 2, Find Center Sequence Order on Axes 2 and 3 to 2, and Find Index Sequence Order on Axes 2 and 3 to 3.

### **Reference Move LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to complete a reference move.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



### Reference Move C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void){
                                         // Board identification number
    u8 boardID;
   u8 boardID;
u8 axis;
f64 acceleration =100;
f64 velocity =200;
u16 found, finding;
u16 csr=0;
i32 position;
i32 scanVar;
// Board luentITICation number
// Axis number
// Acceleration value in RPS/S
// Velocity value in RPM
// Check Reference Statuses
// Axis status
// Communication status Register
// Current position of axis
// Scan variable to read in values no
                                                                   // by the scanf func
    // Variables for modal error handling
                           // The commandID of the function
// The resource ID
// Error code
    u16 commandID;
   u16 resourceID;
    i32 errorCode;
    // Get the board ID
    printf("Enter the Board ID: ");
    scanf("%d", &scanVar);
    boardID=(u8)scanVar;
    // Check if the board is at power up reset condition
    err = flex_read_csr_rtn(boardID, &csr);
    CheckError;
     if (csr & NIMC_POWER_UP_RESET ){
        printf("\nThe FlexMotion board is in the reset condition. Plea
        printf("running this example. The \"flex initialize controlle
        printf("board with settings selected through Measurement and A
        return;
    }
    // Get the axis number
    printf("Enter the axis: ");
    scanf("%d",&scanVar);
    axis=(u8)scanVar;
    // Flush the Stdin
```

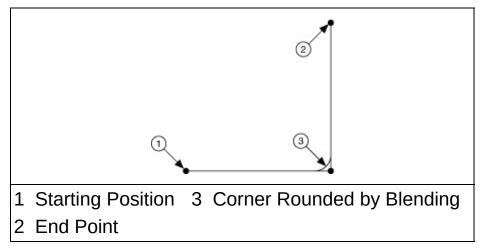
```
fflush(stdin);
// Load acceleration and deceleration to the axis selected
err = flex load rpsps(boardID, axis, NIMC BOTH, acceleration, 0xF
CheckError;
// Load velocity to the axis selected
err = flex_load_rpm(boardID, axis, velocity, 0xFF);
CheckError;
// Start the Find Reference move
err = flex_find_reference(boardID, axis, 0, NIMC_FIND_HOME_REFERE
CheckError;
// Wait for find reference to complete on the axis AND
// also check for modal errors at the same time
do{
   // Read the current position of axis
   err = flex_read_pos_rtn(boardID, axis, &position);
   CheckError;
   // Display the current position of axis
   printf("\rAxis %d position: %10d", axis, position);
  // Check if the axis has stopped b/c of axis off or following
   err = flex_read_axis_status_rtn(boardID, axis, &axisStatus);
   // Check if the reference has finished finding
   err = flex_check_reference(boardID, axis, 0, &found, &finding)
   CheckError;
   // Read the Communication Status Register - check the
   // modal error bit
   err = flex_read_csr_rtn(boardID, &csr);
   CheckError;
   if (csr & NIMC_MODAL_ERROR_MSG)
      flex_stop_motion(boardID, NIMC_AXIS1, NIMC_DECEL_STOP, 0);//
      err = csr & NIMC_MODAL_ERROR_MSG;
      CheckError;
   }
// test for find reference complete, following error, or axis off
}while ( !(axisStatus & (NIMC FOLLOWING ERROR BIT | NIMC AXIS OFF
printf("\nAxis %d position: %10d", axis, position);
if (found)
   printf("\rAxis found reference");
else
   printf("\rAxis did not find reference");
```

```
printf("\n\nFinished.\n");
              // Exit the Application
  return;
  // Error Handling
  //
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC_MODAL_ERROR_MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID, &csr);
     }while(csr & NIMC_MODAL_ERROR_MSG);
            // Display regular error
  else
     nimcDisplayError(err, 0, 0);
  return; // Exit the Application
}
```

### **Blending**

Blending, also called velocity blending, superimposes the velocity profiles of two moves to maintain continuous motion. Blending is useful when continuous motion between concatenated move segments is important. Examples of some applications that can use blending are scanning, welding, inspection, and fluid dispensing.

Blending must occur on velocity profiles of two move segments, so the end positions of each move segment may or may not be reached. For example, if you are blending two straight-line moves that form a 90° angle, the blended move must round the corner to make the move continuous. In this case, the move never reaches the exact position where the two straight lines meet, but instead follows the rounded corner, as shown in the following figure.



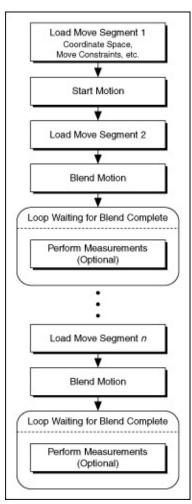
Motion controllers can perform blending between two straight-line moves, between two arc moves, or between straight-line and arc moves. Blending does not work for reference and contoured moves.

Because blending occurs on velocity profiles, the effect of reaching the end positions of the move segments and the maximum velocity depends on the velocity, acceleration, deceleration, and jerk loaded for the two move segments.

Because two move segments are always used while blending, it is very important that you wait for the blend to complete before loading the next move segment you want to blend.

### **Blending Algorithm**

The following figure shows a generic blending algorithm applicable to both C/C++ and LabVIEW code.



### **Blend Factors**

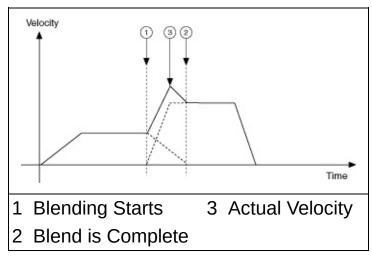
There are three ways you can start the second move in a blend:

- Superimpose the two moves by starting the second move as the first move starts to decelerate
- Start the second move after the first profile is complete
- Start the second move after the first profile is complete and the added delay time has elapsed

Refer to **Move Profiles** for more information about move profiles.

## **Superimpose Two Moves**

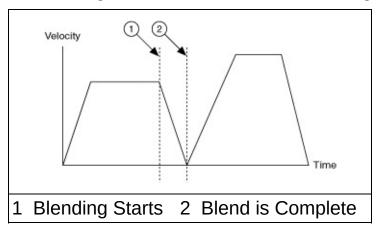
Superimposing two moves is the most common use of blending. In this case, the motion controller tries to maintain continuous motion by superimposing the two move segments such that the second move segment starts its profile while the first move is decelerating, as shown in the following figure.



The velocity during the superimposition depends on the cruising velocity, deceleration, and jerk of the first move segment, and the jerk, acceleration, and cruising velocity of the second move segment.

# **Blend after First Move Is Complete**

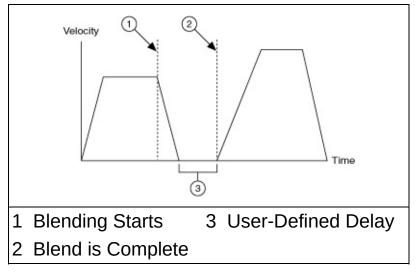
Blending moves after the first move is complete causes the first move segment to come to a complete stop before starting the profile of the second segment, as shown in the following figure.



This type of blending is useful if you want to start two move segments, one after the other, with no delay between them.

# **Blend after Delay**

You can blend two moves after a delay at the end of the first move, as shown in the following figure.



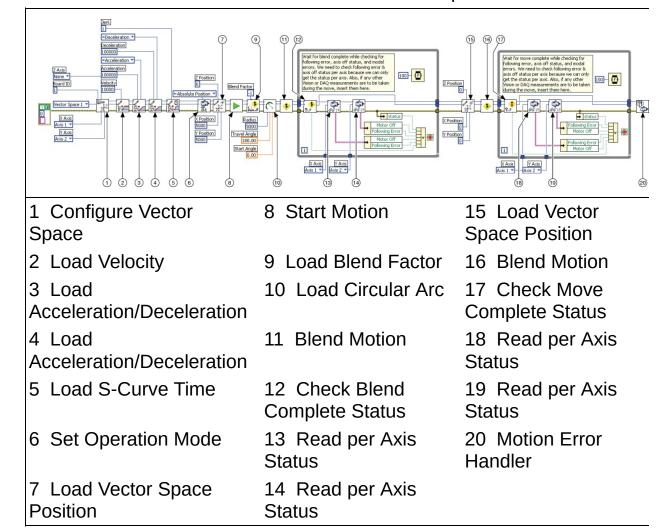
Blending in this manner is useful if you want to start two move segments after a deterministic delay. The two move segments can be either straight-line moves or arc moves.

## **Blended Move LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to complete a blended move.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



### Blended Move C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main Function
void main(void){
   u8 boardID;  // Board identification number
u8 vectorSpace;  // Vector space number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
   u16 status;
   u16 complete; // Move or blend complete status
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID= 1;
   // Set the axis number
   vectorSpace = NIMC_VECTOR_SPACE1;
   // Configure a 2D Vector Space comprising of axes 1, and 2
   err = flex_config_vect_spc(boardID, vectorSpace, 1, 2, 0);
   CheckError;
   // Set the velocity for the move (in counts/sec)
   err = flex_load_velocity(boardID, vectorSpace, 10000, 0xFF);
   CheckError;
   // Set the acceleration for the move (in counts/sec^2)
   err = flex load acceleration(boardID, vectorSpace, NIMC ACCELERAT
   CheckError;
   // Set the deceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, vectorSpace, NIMC_DECELERAT
   CheckError;
   // Set the jerk or scurve in sample periods
   err = flex_load_scurve_time(boardID, vectorSpace, 1, 0xFF);
   CheckError;
```

```
// Set the operation mode to absolute position
err = flex_set_op_mode(boardID, vectorSpace, NIMC_ABSOLUTE_POSITI
CheckError;
// Load the first straight line segments to position 5000, 5000
err = flex load vs pos(boardID, vectorSpace, 5000, 5000, 0, 0xFF
CheckError;
// Start the move
err = flex_start(boardID, vectorSpace, 0);
CheckError;
// Load Circular Arc - making a semicircle in the counter clockwi
err = flex_load_circular_arc (boardID, vectorSpace, 5000/*radius*
CheckError;
// Blend the move
err = flex_blend(boardID, vectorSpace, 0);
CheckError;
// Wait for blend to complete before loading the next segment
do
{
  axisStatus = 0;
   // Check the blend complete status
   err = flex_check_blend_complete_status(boardID, vectorSpace, @
   CheckError;
   // Check the following error/axis off status for axis 1
   err = flex read axis status rtn(boardID, 1, &status);
   CheckError;
   axisStatus |= status;
   // Check the following error/axis off status for axis 2
   err = flex read axis status rtn(boardID, 2, &status);
   CheckError;
   axisStatus |= status;
   // Read the communication status register and check the modal
   err = flex_read_csr_rtn(boardID, &csr);
   CheckError;
   // Check the modal errors
   if (csr & NIMC MODAL ERROR MSG)
   {
      err = csr & NIMC_MODAL_ERROR_MSG;
      CheckError;
   }
```

```
Sleep(50); // Check every 50 ms
}while (!complete && !(axisStatus & NIMC_FOLLOWING_ERROR_BIT) &&
// Load the final straightline segments to position 0, 0
err = flex load vs pos(boardID, vectorSpace, 0, 0, 0, 0xFF);
CheckError;
// Wait for move to complete as this is the final segment
do
{
  axisStatus = 0;
  // Check the move complete status
  err = flex_check_move_complete_status(boardID, vectorSpace, 0,
  CheckError
  // Check the following error/axis off status for axis 1
  err = flex_read_axis_status_rtn(boardID, 1, &status);
  CheckError;
  axisStatus |= status;
  // Check the following error/axis off status for axis 2
  err = flex_read_axis_status_rtn(boardID, 2, &status);
  CheckError:
  axisStatus |= status;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG)
     err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
  Sleep(50); // Check every 50 ms
}while (!complete && !(axisStatus & NIMC_FOLLOWING_ERROR_BIT) &&
return; // Exit the Application
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
```

```
// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG) {
    do{
        // Get the command ID, resource and the error code of the m.
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID,&commandID,&resourceID,&err
        nimcDisplayError(errorCode,commandID,resourceID);

    // Read the Communication Status Register
        flex_read_csr_rtn(boardID,&csr);

    }while(csr & NIMC_MODAL_ERROR_MSG);
}
else    // Display regular error
    nimcDisplayError(err,0,0);
return;    // Exit the Application
}
```

# **Electronic Gearing and Camming**

Use electronic gearing or camming to synchronize the movement of one or more slave axes to the movement of a master device, which can be an encoder, ADC, or the trajectory of another axis. The movement of the slave axes may be at a higher or lower gear ratio than the master. For example, every turn of the master axis may cause a slave axis to turn twice.

### Gearing

Electronic gearing allows one slave motor to be driven in proportion to a master motor or feedback sensor, such as an encoder or torque (analog) sensor.

As the slave follows the master position at a constant ratio, the effect is similar to that of two axes mechanically geared.

Electronic gearing has several advantages over mechanical gears. The most notable is flexibility because you can change gear ratios on-the-fly. The other major advantage to electronic gearing is that you can superimpose a move over a geared axis. The superimposed move is added to the geared profile of the slave axis, which allows the slave axis to be synchronized on-the-fly.

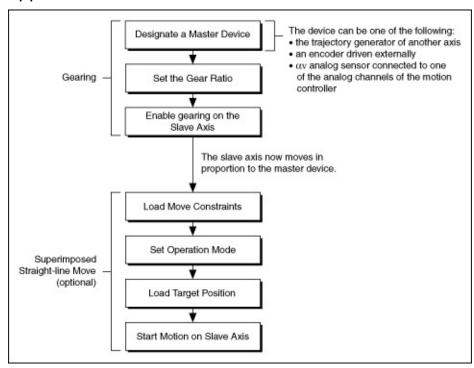
An axis can be geared to another axis, or to an encoder or ADC. When you gear an axis to another axis, the slave axis follows the trajectory generation of the master axis. For example, if you manually move the master axis, the slave axis does not move because the trajectory generator of the master axis is not active.

When you gear an axis to an encoder, or feedback device, the slave axis follows the feedback generated by the encoder. If the encoder detects movement, the slave moves proportionally to information returned by the encoder. For example, if you twist the master axis connected to the encoder, the slave axis also turns because it is using the position information gathered by the encoder.

When an axis is geared to an ADC, the slave axis follows the binary value of the ADC as if it were a position. For example, if the binary code for 2 V is 6553, the slave axis tracks to this position.

### **Electronic Gearing Algorithm**

The following figure shows a generic electronic gearing algorithm applicable to both C/C++ and LabVIEW code.



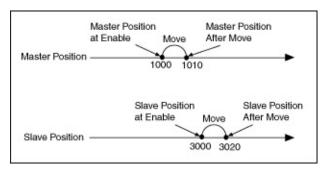
### **Absolute versus Relative Gearing**

The gear ratio is used to determine how far the slave axis must move in proportion to the master when gearing is enabled. The gear ratio can be absolute or relative.

Slave axis move = Master axis position × Gear ratio

#### **Relative Gearing**

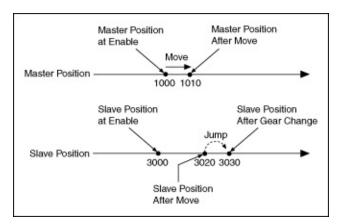
Relative gearing allows you to change the gear ratio on-the-fly. The master move is calculated based on the master reference position, which is updated when gearing is enabled and is updated each time a new gear ratio is loaded. For example, if you have a gearing ratio of 2:1 (slave:master), the slave moves 20 counts when the master device moves 10 counts.



#### **Absolute Gearing**

Absolute gearing behaves similarly to relative gearing in that when gearing is enabled, the slave axis follows the master axis movement as it is defined by the gear ratio. The difference between relative and absolute gearing is that the reference position calculated for the master axis is updated only when gearing is enabled. This difference is apparent when the gear ratio is updated on-the-fly.

For example, if the gear ratio is 2:1, the current master position is 1010, the current slave position is 3020, and the gear ratio is changed to 3:1, the slave axis jumps from 3020 to 3030 but the master position remains the same.



Changing a gear ratio on-the-fly during absolute gearing allows you to quickly synchronize the slave axis with the master axis.



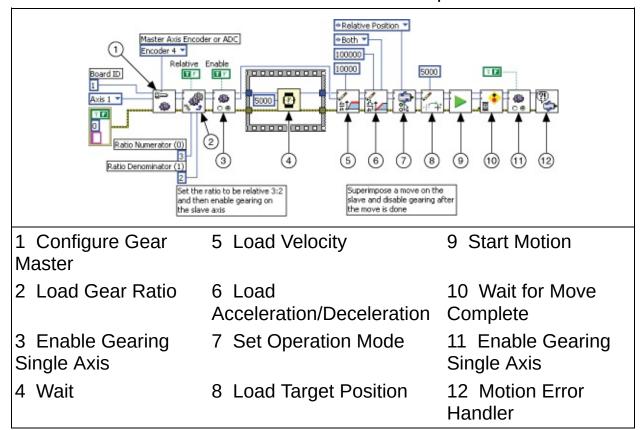
**Note** When the gear ratio is changed on-the-fly, the slave axis moves at full torque to the new position.

#### **Electronic Gearing LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to perform electronic gearing using an encoder as the gear master.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



#### **Electronic Gearing C/C++ Code**

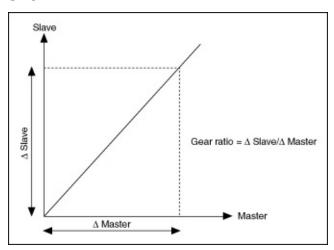
The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u8 boardID;  // Board identification number
u8 slaveAxis;  // Slave axis number
u8 master;  // Gear master
u16 csr = 0;  // Communication status register
  u16 moveComplete;
  // Variables for modal error handling
  u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
  // Set the board ID
  boardID= 1;
  // Set the axis number
  slaveAxis = 1:
  // Master is encoder 4
  master = NIMC ENCODER4;
  //-----
  // Set up the gearing configuration for the slave axis
  // Configure Gear Master
  err = flex_config_gear_master(boardID, slaveAxis, master);
  CheckError;
       Load Gear Ratio 3:2
  err = flex_load_gear_ratio(boardID, slaveAxis, NIMC_RELATIVE_GEA
                            3/* ratioNumerator*/, 2/* ratioDenomin
  CheckError;
  //-----
  // Enable Gearing on slave axis
  //-----
  err = flex_enable_gearing_single_axis (boardID, slaveAxis, NIMC_T
  CheckError;
```

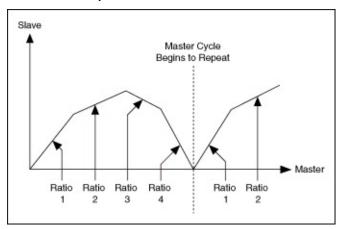
```
// Wait for 5 seconds
Sleep(5000);
//-----
// Set up the move parameters for the superimposed move
//-----
// Set the operation mode to relative
err = flex_set_op_mode(boardID, slaveAxis, NIMC_RELATIVE_TO_CAPTU
CheckError;
// Load Velocity in counts/s
err = flex load velocity(boardID, slaveAxis, 10000, 0xFF);
CheckError;
// Load Acceleration and Deceleration in counts.sec^2
err = flex load acceleration(boardID, slaveAxis, NIMC BOTH, 10006
CheckError;
// Load the target position for the registration (superimposed) m
err = flex load target pos(boardID, slaveAxis, 5000, 0xFF);
CheckError;
// Start registration move on the slave
err = flex_start(boardID, slaveAxis, 0);
CheckError;
err = flex_wait_for_move_complete (boardID, slaveAxis, 0,
                            1000/*ms timeout*/, 20/*ms pol
CheckError;
//-----
// Disable Gearing on slave axis
//-----
err = flex_enable_gearing_single_axis (boardID, slaveAxis, NIMC_F
CheckError;
return; // Exit the Application
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC MODAL ERROR MSG){
  do{
```

## **Camming**

Electronic camming operates similarly to electronic gearing in that the move distance of an axis is proportional to the move distance of its master device. Camming differs from gearing in how the master/slave ratio is handled by the motion controller. Gearing is used in applications where a constant gear value creates a linear slave position profile, as shown.

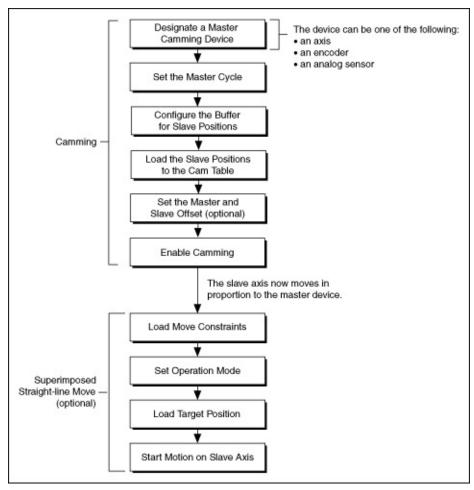


Camming creates a more flexible profile by using more master/slave ratios. These ratios are handled automatically by the motion controller, allowing precise switching of the gear ratios, as shown in the following figure. Camming is used in applications where the slave axis follows a non-linear profile from a master device.



#### **Electronic Camming Algorithm**

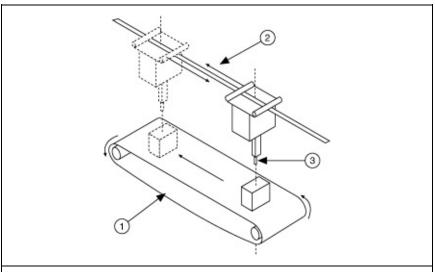
The following figure shows a generic electronic gearing algorithm applicable to both C/C++ and LabVIEW code. Similar to gearing, in a camming application, a slave axis can perform any move when camming is enabled. The move profile is superimposed over the camming profile.



#### **Camming Application Example**

An example of a motion control system that can benefit from the flexibility of electronic camming is welding parts as they travel on a conveyor belt.

The following figure shows that the welding point moves to the first position, and then welds the material for a couple of seconds. Because the conveyor belt keeps moving at a constant rate, the welding point must follow the material at the same speed as the conveyor belt during the weld process. When the welding process is finished for one item, the welding point must quickly return to its initial position and the process repeats.



- 1 Conveyor belt
- 2 Movement of the welder as it follows the object, and then returns to the initial position
- 3 Welding point

In this application, the master device is the position encoder attached to the conveyor belt, and the slave axis is the actuator that moves the welding point. The slave axis repeatedly performs a two-segment movement:

- First, it follows the material with the same speed as the conveyor belt.
- 2. Next, it returns to the initial position as the next material approaches.

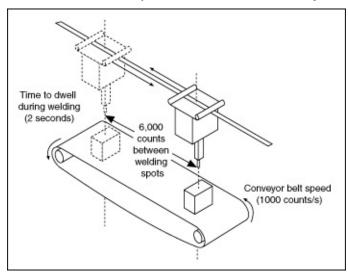
Each segment of the move is represented with a gear ratio that dictates

how fast and in which direction the welding point is moving relative to the conveyor belt.

This application requires the slave axis to switch from one ratio to the other at the correct master position, otherwise the welding process is not repeatable. If this application used gearing instead of camming, the latency, or delay, of loading a new gear ratio might cause an accumulation of position errors.

When a camming operation is active, the slave axis follows a profile that is established using a list of master and slave positions pairs, called the camming table and is loaded into a buffer for use during a camming operation. An example of a camming table is shown below.

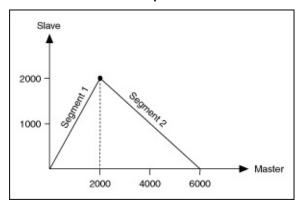
The following figure shows that, in the welding example defined previously, the conveyor belt is moving at 1,000 counts/s and the parts to be welded are 6,000 counts apart. To weld the part, the welding point must follow the part down the conveyor belt for 2 seconds.



In this welding application, the slave axis must follow the part with the same velocity as the conveyor belt while welding is in progress. Because it takes two seconds to weld each part, the welding point and conveyor belt have both moved 2,000 counts by the time the welding is complete. This part of the welding application creates the first move segment.

For the second move segment, the welding point must return to its original position so that it can weld the next part on the conveyor belt. To move the welding point to its original position at the same time that the next part is in the correct position on the conveyor belt, the welding point must travel 2,000 counts in the opposite direction of the conveyor belt at

half the speed that the conveyor belt is traveling at. The following figure shows the move profile of the first and second move segments.



The following table shows the camming table that corresponds to the move profile used in this example.

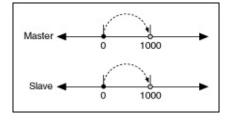
Time (seconds)	Master Position (counts)	Slave Position (counts)	Ratio*	
0	0	0		
2	2000	2000	1	
4	4000	1000	0.5	
6	6000	0	0.5	
* Ratio = $\Delta$ SlaveDistance / $\Delta$ MasterDistance				

The camming cycle is 6,000 counts and is divided into equal length segments.

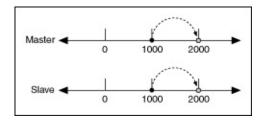


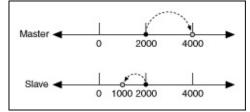
**Tip** Because the camming cycle is 6,000 counts, the master cycle must also be 6,000 counts.

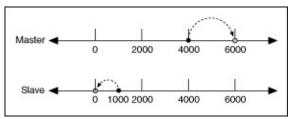
Each row of data defines a gear ratio. The camming profile is repeated after a camming cycle is completed. The master position is always interpreted inside the modulus defined by the camming cycle.



For example, initially, the master axis moves from 0 to 1000. The gear ratio used for this move is 1:1 because the master position is in the 0 to 2000 interval. With a gear ratio of 1:1, the slave axis moves at the same speed as the master device to position 1000.







As shown, the master position in this interval is inside the modulus.

If the master axis moves to position 2000, the gear ratio does not change because the current master position is still inside the 0 to 2000 interval.

When the master axis moves to position 4,000, the gear ratio changes to 0.5. The slave axis travels half the distance that the master axis travels, and it travels in the opposite direction.

When the master reaches position 6000, the slave axis moves back the original position, and the camming cycle begins again.

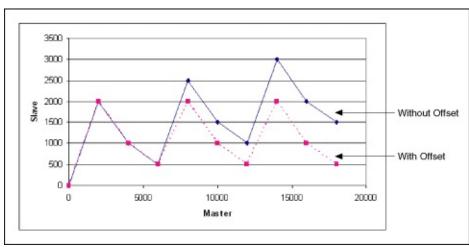
#### **Slave Offset**

In some camming applications, the slave axis might begin and end the camming cycle at different positions, as shown in the following table.

<b>Master Position (counts)</b>	Slave Position (counts)
0	0
2000	2000
4000	1000
6000	500

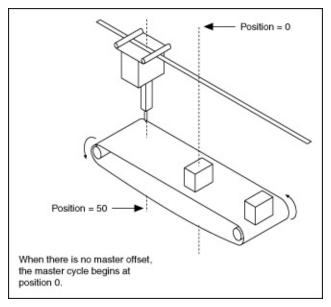
The following figure shows that, after three camming cycles, the slave axis end position is 500 counts away from the starting position (0) with the slave offset, and that without the slave offset, the slave axis end position is 1500 counts away from the starting position (0).

With a slave offset of 500, the slave axis traverses the positions specified in the camming table, but it does not maintain the camming ratio.



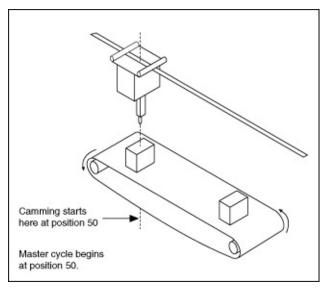
#### **Master Offset**

If the material and welding point are not initially aligned, as shown, the master offset must be applied to consider the position difference.

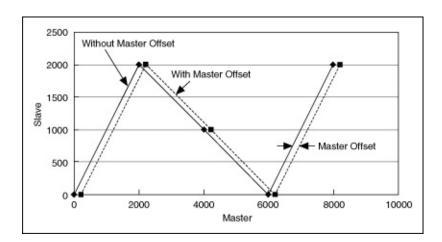


Without the master offset, the master device position is already inside the first interval as soon as the first material passes the welding point.

The following figure shows that the master cycle intervals are offset by 50 counts. The master interval is shifted from 0, 2,000, 4,000, and 6,000 to 50, 2,050, 4,050, and 6,050.



The following figure shows the camming profile used for the application portrayed in the previous figures.

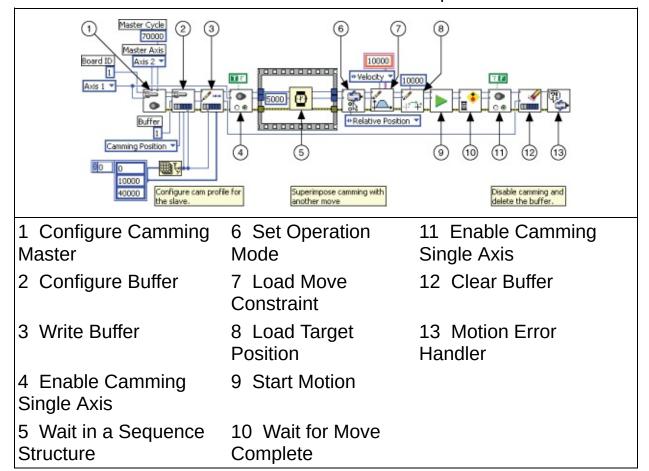


## **Electronic Camming LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to perform electronic camming.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



#### **Electronic Camming C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

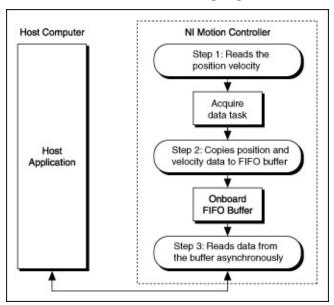
```
// Main Function
void main(void)
  // Locals
  u8 boardID; // Board ID as assigned by MAX
  f64 bufferInterval = 0; // Ignored
  // Master axis information
  // Slave axis information
  u8 slaveAxis;  // Slave axis ID
u8 buffer;  // Buffer to cont
                     // Buffer to contain the cam table
  i32 positionArr[] = {0, 10000, 40000, 45000, 45000, 40000, 10000,
  u32 positionSize = sizeof(positionArr) / sizeof(i32); // Number
  i32 slavePosition; // Current slave position
  // For error handling
  // by the scanf function
  // Get the board ID
  printf("Enter the board ID: ");
  scanf("%d", &scanVar);
  boardID=(u8)scanVar;
  // Get master axis information
  printf("\nEnter information about the master axis... \n");
  printf("Axis ID: ");
  scanf("%d", &scanVar);
  masterAxis=(u8)scanVar;
```

```
printf("Target position: ");
scanf("%d", &targetPos);
printf("Velocity limit: ");
scanf("%lf", &velocity);
// Get the slave Axis
printf("\nEnter information about the slave axis... \n");
printf("Axis ID: ");
scanf("%d", &scanVar);
slaveAxis=(u8)scanVar;
printf("Buffer ID: ");
scanf("%d", &scanVar);
buffer=(u8)scanVar;
// Configure the cam master & master cycle
err = flex configure camming master(boardID, slaveAxis, masterAxi
CheckError;
// Configure the cam table
err = flex_configure_buffer(boardID, buffer, slaveAxis,
                            NIMC_CAMMING_POSITION,
                            positionSize, positionSize,
                            TRUE, bufferInterval, &bufferInterval
CheckError;
// Write the data to the buffer
err = flex_write_buffer(boardID, buffer, positionSize,
                        NIMC REGENERATION NO CHANGE,
                        positionArr, 0xFF);
CheckError;
// Enable camming immediately
err = flex enable camming single axis(boardID, slaveAxis, TRUE, -
CheckError;
// Set to absolute mode
err = flex_set_op_mode(boardID, masterAxis, NIMC_ABSOLUTE_POSITIO)
CheckError;
// Set the maximum velocity
data.doubleData = velocity;
err = flex_load_move_constraint(boardID, masterAxis,
                                TnimcMoveConstraintVelocity, &dat
CheckError;
// Set the target position
err = flex load target pos(boardID, masterAxis, targetPos, 0xFF);
CheckError;
```

```
// Start the master axis movement
err = flex_start(boardID, masterAxis, 0x0);
CheckError;
// Keep running until the master completes or there's a modal err
do
{
  Sleep (100);
  // Get both the master and the slave current positions.
  err = flex_read_pos_rtn(boardID, masterAxis, &masterPosition);
  CheckError;
  err = flex_read_pos_rtn(boardID, slaveAxis, &slavePosition);
  CheckError;
  // Display the master and the slave current positions.
  printf("Axis %d (master) position is %d, Axis %d (slave) posit
     masterAxis, masterPosition, slaveAxis, slavePosition);
  // Read current master trajectory status
  err = flex_read_axis_status_rtn(boardID, masterAxis, &masterSt
  CheckError;
  // Check for modal error
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
} while (!(masterStatus & NIMC_MOVE_COMPLETE_BIT) &&
        !(csr & NIMC MODAL ERROR MSG));
// Error Handling
nimcHandleError; // NIMCCATCHTHIS:
// Disable camming
flex enable camming single axis(boardID, slaveAxis, FALSE, -1.0);
// Clear (delete) the buffer
flex clear buffer(boardID, buffer);
// Make sure the master is not running
flex_stop(boardID, masterAxis, 0x0);
// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG){
  do{
     // Get the command ID, resource and the error code of the m
```

# **Acquiring Time-Sampled Position and Velocity Data**

NI motion controllers can acquire a buffer of position and velocity data that is firmware-timed. After you command the motion controller to acquire position and velocity data, a separate acquire data task is created in the real-time operating system that reads time-sampled position and velocity data into a FIFO buffer on the motion controller. You can read data in from this buffer asynchronously from the host computer, as shown in the following figure.

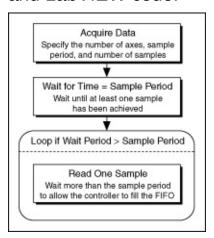


The acquire data task has higher priority than any onboard programs or housekeeping tasks, but it has a lower priority than the I/O reaction and host communication tasks. To achieve the best possible performance, keep host communications to a minimum when acquiring data.

The FIFO buffer is of a fixed size that can accommodate 4,096 samples for one axis. One sample consists of position data, in counts or steps, and velocity data, in counts/s or steps/s. As you increase the number of axes from which you are acquiring data, you also decrease the total number of samples you can acquire per axis. For example, you can acquire up to 1,024 samples per axis for four axes. You also can vary the time period between acquired samples from 3 ms to 65,535 ms.

#### **Acquire Data Algorithm**

The following figure shows a generic algorithm applicable to both C/C++ and LabVIEW code.



The data must be read one sample at a time. A four-axis sample uses the following pattern for returning the data.

Axis :	1 position
Axis :	1 velocity
Axis 2	2 position
Axis 2	2 velocity
Axis 3	3 position
Axis 3	3 velocity
Axis 4	4 position
Axis 4	4 velocity

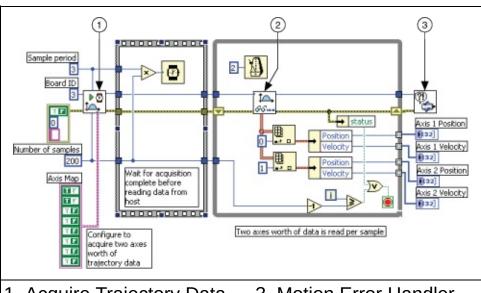
If you request 1,024 samples, you must read each of the 1,024 samples individually.

#### **Acquire Data LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to acquires data for two axes, 200 samples, and three milliseconds apart.

(P

**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



- 1 Acquire Trajectory Data
- 3 Motion Error Handler
- 2 Read Trajectory Data

#### Acquire Data C/C++ Code

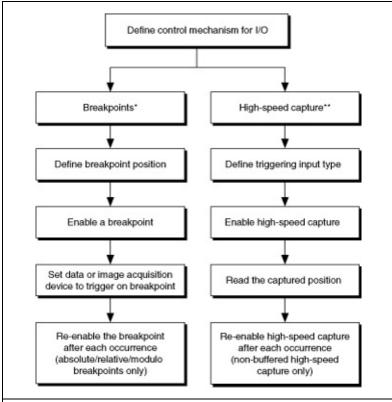
The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
                    // Board identification number
// Communication status register
  u8
       boardID;
  u16 csr = 0;
  i32 i;
  // Bitmap of axes for which data is
  u16 axisMap;
  // Variables for modal error handling
                  // The commandID of the function
// The resource ID
// Error code
  u16 commandID;
  u16 resourceID;
  i32 errorCode;
  // Set the board ID
  boardID= 1;
  // Axes whose data needs to be acquired
  axisMap = ((1 << 1) | (1 << 2)); // Axis 1 and axis 2
  err = flex_acquire_trajectory_data(boardID, axisMap, numSamples,
  CheckError;
  for(i=0; i<numSamples; i++){</pre>
     Sleep (10); // Check every 10 ms and give time for the data
                // to be copied to onboard FIFO
     // Read the trajectory data
     err = flex_read_trajectory_data_rtn(boardID, returnData);
     CheckError;
```

```
// Two axes worth of data is read every sample
     axis1Positions[i] = returnData[0];
     axis1Velocities[i] = returnData[1];
     axis2Positions[i] = returnData[2];
     axis2Velocities[i] = returnData[3];
  return;
              // Exit the Application
  // Error Handling
  //
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC MODAL ERROR MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID,&csr);
     }while(csr & NIMC_MODAL_ERROR_MSG);
  }
            // Display regular error
  else
     nimcDisplayError(err, 0, 0);
  return; // Exit the Application
}
```

#### **Synchronization**

You can synchronize NI motion controllers with NI data and image acquisition devices using breakpoints and high-speed captures. The following figure illustrates an expanded view of the topics covered in this section.



<sup>\*</sup> Breakpoints cause a digital output to change state when a specified position is reached by an encoder. Breakpoints are not supported by the NI SoftMotion Controller.

<sup>\*\*</sup> A high-speed capture records the position of an encoder when a digital line is used as a trigger. High-speed captures are not supported by the NI SoftMotion Controller.



**Note** If you are using RTSI to connect your motion controller to a National Instruments data or image acquisition device, be aware that the NI SoftMotion Controller does not support RTSI.

Timing and triggering with NI-Motion is always related to either position or velocity. Synchronizing position and velocity information with the external world allows you to coordinate measurements with moves. You can program the motion controller to trigger another device at specified positions using RTSI or a pin on the Motion I/O connector. This

functionality is called breakpoints, which are divided into <u>absolute</u> <u>breakpoints</u>, <u>relative position breakpoints</u>, and <u>periodically occurring</u> <u>breakpoints</u>.

In some cases, it may be necessary to synchronize position with some measurement occurring external to the motion controller. For example, you might be aligning two fiber optic cables, in which case the maximum optical power needs to correspond with the alignment position. To align the fibers, the external device that is recording the optical power must trigger the motion controller so that positions and optical power measurements can be synchronized and analyzed. This functionality is known as <a href="https://doi.org/10.10">https://doi.org/10.10</a> connector. When trigger inputs. The motion controller can be triggered by another device using RTSI or externally using a pin on the Motion I/O connector. When triggered, the motion controller can latch the current position of the encoder, which can be read and recorded.

The following table shows the availability of breakpoint modes on each NI motion controller.

Breakpoint Mode	NI 7350	NI 7340, NI 7330, and NI 7390
Absolute*	Υ	Υ
Relative*	Υ	Υ
Periodic	Υ	N
Modulus	N	Υ
Buffered	Υ	N

<sup>\*</sup> Available in buffered and single operation for NI 7350 and in single operation only for all other controllers

## **Absolute Breakpoints**

Absolute position breakpoints allow you to trigger external activities as the motors reach specified positions. For example, if you need to use an image acquisition device to capture an image from a certain position while the device under test is in continuous motion, the motion controller must be able to trigger the image acquisition device as it reaches those positions. The current position is continuously compared against the specified breakpoint position by the encoder circuitry to produce a latency of less than 100 ns.

After a breakpoint triggers, you must re-enable it for the breakpoint to work again. In certain cases, such as buffered and periodic breakpoints, the motion controller automatically re-enables the breakpoints.

The implementation for absolute breakpoints is divided into the <u>buffered</u> <u>breakpoint</u> and <u>single position breakpoint methods</u>.



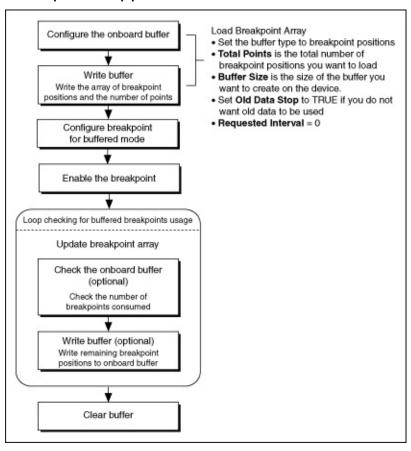
**Note** All breakpoints can be affected by jitter in the motion control system. For example, if you have a very small breakpoint window, the jitter in the motion control system could cause the position to change enough to reach the breakpoint when a breakpoint is not intended. Increase the size of the breakpoint window to compensate for system jitter.

# **Buffered Breakpoints (NI 7350 only)**

Instead of enabling breakpoints in your application at the software level, you can create a buffer of breakpoints that you can pre-load into the motion controller. The motion controller automatically arms the next breakpoint in the buffer when the preceding breakpoint triggers. Therefore, enabling breakpoints occurs on a firmware-timed basis, which enables you to use a higher bandwidth.

#### **Buffered Breakpoint Algorithm**

The following shows the basic algorithm for implementing buffered breakpoints applicable to both C/C++ and LabVIEW code.

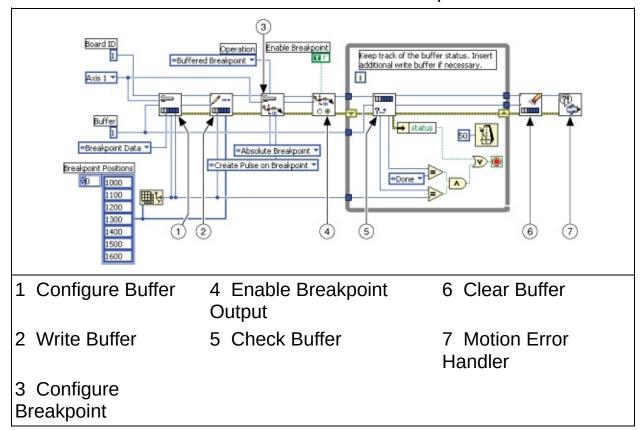


## **Buffered Breakpoint LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to complete a blended move.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



#### **Buffered Breakpoint C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main (void)
  // Modal error handling variables
  // Buffer resources
   i32 breakpointPositions[] = {1000, 1100, 1200, 1300, 1400, 1500,
  u16 numberOfPoints = 7;  // Number of breakpoints
f64 actualInterval;  // Required in the function ca
f64 requestedInterval = 10.0;  // Required in the function ca
u32 backLog:  // Number of space available i
                           // Number or spa
// Buffer state
// Number of bra
                                      // Number of space available i
   u32 backLog;
   u16 bufferState;
                                       // Number of breakpoints done
   u32 pointsDone;
   // Configure the buffer for buffered breakpoint
   err = flex_configure_buffer(boardID, buffer, axis, NIMC_BREAKPO
                                numberOfPoints, numberOfPoints,
                                requestedInterval, &actualInterval);
   CheckError;
   // Write the breakpoint position to the buffer
   err = flex_write_buffer(boardID, buffer, numberOfPoints,
                            NIMC REGENERATION NO CHANGE, breakpointPo
   CheckError;
   // Configure the breakpoint to be buffered breakpoint
   err = flex_configure_breakpoint(boardID, axis, NIMC_ABSOLUTE_BREA
                                    NIMC_PULSE_BREAKPOINT, NIMC_OPERA
   CheckError;
   // Enable the breakpoint
   err = flex_enable_breakpoint(boardID, axis, NIMC_TRUE);
   CheckError;
```

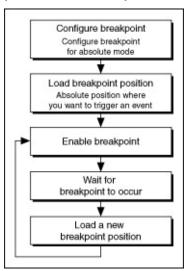
```
// Poll the status of the buffer, if you more breakpoint position
  // to write, insert flex_write_buffer call here.
  do
   {
     // Check the buffer status
     err = flex check buffer rtn(boardID, buffer,
                                &backLog, &bufferState, &pointsDon
     CheckError;
     Sleep(50);
  } while ((pointsDone != numberOfPoints) ||
           (bufferState != NIMC BUFFER DONE));
  // Clear the buffer
  err = flex_clear_buffer(boardID, buffer);
  CheckError;
  return;
  // Error Handling
  //
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC MODAL ERROR MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID,&commandID,&resourceID,&err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID, &csr);
     }while(csr & NIMC_MODAL_ERROR_MSG);
  }
  else
             // Display regular error
     nimcDisplayError(err, 0, 0);
  return; // Exit the Application
}
```

# **Single Position Breakpoints**

Single position breakpoints execute one breakpoint per enabling.

#### **Single Position Breakpoint Algorithm**

The following figures shows the basic algorithm for implementing single position breakpoints applicable to both C/C++ and LabVIEW code.

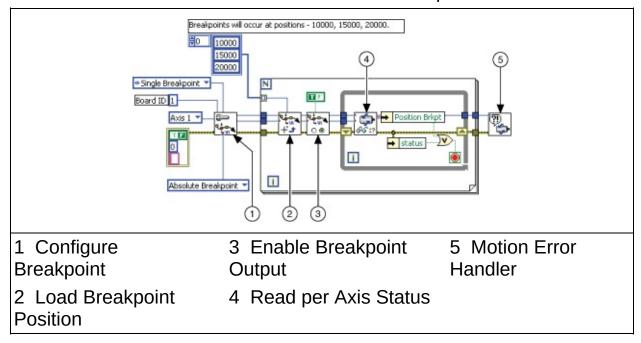


## Single Position Breakpoint LabVIEW Diagram

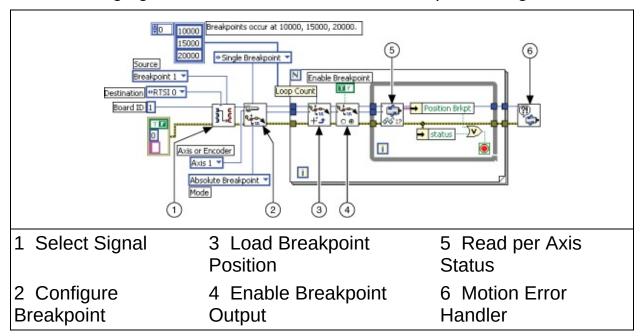
The following block diagrams demonstrate using NI-Motion to program a single position breakpoint, both with and without RTSI.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



The following figure shows how to route this breakpoint using RTSI.



After the breakpoint is routed through RTSI, the trigger appears on both

the RTSI line and the breakpoint line on the Motion I/O connector.

## Single Position Breakpoint C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
   u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
   i32 breakpointPosition[3] = {10000, 15000, 20000};
   i32 i:
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID= 1;
   // Set the axis number
   axis = 1;
   // Route breakpoint 1 to RTSI line 1
   err = flex_select_signal (boardID, NIMC_RTSI1 /*destination*/, NI
   CheckError;
   // Configure Breakpoint
   err = flex_configure_breakpoint(boardID, axis, NIMC_ABSOLUTE_BREA
   CheckError;
   for(i=0; i<3; i++){
      // Load Breakpoint Position - position at which breakpoint sho
      err = flex_load_pos_bp(boardID, axis, breakpointPosition[i], @
      CheckError;
      // Enable the breakpoint on axis 1
      err = flex_enable_breakpoint(boardID, axis, NIMC_TRUE);
```

```
CheckError;
  do
   {
     // Check the breakpoint status
     err = flex_read_axis_status_rtn(boardID, axis, &axisStatus)
     CheckError;
     // Read the communication status register and check the moa
     err = flex_read_csr_rtn(boardID, &csr);
     CheckError;
     // Check for modal errors
     if (csr & NIMC_MODAL_ERROR_MSG)
        err = csr & NIMC_MODAL_ERROR_MSG;
        CheckError;
     Sleep (10); // Check every 10 ms
  }while (!(axisStatus & NIMC POS BREAKPOINT BIT));// Wait for b
return; // Exit the Application
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG){
  do{
     // Get the command ID, resource and the error code of the m
     // error from the error stack on the board
     flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
     nimcDisplayError(errorCode, commandID, resourceID);
     // Read the Communication Status Register
     flex_read_csr_rtn(boardID,&csr);
  }while(csr & NIMC_MODAL_ERROR_MSG);
}
else
          // Display regular error
   nimcDisplayError(err, 0, 0);
return; // Exit the Application
```

}

## **Relative Position Breakpoints**

Relative position breakpoints trigger events based on a change in position relative to the position at which the breakpoint was enabled.

Instead of keeping track of absolute positions and the current position, you can use relative breakpoints to specify the breakpoint relative to the position where the breakpoint is enabled.

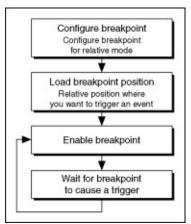
For example, if you are creating a motion control system to control the two-dimensional movement of a microscope, you might use relative position breakpoints to move the microscope a specific distance in a direction, and then hit a breakpoint that triggers a camera snap. The relative breakpoint is useful in this example because the current position is not important. The application must move the axis a specific number of counts from wherever it is, and then generate a breakpoint.



**Note** All breakpoints can be affected by jitter in the motion control system. For example, if you have a very small breakpoint window, the jitter in the motion control system could cause the position to change enough to reach the breakpoint when a breakpoint is not intended. Increase the size of the breakpoint window to compensate for system jitter.

### **Relative Position Breakpoints Algorithm**

The following figure shows the basic algorithm for relative breakpoints applicable to both C/C++ and LabVIEW code.



Notice that relative breakpoints are not ideal for periodic breakpoints. There is a latency between the time a breakpoint generates and is reenabled. If the axis is moving at sufficient velocity, the breakpoint reenables only after the axis has moved slightly. Because a relative breakpoint generates relative to the position the axis was in when the breakpoint was enabled, the latency between generation and re-enabling can cause additional counts between breakpoints.

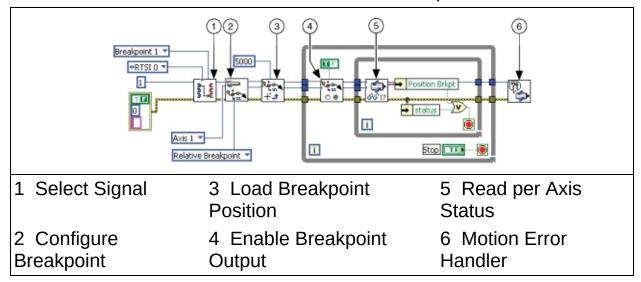
For example, the actual breakpoints might occur at positions 5,000; 10,003; 15,006; and 20,012. In this example, the axis moves three counts between a breakpoint and the subsequent re-enabling. For exact distances between breakpoints at high speeds, use <a href="Buffered Breakpoints">Buffered Breakpoints</a> (NI 7350 only) or <a href="Periodically Occurring Breakpoints">Periodically Occurring Breakpoints</a>.

## **Relative Position Breakpoints LabVIEW Diagram**

In this example, a breakpoint generates and then is re-enabled 5,000 counts from where the move starts using RTSI. The following code examples are designed to illustrate the relative breakpoint algorithm only. These examples are not complete.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



# Relative Position Breakpoints C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
       boardID;
  i32 breakpointPosition = 5000;
  // Variables for modal error handling
  // Set the board ID
  boardID= 1;
  // Set the axis number
  axis = 1:
  // Route breakpoint 1 to RTSI line 1
  err = flex_select_signal (boardID, NIMC_RTSI1 /*destination*/, NI
  CheckError;
  // Configure Breakpoint
  err = flex_configure_breakpoint(boardID, axis, NIMC_RELATIVE_BREA
  CheckError;
  // Load Breakpoint Position - position at which breakpoint shoula
  err = flex_load_pos_bp(boardID, axis, breakpointPosition, 0xFF);
  CheckError;
  for(;;){
     // Enable the breakpoint on axis 1
     err = flex_enable_breakpoint(boardID, axis, NIMC_TRUE);
     CheckError;
     do
```

```
{
     // Check the breakpoint status
     err = flex_read_axis_status_rtn(boardID, axis, &axisStatus)
     CheckError;
     // Read the communication status register and check the mod
     err = flex_read_csr_rtn(boardID, &csr);
     CheckError;
     // Check for modal errors
     if (csr & NIMC_MODAL_ERROR_MSG)
        err = csr & NIMC MODAL ERROR MSG;
        CheckError;
     }
     Sleep (10); // Check every 10 ms
   }while (!(axisStatus & NIMC_POS_BREAKPOINT_BIT));// Wait for b
return;
         // Exit the Application
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC MODAL ERROR MSG){
  do{
     // Get the command ID, resource and the error code of the m
     // error from the error stack on the board
     flex read error msq rtn(boardID, &commandID, &resourceID, &err
     nimcDisplayError(errorCode, commandID, resourceID);
     // Read the Communication Status Register
     flex_read_csr_rtn(boardID,&csr);
   }while(csr & NIMC_MODAL_ERROR_MSG);
}
else
          // Display regular error
   nimcDisplayError(err, 0, 0);
return; // Exit the Application
```

}

## **Periodically Occurring Breakpoints**

NI-Motion allows you to program the motion controller to generate multiple breakpoints at fixed and exact intervals, regardless of the direction of travel or velocity.

There are two ways to create periodically occurring breakpoints using NI-Motion functions, depending on which motion controller you have. For the NI 7350 controller, use <u>periodic breakpoints</u>. For NI 7330, NI 7340, and NI 7390 controllers, use <u>modulo breakpoints</u>.

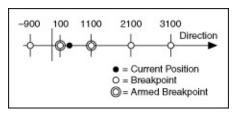


**Note** All breakpoints can be affected by jitter in the motion control system. For example, if you have a very small breakpoint window, the jitter in the motion control system could cause the position to change enough to reach the breakpoint when a breakpoint is not intended. Increase the size of the breakpoint window to compensate for system jitter.

# Periodic Breakpoints (NI 7350 only)

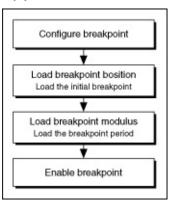
Periodic breakpoints require that you specify an initial breakpoint and an ongoing repeat period. When enabled, the periodic breakpoints begin when the initial breakpoint occurs. From then on, a new breakpoint occurs each time the axis moves a distance equal to the repeat period, with no re-enabling required.

For example, if an axis is enabled at position zero, the initial breakpoint is set for position 100, and the breakpoint period is set at 1,000, then the axis behaves as shown in the following figure.



## **Periodic Breakpoint Algorithm**

The following figure shows the basic algorithm for periodic breakpoints applicable to both C/C++ and LabVIEW code.

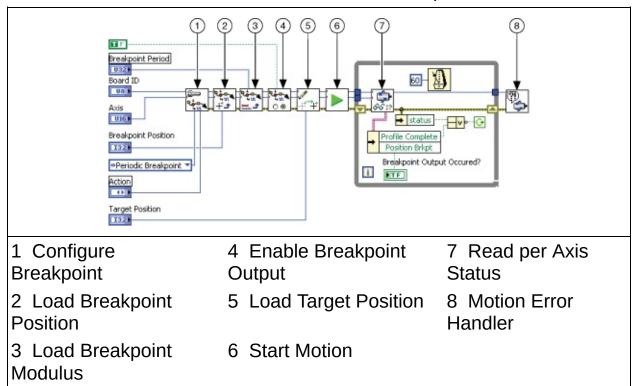


# Periodic Breakpoint LabVIEW Diagram

The following block diagram demonstrates using NI-Motion to generate a periodic breakpoint output.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



## Periodic Breakpoint C/C++ Code

The following section includes C/C++ code for executing a high-speed capture, as well as using RTSI to execute a high-speed capture. The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  // Variables for modal error handling
  u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
  // Get the board ID
   printf("Enter the Board ID: ");
   scanf("%u", &boardID);
   // Get the axis number
   printf("Enter a axis number: ");
   scanf("%u",&axis);
   // Get the Target Position
   printf("Enter a target position: ");
   scanf("%ld",&targetPos);
   // Get the Breakpoint Position
   printf("Enter a breakpoint position: ");
   scanf("%ld",&bpPos);
   // Get the Breakpoint Period
   printf("Enter a breakpoint period: ");
  scanf("%ld",&bpPer);
```

```
// Configure the breakpoint to be absolute
err = flex configure breakpoint(boardID, axis, NIMC PERIODIC BREAKF
CheckError;
// Load the position to start breakpoints
err = flex_load_pos_bp(boardID, axis, bpPos, 0xFF);
CheckError;
// Set the Period
err = flex load bp modulus(boardID, axis, bpPer, 0xFF);
CheckError;
// Enable the breakpoint
err = flex enable breakpoint(boardID, axis, NIMC_TRUE);
CheckError;
// Load a target position
err = flex load target pos(boardID, axis, targetPos, 0xFF);
CheckError;
// Start the motion
err = flex_start(boardID, axis, 0);
CheckError;
printf("\n");
do
{
   // Read the axis status
   err = flex_read_axis_status_rtn(boardID, axis, &axisStatus);
   CheckError;
   err = flex_read_pos_rtn(boardID, axis, &currentPos);
   CheckError;
   // Check the breakpoint bit
   bpStatus = !((axisStatus & NIMC_POS_BREAKPOINT_BIT)==0);
   // Check the profile complete bit
   profileStatus = !((axisStatus & NIMC_PROFILE_COMPLETE_BIT)==0)
   printf("Current Position=%10d Breakpoint Status=%d Profile Com
   // Check for modal errors
   err = flex_read_csr_rtn(boardID,&csr);
   CheckError;
   // Check the modal errors
   if (csr & NIMC_MODAL_ERROR_MSG)
   {
      flex_stop_motion(boardID, NIMC_VECTOR_SPACE1, NIMC_DECEL_STO
```

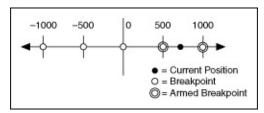
```
err = csr & NIMC_MODAL_ERROR_MSG;
        CheckError;
     }
  }while(!profileStatus);
  printf("\nFinished.\n");
              // Exit the Application
  return;
  // Error Handling
  //
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC MODAL ERROR MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID,&csr);
     }while(csr & NIMC_MODAL_ERROR_MSG);
  }
            // Display regular error
  else
     nimcDisplayError(err, 0, 0);
  return; // Exit the Application
}
```

# Modulo Breakpoints (NI 7330, NI 7340, and NI 7390 only)

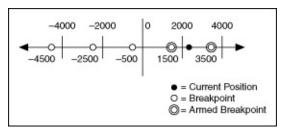
Modulo breakpoints use a breakpoint window, which defines an area around the current position. The two breakpoints around the current position are always enabled.

The breakpoint modulus creates a repeat period for the breakpoints, and the breakpoint position is the offset from absolute zero.

For example, to create a breakpoint every 500 counts, set the repeat period to 500 and the breakpoint position to 0. If the breakpoint is enabled when the axis is at 710, the breakpoints at 1000 and 500 are both armed, as shown in the following figure.



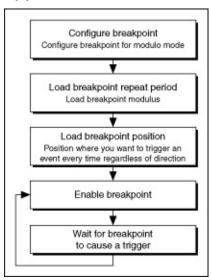
As another example, if you set the breakpoint repeat period to be 2000 counts and the offset to be 500, breakpoints occur at 4500, 2500, 500, 1500, 3500. If the breakpoint is enabled when the axis is at 2210, the breakpoints at 1500 and 3500 are both armed, as shown in the following figure.



Each time a breakpoint occurs, re-enable it to load the next breakpoint.

## **Modulo Breakpoints Algorithm**

The following figure shows the basic algorithm for modulo breakpoints applicable to both C/C++ and LabVIEW code.

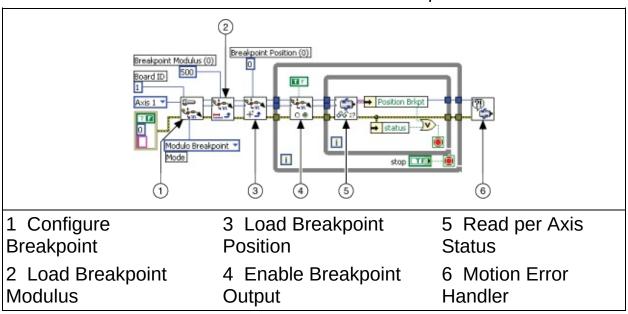


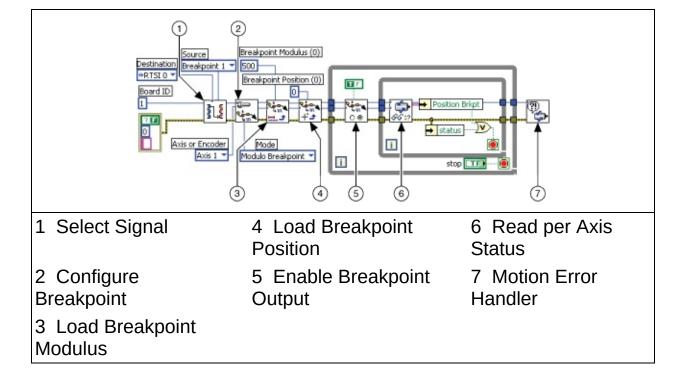
## **Modulo Breakpoints LabVIEW Diagram**

The following block diagrams demonstrate using NI-Motion to generate a modulo breakpoints both with and without RTSI.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.





# Modulo Breakpoints C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

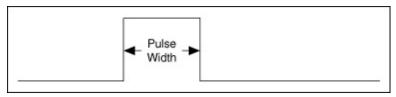
```
// Main function
void main(void)
   u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
   u8 boardID;
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 1;
   // Set the axis number
   axis = 1;
   // Route breakpoint 1 to RTSI line 1
   err = flex_select_signal (boardID, NIMC_RTSI1 /*destination*/, NI
   CheckError;
   // Configure Breakpoint
   err = flex configure breakpoint(boardID, axis, NIMC MODULO BREAKP
   CheckError;
   // Load Breakpoint Modulus - repeat period
   err = flex_load_bp_modulus(boardID, axis, 500, 0xFF);
   CheckError;
   // Load Breakpoint Position - position at which breakpoint shoula
   // every modulo
   err = flex_load_pos_bp(boardID, axis, 0, 0xFF);
   CheckError;
   for(;;){
```

```
// Enable the breakpoint on axis 1
  err = flex_enable_breakpoint(boardID, axis, NIMC_TRUE);
  CheckError;
  do
   {
     // Check the move complete status/following error/axis off
     err = flex read axis status rtn(boardID, axis, &axisStatus)
     CheckError;
     // Read the communication status register and check the moa
     err = flex_read_csr_rtn(boardID, &csr);
     CheckError;
     // Check the modal errors
     if (csr & NIMC_MODAL_ERROR_MSG)
        err = csr & NIMC_MODAL_ERROR_MSG;
        CheckError;
     Sleep (10); //Check every 10 ms
   }while (!(axisStatus & NIMC_POS_BREAKPOINT_BIT)); // Wait for
return; // Exit the Application
// Error Handling
//
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC MODAL ERROR MSG){
  do{
     // Get the command ID, resource and the error code of the m
     // error from the error stack on the board
     flex read error msg rtn(boardID,&commandID,&resourceID,&err
     nimcDisplayError(errorCode, commandID, resourceID);
     // Read the Communication Status Register
     flex_read_csr_rtn(boardID, &csr);
   }while(csr & NIMC_MODAL_ERROR_MSG);
}
          // Display regular error
else
  nimcDisplayError(err, 0, 0);
return; // Exit the Application
```

}

# Configuring the Breakpoint Pulse Width (NI 7350 only)

On NI 7350 devices you can configure a customized pulse width for breakpoints (position compare outputs) to trigger devices that have specific minimum active pulse requirements. The default breakpoint pulse width is 200 ns. You can use MAX or the Write Capture Compare Data VI or function to set a pulse width of between 1  $\mu$ s and 65,535  $\mu$ s. On a breakpoint event, the breakpoint signal is active for the configured pulse width, as shown in the following figure.





**Note** A breakpoint across RTSI always generates an active high pulse of 200 ns duration.



**Caution** The breakpoint frequency depends on the velocity and distance between breakpoints and is limited by the pulse width. The breakpoint frequency must meet the following requirements for correct operation.

### **Breakpoint Frequency Requirements**

The breakpoint pulse must not overlap with the next breakpoint event. If the breakpoint pulse overlaps with the next breakpoint event, breakpoint generation is stopped and the breakpoint is disabled. To recover from this situation change the pulse width, velocity, or distance between breakpoints. In addition, the breakpoint frequency cannot be greater than the maximum value allowed for the specified breakpoint type. Therefore, the breakpoint frequency, in hertz, at the specified pulse width, in seconds, must conform to must conform to the following two conditions:

position compare frequency (hertz) 
$$< \frac{1}{\text{pulse width (seconds)}}$$

#### **AND**

Position Compare (Breakpoint) Output Type	<b>Maximum Frequency</b>
Single	150 Hz
Buffered, PID rate of 62.5 μs to 250 μs	2 kHz
Buffered, PID rate greater than 250 μs	1 kHz
Periodic	4 MHz

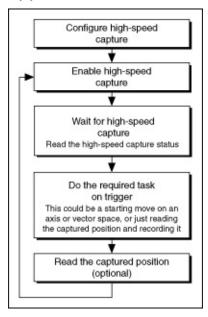
# **High-Speed Capture**

Some motion control applications require that you execute a move and record the locations where external triggers happen. To accomplish this, you must use the high-speed capture functionality of NI motion controllers.

The implementation for high-speed capture is divided into the <u>buffered</u> (NI 7350 controllers only) and <u>non-buffered</u> high-speed capture methods.

# **High-Speed Capture Algorithm**

The following figure shows the basic algorithm for high-speed captures applicable to both C/C++ and LabVIEW code.

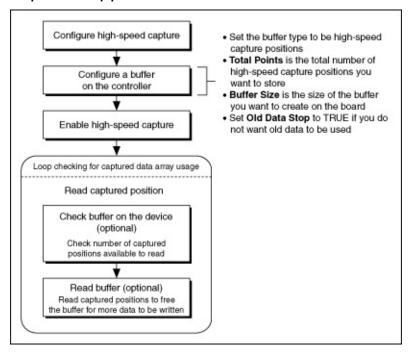


# **Buffered High-Speed Capture (NI 7350 only)**

Buffered high-speed capture lets you create a buffer that holds captured positions that you can read asynchronously from the motion controller. The motion controller automatically arms the next high-speed capture, and writes the captured high-speed data into its onboard buffer. The enabling of high-speed capture occurs on a firmware-timed basis, which provides better frequency than the non-buffered high-speed capture method.

## **Buffered High-Speed Capture Algorithm**

The following figure shows the basic algorithm for buffered high-speed captures applicable to both C/C++ and LabVIEW code.

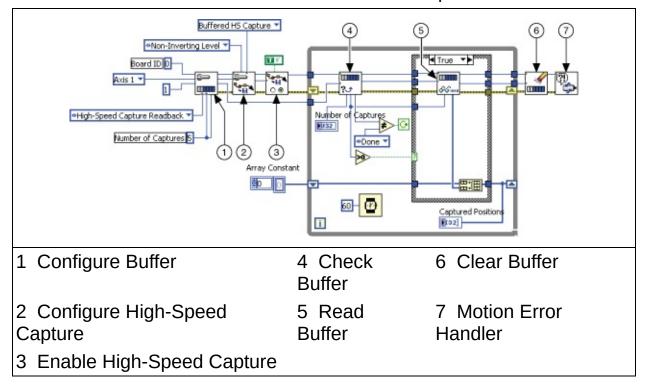


# **Buffered High-Speed Capture LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to perform a buffered high-speed capture.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



## **Buffered High-Speed Capture C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u32 i;
  // Variables for modal error handling
                 // The commandID of the function
// The resource ID
// Error code
  u16 commandID:
  u16 resourceID;
  i32 errorCode;
  // Set the board ID
  boardID = 1;
  // Set the axis number
  axis = NIMC AXIS1;
  // Configure buffer on motion controller memory (RAM)
  // Note requested time interval is hardcoded to 10 milliseconds
  err = flex_configure_buffer(boardID, 1 /*buffer number*/, axis, N
                                 totalPoints, NIMC_TRUE, 10, &a
  CheckError;
  // Configure High Speed Capture
  err = flex_configure_hs_capture(boardID, axis, NIMC_HS_LOW_TO_HIG
  CheckError;
  // Enable the high speed capture on axis
```

```
err = flex_enable_hs_capture(boardID, axis, NIMC_TRUE);
CheckError;
do
  err = flex_check_buffer_rtn(boardID, 1 /*buffer number*/, &bac
  CheckError:
  // Check backlog for captured position in buffer
   if (backlog > 0)
     readBuffer = (i32*)malloc(sizeof(i32)*backlog);
     // If captured position available in the buffer, read the c
     err = flex_read_buffer_rtn(boardID, 1/*buffer number*/, bac
     for(i=0;i<backlog;i++){</pre>
        if(currentDataPoint > totalPoints) break;
        capturedPositions[currentDataPoint] = readBuffer[i];
        printf("capture pos %d\n", capturedPositions[currentDataP
        currentDataPoint++;
     free(readBuffer);
     readBuffer = NULL;
     CheckError;
  }
  // Check for axis off status/following error or any modal erro
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG){
     err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
Sleep(60);
            // Check every 60 ms
} while (bufferState != NIMC BUFFER DONE);
// Free the buffer allocated on the controller memory
err = flex_clear_buffer(boardID, 1/*buffer number*/);
CheckError;
return; // Exit the Application
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
```

```
// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG) {
    do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID,&commandID,&resourceID,&err
        nimcDisplayError(errorCode,commandID,resourceID);

    // Read the Communication Status Register
        flex_read_csr_rtn(boardID,&csr);

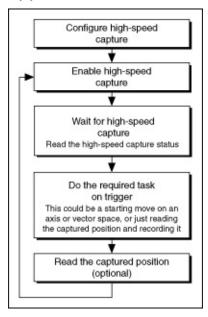
    }while(csr & NIMC_MODAL_ERROR_MSG);
}
else    // Display regular error
    nimcDisplayError(err,0,0);
return;    // Exit the Application
}
```

# **Non-Buffered High-Speed Capture**

Non-buffered high-speed capture allows you to configure a single high-speed capture event. For multiple high-speed captures, you must reenable the high-speed capture each time it triggers.

# **High-Speed Capture Algorithm**

The following figure shows the basic algorithm for high-speed captures applicable to both C/C++ and LabVIEW code.

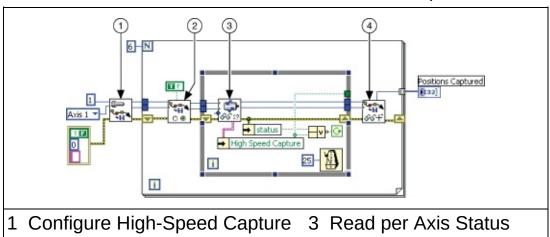


# **High-Speed Capture LabVIEW Diagram**

The following block diagrams demonstrate using NI-Motion to perform a high-speed capture, both with and without RTSI.

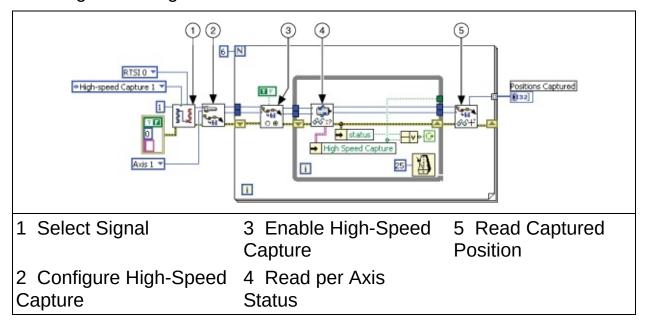


**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



2 Enable High-Speed Capture 4 Read Captured Position

To trigger the high-speed capture from a RTSI line, set the Destination parameter in Select Signal to High Speed Capture 1, as shown in the following block diagram.



## **High-Speed Capture C/C++ Code**

The following section includes C/C++ code for executing a high-speed capture, as well as using RTSI to execute a high-speed capture. The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
   u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
i32 capturedPositions[6];  // Array to store the captured positi
   i32 i;
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 1;
   // Set the axis number
   axis = 1;
   // Route breakpoint 1 to RTSI line 1
   err = flex_select_signal (boardID, NIMC_HS_CAPTURE1 /*destination
   CheckError;
   // Configure High Speed Capture
   err = flex_configure_hs_capture(boardID, axis, NIMC_HS_LOW_TO_HIG
   CheckError;
   for(i=0; i<6; i++){
       // Enable the high speed capture on axis
       err = flex_enable_hs_capture(boardID, axis, NIMC_TRUE);
       CheckError;
```

```
do
   {
     // Check the high speed capture status
     err = flex read axis status rtn(boardID, axis, &axisStatus)
     CheckError;
     // Read the communication status register and check the moa
     err = flex_read_csr_rtn(boardID, &csr);
     CheckError;
     // Check the modal errors
     if (csr & NIMC_MODAL_ERROR_MSG)
        err = csr & NIMC_MODAL_ERROR_MSG;
        CheckError;
     }
     Sleep (10); // Check every 10 ms
  }while (!(axisStatus & NIMC_HIGH_SPEED_CAPTURE_BIT));// Wait f
  err = flex_read_cap_pos_rtn(boardID, axis, &capturedPositions[
  CheckError;
return; // Exit the Application
// Error Handling
//
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC MODAL ERROR MSG){
  do{
     // Get the command ID, resource and the error code of the m
     // error from the error stack on the board
     flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
     nimcDisplayError(errorCode, commandID, resourceID);
     // Read the Communication Status Register
     flex_read_csr_rtn(boardID, &csr);
   }while(csr & NIMC_MODAL_ERROR_MSG);
}
else
          // Display regular error
  nimcDisplayError(err, 0, 0);
return; // Exit the Application
```

}

## **Real-Time System Integration Bus (RTSI)**

RTSI is a dedicated high-speed digital bus designed to facilitate system integration by low-level, high-speed, real-time communication between National Instruments devices.

Many applications, such as scanning and alignment, synchronize measurements made with data and image acquisition devices with position and velocity. This synchronization requires high speeds with low latencies.

Using RTSI, the NI motion controller can share high-speed digital signals with NI data acquisition devices, NI image acquisition devices, digital I/O, or other NI motion devices with no external cabling and without consuming bandwidth on the host bus. The RTSI bus also has built-in switching, so you can route signals to and from the bus on-the-fly using software.

In addition to the breakpoint and high speed capture signals, you can route encoder pulses over the RTSI lines, which serves as a way to trigger an external device on every change in the encoder channels. You can route phase A, phase B, and the index pulse of the encoder over RTSI.

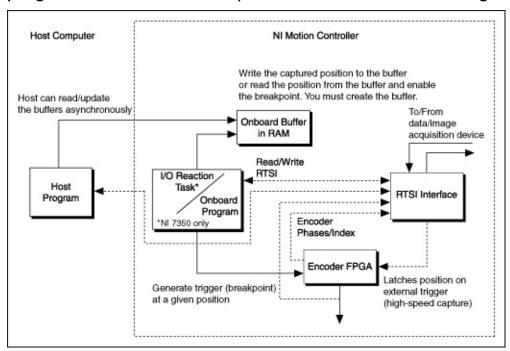
You also can create a software trigger by writing to the RTSI lines directly from software.

You can route position breakpoints and encoder pulses using the RTSI bus to trigger other devices. You also can configure data and image acquisition devices to trigger high-speed captures on the NI motion controllers using the RTSI bus.

## **RTSI Implementation on the Motion Controller**

You can configure an onboard buffer on the motion controller and use the buffered high-speed capture or breakpoint functionality to synchronize the motion application with data or image acquisition.

As shown in the following figure, the I/O reaction task automatically reenables the breakpoints or high-speed captures on the NI 7350 motion controller. On NI 7340 motion controllers, you must write an onboard program or use the host to perform the same re-enabling tasks.

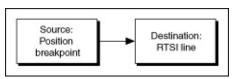


# **RTSI Input and Output**

The following sections explain the usage of RTSI for input and output signals.

### **Position Breakpoints Using RTSI**

You can use the Select Signal VI or <u>function</u> to route position breakpoints using one of the RTSI lines. In this case, the motion controller triggers the external device at a given position, as shown.

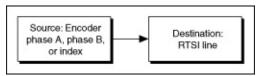




**Note** A breakpoint across RTSI always generates an active high pulse of 200 ns duration.

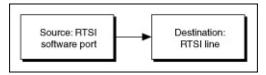
### **Encoder Pulses Using RTSI**

You may need to trigger the external device to acquire data every encoder phase or on an encoder index pulse, as shown.



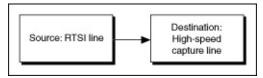
### **Software Trigger Using RTSI**

You can use the Set I/O Port MOMO VI or <u>function</u> to write directly to the RTSI lines to trigger other devices, as shown.



### **High-Speed Capture Input Using RTSI**

When the RTSI line receives the trigger from a data or image acquisition device, the corresponding high-speed capture occurs, as shown.



### **Torque Control**

To maintain constant torque or force, the sensor that returns the feedback to the motion controller must return a value proportional to the torque or force. The motion controller operates torque-control and position-control systems in much the same way. The main difference is that the feedback in position-control systems returns the current position, while the feedback in torque-control systems returns a voltage proportional to the current force or torque.

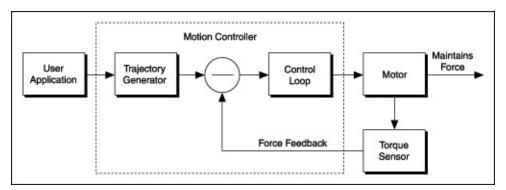
You can implement force feedback on NI motion controllers using either analog feedback or by monitoring force.



**Note** The NI SoftMotion Controller does not support analog feedback.

### **Analog Feedback**

In this mode, the torque or force sensor is connected to one of the analog inputs on the NI motion controller. That analog channel is used as the feedback sensor.

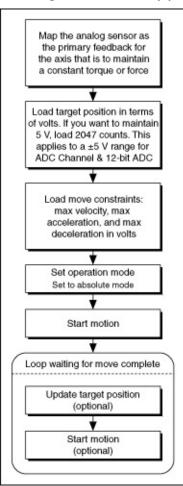


Tuning the control loop with a force sensor, which is an analog feedback sensor, produces the same results as with a position feedback sensor. Depending upon the resolution you are using, the system may require higher gains to ensure a faster response. NI motion controllers have 12-bit or 16-bit analog inputs, whose ranges can be set from 0 V to 5 V, 5 V to +5 V, 0 V to 10 V, and 10 V to +10 V. When you use counts for entering the values of position, velocity, acceleration, and deceleration, you do not need to enter the counts/revolution value for the axis.

Refer to the motion controller user manual for information about analog input ranges.

### **Torque Control Using Analog Feedback Algorithm**

The following figure shows the basic algorithm for torque control using analog feedback applicable to both C/C++ and LabVIEW code.

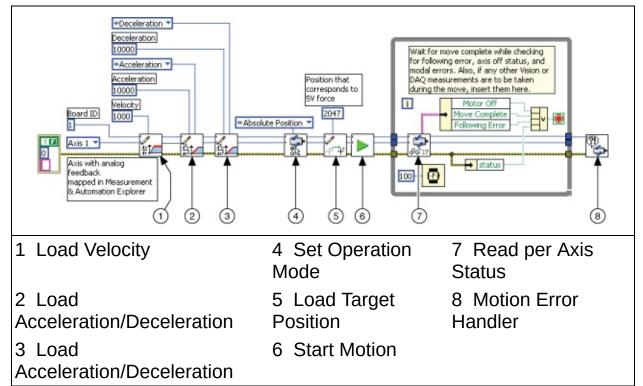


# Torque Control Using Analog Feedback LabVIEW Diagram

The following block diagram demonstrates using NI-Motion to perform torque control using analog feedback.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



# Torque Control Using Analog Feedback C/C++ Code

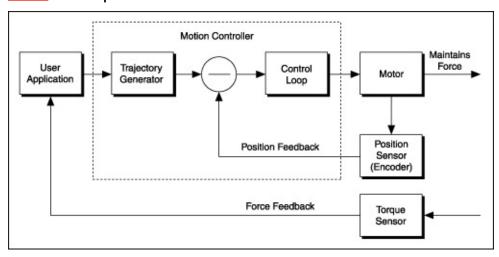
The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
u16 moveComplete;
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
   u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID= 1;
   // Set the axis number
   axis = 1;
   // Is is assumed that the axis being moved has an ADC channel
   // mapped as its primary feedback. Position is treated as binary
   // volts. Hence velocity is loaded in binary volts/sec and
   // acceleration as binary volts/sec^2.
   // Set the velocity for the move (in binary volts/sec)
   err = flex_load_velocity(boardID, axis, 10000, 0xFF);
   CheckError;
   // Set the acceleration for the move (in binary volts/sec^2)
   err = flex_load_acceleration(boardID, axis, NIMC_ACCELERATION, 10
   CheckError;
   // Set the deceleration for the move (in binary volts/sec^2)
   err = flex_load_acceleration(boardID, axis, NIMC_DECELERATION, 10
   CheckError;
```

```
// Set the jerk - s-curve time (in sample periods)
err = flex_load_scurve_time(boardID, axis, 1000, 0xFF);
CheckError;
// Set the operation mode
err = flex_set_op_mode (boardID, axis, NIMC_ABSOLUTE_POSITION);
CheckError;
// Load Position corresponding to the voltage which you want
// the motor to maintain (2047 \sim 5V \text{ in this example})
err = flex_load_target_pos (boardID, axis, 2047, 0xFF);
CheckError;
// Start the move
err = flex_start(boardID, axis, 0);
CheckError;
do
{
  axisStatus = 0;
  // Check the move complete status
  err = flex_check_move_complete_status(boardID, axis, 0, &moveC
  CheckError;
  // Check the following error/axis off status for axis 1
  err = flex read axis status rtn(boardID, axis, &axisStatus);
  CheckError;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG)
     err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
  }
}while (!moveComplete && !(axisStatus & NIMC_FOLLOWING_ERROR_BIT)
return; // Exit the Application
// Error Handling
//
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
```

### **Monitoring Force**

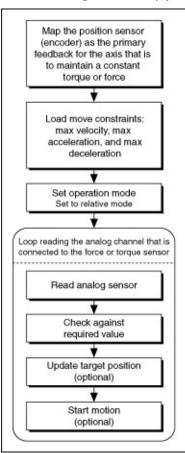
You can use this second force-feedback mode if you have a position sensor on the motor, in addition to the torque sensor. The control loop on the motion controller closes the position and velocity loops as usual. Use MAX to map the encoder as the feedback device for the axis.



For monitoring force, create an outer loop to monitor the torque sensor, and move the motor based on the value read from the torque sensor.

### **Torque Control Using Monitoring Force Algorithm**

The following figure shows the basic algorithm for torque control using monitoring force applicable to both C/C++ and LabVIEW code.

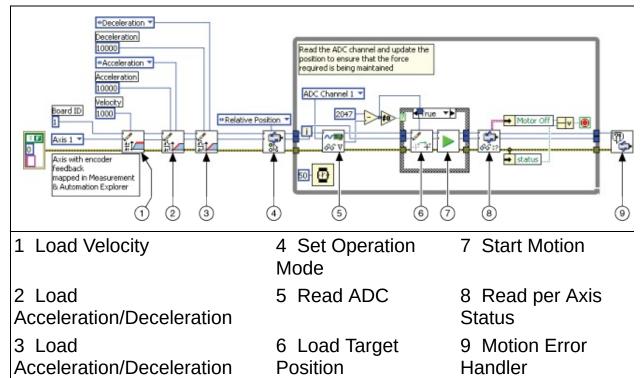


## **Torque Control Using Monitoring Force LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to perform torque control using monitoring force.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



# **Torque Control Using Monitoring Force C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
i32 constant;  // Constant force
i16 adcValue;  // ADC value read
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 1;
   // Set the axis number
   axis = 1:
   // constant force needed to be maintained
   // corresponds to 5V for a +/- 5V ADC settings
   constant = 2047;
   // Is is assumed that the axis being moved has an encoder
   // mapped as its primary feedback
   //-----
   // Set the velocity for the move (in counts/sec)
   err = flex load velocity(boardID, axis, 10000, 0xFF);
   CheckError;
   // Set the acceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, axis, NIMC_ACCELERATION, 10
   CheckError;
```

```
// Set the deceleration for the move (in counts/sec^2)
err = flex_load_acceleration(boardID, axis, NIMC_DECELERATION, 10
CheckError;
// Set the jerk (s-curve value) for the move (in sample periods)
err = flex load scurve time(boardID, axis, 100, 0xFF);
CheckError;
// Set the operation mode to velocity
err = flex set op mode(boardID, axis, NIMC RELATIVE POSITION);
CheckError;
do
{
   // Read the ADC channel number 1 and calculate the position to
   err = flex_read_adc_rtn(boardID, NIMC_ADC1, &adcValue);
   CheckError;
   if( (constant - adcValue) != 0){
      err = flex_load_target_pos(boardID, axis, (constant - adcVa
      CheckError;
      // Move based on delta force
      err = flex_start(boardID, axis, 0);
      CheckError;
   }
   // Check the move complete status/following error/axis off sta
   err = flex_read_axis_status_rtn(boardID, axis, &axisStatus);
   CheckError;
   // Read the communication status register and check the modal
   err = flex_read_csr_rtn(boardID, &csr);
   CheckError:
   // Check the modal errors
   if (csr & NIMC_MODAL_ERROR_MSG)
      err = csr & NIMC_MODAL_ERROR_MSG;
      CheckError;
   }
  Sleep (50); // Check every 10 ms
}while (!(axisStatus & NIMC AXIS OFF BIT)); // Exit on axis off
```

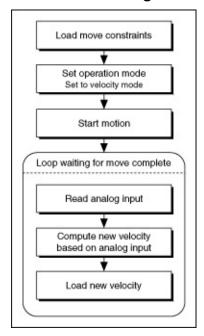
```
return; // Exit the Application
  // Error Handling
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC_MODAL_ERROR_MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID, &csr);
     }while(csr & NIMC_MODAL_ERROR_MSG);
  }
            // Display regular error
  else
     nimcDisplayError(err, 0, 0);
  return; // Exit the Application
}
```

### **Speed Control Based on Analog Value**

In a system where a feed roll must run at speeds based on an input voltage, the algorithm to maintain the speed consists of reading the analog voltage connected to one of the analog channels on the motion controller, and updating the speed of the axis based on the value of the voltage read. In this system, the feedback is a normal position sensor, such as an encoder.

### **Speed Control Based on Analog Feedback Algorithm**

The following figure shows the basic algorithm for speed control using based on analog feedback applicable to both C/C++ and LabVIEW code.



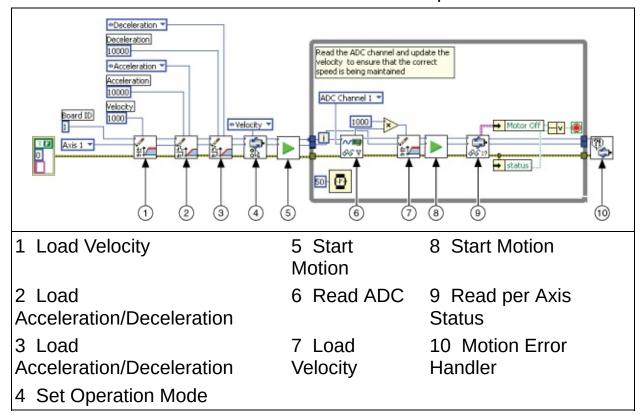
The analog input could be connected to a force sensor, which ensures that the tension of a web being fed is maintained.

### Speed Control Based on Analog Feedback LabVIEW Diagram

The following block diagram demonstrates using NI-Motion to perform speed control based on analog feedback.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



# Speed Control Based on Analog Feedback C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
   u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
i32 constant;  // Constant multiplier
i16 adcValue;  // ADC value read
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID= 1;
   // Set the axis number
   axis = 1;
   // constant to multiply the ADC value read to calculate the regu
   constant = 10;
   // Set the velocity for the move (in counts/sec)
   err = flex_load_velocity(boardID, axis, 10000, 0xFF);
   CheckError;
   // Set the acceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, axis, NIMC_ACCELERATION, 10
   CheckError;
   // Set the deceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, axis, NIMC_DECELERATION, 10
   CheckError;
   // Set the jerk (s-curve value) for the move (in sample periods)
   err = flex_load_scurve_time(boardID, axis, 100, 0xFF);
```

```
CheckError;
// Set the operation mode to velocity
err = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
CheckError;
// Start the move
err = flex_start(boardID, axis, 0);
CheckError;
do
{
  // Read the ADC channel number 1 and calculate the velocity to
  err = flex_read_adc_rtn(boardID, NIMC_ADC1, &adcValue);
  CheckError;
  // Set the velocity based on the ADC value read
  err = flex_load_velocity(boardID, axis, (adcValue * constant),
  CheckError;
  // Update the velocity
  err = flex_start(boardID, axis, 0);
  CheckError;
  // Check the move complete status/following error/axis off sta
  err = flex_read_axis_status_rtn(boardID, axis, &axisStatus);
  CheckError;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG)
     err = csr & NIMC MODAL ERROR MSG;
     CheckError;
  Sleep (50); // Check every 50 ms
}while (!(axisStatus & NIMC_AXIS_OFF_BIT)); // Exit on axis off
return; // Exit the Application
// Error Handling
```

```
//
   nimcHandleError; //NIMCCATCHTHIS:
   // Check to see if there were any Modal Errors
   if (csr & NIMC_MODAL_ERROR_MSG){
      do{
         // Get the command ID, resource and the error code of the m
         // error from the error stack on the board
         flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
         nimcDisplayError(errorCode, commandID, resourceID);
         // Read the Communication Status Register
         flex_read_csr_rtn(boardID, &csr);
      }while(csr & NIMC_MODAL_ERROR_MSG);
   }
   else
              // Display regular error
      nimcDisplayError(err, 0, 0);
   return; // Exit the Application
}
```

## **Onboard Programs**

This section provides information about how onboard programs work for the NI SoftMotion Controller and for NI 73xx motion controllers.

# Using Onboard Programs with the NI SoftMotion Controller

To use onboard programs with the NI SoftMotion Controller, use the LabVIEW Real-Time Module (RT) to target your application to run in the same environment as the NI SoftMotion Controller.

Because the NI SoftMotion Controller onboard program shares the same processor and system resources with the NI SoftMotion Controller, ensure you consider the following points before running your application in LabVIEW RT:

- Ensure that your top level VI is configured to run at normal, above normal, or high priority. If you are targeting LabVIEW RT for ETS, use the timed loop instead of changing the priority of your top level VI.
- Follow the guidelines in the *LabVIEW Real-Time Module User Manual*. The guidelines regarding memory allocation and using shared resources are especially important.
- The jitter of the system increases with the number of devices used in your RT system. Enable only the devices you need to use for the current application.
- Because interrupts cause jitter, National Instruments recommends you configure your application to poll for data periodically rather than wait on an interrupt.

You can further decrease the jitter under ETS by configuring the Ethernet mode to be polling. You configure these settings for the RT controller in Measurement & Automation Explorer (MAX).

Under LabVIEW RT, the NI SoftMotion Controller runs in the background at time critical priority. The NI SoftMotion Controller is designed to consume less than 40% of the processor bandwidth. The rate at which the NI SoftMotion Controller updates its data is typically 100 Hz.

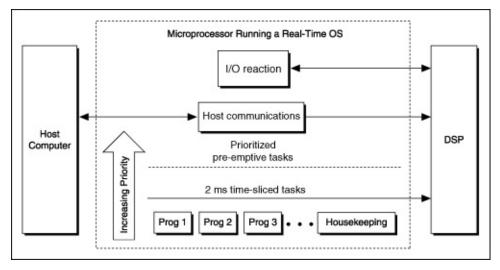
# Using Onboard Programs with NI 73xx Motion Controllers

You can use the real-time operating system on the NI 73xx motion controller to run custom programs. This functionality allows you to offload tasks from the host processor and onto the motion controller. Using onboard variables, which are global data on the device, arithmetic and loop operations, and efficient wait functions, you can write onboard programs to execute parts of the motion application with almost no host interaction. You can execute up to 10 onboard programs simultaneously.

Onboard programs have the least priority in a preemptive multitasking environment running on the embedded microprocessor because the primary function of the embedded processor is supervisory control and I/O reaction. Instead, the onboard programs run in a time-sliced manner at the lowest priority. Each onboard program gets a default time slice of two milliseconds, after which it relinquishes control of the processor to the next onboard program or housekeeping task.

The host communication and I/O reaction tasks take higher priority than the onboard programs and housekeeping tasks, as shown in the following figure. The onboard programs and housekeeping tasks are time-sliced among themselves.

For greater control and determinism for the motion control system, National Instruments offers the LabVIEW Real-Time (RT) module motion control system, which consists of a PXI chassis, PXI motion controller or controllers, LabVIEW RT, and NI-Motion driver software.





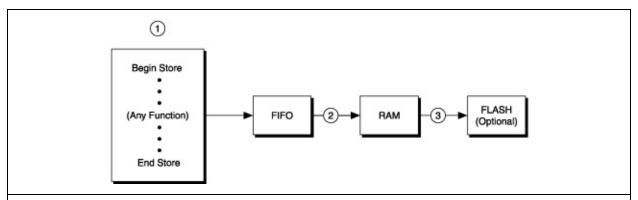
**Note** If you continuously poll data from the host, the onboard program gets preempted and has less time to run. To keep this from happening, insert a small delay in the polling loops on the host. Refer to <u>Timing Loops</u> for information about programming delays in the loops.

### **Writing Onboard Programs**



**Note** This section applies only to NI 73xx motion controllers.

Almost all NI-Motion functions that execute on the host can run onboard. The size and number of programs is completely flexible. It is ultimately limited by the 32 total memory objects in the Object Registry or by total available memory, whichever is reached first. Refer to Begin Program Storage VI or function for more information. These onboard programs remain on the motion controller until you reset it. If you want the onboard programs to persist through a reset of the motion controller, save them to FLASH, as shown.



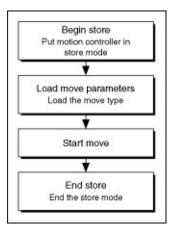
1 Write the program you want to load onto onboard memory.

You can use most NI-Motion functions between the Begin Program Storage VI or function

- and the End Program Storage VI or function.
- 2 Run the program to store the NI-Motion functions onto the onboard RAM.
- 3 Store the program to FLASH memory using the Object Memory Management VI or <u>function</u>
  - or MAX for more permanent storage (optional).

### **Onboard Program Algorithm**

The following figure shows an onboard program algorithm applicable to both C/C++ and LabVIEW code.

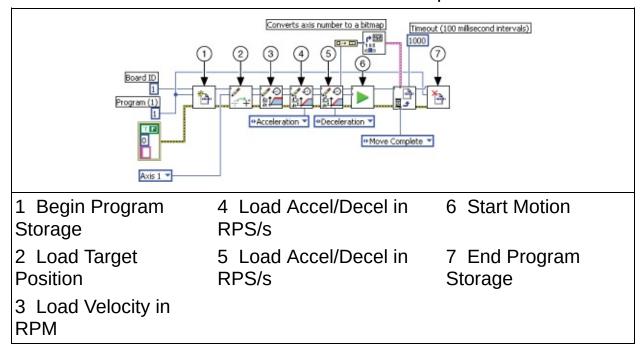


### **Onboard Program LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion with onboard programs.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



### **Onboard Program C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
  // Variables for modal error handling
  u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
  // Set the board ID
  boardID = 1;
  // Set the axis number
  axis = 1;
  //-----
  // Onboard program 1. This onboard program moves axis one
  // clockwise 5000 counts (steps). To execute this onboard progra
  // call the Run Program function
  //-----
  // Begin onboard program storage - program number 1
  err = flex begin store(boardID, 1);
  CheckError;
  // Set the operation mode to relative
  err = flex_set_op_mode(boardID, axis, NIMC_RELATIVE_POSITION);
  CheckError;
  // Load Target Position to move clockwise 5000 counts(steps)
  err = flex_load_target_pos(boardID, axis, 5000, 0xFF);
  CheckError;
  // Load Velocity in RPM
  err = flex_load_rpm(boardID, axis, 100.00, 0xFF);
  CheckError;
```

```
// Load Acceleration and Deceleration in RPS/sec
  err = flex_load_rpsps(boardID, axis, NIMC_BOTH, 50.00, 0xFF);
  CheckError;
  // Start the move
  err = flex_start(boardID, axis, 0);
  CheckError;
  // Wait for move to complete
  err = flex_wait_on_condition(boardID, 0, NIMC_WAIT, NIMC_CONDITI
                                  0, NIMC_MATCH_ALL, 1000 /*time o
  CheckError;
  // End Program Storage
  err = flex_end_store(boardID, 1);
  CheckError;
              // Exit the Application
  return;
  // Error Handling
  //
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC_MODAL_ERROR_MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID,&csr);
     }while(csr & NIMC MODAL ERROR MSG);
  else
             // Display regular error
     nimcDisplayError(err, 0, 0);
  return; // Exit the Application
}
```

## **Running, Stopping, and Pausing Onboard Programs**



**Note** This section applies only to NI 73xx motion controllers.

Use Run Program, Stop Program, and Pause/Resume Program to run, stop, and pause an onboard program that resides in the onboard memory of a motion controller.



**Note** Recursively calling an onboard program generates an error.

#### **Running an Onboard Program**

The Run Program VI or <u>function</u> executes previously stored programs from RAM or FLASH. Typically, you must call Run Program from the host, because it is not possible for an onboard program to run itself. However, it is possible to configure the motion controller to automatically run an onboard program upon powering up the motion control system. You also can call an onboard program from another onboard program using Run Program.

#### **Stopping an Onboard Program**

The Stop Program VI or <u>function</u> ends the execution of an onboard program that is currently running.

Stopping an onboard program using Stop Program completely ends execution. It is not possible to resume execution of the stopped onboard program, but you can re-run the program from the beginning.

You can stop an onboard program with a Stop Program VI or function call from the host or from another onboard program.



**Note** It is not possible for an onboard program to stop itself.



**Tip** Stopping an onboard program is different from stopping the motion of the axis or axes. When you stop an onboard program, any moves that have started continue to run. You must separately call the Stop Motion <u>VI</u> or <u>function</u> to stop the motion of the axis or axes.

#### Pausing/Resuming an Onboard Program

The Pause/Resume Program VI or <u>function</u> suspends execution of a running onboard program, or resumes execution of a previously paused onboard program.

You can pause an onboard program with a function call from the host, from the onboard program itself, or from another running onboard program. You can resume an onboard program with a function call from the host or from another running onboard program.



Note It is not possible for an onboard program to resume itself.



**Tip** Similar to the Stop Program VI or <u>function</u>, Pause/Resume Program has no effect on moves that have started.

#### **Automatic Pausing**

Any run-time error that occurs during execution automatically pauses the onboard program.

An onboard program also pauses automatically when it executes the Start Motion VI or <u>function</u> or the Blend Motion VI or <u>function</u> on an axis that has been stopped by the host, or when an axis is stopped due to a limit, home, software limit, or following error condition.

#### **Single-Stepping Using Pause**

You can use the Pause/Resume Program VI or function to effectively single-step through an onboard program. To single-step, add a Pause/Resume Program call after each function, and then resume the onboard program from the host.

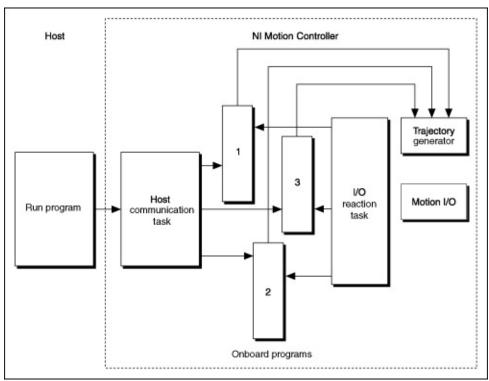
### **Conditionally Executing Onboard Programs**



**Note** This section applies only to NI 73xx motion controllers.

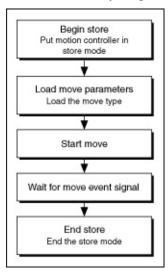
You can set conditions that affect the execution of the onboard programs. For example, you may want the onboard program to wait until a specific event occurs, and then continue executing.

The <u>Wait on Condition</u> VI or the <u>Wait on Event</u> function allows you to create onboard programs that wait for events, such as move complete and blend complete. These onboard programs can send functions to start moves and wait for moves to complete. The onboard program uses almost no processor time while waiting for an event such as move complete. When the move is complete, the trajectory generator enables the I/O reaction task, which causes the onboard program to continue executing the next function in its sequence, as shown in the following figure.



### **Onboard Program Conditional Execution Algorithm**

The following figure shows the basic algorithm for conditionally executing an onboard program applicable to both C/C++ and LabVIEW code.

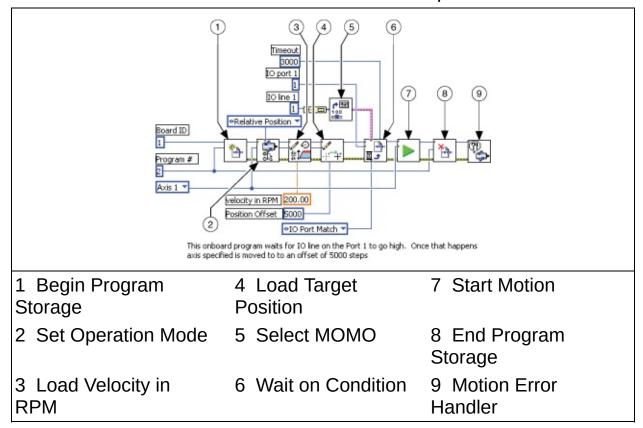


# Onboard Program Conditional Execution LabVIEW Diagram

The following block diagram demonstrates using NI-Motion to conditionally execute an onboard program.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



## Onboard Program Conditional Execution C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
   u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 1;
   // Set the axis number
   axis = 1;
   // Begin onboard program storage - program number 1
   err = flex begin store(boardID, 1);
   CheckError;
   // Load Velocity in RPM
   err = flex_load_rpm(boardID, axis, 100.00, 0xFF);
   CheckError;
   // Load Acceleration and Deceleration in RPS/sec
   err = flex_load_rpsps(boardID, axis, NIMC_BOTH, 50.00, 0xFF);
   CheckError;
   // Set the operation mode to relative
   err = flex_set_op_mode(boardID, axis, NIMC_RELATIVE_POSITION);
   CheckError;
   // Load Target Position to move relative 5000 counts(steps)
   err = flex_load_target_pos(boardID, axis, 5000, 0xFF);
```

```
CheckError;
// Wait for line 1 on port 1 to go active to finish executing
err = flex_wait_on_condition(boardID, NIMC_IO_PORT1, NIMC_WAIT,
                               0, NIMC_MATCH_ALL, 10000 /*time
CheckError;
// Start the move
err = flex_start(boardID, axis, 0);
CheckError;
// Wait for move to complete
err = flex_wait_on_condition(boardID, 0, NIMC_WAIT, NIMC_CONDITI
                               O, NIMC_MATCH_ALL, 1000 /*time o
CheckError;
// End Program Storage
err = flex end store(boardID, 1);
CheckError;
       // Exit the Application
return;
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG){
  do{
     // Get the command ID, resource and the error code of the m
     // error from the error stack on the board
     flex_read_error_msg_rtn(boardID,&commandID,&resourceID,&err
     nimcDisplayError(errorCode, commandID, resourceID);
     // Read the Communication Status Register
     flex_read_csr_rtn(boardID, &csr);
  }while(csr & NIMC_MODAL_ERROR_MSG);
}
else
          // Display regular error
  nimcDisplayError(err, 0, 0);
return; // Exit the Application
```

}

## **Using Onboard Memory and Data**

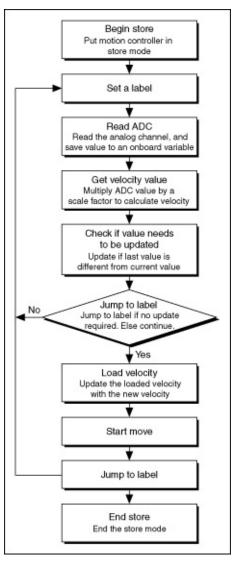


**Note** This section applies only to NI 73xx motion controllers.

NI motion controllers allow you to access the onboard RAM and FLASH to create data buffers and use some general-purpose onboard variables for data manipulation. You can use this memory to update data that is loaded by functions that are executing in an onboard program. You also can synchronize execution or data between the host computer and the motion controller. For example, you may want to update the velocity of an axis based on the analog voltage read from an ADC channel. This memory is statically allocated.

#### **Updating Velocity Based on ADC Channel Algorithm**

The following figure shows the basic algorithm for updating velocity based on an ADC channel using an onboard program applicable to both C/C++ and LabVIEW code.



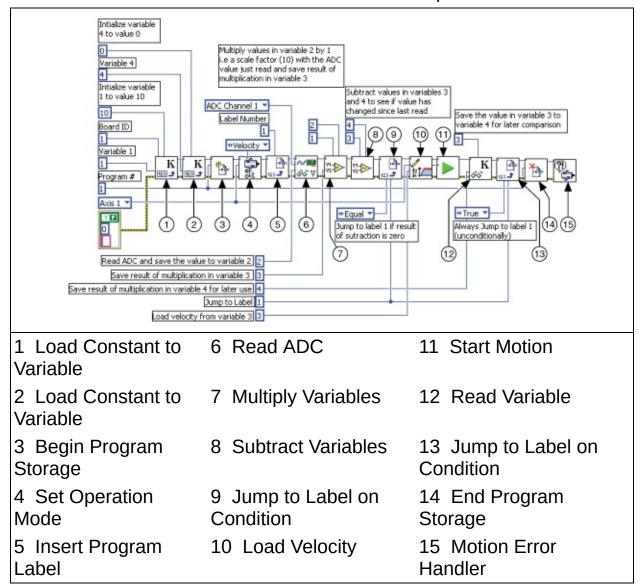
Before you execute this program, set the operation mode of the axis to velocity mode.

# Updating Velocity Based on ADC Channel LabVIEW Diagram

The following block diagram demonstrates using NI-Motion to update velocity based on an ADC channel.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



## **Updating Velocity Based on ADC Channel C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
i32 constant;  // Constant multiplier
  // Variables for modal error handling
  // Set the board ID
  boardID= 1;
  // Set the axis number
  axis = 1:
  // constant to multiply the ADC value read to calculate the regu
  constant = 10;
  // Initialize onboard variable 4 to 0
  err = flex_load_var(boardID, 0, 4);
  CheckError;
  // Initialize onboard variable 1 to the constant multiplier
  err = flex_load_var(boardID, constant, 1);
  CheckError;
  // Begin onboard program storage - program number 1
  err = flex_begin_store(boardID, 1);
  // Set the operation mode to velocity
  err = flex_set_op_mode(boardID, axis, NIMC_VELOCITY);
  CheckError:
  // Insert Label number 1
  err = flex_insert_program_label(boardID, 1);
```

```
CheckError;
// Read ADC channel and store ADC value in variable 2
err = flex read adc(boardID, NIMC ADC1, 2);
CheckError;
// Multiply Variables 2 i.e. the ADC value with 1 i.e. the consta
// Save the result in variable 3
err = flex_mult_vars(boardID, 1, 2, 3);
CheckError;
// Subtract value in variable 3 from variable 4
// We are not interested in the result - just want to
// set the condition on board.
err = flex_sub_vars(boardID, 3, 4, 0);
CheckError;
// Jump to label 1 as the subtraction above set the condition
// to "equal to zero" which implies that the values in variable
// 3 and 4 are the same
err = flex jump label on condition (boardID, 0, NIMC CONDITION E
// Set the velocity for the move (in counts/sec) by loading the
// value from variable 3 which is (adc value * constant)
err = flex_load_velocity(boardID, axis, 0, 3);
CheckError:
// Start the move to update the velocity
err = flex start(boardID, axis, 0);
CheckError;
// Save the value in variable 3 to variable 4 for use
// in next cvcle
err = flex_read_var(boardID, 3, 4);
CheckError;
// Jump back to label 1 unconditionally
err = flex_jump_label_on_condition (boardID, 0, NIMC_CONDITION_T
CheckError;
// End Program Storage
err = flex_end_store(boardID, 1);
// To execute this program use the Run Program function
return; // Exit the Application
```

```
// Error Handling
  //
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC_MODAL_ERROR_MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID, &csr);
     }while(csr & NIMC_MODAL_ERROR_MSG);
  }
  else
            // Display regular error
     nimcDisplayError(err, 0, 0);
  return; // Exit the Application
}
```

## **Branching Onboard Programs**

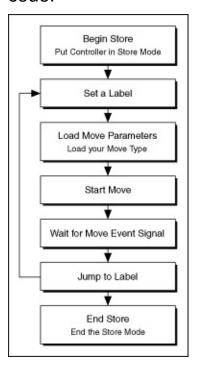


**Note** This section applies only to NI 73xx motion controllers.

To create loops, or conditional if statements, insert labels in the program you are storing and use the <u>Jump to Label on Condition</u> VI or the <u>Jump on Event</u> function to jump to that label based on the condition.

## **Branching Onboard Programs Algorithm**

The following figure shows an onboard program waiting for an I/O line to go active before starting a move applicable to both C/C++ and LabVIEW code.

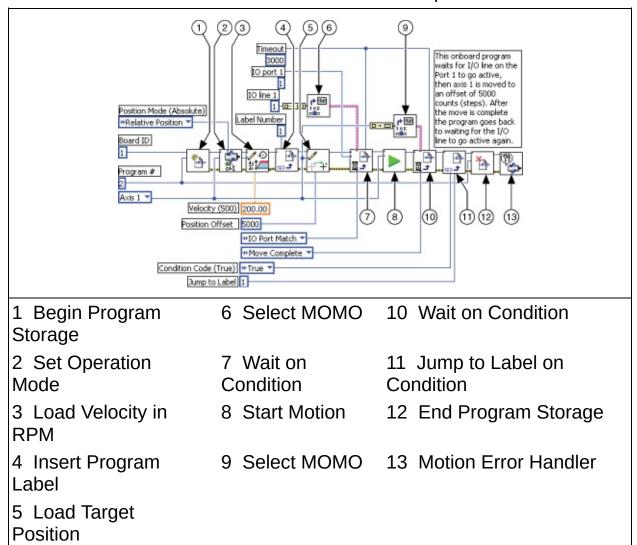


## **Branching Onboard Programs LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to add conditional jumps in an onboard program.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



## **Branching Onboard Programs C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
   u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 1;
   // Set the axis number
   axis = 1;
   // Begin onboard program storage - program number 1
   err = flex_begin_store(boardID, 1);
   CheckError;
   // Load Velocity in RPM
   err = flex_load_rpm(boardID, axis, 100.00, 0xFF);
   CheckError;
   // Load Acceleration and Deceleration in RPS/sec
   err = flex_load_rpsps(boardID, axis, NIMC_BOTH, 50.00, 0xFF);
   CheckError;
   // Set the operation mode to relative
   err = flex set op mode(boardID, axis, NIMC RELATIVE POSITION);
   CheckError;
   // Insert Label number 1
   err = flex_insert_program_label(boardID, 1);
   CheckError;
   // Load Target Position to move relative 5000 counts(steps)
```

```
err = flex_load_target_pos(boardID, axis, 5000, 0xFF);
CheckError;
// Wait for line 1 on port 1 to go active to finish executing
err = flex_wait_on_condition(boardID, NIMC_IO_PORT1, NIMC_WAIT,
                               0, NIMC MATCH ALL, 10000 /*time
CheckError;
// Start the move
err = flex start(boardID, axis, 0);
CheckError;
// Wait for move to complete
err = flex_wait_on_condition(boardID, 0, NIMC_WAIT, NIMC_CONDITI
                               O, NIMC_MATCH_ALL, 1000 /*time o
CheckError;
// Jump unconditionally to label 1 and check IO line again
err = flex_jump_label_on_condition (boardID, 0, NIMC_CONDITION_T
CheckError;
// End Program Storage
err = flex end store(boardID, 1);
CheckError;
return; // Exit the Application
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG){
   do{
     // Get the command ID, resource and the error code of the m
     // error from the error stack on the board
     flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
     nimcDisplayError(errorCode, commandID, resourceID);
     // Read the Communication Status Register
     flex_read_csr_rtn(boardID,&csr);
  }while(csr & NIMC MODAL ERROR MSG);
else
         // Display regular error
  nimcDisplayError(err, 0, 0);
return; // Exit the Application
```

### **Math Operations**



**Note** This section applies only to NI 73xx motion controllers.

NI-Motion always performs math operations on values stored in onboard variables, and all math operations set a global condition that the <u>Jump to Label on Condition</u> VI or the <u>Jump on Event</u> function uses to determine if the operation jumps to a particular label in the onboard program.

To load the onboard variables, use the Load Constant to Variable VI or function or point the return vector in the Read functions to the onboard variable where you want the data to be saved. In the previous example, the ADC channel is read to onboard variable 2. This value is then multiplied with a scale factor loaded into variable 1 using the Load Constant to Variable VI or function.

You can perform Add, Multiply, Subtract, Divide, AND, OR, XOR, NOT, and logical shift math operations. The condition code always reflects the last math operation performed. Less Than implies less than zero, Equal implies equal to zero, and so on.

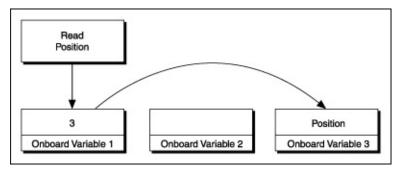
#### **Indirect Variables**



**Note** This section applies only to NI 73xx motion controllers.

If you make the read or load functions point to variables 0x81 to 0xF8, the functions use the value loaded in variables 1 to 0x78 and interpret them as the address where the value is read or loaded. This creates two levels of indirection.

Making the return vector of the Read Position VI or <u>function</u> point to 0x81 causes the position to end up in the address contained in onboard variable 1, as shown.



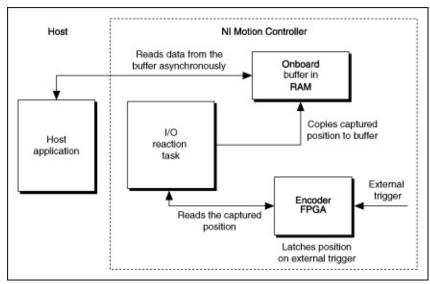
Using indirect variables can be very useful in looping in onboard programs, as well as dynamically changing the input values to functions.

#### **Onboard Buffers**



**Note** This section applies only to NI 73xx motion controllers.

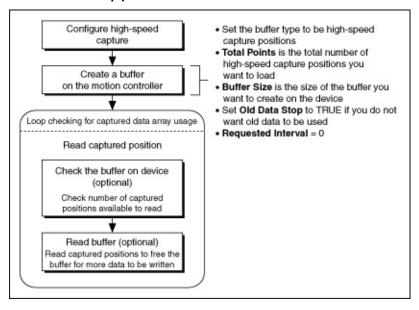
You can use the memory on the NI motion controllers to create generalpurpose buffers to read and write data, as shown in the following figure.



Buffers are created from a dynamic pool of memory, so you must free the memory when the buffer is not required. This same pool of memory is used to store onboard programs in RAM. As the number or size of buffers increases, the available memory for storing onboard programs decreases.

#### **Onboard Buffers Algorithm**

The following figure shows the algorithm for using onboard buffers to store data applicable to both C/C++ and LabVIEW code.



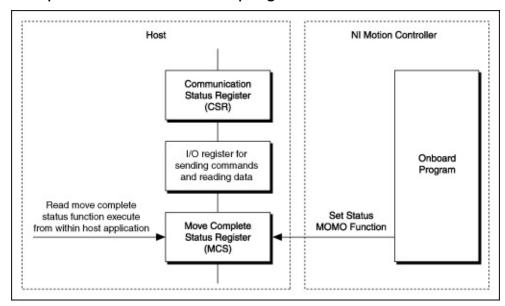
# Synchronizing Host Applications with Onboard Programs



**Note** This section applies only to NI 73xx motion controllers.

The host and the onboard program can write to the move complete status (MCS) register using the Set User Status MOMO VI or function. This VI or function controls the upper three bits in the MCS register using the MustOn/MustOff (MOMO) protocol.

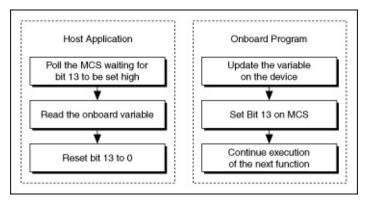
Use these bits to synchronize an application running on the host computer with an onboard program, as shown in the following figure.



For example, consider a host application that reads an onboard variable that has been updated by an onboard program. Use the following algorithm to synchronize the host application with an onboard program, and read an onboard variable that has been updated by an onboard program.

## **Synchronizing Host Applications with Onboard Programs Algorithm**

The following figure shows an algorithm for using an onboard program to synchronize host applications applicable to both C/C++ and LabVIEW code.

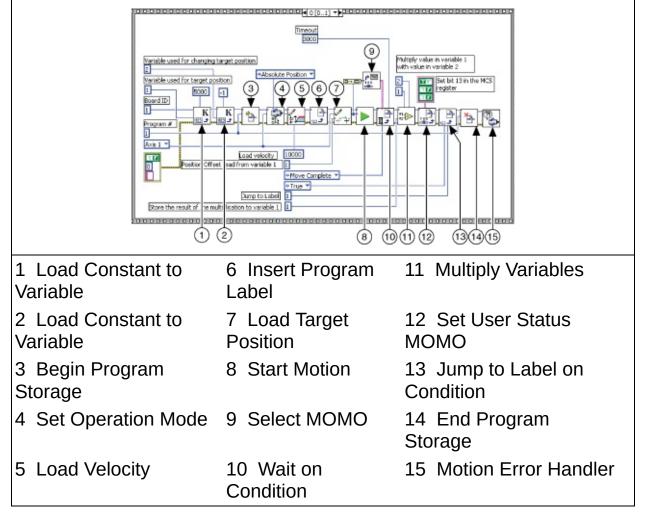


## Synchronizing Host Applications with Onboard Programs LabVIEW Diagram

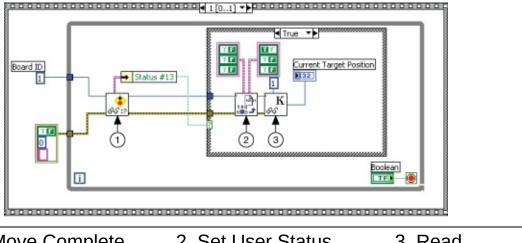


**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.

This example moves axis 1 between target positions of 5000 and 5000. The host reads the target position only after the move has completed, and the new target position has been calculated. The following figure shows the code that runs as an onboard program.



The following figure shows the code that runs on the host.



1 Read Move Complete Status

2 Set User Status MOMO

3 Read Variable



**Note** As the host is polling a register on the motion controller, it is not invoking the Host Communication Task on the real-time operating system on the motion controller. Therefore, the onboard programs executing are not preempted. In this situation, the onboard programs run deterministically.

# Synchronizing Host Applications with Onboard Programs C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  // Variables for modal error handling
  u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
  // Set the board ID
  boardID = 1;
  // Set the axis number
  axis = 1:
  // Set the move length
  targetPosition = 5000;
  // Set the multiplier
  multiplier = -1;
  //-----
  // Onboard program. This onboard moves an axis back and
  // forth between targetPosition and -targetPosition. Before
  // reversing directions it indicates to the host computer that
  // it is about to do so.
  //-----
  // Initialize onboard variable 2 to the multiplier used to change
  // target position
  err = flex_load_var(boardID, multiplier, 2);
  CheckError;
```

```
// Initialize onboard variable 1 to the target position
err = flex_load_var(boardID, targetPosition, 1);
CheckError;
// Begin onboard program storage - program number 1
err = flex begin store(boardID, 1);
// Set the operation mode to absolute position
err = flex set op mode(boardID, axis, NIMC ABSOLUTE POSITION);
CheckError;
// Set the velocity
err = flex_load_velocity(boardID, axis, 10000, 0xFF);
CheckError;
// Insert Label number 1
err = flex insert program label(boardID, 1);
CheckError;
// Load Target Position from onboard variable 1
err = flex_load_target_pos(boardID, axis, 0, 1);
CheckError;
// Start the move
err = flex_start(boardID, axis, 0);
CheckError;
// Wait for move to complete
err = flex_wait_on_condition(boardID, 0, NIMC_WAIT, NIMC_CONDITI
                                 0, NIMC_MATCH_ALL, 3000 /*time o
CheckError;
// Multiply Variables 1 i.e. the target position with 2 i.e. the
// Save the result in variable 1 - this calculates the negative c
err = flex_mult_vars(boardID, 1, 2, 1);
CheckError;
// Set the 13th bit in the move complete status register so that
// knows that the axis is about to reverse direction
err = flex_set_status_momo(boardID, 0x20, 0);
CheckError;
// Jump unconditionally to load new target position
err = flex jump label on condition (boardID, 0, NIMC CONDITION T
CheckError;
// End Program Storage
err = flex end store(boardID, 1);
```

```
CheckError;
// Host program. This programs monitors the 13th bit in the
// move complete status register and records the position
// the axis is going to move to
//-----
do
{
  // Check the move complete status/following error/axis off sta
  err = flex read axis status rtn(boardID, axis, &axisStatus);
  CheckError;
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError;
  // Check the modal errors
  if (csr & NIMC MODAL ERROR MSG)
     err = csr & NIMC_MODAL_ERROR_MSG;
     CheckError;
  }
  // Read the move complete status register and once the 13th bi
  // reset the bit and reads the target position
  err = flex_read_mcs_rtn(boardID, &moveCompleteStatus);
  CheckError;
   if(moveCompleteStatus & (1<<13)){</pre>
      i32 currentTargetPosition;
     // Reset the 13th bit in the move complete status register
     err = flex_set_status_momo(boardID, 0, 0x20);
     CheckError;
     err = flex_read_var_rtn(boardID, 1, &currentTargetPosition)
     CheckError;
  }
  Sleep (50); // Check every 50 ms
}while (!(axisStatus & NIMC FOLLOWING ERROR BIT) && !(axisStatus
return; // Exit the Application
```

```
// Error Handling
  //
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC_MODAL_ERROR_MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID, &csr);
     }while(csr & NIMC_MODAL_ERROR_MSG);
  }
  else
            // Display regular error
     nimcDisplayError(err, 0, 0);
  return; // Exit the Application
}
```

## **Onboard Subroutines**

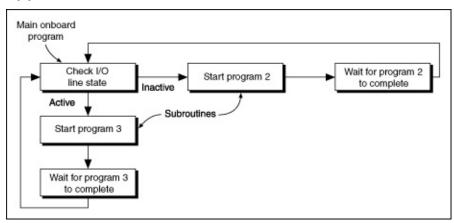


**Note** This section applies only to NI 73xx motion controllers.

You can create subroutines to run as onboard programs and execute them from within an onboard program.

### **Onboard Subroutine Algorithm**

The following figure shows an onboard program algorithm that checks the I/O line state to determine which onboard subroutine to execute applicable to both C/C++ and LabVIEW code.



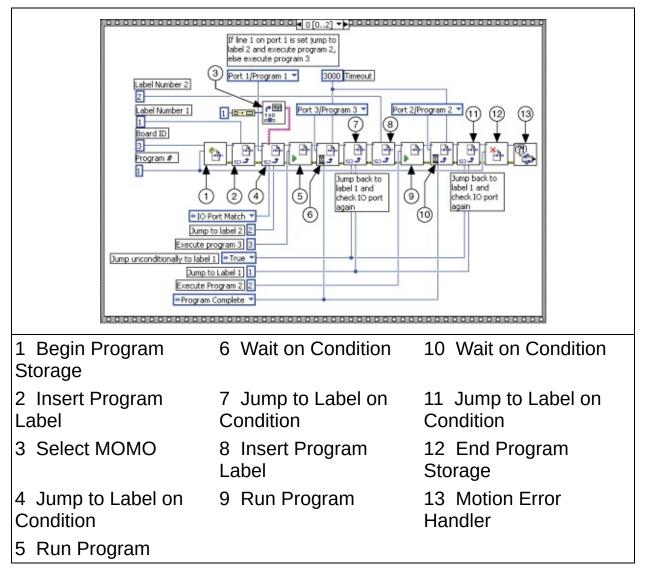
If the I/O line is active, the main onboard program calls an onboard subroutine that rotates the motor clockwise. If the I/O line is inactive, the main onboard program calls an onboard subroutine that rotates the motor counterclockwise.

### **Onboard Subroutine LabVIEW Diagram**

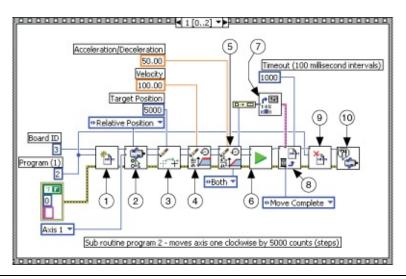


**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.

The following figure shows the main onboard program used to determine the subroutine call.



The following figure shows the subroutine that causes the motor to rotate clockwise.



1 Begin Program Storage

2 Set Operation

Mode

3 Load Target Position

4 Load Velocity in RPM

5 Load Accel/Decel in RPS/sec

6 Start Motion

7 Select MOMO

8 Wait on Condition

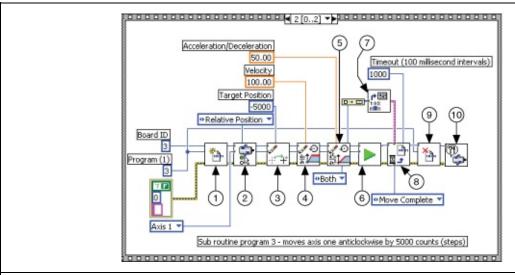
9 End Program

Storage

10 Motion Error

Handler

The following figure shows the subroutine that causes the motor to rotate counter clockwise.



1 Begin Program Storage

5 Load Accel/Decel in RPS/sec

8 Wait on Condition

2 Set Operation Mode	6 Start Motion	9 End Program Storage
3 Load Target Position	7 Select MOMO	10 Motion Error Handler
4 Load Velocity in RPM		

#### **Onboard Subroutine C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u8 boardID;  // Board identification number
u8 axis;  // Axis number
u16 csr = 0;  // Communication status register
  // Variables for modal error handling
  u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
  // Set the board ID
  boardID = 1;
  // Set the axis number
  axis = 1;
  //-----
  // Onboard program 2. This onboard program moves axis one
  // clockwise 5000 counts (steps). This onboard program
  // is executed by onboard program one.
  //-----
  // Begin onboard program storage - program number 2
  err = flex begin store(boardID, 2);
  CheckError;
  // Set the operation mode to relative
  err = flex_set_op_mode(boardID, axis, NIMC_RELATIVE_POSITION);
  CheckError;
  // Load Target Position to move clockwise 5000 counts(steps)
  err = flex_load_target_pos(boardID, axis, 5000, 0xFF);
  CheckError;
  // Load Velocity in RPM
  err = flex_load_rpm(boardID, axis, 100.00, 0xFF);
  CheckError;
```

```
// Load Acceleration and Deceleration in RPS/sec
err = flex_load_rpsps(boardID, axis, NIMC_BOTH, 50.00, 0xFF);
CheckError;
// Start the move
err = flex_start(boardID, axis, 0);
CheckError;
// Wait for move to complete
err = flex_wait_on_condition(boardID, 0, NIMC_WAIT, NIMC_CONDITI
                              0, NIMC_MATCH_ALL, 1000 /*time o
CheckError;
// End Program Storage
err = flex end store(boardID, 2);
CheckError;
//-----
// Onboard program 3. This onboard program moves axis one
// counter clockwise 5000 counts (steps). This onboard program
// is executed by onboard program one.
//-----
// Begin onboard program storage - program number 3
err = flex begin store(boardID, 3);
CheckError;
// Set the operation mode to relative
err = flex_set_op_mode(boardID, axis, NIMC_RELATIVE_POSITION);
CheckError;
// Load Target Position to move counter clockwise 5000 counts(ste
err = flex load target pos(boardID, axis, -5000, 0xFF);
CheckError;
// Load Velocity in RPM
err = flex load rpm(boardID, axis, 100.00, 0xFF);
CheckError;
// Load Acceleration and Deceleration in RPS/sec
err = flex_load_rpsps(boardID, axis, NIMC_BOTH, 50.00, 0xFF);
CheckError;
// Start the move
err = flex_start(boardID, axis, 0);
CheckError;
// Wait for move to complete
```

```
err = flex_wait_on_condition(boardID, 0, NIMC_WAIT, NIMC_CONDITI
                                0, NIMC_MATCH_ALL, 1000 /*time o
CheckError;
// End Program Storage
err = flex end store(boardID, 3);
CheckError;
// Onboard program 1. The main onboard program monitors
// an IO line and based on state of the IO line executes
// onboard program 2 or onboard program 3
//-----
// Begin onboard program storage - program number 1
err = flex begin store(boardID, 1);
CheckError;
// Insert Label number 1
err = flex insert program label(boardID, 1);
CheckError;
// Jump to label 2 if the line 1 on port one is active
err = flex_jump_label_on_condition (boardID, NIMC_IO_PORT1, NIMC
CheckError;
// If the above jump failed i.e. the IO line is not active
// execute program #3
err = flex_run_prog(boardID, 3);
CheckError;
// Wait for program #3 to finish executing
err = flex_wait_on_condition(boardID, 3 /*program #*/, NIMC_WAIT
CheckError;
// Jump unconditionally to label 1 and check IO line again
err = flex jump label on condition (boardID, 0, NIMC CONDITION T
CheckError;
// Insert Label number 2
err = flex_insert_program_label(boardID, 2);
CheckError;
// Execute program #2
err = flex_run_prog(boardID, 2);
CheckError;
// Wait for program #2 to finish executing
```

```
err = flex_wait_on_condition(boardID, 2 /*program #*/, NIMC_WAIT
                                  0, NIMC MATCH ALL, 1000 /*time o
  CheckError;
  // Jump unconditionally to label 1 and check IO line again
  err = flex_jump_label_on_condition (boardID, 0, NIMC_CONDITION_T
  CheckError:
//
  // End Program Storage
  err = flex_end_store(boardID, 1);
  CheckError;
  return;
              // Exit the Application
  // Error Handling
  //
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC_MODAL_ERROR_MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex read error msq rtn(boardID, &commandID, &resourceID, &err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID,&csr);
     }while(csr & NIMC_MODAL_ERROR_MSG);
  }
  else
             // Display regular error
     nimcDisplayError(err, 0, 0);
  return;
            // Exit the Application
}
```

#### **Automatically Starting Onboard Programs**



**Note** This section applies only to NI 73xx motion controllers.

You can configure the onboard program to start automatically without calling the Run Program VI or <u>function</u>. The onboard program runs as soon as the motion controller exits the reset state.

To use this feature, save the onboard program to FLASH, and then call the Enable Auto Start VI or <u>function</u>. The motion controller checks to see if the auto-start flag is set when it boots up. If the flag is set, the motion controller executes the onboard program configured to automatically start. The auto-start requires no host interaction after it is set up.

Automatically starting the onboard programs is very useful if you need to execute monitoring tasks to begin as soon as the computer and controller boot up.

#### **Changing a Time Slice**



**Note** This section applies only to NI 73xx motion controllers.

Use the Load Program Time Slice VI or <u>function</u> to specify the minimum time an onboard program has to be run per watchdog period, with a total of 20 ms allowed for all running onboard programs. The default value of 2 ms allows a maximum of 10 onboard programs running simultaneously with equal time slices.

You can increase the time slice of the program to change its performance. The higher you set the time slice, the more the program can execute, because it commands more processor time.

However, because the processing power is being held longer by the onboard program, the response times of other onboard programs are slower. Also, increasing the time slice of a program may reduce host responsiveness and increase I/O reaction time, even though host communications and I/O reaction have higher priorities than onboard programs because the motion controller must guarantee that every program runs for its allotted time per watchdog period.

#### **Creating Applications Using NI-Motion**

You can combine the moves, input/output, and other functionality discussed in <u>Programming with NI-Motion</u>, to create complete motion control applications.

The following sections show examples of typical motion control applications and how they are implemented using NI-Motion.

- Scanning
- Rotating Knife

#### **Scanning**

The goal of the scanning application is to inspect a wafer under a fixed laser. Multiple detectors collect the scattered laser light and feed the data to an analysis system that maps any defects.

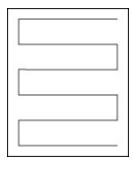
The wafer rests on an XY stage that moves in two dimensions. The objective of the scan is to cover as much space on the wafer as possible in the shortest amount of time. Scanning a greater area increases the chances of detecting all defects. Shortening the scan time lowers the cycle time, and increases the speed of the production or testing.

You can perform a scanning application in one of the following three ways:

- Move the stage in a raster by <u>connecting several straight-line move</u> <u>segments</u>.
- Use <u>blending to perform the scan in a single continuous move</u>.
- Use contouring to create a custom scanning path for the stage.

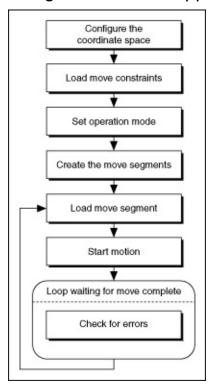
### **Connecting Straight-Line Move Segments**

You can cover the entire area of the wafer by varying the size of the raster area. You can increase the resolution of the scanning path by shortening the distance of the vertical straight-line moves. However, remember that increasing the resolution also increases the cycle time.



#### **Raster Scanning Using Straight Lines Algorithm**

The following figure shows the basic algorithm for raster scanning for straight-line moves applicable to both C/C++ and LabVIEW code.



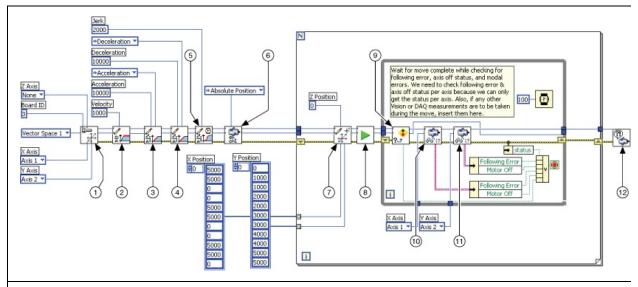
The raster scanning algorithm for straight-line moves stops the motors after every segment of the move, so the cycle time is longer than other methods.

# Raster Scanning Using Straight Lines LabVIEW Diagram

The following block diagram demonstrates using NI-Motion to perform raster scanning using straight lines.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



1 Configure Vector Space	5 Load S-Curve Time	9 Check Move Complete Status
2 Load Velocity	6 Set Operation Mode	10 Read per Axis Status
3 Load Acceleration/Deceleration	7 Load Vector Space Position	11 Read per Axis Status
4 Load Acceleration/Deceleration	8 Start Motion	12 Motion Error Handler

### Raster Scanning Using Straight Lines C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

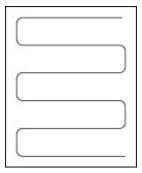
```
// Main function
void main(void){
        u8 boardID;  // Board identification number
u8 vectorSpace;  // Vector space number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
        u16 status;
        u16 moveComplete;
        i32 xPosition[11] = {5000, 5000, 0, 0, 5000, 5000, 0, 0, 5000, 50
        i32 yPosition[11] = \{0, 1000, 1000, 2000, 2000, 3000, 3000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 4000, 40
        // Variables for modal error handling
        // Set the board ID
        boardID = 1;
        // Set the axis number
        vectorSpace = NIMC_VECTOR_SPACE1;
         // Configure a 2D Vector Space comprising of axes 1, and 2
        err = flex_config_vect_spc(boardID, vectorSpace, 1, 2, 3);
        CheckError;
        // Set the velocity for the move (in counts/sec)
        err = flex_load_velocity(boardID, vectorSpace, 10000, 0xFF);
        CheckError;
        // Set the acceleration for the move (in counts/sec^2)
        err = flex_load_acceleration(boardID, vectorSpace, NIMC_ACCELERAT
        CheckError;
        // Set the deceleration for the move (in counts/sec^2)
        err = flex_load_acceleration(boardID, vectorSpace, NIMC_DECELERAT
        CheckError;
```

```
// Set the jerk or s-curve in sample periods
err = flex_load_scurve_time(boardID, vectorSpace, 100, 0xFF);
CheckError;
// Set the operation mode to absolute position
err = flex set op mode(boardID, vectorSpace, NIMC ABSOLUTE POSITI
CheckError;
// Load the straight line segments one by one
for (i=0; i<11; i++){
   // Load Target Position
   err = flex_load_vs_pos(boardID, vectorSpace, xPosition[i], yPo
   CheckError;
   // Start the move
   err = flex_start(boardID, vectorSpace, 0);
   CheckError;
   do
      axisStatus = 0;
      //Check the move complete status
      err = flex_check_move_complete_status(boardID, vectorSpace,
      CheckError;
      // Check the following error/axis off status for axis 1
      err = flex_read_axis_status_rtn(boardID, 1, &status);
      CheckError;
      axisStatus |= status;
      // Check the following error/axis off status for axis 2
      err = flex_read_axis_status_rtn(boardID, 2, &status);
      CheckError;
      axisStatus |= status;
      // Read the communication status register and check the moa
      err = flex_read_csr_rtn(boardID, &csr);
      CheckError:
      // Check the modal errors
      if (csr & NIMC_MODAL_ERROR_MSG)
         err = csr & NIMC MODAL ERROR MSG;
         CheckError;
      Sleep(50); // Check every 50 ms
```

```
}while (!moveComplete && !(axisStatus & NIMC_FOLLOWING_ERROR_B
     if( (axisStatus & NIMC_FOLLOWING_ERROR_BIT) || (axisStatus & N
        break; // Break out of the for loop as an axis was killed
     }
  }
  return; // Exit the Application
  // Error Handling
  //
  nimcHandleError; //NIMCCATCHTHIS:
  // Check to see if there were any Modal Errors
  if (csr & NIMC MODAL ERROR MSG){
     do{
        // Get the command ID, resource and the error code of the m
        // error from the error stack on the board
        flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
        nimcDisplayError(errorCode, commandID, resourceID);
        // Read the Communication Status Register
        flex_read_csr_rtn(boardID,&csr);
     }while(csr & NIMC_MODAL_ERROR_MSG);
  }
            // Display regular error
  else
     nimcDisplayError(err, 0, 0);
  return; // Exit the Application
}
```

#### **Blending Straight-Line Move Segments**

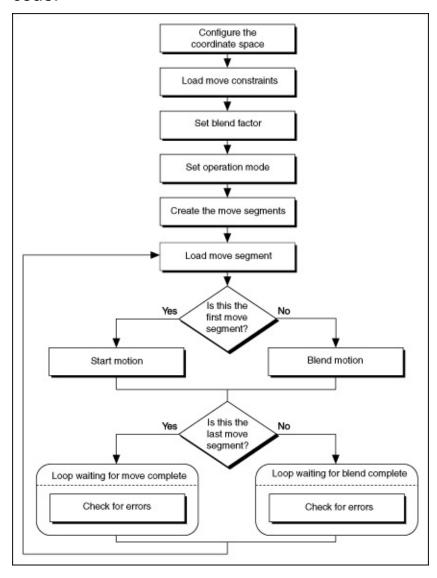
Blending the straight-line move segments enables continuous motion, which decreases the cycle time of the scan. The cycle time is much faster because the motors are not forced to stop after each move segment. The following figure shows the path of the blended move segments.



Refer to **Blending** for information about using blending with NI-Motion.

#### **Raster Scanning Using Blended Straight Lines Algorithm**

The following figure shows the basic algorithm for raster scanning using blended straight-line moves applicable to both C/C++ and LabVIEW code.

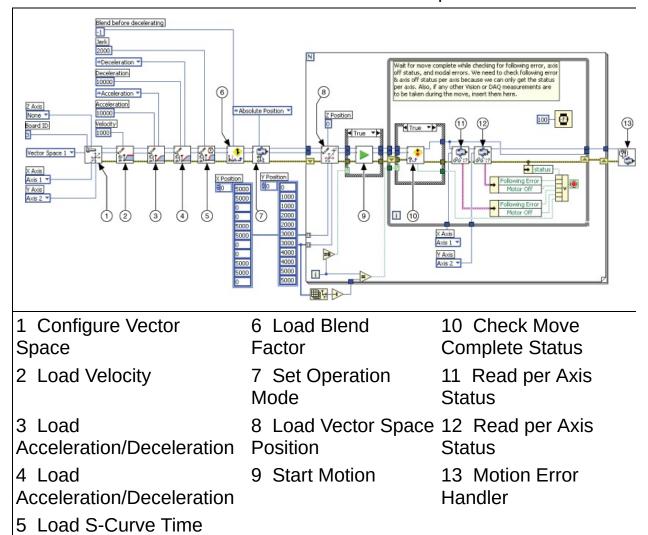


## Raster Scanning Using Blended Straight Lines LabVIEW Diagram

The following block diagram demonstrates using NI-Motion to perform raster scanning using blended straight-line moves.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



## Raster Scanning Using Blended Straight Lines C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
  u8 boardID;  // Board identification number
u8 vectorSpace;  // Vector space number
u16 csr = 0;  // Communication status register
u16 axisStatus;  // Axis status
   u16 status;
   u16 complete;
                   // Move or blend complete status
   u32 i;
   i32 xPosition[11] = {5000, 5000, 0, 0, 5000, 5000, 0, 0, 5000, 50
   i32 yPosition[11] = {0, 1000, 1000, 2000, 2000, 3000, 3000, 4000,
   // Variables for modal error handling
   u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
   // Set the board ID
   boardID = 1;
   // Set the axis number
   vectorSpace = NIMC VECTOR SPACE1;
   // Configure a 2D Vector Space comprising of axes 1, and 2
   err = flex_config_vect_spc(boardID, vectorSpace, 1, 2, 3);
   CheckError;
   // Set the velocity for the move (in counts/sec)
   err = flex_load_velocity(boardID, vectorSpace, 10000, 0xFF);
   CheckError;
   // Set the acceleration for the move (in counts/sec^2)
   err = flex_load_acceleration(boardID, vectorSpace, NIMC_ACCELERAT
   CheckError;
   // Set the deceleration for the move (in counts/sec^2)
   err = flex load acceleration(boardID, vectorSpace, NIMC DECELERAT
```

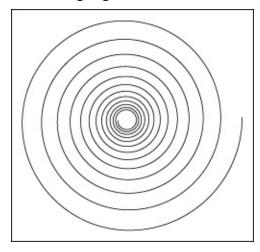
```
CheckError;
// Set the jerk or s-curve in sample periods
err = flex load scurve time(boardID, vectorSpace, 100, 0xFF);
CheckError;
// Set the operation mode to absolute position
err = flex_set_op_mode(boardID, vectorSpace, NIMC_ABSOLUTE_POSITI
CheckError;
// Load the straight line segments one by one
for (i=0; i<11; i++){</pre>
   // Load Target Position
   err = flex_load_vs_pos(boardID, vectorSpace, xPosition[i], yPo
   CheckError;
   if(i==0){
      // Start the move
      err = flex_start(boardID, vectorSpace, 0);
      CheckError;
   }else{
      // Blend the move
      err = flex_blend(boardID, vectorSpace, 0);
      CheckError;
   }
   do
   {
      axisStatus = 0;
      if(i==10){
         // Check the move complete status
         err = flex check move complete status(boardID, vectorSpa
         CheckError;
      }else{
         // Check the blend complete status
         err = flex check blend complete status(boardID, vectorSp
         CheckError;
      }
      // Check the following error/axis off status for axis 1
      err = flex_read_axis_status_rtn(boardID, 1, &status);
      CheckError;
      axisStatus |= status;
      // Check the following error/axis off status for axis 2
```

```
err = flex_read_axis_status_rtn(boardID, 2, &status);
     CheckError;
     axisStatus |= status;
     // Read the communication status register and check the moa
     err = flex_read_csr_rtn(boardID, &csr);
     CheckError;
     // Check the modal errors
     if (csr & NIMC_MODAL_ERROR_MSG)
        err = csr & NIMC_MODAL_ERROR_MSG;
        CheckError;
     Sleep(50); // Check every 50 ms
  }while (!complete && !(axisStatus & NIMC_FOLLOWING_ERROR_BIT)
  if( (axisStatus & NIMC FOLLOWING ERROR BIT) || (axisStatus & N
     break; // Break out of the for loop as an axis was killed
   }
}
return; // Exit the Application
// Error Handling
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG){
  do{
     // Get the command ID, resource and the error code of the m
     // error from the error stack on the board
     flex_read_error_msg_rtn(boardID, &commandID, &resourceID, &err
     nimcDisplayError(errorCode, commandID, resourceID);
     // Read the Communication Status Register
     flex_read_csr_rtn(boardID,&csr);
  }while(csr & NIMC_MODAL_ERROR_MSG);
}
else
          // Display regular error
   nimcDisplayError(err, 0, 0);
return; // Exit the Application
```

}

#### **User-Defined Scanning Path**

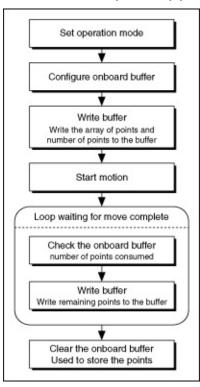
You can create a custom path that covers the maximum scan area in the shortest time using the contoured move feature of the NI motion controller. This way you bypass the trajectory generator and send exact positions to the motion controller. The controller then interpolates the distance between the given points using a cubic spline algorithm. The following figure shows the scanning path used in the examples.



Using the contoured move gives you the greatest amount of flexibility regarding the scan area and speed. However you lose the benefit of the trajectory generator of the NI motion controller. Refer to <a href="Contoured">Contoured</a> <a href="Moves">Moves</a> for information about using contoured moves with NI-Motion.

#### **User-Defined Scanning Path Algorithm**

The following figure shows the basic algorithm for raster scanning using a user-defined path applicable to both C/C++ and LabVIEW code.

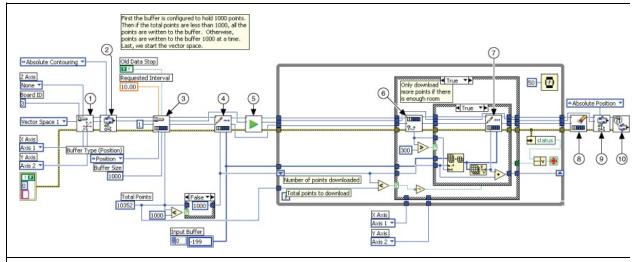


#### **User-Defined Scanning Path LabVIEW Diagram**

The following block diagram demonstrates using NI-Motion to perform scanning using a contoured move.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.



- 1 Configure Vector Space
- 2 Set Operation Mode
- 3 Configure Buffer
- 4 Write Buffer

- 5 Start Motion
- 6 Check Buffer
- 7 Write Buffer
- 8 Clear Buffer
- 9 Set Operation Mode
- 10 Motion Error Handler

#### **User-Defined Scanning Path C/C++ Code**

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
    i32 i;
    \dot{i32} points[1994] = NIMC_SPIRAL_ARRAY; // Array of 2D points t
    132 points[1994] = NIMC_SPIRAL_ARRAY; // Array of 2D points t
u32 numPoints = 1994; // Total number of points to contour
i32 bufferSize = 1000; // The size of the buffer to allocate
f64 actualInterval; // The interval the controller can re
i32* downloadData = NULL; // The temporary array that is create
u32 currentDataPoint = 0; // Indicates the next point in the pc
i32 backlog; // Indicates the available space to a
u16 bufferState; // Indicates the state of the onboard
u32 pointsDone; // Indicates the number of points tha
u32 dataCopied = 0; // Keeps tack of the points copied
    // Variables for modal error handling
    u16 commandID;  // The commandID of the function
u16 resourceID;  // The resource ID
i32 errorCode;  // Error code
    // Set the board ID
    boardID= 1;
    // Set the axis number
    vectorSpace = NIMC_VECTOR_SPACE1;
    // Configure a 2D Vector Space comprising of axes 1 and 2
    err = flex_config_vect_spc(boardID, vectorSpace, 1, 2, 3);
    CheckError;
    // Set the operation mode to absolute position
    err = flex_set_op_mode(boardID, vectorSpace, NIMC_ABSOLUTE_CONTOU
    CheckError;
```

```
// Configure buffer on motion controller memory (RAM)
// Note requested time interval is hardcoded to 10 milliseconds
err = flex_configure_buffer(boardID, 1 /*buffer number*/, vectorS
                                    numPoints, NIMC_TRUE, 10, &act
// Send the first 1000 points of the data
downloadData = malloc(sizeof(i32)*bufferSize);
for(i=0;i<bufferSize;i++){</pre>
   downloadData[i] = points[i];
   currentDataPoint++;
err = flex_write_buffer(boardID, 1/*buffer number*/, bufferSize,
free(downloadData);
downloadData = NULL;
CheckError;
// Start Motion
err = flex_start(boardID, vectorSpace, 0);
CheckError;
for(;;){
   axisStatus = 0;
   // Check for available space and download remaining points eve
   Sleep(50);
   // Check to see if we have more points to download
   if(currentDataPoint < numPoints){</pre>
      err = flex_check_buffer_rtn(boardID, 1/*buffer number*/, &b
      CheckError;
      if(backlog >= 300){
         downloadData = malloc(sizeof(i32)*backlog);
         dataCopied = 0;
         for(i=0;i<backlog;i++){</pre>
            if(currentDataPoint > numPoints) break;
            downloadData[i] = points[currentDataPoint];
            currentDataPoint++;
            dataCopied++;
         }
         err = flex_write_buffer (boardID, 1 /*buffer number*/, d
         free(downloadData);
         downloadData = NULL;
         CheckError;
      }
   }
```

```
// Check the move complete status
  err = flex_check_move_complete_status(boardID, vectorSpace, 0,
  CheckError;
   if(moveComplete) break;
  // Check for axis off status/following error or any modal erro
  // Read the communication status register and check the modal
  err = flex_read_csr_rtn(boardID, &csr);
  CheckError:
  // Check the modal errors
  if (csr & NIMC_MODAL_ERROR_MSG){
     err = csr & NIMC MODAL ERROR MSG;
     CheckError;
  }
  // Check the motor off status on all the axes or axis
  err = flex read axis status rtn(boardID, 1, &status);
  CheckError;
  axisStatus |= status;
  err = flex_read_axis_status_rtn(boardID, 2, &status);
  CheckError;
  axisStatus |= status;
  if( (axisStatus & NIMC_FOLLOWING_ERROR_BIT) || (axisStatus & N
     break;// Break out of the for loop as an axis was killed
  }
}
// Set the mode back to absolute mode to get the controller out o
// mode
err = flex_set_op_mode(boardID, vectorSpace, NIMC_ABSOLUTE_POSITI
CheckError;
// Free the buffer allocated on the controller memory
err = flex clear buffer(boardID, 1/*buffer number*/);
CheckError;
return; // Exit the Application
// Error Handling
//
nimcHandleError; //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
```

#### **Rotating Knife**

The purpose of this application is to cut a web with a rotating knife. The blade must cut precisely between labels on the web. Because the web material can stretch under certain conditions, it is not enough to cut the web at constant length, because the length of each label can vary. To accomplish this task, the web is marked one time per cycle at the required cutting location. The motion controller reads this mark using a sensor and performs the necessary correction.

To simplify this example, assume that the length of the cut is equal to the circumference of the knife. Under ideal conditions, the mark should be read when the blade is at position A, as shown in the figure below. Therefore, the motor should move one revolution without any correction before causing the cut.

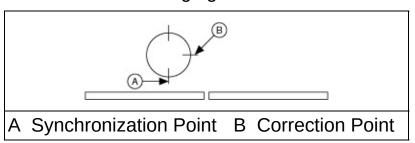


**Tip** Refer to <u>Electronic Gearing and Camming</u> for information about superimposed moves/registration applications.

#### **Solution**

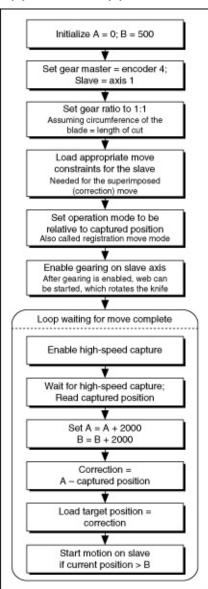
The rotary knife is electronically geared to the web with a gear ratio of 1:1, which ensures that at the time of cut, the speed of the web and the knife is the same. The speed of each must be the same to make a clean cut without stretching the web. Also, under ideal conditions, the web and rotating knife move the exact same distance. For example, the length of the cut might be one revolution, which is equal to 2,000 counts.

The sensor reading the mark is connected to one of the high-speed capture lines on the motion controller. Because the elasticity of the web material results in varying label lengths, the mark can be read before the blade is at position A or after it is at position A. The application must correct the position where the blade of the rotary knife should be when the high-speed capture occurs. This correction must occur after the blade has crossed position A so that the current cut is not damaged. To accomplish this goal, mark the correction point to be at position B, as shown in the following figure.



#### **Rotating Knife Algorithm**

The following figure shows the basic algorithm for the rotating knife application applicable to both C/C++ and LabVIEW code.

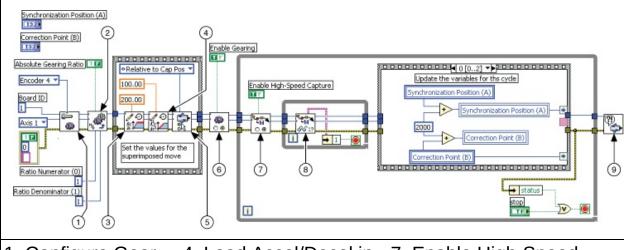


#### **Rotating Knife LabVIEW Diagram**

The following block diagrams demonstrate using NI-Motion to program the rotating knife application.



**Tip** Refer to NI-Motion\Documentation\Examples\NI-Motion User Manual\ for the LabVIEW VIs for these examples.

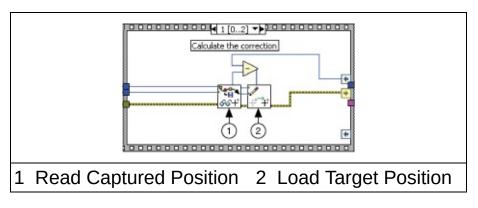


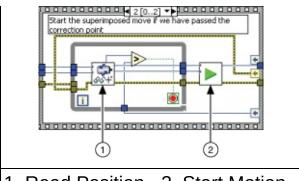
1 Configure Gear 4 Load Accel/Decel in 7 Enable High-Speed Capture

2 Load Gear Ratio 5 Set Operation 8 Read High-Speed Capture Status

3 Load Velocity in 6 Enable Gearing 9 Motion Error Handler Single Axis

The following figures show the remaining cases for the block diagram above.





1 Read Position 2 Start Motion

#### Rotating Knife C/C++ Code

The following example code is not necessarily complete, and may not compile if copied exactly. Refer to the examples folder on the NI-Motion CD for files that are complete and compile as is.

```
// Main function
void main(void)
                                     // Board identification number
  u8 boardID;
  u8 slaveAxis;
                                     // Slave axis number
  u8 master;
                                    // Gear master
                                    // Communication status registe
  u16 csr = 0;
  i32 synchronizationPosition = 0; // Synchronization position
  i32 correctionPoint = 500;  // Point where the correction c
i32 cyclePosition = 2000;  // One revolution is 2000 count
i32 currentPosition;  // The current slave position
i32 capturedPosition;  // The position at which the tr
  u16 axisStatus;
  // Variables for modal error handling
  u16 commandID;
                                    // The commandID of the functio
                                     // The resource ID
  u16 resourceID;
  i32 errorCode;
                                     // Error code
   // Set the board ID
  boardID= 1;
   // Set the axis number
  slaveAxis = 1;
   // Master is encoder 4
  master = NIMC ENCODER4;
   //-----
   // Set up the gearing configuration for the slave axis
   //-----
  // Configure Gear Master
   err = flex_config_gear_master(boardID, slaveAxis, master);
  CheckError;
   // Load Gear Ratio 1:1
   err = flex_load_gear_ratio(boardID, slaveAxis, NIMC_ABSOLUTE_GEA
                             1/* ratioNumerator*/, 1/* ratioDenomin
  CheckError;
```

```
// Set up the move parameters for the superimposed move
// to be done on registration
//-----
// Set the operation mode to relative
err = flex_set_op_mode(boardID, slaveAxis, NIMC_RELATIVE_TO_CAPTU
CheckError;
// Load Velocity in RPM
err = flex_load_rpm(boardID, slaveAxis, 100.00, 0xFF);
CheckError;
// Load Acceleration and Deceleration in RPS/sec
err = flex load rpsps(boardID, slaveAxis, NIMC BOTH, 50.00, 0xFF)
CheckError;
//-----
// Enable Gearing on slave axis
//-----
err = flex_enable_gearing_single_axis (boardID, slaveAxis, NIMC_T
CheckError;
//-----
// Wait for trigger to do the registration move
//-----
for(;;){
  // Enable High speed capture for slave axis
  err = flex_enable_hs_capture(boardID, slaveAxis, NIMC_TRUE);
  CheckError;
  do
  {
    // Check the high speed capture status/following error/axis
     err = flex read axis status rtn(boardID, slaveAxis, &axisSt
    CheckError;
    // Read the communication status register and check the moa
     err = flex_read_csr_rtn(boardID, &csr);
    CheckError;
    // Check the modal errors
    if (csr & NIMC_MODAL_ERROR_MSG)
     {
       err = csr & NIMC_MODAL_ERROR_MSG;
       CheckError;
     }
```

```
}while (!(axisStatus & NIMC_HIGH_SPEED_CAPTURE_BIT) && !(axisS
  if((axisStatus & NIMC_FOLLOWING_ERROR_BIT) || (axisStatus & NI
     break; // Break out of the for loop
  }
  // Update the variables for this cycle
   synchronizationPosition += cyclePosition;
  correctionPoint += cyclePosition;
  // Read the captured position
  err = flex_read_cap_pos_rtn(boardID, slaveAxis, &capturedPosit
  CheckError;
  // Load the target position for the registration (superimposed
  err = flex_load_target_pos(boardID, slaveAxis, (synchronizatio
  CheckError;
  // Wait until we have passed the correction point before we ap
  currentPosition = 0;
  while (currentPosition < correctionPoint){</pre>
      err = flex_read_pos_rtn(boardID, slaveAxis, &currentPositio
     CheckError;
  }
  // Start registration move on the slave
  err = flex start(boardID, slaveAxis, 0);
  CheckError;
}// For loop
return; // Exit the Application
// Error Handling
nimcHandleError: //NIMCCATCHTHIS:
// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG){
  do{
      // Get the command ID, resource and the error code of the m
     // error from the error stack on the board
     flex read error msq rtn(boardID,&commandID,&resourceID,&err
     nimcDisplayError(errorCode, commandID, resourceID);
     // Read the Communication Status Register
     flex read csr rtn(boardID, &csr);
```

```
}while(csr & NIMC_MODAL_ERROR_MSG);
}
else  // Display regular error
  nimcDisplayError(err,0,0);
return;  // Exit the Application
}
```

#### **Tuning Servo Systems**

When your motion control system includes a servo motor, you must tune and calibrate the system to ensure proper performance. This chapter covers general information about tuning and calibrating your servo system using control loop parameters. Refer to the <a href="Tuning Servo Motors">Tuning Servo Motors</a> section of the *Measurement & Automation Explorer Help for Motion* for more information about and instructions for tuning servo motors in Measurement & Automation Explorer (MAX).

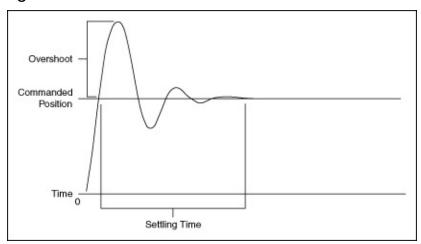


**Note** This section does not apply if you are using the NI SoftMotion Controller because the control loop is implemented on the drive. Refer to the drive documentation for information about tuning the servo motors you are using with the CANopen drive.

#### **Using Control Loops to Tune Servo Motors**

Tuning maximizes the performance of your servo motors. A servo system uses feedback to compensate for errors in position and velocity. For example, when the servo motor reaches the desired position, it cannot stop instantaneously. There is a normal overshoot that must be corrected. The controller turns the motor in the opposite direction for the amount of distance equal to the detected overshoot. However, this corrective move also exhibits a small overshoot, which must also be corrected in the same manner as the first overshoot.

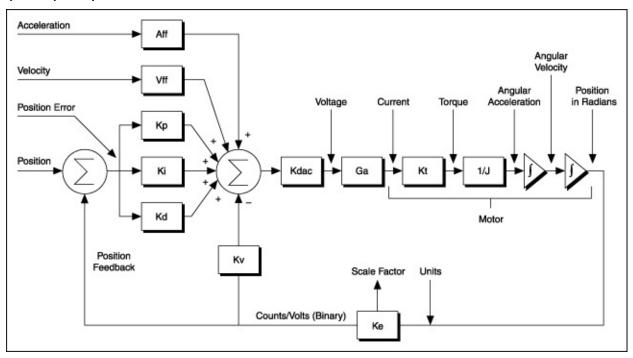
A properly tuned servo system exhibits overshoot as shown the following figure.



The amount of time required for the motors to settle on the commanded position is called the *settling time*. By tuning the servo motors, you can affect the settling time, the amount of overshoot, and various other performance characteristics.

# **Control Loop**

NI motion servo control uses control loops to continuously correct errors in position and velocity. You can configure the control loop to perform a Proportional, Integral and Derivative (PID) loop or a more advanced control loop, such as the velocity feedback (PIV) or velocity feedforward (PIVff) loops.



# **PID Loop Descriptions**

The following are common variables relating to the PID control loop.

#### **Kp (Proportional Gain)**

The proportional gain (Kp) determines the contribution of restoring force that is directly proportional to the position error. This restoring force functions in much the same way as a spring in a mechanical system.

Each sample period, the PID loop calculates the position error, which is the difference between the instantaneous trajectory position and the primary feedback position, and multiplies the position error by Kp to produce the proportional component of the 16-bit DAC command output.

An axis with too small a value of Kp is unable to hold the axis in position and is very soft. Increasing Kp stiffens the axis and improves its disturbance torque rejection. However, too large a value of Kp often results in instability.

#### Ki (Integral Gain)

The integral gain (Ki) determines the contribution of restoring force that increases with time, ensuring that the static position error in the servo loop is forced to zero. This restoring force works against constant torque loads to help achieve zero position error when the axis is stopped.

Each sample period, the position error is added to the accumulation of previous position errors to form an integration sum. This integration sum is scaled by dividing by 256 prior to being multiplied by Ki.

In applications with small static torque loads, this value can be left at its default value of zero (0). For systems having high static torque loads, this value should be tuned to minimize position error when the axis is stopped.

Although non-zero values of Ki cause reduced static position error, they tend to cause increased position error during acceleration and deceleration. This effect can be mitigated through the use of the Integration Limit parameter. Too high a value of Ki often results in servo loop instability. National Instruments therefore recommends that you leave Ki at its default value of zero until the servo system operation is stable. Then you can add a small amount of Ki to minimize static position errors.

#### Kd (Derivative Gain)

The derivative gain (Kd) determines the contribution of restoring force proportional to the rate of change (derivative) of position error. This force acts much like viscous damping in a damped spring and mass mechanical system. A shock absorber is an example of this effect.

The PID loop computes the derivative of position error every derivative sample period. A non-zero value of Kd is required for all systems that use torque block amplifiers, where the command output is proportional to motor torque, for the servo loop operation to be stable. Too small a Kd value results in servo loop instability.

With velocity block amplifiers, where the command output is proportional to motor velocity, it is typical to set Kd to zero or a very small positive value.

#### **Kv (Velocity Feedback)**

You can use a primary or secondary feedback encoder for velocity feedback. Setting the velocity feedback gain (Kv) to a value other than zero (0) enables velocity feedback using the secondary encoder, if configured, or the primary encoder if a secondary encoder is not configured.

Kv is used to scale this velocity feedback before it is added to the other components in the 16-bit DAC command output. Kv is similar to derivative gain (Kd) except that it scales the velocity estimated from encoder resources only. The derivative gain scales the derivative of the position error, which is the difference between the instantaneous trajectory position and the primary feedback position. Like the Kd term, the velocity feedback derivative is calculated every derivative sample period and the contribution is updated every PID sample period.

Velocity feedback is estimated through a combination of speeddependent algorithms. Velocity is measured based on the time elapsed between each encoder count.

#### **Vff (Velocity Feedforward)**

The velocity feedforward gain (Vff) determines the contribution in the 16-bit DAC command output that is directly proportional to the instantaneous trajectory velocity. This value is used to minimize following error during the constant velocity portion of a move and can be changed at any time to tune the PID loop.

Velocity feedforward is an open-loop compensation technique and cannot affect the stability of the system. However, if you use too large a value for Vff, following error can reverse during the constant velocity portion, thus degrading performance, rather than improving it.

Velocity feedforward is typically used when operating in PIVff mode with either a velocity block amplifier or substantial amount of velocity feedback (Kv). In these cases, the uncompensated following error is directly proportional to the desired velocity. You can reduce this following error by applying velocity feedforward. Increasing the integral gain (Ki) also reduces the following error during constant velocity but only at the expense of increased following error during acceleration and deceleration and reduced system stability. For these reasons, increasing Ki is not a recommended solution.



**Tip** In PIVff mode, the Kd and Kv gains are set to zero.

Velocity feedforward is rarely used when operating in PID mode with torque block amplifiers. In this case, because the following error is proportional to the torque required, rather than the velocity, it is typically much smaller and does not require velocity feedforward.

#### **Aff (Acceleration Feedforward)**

The acceleration feedforward gain (Aff) determines the contribution in the 16-bit DAC command output that is directly proportional to the instantaneous trajectory acceleration. Aff is used to minimize following error (position error) during acceleration and deceleration and can be changed at any time to tune the PID loop.

Acceleration feedforward is an open-loop compensation technique and cannot affect the stability of the system. However, if you use too large a value of Aff, following error can reverse during acceleration and deceleration, thus degrading performance, rather than improving it.

#### Kdac

Kdac is the Digital to Analog Converter (DAC) gain. Use the following equation to calculate Kdac:

$$Kdac = \frac{20 \text{ V}}{2^{16}}$$

20 V represents the ±10 V range in the motion controller.

Ga

Ga is the Amplifier Gain.

#### Kt

Kt is the Torque Constant of the motor. Kt is represented in Newton Meters per Amp.

#### **1/J**

1/J represents the motor plus load inertia of the motion system.

#### Ke

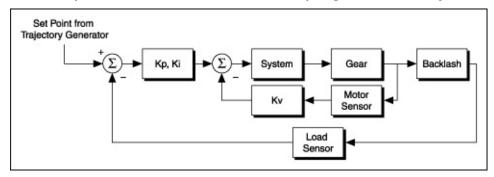
Ke represents the conversion factor to revolutions. This may involve a scaling factor.

# **Dual Loop Feedback**

Motion control systems often use gears to increase output torque, increase resolution, or convert rotary motion to linear motion. The main disadvantage of using gears is the backlash created between the motor and the load. This backlash can cause a loss of position accuracy and system instability.

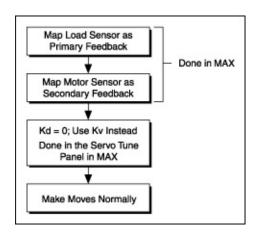
The control loop on the motion system corrects for errors and maintains tight control over the trajectory. The control loop consists of three main parts—proportional, integral and derivative—known as PID parameters. The derivative part estimates motor velocity by differentiating the following error (position error) signal. This velocity signal adds, to the loop, damping and stability. If backlash is present between the motor and the position sensor, the positions of the motor and the sensor are no longer the same. This difference causes the derived velocity to become ineffective for loop damping purposes, which creates inaccuracy in position and system instability.

Using two position sensors for an axis can help solve the problems caused by backlash. As shown in the following figure, one position sensor resides on the load and the other on the motor before the gears. The motor sensor is used to generate the required damping and the load sensor for position feedback. The mix of these two signals provides the correct position feedback with damping and stability.





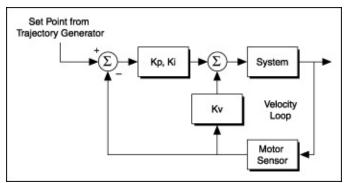
**Tip** You can enable dual-loop feedback on the NI motion controller by mapping an encoder as the secondary feedback for the axis, and then using the velocity feedback gain instead of the derivative gain to dampen and stabilize the system, as shown in the following figure.



# **Velocity Feedback**

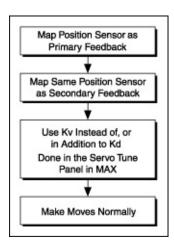
You can configure the NI motion controller for velocity feedback using the Kv (velocity feedback) gain. Using Kv creates a minor velocity feedback loop. This is very similar to the traditional analog servo control method of using a tachometer for closing the velocity loop. This type of feedback is necessary for systems where precise speed control is essential.

You can use a less expensive standard torque, or current mode, amplifier with the velocity feedback loop on NI motion controllers to achieve the same results you would get from using velocity amplifiers, as shown in the following figure.



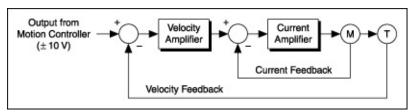
Setting any non-zero value for Kv allows you to use the Kv term instead of or in addition to the Kd term to stabilize the system.

Velocity feedback gain (Kv) is similar to derivative gain (Kd) except that it scales the velocity estimated from encoder resources only. The derivative gain scales the derivative of the position error, which is the difference between the instantaneous trajectory position and the primary feedback position. Like the Kd term, the velocity feedback derivative is calculated every derivative sample period, and the contribution is updated every PID sample period, as shown in the following figure.



# **NI Motion Controllers with Velocity Amplifiers**

Velocity amplifiers close the velocity loop using a tachometer on the amplifier itself, as shown in the following figure. In this case, the controller must ensure that the voltage output is proportional to the velocity. Use the velocity feedforward term (Vff) to ensure that there is minimum following error during the constant velocity profiles.



The following figure describes how to use NI motion controllers with velocity amplifiers.

You typically use velocity feedforward when using controllers with velocity amplifiers. The uncompensated following error is directly proportional to the specified velocity. You can reduce the following error by applying velocity feedforward. Increasing the integral gain (Ki) also reduces the following error during constant velocity, but at the expense of increased following error during acceleration and deceleration and reduced system stability.

M

**Note** National Instruments does not recommend increasing Ki.

Velocity feedforward is rarely used when operating in PID mode with torque block amplifiers. In this case, following error is typically much smaller because it is proportional to the torque required rather than to the velocity. When operating in PID mode with torque block amplifiers, velocity feedforward is not required.

# **Sinusoidal Commutation for Brushless Servo Motion Control**

Sinusoidal commutation allows you to use less expensive servo motor drives with NI motion controllers that support this feature.

- Phase Initialization
- Determining the Counts per Electrical Cycle of the Motor
- Commutation Frequency
- Troubleshooting Hall Effect Sensor Connections

### **Phase Initialization**

When the system is first powered on, the controller must determine the initial commutation phase. NI motion controllers support several methods of phase initialization, including Hall effect sensors, shake and wake, and direct set.

#### **Hall Effect Sensors**

The controller can use Hall effect sensors to estimate the commutation phase based on the state of the sensors. After a Hall effect state transition occurs, the controller recalculates the phase angle based on the transition location. To obtain maximum torque at the beginning of the move, perform a move that is 1/6th of the magnetic cycle after system initialization. Refer to the hardware documentation for Hall effect sensor types and connection schemes.

#### **Shake and Wake**

"Shake and wake" is an initialization method where the motion controller outputs a specified voltage for a specified duration. This drives the system to the zero-degree phase position and allows you to establish the position as a baseline for all other phase positions.

During this process, the motor moves to the zero-degree position with high torque. Ensure the system is away from any limits before performing shake and wake initialization.

If the system has load or is moving against gravity, increase the shake and wake voltage. If there is significant jitter as the axis approaches zero, increase the duration.

#### **Direct Set**

Direct set is an initialization method where the controller sets the current position as the specified phase angle. This initialization method is recommended only for a custom system with known initial phase angle.

Whenever the axis is enabled, the controller must perform the phase initialization procedure to determine the phase.

# **Determining the Counts per Electrical Cycle of** the Motor

The controller needs to know the counts per electrical cycle of the motor to determine the commutation phase. The motor manufacturer usually gives this specification. In many cases, the information also may be specified as the number of poles.

To convert from the number of poles to the number of counts per electrical cycle, use the following formula:

counts per electrical cycle =  $\frac{\text{counts per revolution} \times 2}{\text{number of poles}}$ 



**Caution** Counts per electrical cycle must be set correctly to avoid overheating and damaging your motor.

# **Commutation Frequency**

The controller updates the command voltage and the commutation phase every update period. To commutate brushless motors smoothly, the controller must update the phase at least six times per electrical cycle. Therefore, the commutation frequency is limited by the update rate of the control loop. To calculate the maximum commutation frequency supported at a particular PID update rate, use the following formula:

 $commutation \ frequency = \frac{\frac{counts \ per \ electrical \ cycle}{PID \ rate \ x \ 6}$ 

# **Troubleshooting Hall Effect Sensor Connections**

Complete the following steps if you have problems with Hall effect sensor connections.

- 1. Check the manuals that shipped with the hardware for connection procedures.
- 2. Perform a "shake and wake" phase initialization. During this process, the motor is driven to the zero degree phase position with the commanded voltage. Make sure the motor is clear of any limits before you start.
- 3. Record the Hall effect sensors states by reading the DIO lines connected to the Hall sensors. Refer to the hardware documentation for the Hall effect sensor lines. This is the state of the Hall effect sensors at the zero-degree phase position.
- 4. Command the motor to move forward at a slow velocity. Record the state of the Hall effect sensors at each state transition. The state of the Hall effect sensors should return to the state recorded in step 2 after six state transitions.
- Use the Hall sensors transition state as the Hall sensors diagram. Refer to the hardware documentation for more information on Hall sensor diagrams. Follow the procedure outlined in the hardware documentation.

# Switching Feedback on the Fly (NI 7350 Only)

NI 7350 devices allow switching between feedback devices without disabling the axis, such as when switching between encoder and analog feedback. Complete the following steps to configure your system for switching feedback devices on the fly:

- 1. Tune the system with the first feedback device in MAX.
- 2. Tune the system with the second feedback device in MAX.
- 3. Store the first set of PID gains using the Load All PID Parameters VI or function.
- Store the second set of PID gains using the Load All PID Parameters VI or function, following the instructions in the Loading a Second Set of PID Parameters section of the topic.
- 5. Enable the controller to use the second set of PID parameters when switching feedback on the fly using the Load Advanced Control Parameter VI or function.
- 6. Use the Configure Axis Resources VI or <u>function</u> to configure the axis to use the first feedback device.

To switch between the first and second feedback devices, call Configure Axis Resources with the other feedback device when the axis is not moving.

# **Initializing the Controller Programmatically**

You can initialize the motion controller from within a LabVIEW, Visual Basic, or C/C++ program, in addition to <u>initializing controllers in Measurement & Automation Explorer (MAX)</u>.

Refer to the following table for the steps you must take to initialize a controller programmatically and the functions and VIs you use for each step.

Step		Function and/or VI	
1.	Clear the power-up state.	Use the Clear Power Up Status VI or function.	
2.	Review any errors that occurred on the controller to determine how best to handle them.	Use the Read Error Message VI or function.	
3.	Make sure all axes are stopped and disabled.	Use the Stop Motion VI or function and Enable Axes VIs or functions. In the Stop Motion VI or function, set Stop Type (Decel) set to Halt. In the Enable Axes VI or function, set Axis Bitmap to False for each axis you want to disable.	
4.	Unconfigure vector spaces.	Use the Configure Vector Space VI or function with the X Axis, Y Axis, and Z Axis inputs set to None.	
5.	Configure resources for axes.	Use the Configure Axis Resources VI or function.	
6.	Load all axis configuration options you want to use.	Use the Axis & Resource Configuration VIs or functions.	

[	
7. Initialize encoders and ADCs, as appropriate.	Use the appropriate VIs or functions on the Analog & Digital I/O palette.
8. Enable the axes, but leave them deactivated.	Use the Enable Axes VI or function.
9. Load the appropriate control loop parameters.	Use the Load Advanced Control Parameter VI or function and either Load Single PID Parameter VI or function or Load All PID Parameters VI or function.
10. Call halt on all axes to activate them.	Use the Stop Motion VI or function with Stop Type (Decel) set to Halt stop.
11. Configure capture and compare settings.	Use the VIs or functions on the Motion I/O palette to configure the capture and compare settings.
<ul> <li>12. Configure the following optional settings:</li> <li>Configure trajectory settings</li> <li>Configure find reference settings</li> <li>Configure DIO settings</li> <li>Configure</li> </ul>	Use the following palettes or VIs/functions to configure the optional settings:  • Trajectory Control VIs or functions  • Find Reference VIs or functions  • Analog & Digital I/O VIs or functions  • Configure PWM Output VI or function and Load PWM Duty Cycle VI or function  • Gearing VIs or functions

# PWM settings

Gearing

# **Using the Motion Controller with the LabVIEW Real-Time Module**

Using NI-Motion on a real-time (RT) system is designed to be almost transparent for anyone familiar with NI-Motion. Using NI-Motion with RT requires the following hardware and software:

- NI PXI chassis with an available PXI slot
- NI PXI Motion controller
- Host computer
- LabVIEW Real-Time Module
- One of the following motion software options:
  - NI-Motion (73xx controller support)
  - NI SoftMotion Controller

Refer to the Remote Systems and NI-Motion topic in the Measurement & Automation Explorer Help for Motion for detailed instructions about configuring the NI motion controller on a remote PXI chassis.

# Glossary

Prefixes Numbers/Symbols A B C D E F G H I J K L M N O P Q R S T U V W Z

### **Prefixes**

Symbol	Prefix	Value
р	pico	10 -12
n	nano	10 -9
μ	micro	10 -6
m	milli	10 -3
k	kilo	10 <sup>3</sup>
М	mega	10 6
G	giga	10 <sup>9</sup>
Т	tera	10 12

# Numbers/Symbols

nV	nanovolts	10 <sup>-9</sup> volts
μV	microvolts	10 <sup>-6</sup> volts
μΩ	microohms	10 <sup>-6</sup> ohms
mΩ	milliohms	10 <sup>-3</sup> ohms
МΩ	megaohms	10 <sup>6</sup> ohms
рА	picoamps	10 <sup>-12</sup> amperes
nA	nanoamps	10 <sup>-9</sup> amperes
μΑ	microamps	10 <sup>-6</sup> amperes
mA	milliamps	10 <sup>-3</sup> amperes

Α

A amperes

A/D analog-to-digital

absolute mode Treat the target position loaded as position

relative to zero (0) while making a move.

absolute position Position relative to zero.

acceleration/deceleration Measurement of the change in velocity as a

function of time. Acceleration and

deceleration describes the period when velocity is changing from one value to

another, such as from a stop (zero velocity) to a specified speed (target velocity) or vice versa. Deceleration is also considered

negative acceleration.

active closed A signal is active when current is flowing

through the optocoupled input.

active high Signal is active when its value goes high (1).

See also <u>non-inverting</u>.

active low Signal is active when its value goes low (0).

See also <u>inverting</u>.

active open A signal is active when current is not flowing

through the optocoupled input.

ADC analog-to-digital converter

address Character code that identifies a specific

location, or series of locations, in memory or

on a host PC bus system.

amplifier Drive that delivers power to operate the motor

in response to control signals. In general, the

amplifier is designed to operate with a particular motor type. For example, you cannot use a stepper drive to operate a DC

brush motor.

API application programming interface

Unit that controls a motor or any similar motion or control device.

axis

В

b bit—One binary digit, either 0 or 1.

base Memory address that serves as the starting address for address programmable or I/O bus registers. All other addresses are

located by adding to the base address.

binary Number system with a base of 2.

blending Blending connects move segments while maintaining velocity within the given physical constraints. These constraints are the specific move segments being blended, for example, velocity, acceleration, deceleration, and s-curve for each move.

buffer Temporary storage for acquired or generated data (in software).

bus Group of conductors that interconnect individual circuitry in a computer. Typically, a bus is the expansion vehicle to which I/O or other devices are connected. Examples of PC buses are the ISA and PCI bus.

byte Eight related bits of data, an 8-bit binary number. Also used to denote the amount of memory required to store one byte of data.

C

CCW counterclockwise—Implies the direction the motor rotates

in.

closed-loop Broadly applied term relating to any system where output

is measured and compared to input. The output is then adjusted to reach the appropriate condition. In motion control this term applies to a motion system that uses a feedback device to provide position and velocity data for status reporting and accurately controlling position and

velocity.

commutation Sequential control of switched waveforms from the power

driver amplifier into the motor phase windings that causes rotation or linear motion depending on motor type. Brush type motors auto-commutate due to the brush contact with the motor windings. Brushless type motors require the advance information of position and direction to accurately provide correct waveform switching sequences. Brushless motors typically use hall-effect type sensors to generate

the commutation control waveforms.

control Measure of a closed-loop system's response and is system typically represented as a frequency range or an up

typically represented as a frequency range or an update period for the PID loop in a digital servo controller. For example, if a PID loop has an update rate of 250 µs, it

would have a bandwidth of 4 kHz.

CPU central processing unit

bandwidth

CSR communications status register

CW clockwise—Implies the direction the motor rotates in.

#### D

DAC digital-to-analog converter

data The process of collecting and measuring electrical signals acquisition from sensors, transducers, and test probes or fixtures and

inputting them to a computer for processing.

DC direct current

DGND digital ground

digital I/O Group of digital input/output signals.

port

DIP dual inline package

DLL Dynamic link library for Windows. Provides the API for the

motion control devices.

drive Electronic signal amplifier that converts motor control

command signals into higher-voltage signals suitable for

driving motors.

driver Software that controls a specific hardware device such as a

DAQ board or a motion controller.

Ε

encoder Device that translates mechanical motion into electrical

signals; used for monitoring position or velocity in a closed-

loop system.

encoder Number of encoder lines between consecutive encoder resolution marker or Z-bit indexes. If the encoder does not have an

index output the encoder resolution can be referred to as

lines per revolution.

F

F farad

**FIFO** 

First-in-first-out memory buffer—the first data stored is the first data sent to the acceptor. FIFOs are often used on DAQ devices to temporarily store incoming or outgoing data until that data can be retrieved or output. For example, an analog input FIFO stores the results of A/D conversions until the data can be retrieved into system memory, a process that requires the servicing of interrupts and often the programming of the DMA controller. This process can take several milliseconds in some cases. During this time, data accumulates in the FIFO for future retrieval. With a larger FIFO, longer latencies can be tolerated. In the case of analog output, a FIFO permits faster update rates, because the waveform data can be stored on the FIFO ahead of time. This again reduces the effect of latencies associated with getting the data from system memory to the DAQ device.

filter Indicates the control loop parameter gains (PID gains) for a parameters given axis.

Type of signal conditioning that filters unwanted signals from the signal being measured.

flash ROM Type of electrically reprogrammable read-only memory.

following error trip point The difference between the instantaneous commanded trajectory position and the feedback position. If the following error increases beyond the maximum allowable value entered—referred to as the following error trip point—the motor trips on following error and is killed, preventing the axis from running away.

freewheel Condition of a motor when power is de-energized and the motor shaft is free to turn with only frictional forces to impede it.

full-step Full-step mode of a stepper motor—For a two-phase motor, this refers to energizing two windings or phases simultaneously.

# G

Gnd ground GND ground

#### Н

motor

half-step Mode of a stepper motor—For a two phase motor, this

refers to alternately energizing two windings and then only one. In half step mode, alternate steps are strong and weak, but there is significant improvement in low-speed

smoothness over the full-step mode.

hex hexadecimal

holding Force that a motor can provide or withstand while still torque remaining in a fixed stop location without any rotation,

translation or movement.

home Reference position in a motion control system derived switch/home from a mechanical datum or switch. Often designated as

position the zero position in an absolute position frame of

(input) reference. The motion controller halts the motor if it finds

this switch active while performing a find home sequence.

host Computer in which the motion controller is installed, or that is controlling the remote system in which the motion

controller is installed.

hybrid Motor type designed to move in discrete step increments, stepper (typically specified in degrees). Hybrid stepper motors

(typically specified in degrees). Hybrid stepper motors have permanent magnet rotor elements with a coil wound stator (outer shell) and no brushes contacting between the two. The current through the coil phases is switched in a predetermined sequence (commutated) to produce the

appropriate motion in a given direction.

Hz hertz—the number of scans read or updates written per

second.

I

I/O input/output—The transfer of data to and from a computer

system involving communications channels, operator interface

devices, data acquisition, and/or motion control interfaces.

ID identification

import Windows-specific file that contains information about the VIs library

contained in a companion dynamic link library (DLL). Windows

applications are typically linked to one or more import libraries.

in inches

index Marker between consecutive encoder revolutions.

inverting Defines the polarity of a switch (limit switch, home switch, and

so on) when it is in its active state. If these inputs are active low they are said to have inverting polarity. See also active

low.

IRQ interrupt request

industry-standard architecture ISA

jerk Derivative of acceleration (change of acceleration per unit time) measured in units of counts (steps)/s³.

# Κ

- k kilo—The standard metric prefix for 1,000, or 10³, used with units of measure such as volts, hertz, and meters.
- K kilo—The prefix for 1,024, or 2<sup>10</sup>, used with B in quantifying data or computer memory.

L

latching Signal that maintains its value while in a given state, as

opposed to a signal that momentarily pulses when

entering or exiting a state.

LIFO last-in, first-out

limit switch/end- Sensors that alert the control electronics that the

of-travel physical end of travel is near and that the motion must

position (input) stop.

#### M

m meters

MCS Move Complete Status

microstep Proportional control of energy in the coils of a stepper motor

that allows the motor to move to or stop at locations other

than the fixed magnetic/mechanical pole positions

determined by the motor specifications. This capability facilitates the subdivision of full mechanical steps on a stepper motor into finer microstep locations that greatly smooth motor running operation and increase the resolution

or number of discrete positions that a stepper motor can

attain in each revolution.

modulo Treat the position as if it is within the range of total

position quadrature counts per revolution for an axis.

MustOff State or bit that is forced off (False) or must be off to satisfy a

condition.

MustOn State or bit that is forced on (True) or must be on to satisfy a

condition.

### Ν

noise

An undesirable electrical signal. Noise comes from external sources such as the AC power line, motors, generators, transformers, fluorescent lights, soldering irons, CRT displays, computers, electrical storms, welders, radio transmitters, and internal sources such as semiconductors, resistors, and capacitors. Noise corrupts signals you are trying to send or receive.

non-

Defines the polarity of a switch (limit switch, home switch, and inverting so on) when it is in its active state. If these inputs are active high they are said to have non-inverting polarity. See also active high.

### 0

open Method of output capable of sinking current, but not sourcing collector current.

openloop Refers to a motion control system where no external sensors, or feedback devices, are used to provide position or velocity correction signals. P

packets Command and data sent as a group over a computer

bus.

PCI peripheral component interconnect—A high-performance

expansion bus architecture originally developed by Intel to replace ISA and EISA. PCI is achieving widespread acceptance as a standard for PCs and workstations; it offers a theoretical maximum transfer rate of 132 MB/s.

phase angle/phase margin

Value presented in PID Loop Tuning Bode Plot analysis that represents the advance or lead of an input signal to the output signal in a closed-loop servo controller. Used to determine closed-loop system stability at a given

crossover frequency.

PID control loop

proportional-integral-derivative control loop—A control method in which the controller output is proportional to the error, the error time history, and the rate at which the error is changing. The error is the difference between the observed and the commanded values of a variable that is under control action.

PID loop tuning/servo

Flexible adjustment of the Proportional, Integral, and Derivative Gain Parameters along with loop update rate compensation or frequency to assure stable operation and appropriate dynamic response of a closed-loop servo system.

PIVff control loop

proportional-integral-velocity feed forward control loop— A control method that operates with zero derivative gain and either velocity feedback or a velocity block amplifier.

port

- 1. Communications connection on a computer or a remote controller.
- 2. Digital port, consisting of four or eight lines of digital input, and/or output.

position breakpoint Allows a motor to stop at a given point so that another action, such as a data acquisition or an image acquisition, can take place. You can set position breakpoints in absolute or relative quadrature counts.

When the encoder reaches a position breakpoint, the associated breakpoint output immediately transitions.

position resolution

Typically determined by the smallest increment of motion that can be controlled. In stepper motor systems, it is determined by the number of steps per revolution, typically as a limitation of the stepper driver microstepping value or of the feedback device resolution. In servo motor systems, it is entirely determined by the resolution of the feedback device in counts per revolution of the motor.

power cycling Turning the host computer off and then back on. This

resets the motion controller.

profile Instantaneous position versus time output of a trajectory

generator.

pull-in move When stepper motors are run in closed-loop mode, the

encoder feedback is used to verify the position of an axis

when the motion ends. The motion controller then

commands the axis to do a final move so that it is at the

specified target position.

PWM pulse width modulation—Method of controlling the

average current in a motor phase winding by varying the

on-time (duty cycle) of transistor switches.

PXI PCI eXtensions for Instrumentation

# Q

quadrature Encoder line resolution times four. The encoder resolution is the number of encoder lines between consecutive encoder indexes (marker or Z-bit). If the encoder does not have an index output the encoder resolution can be referred to as lines per revolution, lines per inch, lines per millimeter, and so on.

R

RAM random-access memory

RDB return data buffer

relative Sets the position breakpoint for an encoder in relative

breakpoint quadrature counts.

relative Destination or target position for motion specified with position respect to the current location, regardless of its value.

relative Treat the target position loaded as position relative to current

position position while making a move.

mode

ribbon Flat cable in which the conductors are side by side.

cable

ROM read-only memory—non-volatile memory used for storing

code, programs, and data.

rotary axis Axis for which rotary counts are loaded. The axis moves to

the target position by taking the shortest path, either forward or backwards, while remaining within the one revolution

defined by the loaded modulo value.

RPM revolutions per minute—Units for velocity.

RPSPS or revolutions per second squared—Units for acceleration and

RPS/S deceleration.

RTR Ready to Receive

RTSI real-time system integration bus—the National Instruments

timing bus that connects controllers directly, by means of

connectors on top of the controllers, for precise

synchronization of functions.

rate

s, sec seconds

servo 1. Specifies an axis that controls a servo motor.

2. Specifies when a servo motor becomes active.

sinusoidal Method of controlling current in the windings of a

commutation brushless servo motor by using the pattern of a sine wave

to shape the smooth delivery of current to three motor

inputs, each 120° out of phase from the next.

slot Position where a module can be inserted into an ISA or

PCI backplane.

step output The frequency of the step output control pulses generated

by a controller/indexer and provided to an amplifier driver. The combination of step output rate (steps/s), steps per revolution (steps/rev), and time (60 s/minute) provide a basic representation of motor velocity in RPM. Linear speed may be further derived by additional mechanical

data, lead screw revolutions per inch, and so on.

stepper Specifies an axis that controls a stepper motor.

T

toggle Changing state from high to low, back to high, and so on.

torque Force tending to produce rotation.

totem pole Method of output capable of sinking and sourcing current.

trapezoidal Typical motion trajectory, where a motor accelerates up to

profile the programmed velocity using the programmed

acceleration, traverses at the programmed velocity, and then decelerates at the programmed acceleration to the

target position.

trigger Any event that causes or starts some form of data capture.

trigger Digital signal that begins the execution of a sequence of motion commands stored on the controller/indexer allowing the commands to fully execute until the end of the sequence

is reached.

TTL transistor-transistor logic

# U

UOM unit of measure

# ٧

V volts

velocity mode Move the axis continuously at a specified velocity.

### W

watchdog Timer task that shuts down, or resets, the motion control device if any serious error occurs.

word Standard number of bits that a processor or memory manipulates at one time, typically 8-bit, 16-bit, or 32-bit.

# Z

Z-bit Marker or index between consecutive encoder revolutions.

# **Important Information**

**Warranty** 

Copyright

**Trademarks** 

<u>Patents</u>

Warning Regarding Use of NI Products

# Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action

accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

# Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

### **Trademarks**

National Instruments, NI, ni.com, and LabVIEW are trademarks of National Instruments Corporation. Refer to the *Terms of Use* section on ni.com/legal for more information about National Instruments trademarks.

FireWire® is the registered trademark of Apple Computer, Inc.

Handle Graphics®, MATLAB®, Real-Time Workshop®, Simulink®, Stateflow®, and xPC TargetBox® are registered trademarks, and TargetBox $^{\text{TM}}$  and Target Language Compiler $^{\text{TM}}$  are trademarks of The MathWorks, Inc.

Tektronix® and Tek are registered trademarks of Tektronix, Inc.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

# **Patents**

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the patents.txt file on your CD, or <u>ni.com/patents</u>.

# WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

- (1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.
- (2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS. SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE. INJURY. OR DEATH. THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES. INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR

APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

# **Technical Support and Professional Services**

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- <u>Support</u>—Online technical support resources at ni.com/support include the following:
  - Self-Help Resources—For answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable <u>KnowledgeBase</u>, <u>product</u> <u>manuals</u>, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
  - Free Technical Support
     —All registered users receive free
     Basic Service, which includes access to hundreds of
     Applications Engineers worldwide in the NI Discussion
     —Forums at ni.com/forums. National Instruments Applications
     —Engineers make sure every question receives an answer.

     For information about other technical support options in your area, visit ni.com/services or contact your local office at ni.com/contact.
- <u>Training and Certification</u>—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- <u>System Integration</u>—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

# **Branch Offices**

	+
Office	Telephone Number
Australia	1800 300 800
Austria	43 662 457990-0
Belgium	32 (0) 2 757 0020
Brazil	55 11 3262 3599
Canada	800 433 3488
China	86 21 6555 7838
Czech Republic	420 224 235 774
Denmark	45 45 76 26 00
Finland	385 (0) 9 725 72511
France	33 (0) 1 48 14 24 24
Germany	49 89 7413130
India	91 80 41190000
Israel	972 0 3 6393737
Italy	39 02 413091
Japan	81 3 5472 2970
Korea	82 02 3451 3400
Lebanon	961 (0) 1 33 28 28
Malaysia	1800 887710
Mexico	01 800 010 0793
Netherlands	31 (0) 348 433 466
New Zealand	0800 553 322
Norway	47 (0) 66 90 76 60
Poland	48 22 3390150
Portugal	351 210 311 210
Russia	7 495 783 6851
Singapore	1800 226 5886
Slovenia	386 3 425 42 00

South Africa	27 0 11 805 8197
Spain	34 91 640 0085
Sweden	46 (0) 8 587 895 00
Switzerland	41 56 2005151
Taiwan	886 02 2377 2222
Thailand	662 278 6777
Turkey	90 212 279 3031
United Kingdom	44 (0) 1635 523545
United States (Corporate)	512 683 0100