Motion Control

NI-Motion™ User Manual

November 2005
371242B-01

NATIONAL INSTRUMENTS™

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, 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 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, India 91 80 51190000, 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 7 095 783 68 51, Singapore 1800 226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 02 2377 2222, Thailand 662 278 6777, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code feedback.

© 2003–2005 National Instruments Corporation. All rights reserved.

Important Information

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.

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. Other product and company names mentioned herein are trademarks or trade names of their respective companies.

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 `ni.com/patents`.

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.

Contents

PART III
Programming with NI-Motion

Chapter 4
What You Need to Know about Moves

Chapter 5
Straight-Line Moves

Chapter 6
Arc Moves

Chapter 7
Contoured Moves

Chapter 8
Reference Moves

Chapter 9
Blending Moves

Chapter 10
Electronic Gearing and Camming

Chapter 11
Acquiring Time-Sampled Position and Velocity Data

Chapter 12
Synchronization

Chapter 13
Torque Control

Chapter 14
Onboard Programs

Part IV
Creating Applications Using NI-Motion

Chapter 15
Scanning

About This Manual

This manual provides information about the NI-Motion driver software, including background, configuration, and programming information. The purpose of this manual 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 manual is intended for experienced LabVIEW, C/C++, or other developers. Code instructions and examples assume a working knowledge of the given programming language. This manual also assumes a general knowledge of motion control terminology and development requirements.

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

Conventions

The following conventions appear in this manual:

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

italic	Italic text denotes variables, emphasis, a cross reference, 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.
`monospace bold`	Monospace bold text indicates a portion of code with structural significance.
`monospace italic`	Monospace italic text indicates a portion of code that is commented out.

Documentation and Examples

In addition to this manual, NI-Motion includes the following documentation to help you create motion applications:

- *Getting Started with NI-Motion for NI 73xx Motion Controllers*—This document provides installation instructions and general information about the NI-Motion product.
- *Getting Started: NI SoftMotion Controller for Ormec ServoWire SM Drives*—Refer to this document for information about getting started with the NI SoftMotion Controller for Ormec.
- *Getting Started: NI SoftMotion Controller for Copley CANopen Drives*—Refer to this document for information about getting started with the NI SoftMotion Controller for CANopen.
- *NI-Motion VI Help*—Refer to this document for specific information about NI-Motion LabVIEW VIs.
- *NI-Motion Function Help*—Refer to this document for specific information about NI-Motion C/C++ functions.
- *Measurement & Automation Explorer Help for Motion*—Refer to this document for configuration information.
- *NI-Motion ReadMe*—Refer to this HTML document for information about hardware and software installation and information about changes to the NI-Motion driver software in the current version. This document also contains last-minute information about NI-Motion.
- Application notes—For information about advanced NI-Motion concepts and applications, visit `ni.com/appnotes.nsf/`.

- NI Developer Zone (NIDZ)—Visit the NI Developer Zone, at `ni.com/zone`, for example programs, tutorials, technical presentations, the Instrument Driver Network, a measurement glossary, an online magazine, a product advisor, and a community area where you can share ideas, questions, and source code with motion developers around the world.
- Motion Hardware Advisor—Visit the National Instruments Motion Hardware Advisor 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`.

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.

Part I

Introduction

This user manual provides 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.

Part I covers the following topics:

- *Introduction to NI-Motion*
- *Creating NI-Motion Applications*

1

Introduction to NI-Motion

About NI-Motion

NI-Motion is the driver software for National Instruments 73*xx* 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 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.

Software and Hardware Interaction

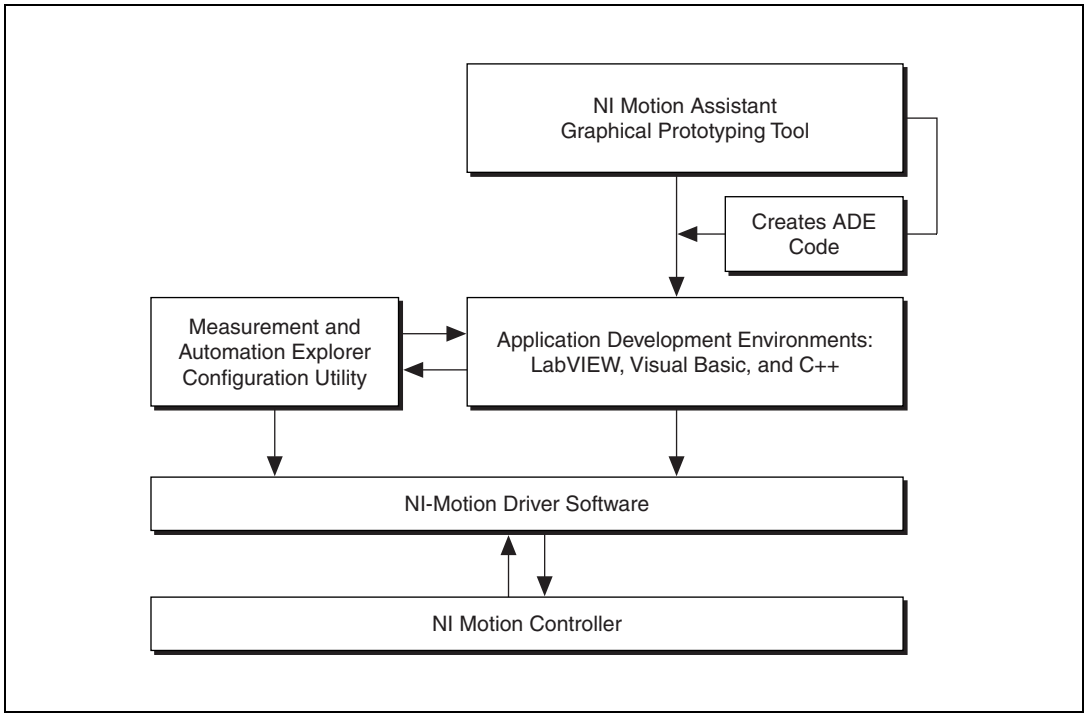


Figure 1-1. NI Motion Control Hardware and Software Interaction

Note The last block in Figure 1-1 is not applicable to the NI SoftMotion Controller.

NI Motion Controller Architecture

This section includes information about the architecture for both the 73*xx* family of NI motion controllers and the NI SoftMotion Controller.

NI 73*xx* Architecture

NI 73*xx* 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 plugs into a variety of slots, including PCI slots, or to a PC using a high-speed serial interface, such as IEEE 1394 (FireWire®).

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 Chapter 12, *Synchronization*, 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.

Figure 1-2 illustrates the physical architecture of the NI motion controller hardware.

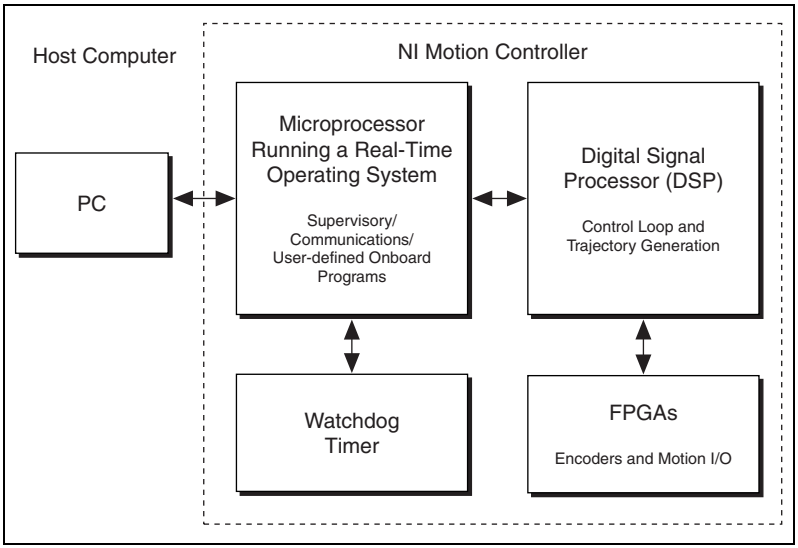


Figure 1-2. Physical NI Motion Controller Architecture

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

NI Motion Controller Functional Architecture

Functionally, the architecture of the NI 73*xx* 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. Refer to Figure 1-3 and Figure 1-4 for an illustration of how the components of the 73*xx* and NI SoftMotion Controller interact.

Figure 1-3 shows the components of the NI 73*xx* motion controllers.

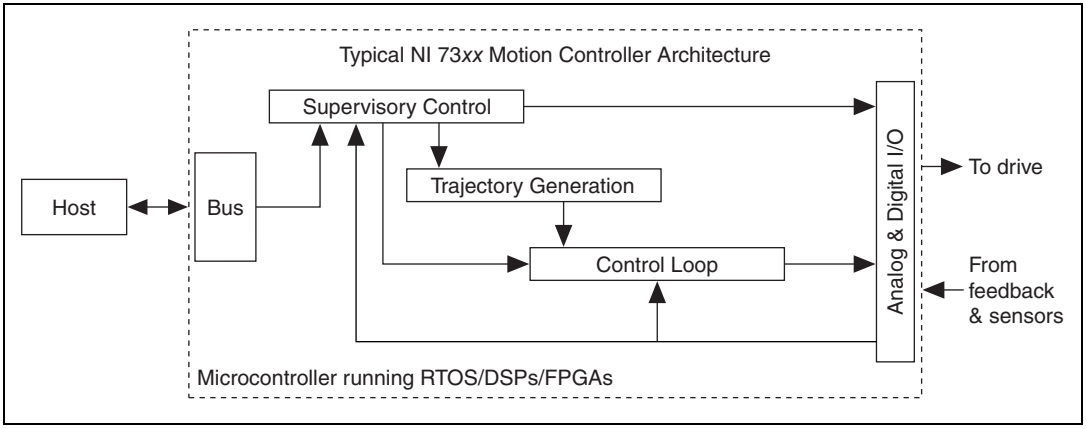


Figure 1-3. Typical NI 73*xx* Motion Controller Functional Architecture

Figure 1-4 shows the components of the NI SoftMotion controller.

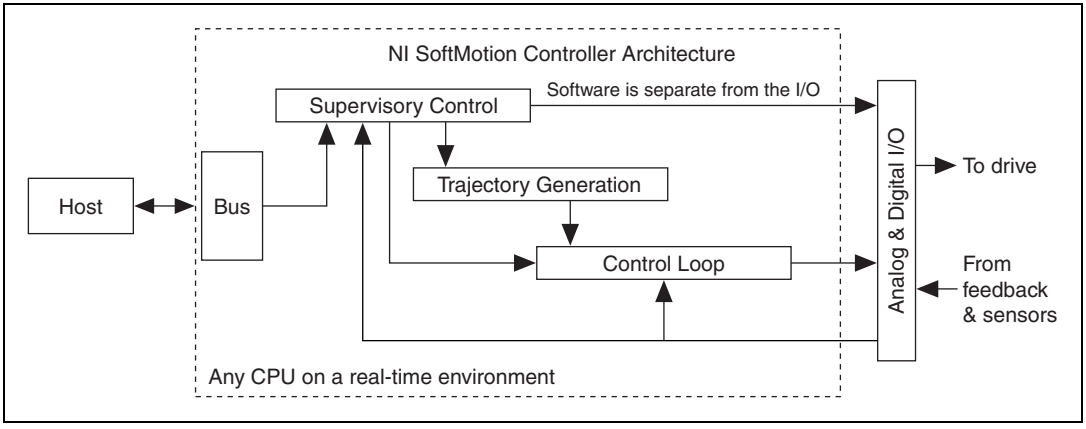


Figure 1-4. NI SoftMotion Controller Functional Architecture

Figure 1-5 illustrates the functional architecture of NI motion controllers.

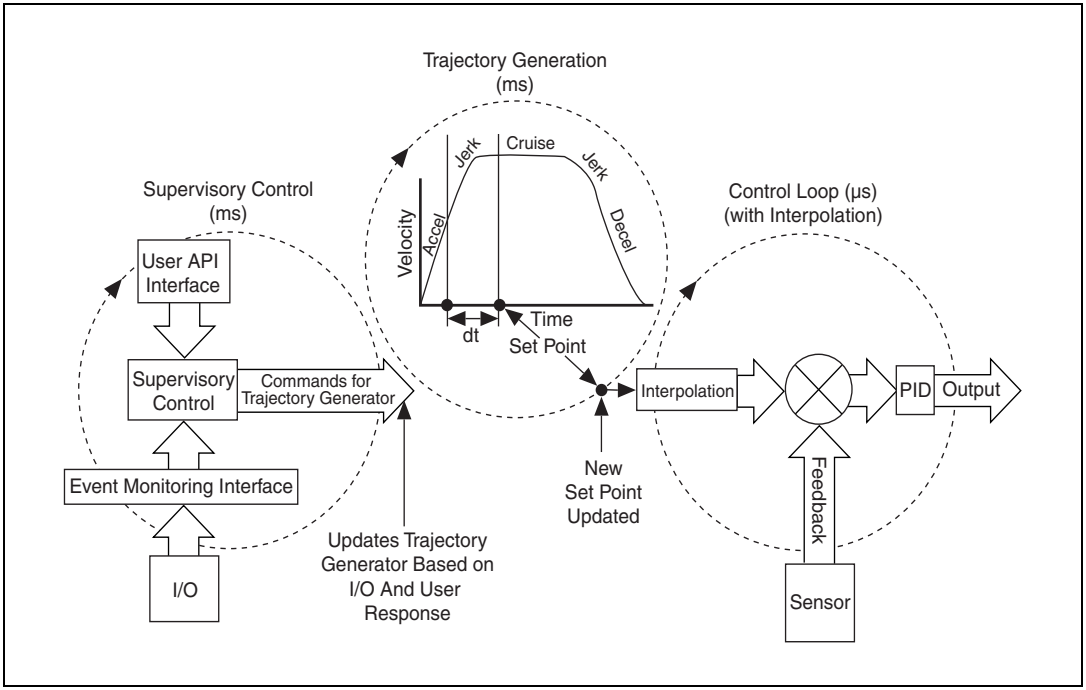


Figure 1-5. NI Motion Controller Functional Architecture

The following list describes how each component of the 73*xx* 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. Refer to Figure 1-5 for an illustration of the spline engine.

- 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 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. Open standards, such as IEEE 1394 and CANopen, are used to connect these components.

NI SoftMotion Controller for Ormec

When you use the NI SoftMotion Controller with an Ormec device, you can daisy chain up to 15 drives together and connect them to the real-time host. The real-time isochronous mode of the IEEE 1394 bus is used to transfer data between the drives and the host. Figure 1-6 shows the NI SoftMotion Controller component architecture that applies when the controller is used with an Ormec device.

The supervisory control and trajectory generation loops execute every millisecond. If the control loop is configured to execute faster than every millisecond, the trajectory data is interpolated before the control loop uses it.

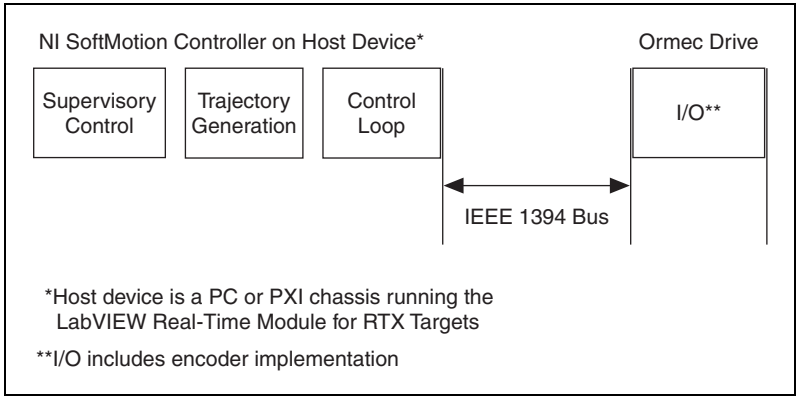


Figure 1-6. NI SoftMotion Controller Functional Architecture for Ormec

NI SoftMotion Controller for CANopen

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.

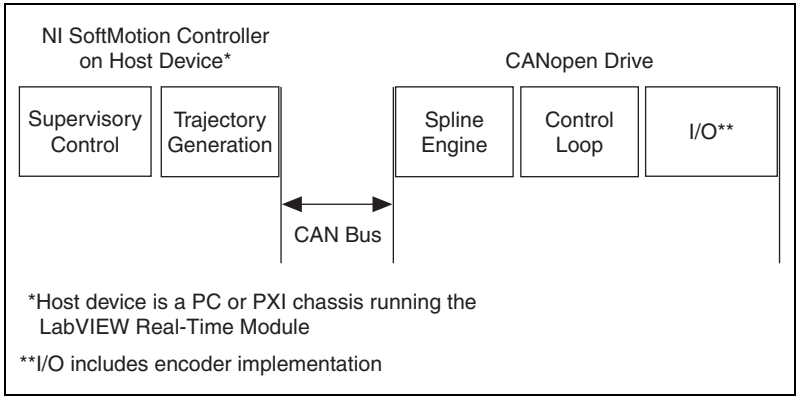


Figure 1-7. NI SoftMotion Controller Functional Architecture for CANopen

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 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.

2

Creating NI-Motion Applications

This chapter describes the basic form of an NI-Motion application and its interaction with other I/O, such as a National Instruments data and/or image acquisition device.

Creating a Generic NI-Motion Application

Figure 2-1 illustrates the steps for creating an application with NI-Motion, and describes the generic steps required to design a motion application.

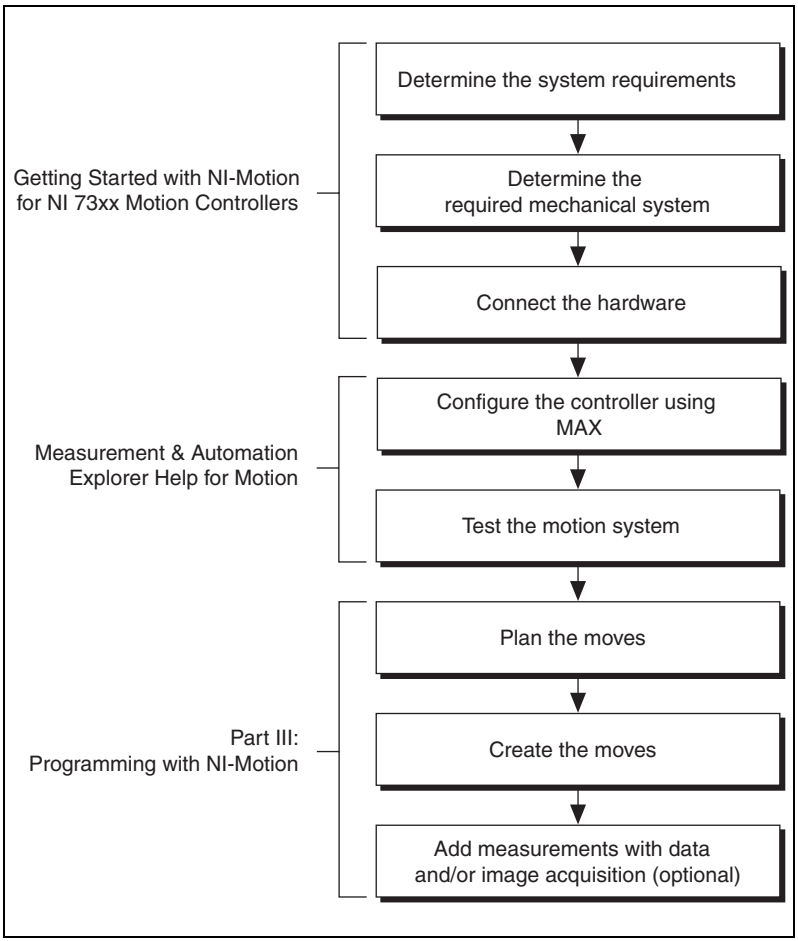
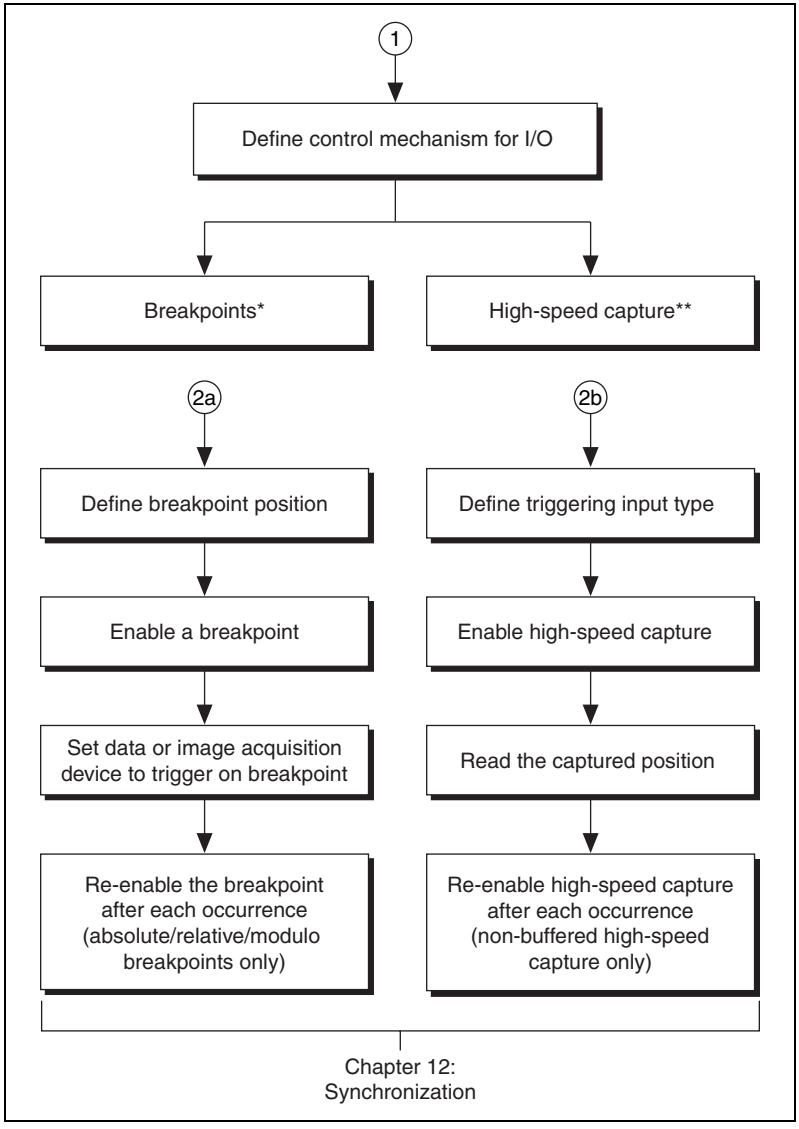


Figure 2-1. Generic Steps for Designing a Motion Application

Adding Measurements to an NI-Motion Application

Figure 2-2 illustrates an expanded view of the topics covered in Part III, *Programming with NI-Motion*, of this manual. For information about items in the diagram, refer to Chapter 12, *Synchronization*.



* 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 when it is used with an Ormec or CANopen device.

** 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 NI SoftMotion Controller for CANopen. You can use two high-speed captures per axis when you are using the NI SoftMotion Controller with an Ormec device.

Figure 2-2. Input/Output with Data and Image Acquisition

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.

Part II

Configuring Motion Control

Motion control is divided into two parts: configuration and execution. Part II of this manual discusses configuring the hardware and software components of a motion control system using NI-Motion.

Part II covers the following topic:

- *Tuning Servo Systems*

3

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 *Measurement & Automation Explorer Help for Motion* for more information about and instructions for tuning servo motors in Measurement & Automation Explorer (MAX).

NI SoftMotion Controller Considerations

This section includes information you need if you are using the NI SoftMotion Controller.

NI SoftMotion Controller for CANopen

This chapter does not apply if you are using the NI SoftMotion Controller for CANopen 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.

NI SoftMotion Controller for Ormec

If you are using the NI SoftMotion Controller for Ormec with an Ormec ServoWire drive in position mode, you must tune the control loop using the drive configuration utility provided by Ormec.

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 in Figure 3-1.

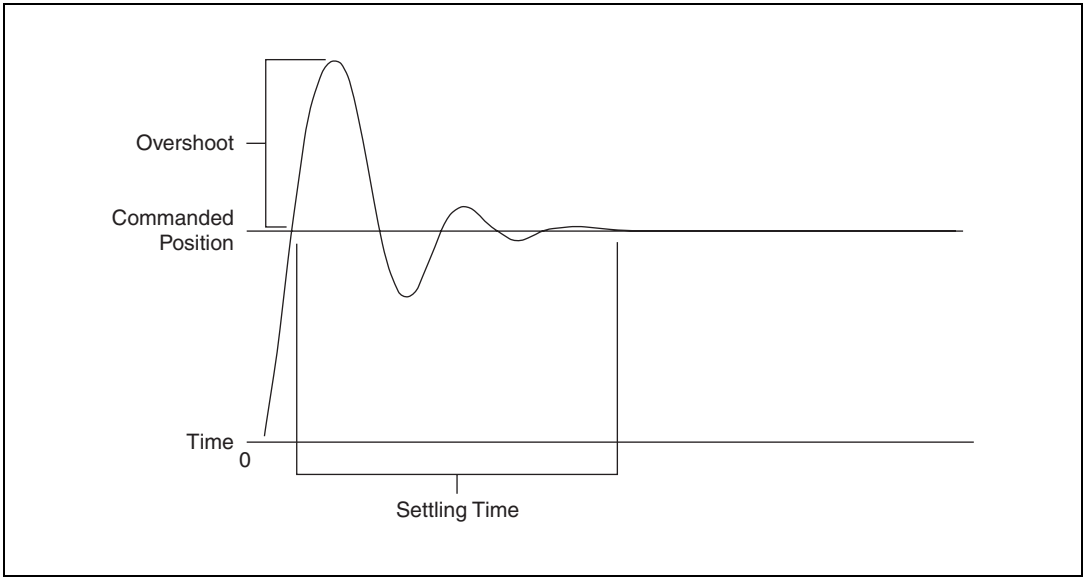


Figure 3-1. Properly Tuned Servo Motor Behavior

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.

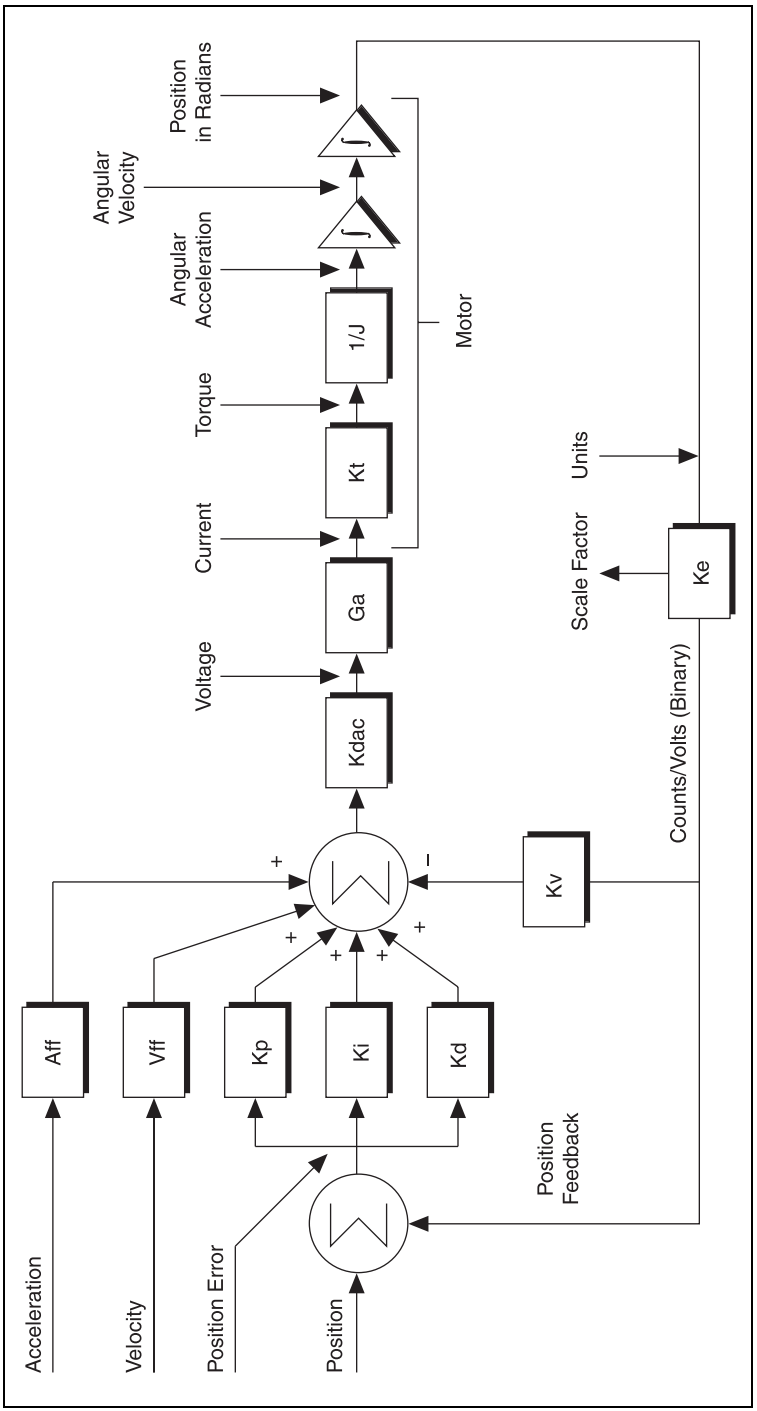


Figure 3-2. NI-Motion Servo PID Loop

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 speed-dependent 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:

$$\text{Kdac} = \frac{20\ \text{V}}{2^{16}}$$

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 Figure 3-3, 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.

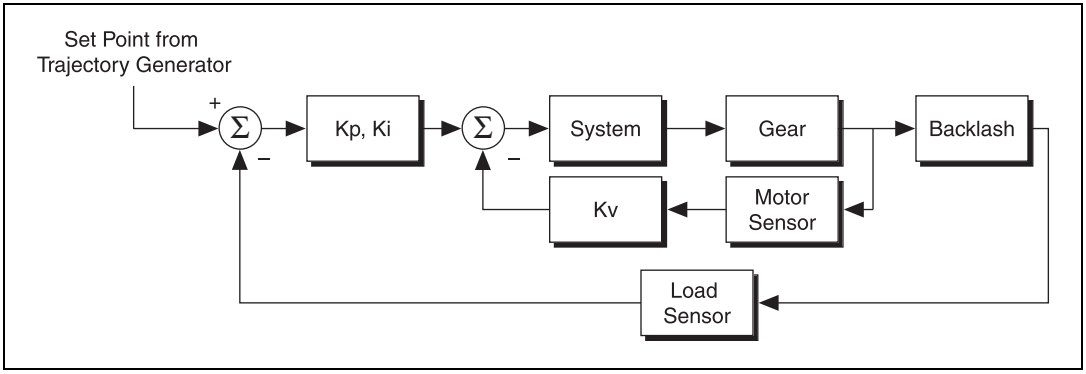


Figure 3-3. Dual Loop Feedback

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 Figure 3-4.

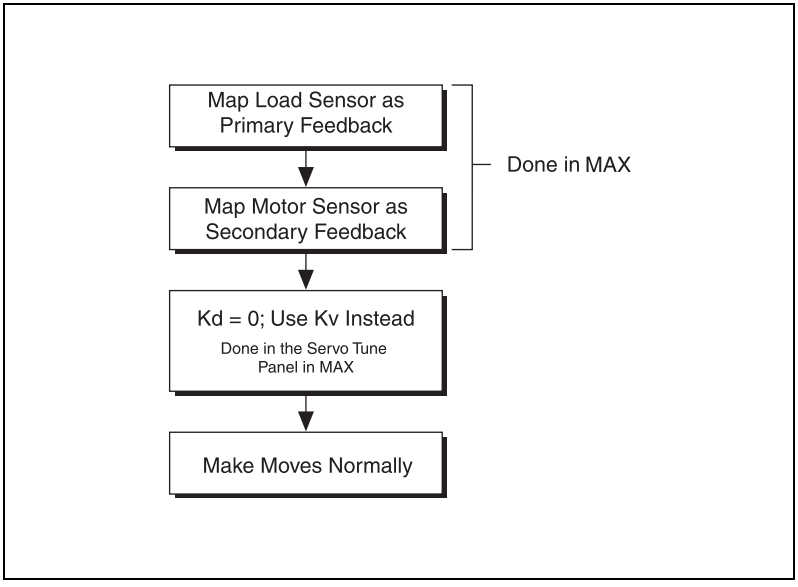


Figure 3-4. Dual Loop Feedback Algorithm

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 Figure 3-5.

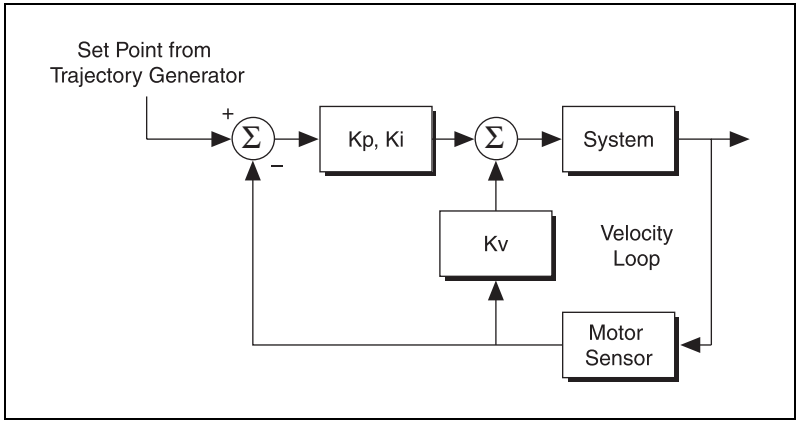


Figure 3-5. Velocity Feedback

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 Figure 3-6.

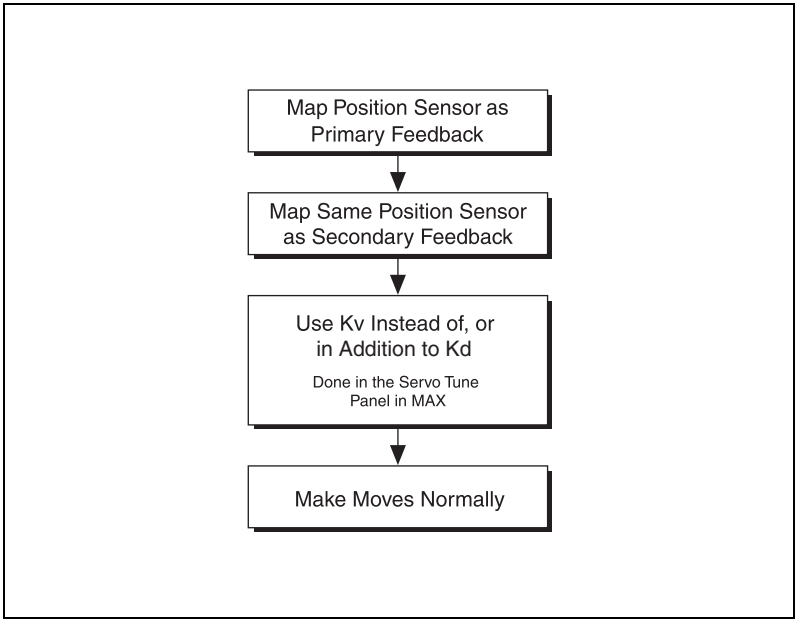


Figure 3-6. Alternate Dual-Loop Feedback Algorithm

NI Motion Controllers with Velocity Amplifiers

Velocity amplifiers close the velocity loop using a tachometer on the amplifier itself, as shown in Figure 3-7. 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.

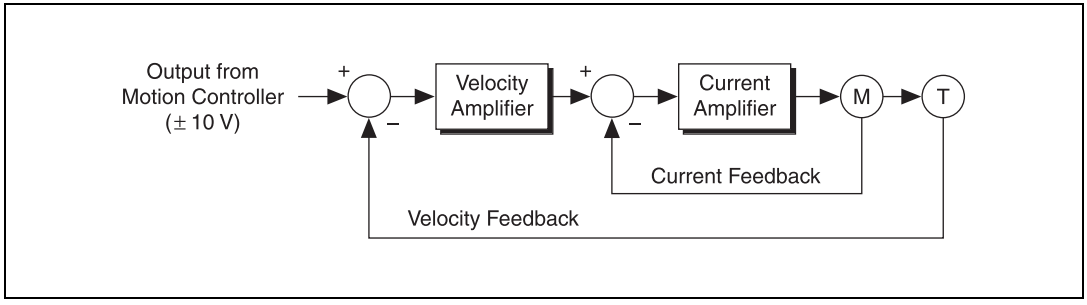


Figure 3-7. NI Motion Controllers with Velocity Amplifiers

Figure 3-8 describes how to use NI motion controllers with velocity amplifiers.

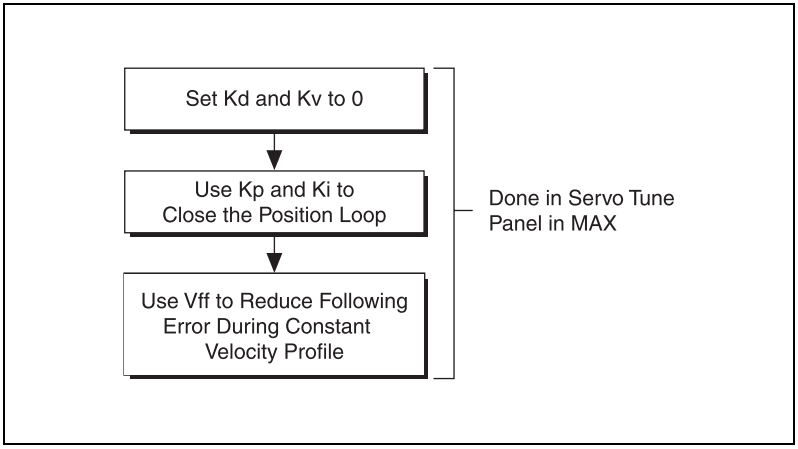


Figure 3-8. NI Motion Controllers with Velocity Amplifiers Algorithm

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.

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.

Part III

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. Part III of this manual 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.

Part III covers the following topics:

- *What You Need to Know about Moves*
- *Straight-Line Moves*
- *Arc Moves*
- *Contoured Moves*
- *Reference Moves*
- *Blending Moves*
- *Electronic Gearing and Camming*
- *Acquiring Time-Sampled Position and Velocity Data*

- *Synchronization*
- *Torque Control*
- *Onboard Programs*

4

What You Need to Know about Moves

This chapter discusses the concepts necessary for programming motion control.

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 in Figure 4-1.

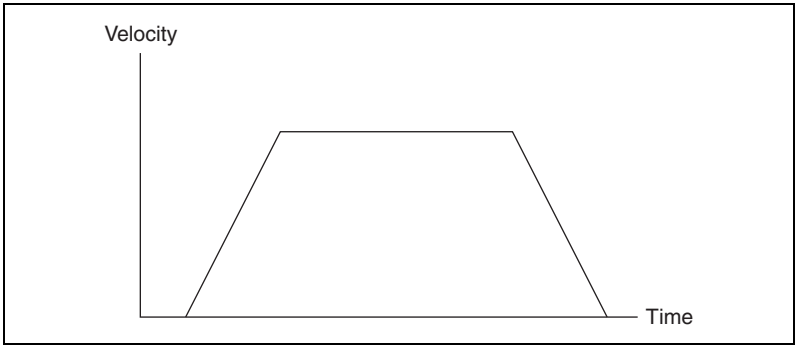


Figure 4-1. Trapezoidal Move Profile

S-Curve

The acceleration and deceleration portions of an s-curve motion profile are smooth, resulting in less abrupt transitions, as shown in Figure 4-2. 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.

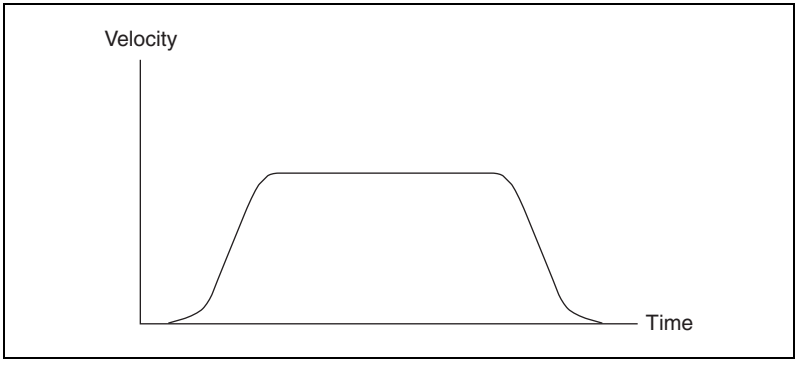


Figure 4-2. S-Curve Move Profile

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**—Is 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, such as the XYZ axis shown in Figure 4-3. Arc moves always execute on a coordinate space.

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 Figure 4-3.

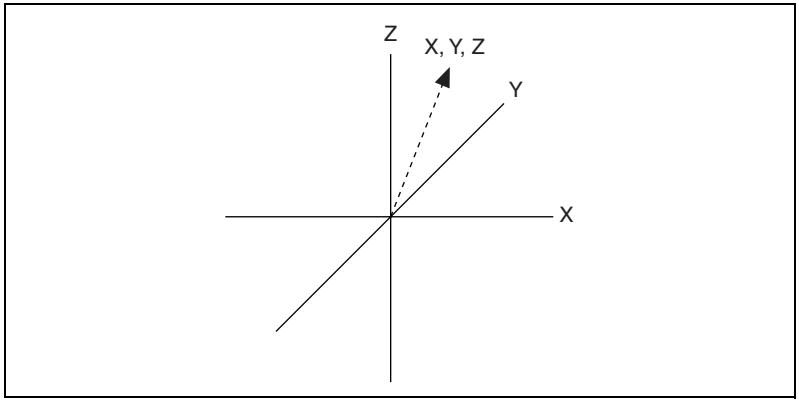


Figure 4-3. 3D Coordinate Space

Use the Configure Vector Space function 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, this behavior is 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.

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 73*xx* Floating-Point versus Fixed-Point

Note The information in this section applies only to NI 73*xx* motion controllers. These restrictions are not applicable to the NI SoftMotion Controller.

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.

NI 73*xx* Time Base

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 function. This means that the range for these parameters depends on the update rate selected, as shown in the following examples.

Table 4-1 lists minimum and maximum update rates for acceleration and velocity in various units.

NI 73*xx* 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.

Table 4-1 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.

Table 4-1. Velocity in Counts/Min

Update Rate	MIN	MAX	Calculation Based on Units
62.5 µs	14.648438 counts/min	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 7350 • 240 million counts/min for 7330, 7340, and 7390	$\pm RPM_{min} = MIN \times 1/r$ 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:

$$\text{minimum RPM} = Vmin \times \left(\frac{1}{Ts}\right) \times 60 \times \left(\frac{1}{R}\right)$$

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

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

You also can calculate the minimum velocity using the formula shown in the *Calculation Based on Units* column in Table 4-1.

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

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

or

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

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

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

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:

$$\text{velocity} \leq (65{,}536 \times \text{deceleration}) - \text{acceleration}$$

where velocity is in counts/sample and acceleration and deceleration are in counts/sample^2.

From the example, the maximum RPM is

$$(20 \times 10^6) \times 60 \times \left(\frac{1}{2{,}000}\right) = 600{,}000 \text{ RPM}$$

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

NI 73*xx* 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 function, as shown in the following formula:

$$\text{Velocity in counts or steps/s} = Vmin \times \left(\frac{1}{Ts}\right)$$

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

Ts = sample period in seconds per sample

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

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

The Load Velocity function 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 function.

You can calculate the maximum velocity with the following equation:

$$\text{maximum velocity} = Vmax$$

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:

$$\text{velocity} \leq (65{,}536 \times \text{deceleration}) - \text{acceleration}$$

where velocity is in counts/sample and acceleration and deceleration are in counts/sample^2.

NI 73*xx* Acceleration in Counts/s^2

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

Table 4-2 shows the minimum and maximum acceleration update rates in counts/sec^2.

Table 4-2. Acceleration Update Rate in Counts/Sec^2

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

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

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

where $Amin$ = 1/65,536 counts/sample^2 or steps/sample^2

Ts = sample period in seconds per sample

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

$$\left(\frac{1}{65536}\right) \times \left(\frac{1}{250\mu s}\right)^2 = 244 \text{ counts/second}^2$$

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

$$\text{maximum acceleration/deceleration} = Amax \times \left(\frac{1}{Ts}\right)^2$$

where $Amax$ = 32 counts/sample^2

Ts = sample period in seconds per sample

and is constrained according to the following equations:

$$\text{acceleration} \leq 256 \times \text{deceleration}$$

$$\text{deceleration} \leq 65536 \times \text{acceleration}$$

NI 73*xx* 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^2 or steps/sample^2 before being used by the trajectory generator.

Table 4-3 shows the minimum and maximum acceleration update rates in counts/sec^2.

Table 4-3. Acceleration Update Rate in RPS/s

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

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

$$\text{RPS/s} = Amin \times \left(\frac{1}{Ts}\right)^2 \times \left(\frac{1}{R}\right)$$

where $Amin$ = 1/65,536 counts/sample^2 or steps/sample^2

Ts = sample period in seconds per sample

R = counts or steps per revolution

or

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

For a typical servo axis with 2,000 counts or steps per revolution operating at the 250 µs 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:

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

where $Amax = 32$ counts/sample^2

Ts = sample period in seconds per sample

R = counts or steps per revolution

and is constrained according to the following equations:

$$\text{acceleration} \leq 256 \times \text{deceleration}$$

$$\text{deceleration} \leq 65536 \times \text{acceleration}$$

or

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

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

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

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

NI 73*xx* Velocity Override in Percent

The Load Velocity Override function 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 73*xx* Arc Angles in Degrees

The Load Circular Arc, Load Helical Arc, and Load Spherical Arc 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 = quotient, the integer multiple of 45°

R = remainder

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

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

$$\text{Angle in 16.16 format} = 2.\left(\frac{4.7^\circ}{45^\circ} \times 65{,}536\right) = \text{0x0002.1ABD}$$

The minimum angular increment is

$$\left(\frac{1}{65.536}\right) \times 45^\circ = 0.000687^\circ$$

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

NI 73*xx* Arc Move Limitations

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

Arc moves must use the following equations or an `NIMC_invalidVelocityError` is generated:

$$V \times P \times 4 \geq R$$

and

$$1{,}677{,}216 \geq \frac{(V \times P^2 \times 83{,}443)}{R \times I} \geq 16$$

where V = Velocity in counts/s

P = PID sample rate in seconds

I = Arc Interval (10 ms or 20 ms) in seconds

R = Radius in counts

Arc moves must use the following equations or an `NIMC_invalidAccelerationError` is generated:

$$A \times P \times 4 \geq R$$

and

$$65{,}536 \geq \frac{A \times P^3 \times 83{,}443}{R \times I^2} \geq 1$$

where P = PID sample rate in seconds

I = Arc Interval (10 ms or 20 ms) in seconds

R = Radius in counts

A = Acceleration/deceleration in counts/s^2

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.

5

Straight-Line Moves

A straight-line move executes the shortest move between two points.

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.

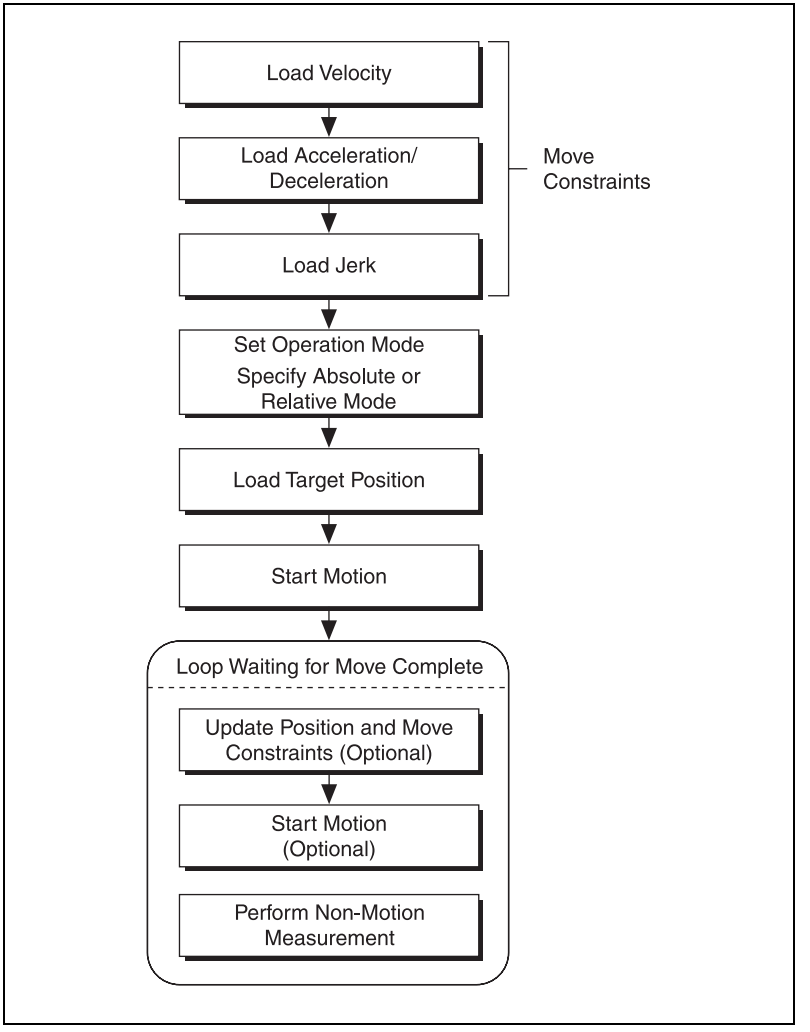
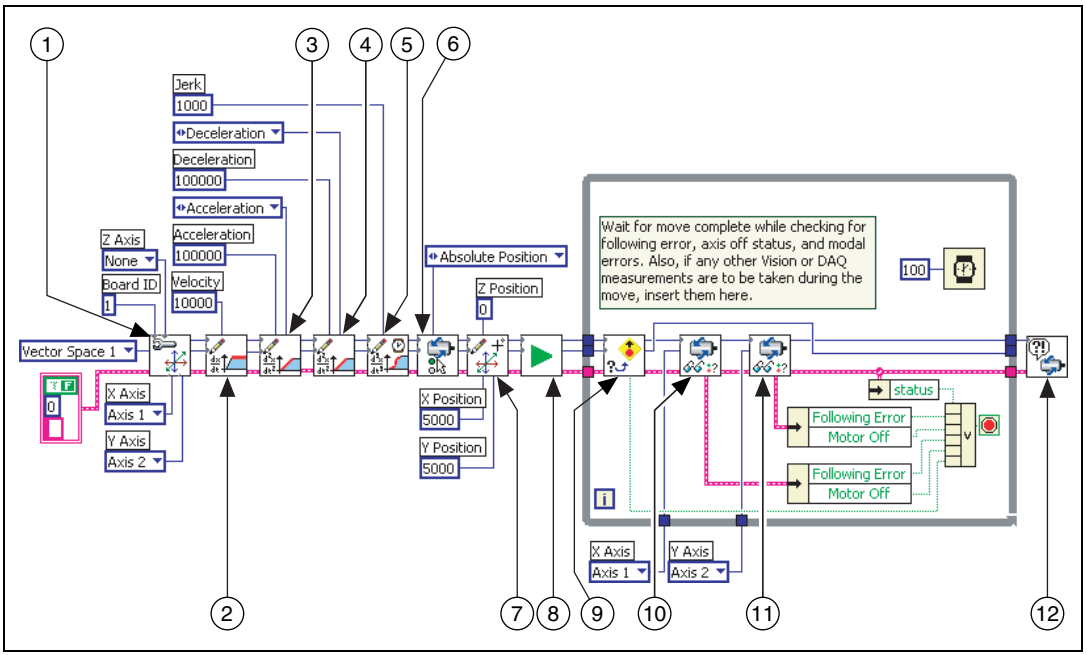
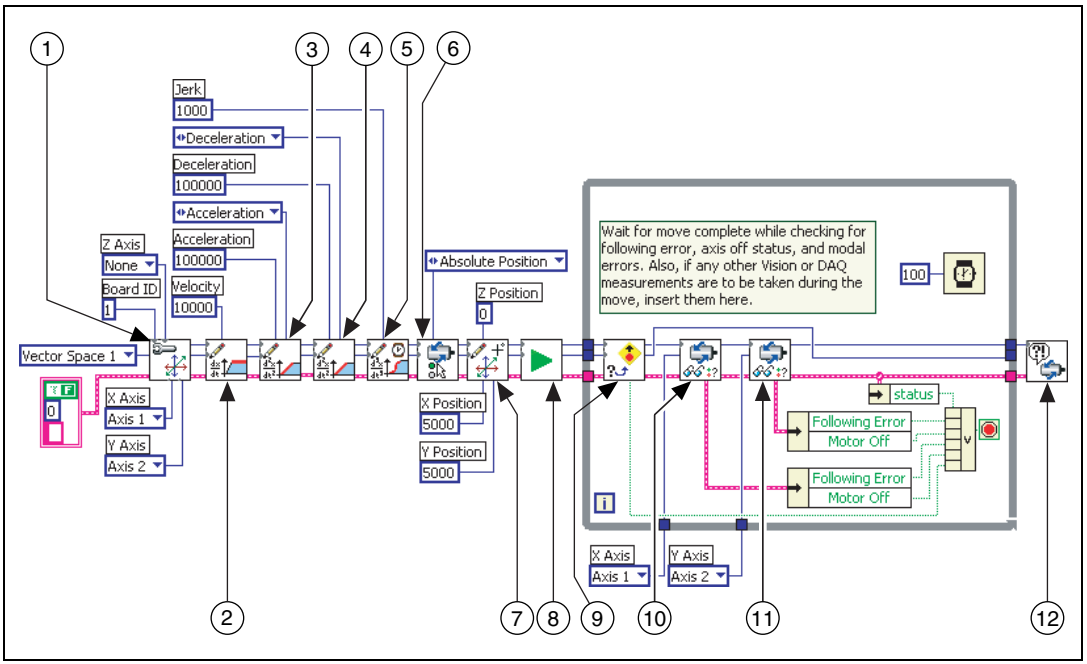


Figure 5-1. Position-Based Straight-Line Move Algorithm



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

Figure 5-2. 1D Straight-Line Move in LabVIEW



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 Target Position	11	Read per Axis Status
4	Load Acceleration/Deceleration	8	Start Motion	12	Motion Error Handler

Figure 5-3. 2D Straight-Line Move in LabVIEW

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.

1D Straight-Line Move Code

```
// 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 = NIMC_AXIS1;
    ////////////////////////////////

    // 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, 100000, 0xFF);
    CheckError;

    // Set the deceleration for the move (in
    counts/sec^2)
    err = flex_load_acceleration(boardID, axis,
    NIMC_DECELERATION, 100000, 0xFF);
    CheckError;

    // Set the jerk - scurve 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
err = flex_load_target_pos (boardID, axis, 5000,
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, &moveComplete);
    CheckError;

    // Check the following error/axis off status for
    the axis
    err = flex_read_axis_status_rtn(boardID, axis,
    &axisStatus);
    CheckError;

    // Read the communication status register and
    check the modal //errors
    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) && !(axisStatus &
NIMC_AXIS_OFF_BIT));
//Exit on move complete/following error/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 ID, and the
                error code of the //modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D, &resourceID,&errorCode);
                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
}
```

2D Straight-Line Move Code

```
// 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 vector space
    vectorSpace = NIMC_VECTOR_SPACE1;
```

```
// Configure a 2D vector space comprised 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_ACCELERATION, 100000, 0xFF);
CheckError;

// Set the deceleration for the move (in
counts/sec^2)
err = flex_load_acceleration(boardID, vectorSpace,
NIMC_DECELERATION, 100000, 0xFF);
CheckError;

// Set the jerk - scurve time (in sample periods)
err = flex_load_scurve_time(boardID, vectorSpace,
1000, 0xFF);
CheckError;

// Set the operation mode
err = flex_set_op_mode (boardID, vectorSpace,
NIMC_ABSOLUTE_POSITION);
CheckError;

// Load vector space position
err = flex_load_vs_pos (boardID, vectorSpace,
5000/*x Position*/, 10000/*y Position*/, 0/* z
Position*/, 0xFF);
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, &moveComplete);
```

```
    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 //errors
    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) && !(axisStatus &
NIMC_AXIS_OFF_BIT));

//Exit on move complete/following error/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 ID, and the
          error code of the //modal error from the
          error stack on the device
          flex_read_error_msg_rtn(boardID,&commandI
          D,&resourceID, &errorCode);
          nimcDisplayError(errorCode,commandID,res
          ourceID);

          //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-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.

Algorithm

Figure 5-4 is a generic algorithm applicable to both C/C++ and VI code.

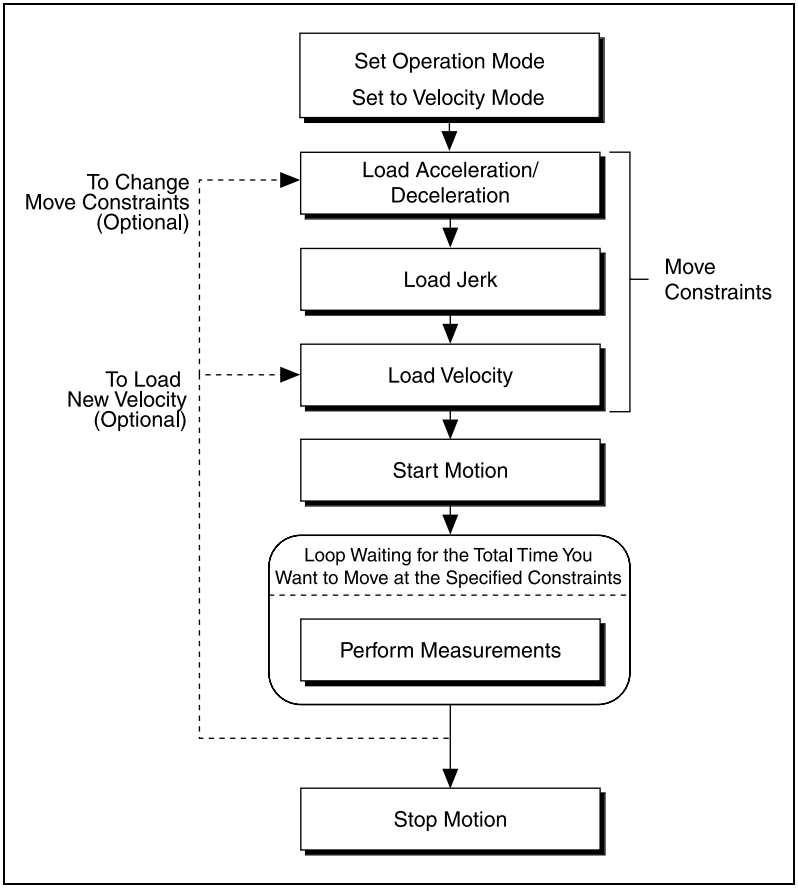


Figure 5-4. Velocity-Based Straight-Line Move Algorithm

Loading a second velocity and executing Start Motion 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 function. The velocity profile created in the example code is shown in Figure 5-5.

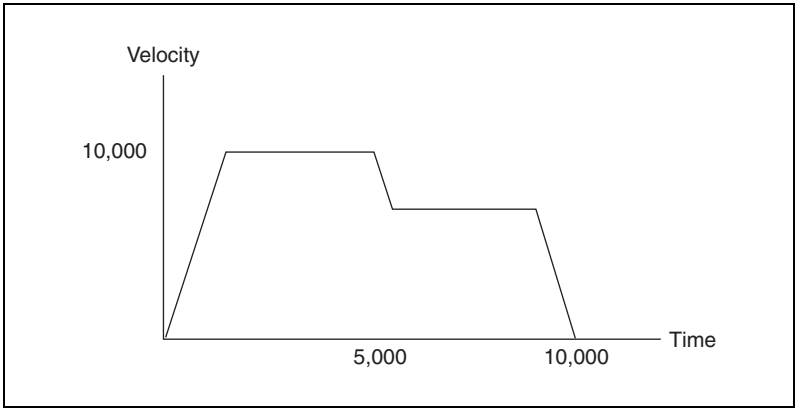
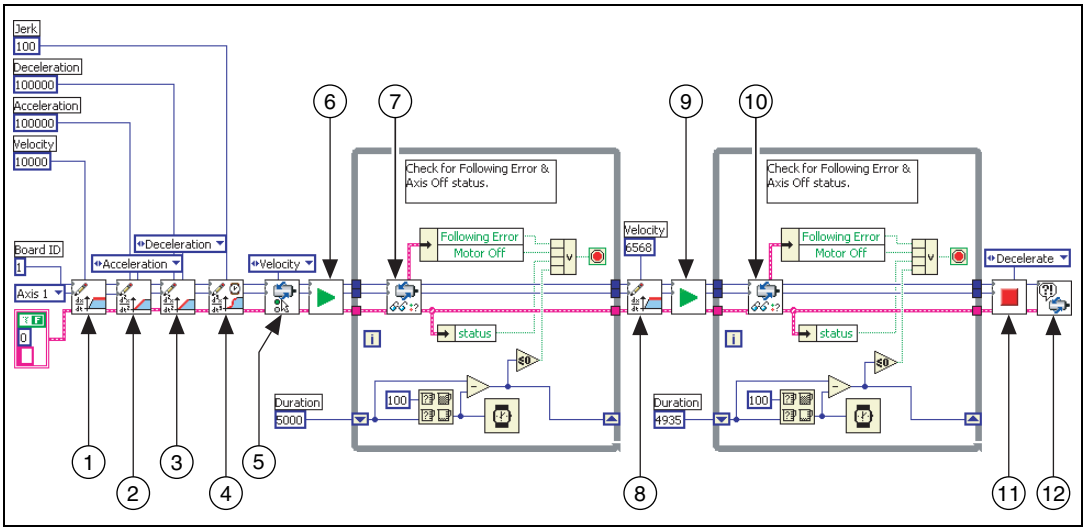


Figure 5-5. Velocity Profile

LabVIEW Code



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

Figure 5-6. Velocity-Based Straight-Line Move in LabVIEW

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 = NIMC_AXIS1;

// 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, 100000, 0xFF);
CheckError;

// Set the deceleration for the move (in
counts/sec^2)
err = flex_load_acceleration(boardID, axis,
NIMC_DECELERATION, 100000, 0xFF);
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 following error/axis off status
    err = flex_read_axis_status_rtn(boardID, axis,
    &axisStatus);
    CheckError;

    // Read the communication status register and
    check the modal //errors
    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 first //segment
    currentTime = timeGetTime();
    if((currentTime - initialTime) >= moveTime1)
    break;
    Sleep (100); //Check every 100 ms

}while (!(axisStatus) && !(axisStatus &
NIMC_FOLLOWING_ERROR_BIT) && !(axisStatus &
NIMC_AXIS_OFF_BIT)); //Exit on following error/axis
off
//-------------------------------------------------
// 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 following error/axis off status
    err = flex_read_axis_status_rtn(boardID, axis,
    &axisStatus);
```

```
    CheckError;
    // Read the communication status register and
    check the modal // errors
    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 //second segment
    currentTime = timeGetTime();
    if((currentTime - initialTime) >= moveTime2)
    break;
    Sleep (100); //Check every 100 ms
}while (!(axisStatus) && !(axisStatus &
NIMC_FOLLOWING_ERROR_BIT) && !(axisStatus &
NIMC_AXIS_OFF_BIT)); //Exit on move
complete/following error/axis off
// 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 ID, and the
          error code of the //modal error from the
          error stack on the device
          flex_read_error_msg_rtn(boardID,&commandI
          D,&resourceID, &errorCode);
          nimcDisplayError(errorCode,commandID,res
          ourceID);
          //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 Load Velocity Override 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.

Algorithm

Figure 5-7 is a generic algorithm applicable to both C/C++ and VI code.

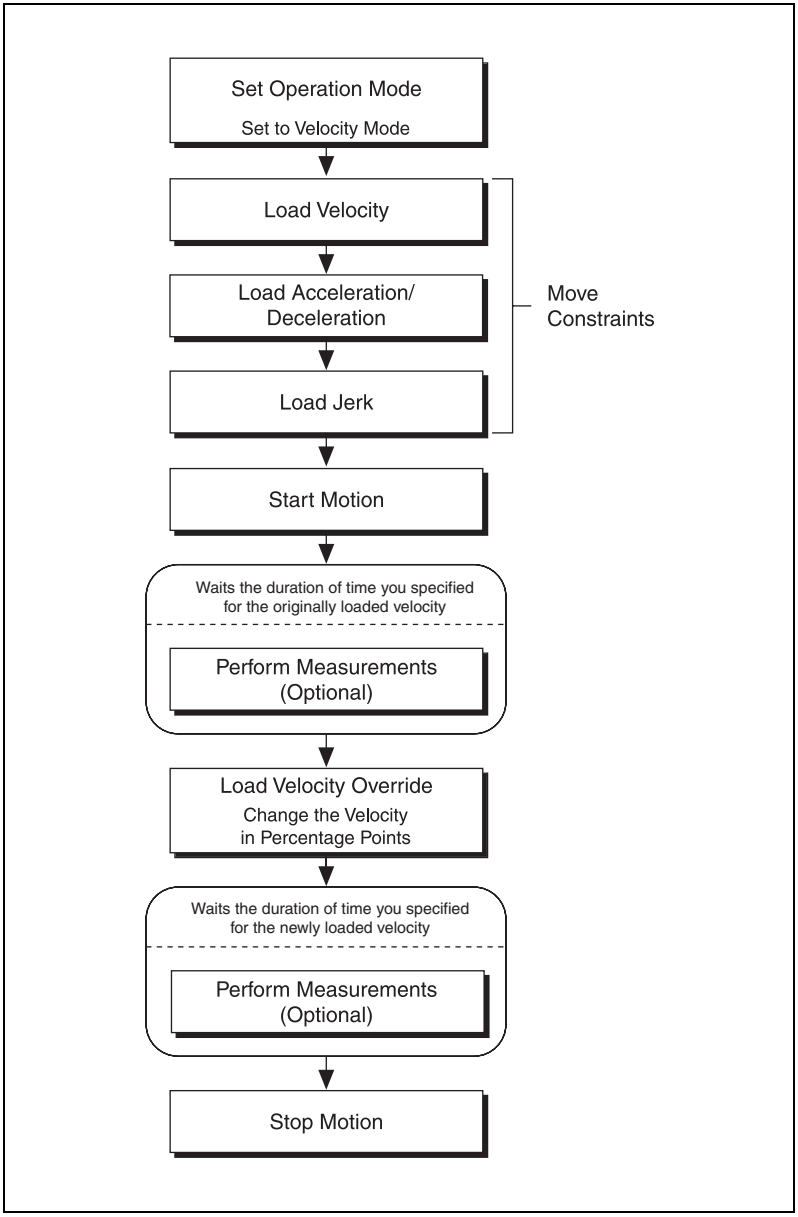
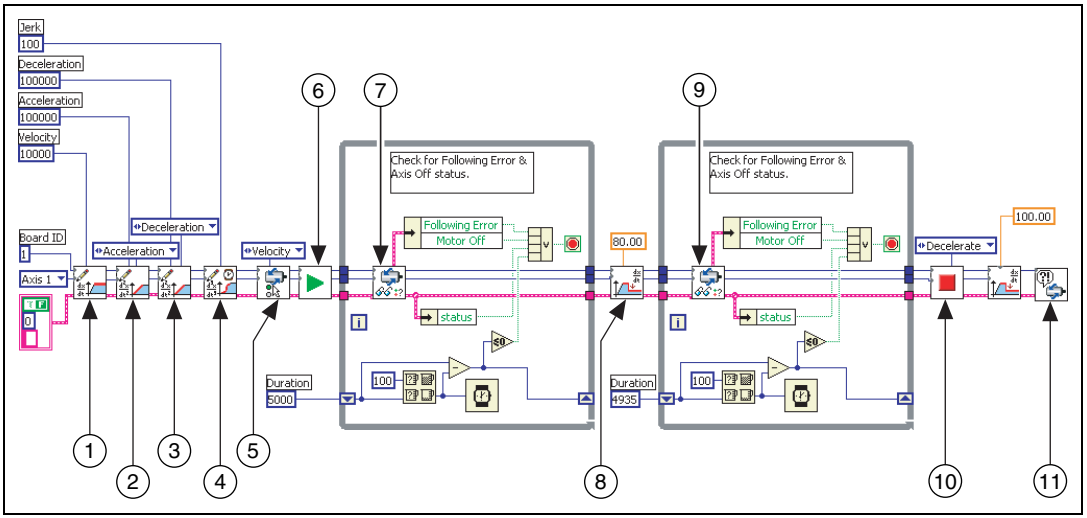


Figure 5-7. Velocity Override Algorithm

LabVIEW Code



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		

Figure 5-8. Velocity-Based Move Using Velocity Override in LabVIEW

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 = NIMC_AXIS1;
    // 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, 100000, 0xFF);
    CheckError;
```

```
// Set the deceleration for the move (in
counts/sec^2)
err = flex_load_acceleration(boardID, axis,
NIMC_DECELERATION, 100000, 0xFF);
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 following error/axis off status
    err = flex_read_axis_status_rtn(boardID, axis,
    &axisStatus);
    CheckError;

    // Read the communication status register and
    check the modal //errors
    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 first //segment
    currentTime = timeGetTime();
    if((currentTime - initialTime) >= moveTime1)
    break;

    Sleep (100); //Check every 100 ms
```

```
}while (!(axisStatus) && !(axisStatus &
NIMC_FOLLOWING_ERROR_BIT) && !(axisStatus &
NIMC_AXIS_OFF_BIT)); //Exit on move
//complete/following error/axis off

//-------------------------------------------------
// 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 following error/axis off status
    err = flex_read_axis_status_rtn(boardID, axis,
    &axisStatus);
    CheckError;

    // Read the communication status register and
    check the modal //errors
    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 second //segment
    currentTime = timeGetTime();
    if((currentTime - initialTime) >=  moveTime2)
    break;

    Sleep (100); //Check every 100 ms
}while (!(axisStatus) && !(axisStatus &
NIMC_FOLLOWING_ERROR_BIT) && !(axisStatus &
NIMC_AXIS_OFF_BIT)); //Exit on move
complete/following error/axis off

// 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{
              //Get the command ID, resource ID, and the
              error code of the //modal error from the
              error stack on the device
              flex_read_error_msg_rtn(boardID,&commandI
              D,&resourceID, &errorCode);
              nimcDisplayError(errorCode,commandID,res
              ourceID);

              //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
}
```

6

Arc Moves

An arc move causes a coordinate space of axes to move on a circular, spherical, or helical 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 Figure 6-1.

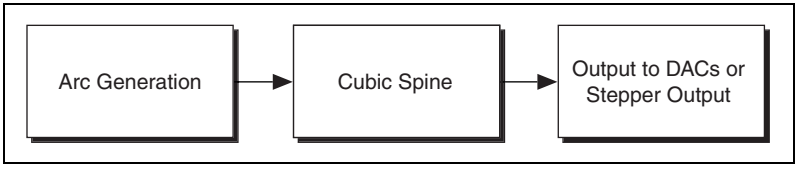


Figure 6-1. Arc Move Data Path

Circular Arcs

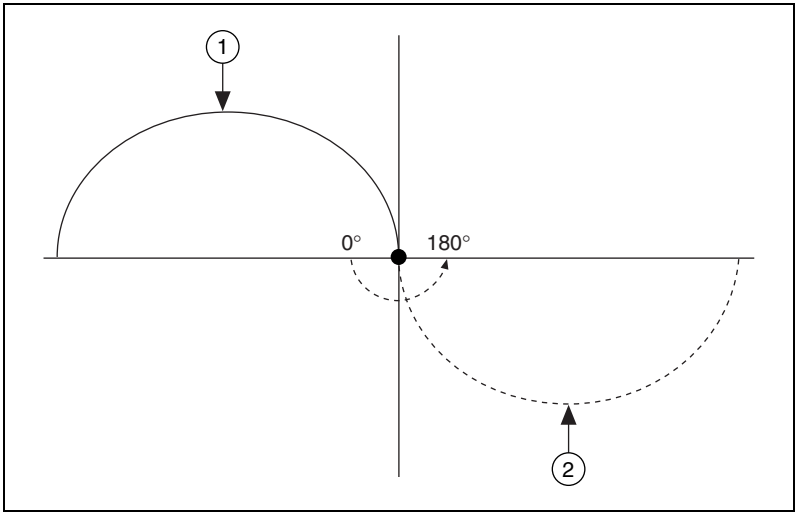
A circular arc defines an arc in the XY plane of a 2D or 3D coordinate space. The arc is specified by a radius, starting angle, and travel angle. Also, like all coordinate space moves, the arc uses the values of move constraints—maximum velocity, maximum acceleration, and maximum deceleration.

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 73*xx* 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:

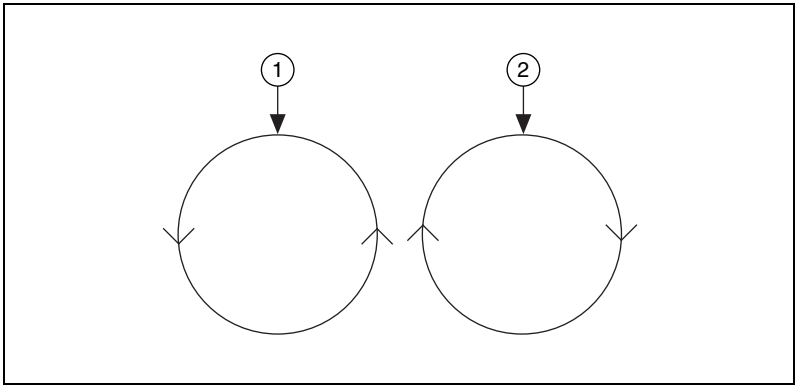
- **Radius**—Specifies the distance from the center of the arc to its edge
- **Start Angle**—Orients the arc on its plane using the starting point as an axis to spin around. Because the starting point for a new arc is fixed based on the current position, moving its center around the starting point alters the orientation of a new arc. For example, Figure 6-2 shows the effect of changing the start angle from 0° to 180°.



1 Original Arc 2 Arc with 180° Start Angle

Figure 6-2. Rotating Start Angle

- **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 Figure 6-3.



1 Positive Travel Angle 2 Negative Travel Angle

Figure 6-3. Positive and Negative Travel Angles

Arc Move Algorithm

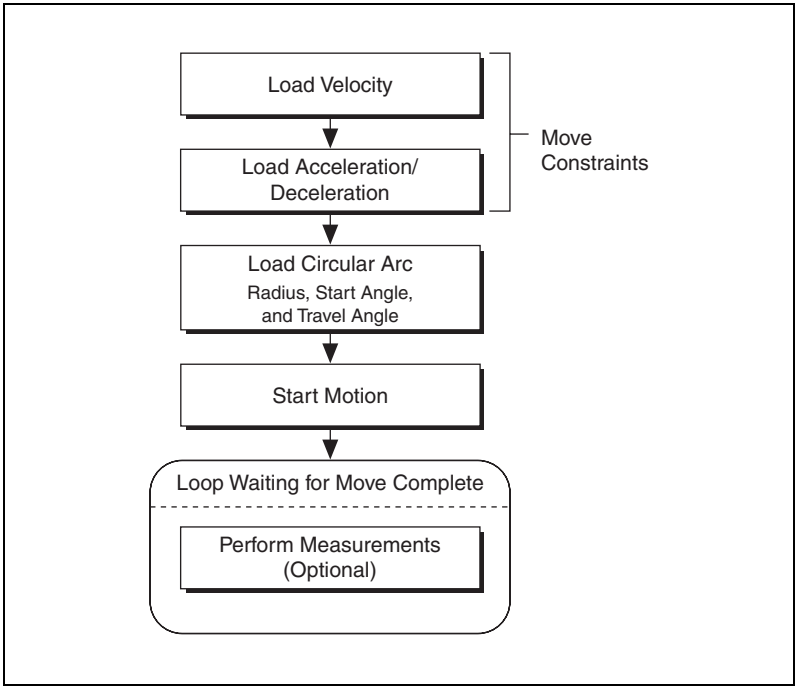
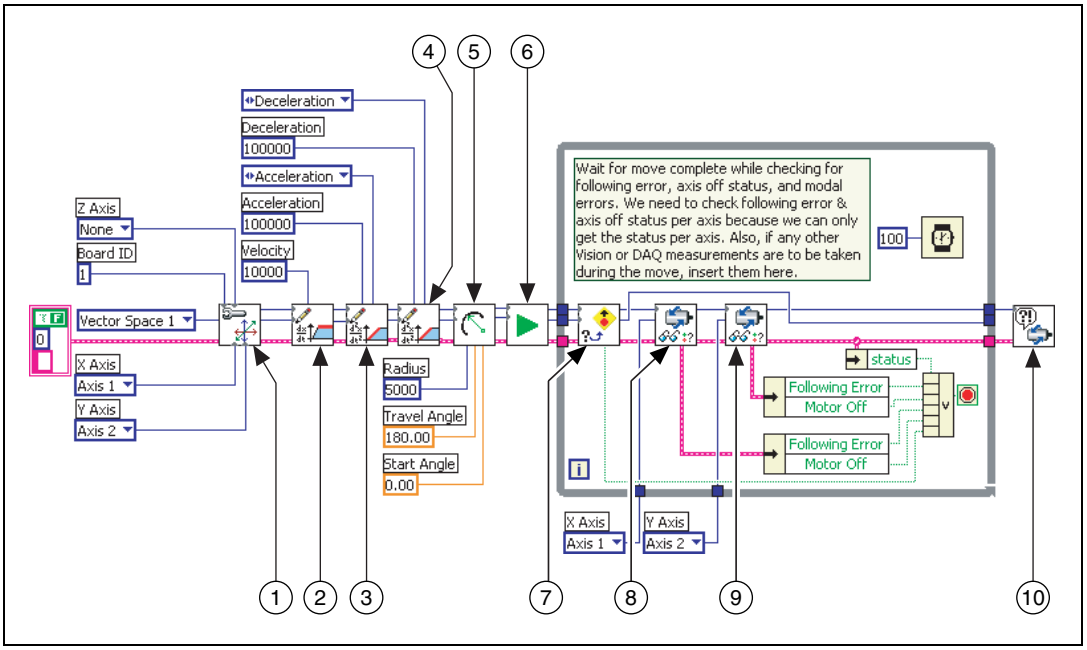


Figure 6-4. Circular Arc Move Algorithm

LabVIEW Code



1	Configure Vector Space	5	Load Circular Arc	8	Read per Axis Status
2	Load Velocity	6	Start Motion	9	Read per Axis Status
3	Load Acceleration/Deceleration	7	Check Move Complete Status	10	Motion Error Handler
4	Load Acceleration/Deceleration				

Figure 6-5. Circular Arc Move in LabVIEW

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 vector space number
vectorSpace = NIMC_VECTOR_SPACE1;
////////////////////////////////

// Configure a 2D vector space comprising of axes 1
and 2
err = flex_config_vect_spc(boardID, vectorSpace,
NIMC_AXIS1, NIMC_AXIS2, 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_ACCELERATION, 100000, 0xFF);
CheckError;

// Set the deceleration for the move (in
counts/sec^2)
err = flex_load_acceleration(boardID, vectorSpace,
NIMC_DECELERATION, 100000, 0xFF);
CheckError;

// Load Spherical Arc
err = flex_load_circular_arc (boardID, vectorSpace,
5000/*radius*/, 0.0/*startAngle*/,
180.0/*travelAngle*/, 0xFF);
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, &moveComplete);
        CheckError;

        // Check the following error/axis off status for
        axis 1
        err = flex_read_axis_status_rtn(boardID,
        NIMC_AXIS1, &status);
        CheckError;
        axisStatus |= status;

        // Check the following error/axis off status for
        axis 2
        err = flex_read_axis_status_rtn(boardID,
        NIMC_AXIS2, &status);
        CheckError;
        axisStatus |= status;

        // Read the communication status register and
        check the modal // errors
        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) && !(axisStatus &
    NIMC_AXIS_OFF_BIT));
    //Exit on move complete/following error/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 ID and the
              error code of the //modal error from the
              error stack on the device
```

```
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);
                //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
}
```

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 and yaw from the normal 3D coordinate space (XYZ), as shown in Figures 6-6 and 6-7.

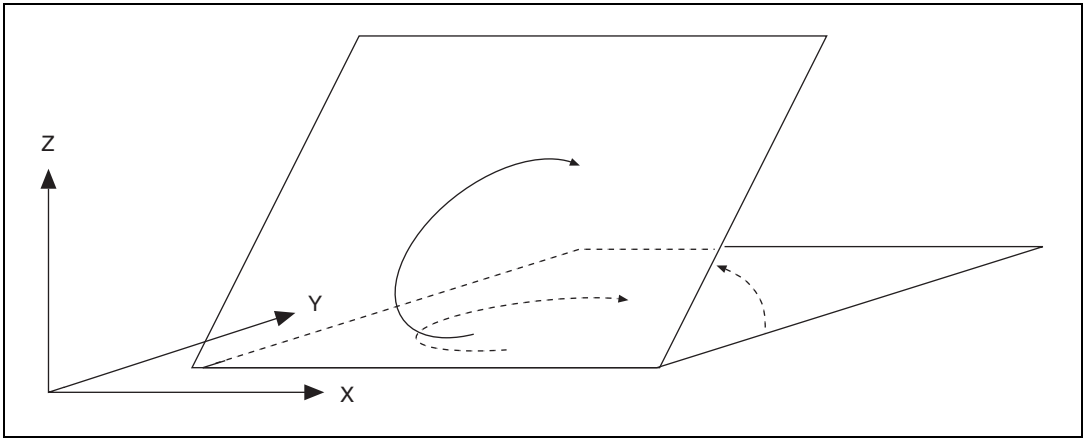


Figure 6-6. Changing Pitch by Rotating the X Axis

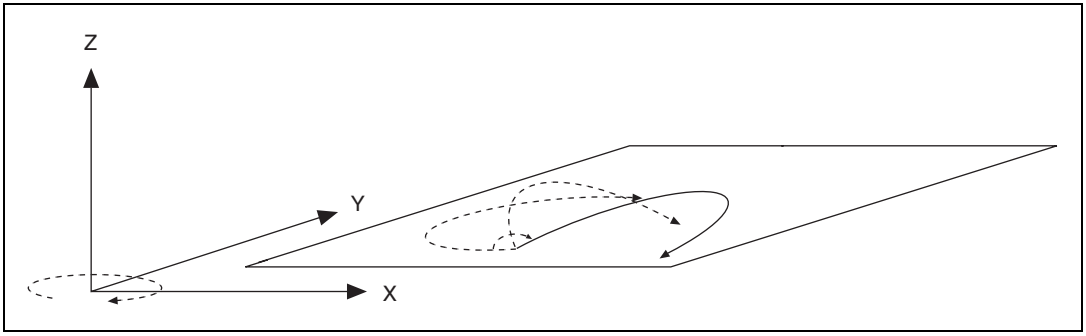


Figure 6-7. Changing Yaw by Rotating the Z Axis

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.

Algorithm

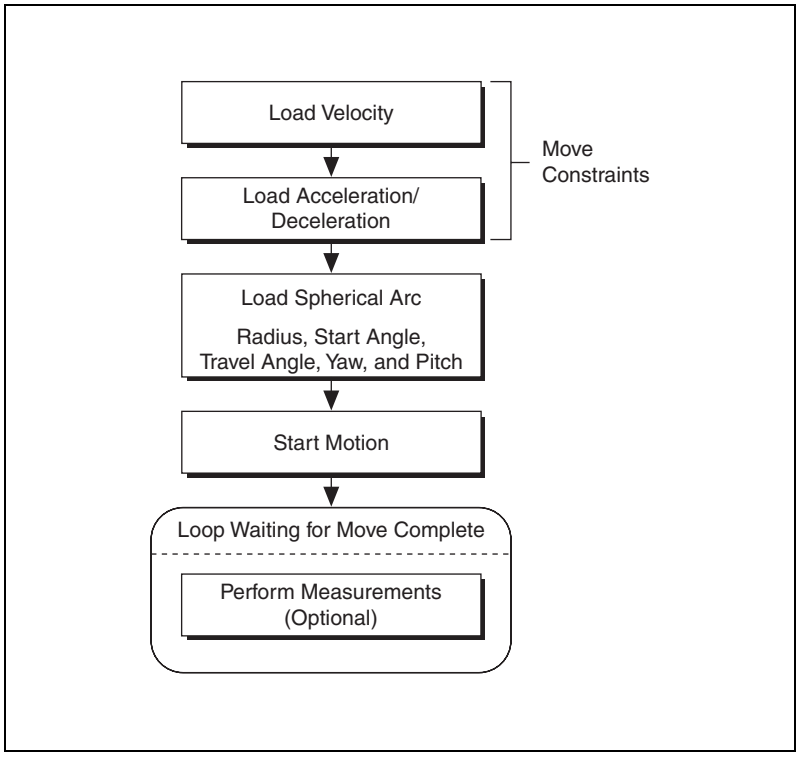
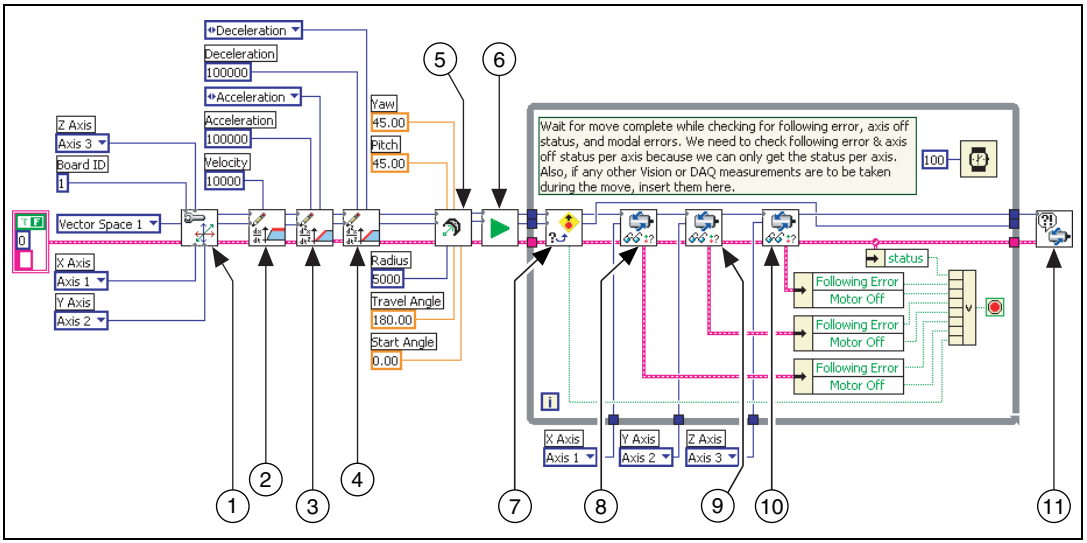


Figure 6-8. Spherical Arc Algorithm

LabVIEW Code



1	Configure Vector Space	5	Load Spherical Arc	9	Read per Axis Status
2	Load Velocity	6	Start Motion	10	Read per Axis Status
3	Load Acceleration/Deceleration	7	Check Move Complete Status	11	Motion Error Handler
4	Load Acceleration/Deceleration	8	Read per Axis Status		

Figure 6-9. Spherical Arc Move in LabVIEW

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 vector space number
vectorSpace = NIMC_VECTOR_SPACE1;
////////////////////////////////

// Configure a 3D vector space comprising of axes 1,
2 and 3
err = flex_config_vect_spc(boardID, vectorSpace,
NIMC_AXIS1, NIMC_AXIS2, NIMC_AXIS3);
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_ACCELERATION, 100000, 0xFF);
CheckError;

// Set the deceleration for the move (in
counts/sec^2)
err = flex_load_acceleration(boardID, vectorSpace,
NIMC_DECELERATION, 100000, 0xFF);
CheckError;

// Load Spherical Arc
err = flex_load_spherical_arc (boardID, vectorSpace,
5000/*radius*/, 45.0/*planePitch*/,
45.0/*planeYaw*/, 0.0/*startAngle*/,
180.0/*travelAngle*/, 0xFF);
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, &moveComplete);
    CheckError;
```

```
    // Check the following error/axis off status for
    axis 1
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS1, &status);
    CheckError;
    axisStatus |= status;

    // Check the following error/axis off status for
    axis 2
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS2, &status);
    CheckError;
    axisStatus |= status;

    // Check the following error/axis off status for
    axis 3
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS3, &status);
    CheckError;
    axisStatus |= status;

    //Read the communication status register and
    check the modal //errors
    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) && !(axisStatus &
NIMC_AXIS_OFF_BIT));

//Exit on move complete/following error/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{
```

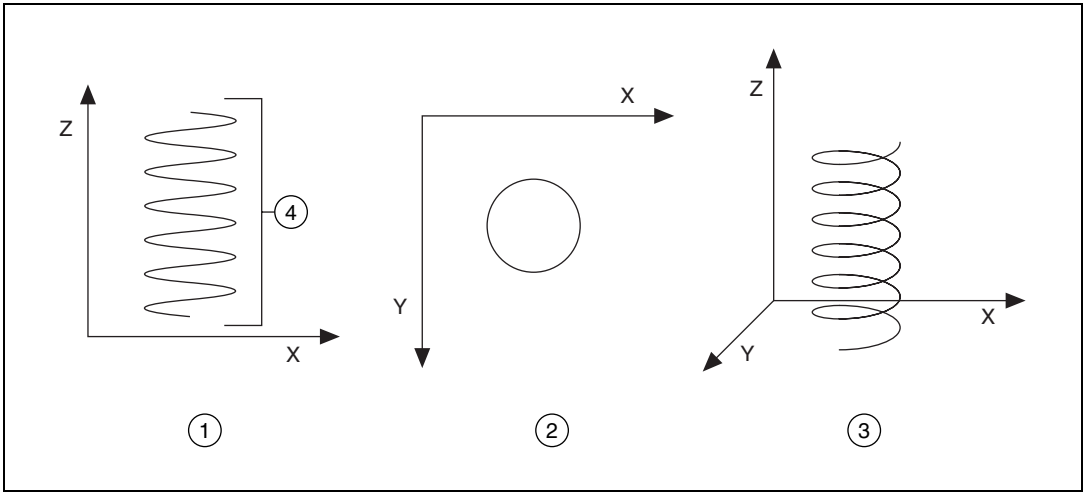
```
                //Get the command ID, resource ID, and the
                error code of the //modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);

                //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 Figure 6-10.



1 Side View of Helix 2 Top View of Helix 3 Isometric View of Helix 4 Linear Travel

Figure 6-10. Helical Arc

Algorithm

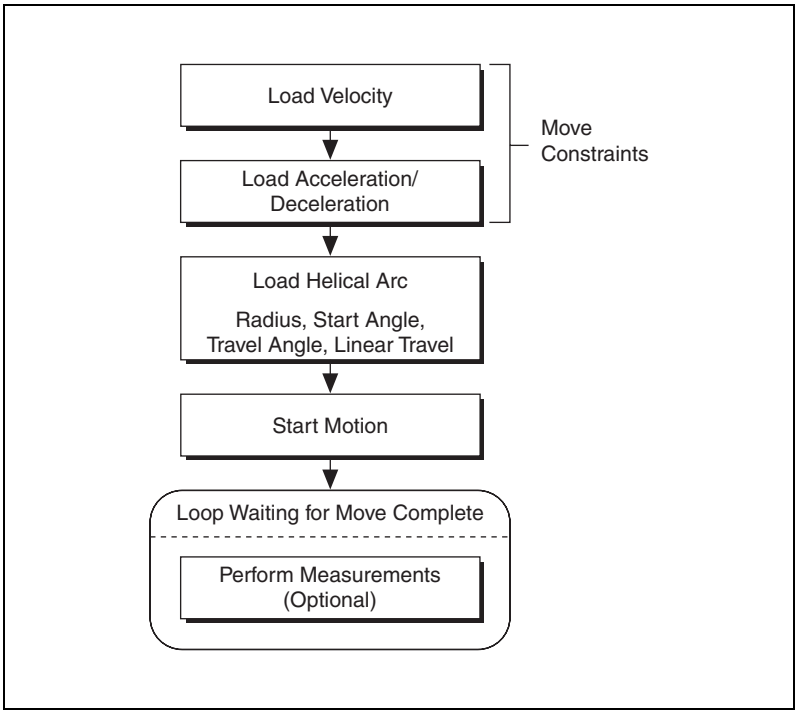
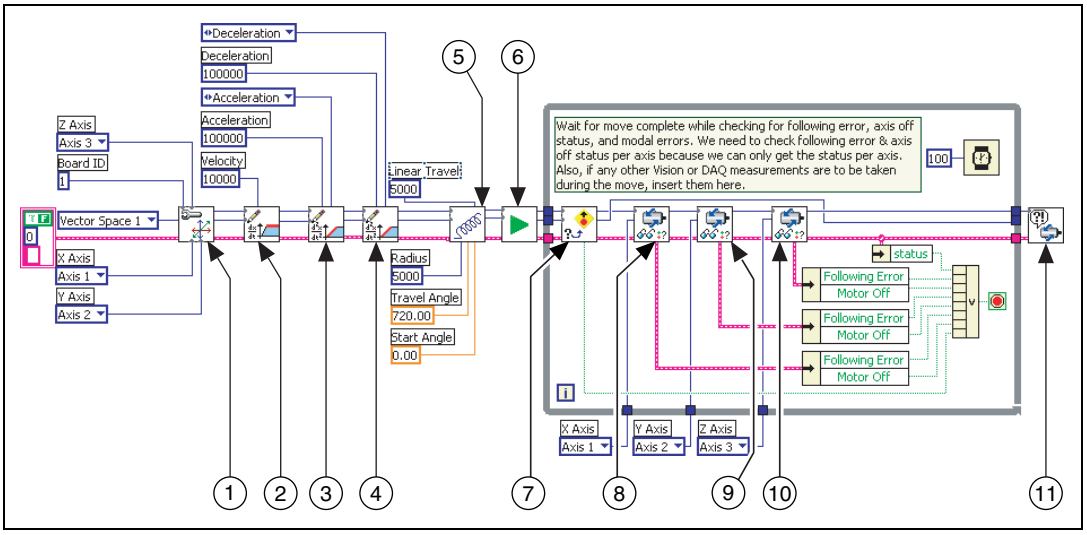


Figure 6-11. Helical Arc Algorithm

LabVIEW Code



1	Configure Vector Space	5	Load Helical Arc	9	Read per Axis Status
2	Load Velocity	6	Start Motion	10	Read per Axis Status
3	Load Acceleration/Deceleration	7	Check Move Complete Status	11	Motion Error Handler
4	Load Acceleration/Deceleration	8	Read per Axis Status		

Figure 6-12. Helical Arc Move in LabVIEW

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 vector space number
vectorSpace = NIMC_VECTOR_SPACE1;
////////////////////////////////

// Configure a 3D vector space comprising of axes 1,
2 and 3
err = flex_config_vect_spc(boardID, vectorSpace,
NIMC_AXIS1, NIMC_AXIS2, NIMC_AXIS3);
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_ACCELERATION, 100000, 0xFF);
CheckError;

// Set the deceleration for the move (in
counts/sec^2)
err = flex_load_acceleration(boardID, vectorSpace,
NIMC_DECELERATION, 100000, 0xFF);
CheckError;

// Load Helical Arc
err = flex_load_helical_arc (boardID, vectorSpace,
5000/*radius*/, 0.0/*startAngle*/,
720.0/*travelAngle*/, 5000 /*linear travel*/, 0xFF);
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, &moveComplete);
    CheckError;

    // Check the following error/axis off status for
    axis 1
```

```
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS1, &status);
    CheckError;
    axisStatus |= status;

    // Check the following error/axis off status for
    axis 2
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS2, &status);
    CheckError;
    axisStatus |= status;

    // Check the following error/axis off status for
    axis 3
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS3, &status);
    CheckError;
    axisStatus |= status;

    //Read the communication status register and
    check the modal //errors
    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) && !(axisStatus &
NIMC_AXIS_OFF_BIT));
//Exit on move complete/following error/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 ID, and the
           error code of the //modal error from the
           error stack on the device
```

```
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);
                //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
}
```

7

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 Figure 7-1.

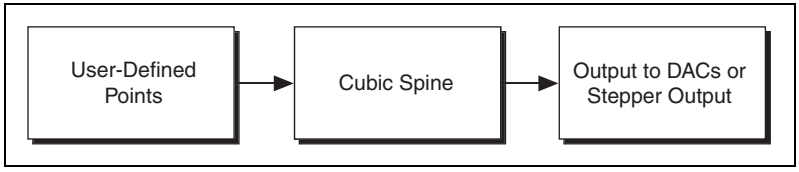


Figure 7-1. Contoured Move Data Path

Overview

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:

$$\text{Point}[n] = \text{Point}[n\text{-}1] + \text{bufferPosition}[n]$$

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 $2^{15} - 1$.

Arbitrary Contoured Moves

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

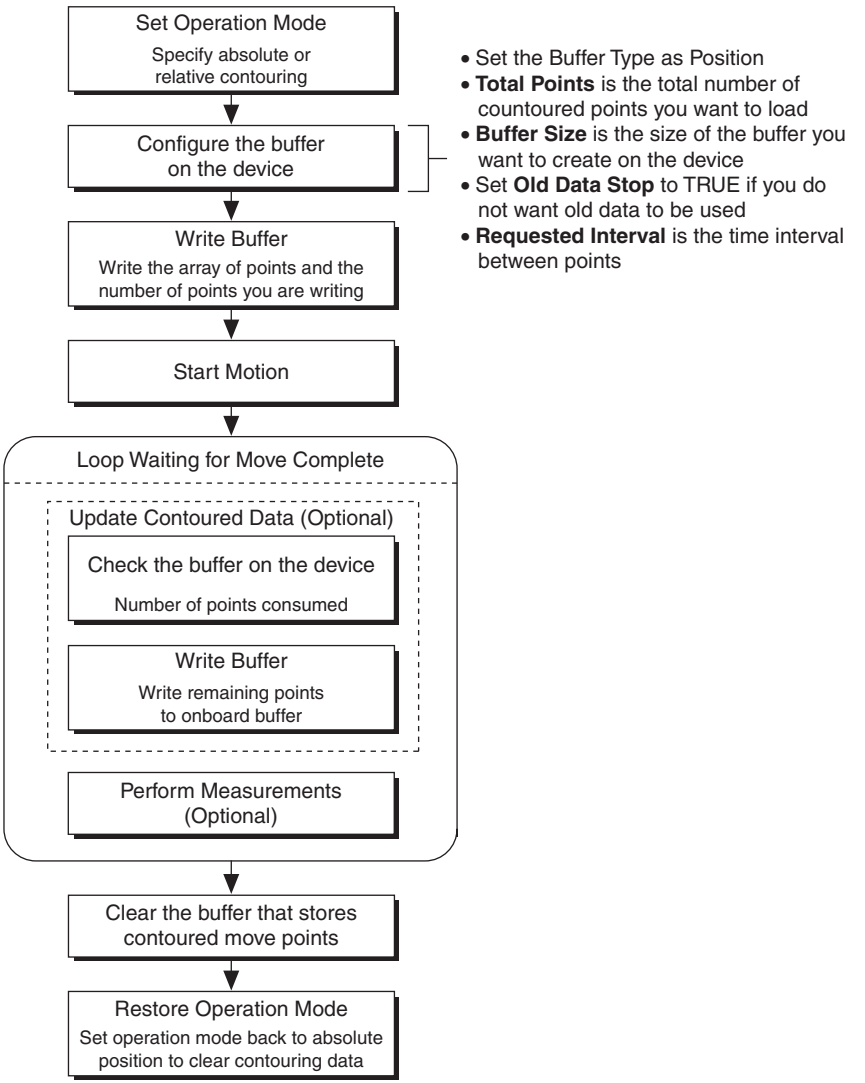


Figure 7-2. Contoured Move Algorithm

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

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.

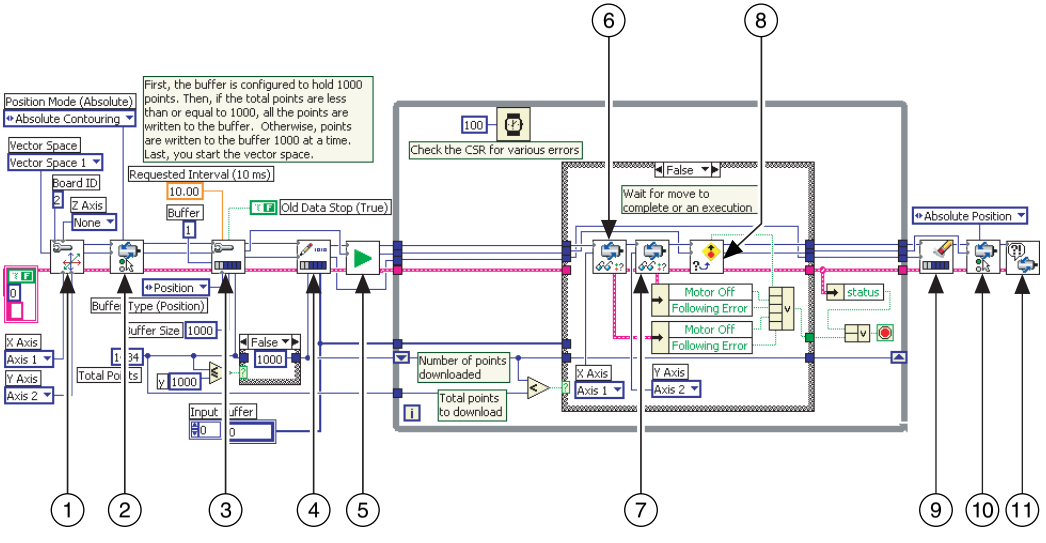
1	3	6	10	14	18	22	25	27	28

Figure 7-3. Absolute Contouring Buffer Values

1	2	3	4	4	4	4	3	2	1

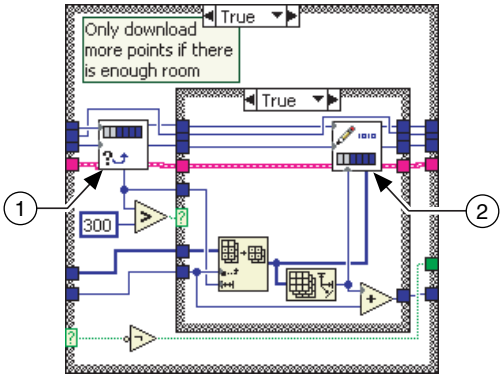
Figure 7-4. Relative Contouring Buffer Values

LabVIEW Code



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

Figure 7-5. Contoured Move in LabVIEW



1	Check Buffer	2	Write Buffer

Figure 7-6. Contoured Move True Case in LabVIEW

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;// Temporary copy of status
    u16 moveComplete;// Move complete status
    i32 i;
    i32 points[1994] =NIMC_SPIRAL_ARRAY;// Array of 2D
    points to move
    u32 numPoints = 1994;//Total number of points to
    contour through
    i32 bufferSize = 1000;// The size of the buffer to
    allocate on the //motion controller
    f64 actualInterval;// The interval at which the
    motion controller can // really contour
```

```
i32* downloadData = NULL;// The temporary array that
is created to // download the points to the motion
controller
u32 currentDataPoint = 0;// Indicates the next point
in the points // array to download
i32 backlog;// Indicates the available space to
download more //points
u16 bufferState;// Indicates the state of the onboard
buffer
u32 pointsDone;// Indicates the number of points that
have been // consumed
u32 dataCopied = 0;// Keeps track 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 vector space number
vectorSpace = NIMC_VECTOR_SPACE1;
////////////////////////////////

// Configure a 2D vector space comprising of axes 1
and 2
err = flex_config_vect_spc(boardID, vectorSpace,
NIMC_AXIS1, NIMC_AXIS2, NIMC_NOAXIS);
CheckError;

//Set the operation mode to absolute position
err = flex_set_op_mode(boardID, vectorSpace,
NIMC_ABSOLUTE_CONTOURING);
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*/, vectorSpace, NIMC_POSITION_DATA,
bufferSize, numPoints, NIMC_TRUE, 10,
&actualInterval);
CheckError;

// Send the first 1,000 points of the data
downloadData = malloc(sizeof(i32)*bufferSize);
```

```
for(i=0;i<bufferSize;i++){
    downloadData[i] = points[currentDataPoint++];
}
err = flex_write_buffer(boardID, 1/*buffer number*/,
bufferSize, NIMC_REGENERATION_NO_CHANGE,
downloadData, 0xFF);
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 // every 50 milliseconds
    Sleep(50);
    // Check to see if there are more points to
    download
    if(currentDataPoint < numPoints){
           err = flex_check_buffer_rtn(boardID,
           1/*buffer number*/, &backlog,
           &bufferState, &pointsDone);
           CheckError;
           if(backlog >= 300){
                  downloadData =
                  malloc(sizeof(i32)*backlog);
                  dataCopied = 0;
                  for(i=0;i<backlog;i++){
                         if(currentDataPoint >
                         numPoints) break;
                         downloadData[i] =
                         points[currentDataPoint++];
                         dataCopied++;
                  }
                  err = flex_write_buffer (boardID, 1
                  /*buffer number*/, dataCopied,
                  NIMC_REGENERATION_NO_CHANGE,
                  downloadData, 0xFF);
                  free(downloadData);
                  downloadData = NULL;
```

```
                CheckError;
            }
        }

        // Check the move complete status
        err = flex_check_move_complete_status(boardID,
        vectorSpace, 0, &moveComplete);
        CheckError;

        if(moveComplete) break;

        // Check for axis off status/following error or
        modal errors

        //Read the communication status register check
        the modal errors
        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,
        NIMC_AXIS1, &status);
        CheckError;
        axisStatus |= status;
        err = flex_read_axis_status_rtn(boardID,
        NIMC_AXIS2, &status);
        CheckError;
        axisStatus |= status;
        if( (axisStatus & NIMC_FOLLOWING_ERROR_BIT) ||
        (axisStatus & NIMC_AXIS_OFF_BIT) ){
        break;//Break out of the for loop because an axis
        was killed

        }
    }

    // Set the mode back to absolute mode to get the
    motion controller out of // contouring mode
    err = flex_set_op_mode(boardID, vectorSpace,
    NIMC_ABSOLUTE_POSITION);
    CheckError;
```

```
    // Free the buffer allocated on the motion 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 ID, and the
                error code of the // modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);

                //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
}
```

8

Reference Moves

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

Use the Check Reference function 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 the Wait Reference function if there is no need to monitor the status of the Find Reference function.

Find Reference Move

Use a Find Reference move to initiate a search operation to find a reference position. Available search operations include home switch, index pulse, forward limit switch, reverse limit switch, center, or run sequence. Refer to the *NI-Motion VI Help* or the *NI-Motion Function Help* for information about reference move options.

Reference Move Algorithm

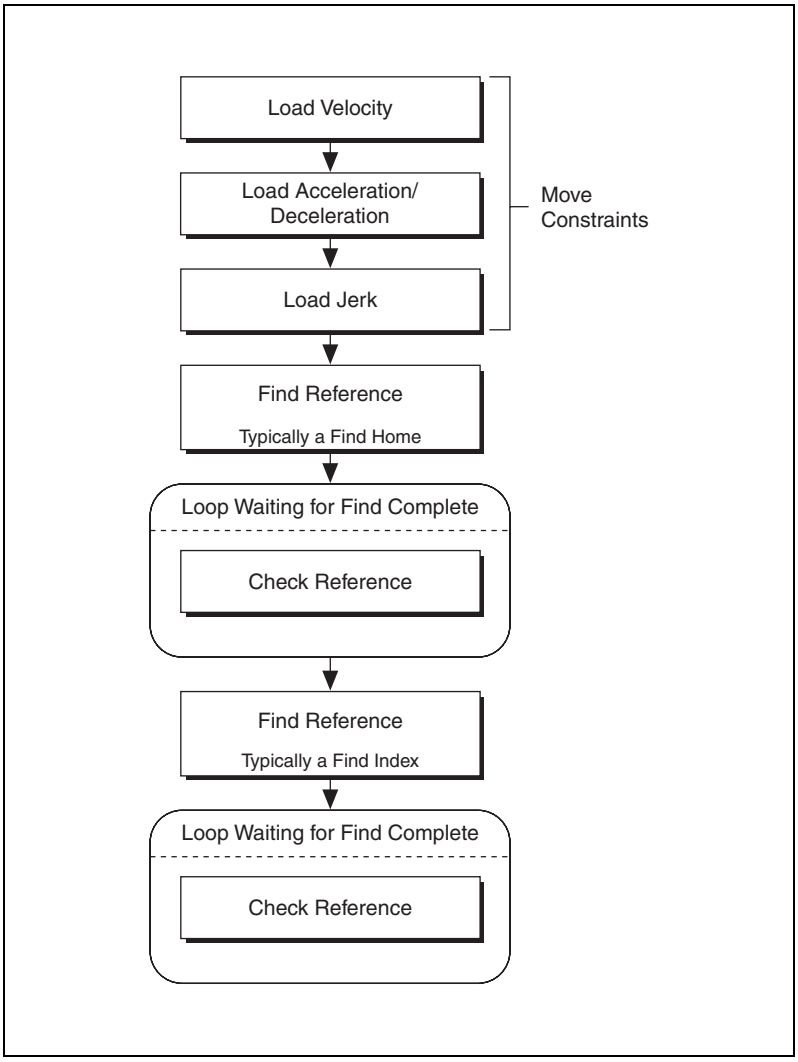
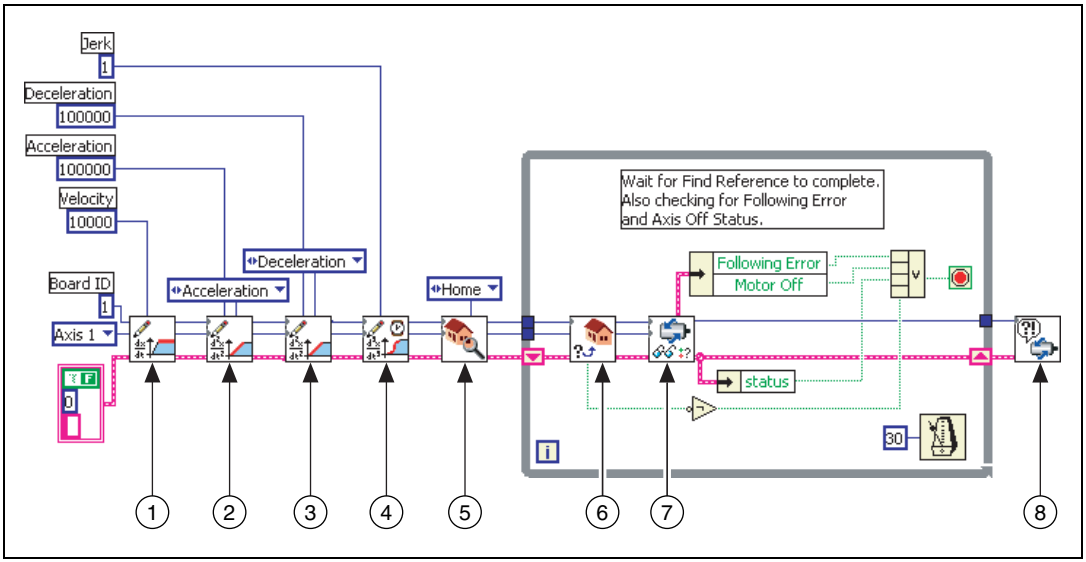


Figure 8-1. Find Reference Move Algorithm

LabVIEW Code



1	Load Velocity	4	Load S-Curve Time	7	Read per Axis Status
2	Load Acceleration/Deceleration	5	Find Reference	8	Check Reference
3	Load Acceleration/Deceleration	6	Check Reference	9	Motion Error Handler

Figure 8-2. Find Reference Move in LabVIEW

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
    f64 acceleration=100;// Acceleration value in RPS/s
    f64 velocity=200;// Velocity value in RPM
    u16 found, finding;// Check reference statuses
    u16 axisStatus;// Axis status
    u16 csr = 0;// Communication status register
    i32 position;// Current position of axis
    i32 scanVar;// Scan variable to read in values not
    supported by
    // the scanf function
```

```
//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("%d", &scanVar);
boardID=(u8)scanVar;

//Check if the device is at power up reset condition
err = flex_read_csr_rtn(boardID, &csr);
CheckError;

if (csr & NIMC_POWER_UP_RESET ){
   printf("\nThe FlexMotion device is in the reset
   condition. Please initialize the device before
   ");
   printf("running this example.  The
   \"flex_initialize_controller\" function will
   initialize the ");
   printf("board with settings selected through
   Measurement & Automation Explorer.\n");
   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, 0xFF);
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_REFERENCE);
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 because of axis
            off or following //error
            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);//Stop the Motion
                  err = csr & NIMC_MODAL_ERROR_MSG;
                  CheckError;
            }

            //test for find reference complete, following
            error, or axis //off status

        }while (!(axisStatus & (NIMC_FOLLOWING_ERROR_BIT |
        NIMC_AXIS_OFF_BIT)) && finding);

        printf("\nAxis %d position: %10d", axis, position);

        if (found)
            printf("\rAxis found reference");
        else
            printf("\rAxis did not find reference");

        printf("\n\nFinished\n");

    return;// Exit the Application
```

```
////////////////////////////////////////////////////
/////////////
// Error Handling
//
nimcHandleError;

// Check to see if there were any Modal Errors
if (csr & NIMC_MODAL_ERROR_MSG){
   do{
      //Get the command ID, resource ID, and the error
      code of the modal
      //error from the error stack on the device
      flex_read_error_msg_rtn(boardID,&commandID,&reso
      urceID, &errorCode);
      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
}
```

9

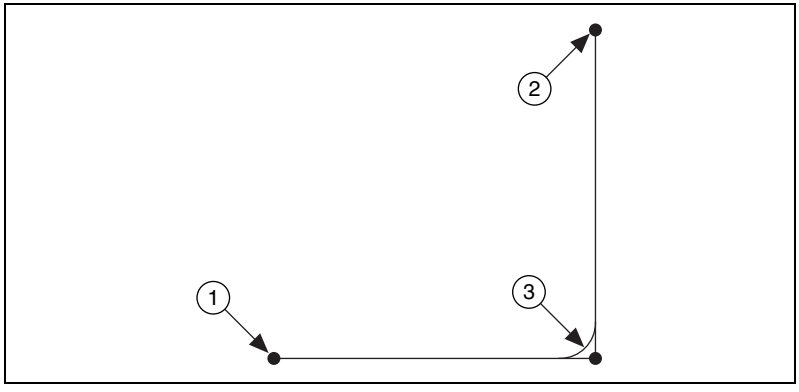
Blending Moves

Use blending moves to create continuous motion between two or more move segments.

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 90º 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 Figure 9-1.



1 Starting Position 2 End Point 3 Corner Rounded by Blending

Figure 9-1. Two Blended Straight-Line Moves

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.

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 the *Move Profiles* section of Chapter 4, *What You Need to Know about Moves*, 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 Figure 9-2.

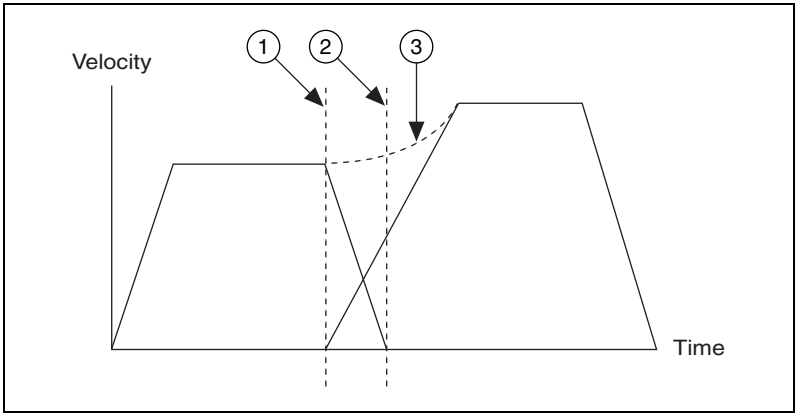
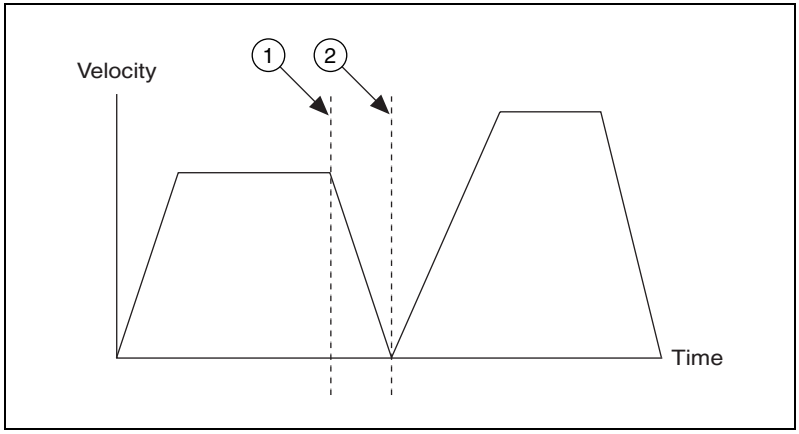


Figure 9-2. Superimposing Two Moves

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 Figure 9-3.



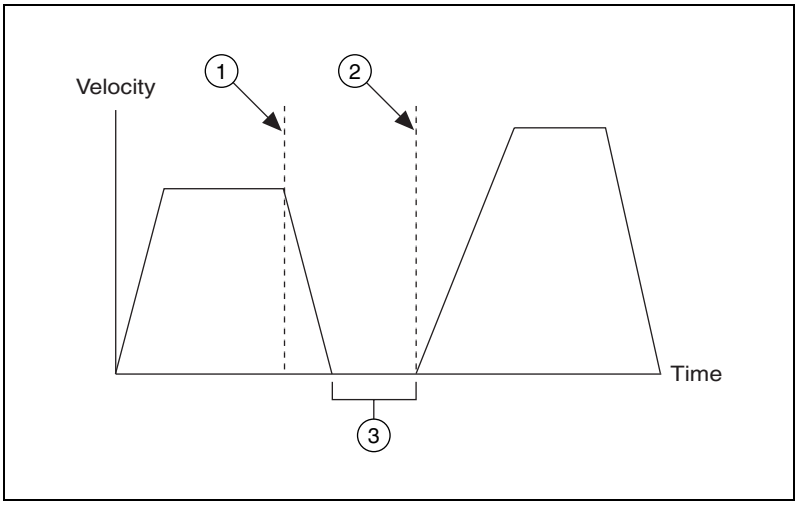
1 Blending Starts 2 Blend is Complete

Figure 9-3. Blending after Move Complete

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 Figure 9-4.



1 Blending Starts 2 Blend is Complete 3 User-Defined Delay

Figure 9-4. Blending after a Delay

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.

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

Figure 9-5 illustrates a generic algorithm for blending moves.

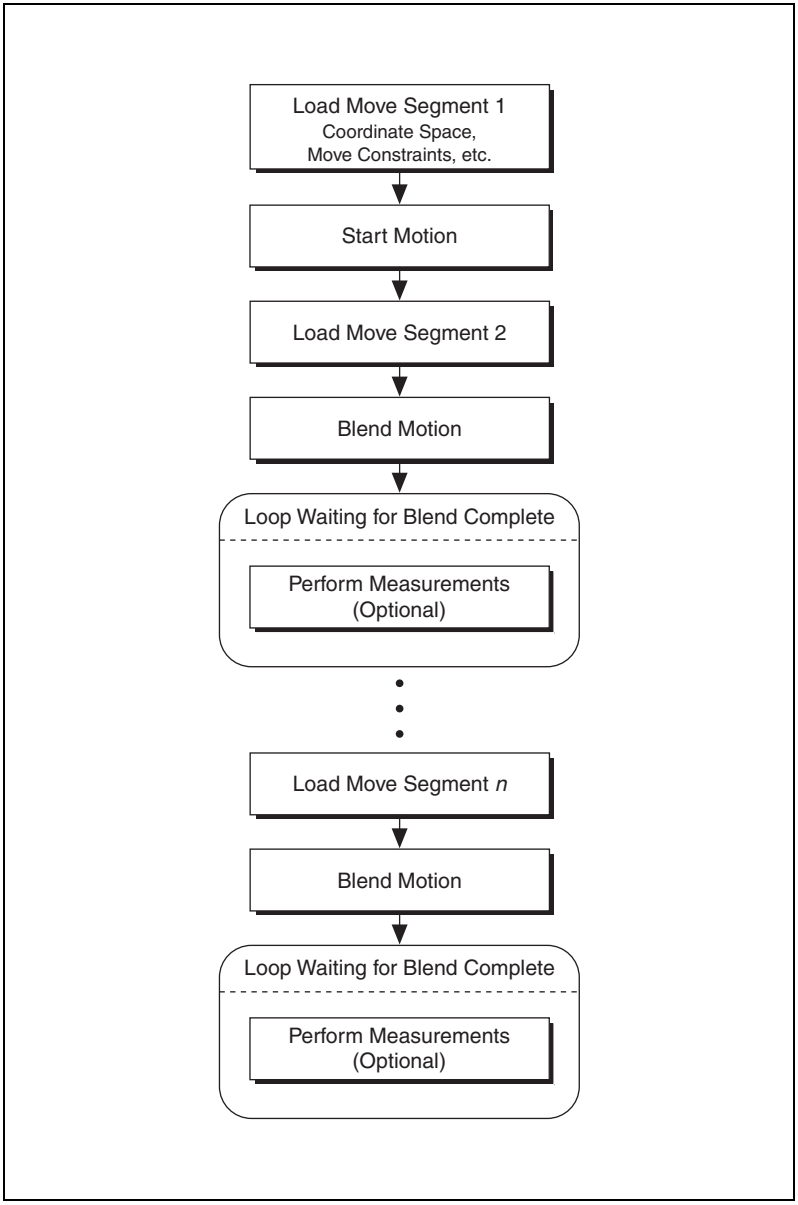
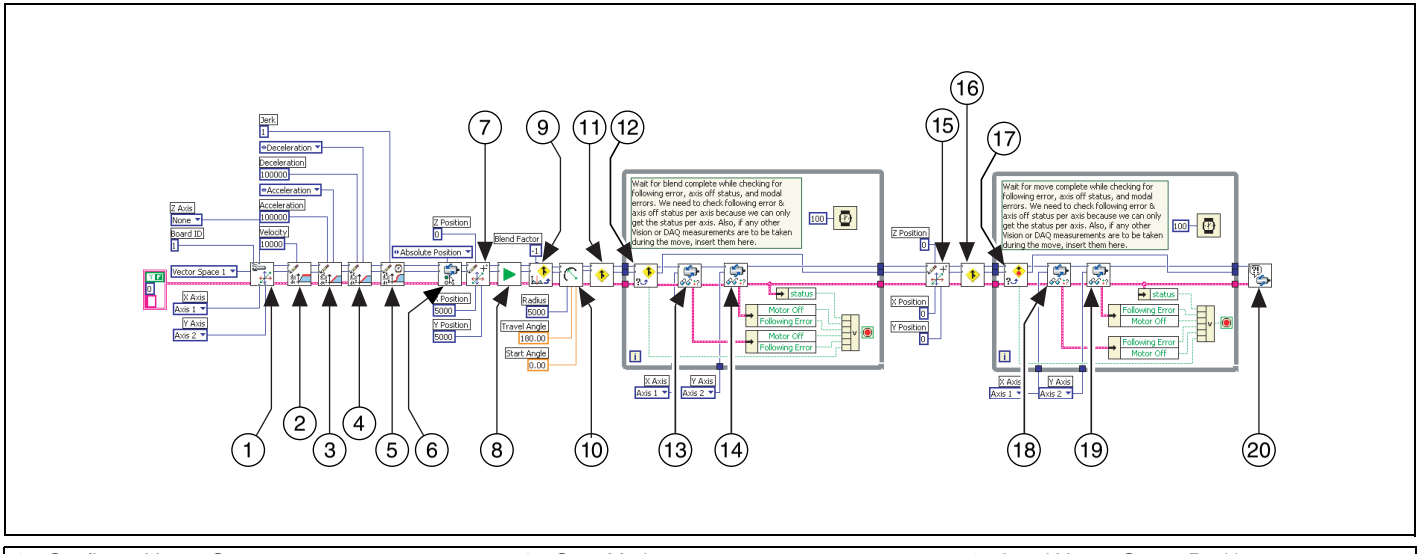


Figure 9-5. Blending Algorithm

LabVIEW Code



1 Configure Vector Space	8 Start Motion	15 Load Vector Space Position
2 Load Velocity	9 Load Blend Factor	16 Blend Motion
3 Load Acceleration/Deceleration	10 Load Circular Arc	17 Check Move Complete Status
4 Load Acceleration/Deceleration	11 Blend Motion	18 Read per Axis Status
5 Load S-Curve Time	12 Check Blend Complete Status	19 Read per Axis Status
6 Set Operation Mode	13 Read per Axis Status	20 Motion Error Handler
7 Load Vector Space Position	14 Read per Axis Status	

Figure 9-6. Blended Straight-Line Move and Arc Move in LabVIEW

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 vector space number
    vectorSpace = NIMC_VECTOR_SPACE1;
    ////////////////////////////////

    // Configure a 2D coordinate space comprised of axes 1,
    and 2
    err = flex_config_vect_spc(boardID, vectorSpace,
    NIMC_AXIS1, NIMC_AXIS2, NIMC_NOAXIS);
    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_ACCELERATION, 100000, 0xFF);
    CheckError;

    // Set the deceleration for the move (in counts/sec^2)
    err = flex_load_acceleration(boardID, vectorSpace,
    NIMC_DECELERATION, 100000, 0xFF);
    CheckError;

    // Set the jerk or s-curve 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_POSITION);
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 counter-clockwise
semi-circle
err = flex_load_circular_arc (boardID, vectorSpace,
5000/*radius*/, 0.0/*startAngle*/,
180.0/*travelAngle*/, 0xFF);
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, 0, &complete);
    CheckError;
    // Check the following error/axis off status for axis
    1
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS1, &status);
    CheckError;
    axisStatus |= status;
    // Check the following error/axis off status for axis
    2
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS2, &status);
    CheckError;
    axisStatus |= status;
    //Read the communication status register and check
    the modal //errors
    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) && !(axisStatus &
NIMC_AXIS_OFF_BIT)); //Exit on move
//complete/following error/axis off
// 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 because this is the final
segment
do
{
    axisStatus = 0;
    // Check the move complete status
    err = flex_check_move_complete_status(boardID,
    vectorSpace, 0, &complete);
    CheckError;
    // Check the following error/axis off status for axis
    1
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS1, &status);
    CheckError;
    axisStatus |= status;
    // Check the following error/axis off status for axis
    2
    err = flex_read_axis_status_rtn(boardID,
    NIMC_AXIS2, &status);
    CheckError;
    axisStatus |= status;
    //Read the communication status register and check
    the modal //errors
    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) && !(axisStatus &
```

```
    NIMC_AXIS_OFF_BIT));
    //Exit on move complete/following error/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 ID, and the
                error code of the //modal error from the error
                stack on the device
                flex_read_error_msg_rtn(boardID,&commandID,
                &resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,resou
                rceID);
                //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
}
```

10

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.

Algorithm

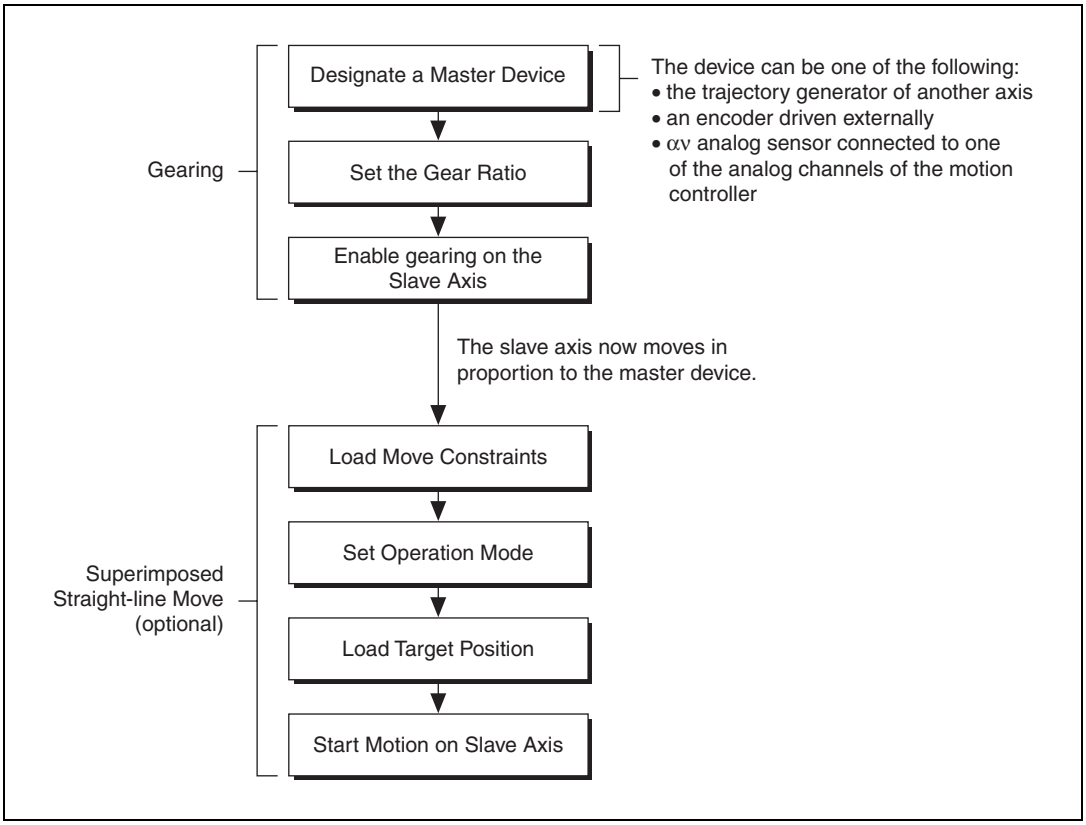


Figure 10-1. Electronic Gearing Algorithm

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.

$$\text{Slave axis move} = \text{Master axis position} \times \text{Gear ratio}$$

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.

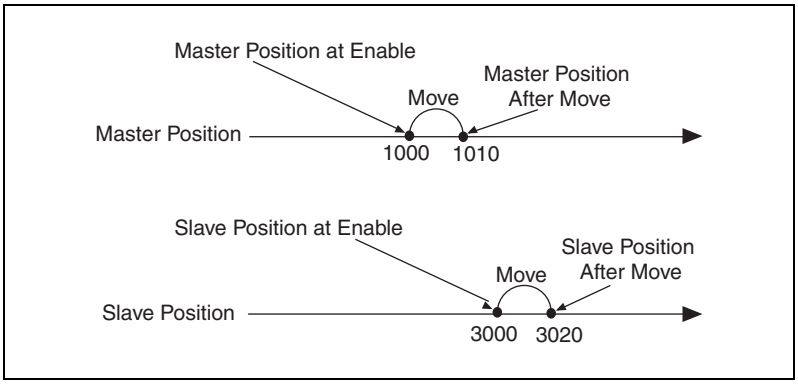


Figure 10-2. Relative Gearing at Enable

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.

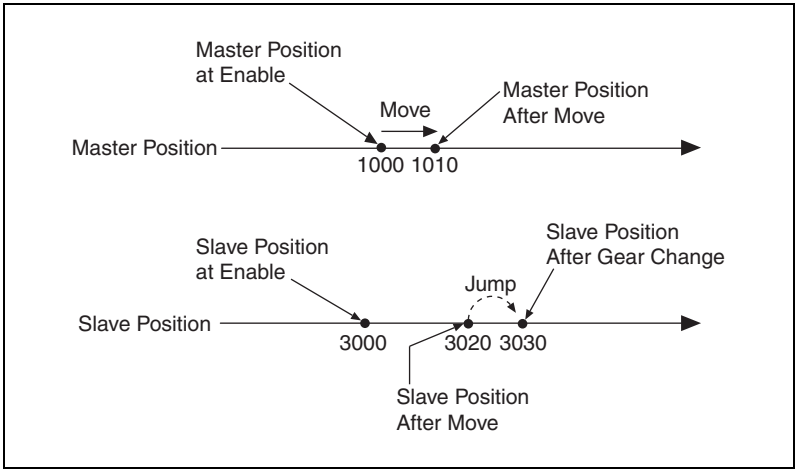


Figure 10-3. Absolute Gearing at Gear Ratio Change

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.

Gear Master

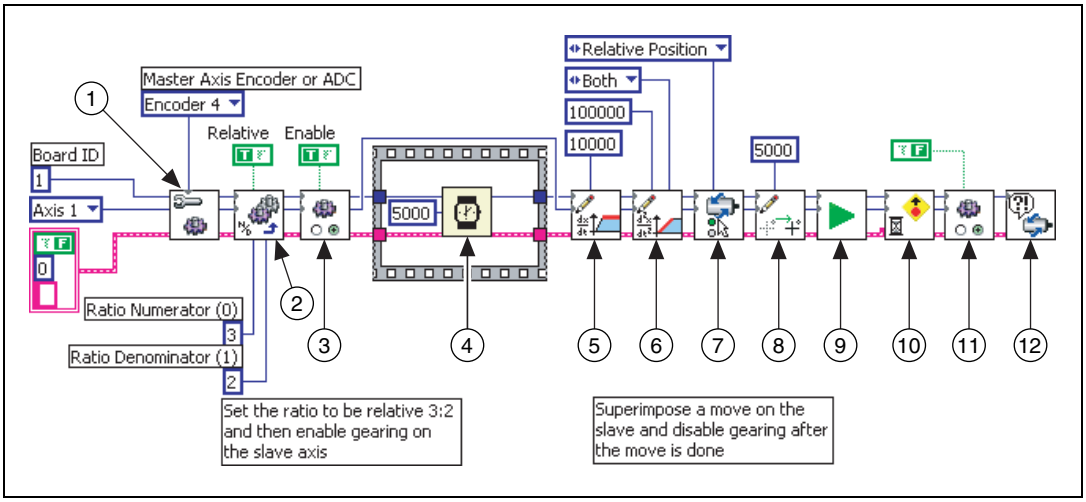
An axis can be geared to another axis, or to an encoder or ADC.

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.

LabVIEW Code



1	Configure Gear Master	7	Set Operation Mode
2	Load Gear Ratio	8	Load Target Position
3	Enable Gearing	9	Start Motion
4	Wait	10	Wait for Move Complete
5	Load Velocity	11	Enable Gearing Single Axis
6	Load Acceleration/Deceleration	12	Read per Axis Status

Figure 10-4. Tracking an Encoder Using Electronic Gearing with Superimposed 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 slaveAxis; // Slave axis number
    u8 master; // Gear master
    u16 csr = 0; // Communication status register
    u16 moveComplete;

    //Variables for modal error handling
    u16 commandID;// The command ID of the function
    u16 resourceID;// The resource ID of the function
    i32 errorCode;// Error code

    ////////////////////////////////
```

```
// Set the board ID
boardID = 1;
// Set the axis number
slaveAxis = NIMC_AXIS1;
// Master is encoder 4
master = NIMC_ENCODER4;
////////////////////////////////

// Set up the gearing configuration for the slave
axis
err = flex_config_gear_master(boardID, slaveAxis,
master);
CheckError;

//Load Gear Ratio 3:2
err =  flex_load_gear_ratio(boardID, slaveAxis,
NIMC_RELATIVE_GEARING, 3/* ratioNumerator*/, 2/*
ratioDenominator*/, 0xFF);
CheckError;

//-------------------------------------------------
// Enable gearing on slave axis
//-------------------------------------------------
err = flex_enable_gearing_single_axis (boardID,
slaveAxis, NIMC_TRUE);
CheckError;

// Wait for 5,000 ms (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_POSITION);
CheckError;

// Load velocity in counts/s
err = flex_load_velocity(boardID, slaveAxis, 10000,
0xFF);
CheckError;

// Load acceleration and deceleration in counts/s^2
err = flex_load_acceleration(boardID, slaveAxis,
NIMC_BOTH, 100000, 0xFF);
CheckError;
```

```
    // Load the target position for the registration
    (superimposed) //move
    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 pollInterval*/, &moveComplete);
    CheckError;

    //-------------------------------------------------
    // Disable gearing on slave axis
    //-------------------------------------------------
    err = flex_enable_gearing_single_axis (boardID,
    slaveAxis, NIMC_FALSE);
    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 ID, and the
              error code of the //modal error from the
              error stack on the device
              flex_read_error_msg_rtn(boardID,&commandI
              D,&resourceID, &errorCode);
              nimcDisplayError(errorCode,commandID,res
              ourceID);

              //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
}
```

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 in Figure 10-5.

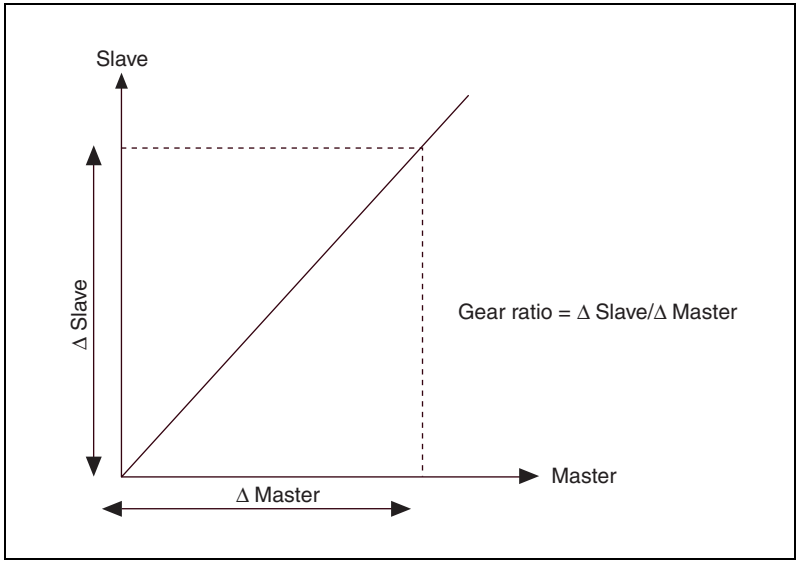


Figure 10-5. Master/Slave Ratio in Gearing

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 Figure 10-6. Camming is used in applications where the slave axis follows a non-linear profile from a master device.

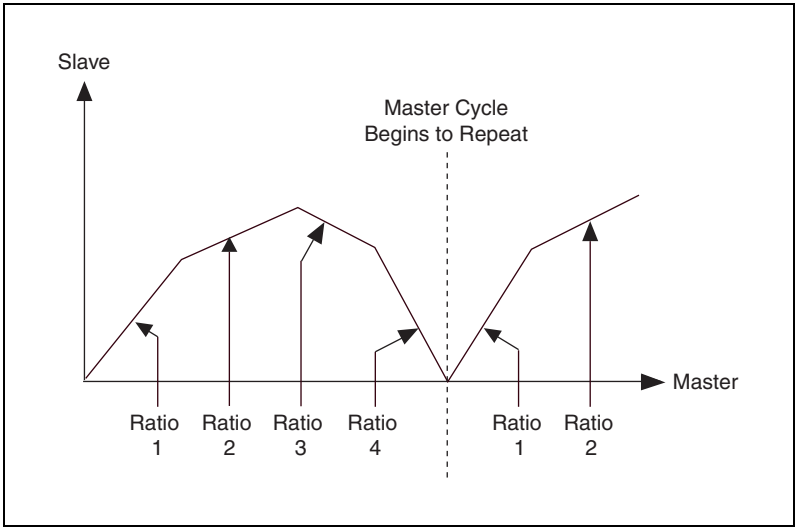
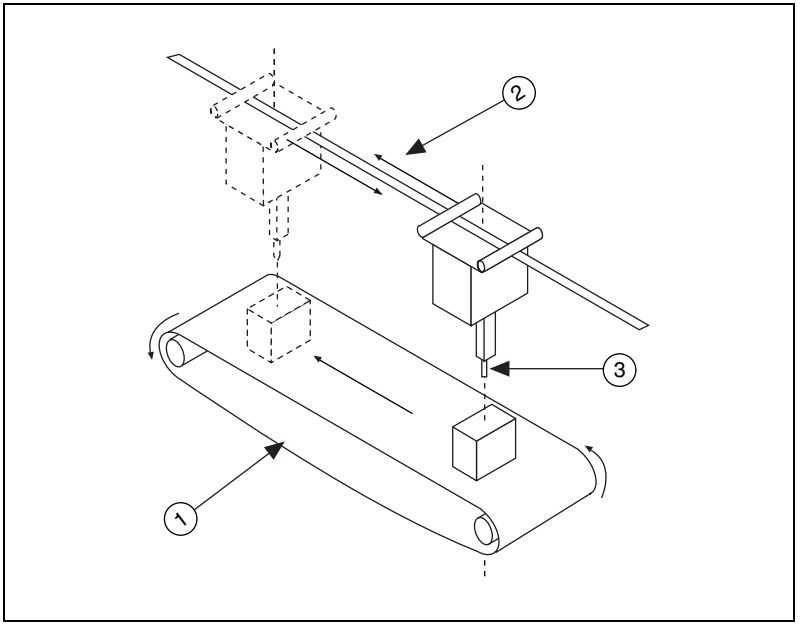


Figure 10-6. Multiple Camming Gear Ratios

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.

Figure 10-7 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

Figure 10-7. Welding Application

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:

1. 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 which direction the welding point is moving in 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.

Algorithm

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.

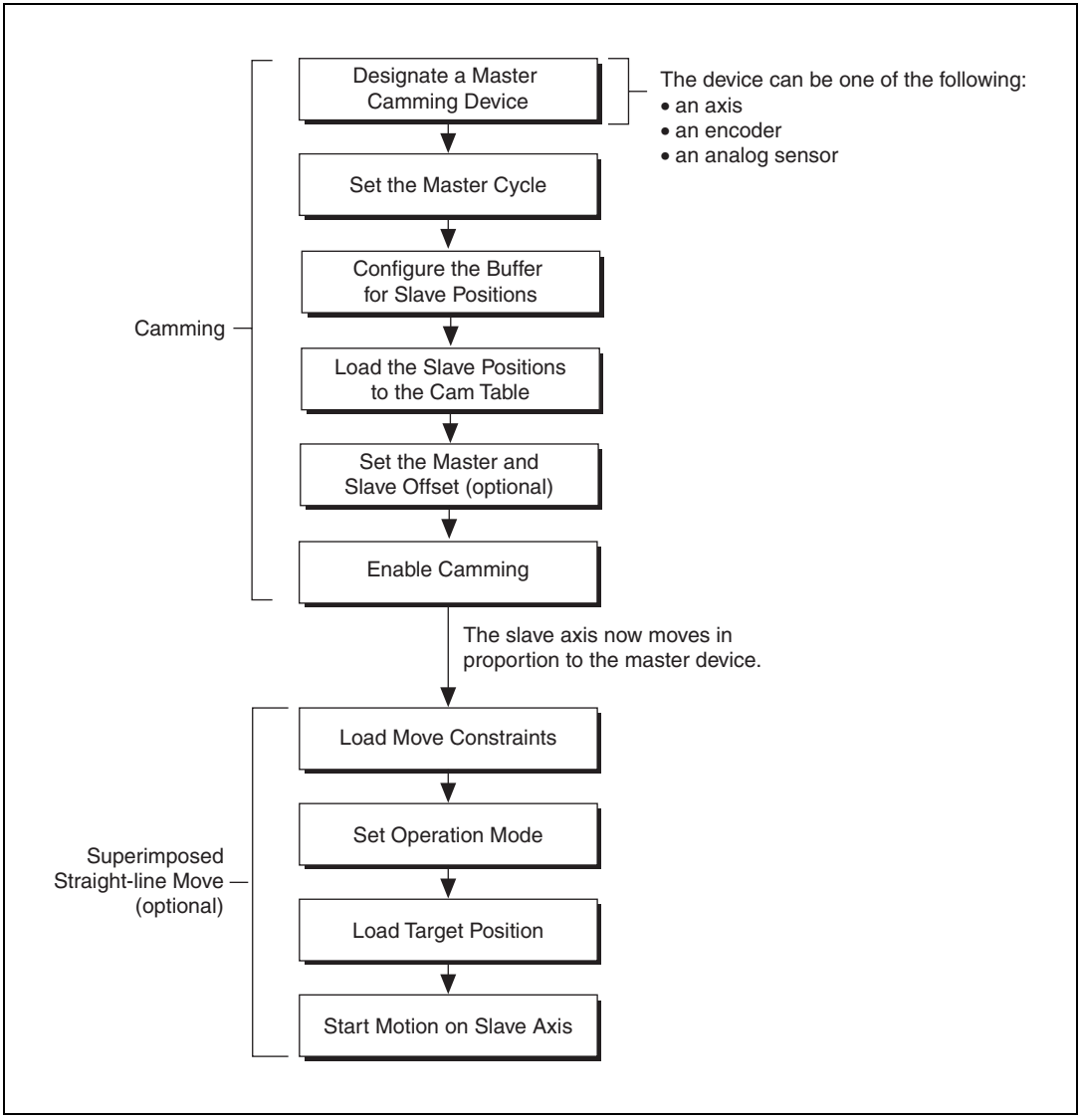


Figure 10-8. Camming Algorithm

Camming Table

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. Refer to Table 10-1 for an example of a camming table.

Figure 10-9 shows that, in the welding example defined in Figure 10-7, 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.

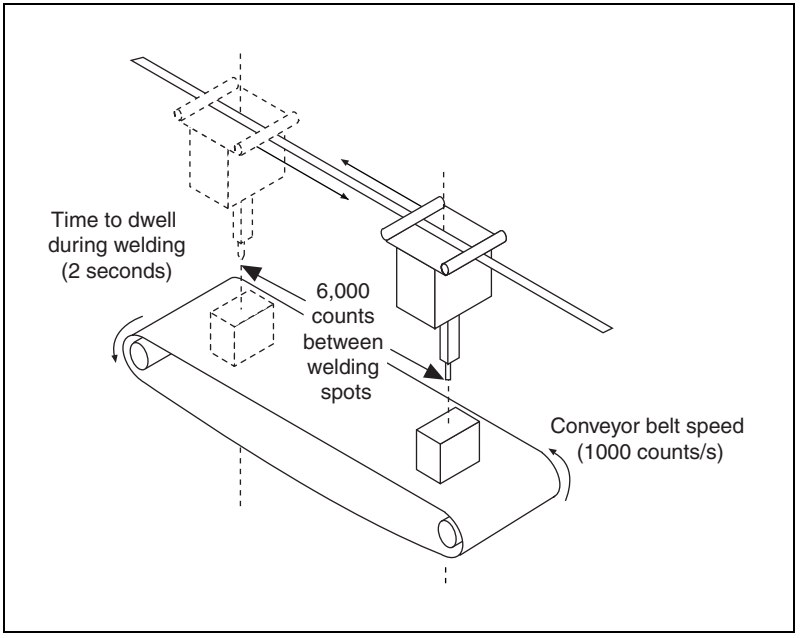


Figure 10-9. Welding Application Time and Speed Constraints

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. Figure 10-10 shows the move profile of the first and second move segments.

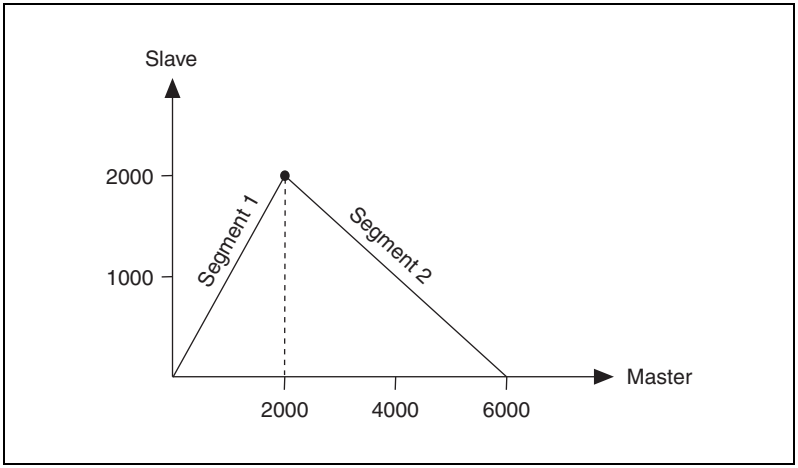


Figure 10-10. First and Second Move Segment Profile

Table 10-1 shows the camming table that corresponds to the move profile in Figure 10-10

Table 10-1. Welding Application Cam Table

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 = ΔSlaveDistance / ΔMasterDistance			

Table 10-1 shows that 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.

Figure 10-11 shows that the master position in this interval is inside the modulus.

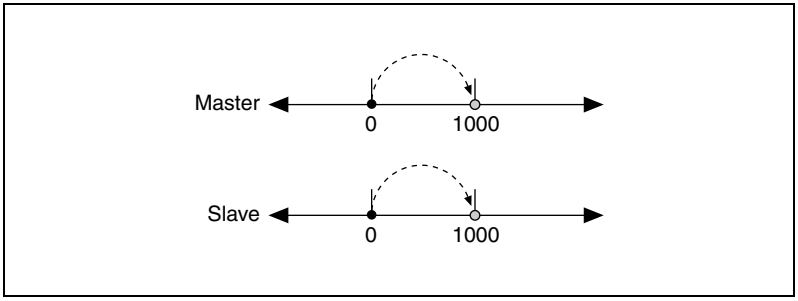


Figure 10-11. Master and Slave Positions at Enable

Figure 10-12 shows that 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.

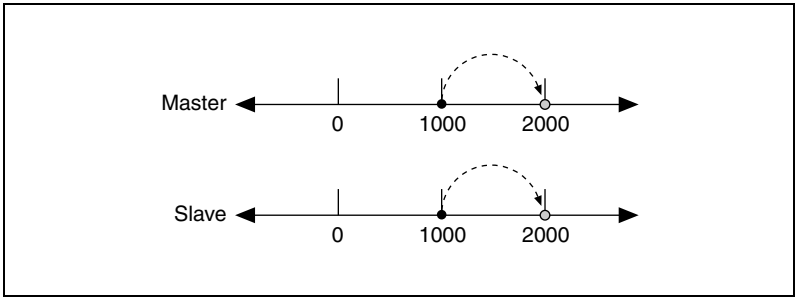


Figure 10-12. Master Axis Moves within Interval

Figure 10-13 shows that 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.

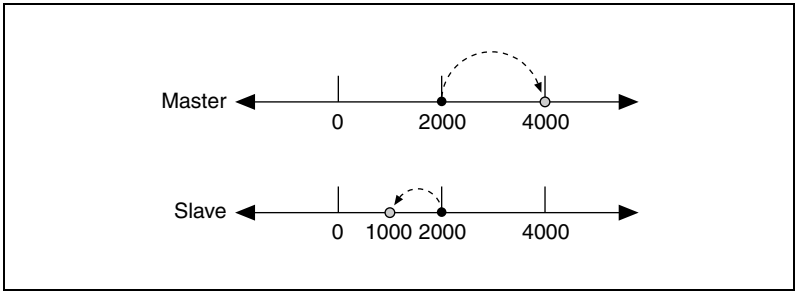


Figure 10-13. Gear Ratio Change

Figure 10-14 shows that when the master reaches position 6000, the slave axis moves back the original position, and the camming cycle begins again.

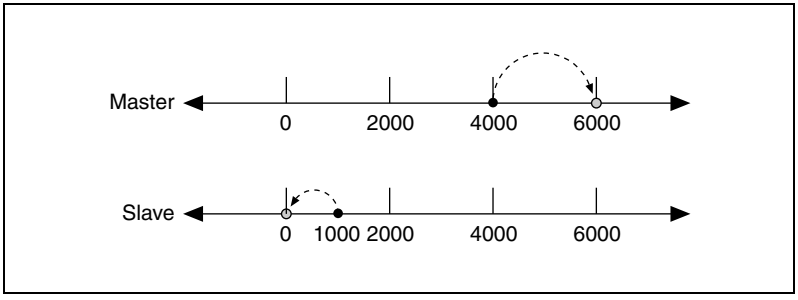


Figure 10-14. Camming Cycle Repeats

Slave Offset

In some camming applications, the slave axis might begin and end the camming cycle at different positions, as shown in Table 10-2.

Table 10-2. Camming Profile with and without Slave Offset

Master Position (counts)	Slave Position (counts)
0	0
2000	2000
4000	1000
6000	500

Figure 10-15 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.

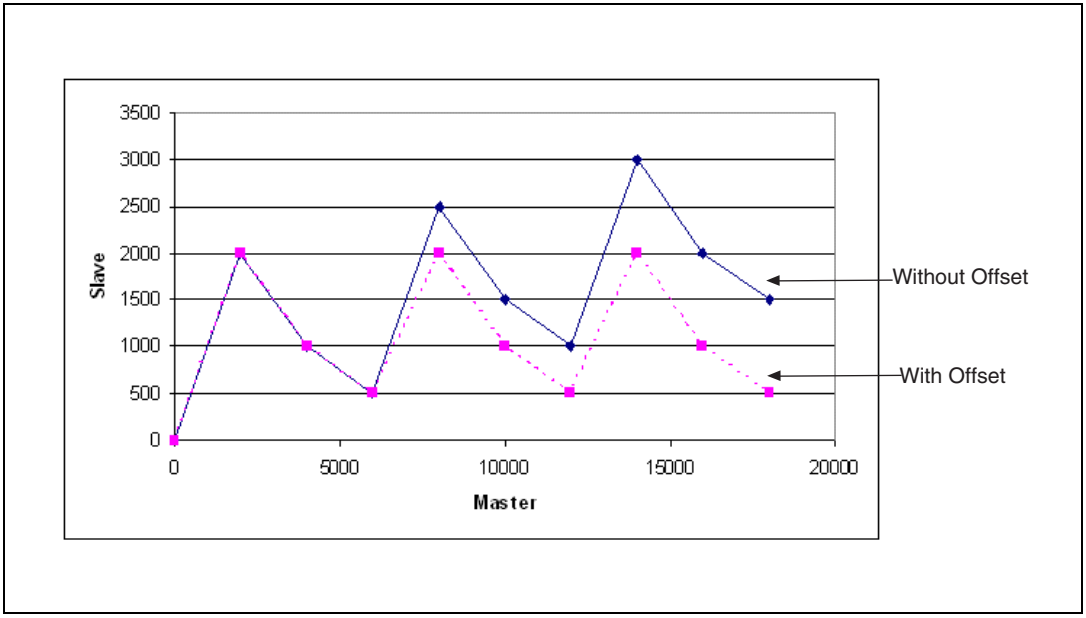


Figure 10-15. Camming without Offset

Master Offset

If the material and welding point are not initially aligned, as shown in Figure 10-16, the master offset must be applied to consider the position difference.

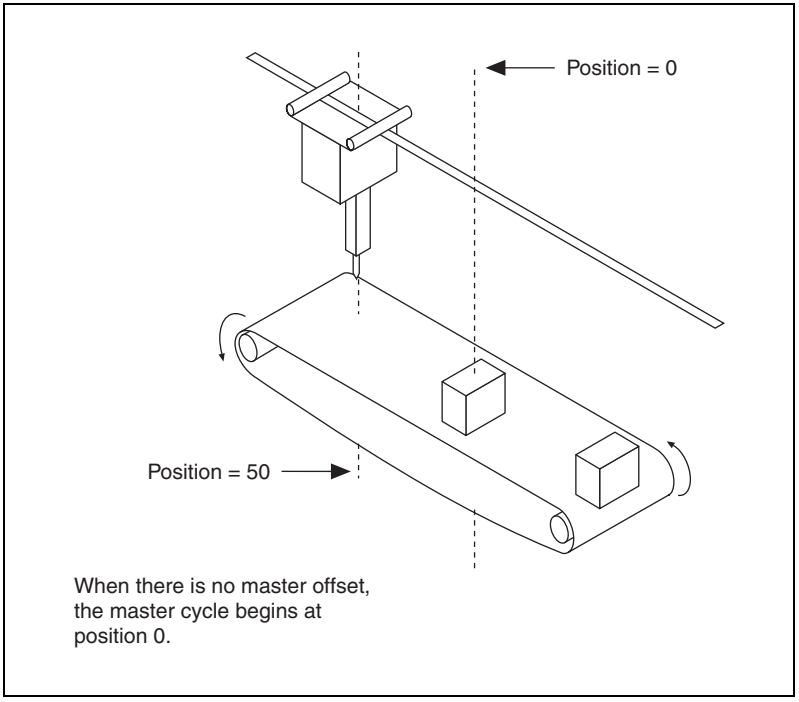


Figure 10-16. Misaligned Material and Welding Point

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

Figure 10-16 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.

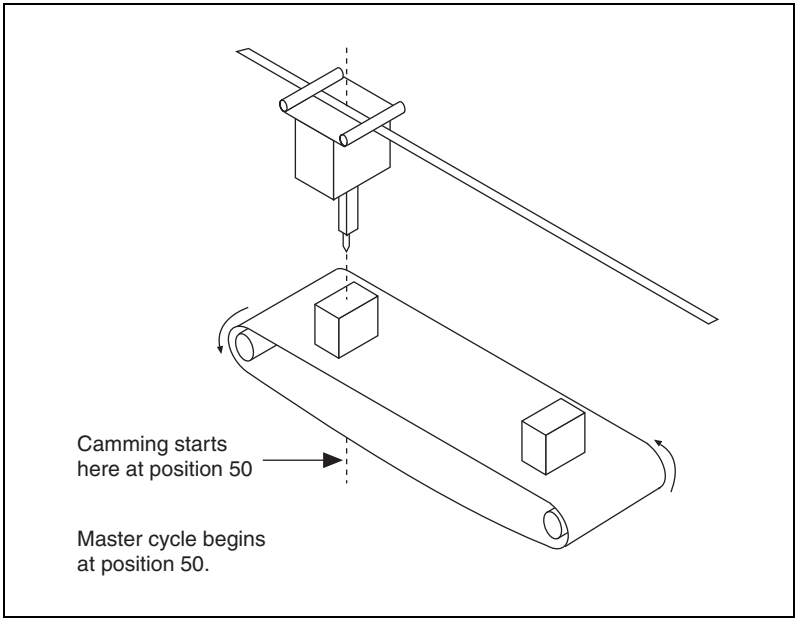


Figure 10-17. Camming Profile Starts when First Material Passes

Figure 10-18 shows the camming profile used for the application portrayed in Figure 10-16 and Figure 10-17.

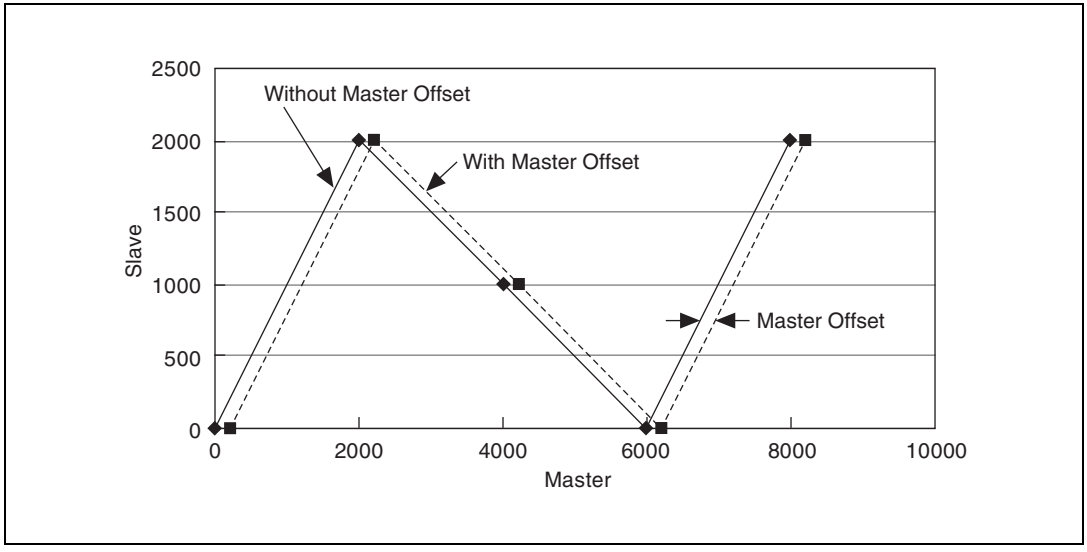
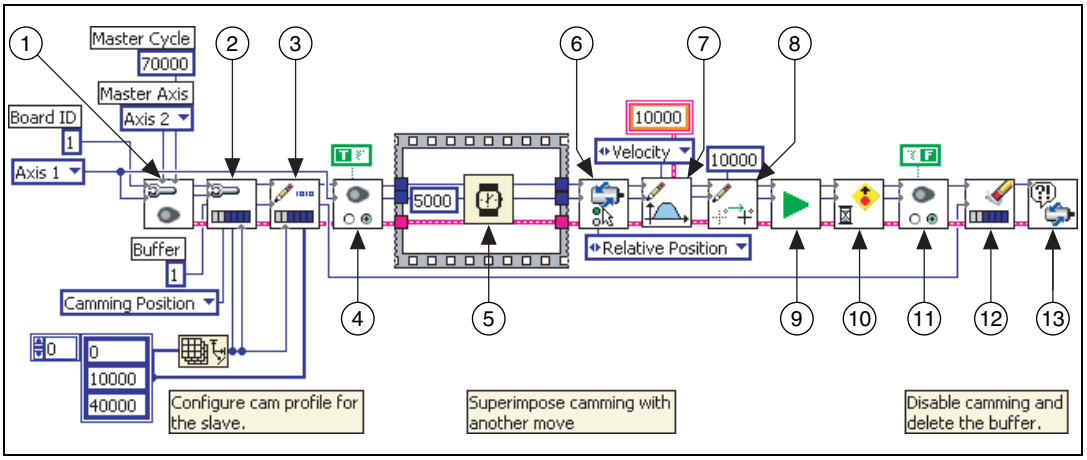


Figure 10-18. Camming Profiles with and without Master Offset

LabVIEW Code



1 Configure Camming Master	8 Load Target Position
2 Configure Buffer	9 Start Motion
3 Write Buffer	10 Wait for Move Complete
4 Enable Camming Single Axis	11 Enable Camming Single Axis
5 Wait in a Sequence Structure	12 Clear Buffer
6 Set Operation Mode	13 Motion Error Handler
7 Load Move Constraint	

Figure 10-19. Axis to Axis 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 = 1;         // Board ID as assigned by MAX
   f64 bufferInterval = 0; // Ignored

   // Master axis information
   u8 masterAxis = 2;      // Master axis ID
   f64 camCycle = 70000;   // Position cycle to repeat
the camming process

   // Slave axis information
```

```
   u8 slaveAxis = 1;            // Slave axis ID
   u8 buffer = 1;              // Buffer to contain the
cam table
   i32 positionArr[] = {0, 10000, 40000, 45000, 45000,
40000, 10000, 0}; // Position array
  u32 positionSize = sizeof(positionArr) / sizeof(i32);
// Number of positions in the array
   NIMC_DATA data;          // Generic data structure
   i32 targetPos = 10000;  // Position to move to
  f64 velocity = 10000;   // Velocity limit for this move
   u16 moveComplete;

   // For error handling
   u16 csr;                 // Communication status
   u16 commandID;          // Command ID that causes the
error
   u16 resourceID;         // Resource ID that is set on
the failed command
   i32 errorCode;           // Error code from the
controller

// Configuring camming profile.
// Configure the cam master & master cycle
   err = flex_configure_camming_master(boardID,
slaveAxis, masterAxis, camCycle);
   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, -1.0);
   CheckError;

   // At this point camming is engaged or started. You
can start the master
   // axis or insert other functionality here.
   Sleep(5000);
```

```
// Configure the superimposed move (optional)
// Set to absolute mode
   err = flex_set_op_mode(boardID, slaveAxis,
NIMC_ABSOLUTE_POSITION);
   CheckError;

   // Set the maximum velocity
   data.doubleData = velocity;
   err = flex_load_move_constraint(boardID, slaveAxis,
TnimcMoveConstraintVelocity, &data);
   CheckError;

   // Set the target position
   err = flex_load_target_pos(boardID, slaveAxis,
targetPos, 0xFF);
   CheckError;

   // Start the master axis movement
   err = flex_start(boardID, slaveAxis, 0x0);
   CheckError;

   // Wait for move to complete
   err = flex_wait_for_move_complete (boardID,
slaveAxis, 0x1, 20000, 20, &moveComplete);
   CheckError;

// 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);

// Check to see if there were any Modal Errors
   flex_read_csr_rtn(boardID, &csr);
if (csr & NIMC_MODAL_ERROR_MSG)
   {
do
      {
// Get the command ID, resource and the error code of the
modal
//error from the error stack on the board
```

```
flex_read_error_msg_rtn(boardID, &commandID,
&resourceID, &errorCode);
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
}
```

11

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 Figure 11-1.

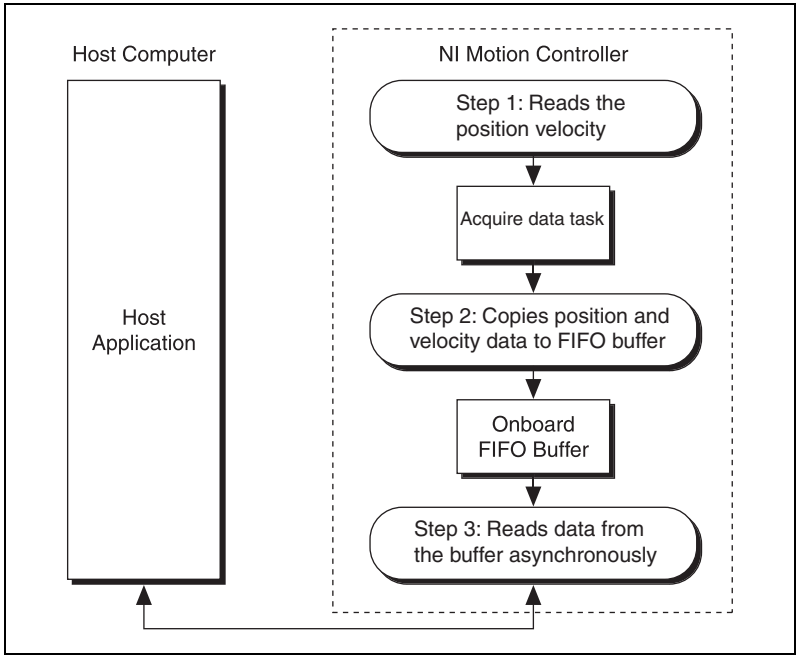


Figure 11-1. Acquire Data Path

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.

Algorithm

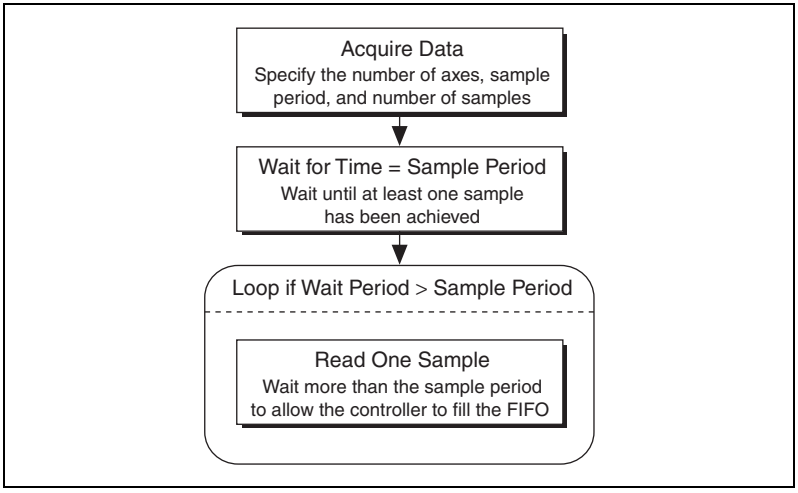


Figure 11-2. Acquire Data Algorithm

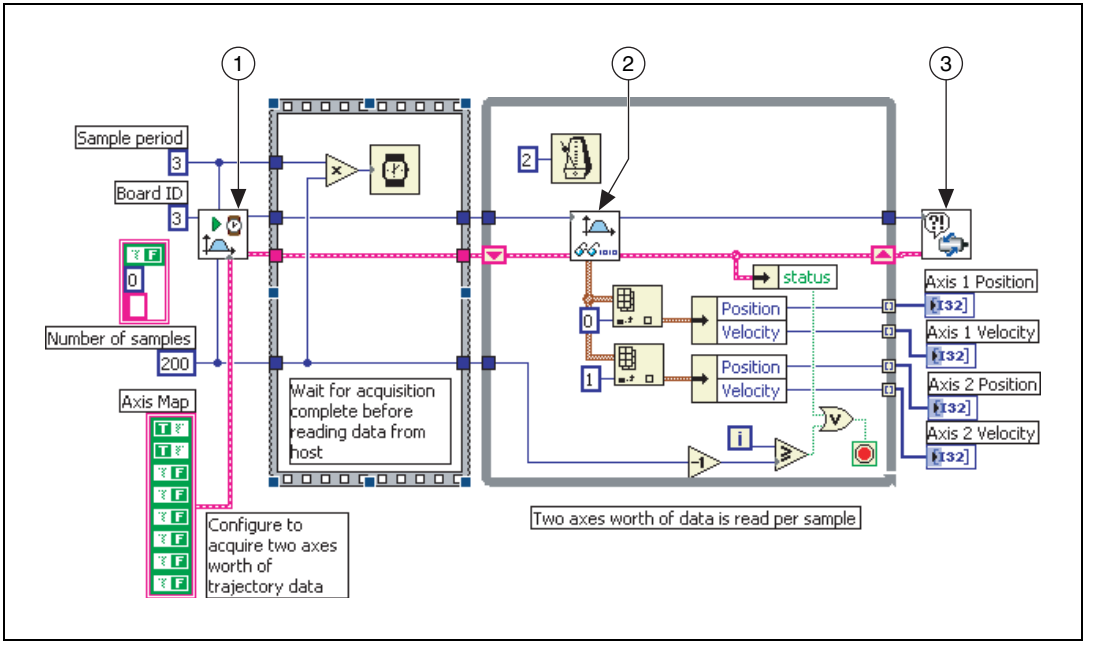
The data must be read one sample at a time. A four-axis sample uses the following pattern for returning the data.

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

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

LabVIEW Code

Figure 11-3 acquires data for two axes, 200 samples, and three milliseconds apart.



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

Figure 11-3. Acquire Data Using LabVIEW

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
    u16 csr = 0;// Communication status register
    i32 i;
    u16 axisMap;// Bitmap of axes for which data is
    requested
```

```
i32 axis1Positions[200];// Array to store the
positions (1)
i32 axis1Velocities[200];// Array to store
velocities(1)
i32 axis2Positions[200];// Array to store the
positions (2)
i32 axis2Velocities[200];// Array to store
velocities(2)
u16 numSamples = 200;// Number of samples
i32 returnData[4];// Need size of 4 for 2 axes worth
of data

//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;
// 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, 3/* ms time period*/);
CheckError;
Sleep(numSamples * 3/* ms time period*/);

for(i=0; i<numSamples; i++){
    Sleep (2);
    // 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 ID, and the
                error code of the //modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);

                //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
}
```

12

Synchronization

You can synchronize NI motion controllers with NI data and image acquisition devices using breakpoints and high-speed captures.

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 *Absolute Breakpoints*, *Relative Position Breakpoints*, and *Periodically Occurring Breakpoints*.

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 *High-Speed Capture* or *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.

Table 12-1 shows the availability of breakpoint modes on each NI motion controller.

Table 12-1. Breakpoint Modes on NI Motion Controllers

Breakpoint Mode	NI 7350	NI 7340, NI 7330, and NI 7390
Absolute*	Y	Y
Relative*	Y	Y
Periodic	Y	N
Modulus	N	Y
Buffered	Y	N
* Available in buffered and single operation for NI 7350 and in single operation only for all other controllers		

Note If you are using a data or image acquisition device with your motion control system, be aware that the NI SoftMotion Controller does not support the RTSI bus.

Note Breakpoints are not supported on the NI SoftMotion Controller.

Note When you are using the NI SoftMotion Controller with an Ormec device, you can use two high speed captures per axis.

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 buffered breakpoint and single position breakpoint methods.

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

Figure 12-1 shows the basic algorithm for implementing buffered breakpoints.

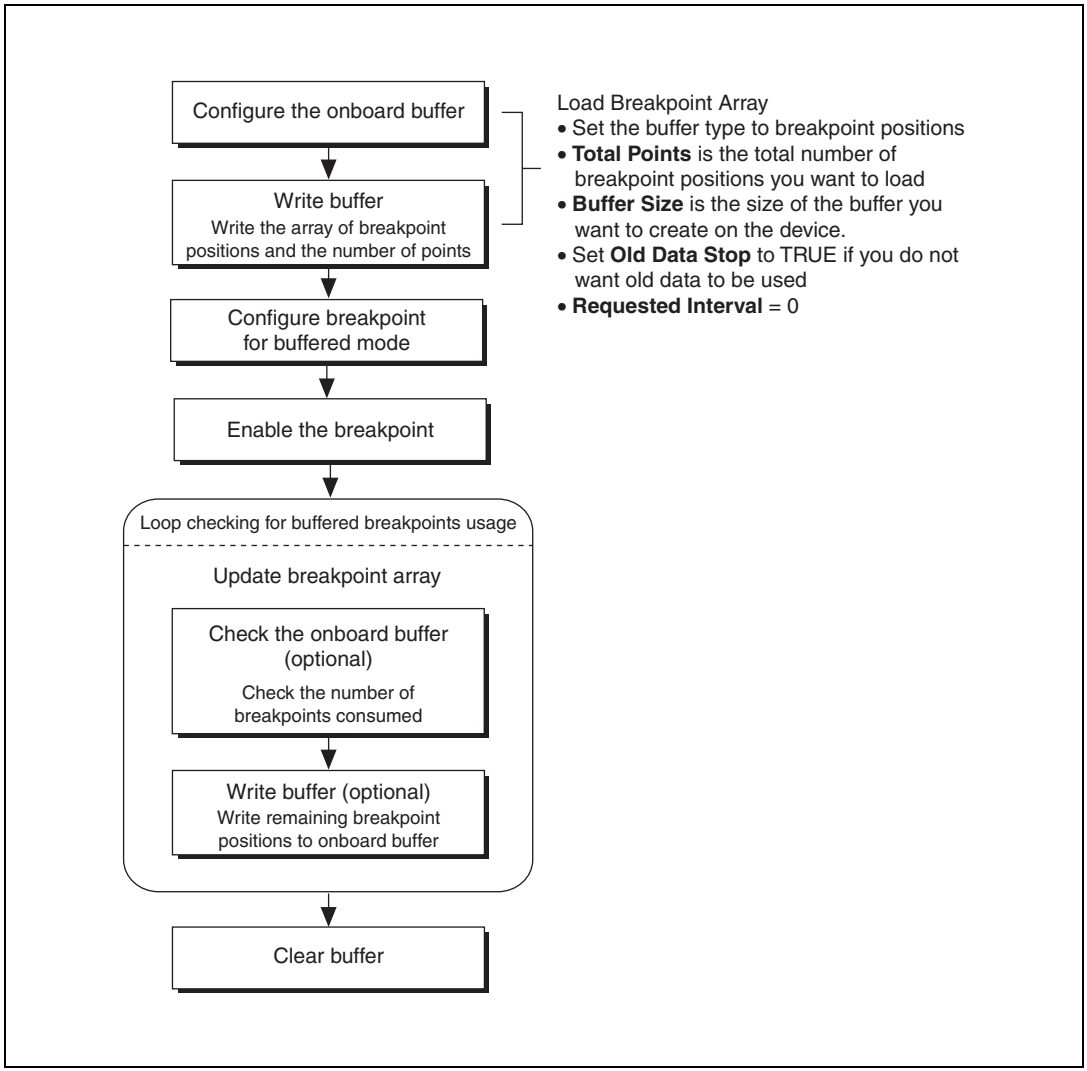
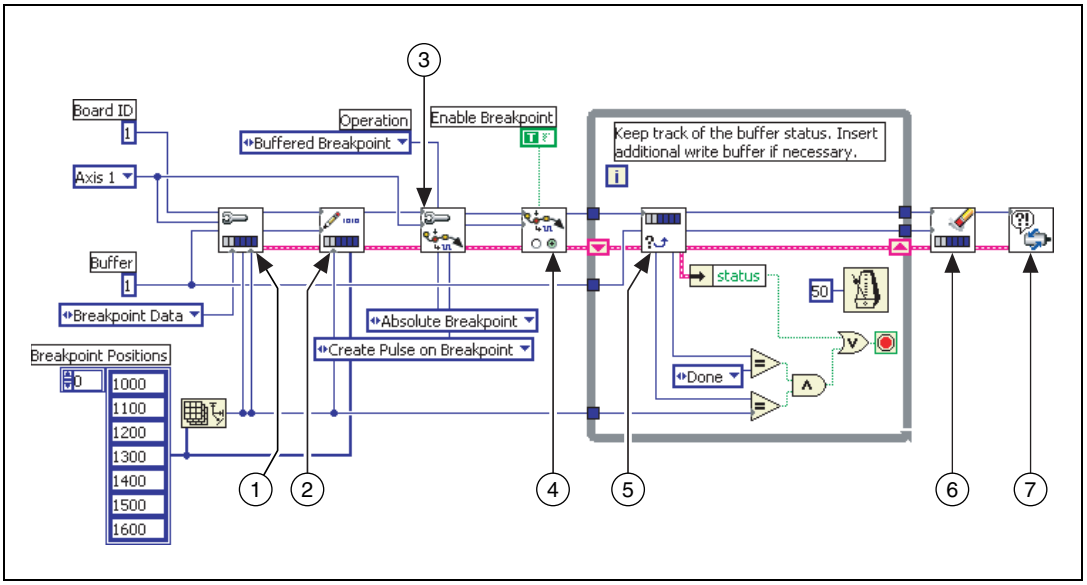


Figure 12-1. Buffered Breakpoint Algorithm

LabVIEW Code



1 Configure Buffer	4 Enable Breakpoint Output	6 Clear Buffer
2 Write Buffer	5 Check Buffer	7 Motion Error Handler
3 Configure Breakpoint		

Figure 12-2. Buffered Position Breakpoint in LabVIEW

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)
{

    // Resource variables
    u8boardID = 1;// Board identification number
    u8axis = NIMC_AXIS1;// Axis number
    u8 buffer = 1;// Buffer number

    // Modal error handling variables
    u16 commandID;// The commandID of the function
    u16 resourceID;// The resource ID
    i32 errorCode;// Error code
    u16 csr = 0;// Communication status
```

```
// Buffer resources
i32 breakpointPositions[] = {1000, 1100, 1200, 1300,
1400, 1500, 1600};
u16 numberOfPoints = 7;// Number of breakpoints
f64 actualInterval;// Required in the function call
but not being //used
f64 requestedInterval = 10.0;// Required in the
function call but //not being used
u32 backLog;// Number of space available in buffer
u16 bufferState;// Buffer state
u32 pointsDone;// Number of breakpoints done or
consumed

// Configure the buffer for buffered breakpoint
err = flex_configure_buffer(boardID, buffer,axis,
NIMC_BREAKPOINT_DATA, numberOfPoints,
numberOfPoints,NIMC_TRUE, requestedInterval,
&actualInterval);
CheckError;

// Write the breakpoint position to the buffer
err = flex_write_buffer(boardID, buffer,
numberOfPoints, NIMC_REGENERATION_NO_CHANGE,
breakpointPositions, 0xFF);
CheckError;

// Configure the breakpoint to be buffered breakpoint
err = flex_configure_breakpoint(boardID, axis,
NIMC_ABSOLUTE_BREAKPOINT, NIMC_PULSE_BREAKPOINT,
NIMC_OPERATION_BUFFERED);
CheckError;
// Enable the breakpoint
err = flex_enable_breakpoint(boardID, axis,
NIMC_TRUE);
CheckError;

// Poll the status of the buffer, if you have more
breakpoint //positions to write, insert
flex_write_buffer call here.
do
{
    // Check the buffer status
    err = flex_check_buffer_rtn(boardID, buffer,
    &backLog, &bufferState, &pointsDone);

    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 ID, and the
              error code of the //modal error from the
              error stack on the device
              flex_read_error_msg_rtn(boardID,&commandI
              D,&resourceID, &errorCode);

              nimcDisplayError(errorCode,commandID,res
              ourceID);

              //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

Figure 12-3 shows the basic algorithm for implementing single position breakpoints.

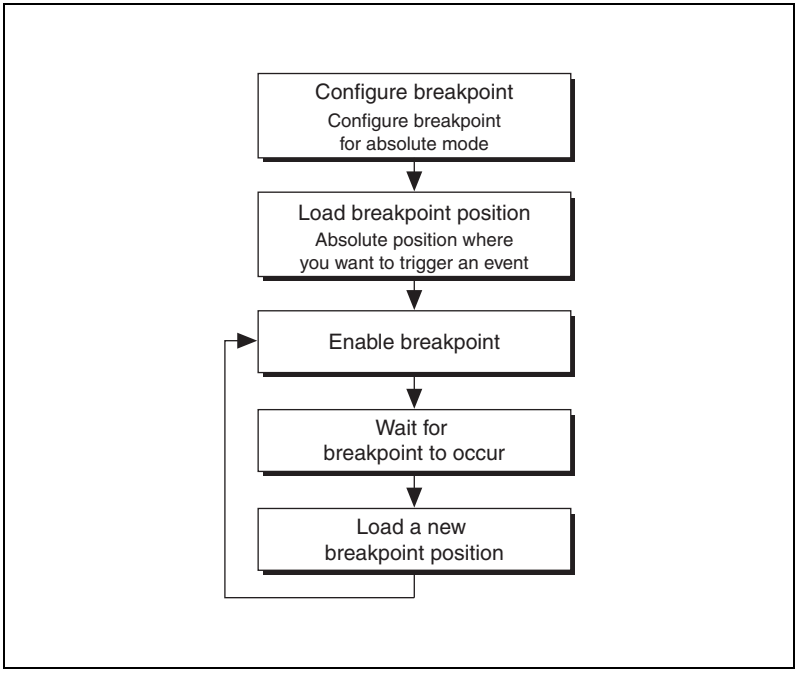
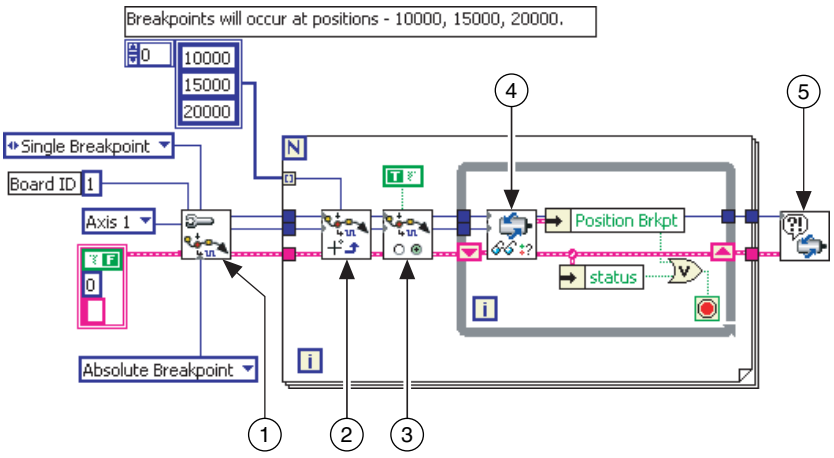


Figure 12-3. Single Position Breakpoint Algorithm

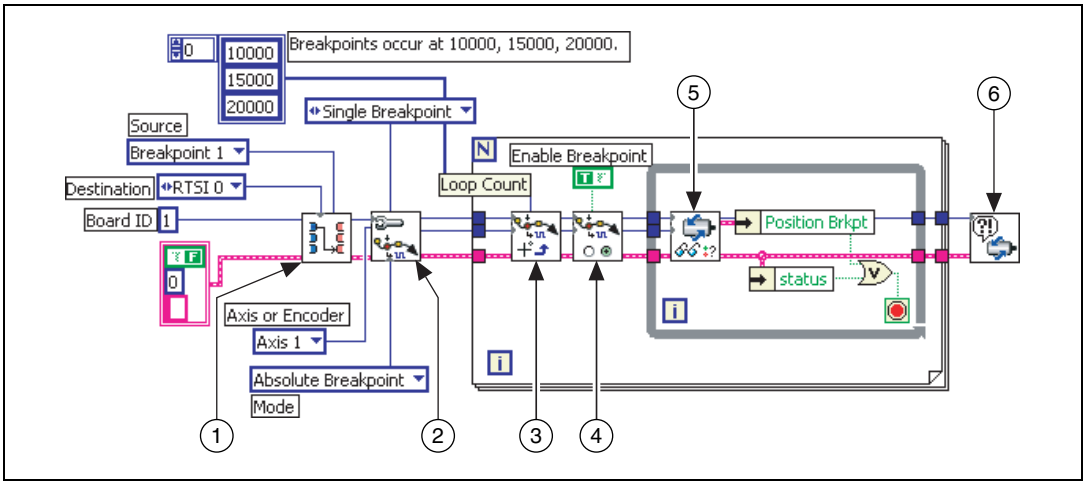
LabVIEW Code



1	Configure Breakpoint	3	Enable Breakpoint Output	5	Motion Error Handler
2	Load Breakpoint Position	4	Read per Axis Status		

Figure 12-4. Single Position Breakpoint in LabVIEW

Refer to Figure 12-5 for an example of how to route this breakpoint using RTSI.



1	Select Signal	3	Load Breakpoint Position	5	Read per Axis Status
2	Configure Breakpoint	4	Enable Breakpoint Output	6	Motion Error Handler

Figure 12-5. Single Position Breakpoint With RTSI Using LabVIEW

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.

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 = NIMC_AXIS1;
////////////////////////////////

// Route breakpoint 1 to RTSI line 1
err = flex_select_signal (boardID, NIMC_RTSI0
/*destination*/, NIMC_BREAKPOINT1/*source*/);
CheckError;

// Configure the breakpoint
err = flex_configure_breakpoint(boardID, axis,
NIMC_ABSOLUTE_BREAKPOINT /*mode*/,
NIMC_SET_BREAKPOINT /*action*/,
NIMC_OPERATION_SINGLE /*single operation*/);
CheckError;

for(i=0; i<3; i++){

    // Load breakpoint position - where breakpoint
    should occur
    err = flex_load_pos_bp(boardID, axis,
    breakpointPosition[i], 0xFF);
    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 modal //errors
            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 breakpoint to be triggered
    }
    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 ID, and the
            error code of the //modal error from the
            error stack on the device
            flex_read_error_msg_rtn(boardID,&commandI
            D,&resourceID, &errorCode);
            nimcDisplayError(errorCode,commandID,res
            ourceID);
            //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

Figure 12-6 shows the basic algorithm for relative breakpoints.

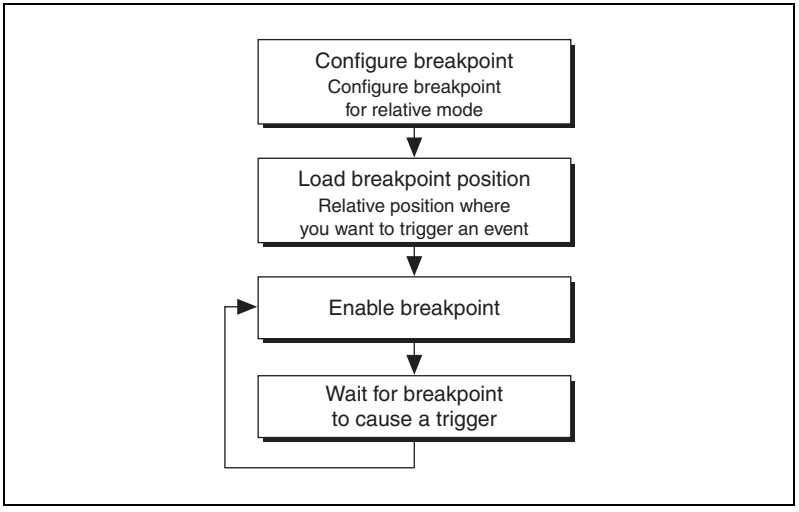


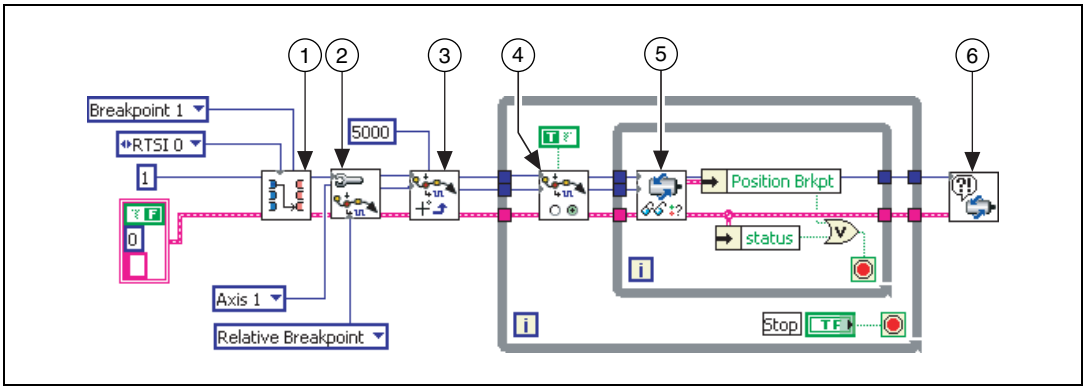
Figure 12-6. Relative Position Breakpoints Algorithm

Notice that relative breakpoints are not ideal for periodic breakpoints. There is a latency between the time a breakpoint generates and is re-enabled. If the axis is moving at sufficient velocity, the breakpoint re-enables 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 *Buffered Breakpoints (NI 7350 only)* or *Periodically Occurring Breakpoints*.

LabVIEW Code

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



1	Select Signal	3	Load Breakpoint Position	5	Read per Axis Status
2	Configure Breakpoint	4	Enable Breakpoint Output	6	Motion Error Handler

Figure 12-7. Relative Position Breakpoint with RTSI Using LabVIEW

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 = 5000;

    //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 = NIMC_AXIS1;
//////////////////////////////

// Route breakpoint 1 to RTSI line 1
err = flex_select_signal (boardID, NIMC_RTSI1
/*destination*/, NIMC_BREAKPOINT1/*source*/);
CheckError;

// Configure Breakpoint
err = flex_configure_breakpoint(boardID, axis,
NIMC_RELATIVE_BREAKPOINT, NIMC_SET_BREAKPOINT, 0);
CheckError;

// Load breakpoint position, which is position where
breakpoint should occur
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 modal //errors
            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 breakpoint to be triggered
}
```

```
    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 ID, and the
                error code of the //modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);
                //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 periodic breakpoints. For NI 7330, NI 7340, and NI 7390 controllers, use modulo breakpoints.

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 Figure 12-8.

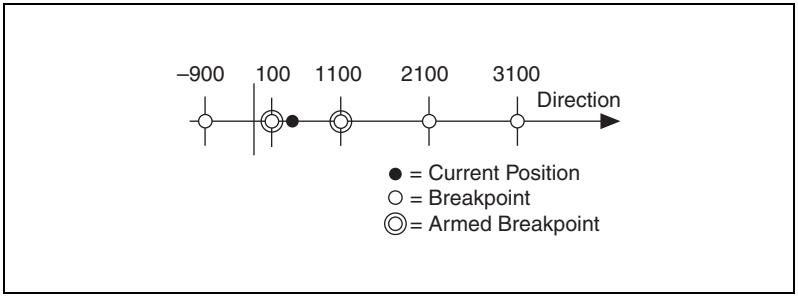


Figure 12-8. Periodic Breakpoint Every 1,000 Counts/Steps

Periodic Breakpoint Algorithm

Figure 12-9 shows the basic algorithm for periodic breakpoints.

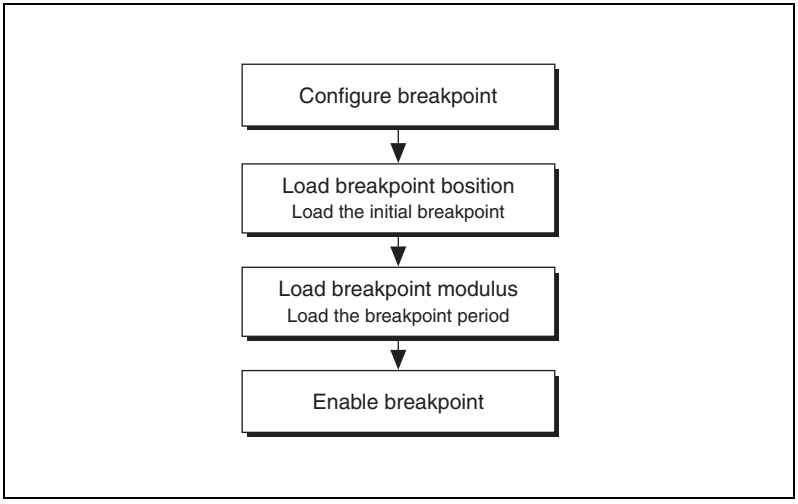
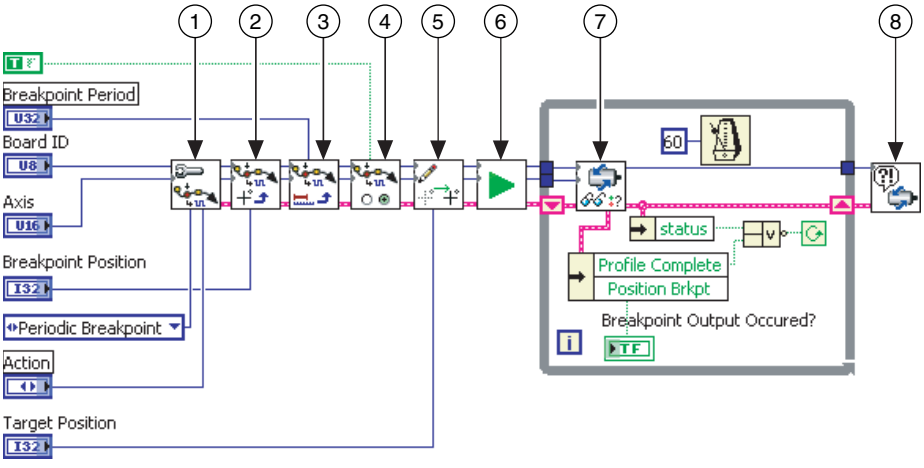


Figure 12-9. Periodic Breakpoint Algorithm

LabVIEW Code



1	Configure Breakpoint	4	Enable Breakpoint Output	7	Read per Axis Status
2	Load Breakpoint Position	5	Load Target Position	8	Motion Error Handler
3	Load Breakpoint Modulus	6	Start Motion		

Figure 12-10. Periodic Breakpoint Output

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
    u16 csr* 0; // Communication Status Register
    u8 profileStatus; // Profile Complete Status
    u8  bpStatus; // Breakpoint found Status
    i32 bpPos; // Breakpoint Position
    i32 bpPer; // Breakpoint Period
    i32 targetPos; // Target Position
    i32 currentPos; // Current Position
    u16 axisStatus; // Status of the axis
```

```
//Variables for modal error handling
u16 commandID; // The commandID of the function
u16 resourceID; // The resource ID
i32 errorCode;

//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_PERIODI
C_BREAKPOINT,NIMC_NO_CHANGE,0);
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,&axisStat
    us);
    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
    Complete=%d\r",currentPos,bpStatus,profileStatus
    );

    //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_SPA
           CE1, NIMC_DECEL_STOP, 0);//Stop the
           Motion
           err = csr & NIMC_MODAL_ERROR_MSG;
           CheckError;
    }
}while(!profileStatus);

printf("\nFinished.\n");

return; // Exit the Application
////////////////////////////////////////////////////
//////////////////////
// Error Handling
//
```

```
    nimcHandleError;
    // 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 modal
                //error from the error stack on the board
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID,&errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);
                //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
}
```

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 Figure 12-11.

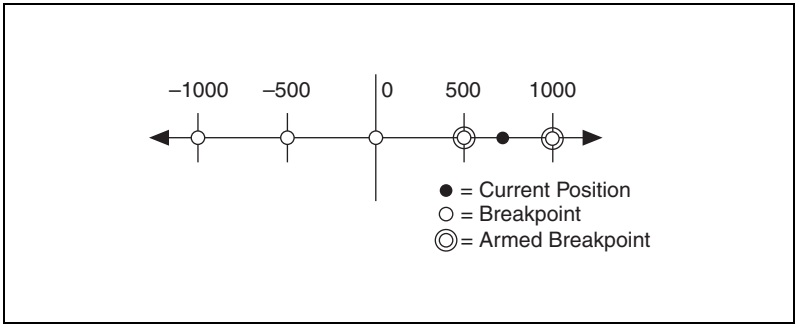


Figure 12-11. Breakpoint Modulus of 500

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 Figure 12-12.

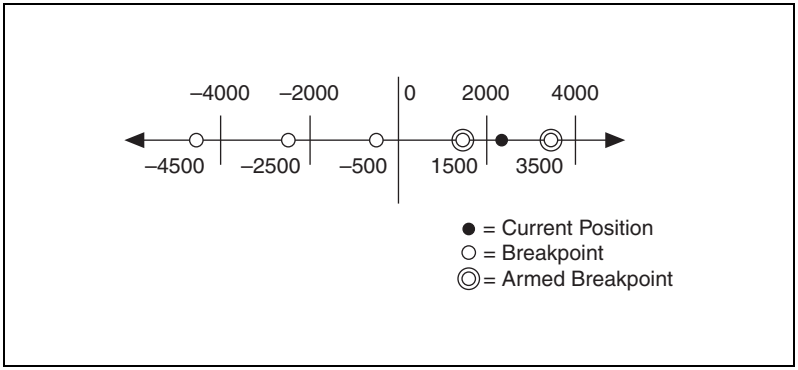


Figure 12-12. Breakpoint Modulus of 2000 with an Offset of 500

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

Modulo Breakpoints Algorithm

Figure 12-13 shows the basic algorithm for modulo breakpoints.

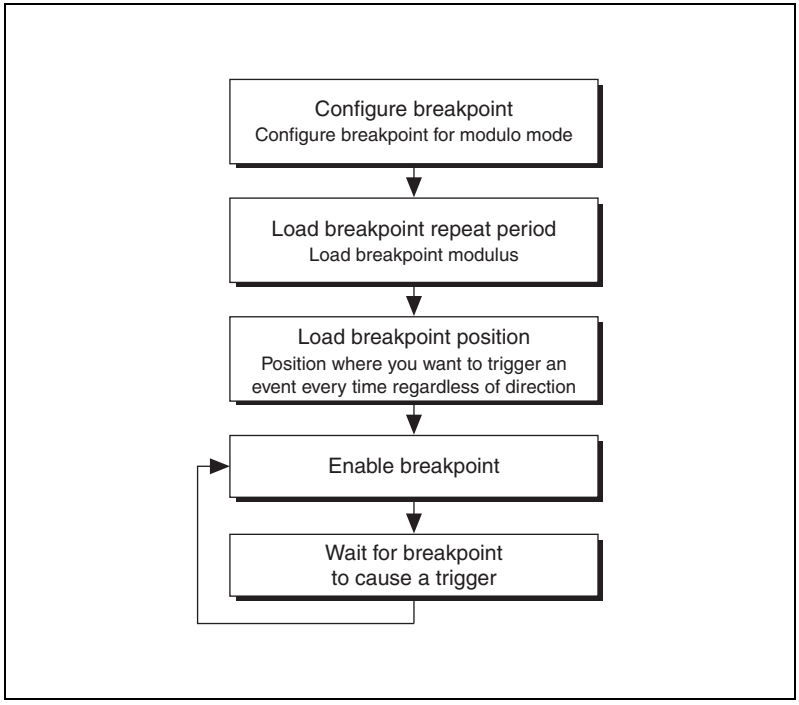
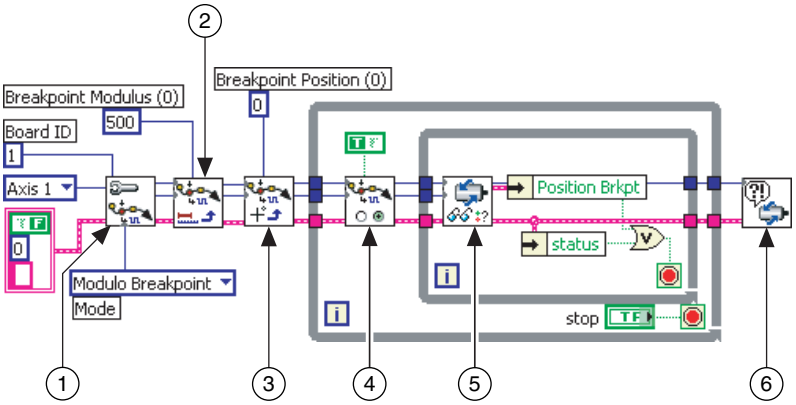


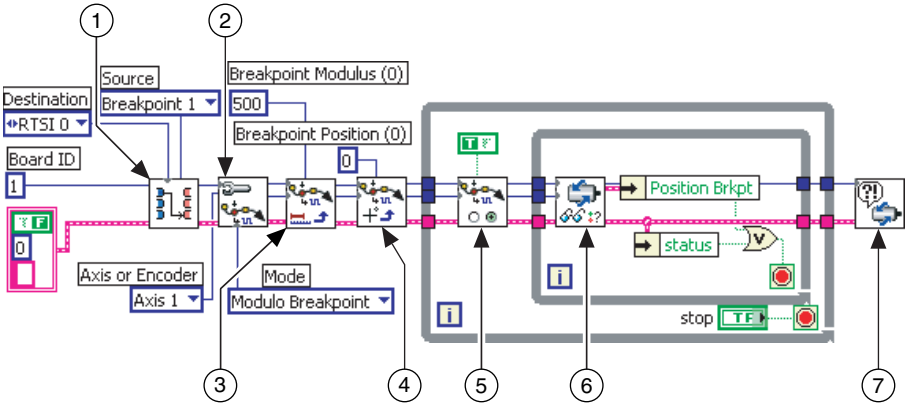
Figure 12-13. Modulo Breakpoints Algorithm

LabVIEW Code



1	Configure Breakpoint	3	Load Breakpoint Position	5	Read per Axis Status
2	Load Breakpoint Modulus	4	Enable Breakpoint Output	6	Motion Error Handler

Figure 12-14. Modulo Breakpoint Using LabVIEW



1	Select Signal	4	Load Breakpoint Position	6	Read per Axis Status
2	Configure Breakpoint	5	Enable Breakpoint Output	7	Motion Error Handler
3	Load Breakpoint Modulus				

Figure 12-15. Modulo Breakpoint with RTSI Using LabVIEW

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

    //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 = NIMC_AXIS1;
    ////////////////////////////////

    // Route breakpoint 1 to RTSI line 1
    err = flex_select_signal (boardID, NIMC_RTSI1
    /*destination*/, NIMC_BREAKPOINT1/*source*/);
    CheckError;

    //Configure Breakpoint
    err = flex_configure_breakpoint(boardID, axis,
    NIMC_MODULO_BREAKPOINT, NIMC_SET_BREAKPOINT,
    NIMC_OPERATION_SINGLE);
    CheckError;

    // Load Breakpoint Modulus - repeat period
    err = flex_load_bp_modulus(boardID, axis, 500,
    0xFF);
    CheckError;

    // Load Breakpoint Position - position at which
    breakpoint should //occur 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 //status
          err = flex_read_axis_status_rtn(boardID,
          axis, &axisStatus);
          CheckError;

          // Read the communication status register
          and check the modal //errors
          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 breakpoint to be triggered
}
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 ID, and the error
    code of the modal //error from the error stack on
    the device
    flex_read_error_msg_rtn(boardID,&commandID,&reso
    urceID, &errorCode);
    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
}
```

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 buffered and non-buffered high-speed capture methods.

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

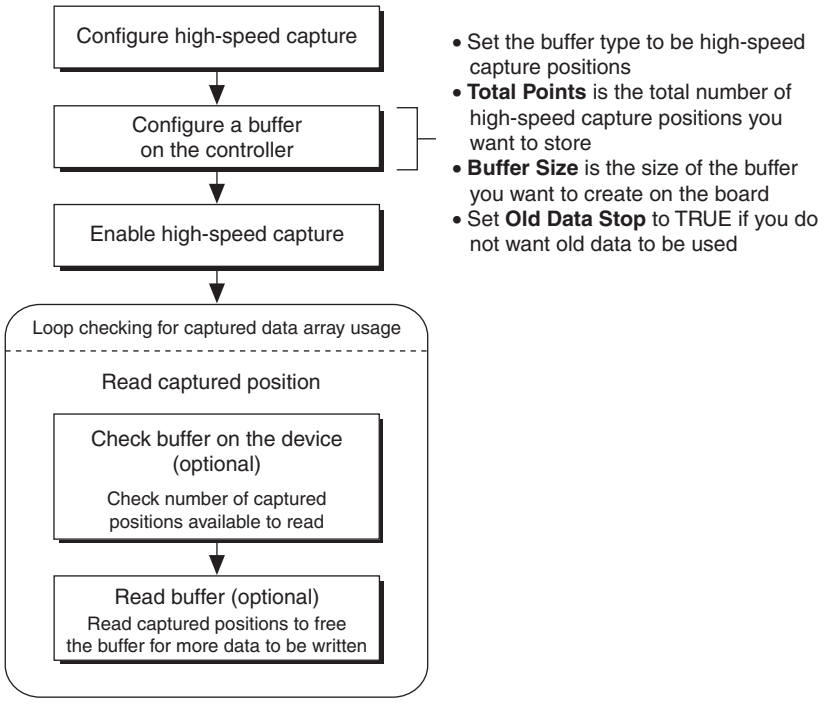
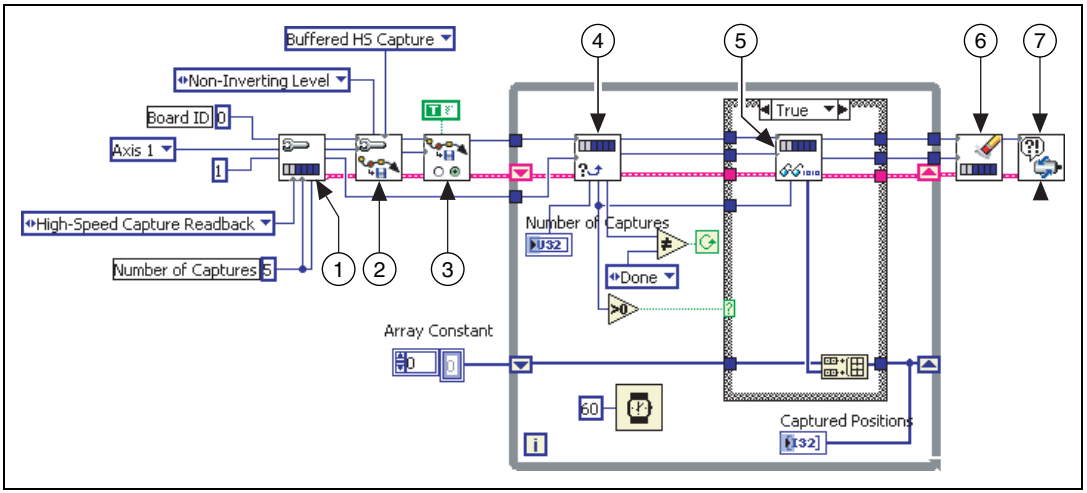


Figure 12-16. Buffered High-Speed Capture Algorithm

LabVIEW Code



1	Configure Buffer	4	Check Buffer	6	Clear Buffer
2	Configure High-Speed Capture	5	Read Buffer	7	Motion Error Handler
3	Enable High-Speed Capture				

Figure 12-17. Buffered High-Speed Capture in LabVIEW

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 bufferSize = 100;// The size of the buffer to
    allocate on the //motion controller
    u32 totalPoints = 100;// The number of high speed
    capture to //acquire
    i32 capturedPositions[100];// Array to store the
    captured //positions
    f64 actualInterval;// The interval at which the
    motion controller can //really contour
    u32 backlog;// Indicates the available space for
    captured positions
```

```
u32 pointsDone;// Indicates the number of points that
have been //consumed
u16 bufferState;// Indicates the state of the onboard
buffer
u32 currentDataPoint = 0;// Indicates the next points
to be read //from the buffer
i32* readBuffer = NULL;// The temporary array that is
created to //read captured positions
u32 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 = NIMC_AXIS1;

////////////////////////////////
// Configure buffer on motion controller memory (RAM)
// Notice requested time interval is hardcoded to 10
milliseconds
err = flex_configure_buffer(boardID, 1 /*buffer
number*/, axis, NIMC_HS_CAPTURE_READBACK,
bufferSize, totalPoints, NIMC_TRUE, 10,
&actualInterval);
CheckError;

// Configure High-Speed Capture
err = flex_configure_hs_capture(boardID, axis,
NIMC_HS_LOW_TO_HIGH_EDGE, NIMC_OPERATION_BUFFERED);
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*/, &backlog, &bufferState, &pointsDone);

    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 //captured position from
           the buffer
           err = flex_read_buffer_rtn(boardID,
           1/*buffer number*/, backlog, readBuffer);
           for(i=0;i<backlog;i++){
                  if(currentDataPoint > totalPoints)
                  break;
                         capturedPositions[currentDa
                         taPoint] = readBuffer[i];
                  printf("capture pos %d\n",
                  capturedPositions[currentDataPoin
                  t]);
                  currentDataPoint++;
           }
           free(readBuffer);
           readBuffer = NULL;
           CheckError;
    }
    // Check for axis off status/following error or
    any modal //errors; Read the communication status
    register and check the //modal errors
    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 motion 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 ID, and the
                error code of the //modal error from the
                error stack on the device
                err = flex_read_error_msg_rtn(boardID,
                &commandID, &resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);
                //Read the communication status register
                err = 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 re-enable the high-speed capture each time it triggers.

High-Speed Capture Algorithm

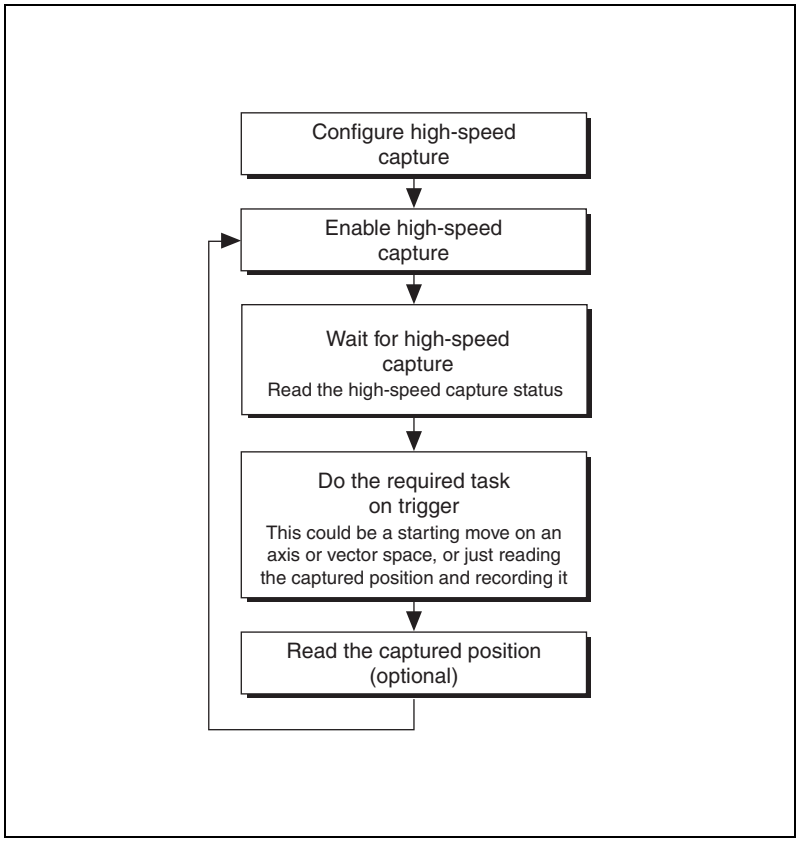
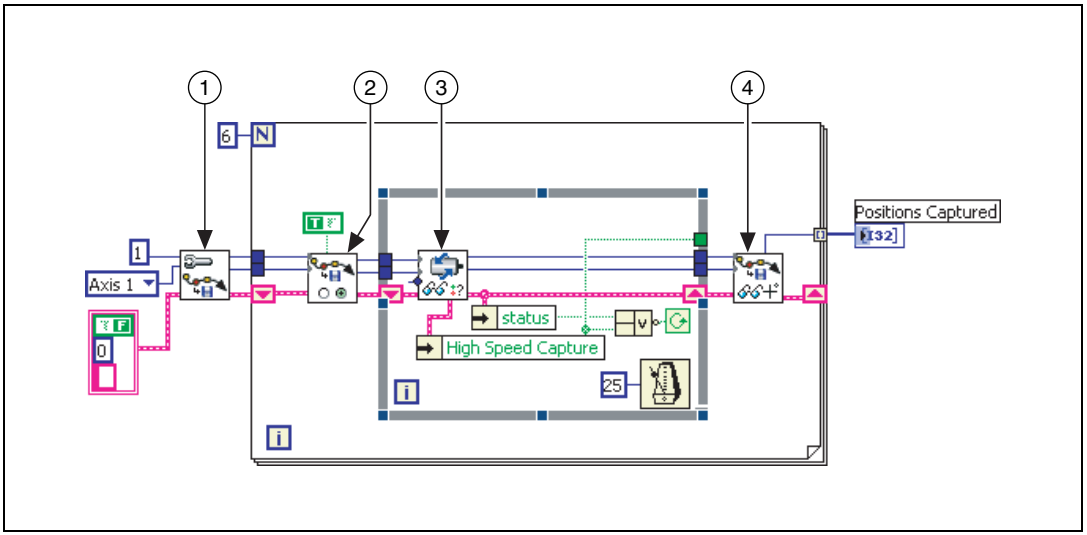


Figure 12-18. High-Speed Capture Algorithm

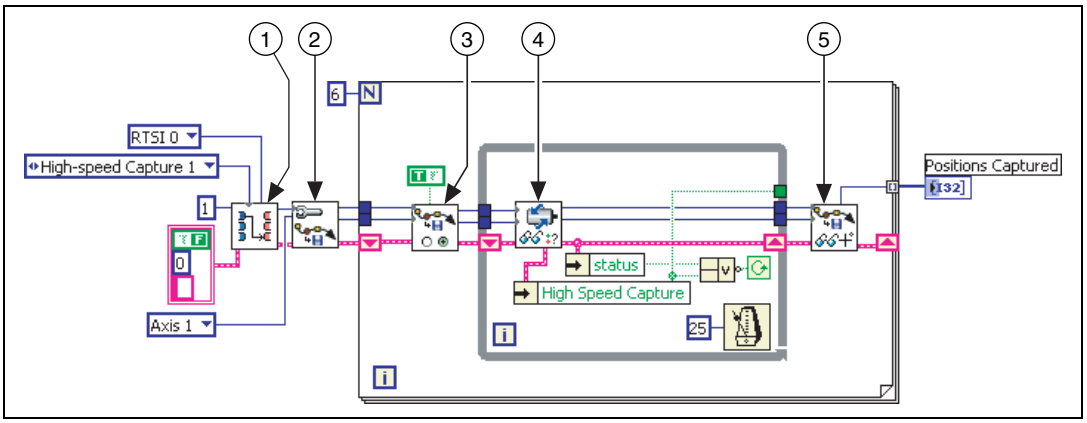
LabVIEW Code



1	Configure High-Speed Capture	3	Read per Axis Status
2	Enable High-Speed Capture	4	Read Captured Position

Figure 12-19. High-Speed Capture Using LabVIEW

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 Figure 12-20.



1 Select Signal	4 Read per Axis Status
2 Configure High-Speed Capture	5 Read Captured Position
3 Enable High-Speed Capture	

Figure 12-20. High-Speed Capture with RTSI Using LabVIEW

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 positions
    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 = NIMC_AXIS1;
////////////////////////////////

// Route HSC 1 to RTSI line 1
err = flex_select_signal (boardID, NIMC_HS_CAPTURE1
/*destination*/, NIMC_RTSI1/*source*/);
CheckError;

//Configure High-Speed Capture
err = flex_configure_hs_capture(boardID, axis,
NIMC_HS_LOW_TO_HIGH_EDGE, 0);
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 modal //errors
           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 for high-speed capture to be triggered

    err = flex_read_cap_pos_rtn(boardID, axis,
    &capturedPositions[i]);
    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 ID, and the
                error code of the //modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);

                nimcDisplayError(errorCode,commandID,res
                ourceID);

                //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 Figure 12-21, the I/O reaction task automatically re-enables 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.

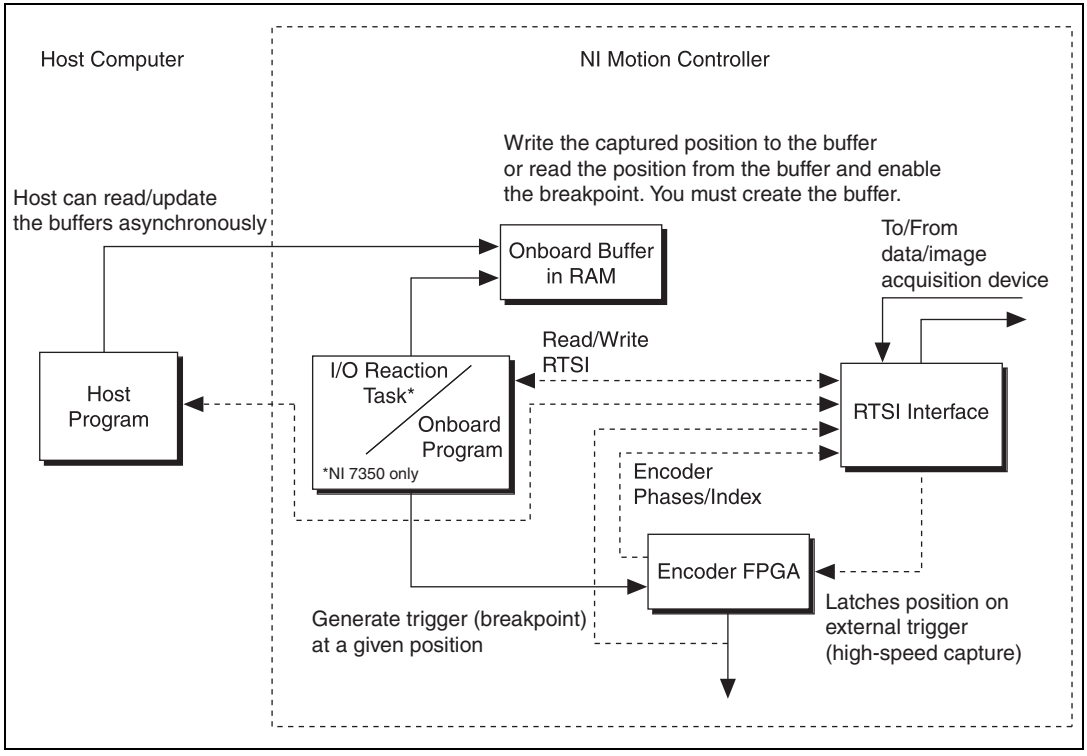


Figure 12-21. RTSI Implementation on the Motion Controller

Position Breakpoints Using RTSI

You can use the Select Signal function 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 in Figure 12-22.

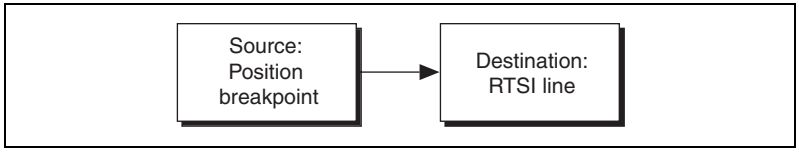


Figure 12-22. Position Breakpoint Using RTSI

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 in Figure 12-23.

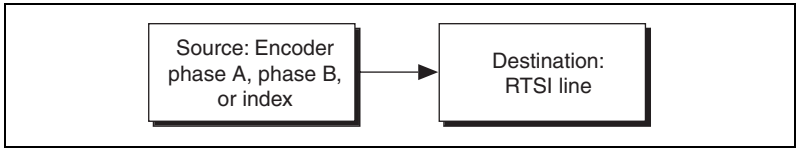


Figure 12-23. Encoder Pulses Using RTSI

Software Trigger Using RTSI

You can use the Set I/O Port MOMO function to write directly to the RTSI lines to trigger other devices, as shown in Figure 12-24.

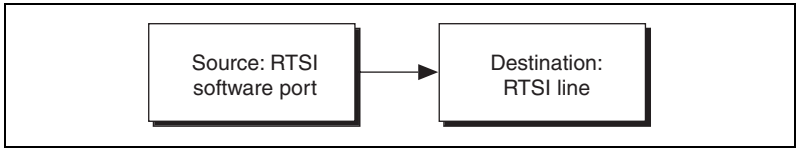


Figure 12-24. Software Trigger Using RTSI

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 in Figure 12-25.

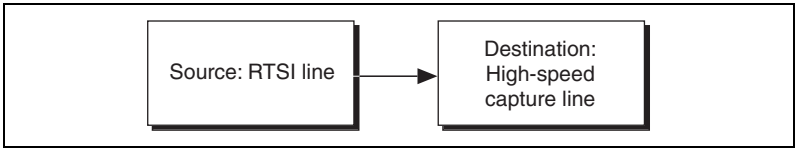


Figure 12-25. High-Speed Capture Input Using RTSI

13

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.

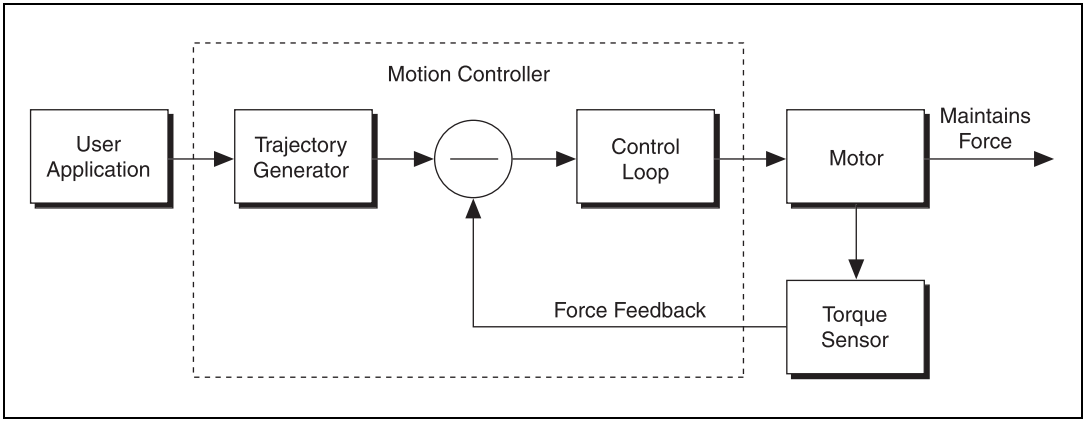


Figure 13-1. Torque Control Using Analog Feedback Flowchart

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

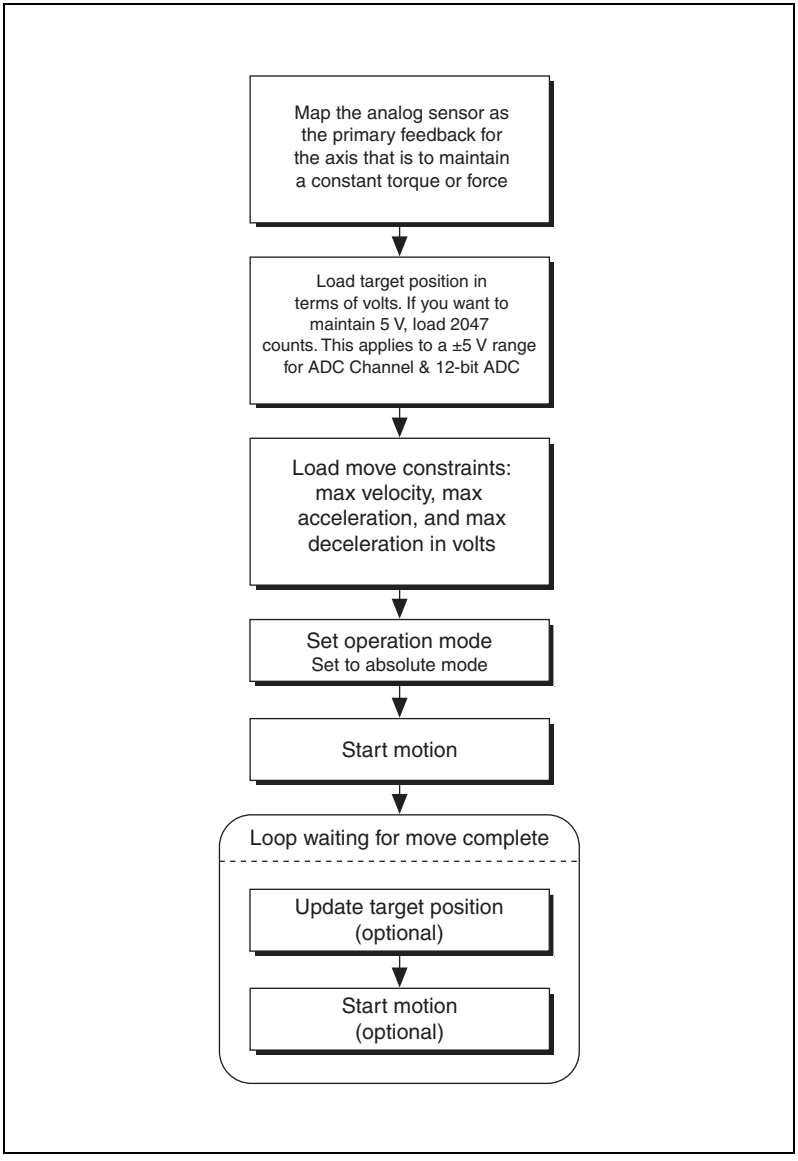
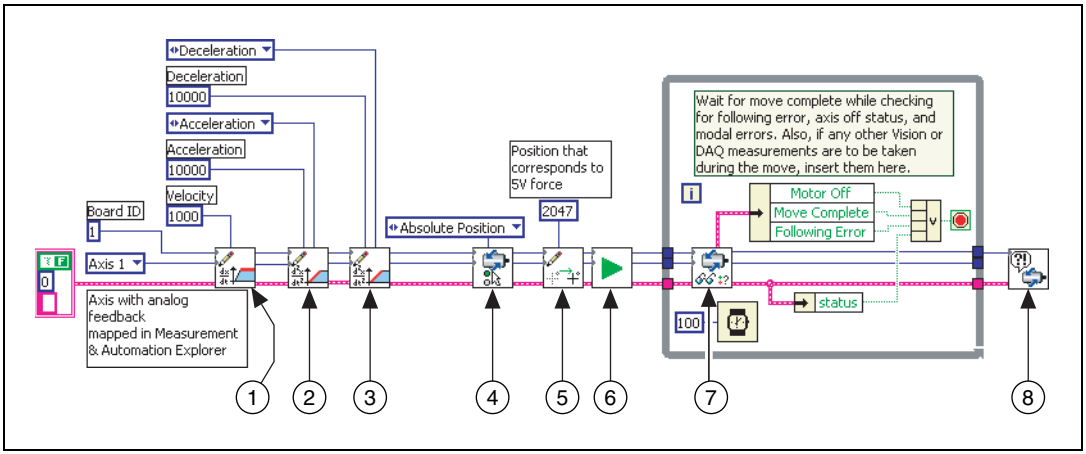


Figure 13-2. Torque Control Using Analog Feedback Algorithm

LabVIEW Code



1	Load Velocity	4	Set Operation Mode	7	Read per Axis Status
2	Load Acceleration/Deceleration	5	Load Target Position	8	Motion Error Handler
3	Load Acceleration/Deceleration	6	Start Motion		

Figure 13-3. Torque Control Using Analog Feedback Using LabVIEW

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 = NIMC_AXIS1;
    ////////////////////////////////

    //-------------------------------------------------
    //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, 100000, 0xFF);
    CheckError;

    // Set the deceleration for the move (in binary
    volts/sec^2)
```

```
err = flex_load_acceleration(boardID, axis,
NIMC_DECELERATION, 100000, 0xFF);
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 ~ 5V 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, &moveComplete);
    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 //errors
    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) && !(axisStatus &
    NIMC_AXIS_OFF_BIT));
    //Exit on move complete/following error/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 ID, and the
                error code of the //modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);

                //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
}
```

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.

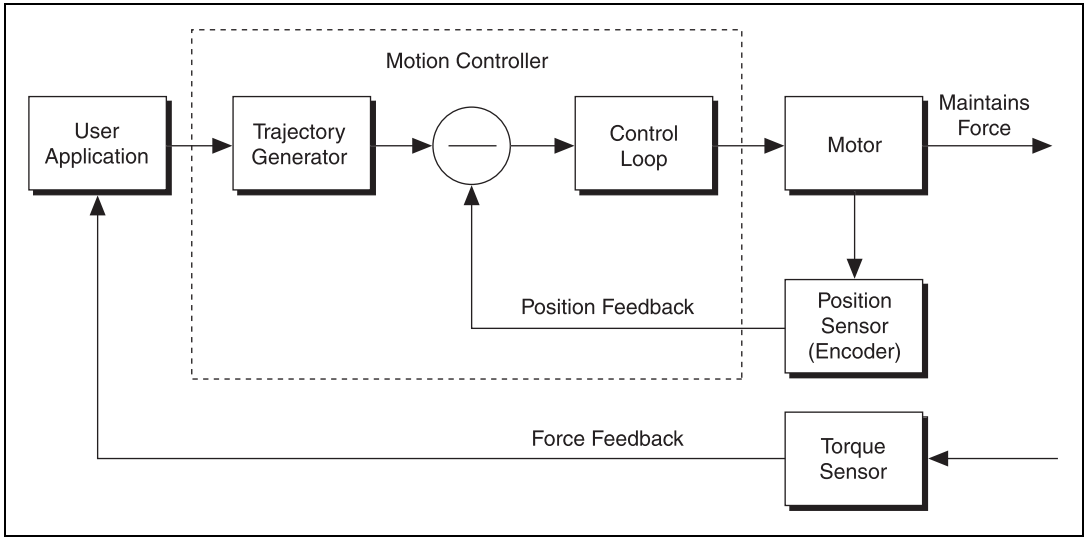


Figure 13-4. Torque Control Using Analog Feedback Flowchart

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

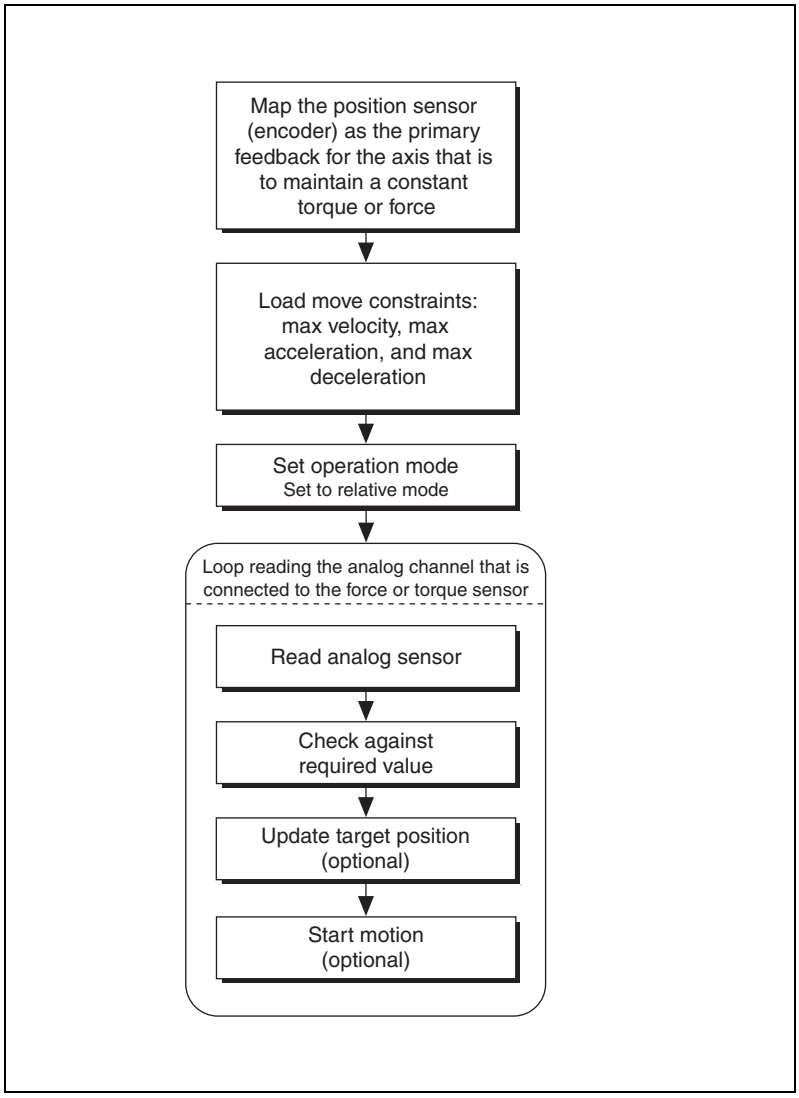
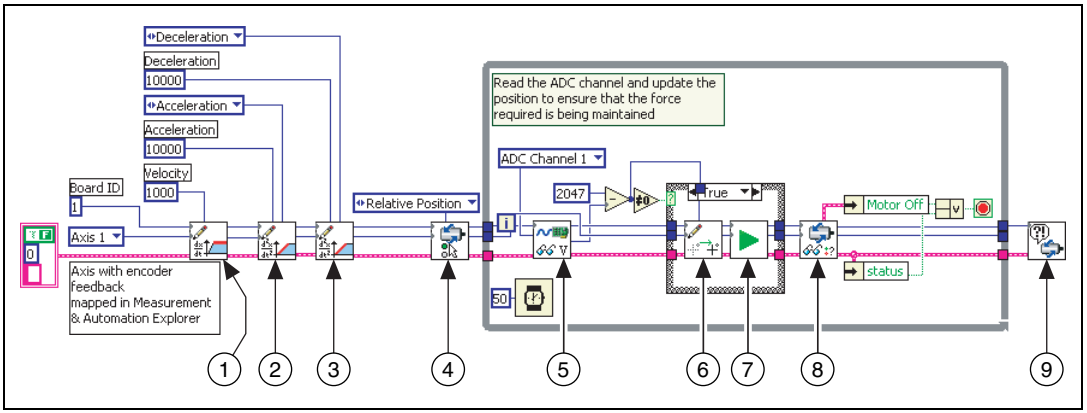


Figure 13-5. Torque Control Using Monitoring Force Algorithm

LabVIEW Code



1	Load Velocity	4	Set Operation Mode	7	Start Motion
2	Load Acceleration/Deceleration	5	Read ADC	8	Read per Axis Status
3	Load Acceleration/Deceleration	6	Load Target Position	9	Motion Error Handler

Figure 13-6. Torque Control Using Monitoring Force in LabVIEW

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 = NIMC_AXIS1;
    // 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, 100000, 0xFF);
    CheckError;

    // Set the deceleration for the move (in
    counts/sec^2)
```

```
err = flex_load_acceleration(boardID, axis,
NIMC_DECELERATION, 100000, 0xFF);
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 //be updated
    err = flex_read_adc16_rtn(boardID, NIMC_ADC1,
    &adcValue);
    CheckError;

    if( (constant - adcValue) != 0){
           err = flex_load_target_pos(boardID, axis,
           (constant - adcValue), 0xFF);
           CheckError;

           // Move based on delta force
           err = flex_start(boardID, axis, 0);
           CheckError;
    }

    // Check the move complete status/following
    error/axis off //status
    err = flex_read_axis_status_rtn(boardID, axis,
    &axisStatus);
    CheckError;

    // Read the communication status register and
    check the modal //errors
    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 ID, and the
            error code of the //modal error from the
            error stack on the device
            flex_read_error_msg_rtn(boardID,&commandI
            D,&resourceID, &errorCode);
            nimcDisplayError(errorCode,commandID,res
            ourceID);
            //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
}
```

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

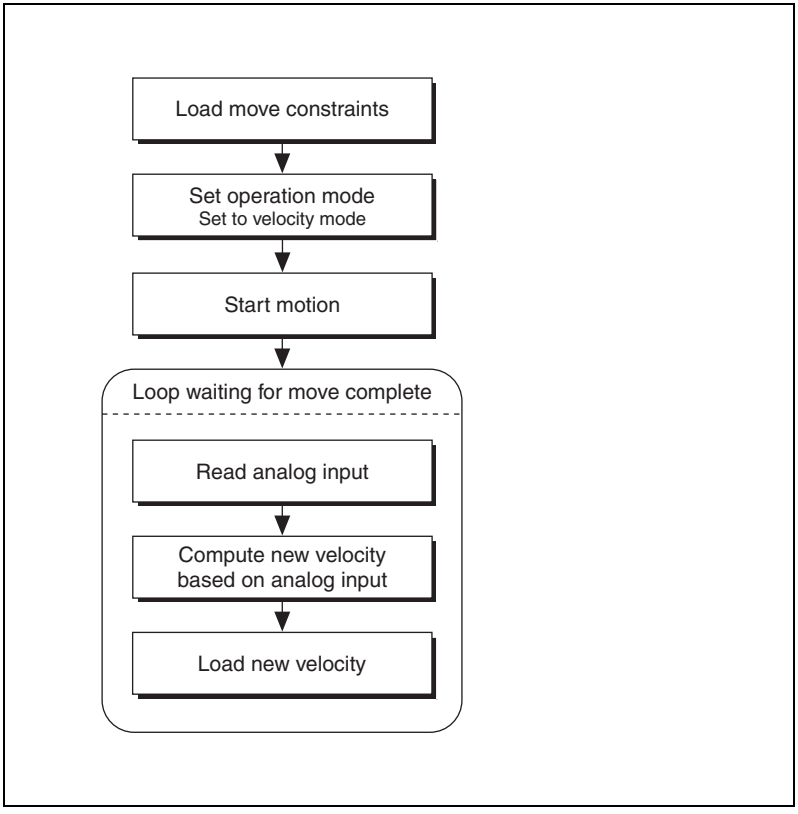
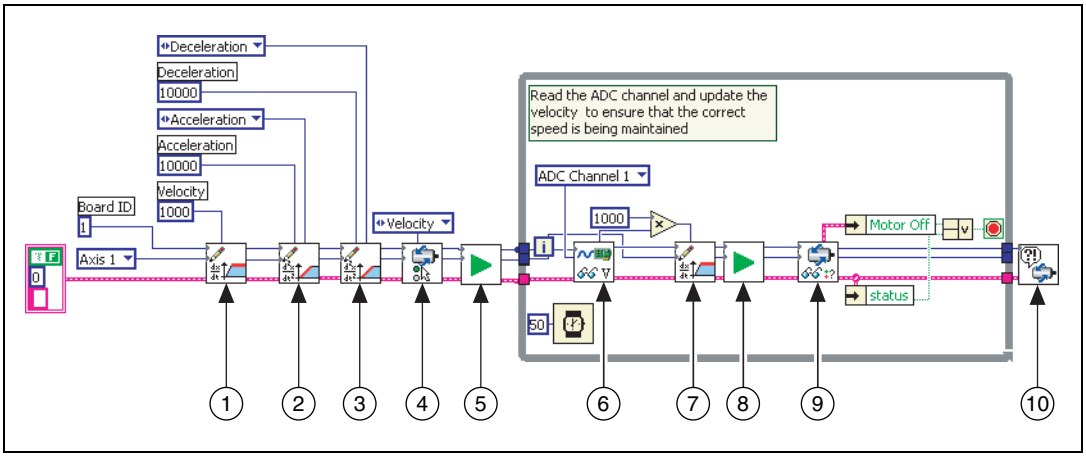


Figure 13-7. Speed Control Based on Analog Feedback Algorithm

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

LabVIEW Code



1	Load Velocity	5	Start Motion	8	Start Motion
2	Load Acceleration/Deceleration	6	Read ADC	9	Read per Axis Status
3	Load Acceleration/Deceleration	7	Load Velocity	10	Motion Error Handler
4	Set Operation Mode				

Figure 13-8. Speed Control Based on Analog Feedback Using LabVIEW

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 = NIMC_AXIS1;
    // constant to multiply the ADC value read to
    calculate the //required velocity
    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, 100000, 0xFF);
    CheckError;

    // Set the deceleration for the move (in
    counts/sec^2)
    err = flex_load_acceleration(boardID, axis,
    NIMC_DECELERATION, 100000, 0xFF);
    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 //be updated
    err = flex_read_adc16_rtn(boardID, NIMC_ADC1,
    &adcValue);
    CheckError;

    // Set the velocity based on the ADC value read
    err = flex_load_velocity(boardID, axis, (adcValue
    * constant), 0xFF);
    CheckError;

    // Update the velocity
    err = flex_start(boardID, axis, 0);
    CheckError;

    // Check the move complete status/following
    error/axis off //status
    err = flex_read_axis_status_rtn(boardID, axis,
    &axisStatus);
    CheckError;

    // Read the communication status register and
    check the modal //errors
    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 ID, and the
                error code of the //modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);

                //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
}
```

14

Onboard Programs

This chapter provides information about how onboard programs work for the NI SoftMotion Controller and for NI 73*xx* 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 1 KHz for Ormec and 100 Hz for CANopen.

Using Onboard Programs with NI 73*xx* Motion Controllers

You can use the real-time operating system on the NI 73*xx* motion controller to run custom programs. This functionality allows you to offload some motion-specific 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 Figure 14-1. 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.

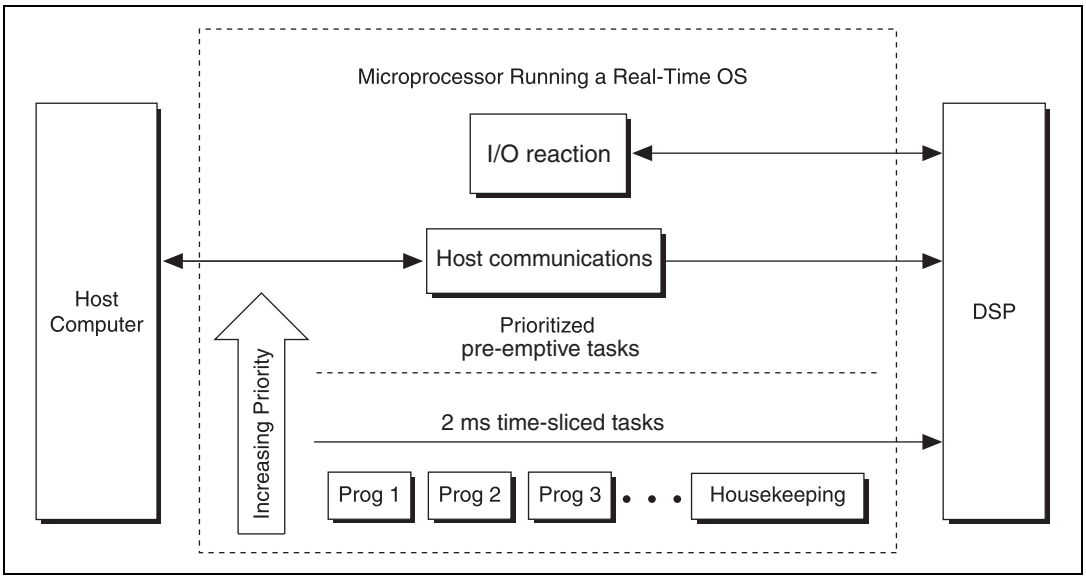


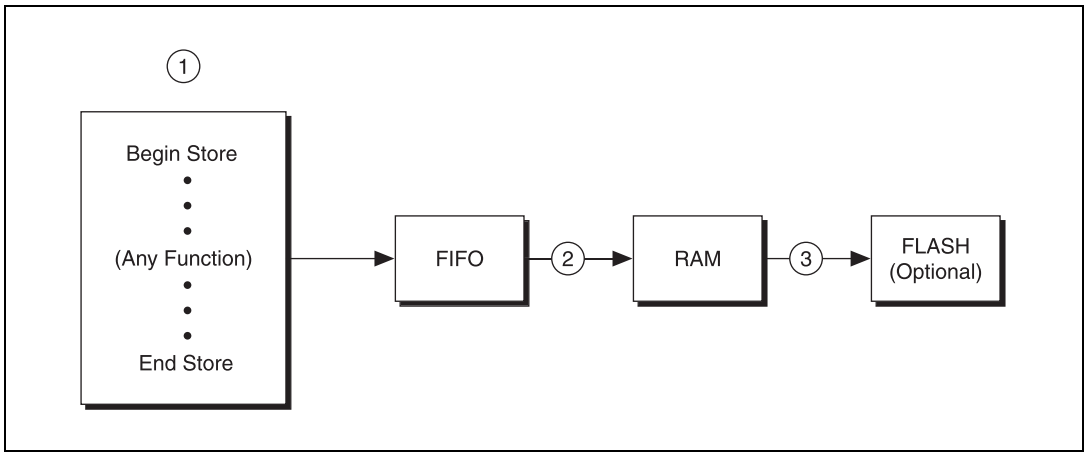
Figure 14-1. Onboard Program Priority

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 the *Timing Loops* section of Chapter 4, *What You Need to Know about Moves*, for information about programming delays in the loops.

Writing Onboard Programs

Note This section and the sections that follow it apply only to the NI 73*xx* motion controllers.

Almost all NI-Motion functions that execute on the host can run onboard. You can store up to 32 onboard programs on the motion controller. 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 in Figure 14-2.



1 Write the program you want to load onto onboard memory. You can use any NI-Motion functions between Begin and End Store.
2 Transfer the program to onboard RAM using the host communication handler.
3 Store the program to FLASH memory for more permanent storage (optional).

Figure 14-2. Writing Onboard Programs

Algorithm

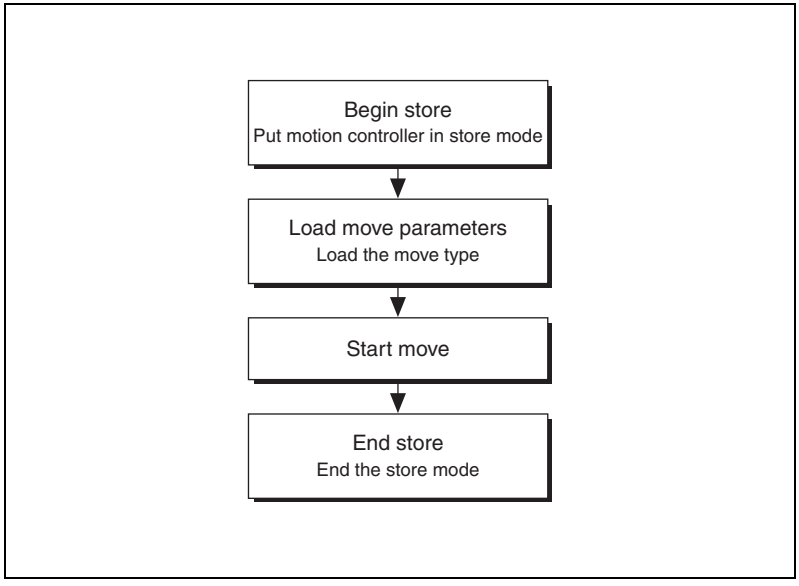
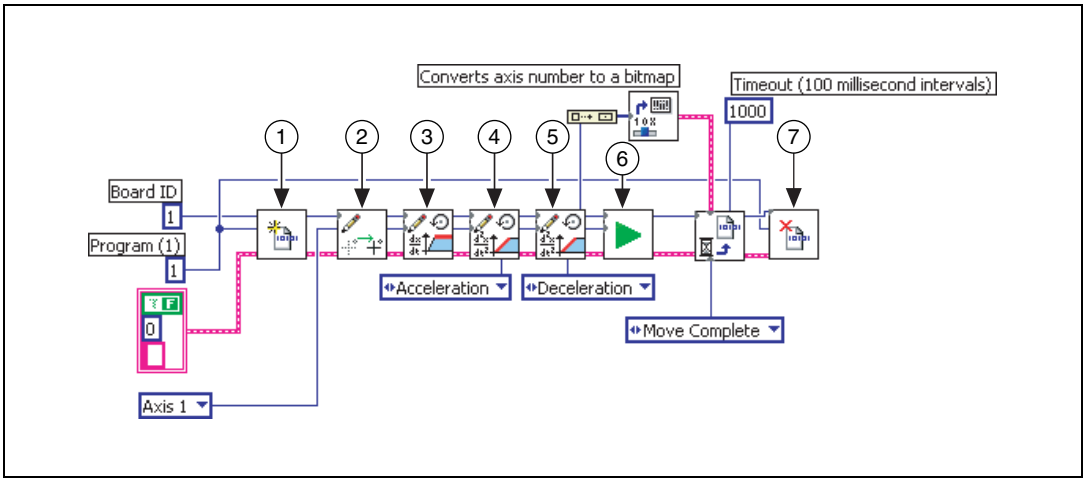


Figure 14-3. Basic Onboard Program Algorithm

LabVIEW Code



1	Begin Program Storage	5	Load Accel/Decel in RPS/s
2	Load Target Position	6	Start Motion
3	Load Velocity in RPM	7	End Program Storage
4	Load Accel/Decel in RPS/s		

Figure 14-4. Onboard Program in LabVIEW

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 = NIMC_AXIS1;
    ////////////////////////////////

    //-----------------------------------------------
    // Onboard program 1. This onboard program moves axis
    one clockwise //5,000 counts (steps). To execute this
    onboard program 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 5,000
    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;

    // 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 ID, and the
              error code of the //modal error from the
              error stack on the device
              flex_read_error_msg_rtn(boardID,&commandI
              D,&resourceID, &errorCode);
              nimcDisplayError(errorCode,commandID,res
              ourceID);

              //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

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

Running an Onboard Program

Run Program executes previously stored programs from RAM or FLASH. Typically, you must call the Run Program function 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 the Run Program function.

Note Recursively calling an onboard program generates an error.

Stopping an Onboard Program

Stop Program ends the execution of an onboard program that is currently running.

Stopping an onboard program using the Stop Program function 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 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 function to stop the motion of the axis or axes.

Pausing/Resuming an Onboard Program

The Pause/Resume Program function 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 Similarly to the Stop Program function, 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 function or the Blend Motion function 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 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

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 Wait on Condition 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 Figure 14-5.

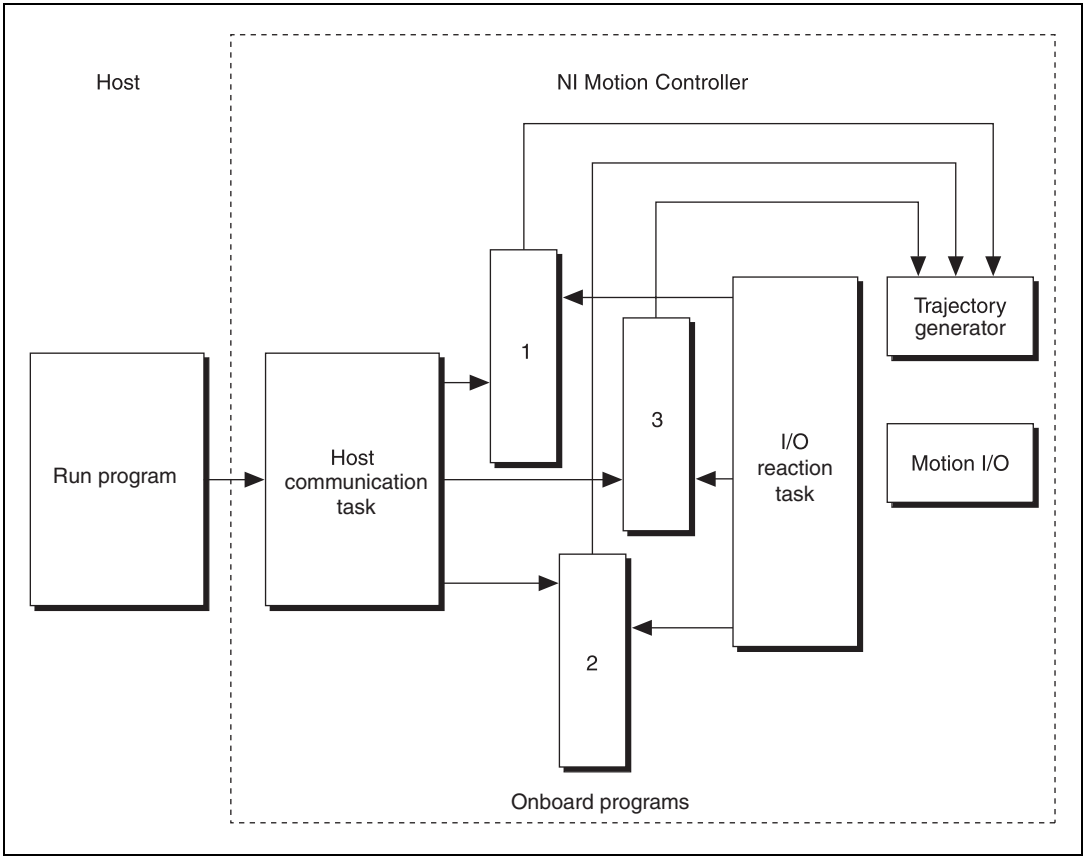


Figure 14-5. Executing Onboard Programs

Onboard Program Conditional Execution Algorithm

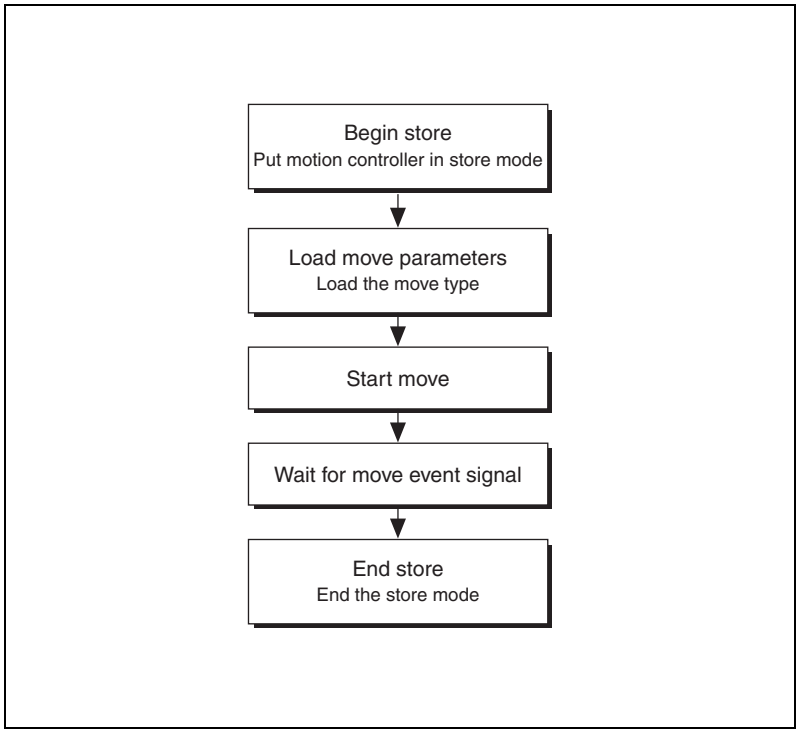
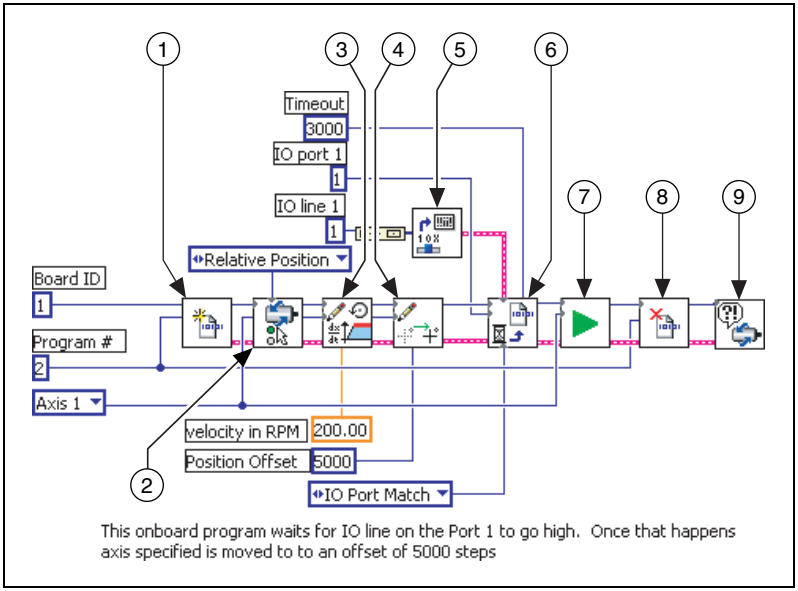


Figure 14-6. Onboard Program Conditional Execution Algorithm

LabVIEW Code



1	Begin Program Storage	4	Load Target Position	7	Start Motion
2	Set Operation Mode	5	Select MOMO	8	End Program Storage
3	Load Velocity in RPM	6	Wait on Condition	9	Motion Error Handler

Figure 14-7. Onboard Program Waiting for an I/O Line to Go Active

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 = NIMC_AXIS1;
////////////////////////////////

// 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 5,000
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_event(boardID, NIMC_IO_PORT1,
NIMC_WAIT, NIMC_CONDITION_IO_PORT_MATCH,
(u8)(1<<1)/*Indicates line 1*/, 0, NIMC_MATCH_ALL,
10000 /*time out*/, 0);
CheckError;

// Start the move
err = flex_start(boardID, axis, 0);
CheckError;

// Wait for move to complete
err = flex_wait_on_event(boardID, 0, NIMC_WAIT,
NIMC_CONDITION_MOVE_COMPLETE, (u8)(1<<axis), 0,
NIMC_MATCH_ALL, 1000 /*time out*/, 0);
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 ID, and the
                error code of the //modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);

                //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

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.

Algorithm

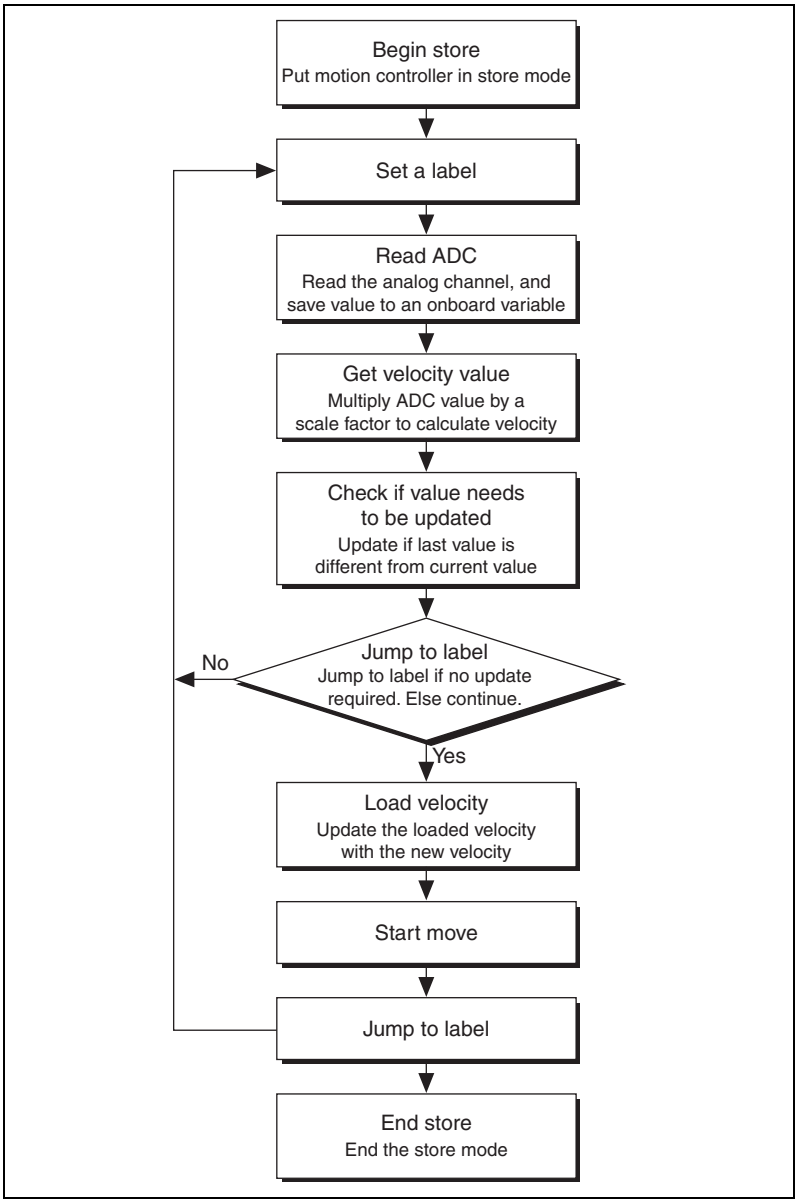
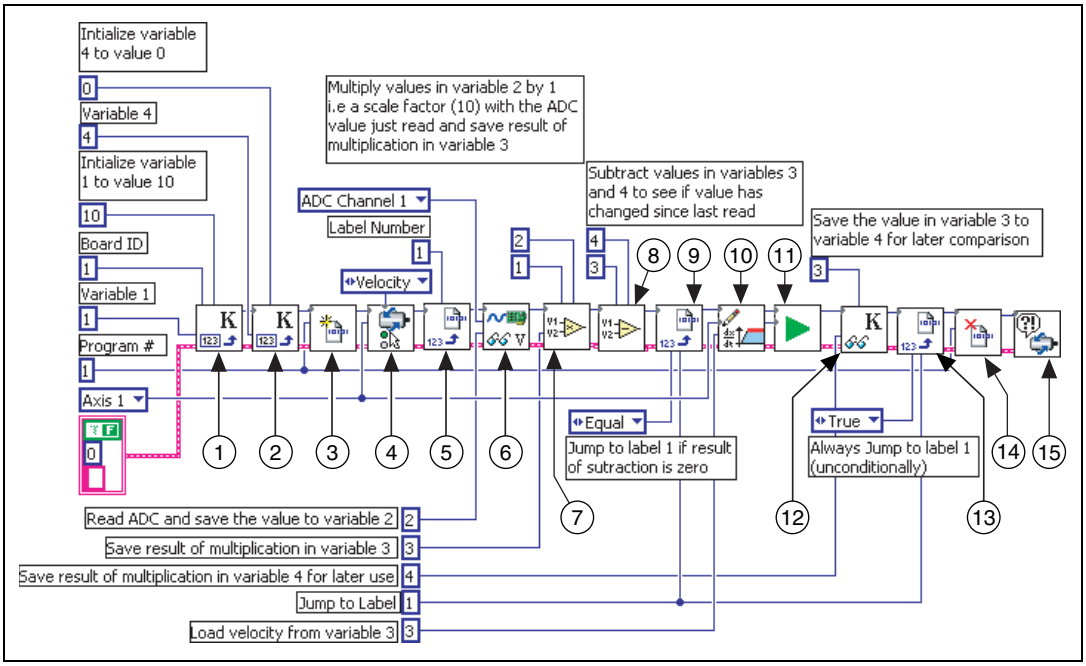


Figure 14-8. Updating Velocity Based on ADC Channel Algorithm

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

LabVIEW Code



1	Load Constant to Variable	6	Read ADC	11	Start Motion
2	Load Constant to Variable	7	Multiply Variables	12	Read Variable
3	Begin Program Storage	8	Subtract Variables	13	Jump to Label on Condition
4	Set Operation Mode	9	Jump to Label on Condition	14	End Program Storage
5	Insert Program Label	10	Load Velocity	15	Motion Error Handler

Figure 14-9. Updating Velocity Based on ADC Channel in LabVIEW

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
    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 = NIMC_AXIS1;

    // constant to multiply the ADC value read to
    calculate the //required velocity
    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_adc16(boardID, NIMC_ADC1, 2);
CheckError;

//Multiply variable 2 (ADC value) with variable 1
(constant)
// Save the result in variable 3
err = flex_mult_vars(boardID, 1, 2, 3);
CheckError;

//Subtract value in variable 3 from variable 4. The
result is //unimportant, you 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_on_event (boardID, 0,
NIMC_CONDITION_EQUAL, 0, 0, NIMC_MATCH_ALL, 1/*label
number*/);

// 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 cycle
err = flex_read_var(boardID, 3, 4);
CheckError;

// Jump back to label 1 unconditionally
err = flex_jump_on_event (boardID, 0,
NIMC_CONDITION_TRUE, 0, 0, NIMC_MATCH_ALL, 1/*label
number*/);
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 ID, and the
            error code of the //modal error from the
            error stack on the device
            flex_read_error_msg_rtn(boardID,&commandI
            D,&resourceID, &errorCode);
            nimcDisplayError(errorCode,commandID,res
            ourceID);
            //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

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

Onboard Program Algorithm

Figure 14-10 shows an onboard program waiting for an I/O line to go active before starting a move.

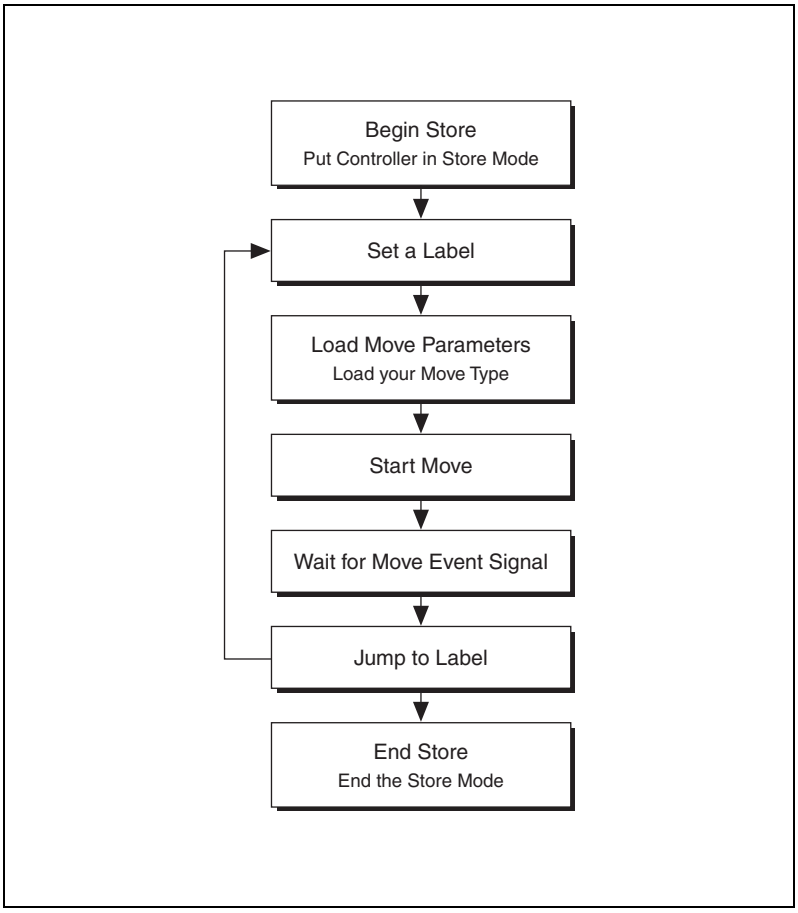
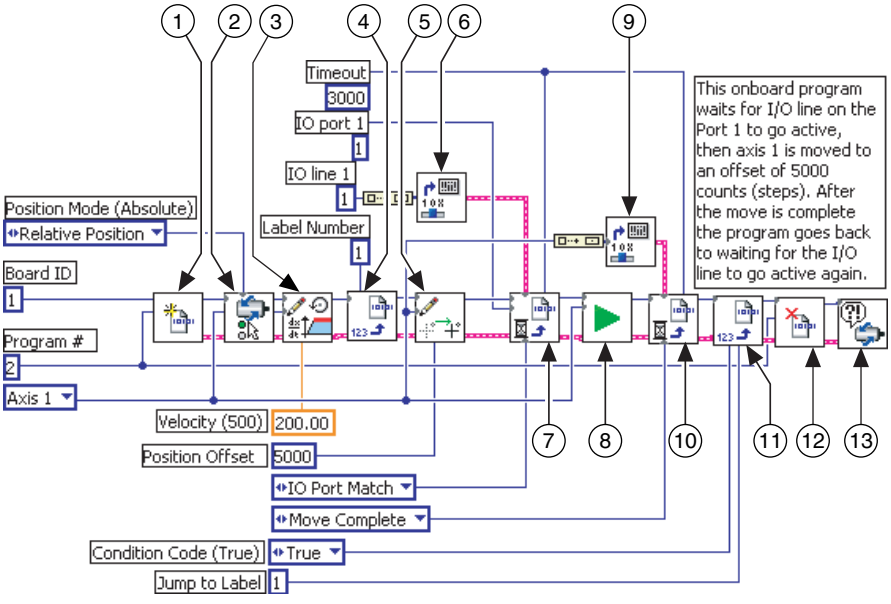


Figure 14-10. Using Labels with Onboard Programs

LabVIEW Code



1	Begin Program Storage	6	Select MOMO	10	Wait on Condition
2	Set Operation Mode	7	Wait on Condition	11	Jump to Label on Condition
3	Load Velocity in RPM	8	Start Motion	12	End Program Storage
4	Insert Program Label	9	Select MOMO	13	Motion Error Handler
5	Load Target Position				

Figure 14-11. Continuously Executing Onboard Program in LabVIEW

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 = NIMC_AXIS1;
    ////////////////////////////////

    // 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_event(boardID, NIMC_IO_PORT1,
NIMC_WAIT, NIMC_CONDITION_IO_PORT_MATCH,
(u8)(1<<1)/*Indicates line 1*/, 0, NIMC_MATCH_ALL,
10000 /*time out*/, 0);
CheckError;

// Start the move
err = flex_start(boardID, axis, 0);
CheckError;

// Wait for move to complete
err = flex_wait_on_event(boardID, 0, NIMC_WAIT,
NIMC_CONDITION_MOVE_COMPLETE, (u8)(1<<axis), 0,
NIMC_MATCH_ALL, 1000 /*time out*/, 0);
CheckError;

// Jump unconditionally to label 1 and check IO line
again
err = flex_jump_on_event (boardID, 0,
NIMC_CONDITION_TRUE, 0, 0, NIMC_MATCH_ALL, 1/*label
number*/);
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 ID, and the
         error code of the //modal error from the
         error stack on the device
         flex_read_error_msg_rtn(boardID,&commandI
         D,&resourceID, &errorCode);
         nimcDisplayError(errorCode,commandID,res
         ourceID);
```

```
                //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

NI-Motion always performs math operations on values stored in onboard variables, and all math operations set a global condition that the Jump to Label 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 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 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

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 function point to 0x81 causes the position to end up in the address contained in onboard variable 1, as shown in Figure 14-12.

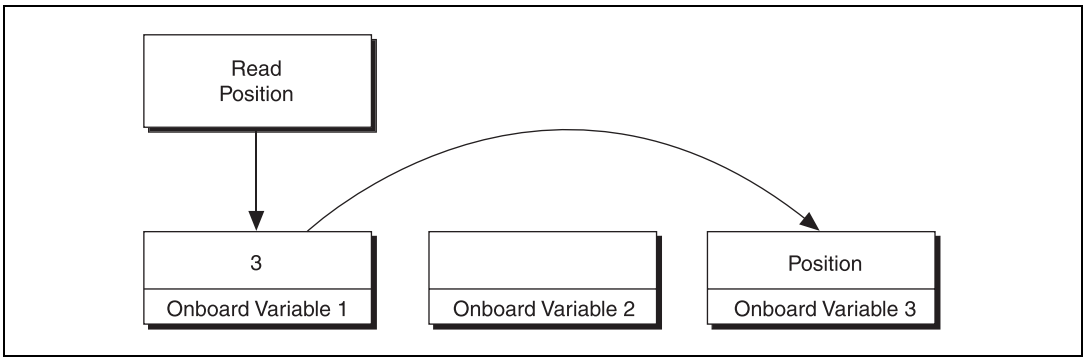


Figure 14-12. Reading an Indirect Variable

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

Onboard Buffers

You can use the memory on the NI motion controllers to create general-purpose buffers to read and write data, as shown in Figure 14-13.

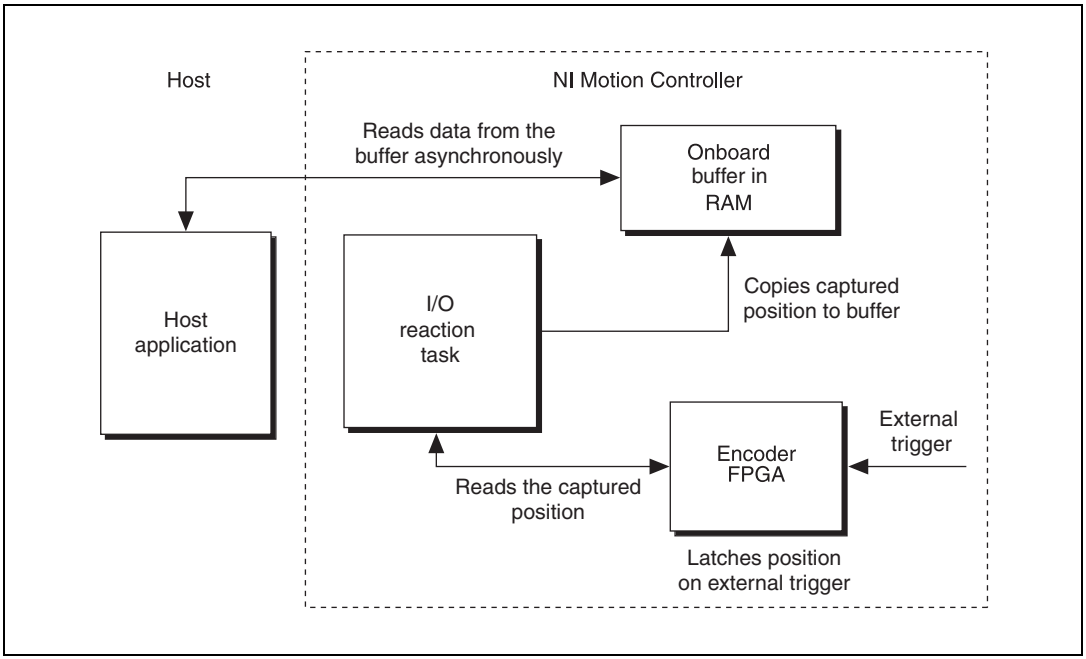


Figure 14-13. Onboard Buffer Data Flow

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.

Algorithm

Figure 14-14 shows the algorithm for using onboard buffers to store data.

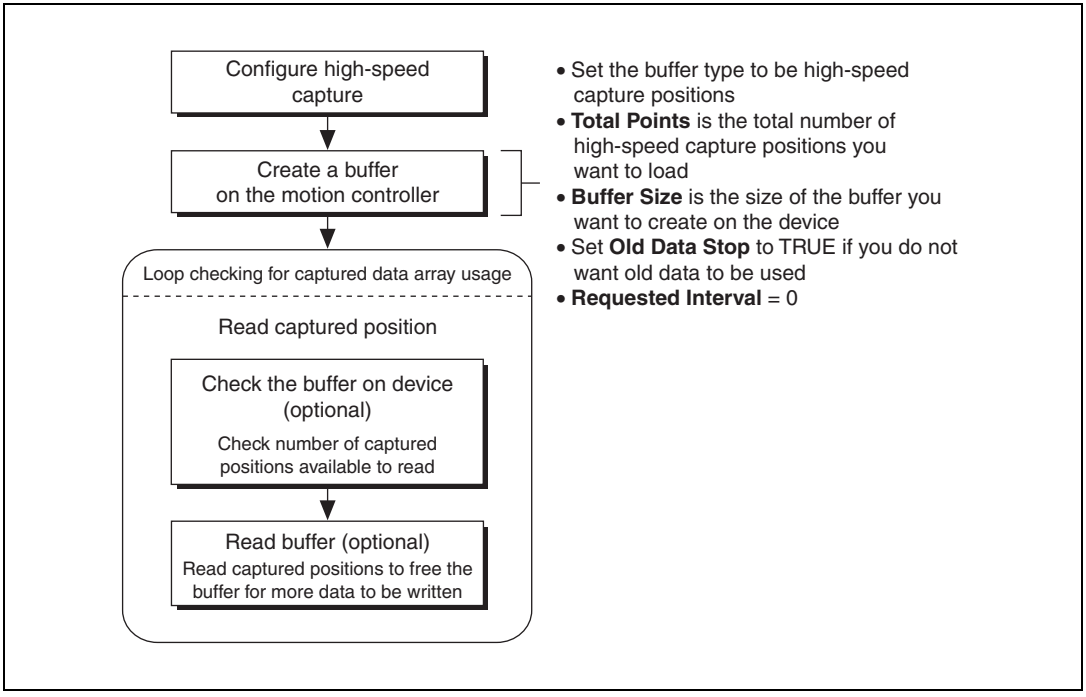


Figure 14-14. Onboard Buffer Algorithm

Synchronizing Host Applications with Onboard Programs

The host and the onboard program can write to the move complete status (MCS) register using the Set Status MOMO function. This 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 Figure 14-15.

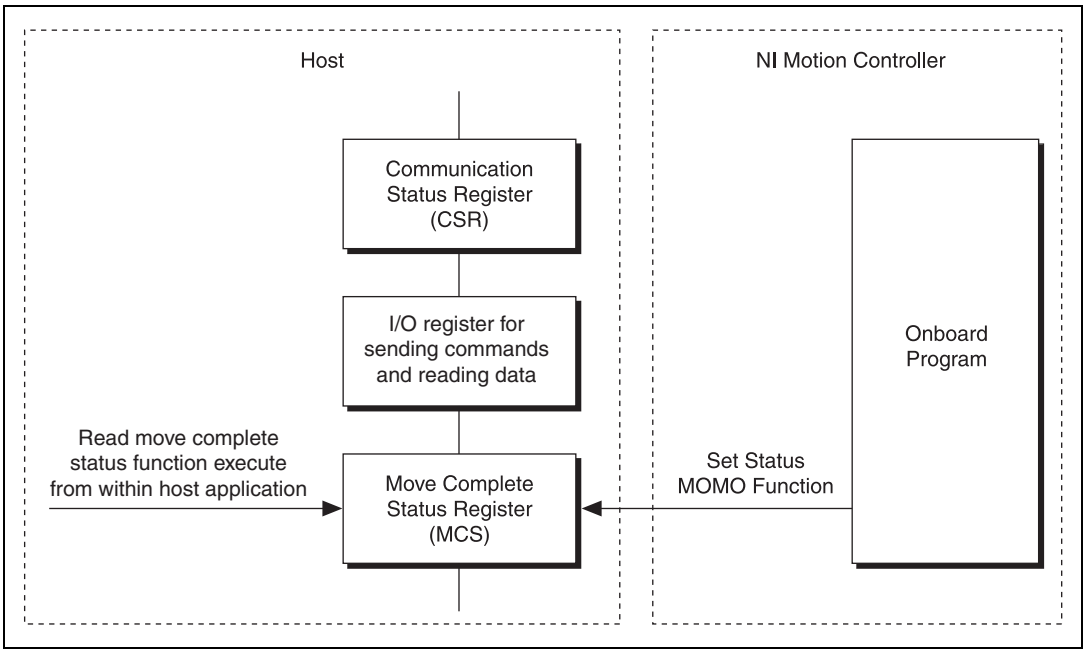


Figure 14-15. Synchronizing Host Applications with Onboard Programs

For example, consider a host application that reads an onboard variable that has been updated by an onboard program. Use the algorithm in Figure 14-16 to synchronize the host application with an onboard program, and read an onboard variable that has been updated by an onboard program.

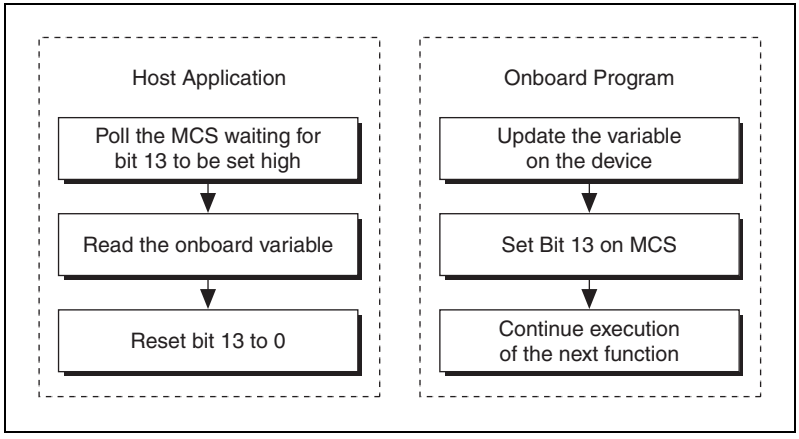
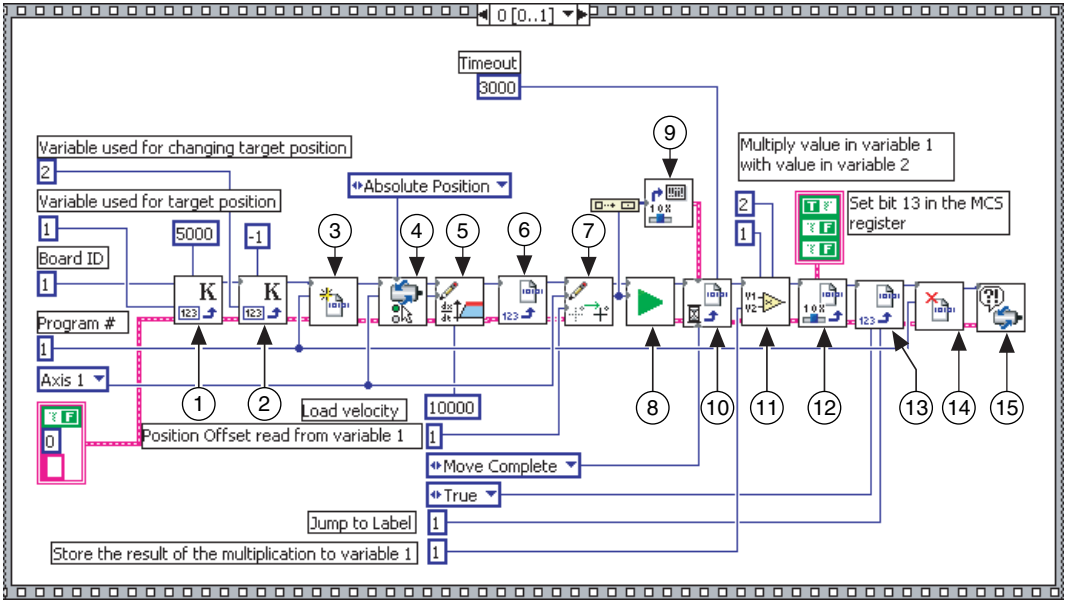


Figure 14-16. Synchronization Algorithm

LabVIEW Code

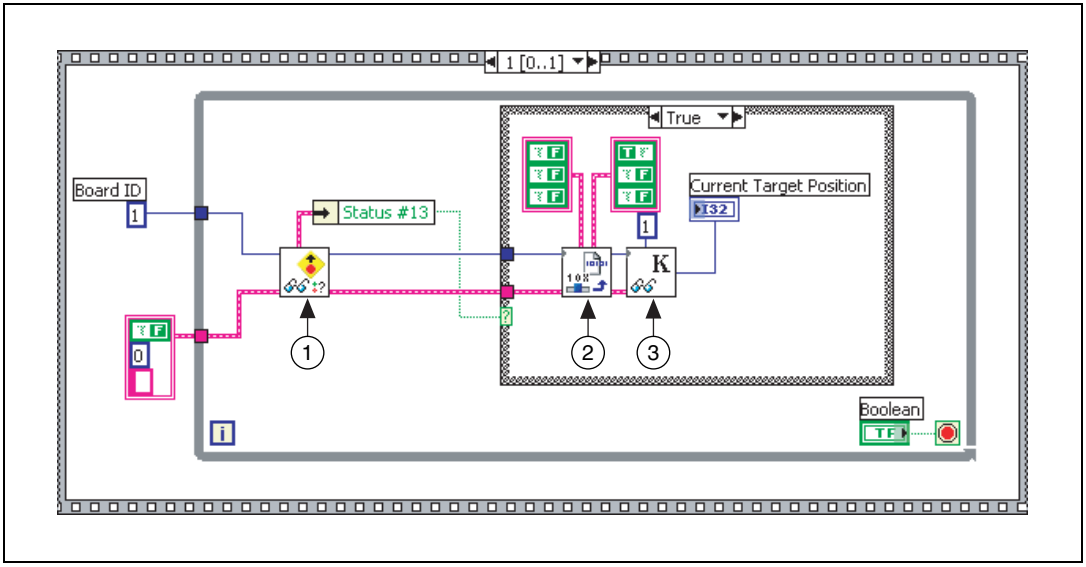
This example moves axis 1 between target positions of 5000 and –5000. The host reads the target position only after the move has completed, and the new target position has been calculated. Figure 14-17 shows the code that runs as an onboard program.



1	Load Constant to Variable	6	Insert Program Label	11	Multiply Variables
2	Load Constant to Variable	7	Load Target Position	12	Set User Status MOMO
3	Begin Program Storage	8	Start Motion	13	Jump to Label on Condition
4	Set Operation Mode	9	Select MOMO	14	End Program Storage
5	Load Velocity	10	Wait on Condition	15	Motion Error Handler

Figure 14-17. Synchronization Onboard Code in LabVIEW

Figure 14-18 shows the code that runs on the host.



1	Read Move Complete Status	2	Set User Status MOMO	3	Read Variable

Figure 14-18. Synchronization Host Code in LabVIEW

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.

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 targetPosition;
    i32 multiplier;
    u16 axisStatus;
    u16 moveCompleteStatus;

    //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 = NIMC_AXIS1;

    // Set the move length
    targetPosition = 5000;

    // Set the multiplier
    multiplier = -1;

    //-----------------------------------------------
    // Onboard program. This onboard program 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 //the 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_event(boardID, 0, NIMC_WAIT,
NIMC_CONDITION_MOVE_COMPLETE,
(u8)(1<<axis)/*Indicates axis to wait on*/, 0,
NIMC_MATCH_ALL, 3000 /*time out*/, 0);
CheckError;

// Multiply variable 1 (target position) with 2
(multiplier)

// Save the result in variable 1 - this calculates
the negative of //last target position
err = flex_mult_vars(boardID, 1, 2, 1);
CheckError;

// Set the 13th bit in the move complete status
register so that //the host 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_on_event (boardID, 0,
NIMC_CONDITION_TRUE, 0, 0, NIMC_MATCH_ALL, 1/*label
number*/);
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 //status
    err = flex_read_axis_status_rtn(boardID, axis,
    &axisStatus);
    CheckError;

    // Read the communication status register and
    check the modal //errors
    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 bit //is set; reset the bit and read the
    target position.
    err = flex_read_mcs_rtn(boardID,
    &moveCompleteStatus);
    CheckError;
    if(moveCompleteStatus & (1<<13)){
          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 & NIMC_AXIS_OFF_BIT));
    //Exit on move complete/following error/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 ID, and the
            error code of the //modal error from the
            error stack on the device
            flex_read_error_msg_rtn(boardID,&commandI
            D,&resourceID, &errorCode);
            nimcDisplayError(errorCode,commandID,res
            ourceID);
            //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

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

Algorithm

Figure 14-19 shows an onboard program algorithm that checks the I/O line state to determine which onboard subroutine to execute.

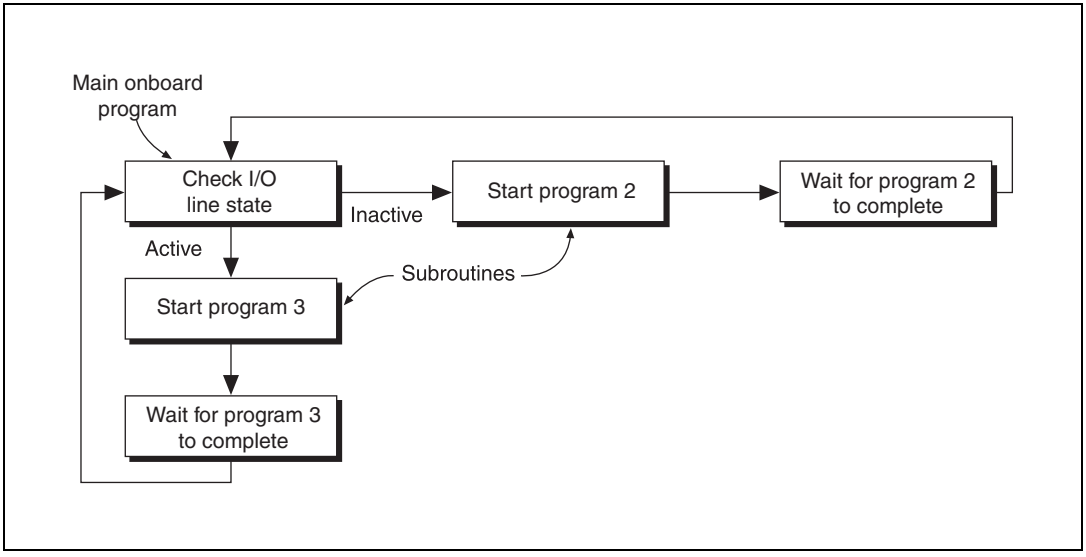
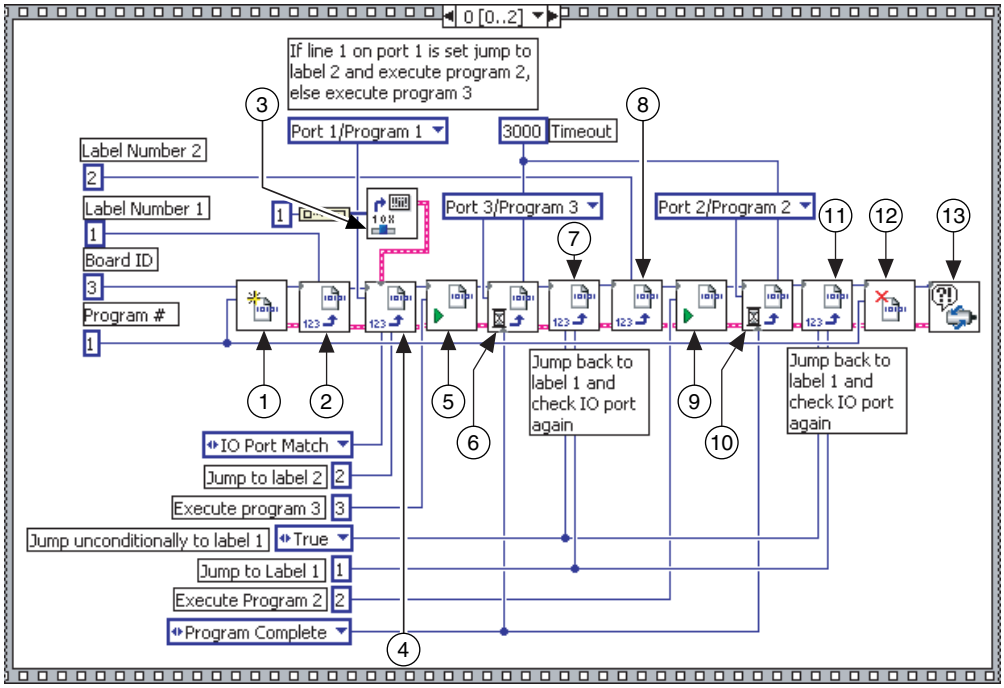


Figure 14-19. Onboard Subroutine Algorithm

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 to counter clockwise.

LabVIEW Code

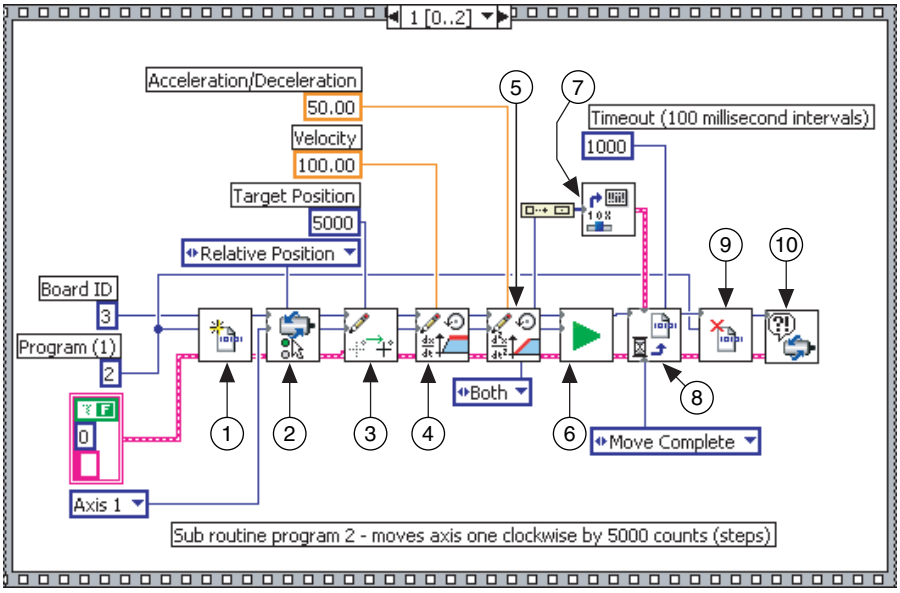
Figure 14-20 shows the main onboard program used to determine the subroutine call.



1	Begin Program Storage	6	Wait on Condition	10	Wait on Condition
2	Insert Program Label	7	Jump to Label on Condition	11	Jump to Label on Condition
3	Select MOMO	8	Insert Program Label	12	End Program Storage
4	Jump to Label on Condition	9	Run Program	13	Motion Error Handler
5	Run Program				

Figure 14-20. Onboard Subroutine Call Using LabVIEW

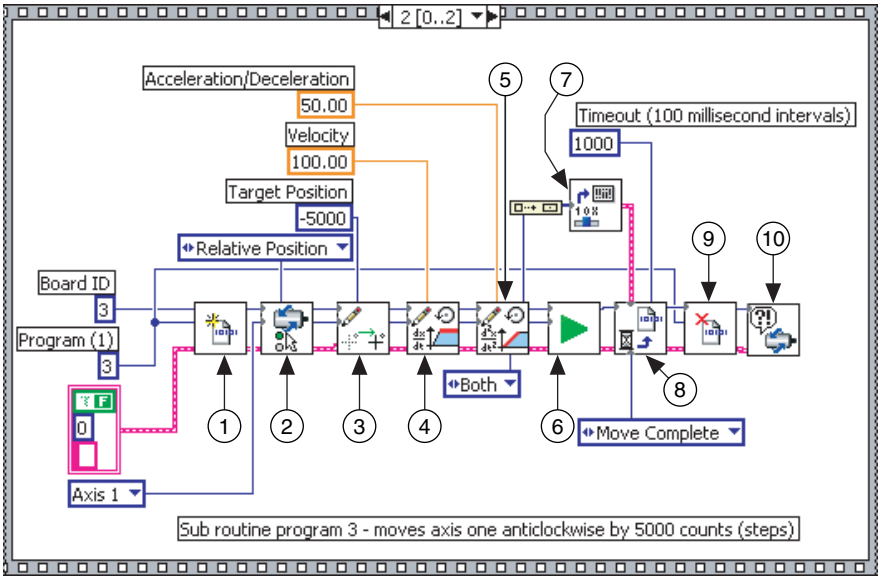
Figure 14-21 shows the subroutine that causes the motor to rotate clockwise.



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				

Figure 14-21. Clockwise Subroutine Using LabVIEW

Figure 14-22 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				

Figure 14-22. Counter Clockwise Subroutine Using LabVIEW

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 = NIMC_AXIS1;

    //-----------------------------------------------
    // Onboard program 2. This onboard program moves axis
    one clockwise //5,000 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 5,000
    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_event(boardID, 0, NIMC_WAIT,
NIMC_CONDITION_MOVE_COMPLETE, 2/*Indicates axis 1*/,
0, NIMC_MATCH_ALL, 1000 /*time out*/, 0);
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(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_event(boardID, 0, NIMC_WAIT,
NIMC_CONDITION_MOVE_COMPLETE, 2/*Indicates axis 1*/,
0, NIMC_MATCH_ALL, 1000 /*time out*/, 0);
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_on_event (boardID, NIMC_IO_PORT1,
NIMC_CONDITION_IO_PORT_MATCH, 2/*Indicates line 1*/,
0, NIMC_MATCH_ALL, 2/*label number*/);
CheckError;

// If the above jump failed, 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_event(boardID, 3 /*program #*/,
NIMC_WAIT, NIMC_CONDITION_PROGRAM_COMPLETE, 0, 0,
NIMC_MATCH_ALL, 1000 /*time out*/, 0);
CheckError;

// Jump unconditionally to label 1 and check IO line
again
err = flex_jump_on_event (boardID, 0,
NIMC_CONDITION_TRUE, 0, 0, NIMC_MATCH_ALL, 1/*label
number*/);
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_event(boardID, 2 /*program #*/,
    NIMC_WAIT, NIMC_CONDITION_PROGRAM_COMPLETE, 0, 0,
    NIMC_MATCH_ALL, 1000 /*time out*/, 0);
    CheckError;

    // Jump unconditionally to label 1 and check IO line
    again
    err = flex_jump_on_event (boardID, 0,
    NIMC_CONDITION_TRUE, 0, 0, NIMC_MATCH_ALL, 1/*label
    number*/);
    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 ID, and the
            error code of the //modal error from the
            error stack on the device
            flex_read_error_msg_rtn(boardID,&commandI
            D,&resourceID, &errorCode);
            nimcDisplayError(errorCode,commandID,res
            ourceID);

            //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

You can configure the onboard program to start automatically without calling the Run Program function. 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 function. 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

Use the Load Program Time Slice function 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.

Part IV

Creating Applications Using NI-Motion

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

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

- *Scanning*
- *Rotating Knife*

15

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 connecting several straight-line move segments.
- Use blending to perform the scan in a single continuous move.
- 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.

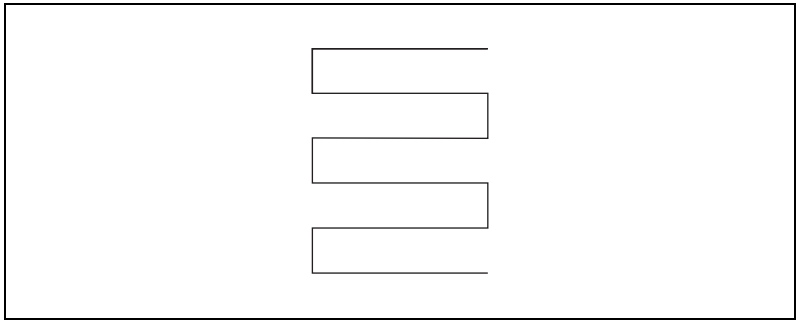


Figure 15-1. Raster Scanning Path

Raster Scanning Using Straight Lines Algorithm

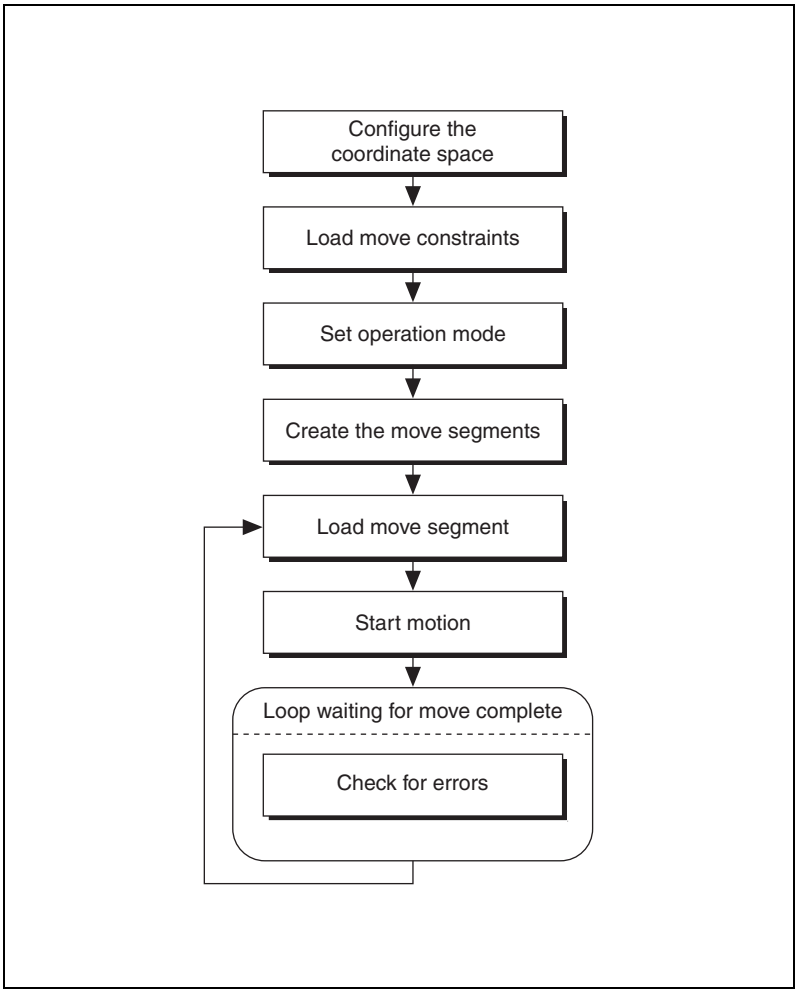
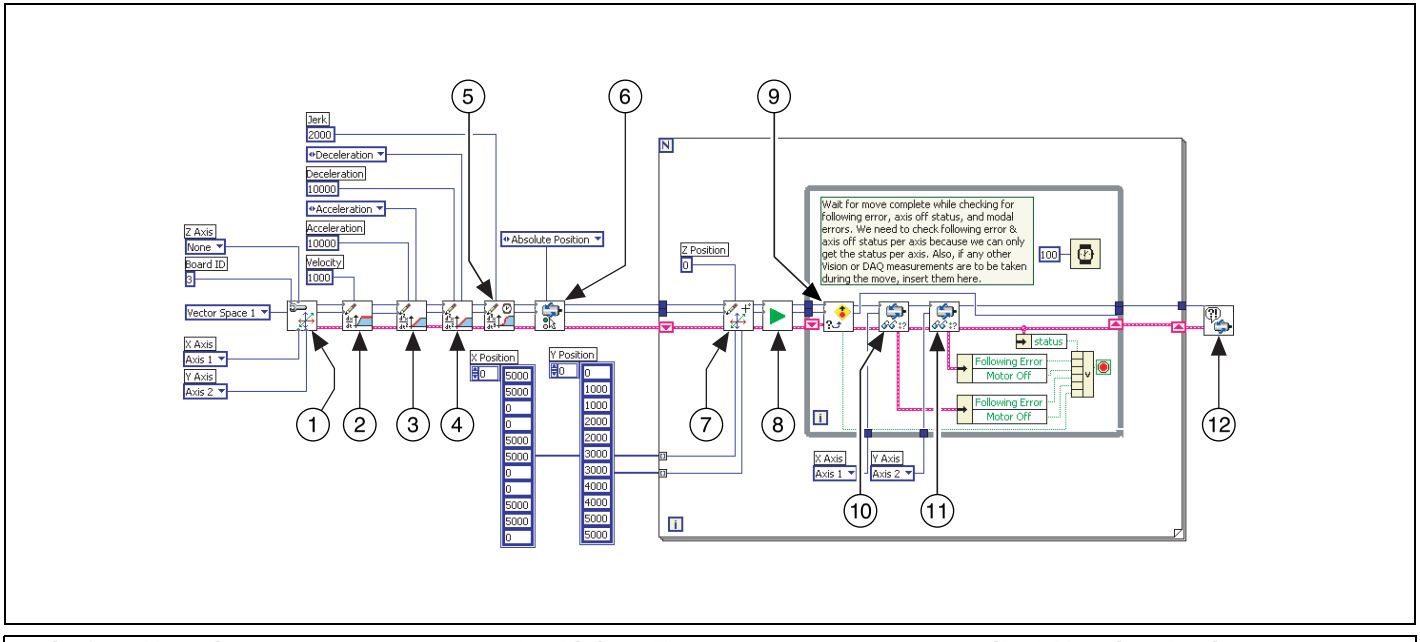


Figure 15-2. Raster Scanning Using Straight Lines Algorithm

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.

LabVIEW Code



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

Figure 15-3. Scanning Using LabVIEW

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.

```
# define d_numberOfSegments
// 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;
    u32 i;
    i32 xPosition[d_numberOfSegments] = {5000, 5000, 0,
    0, 5000, 5000, 0, 0, 5000, 5000, 0};
    i32 yPosition[d_numberOfSegments] = {0, 1000, 1000,
    2000, 2000, 3000, 3000, 4000, 4000, 5000, 5000};

    //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 vector space number
    vectorSpace = NIMC_VECTOR_SPACE1;

    // Configure a 2D vector space comprised of axes 1
    and 2
    err = flex_config_vect_spc(boardID, vectorSpace,
    NIMC_AXIS1, NIMC_AXIS2, NIMC_AXIS3);
    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_ACCELERATION, 100000, 0xFF);
    CheckError;
```

```
// Set the deceleration for the move (in
counts/sec^2)
err = flex_load_acceleration(boardID, vectorSpace,
NIMC_DECELERATION, 100000, 0xFF);
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_POSITION);
CheckError;

// Load the straight-line segments one by one
for (i=0; i<d_numberOfSegments; i++){

    //Load Target Position
    err = flex_load_vs_pos(boardID, vectorSpace,
    xPosition[i], yPosition[i], 0, 0xFF);
    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, &moveComplete);
          CheckError;

          // Check the following error/axis off
          status for axis 1
          err = flex_read_axis_status_rtn(boardID,
          NIMC_AXIS1, &status);
          CheckError;
          axisStatus |= status;

          // Check the following error/axis off
          status for axis 2
          err = flex_read_axis_status_rtn(boardID,
          NIMC_AXIS2, &status);
          CheckError;
          axisStatus |= status;
```

```
                        //Read the communication status register
                        and check the modal //errors
                        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 (!moveComplete && !(axisStatus &
                  NIMC_FOLLOWING_ERROR_BIT) && !(axisStatus &
                  NIMC_AXIS_OFF_BIT)); //Exit on move
                  complete/following //error/axis off
                  if( (axisStatus & NIMC_FOLLOWING_ERROR_BIT) ||
                  (axisStatus & NIMC_AXIS_OFF_BIT) ){
                        break;//Break out of the for loop because
                        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 ID, and the
                        error code of the //modal error from the
                        error stack on the device
                        flex_read_error_msg_rtn(boardID,&commandI
                        D,&resourceID, &errorCode);
                        nimcDisplayError(errorCode,commandID,res
                        ourceID);
                        //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
      }
```

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. Figure 15-4 shows the path of the blended move segments.

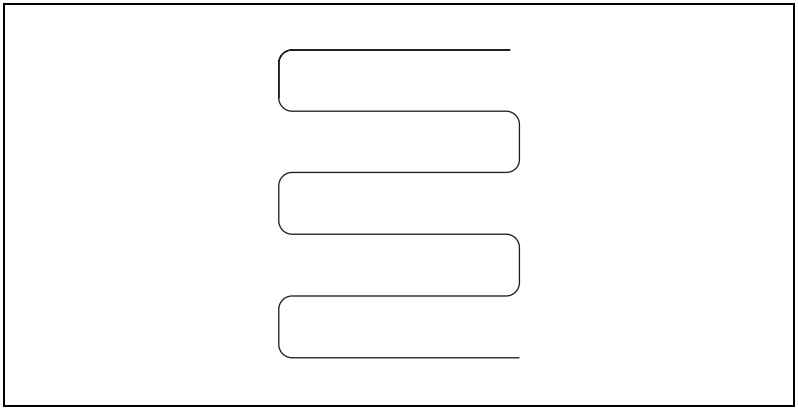


Figure 15-4. Blended Raster Scanning Path

Refer to Chapter 9, *Blending Moves*, for information about using blending with NI-Motion.

Raster Scanning Using Blended Straight Lines Algorithm

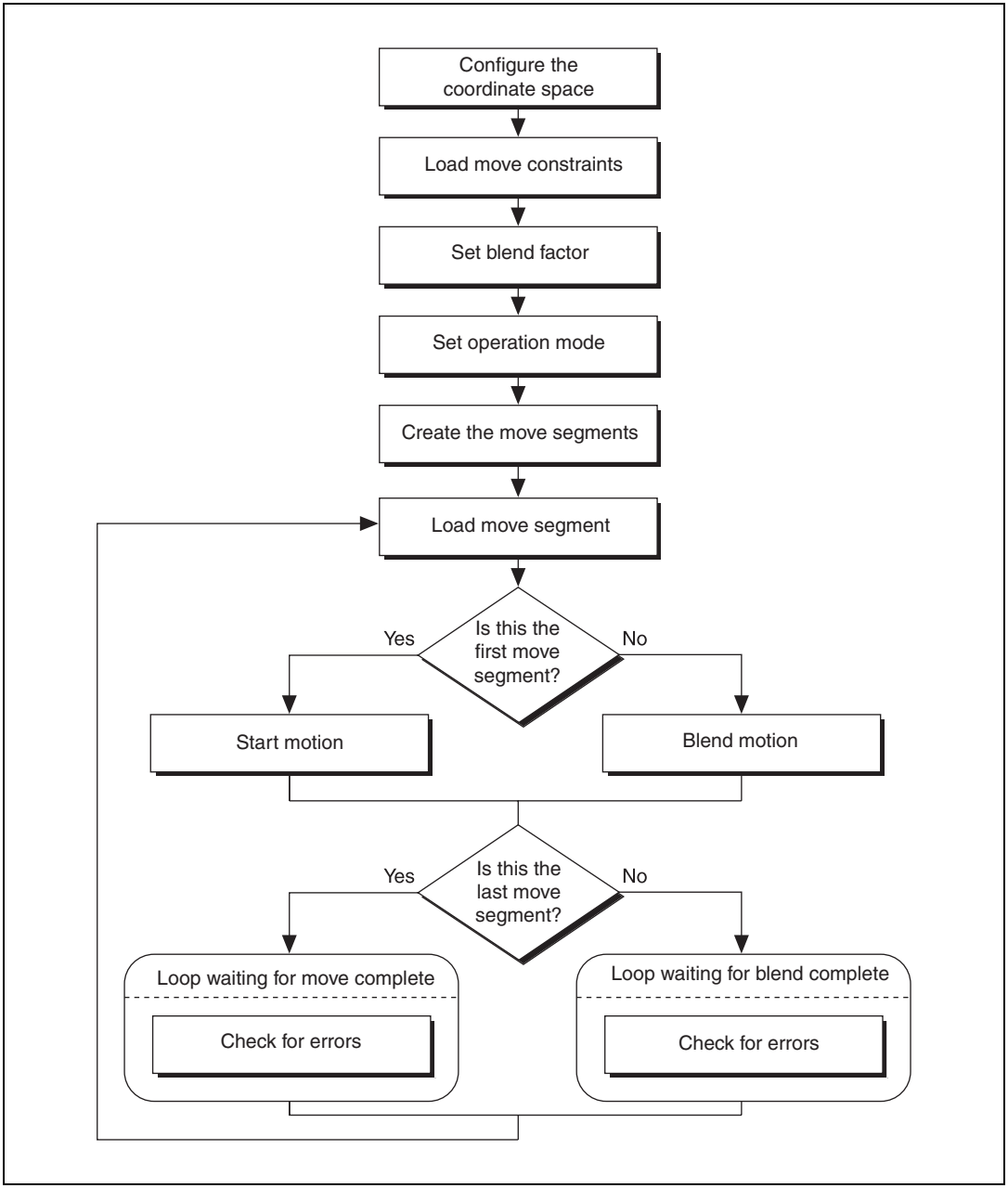
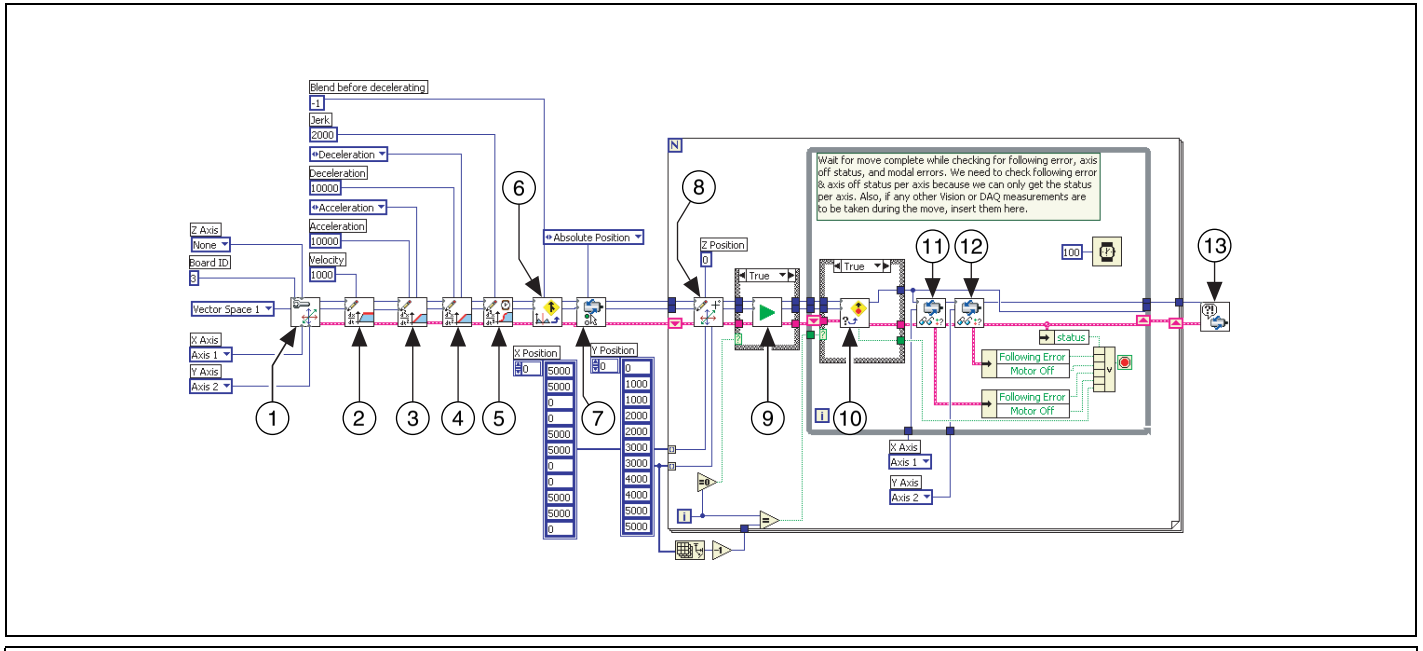


Figure 15-5. Raster Scanning Using Blended Straight Lines Algorithm

LabVIEW Code



1	Configure Vector Space	6	Load Blend Factor	10	Check Move Complete Status
2	Load Velocity	7	Set Operation Mode	11	Read per Axis Status
3	Load Acceleration/Deceleration	8	Load Vector Space Position	12	Motion Error Handler
4	Load Acceleration/Deceleration	9	Start Motion	13	Motion Error Handler
5	Load S-Curve Time				

Figure 15-6. Scanning Using Blending

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.

```
#define d_numberOfSegments
// 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[d_numberOfSegments] = {5000, 5000, 0,
    0, 5000, 5000, 0, 0, 5000, 5000, 0};
    i32 yPosition[d_numberOfSegments] = {0, 1000, 1000,
    2000, 2000, 3000, 3000, 4000, 4000, 5000, 5000};

    //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 vector space number
    vectorSpace = NIMC_VECTOR_SPACE1;

    // Configure a 2D vector space comprised of axes 1
    and 2
    err = flex_config_vect_spc(boardID, vectorSpace,
    NIMC_AXIS1, NIMC_AXIS2, NIMC_AXIS3);
    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_ACCELERATION, 100000, 0xFF);
    CheckError;
```

```
// Set the deceleration for the move (in
counts/sec^2)
err = flex_load_acceleration(boardID, vectorSpace,
NIMC_DECELERATION, 100000, 0xFF);
CheckError;

// Set the jerk or s-curve in sample periods
err = flex_load_scurve_time(boardID, vectorSpace,
100, 0xFF);
CheckError;

// Load the blending factor
err = flex_load_blend_fact(boardID, vectorSpace, -1,
0xFF);
CheckError;

// Set the operation mode to absolute position
err = flex_set_op_mode(boardID, vectorSpace,
NIMC_ABSOLUTE_POSITION);
CheckError;

// Load the straight-line segments one by one
for (i=0; i<d_numberOfSegments; i++){
    //Load Target Position
    err = flex_load_vs_pos(boardID, vectorSpace,
    xPosition[i], yPosition[i], 0, 0xFF);
    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==d_numberOfSegments-1){
                  // Check the move complete status
                  err =
                  flex_check_move_complete_status(bo
                  ardID, vectorSpace, 0, &complete);
```

```
                CheckError;
            }else{
                // Check the blend complete status
                err =
                flex_check_blend_complete_status(b
                oardID, vectorSpace, 0,
                &complete);
                CheckError;
            }
            // Check the following error/axis off
            status for axis 1
            err = flex_read_axis_status_rtn(boardID,
            NIMC_AXIS1, &status);
            CheckError;
            axisStatus |= status;
            // Check the following error/axis off
            status for axis 2
            err = flex_read_axis_status_rtn(boardID,
            NIMC_AXIS2, &status);
            CheckError;
            axisStatus |= status;
            //Read the communication status register
            and check the modal //errors
            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 (!complete && !(axisStatus &
        NIMC_FOLLOWING_ERROR_BIT) && !(axisStatus &
        NIMC_AXIS_OFF_BIT));
        //Exit on move complete/following error/axis off
        if( (axisStatus & NIMC_FOLLOWING_ERROR_BIT) ||
        (axisStatus & NIMC_AXIS_OFF_BIT) ){
            break;//Break out of the for loop because
            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 ID, and the
                error code of the //modal error from the
                error stack on the device
                flex_read_error_msg_rtn(boardID,&commandI
                D,&resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,res
                ourceID);
                //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 your given points using a cubic spline algorithm. Figure 15-7 shows the scanning path used in the example that follows the figure.

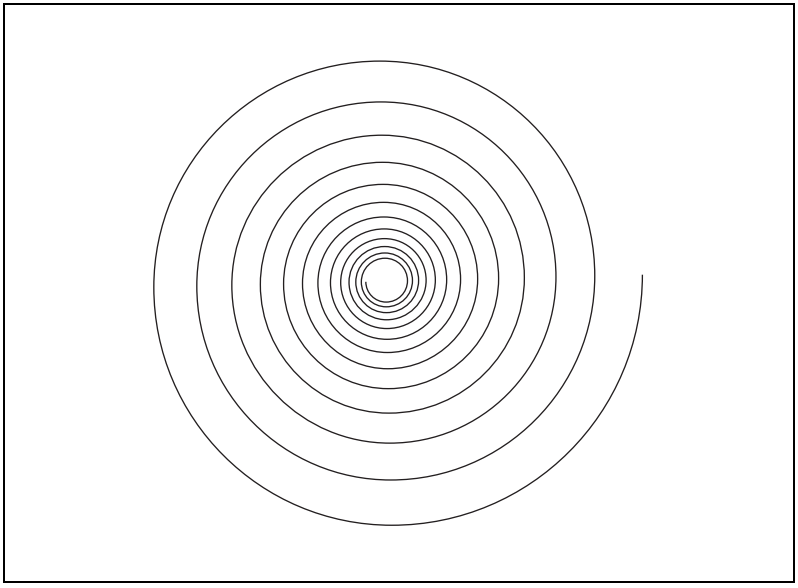


Figure 15-7. User-Defined Scanning Path

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 Chapter 7, *Contoured Moves*, for information about using contoured moves with NI-Motion.

User-Defined Scanning Path Algorithm

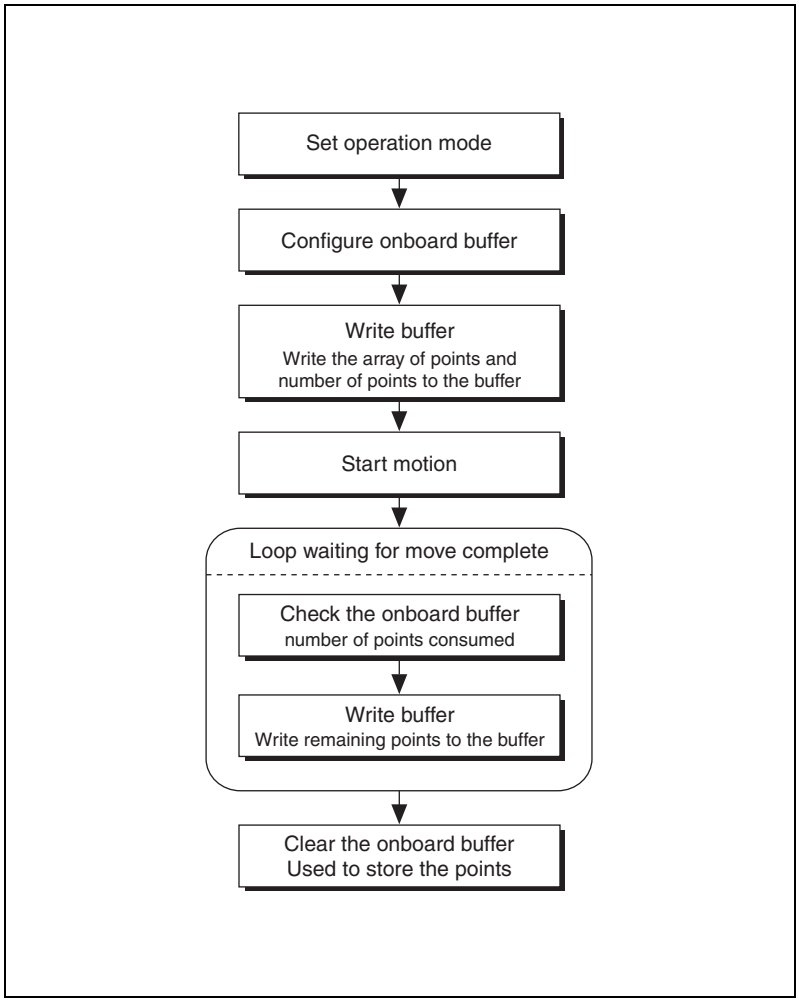
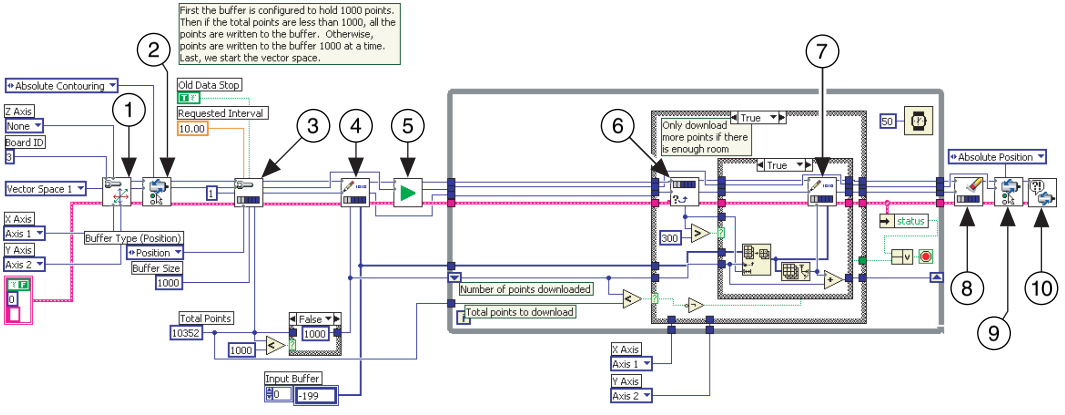


Figure 15-8. User-Defined Scanning Path Algorithm

LabVIEW Code



1	Configure Vector Space	5	Start Motion	8	Clear Buffer
2	Set Operation Mode	6	Check Buffer	9	Set Operation Mode
3	Configure Buffer	7	Write Buffer	10	Motion Error Handler
4	Write Buffer				

Figure 15-9. Scanning Using Contouring

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;// Temporary copy of status
    u16 moveComplete;// Move complete status
    i32 i;
    i32 points[1994] = NIMC_SPIRAL_ARRAY;// Array of 2D
    points to move
    u32 numPoints = 1994;// Total number of points to
    contour through
    i32 bufferSize = 1000;// The size of the buffer to
    allocate on the //motion controller
    f64 actualInterval;// The interval at which the
    motion controller can //really contour
    i32* downloadData = NULL;// The temporary array that
    is created to //download the points to the motion
    controller
    u32 currentDataPoint = 0;// Indicates the next point
    in the points //array that is to be downloaded
    i32 backlog;// Indicates the available space to
    download more //points
    u16 bufferState;// Indicates the state of the onboard
    buffer
    u32 pointsDone;// Indicates the number of points that
    have been //consumed
    u32 dataCopied = 0;// Keeps track 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 vector number
```

```
vectorSpace = NIMC_VECTOR_SPACE1;

// Configure a 2D vector space comprised of axes 1
and 2
err = flex_config_vect_spc(boardID, vectorSpace,
NIMC_AXIS1, NIMC_AXIS2, NIMC_AXIS3);
CheckError;

// Set the operation mode to absolute position
err = flex_set_op_mode(boardID, vectorSpace,
NIMC_ABSOLUTE_CONTOURING);
CheckError;

// Configure buffer on motion controller memory (RAM)
// Notice requested time interval is hardcoded to 10
milliseconds
err = flex_configure_buffer(boardID, 1 /*buffer
number*/, vectorSpace, NIMC_POSITION_DATA,
bufferSize, numPoints, NIMC_TRUE, 10,
&actualInterval);

// Send the first 1000 points of the data
downloadData = malloc(sizeof(i32)*bufferSize);
for(i=0;i<bufferSize;i++){downloadData[i] =
points[i];currentDataPoint++;}

err = flex_write_buffer(boardID, 1/*buffer number*/,
bufferSize, 0, downloadData, 0xFF);
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 //every 50 milliseconds
    Sleep(50);

    // Check to see if there are more points to
    download
    if(currentDataPoint < numPoints){
           err = flex_check_buffer_rtn(boardID,
           1/*buffer number*/, &backlog,
           &bufferState, &pointsDone);
           CheckError;
```

```
        if(backlog >= 300){
                downloadData =
                malloc(sizeof(i32)*backlog);
                dataCopied = 0;
                for(i=0;i<backlog;i++){
                        if(currentDataPoint >
                        numPoints) break;
                        downloadData[i] =
                        points[currentDataPoint];
                        currentDataPoint++;
                        dataCopied++;
                }
                err = flex_write_buffer (boardID, 1
                /*buffer number*/, dataCopied, 0,
                downloadData, 0xFF);
                free(downloadData);
                downloadData = NULL;
                CheckError;
        }
}

// Check the move complete status
err = flex_check_move_complete_status (boardID,
vectorSpace, 0, &moveComplete);
CheckError;

if(moveComplete) break;

// Check for axis off status/following error or
any modal //errors

//Read the communication status register and
check the modal //errors
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
err = flex_read_axis_status_rtn(boardID,
NIMC_AXIS1, &status);
CheckError;
axisStatus |= status;
```

```
        err = flex_read_axis_status_rtn(boardID,
        NIMC_AXIS2, &status);
        CheckError;
        axisStatus |= status;

        if( (axisStatus & NIMC_FOLLOWING_ERROR_BIT) ||
        (axisStatus & NIMC_AXIS_OFF_BIT) ){
              break;//Break out of the for loop because
              an axis was killed
        }
    }

    //Set the mode back to absolute mode to get the motion
    controller out of //contouring mode
    err = flex_set_op_mode(boardID, vectorSpace,
    NIMC_ABSOLUTE_POSITION);
    CheckError;

    // Free the buffer allocated on the motion 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 ID, and the
              error code of the //modal error from the
              error stack on the device
              flex_read_error_msg_rtn(boardID,&commandI
              D,&resourceID, &errorCode);
              nimcDisplayError(errorCode,commandID,res
              ourceID);

              //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
}
```

16

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.

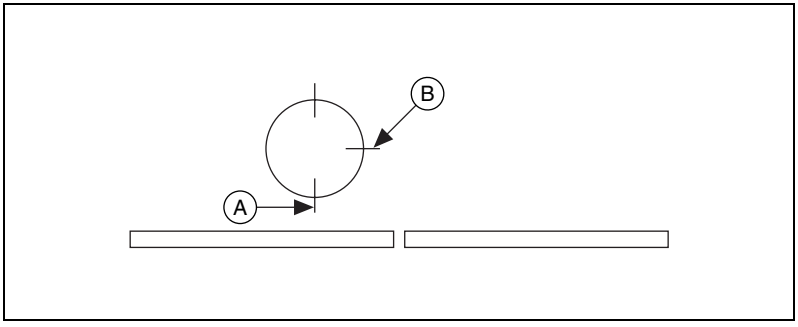
To simplify this example, assume that the length of the cut is equal to the circumference of the knife. Under ideal conditions, the mark should be read when the blade is at position A, as shown in Figure 16-1. Therefore, the motor should move one revolution without any correction before causing the cut.

Tip Refer to Chapter 10, *Electronic Gearing and Camming*, 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 Figure 16-1.



A Synchronization Point B Correction Point

Figure 16-1. Rotating Knife

Algorithm

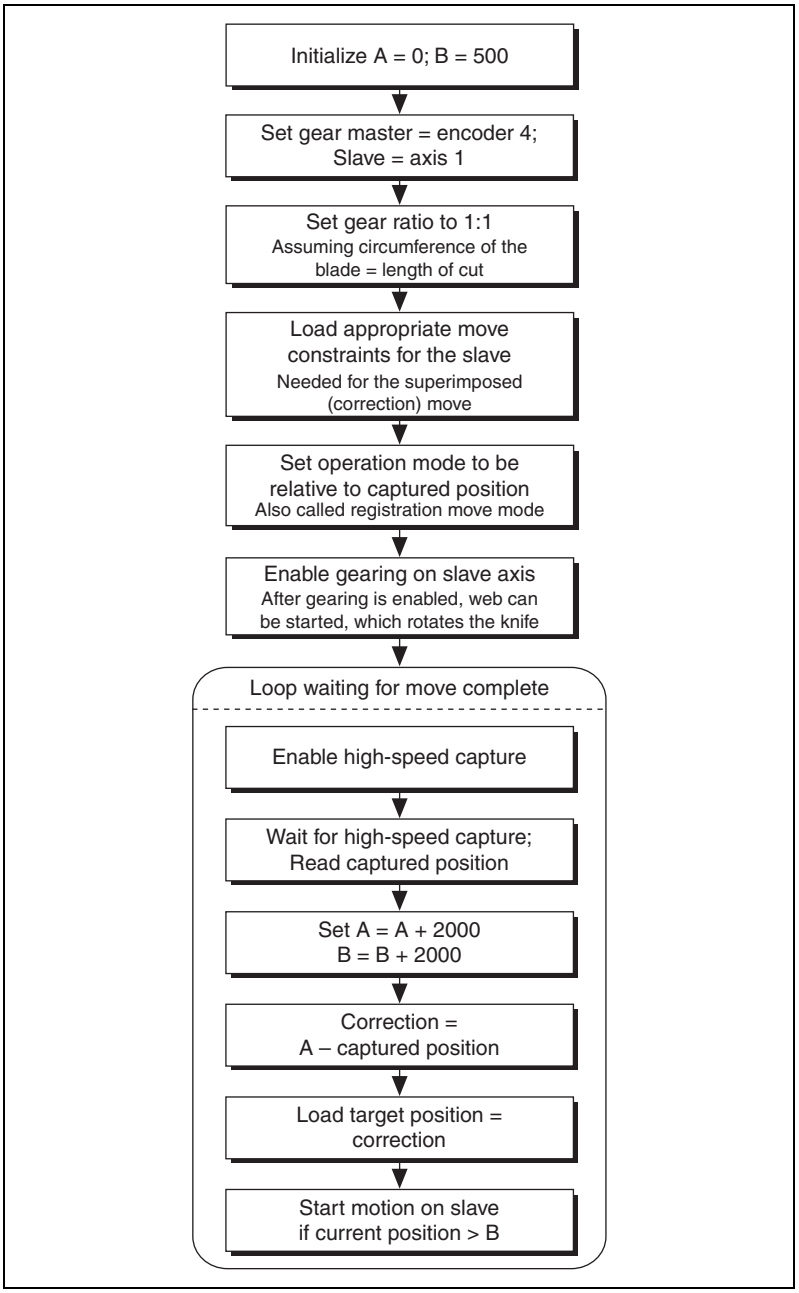
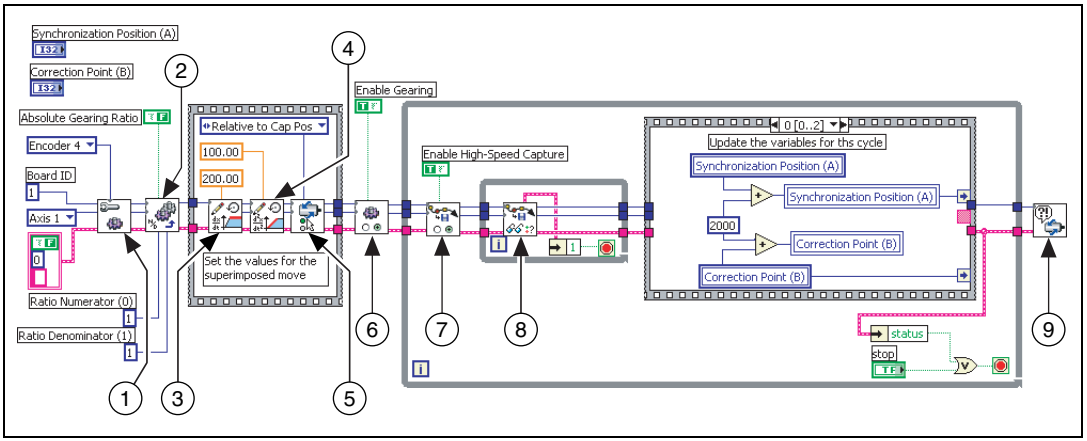


Figure 16-2. Rotating Knife Application Algorithm

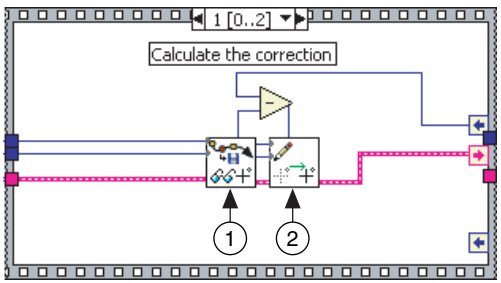
LabVIEW Code



1	Configure Gear Master	6	Enable Gearing Single Axis
2	Load Gear Ratio	7	Enable High-Speed Capture
3	Load Velocity in RPM	8	Read High-Speed Capture Status
4	Load Accel/Decel in RPS/s	9	Motion Error Handler
5	Set Operation Mode		

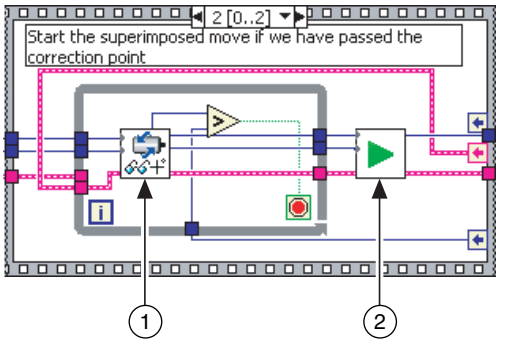
Figure 16-3. Rotating Knife Application Using LabVIEW

Figures 16-4 and 16-5 show the remaining cases for the block diagram in Figure 16-3.



1	Read Captured Position	2	Load Target Position

Figure 16-4. Figure 16-3 Sequence Structure 1



1	Read Position	2	Start Motion

Figure 16-5. Figure 16-3 Sequence Structure 2

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
    i32 synchronizationPosition = 0; // Synchronization
    position
    i32 correctionPoint = 500;// Point where the correction
    can be //applied
    i32 cyclePosition = 2000;// One revolution is 2,000
    counts
    i32 currentPosition;// The current slave position
    i32 capturedPosition;// The position at which the
    trigger happens
    u16 axisStatus;

    //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 = NIMC_AXIS1;
// 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_GEARING, 1/*ratioNumerator*/,
1/*ratioDenominator*/, 0xFF);
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_CAPTURE);
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_TRUE);
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 //off status
        err = flex_read_axis_status_rtn(boardID,
        slaveAxis, &axisStatus);
        CheckError;
        // Read the communication status register and
        check the modal //errors
        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)
&& !(axisStatus & NIMC_FOLLOWING_ERROR_BIT) &&
!(axisStatus & NIMC_AXIS_OFF_BIT));
//Exit on following error/axis off & high-speed
capture
if((axisStatus & NIMC_FOLLOWING_ERROR_BIT) ||
(axisStatus & NIMC_AXIS_OFF_BIT)){
        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,
&capturedPosition);
CheckError;
// Load the target position for the registration
//(superimposed) move
err = flex_load_target_pos(boardID, slaveAxis,
(synchronizationPosition - capturedPosition),
0xFF);
CheckError;
// Wait until the axis has passed the correction
point before //applying the correction
currentPosition = 0;
while (currentPosition < correctionPoint){
        err = flex_read_pos_rtn(boardID, slaveAxis,
        &currentPosition);
        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 ID, and the
                error code of the //modal error from the error
                stack on the device
                flex_read_error_msg_rtn(boardID,&commandID,
                &resourceID, &errorCode);
                nimcDisplayError(errorCode,commandID,resou
                rceID);
                //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
}
```

A

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

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:

$$\text{counts per electrical cycle} = \frac{\text{counts per revolution} \times 2}{\text{number of poles}}$$

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:

$$\text{commutation frequency} = \frac{\text{counts per electrical cycle}}{\text{PID rate} \times 6}$$

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.
5. 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.

B

Initializing the Controller Programmatically

You can initialize the motion controller from within a LabVIEW, Visual Basic, or C/C++ program, in addition to initializing controllers in Measurement & Automation Explorer (MAX).

Refer to Table B-1 for the steps you must take to initialize a controller programmatically and the functions and VIs you use for each step.

Table B-1. Steps for Programmatically Initializing Controllers

Step	Function and/or VI
1. Clear the power-up state.	Use the Clear Power Up Status VI.
2. Review any errors that occurred on the controller to determine how best to handle them.	Use the Read Error Message VI.
3. Make sure all axes are stopped and disabled.	Use the Stop Motion and Enable Axes VIs. In the Stop Motion VI, set **Stop Type (Decel)** set to Halt. In the Enable Axes VI, you must set **Axis Bitmap** to **False** for each axis you want to disable.
4. Unconfigure vector spaces.	Use the Configure Vector Space VI with the **X Axis**, **Y Axis**, and **Z Axis** terminals set to **None**.
5. Configure resources for axes.	Use the Configure Axis Resources VI.
6. Load all axis configuration options you want to use.	Use the Axis & Resource Configuration palette.
7. Initialize encoders and ADCs, as appropriate.	Use the appropriate VIs on the Analog & Digital I/O palette.
8. Enable the axes, but leave them deactivated.	Use the Enable Axes VI.
9. Load the appropriate control loop parameters.	Use the Load Advanced Control Parameter and either Load Single PID Parameter or Load All PID Parameters VIs.

Table B-1. Steps for Programmatically Initializing Controllers (Continued)

Step	Function and/or VI
10. Call halt on all axes to activate them.	Use the Stop Motion VI with S**top Type (Decel)** set to **Halt stop**.
11. Configure capture and compare settings.	Use the VIs on the Motion I/O palette to configure the capture and compare settings.
12. Configure the following optional settings: • Configure trajectory settings • Configure find reference settings • Configure DIO settings • Configure PWM settings • Gearing	Use the following palettes or VIs to configure the optional settings: • Trajectory Control palette • Find Reference palette • Analog & Digital I/O palette • Configure PWM Output and Load PWM Duty Cycle VIs • Gearing palette

C

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 (73*xx* controller support)
  - **NI SoftMotion Controller 1.0»CANopen Drive Support**

For an RT system, you can configure an NI motion controller on a remote PXI chassis through the remote configuration feature of MAX. You must install NI-Motion onto the remote system to use RT. Then, program the RT NI-Motion application exactly the way you would program any other NI-Motion application.

Complete the following steps to install NI-Motion onto the remote system.

1. Install one of the following software options onto the host system:
   - NI-Motion (73*xx* controller support)
   - **NI SoftMotion Controller 1.0»CANopen Drive Support**

Tip Refer to the *Getting Started: NI SoftMotion Controller for Copley CANopen Drives* manual for information about this product.

2. Launch MAX.
3. Expand the **Remote Systems** tree.
4. Highlight the system on which to install NI-Motion.
5. Select the **Software** tab.

6. If NI-Motion is not already installed, right-click within the dialog box and select **Install Software**. A dialog appears that lets you select what to download. Make sure the checkbox next to NI-Motion RT is selected.
7. Click **OK** and wait for the software to download.

After the software downloads onto the remote system, complete the following steps to configure the remote NI motion controller.

1. Wait for the remote system to reboot so MAX is able to communicate with it.
2. Expand the **Remote Systems** tree and then expand the **Devices and Interfaces** tree.
3. Right-click the remote motion controller icon and select **Map to Local Machine**. This assigns a local board ID to the remote motion controller in the host system.

   Mapping the remote controller into the local system allows you to configure the controller through MAX exactly as you would a controller that is in the host system. You can initialize the controller, download firmware, and use the interactive and configuration panels exactly as you would on a controller installed in the host machine. You also can write VIs using the remote motion controller through the local board ID assigned to it.

   This allows you to write and debug your VIs on the host, and then download them to the remote system when you are ready. All you need to change is the board ID in your VI from the locally assigned Board ID to the ID assigned by the remote system.
4. Browse to **Devices and Interfaces** under **My System**, where there is a shortcut icon next to a new controller name.

   For example, if the motion controller on the remote system is a PXI-7334, and the remote system has an IP address of `123.456.789.000`, then the shortcut device would show a name like **PXI-7334 (X) on 10.0.58.48 (Y)**.

   X is the board ID assigned to the board by the remote system. Use this board ID for VIs that are downloaded to the remote system through LabVIEW RT.

   Y is the board ID assigned to the remote motion controller by the local system. Use this board ID for any VIs that run on the host and use the remote motion controller.

To remove the mapped motion controller, browse to **My System** under **Device and Interfaces**. Right-click the mapped controller and select **Unmap Remote Device**. You should unmap devices when you no longer need to use them from the host machine.

D

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—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 KnowledgeBase, product manuals, 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 Application Engineers worldwide in the NI Developer Exchange at ni.com/exchange. National Instruments Application 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.
- **Training and Certification**—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.
- **System Integration**—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. Phone numbers for our worldwide offices are listed at the front of this manual. 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.

Glossary

A

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.
active high	Signal is active when its value is high (1).
active low	Signal is active when its value is low (0).
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
axis	Unit that controls a motor or any similar motion or control device.

B

b	bit—One binary digit, either 0 or 1.
base address	Memory address that serves as the starting address for 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.
buffer	Temporary storage for acquired or generated data.
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.
byte	Eight related bits of data, an 8-bit binary number. Also used to denote the amount of memory required to store 1 byte of data.

C

CCW	counter-clockwise—Implies the direction the motor rotates in.
closed-loop	Motion control system that uses a feedback device to provide position and velocity data for status reporting and accurately controlling position and velocity.
CPU	central processing unit
CSR	communications status register
CW	clockwise—Implies the direction the motor rotates in.

D

DAC	digital-to-analog converter
data acquisition	The process of collecting and measuring electrical signals from sensors, transducers, and test probes or fixtures and inputting them to a computer for processing.
digital I/O port	Group of digital input/output signals.
DLL	dynamic link library—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 communicates commands to control a specific motion control device.

E

encoder	Device that translates mechanical motion into electrical signals; used for monitoring position or velocity in a closed-loop system.
encoder resolution	Number of encoder lines between consecutive encoder 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
filter parameters	Indicates the control loop parameter gains (PID gains) for a given axis.
filtering	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	Difference between the instantaneous commanded trajectory position and the feedback position.
full-step	Full-step mode of a stepper motor—For a two-phase motor. this refers to energizing both windings or phases simultaneously.

G

Gnd	ground
GND	ground

H

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.

home switch (input)	Physical position determined by the mechanical system or designer as the reference location for system initialization. Frequently, the home position is also regarded as the zero position in an absolute position frame of reference.
host computer	Computer in which the motion controller is installed, or that is controlling the remote system in which the motion controller is installed.

I

I/O	input/output—The transfer of data to and from a computer system involving communications channels, operator interface devices, and/or motion control interfaces.
ID	identification
index	Marker between consecutive encoder revolutions.
inverting	Polarity of a limit switch, home switch, and so on in *active* state. If these switches are active low, they have inverting polarity.
IRQ	interrupt request

K

k	kilo—The standard metric prefix for 1,000, or 10^3, used with units of measure such as volts, hertz, and meters.
K	kilo—The prefix for 1,024, or 2^{10}, used with B in quantifying data or computer memory.

L

LIFO	last-in, first-out
limit switch/ end-of-travel position (input)	Sensors that alert the control electronics that the physical end of travel is near and that the motion must stop.

M

m	meters
MCS	Move Complete Status
microstep	Proportional control of energy in the coils of a Stepper Motor that allow 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 position	Treat the position as if it is within the range of total quadrature counts per revolution for an axis.

N

noise	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.
non-inverting	Polarity of a limit switch, home switch, and so on, in *active* state. If these switches are active high, they have non-inverting polarity.

O

open collector	Method of output capable of sinking current, but not sourcing current.
open-loop	Refers to a motion control system where no external sensors, or feedback devices, are used to provide position or velocity correction signals.

P

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.
PID	proportional-integral-derivative control loop
PIVff	proportional-integral-velocity feed forward
port	(1) Communications connection on a computer or a remote controller; (2) Digital port, consisting of 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.
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 counts	Encoder line resolution multiplied by four.

R

RAM	random-access memory
relative breakpoint	Sets the position breakpoint for an encoder in relative quadrature counts.
relative position	Destination or target position for motion specified with respect to the current location, regardless of its value.
relative position mode	Position relative to current position.

RPM	revolutions per minute—Units for velocity.
RPSPS or RPS/S	revolutions per second squared—Units for acceleration and deceleration.
RTR	Ready to Receive

S

s	seconds
servo	Specifies an axis that controls a servo motor.
sinusoidal commutation	Method of controlling current in the windings of a 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.
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 profile	Typical motion trajectory, where a motor accelerates up to 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.
TTL	transistor-transistor logic

V

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.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W