Motion Control

NI-Motion™ User Manual

NATIONAL INSTRUMENTS™

May 2003 Edition
Part Number 373367A-01

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 (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949, Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530, China 86 21 6555 7838, Czech Republic 420 2 2423 5774, 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, Greece 30 2 10 42 96 427, India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Malaysia 603 9131 0918, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466, New Zealand 1800 300 800, Norway 47 0 66 90 76 60, Poland 48 0 22 3390 150, Portugal 351 210 311 210, Russia 7 095 238 7139, Singapore 65 6226 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 2 2528 7227, Thailand 662 992 7519, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to `techpubs@ni.com`.

© 2003 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

IMAQ™, LabVIEW™, National Instruments™, NI™, ni.com™, NI Developer Zone™, NI-Motion™, and NI Motion Assistant™ trademarks of National Instruments Corporation.

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

About This Manual

Conventions xiii
Related Documentation xiv

Part I
Introduction

Chapter 1
Introduction to NI-Motion

About NI-Motion 1-1
NI-Motion Architecture 1-1
  Software and Hardware Interaction 1-2
  NI Motion Controller Architecture 1-2
    Physical Architecture 1-2
    Functional Architecture 1-4
Documentation and Examples 1-5

Chapter 2
Creating NI-Motion Applications

Creating a Generic NI-Motion Application 2-1
Adding Measurements to Your NI-Motion Application 2-2

Part II
Configuring Motion Control

Chapter 3
Setting up the Motion Hardware

Motion Control Components 3-1
  Controller 3-1
    Servo Control 3-2
    Step Generation 3-2
  Amplifier 3-3
  UMI Breakout Box 3-4
  Motors and Actuators 3-4
  Encoders and Other Feedback Devices 3-5
  Mechanical System 3-6

Digital I/O 3-6
Analog I/O.......... 3-6
Motion I/O.......... 3-6
Limits.......... 3-6
Home 3-7
Breakpoints.......... 3-7
Trigger Inputs (High-Speed Capture).......... 3-8
Inhibit Output 3-8

Chapter 4
Configuring the Motion System Using MAX

Configuring the System.......... 4-1
Device Resources.......... 4-4
Initialization Settings 4-4
Axis Settings.......... 4-9
Trajectory Settings.......... 4-13
Find Reference Settings.......... 4-14
Digital I/O Settings.......... 4-17
Gearing Settings 4-18
ADC Settings.......... 4-18
Encoder Settings 4-19
PWM Settings.......... 4-19
Synchronization Settings 4-20
Interactive.......... 4-20
1D Interactive 4-20
2D Interactive 4-23
Calibration (734*x* and 735*x*).......... 4-24
Tuning the Servo Motors.......... 4-24
Control Loop 4-25
PID Loop Descriptions 4-27
Dual Loop Feedback.......... 4-30
Velocity Feedback 4-31
NI Motion Controllers with Velocity Amplifiers 4-33
Sinusoidal Commutation for Brushless Servo Motion Control.......... 4-34
Phase Initialization.......... 4-34
Hall Effect Sensors 4-35
Shake and Wake 4-35
Direct Set 4-35
Determining the Counts per Electrical Cycle of the Motor 4-35
Commutation Frequency.......... 4-36
Troubleshooting Hall Effect Sensor Connections.......... 4-36
Initializing the Controller Programmatically.......... 4-37
Using the Motion Controller with RT 4-37

Chapter 5
Testing the Motion System

Testing the Encoders....5-1
Testing the Motors....5-2
  Troubleshooting the Motors....5-3
    Check the Motion Controller....5-3
    Check the Drive....5-3
Testing Limit and Home Switches....5-4
1D Interactive Test....5-4

PART III
Programming NI-Motion

What You Need to Know about Moves....III-2
  Move Profiles....III-2
    Trapezoidal....III-2
    S-Curve....III-3
  Basic Moves....III-3
  Coordinate Space (Vector Space)....III-4
    Multi-Starts versus Coordinate Spaces....III-5
  Trajectory Parameters....III-5
    Floating-Point versus Fixed-Point....III-5
    Time Base....III-6
  Arc Move Limitations....III-11
Timing Your Loops....III-12
  Status Display....III-12
  Graphing Data....III-12
  Event Polling....III-13

Chapter 6
Straight-Line Moves

Position-Based Straight-Line Moves....6-1
  Algorithm....6-1
  C/C++ Code....6-5
Velocity-Based Straight-Line Moves
(Jogging/Velocity Profiling)....6-9
  Algorithm....6-10
  LabVIEW Code....6-12
  C/C++ Code....6-13

Velocity Profiling Using Velocity Override 6-16
Algorithm 6-17
LabVIEW Code 6-18
C/C++ Code 6-19

Chapter 7
Arc Moves

Circular Arcs 7-1
Algorithm 7-3
LabVIEW Code 7-4
C/C++ Code 7-5
Spherical Arcs 7-7
Algorithm 7-9
LabVIEW Code 7-10
C/C++ Code 7-11
Helical Arcs 7-14
Algorithm 7-15
LabVIEW Code 7-16
C/C++ Code 7-17

Chapter 8
Contoured Moves

Arbitrary Contoured Moves 8-1
Algorithm 8-2
Absolute versus Relative Contouring 8-3
LabVIEW Code 8-4
C/C++ Code 8-6

Chapter 9
Reference Moves

Find Reference Move 9-1
Algorithm 9-2
LabVIEW Code 9-3
C/C++ Code 9-4

Chapter 10
Blending Your Moves

Blending 10-1
Superimpose Two Moves 10-2
Blend after First Move Is Complete 10-3

Blend after Delay........10-4
Algorithm........10-5
LabVIEW Code........10-6
C/C++ Code........10-7

Chapter 11
Electronic Gearing

Gearing........11-1
Algorithm........11-2
Gear Master........11-3
LabVIEW Code........11-4
C/C++ Code........11-5

Chapter 12
Acquiring Time-Sampled Position and Velocity Data

Algorithm........12-2
LabVIEW Code........12-3
C/C++ Code........12-4

Chapter 13
Synchronization

Absolute Breakpoints........13-2
Buffered Breakpoints (735*x* only)........13-2
Algorithm........13-3
LabVIEW Code........13-4
C/C++ Code........13-4
Single Position Breakpoints........13-7
Algorithm........13-7
LabVIEW Code........13-8
C/C++ Code........13-9
Relative Position Breakpoints........13-11
Algorithm........13-12
LabVIEW Code........13-13
C/C++ Code........13-13
Periodically Occurring Breakpoints........13-15
Periodic Breakpoints (735*x* only)........13-15
Algorithm........13-16
LabVIEW Code........13-17
C/C++ Code........13-17

Modulo Breakpoints (734*x* and 733*x* only) ... 13-20
Algorithm ... 13-21
LabVIEW Code ... 13-22
C/C++ Code ... 13-23
High-Speed Capture ... 13-25
Buffered High-Speed Capture (735*x* only) ... 13-25
Algorithm ... 13-26
LabVIEW Code ... 13-27
C/C++ Code ... 13-27
Non-Buffered High-Speed Capture ... 13-31
Algorithm ... 13-31
LabVIEW Code ... 13-32
C/C++ Code ... 13-33
Real-Time System Integration Bus (RTSI) ... 13-36
RTSI Implementation on the Motion Controller ... 13-36
Position Breakpoints Using RTSI ... 13-37
Encoder Pulses Using RTSI ... 13-38
Software Trigger Using RTSI ... 13-38
High-Speed Capture Input Using RTSI ... 13-38

Chapter 14
Torque Control

Analog Feedback ... 14-1
Algorithm ... 14-3
LabVIEW Code ... 14-4
C/C++ Code ... 14-5
Monitoring Force ... 14-8
Algorithm ... 14-9
LabVIEW Code ... 14-10
C/C++ Code ... 14-11
Speed Control Based on Analog Value ... 14-13
Algorithm ... 14-14
LabVIEW Code ... 14-15
C/C++ Code ... 14-16

Chapter 15
Onboard Programs

Writing Onboard Programs ... 15-3
Algorithm ... 15-4
LabVIEW Code ... 15-4
C/C++ Code ... 15-5

Running, Stopping, and Pausing Onboard Programs 15-6
Running an Onboard Program 15-6
Stopping an Onboard Program 15-7
Pausing/Resuming an Onboard Program 15-7
Automatic Pausing 15-7
Single-Stepping Using Pause 15-8
Conditionally Executing Onboard Programs 15-8
Algorithm 15-10
LabVIEW Code 15-11
C/C++ Code 15-11
Using Onboard Memory and Data 15-13
Algorithm 15-14
LabVIEW Code 15-15
C/C++ Code 15-16
Branching Onboard Programs 15-18
Algorithm 15-19
LabVIEW Code 15-20
C/C++ Code 15-21
Math Operations 15-23
Indirect Variables 15-23
Onboard Buffers 15-24
Algorithm 15-25
Synchronizing Host Applications with Onboard Programs 15-25
LabVIEW Code 15-27
C/C++ Code 15-29
Onboard Subroutines 15-33
Algorithm 15-33
LabVIEW Code 15-34
C/C++ Code 15-37
Automatically Starting Onboard Programs 15-41
Changing a Time Slice 15-41

PART IV
Creating Applications Using NI-Motion

Chapter 16
Scanning

Connecting Straight-Line Move Segments 16-1
Algorithm 16-2
LabVIEW Code 16-3
C/C++ Code 16-4

Blending Straight-Line Move Segments 16-7
Algorithm 16-8
LabVIEW Code 16-9
C/C++ Code 16-10
User-Defined Scanning Path 16-13
Algorithm 16-14
LabVIEW Code 16-15
C/C++ Code 16-16

Chapter 17
Rotating Knife

Solution 17-2
Algorithm 17-3
LabVIEW Code 17-4
C/C++ Code 17-5

Appendix A
NI-Motion Issues

Appendix B
Technical Support and Professional Services

Glossary

Index

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 guide you to a basic understanding of the NI-Motion driver software, and provide programming steps and examples to help you develop your NI-Motion applications.

This manual is intended for experienced LabVIEW, C/C++ or other programmers. 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 issues.

This manual is written for 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, DBIO<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 note, which alerts you to important information.

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

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. This font 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, and code excerpts.
`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.

Related Documentation

The following documents contain information that you might find helpful as you read this manual:

- *NI-Motion ReadMe.htm*
- *Getting Started With NI Motion Control*
- *NI-Motion C Reference Help*
- *NI-Motion VI Help*
- *NI-Motion LabWindows/CVI Reference Help*
- *NI-Motion Visual BASIC Reference Help*
- *MAX for Motion Online Help*

Part I

Introduction

This user manual provides information about the NI-Motion driver software, motion control setup, and specific task-based help with creating your motion control application 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 NI-735*x*, NI-734*x*, and NI-733*x* families of motion controllers. 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 graphical application development software, as well as 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 includes the hardware and software interface and the physical and functional architecture of the NI motion controllers.

Software and Hardware Interaction

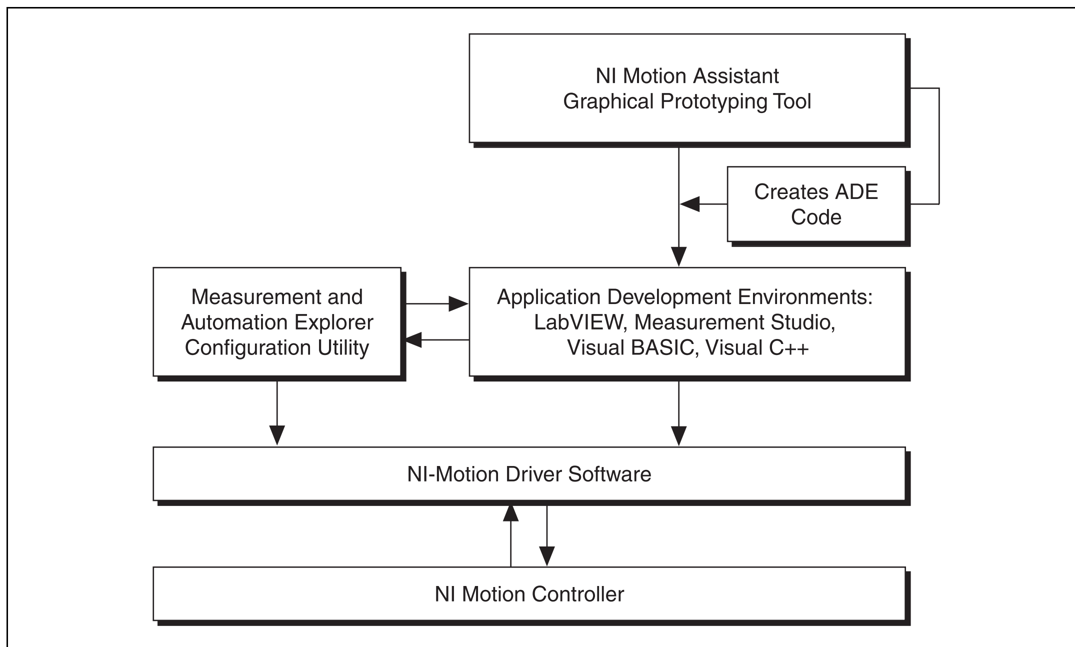


Figure 1-1. NI Motion Control Hardware and Software Interaction

NI Motion Controller Architecture

NI motion 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 (Fierier).

Physical Architecture

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 position, velocity and trajectory maintenance on multiple axes simultaneously. 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 13, *Synchronization*, for more information about breakpoints and high-speed capture.

The watchdog timer checks for proper processor operation. If the firmware on the controller is unable to process functions within 62 ms, the watchdog timer resets the controller and disallows further communications until you explicitly reset the controller. This ensures the real-time operation of the motion 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 C Reference Help* or the *NI-Motion VI Help* for more information.

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

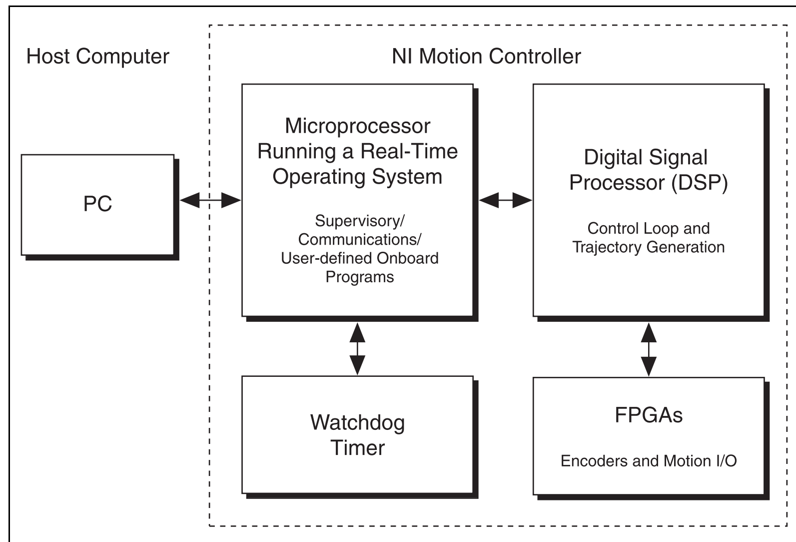


Figure 1-2. Physical NI Motion Controller Architecture

Functional Architecture

Functionally, the architecture of the NI motion controller is divided into four parts: supervisory control, trajectory generator, control loop, and motion I/O.

- Supervisory control—Performs all the command sequencing and co-ordination required to carry out the desired operation
- Trajectory generator—Generates the motion profile on the fly for multiple axes
- Control loop—Performs fast, closed-loop control with position, velocity, and trajectory maintenance on multiple axes simultaneously
- Motion I/O—Performs position breakpoint and high speed capture; is also used by the supervisory control to achieve certain required functionality, such as reacting to limit switches and creating the movement modes needed to initialize the system

Figure 1-3 diagrams the functional architecture of NI motion controllers.

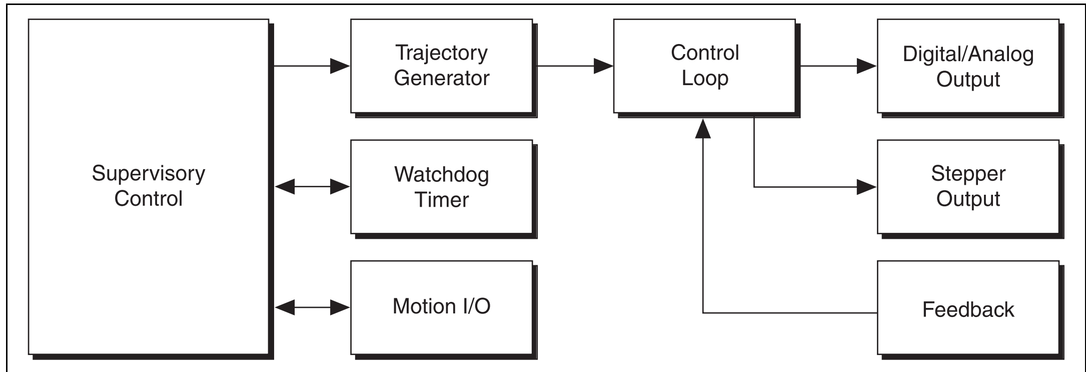


Figure 1-3. NI Motion Controller Functional Architecture

Documentation and Examples

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

- *Getting Started With NI Motion Control*—If you are new to motion control using NI motion control, use this guide to bring the motion control system through installation, setup, and testing.
- *NI-Motion VI Help*—Refer to this document for specific information about NI-Motion LabVIEW VIs.
- *NI-Motion C Reference Help*—Refer to this document for specific information about NI-Motion C/C++ functions.
- *NI-Motion LabWindows/CVI Reference Help*—Refer to this document for specific information about NI-Motion functions.
- *NI-Motion Visual BASIC Reference Help*—Refer to this document for specific information about NI-Motion functions.
- *MAX for Motion Online Help*—Refer to this document for configuration information.
- *NI-Motion ReadMe.htm file*—Consult this document for information about hardware and software installation and any changes to the NI-Motion driver software in the current version. This document also contains any last-minute information about your version of NI-Motion.
- Application notes—If you want to know more about advanced NI-Motion concepts and applications, refer to the Application Notes on the National Instruments Web site at `ni.com/appnotes.nsf/`.

- NI Developer Zone (NIDZ)—For even more information about developing your motion application, visit the NI Developer Zone at `ni.com/zone`. The NI Developer Zone contains 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 your motion control application.

In addition to the NI Developer Zone, you can find NI-Motion C/C++ and LabVIEW VI programming examples in the `NI-Motion\Examples` folder where you installed NI-Motion. The default directory is `Program Files\National Instruments\NIMotion`.

You can find other 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.

2 Creating NI-Motion Applications

This chapter describes the basic form of an NI-Motion application and its interaction with other measurements, such as data acquisition (DAQ) or image acquisition (IMAQ).

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.

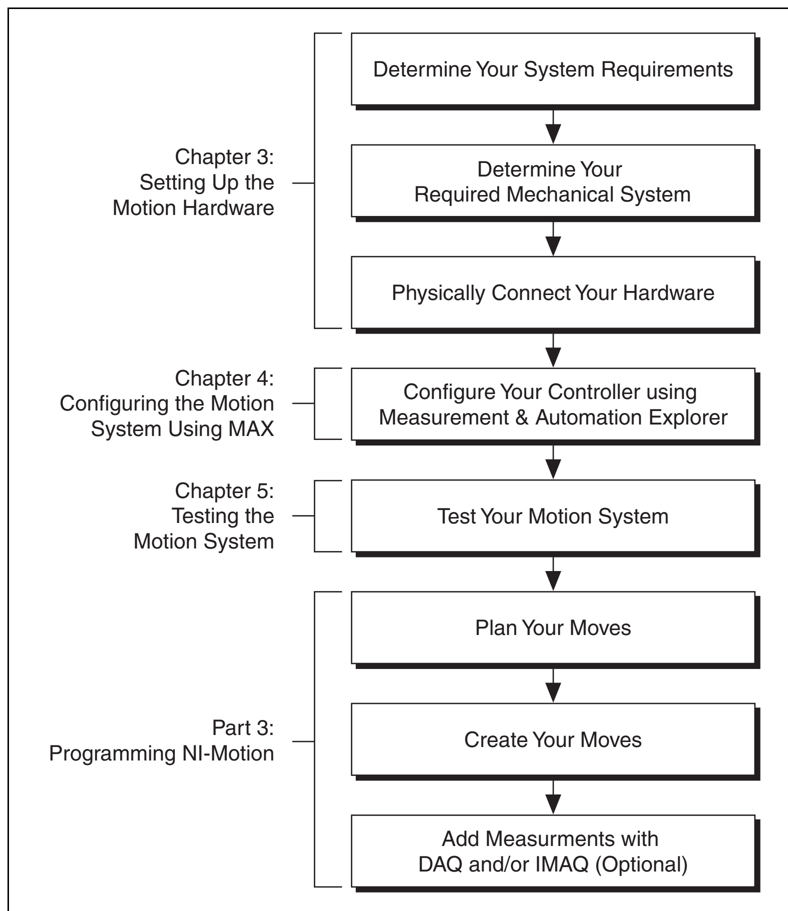
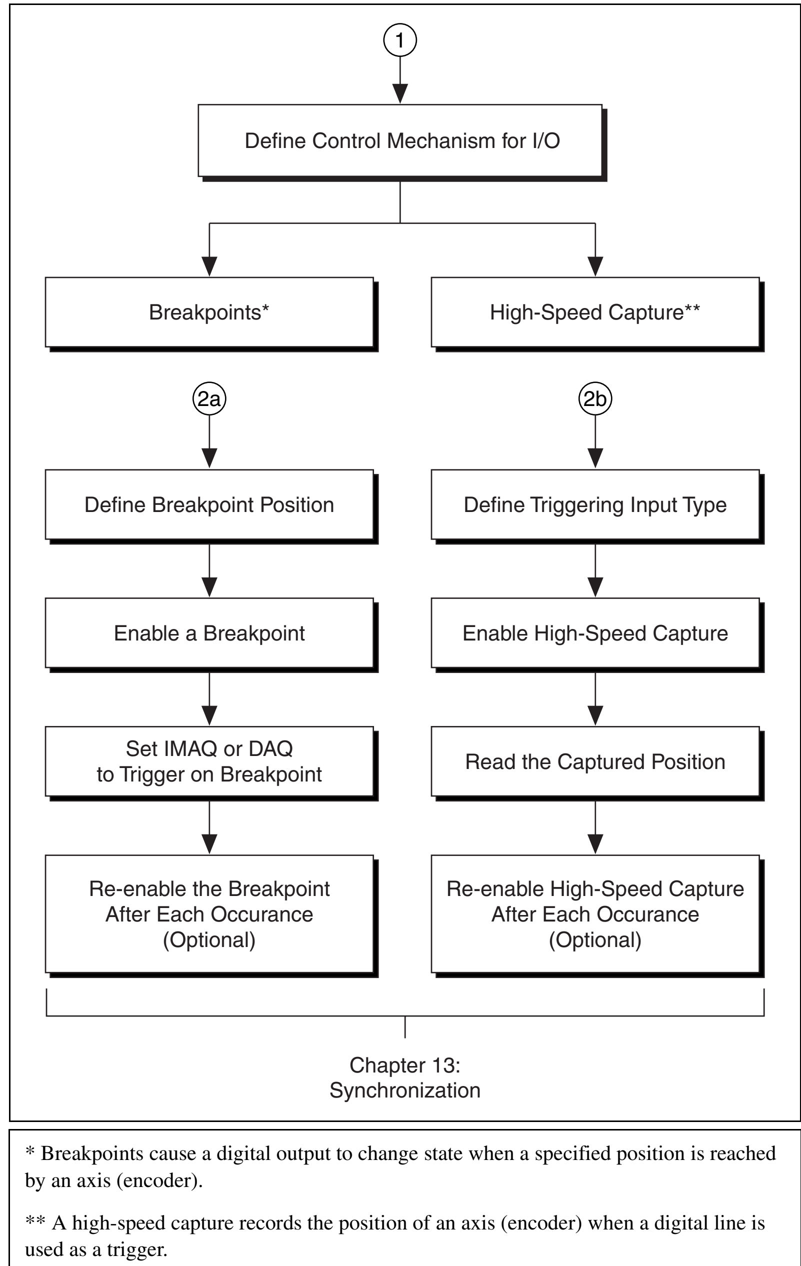


Figure 2-1. Generic Steps to Designing a Motion Application

Adding Measurements to Your NI-Motion Application

Figure 2-2 illustrates an exploded view of the last block in Figure 2-1. For more information about items in the diagram, refer to Chapter 13, *Synchronization*.



* Breakpoints cause a digital output to change state when a specified position is reached by an axis (encoder).

** A high-speed capture records the position of an axis (encoder) when a digital line is used as a trigger.

Figure 2-2. Input/Output with DAQ and IMAQ

Part II

Configuring Motion Control

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

Part II covers the following topics:

- *Setting up the Motion Hardware*
- *Configuring the Motion System Using MAX*
- *Testing the Motion System*

3

Setting up the Motion Hardware

This chapter discusses the necessary components of an NI motion control system.

Motion Control Components

A typical motion control system consists of several hardware items, as shown in Figure 3-1.

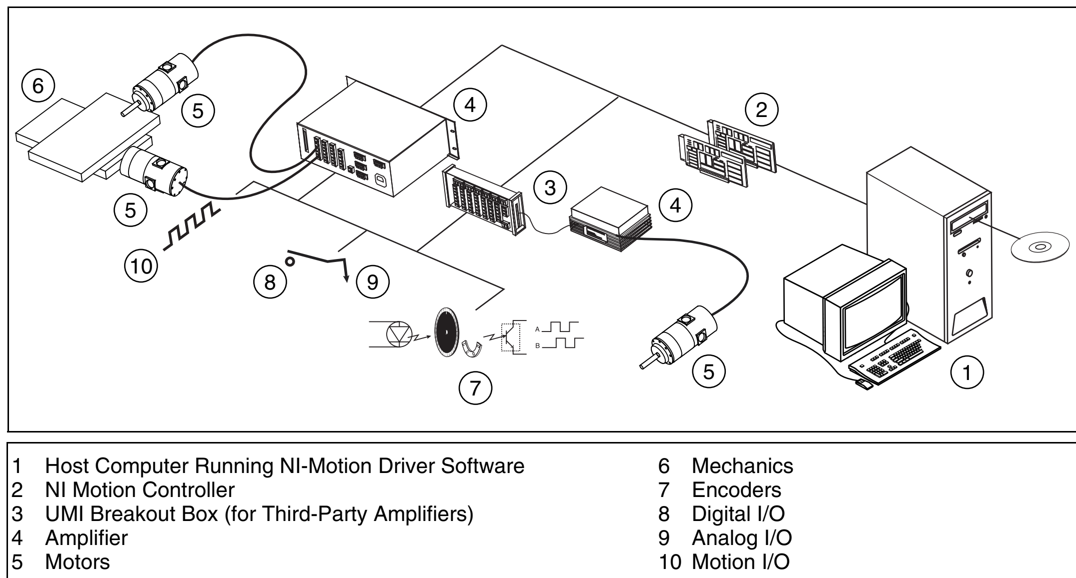


Figure 3-1. Typical Motion Control System

Controller

The NI-Motion driver software works with NI motion controllers exclusively. NI motion controllers use commands coded in LabVIEW or C/C++, along with configuration settings from Measurement & Automation Explorer (MAX), as roadmaps to generate command signals that move the motors. Examples of NI motion controllers include the NI PCI/PXI/FW-734*x* and 735*x* series of stepper/servo controllers and the

NI PCI/PXI-733*x* series of stepper-only controllers. Figure 3-2 shows a PCI-7334 motion controller.

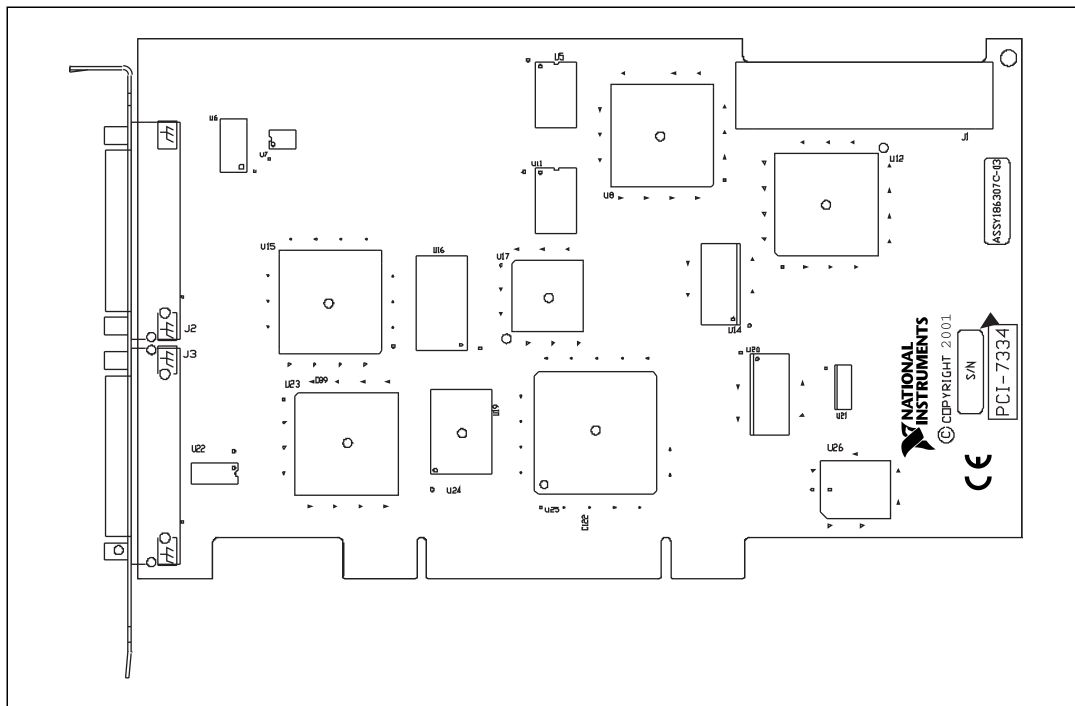


Figure 3-2. NI PCI-7334 Motion Controller

Servo Control

By default, NI-Motion generates servo motor control with an analog command signal of ±10 V. Use the torque control panel in MAX to tweak servo voltages. Use the **Calibration** panel in MAX to configure servo loop parameters.

Step Generation

NI-Motion supports the two industry-standard stepper output configurations: Step-and-Direction and CW/CCW pulses. The most popular mode is Step-and-Direction, where one output produces the step pulses and the other output produces a direction signal.

In clockwise (CW) and counter-clockwise (CCW) mode, the first output produces pulses when moving forward, or CW, while the second output produces pulses when moving reverse, or CCW.

In either mode, you can set the active polarity with the polarity bit to be active low (inverting) or active high (nonirritating). For example, in Step-and-Direction mode, the polarity bit determines whether a high direction output is forward or reverse. It also determines the resting states of outputs when they are not pulsing.

NI-Motion also supports microstepping with appropriate power drives. It is often possible to further refine the accuracy of a stepper motor by using microstepping. Microstepping creates multiple positions between each physical step by directing varying amounts of current to two adjacent steps. For example, suppose the step the motor points to reduces its level of current to 90% and the next step increases its level of current to 10%. The motor re-orients itself to a position 10% of the way to the second step, thereby dividing the steps into tenths. A 200-step motor becomes a 2,000-step motor. This is known as 10X microstepping.

Amplifier

NI motion controllers operate in the low voltage ranges normally seen in a PC bus. This relatively small amount of current provides insufficient power for turning most motors. Therefore, motion systems use amplifiers, or motor drives, to increase the power of motion controller signals, as shown in Figure 3-3. Refer to the amplifier documentation for more information on installation and use.

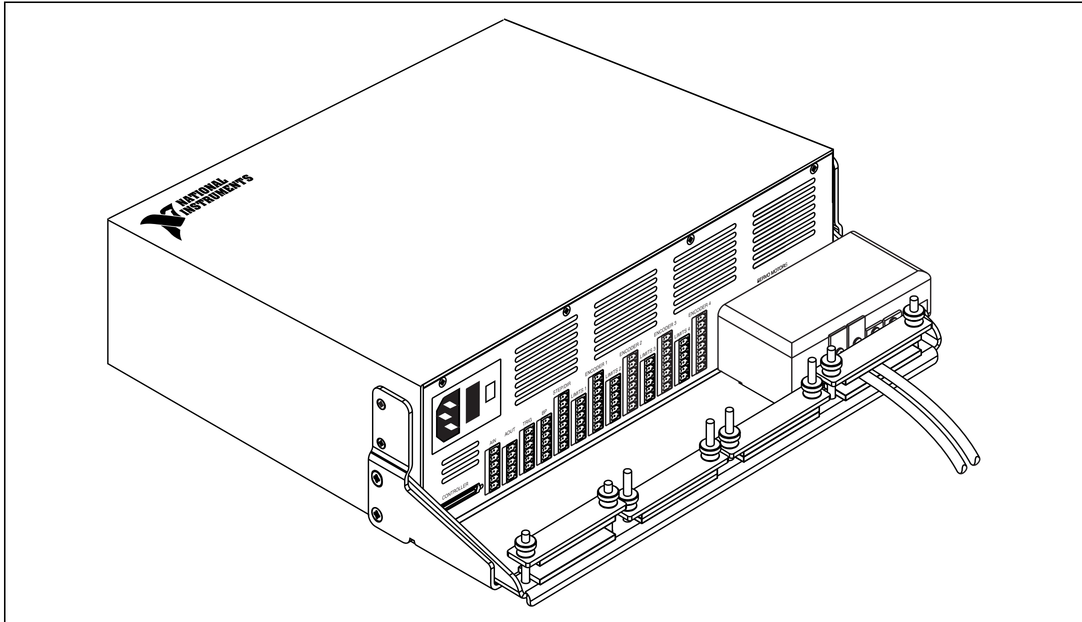


Figure 3-3. MID-7654 Power Servo Motor Amplifier

UMI Breakout Box

Whereas National Instruments motor amplifiers connect directly to NI motion controllers, amplifiers made by third-party companies require a Universal Motion Interface (UMI). An NI UMI is a breakout box that enables pin-level connectivity between the NI motion controller and any third-party amplifier.

Motors and Actuators

Motors and actuators convert the current from the amplifier to physical motion. Physical motion can be in the form of rotational motion, in the case of the traditional rotary motor, linear movement, in the case of linear motors, and other forms of motion. The UMI enables NI-Motion and NI motion controllers to support a wide variety of motor and actuator types.

Encoders and Other Feedback Devices

NI-Motion supports motor feedback in the form of signals produced by the following devices:

- Encoders—Sensors attached to the motors that send feedback to the motion controller about position
- Hall effect sensors (735*x* only)—Sensors attached to the mechanical system that detect changes in commutation phase during calibration
- Analog Transducers—Sensors that convert measurements to an analog voltage

Encoders attach to the motors and send feedback about rotation to the motion controller, as shown in Figure 3-4. The controller uses this feedback to monitor the position and velocity of the motors.

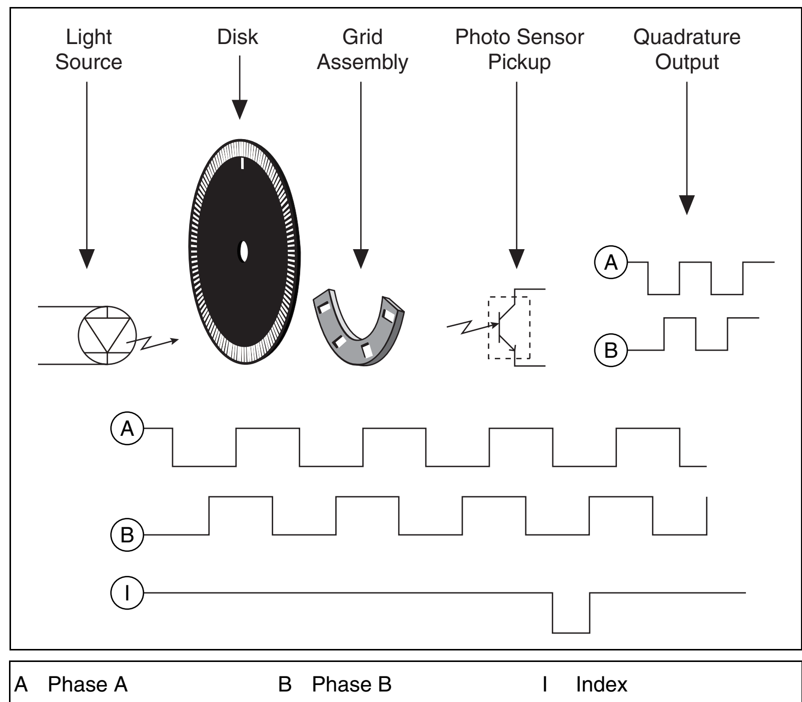


Figure 3-4. Encoders With Waveform Output

Mechanical System

The mechanical system converts the rotational torque or linear motion of a motor into usable motion. The most common form of mechanical system used with NI motion products are *stages*. NI motion controllers are designed to work with a variety of stages and linear actuators from many different stage vendors. Visit `ni.com/motion` for more information about preferred stage vendors and connectivity.

Digital I/O

NI motion controllers feature auxiliary digital I/O. You can use this for various general-purpose I/O-related functions. For example, you could use the digital I/O on the motion controller to control a valve.

Analog I/O

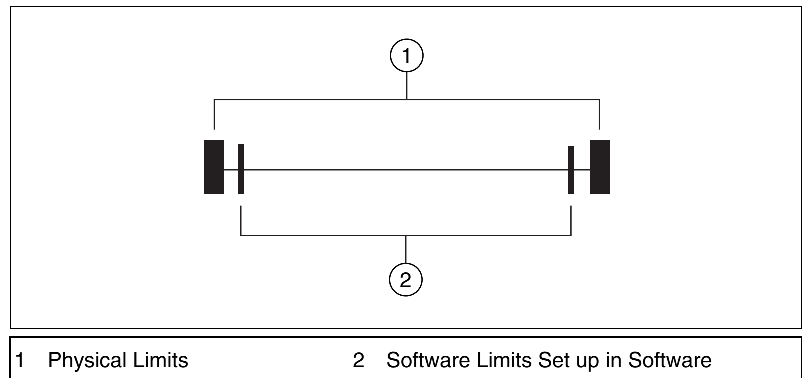
Analog I/O can be used to control a system with an analog transducer feedback device. For example, the motion controller can use analog feedback from a torque sensor to control the torque output of the system. You also can configure the controller to track the analog feedback signal using gearing.

Motion I/O

Motion I/O includes limits and home inputs, breakpoints, trigger inputs, and inhibit outputs.

Limits

Limits signal the ends-of-travel on a motion system. You can connect physical limit switches to the controller to protect the motion system from physical damage by indicating a hard limit of travel. In addition to physical limit switches, you can configure “virtual” limit switches in the software to signal an upcoming physical limit switch. These software limits provide additional protection for high-speed motion systems, as shown in Figure 3-5.



1 Physical Limits	2 Software Limits Set up in Software

Figure 3-5. Physical and Software Limits

Limits can affect multiple axes of motion. For example, if any axis in a *Coordinate Space (Vector Space)* move exceeds a limit, all axes in the coordinate space decelerate to a stop.

Hardware and software limits enhance the NI motion controller and are not required for basic motion control. You can operate all the motion control functions and VIs without enabling or using limits except the *Find Reference Move*, which requires enabled limit and home inputs.

Home

A home switch is a physical switch placed somewhere within the range of travel of the motion system. You can search for the home switch to find a known reference location for your motion application.

Breakpoints

Breakpoints send an output signal whenever the motion system reaches designated positions along the move path between the starting position and the target position. You can use breakpoints to trigger non-motion activities at specific locations, such as data acquisition (DAQ) or image acquisition (IMAQ).

You also can use a breakpoint as a general-purpose output.

Refer to Chapter 13, *Synchronization*, for more information about breakpoints.

Trigger Inputs (High-Speed Capture)

Trigger inputs, or high-speed captures, capture the precise position of the motion system when an external trigger signal occurs. For example, a data acquisition (DAQ) device connected to a thermocouple moved by a motor can trigger a high-speed capture whenever it acquires a temperature that exceeds a designated threshold.

You also can use trigger inputs as general-purpose inputs.

Refer to Chapter 13, *Synchronization*, for more information about high-speed capture.

Inhibit Output

Inhibit outputs are typically used to disable the servo amplifier or stepper driver for power savings, safety, or other specific application reasons. You also can use inhibit outputs as general-purpose outputs.

4

Configuring the Motion System Using MAX

Each motion system is a combination of hardware components with different characteristics that affect system behavior and performance. Similarly, each motion application has different requirements that must be met by the motion system to maximize performance. NI-Motion uses the Measurement & Automation Explorer (MAX) configuration utility to give you complete control over the characteristics of the motion system.

This chapter details the characteristics of a system you can configure using MAX and describes each of the configuration options available.

Configuring the System

To begin configuring the motion system, open MAX by double clicking the MAX icon on the desktop.

MAX features a navigation tree on the left side of the application window for switching between configuration panels for one or more devices. MAX displays the installed motion controller in the navigation tree, and labels it by its bus and number designation, as shown in Figure 4-1.

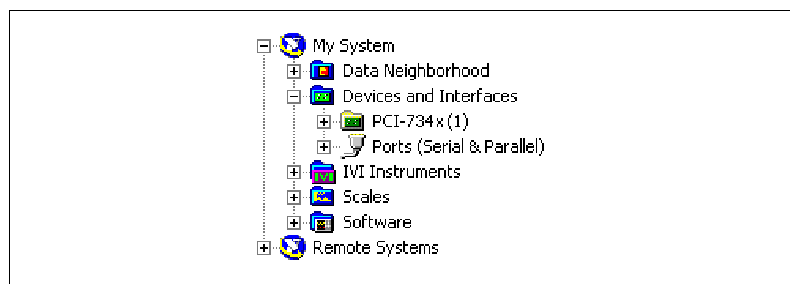


Figure 4-1. MAX Navigation Tree

To view information about the motion controller, its firmware, resources, and status, click the controller icon. When you select the main device icon, four tabs display information in the center window about the motion controller. These tabs are **General**, **Resources**, **Firmware**, and **Status**.

- The **General** tab displays information about the controller, its basic features, and the most recent firmware version.
- The **Resources** tab displays information about the system resources used by the motion controller. You also can perform a basic diagnostic on the motion controller from within this tab.
- The **Firmware** tab displays information about the firmware on the controller and provides an easy interface for downloading newly-installed firmware to the controller, as shown in Figure 4-2.

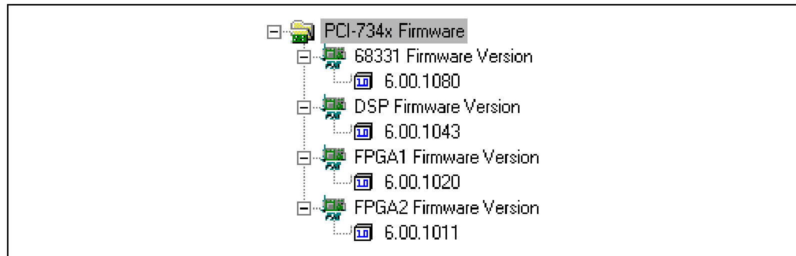


Figure 4-2. Firmware Tab

- The **Status** tab displays the status of the different bits in the communication status register of the controller, as shown in Figure 4-3. You also can reset the motion controller from this tab.

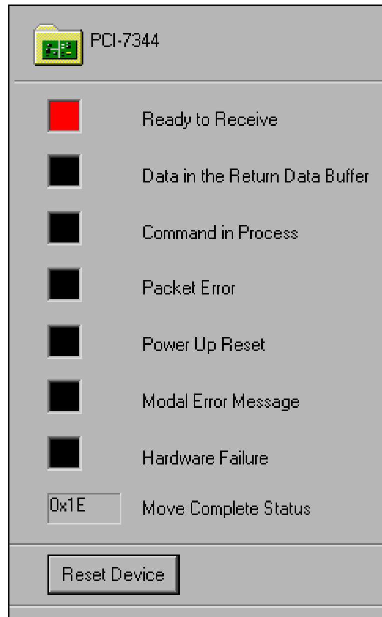


Figure 4-3. Status Tab

Below the main device icon are four main categories of configuration: Device Resources, Initialization Settings, Interactive panels, and Calibration (734*x* and 735*x* controllers only), as shown in Figure 4-4.

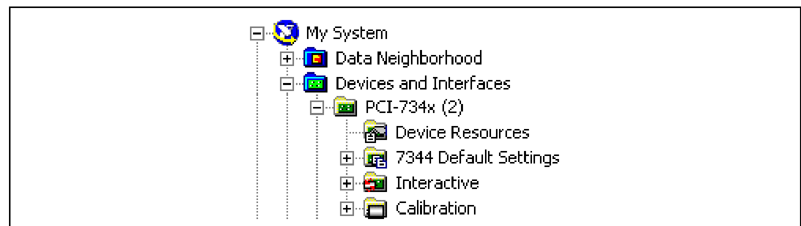


Figure 4-4. Device Categories in MAX

Device Resources

Click the **Device Resources** icon to view the resources available to the selected motion controller. Resources can include analog channels, axes, DIO bits, and stepper outputs, as shown in Figure 4-5.

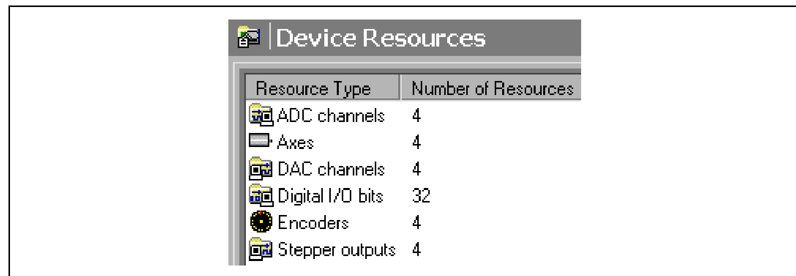


Figure 4-5. Device Attributes Tab

The Onboard Memory Manager tab within Device Resources allows you to manage all objects in onboard memory. From this location you can execute, stop, or pause onboard programs previously stored. You also can save onboard programs to FLASH memory or delete them from onboard RAM and FLASH. This tab serves as a debugging tool when you design your onboard programs.

Initialization Settings

Click the **Initialization Settings** (73*xx* Default Settings) icon to determine the controller settings that are imposed when you initialize the controller. Initialization settings are the configuration settings for the motion controller, and are stored in a database on the host computer. You can customize these settings based on the type of motor or mechanical device you want to control.

You can move your settings to another system by right-clicking on the initialization settings icon and selecting **Initialization Settings»Export Settings**. This action generates a `.xylem` file with the same file name as the settings name in MAX. To avoid confusion, National Instruments recommends you use the default name in most instances. If you do change the settings name, remember that the name of the settings imported into the new system is different than the `.xml` file name.

Complete the following steps to import your settings into another machine.

1. Right-click the settings icon under the appropriate device and select **Initialization Settings»Import Settings**.
2. Select the settings file you exported from another computer.
   The Initialization Settings dialog box appears automatically and shows all currently available settings for the current device in the database.
3. Select the settings you just imported and click **Change**.

The new settings now will be used by the controller. You can copy, paste, delete, and rename your initialization settings using the Change Current Settings dialog box.

The categories under Initialization Settings are Axis Configuration, Axis Settings, Trajectory Settings, Find Reference Settings, Digital I/O Settings, Gearing Settings, ADC Settings, Encoder Settings, PWM Settings, and Synchronization Settings.

For categories that expand to show individual axes, you can set a general change with the root item, or expand the item to apply the change to a specific axis, port, ADC, encoder, or PWM output.

Apply your changes by clicking **Apply**, or by clicking away from the changed item to automatically apply you changes. To send the new settings to the motion controller, click **Initialize**.

Use the Initialization Settings item to reset your settings to default configuration values, as shown in Figure 4-6.

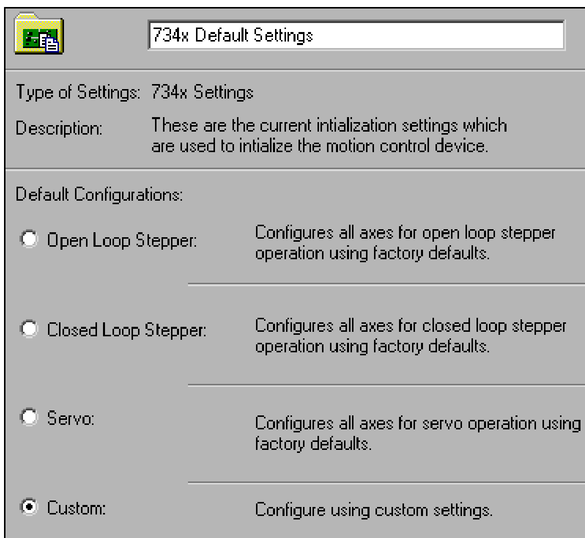


Figure 4-6. Current Settings Tab

For example, you can set the motion system to be open-loop stepper by selecting **Open Loop Stepper** on the Initialization Settings page and clicking **Apply**.

When you select one of the first three options, all axes use the default settings for the type you select. The Custom selection reflects that you have changed at least one of the default values.

Use the Initialization Preferences tab to set which configuration groups are initialized each time, as shown in Figure 4-7.

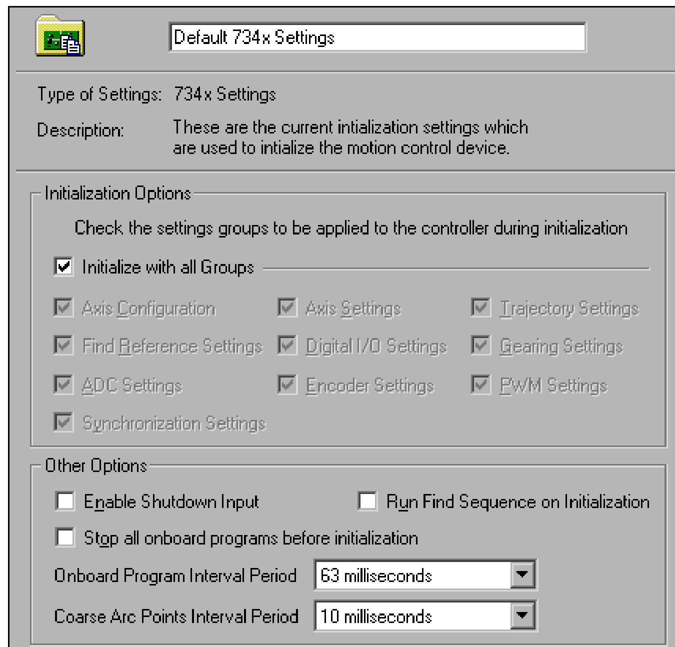


Figure 4-7. Initialization Preferences Tab

Other options on this tab allow you to enable or disable the shutdown input (E-stop), configure the way the controller behaves during initialization, and set the interval periods for arc point generation and onboard programs.

Refer to the *MAX for Motion Online Help* for more information about these options.

Axis Configuration

Use the Axis Configuration item to configure the axes, with or without feedback, and to set the control loop period, as shown in Figure 4-8.

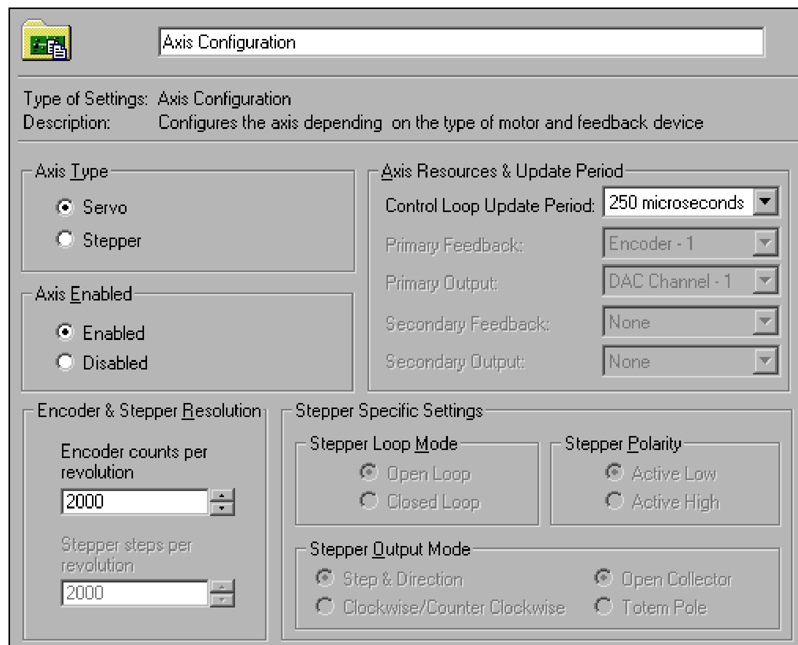


Figure 4-8. Axis Configuration Window

You can set the following options:

- Axis Type—Sets the axes to servo or stepper
- Axis Enabled—Enables/disables the axes
- Encoder & Stepper Resolution—Sets the resolution for the feedback and stepper motor (with microstepping)
- Axis Resources & Update Period—Configures the output and feedback channels and how often the controller updates the state of the motors
- Stepper-Specific Settings—Sets the nature of the stepper motor system

Axis Settings

Use Axis Settings to configure the input and output characteristics, the control loop behavior, and other miscellaneous axis settings, as shown in Figure 4-9.

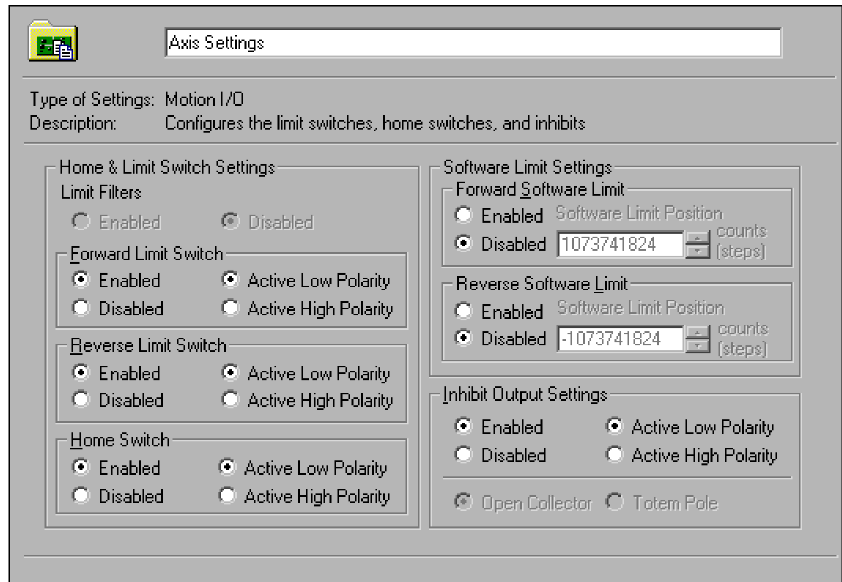


Figure 4-9. Motion I/O Tab

Use the Motion I/O tab to configure the limit and home switches.

Use the Breakpoint & Trigger tab to configure position breakpoints and trigger inputs, as shown in Figure 4-10.

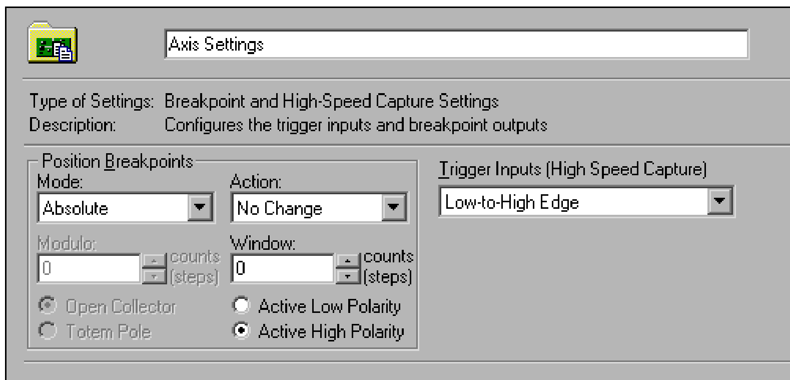


Figure 4-10. Breakpoint & Trigger Tab

Use the Control Loop tab to configure the PID loop of the motion controller, as shown in Figure 4-11.

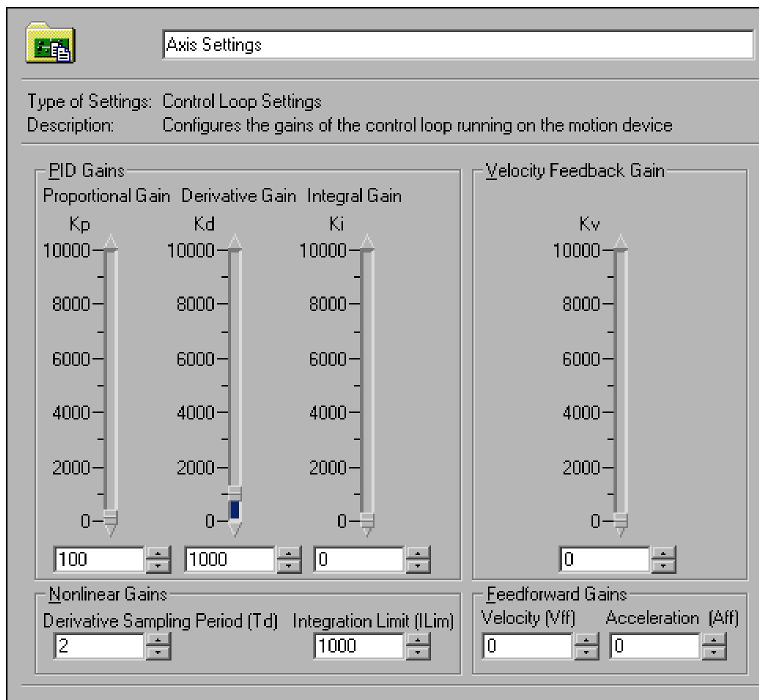


Figure 4-11. Control Loop Tab

Use the Static Friction tab to configure static friction (stiction) compensation, as shown in Figure 4-12. Stiction compensation adjusts for increased power needed to overcome inertia when beginning a move from a standstill.

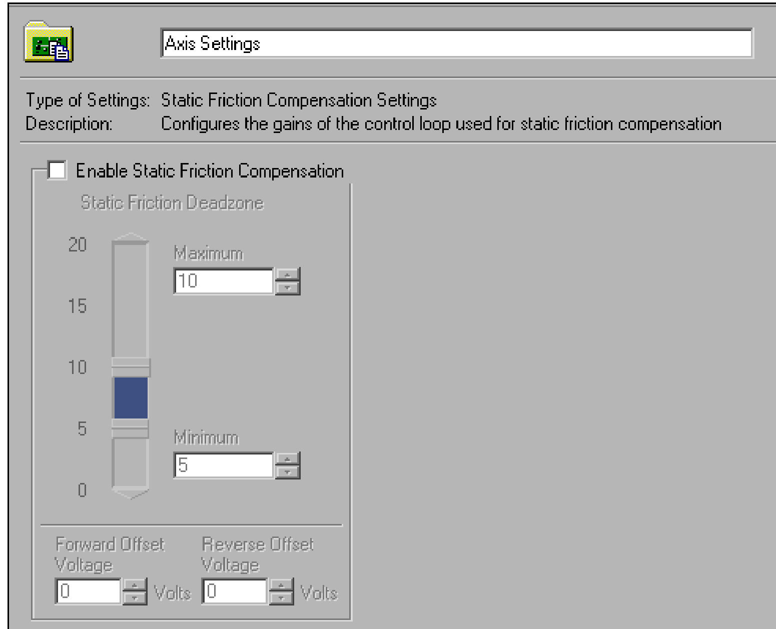


Figure 4-12. Static Friction Tab

Use the Miscellaneous tab to configure torque limits and offsets, as well as filter settings for reading velocity, as shown in Figure 4-13. These gains are used when the controller estimates the velocity of the axis.

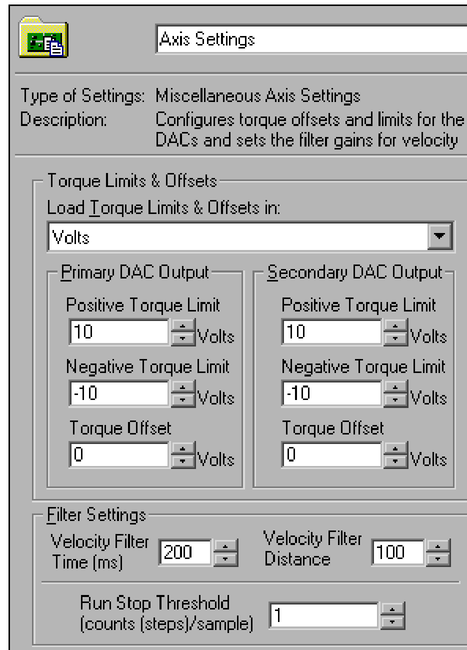


Figure 4-13. Miscellaneous Tab

Trajectory Settings

Use the Trajectory Settings tab to configure the trajectory parameters, such as operation mode and maximum velocity and acceleration. Figure 4-14 shows the configuration options.

Trajectory Settings

Type of Settings: Trajectory Settings
Description: Configures trajectory parameters for the axis

Move Settings
Operation Mode: Absolute Position
Stop Mode: Decelerate
Load Velocity in: rpm
Blending Options: Blend before decelerating
Blending Delay: -1 milliseconds
Velocity: 200 rpm
Following Error: 32767 counts (steps)
Acceleration: 100 rps/s
S Curve Time: 1 sample periods
Deceleration: 100 rps/s
Position Modulus: 0 counts (steps)

Advanced Move Settings
Velocity Threshold: 5000 rpm
Base Velocity: 0 steps/s
Velocity Override: 100 % of loaded velocity
Accel Factor: 1

Figure 4-14. Trajectory Settings Tab

Use the Move Complete Criteria tab to specify conditions that must be satisfied for a move to be considered complete, as shown in Figure 4-15.

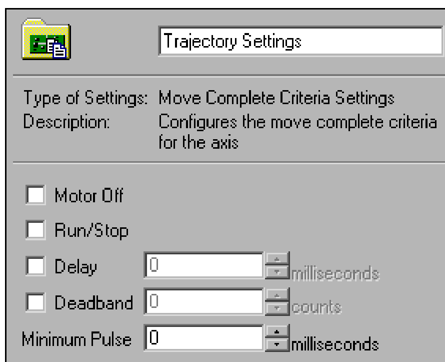


Figure 4-15. Move Complete Criteria Tab

Find Reference Settings

Use the Find Reference Settings tabs to configure the settings used to find the limit/home switches and encoder index pulse. Figures 4-16 and 4-17 show the Home and Index configuration tabs, respectively.

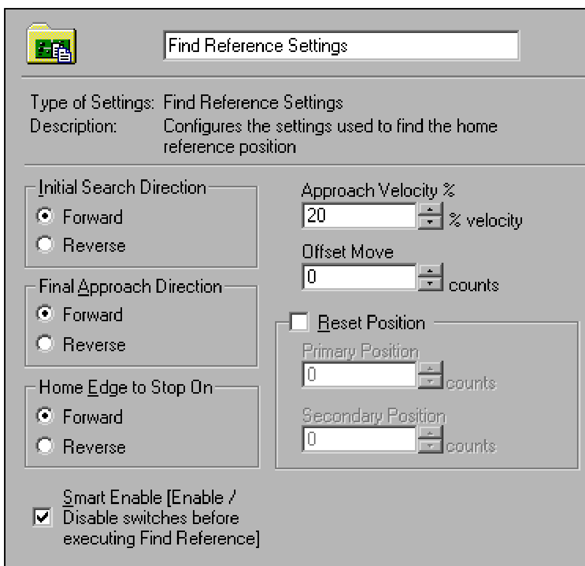


Figure 4-16. Home Settings

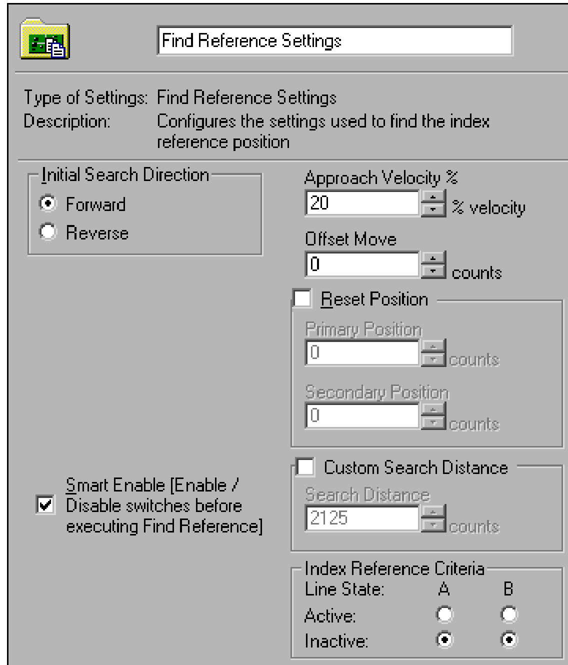


Figure 4-17. Index Settings

Use the Sequence tab, shown in Figure 4-18, to configure the Find Reference sequence.

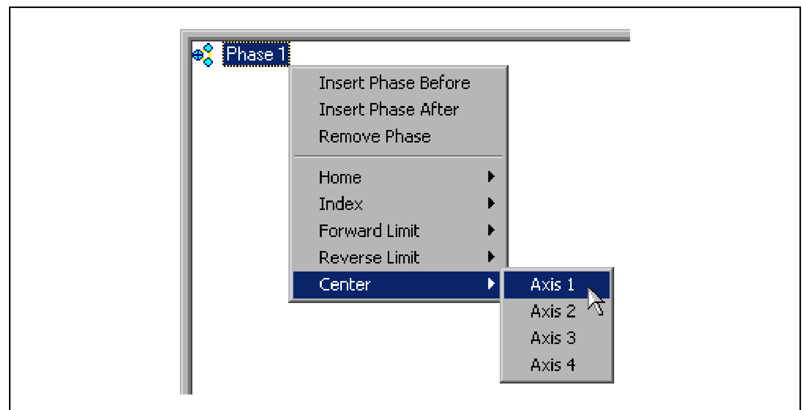


Figure 4-18. Sequence Tab

Use the Forward Limit, Reverse Limit, and Center tabs to configure the settings for the limits and center. The Forward Limit tab is shown in Figure 4-19.

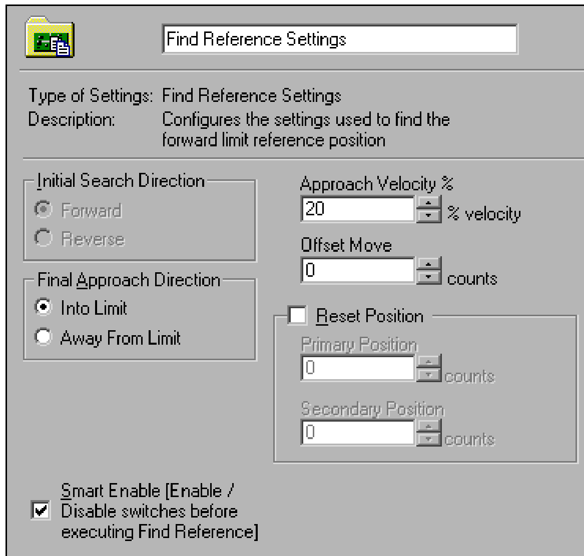


Figure 4-19. Forward Limit Tab

Limit switches and the home switch are physical switches placed somewhere in the range of travel of the axis. The index pulse occurs once for every 360º turn of a rotary motor. You can use the home switch and index pulse to reset the motion system to a known starting state and position.

Digital I/O Settings

Use the Digital I/O Settings item to configure the digital I/O lines on the motion controller, as shown in Figure 4-20.

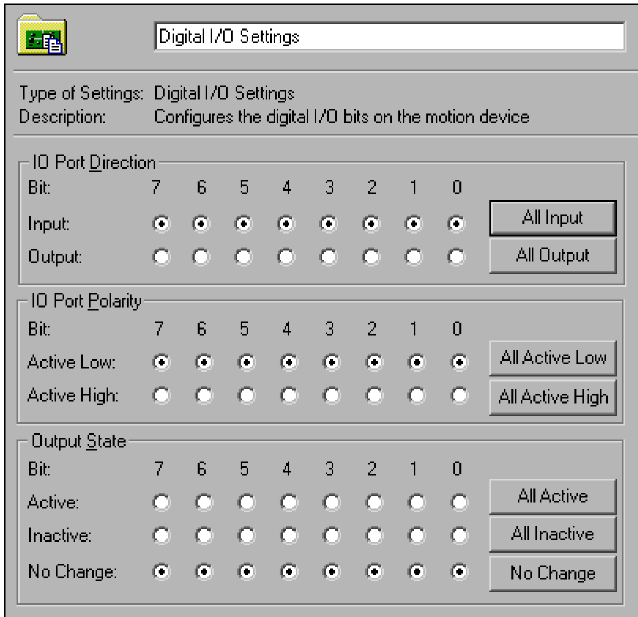


Figure 4-20. Digital I/O Settings

You can designate a line as input or output, change its polarity to active high or low, and make it active or inactive.

Gearing Settings

Use the Gearing Settings item to configure the axes for gearing. You can enable or disable gearing, set master and slave axes, and change the gearing ratio, as shown in Figure 4-21.

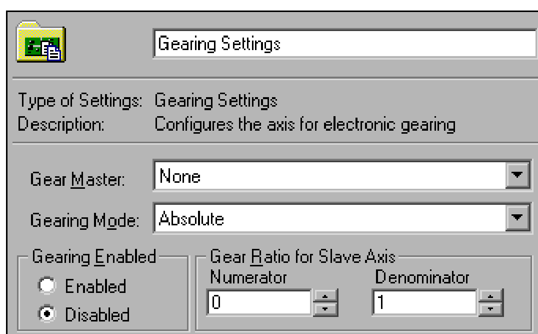


Figure 4-21. Gearing Settings

You can use electronic gearing to tie the movements of one or more slave axes to those of a master device, such that one revolution of the master axis causes a proportional number of revolutions of the slave axes, depending on the ratio you choose.

ADC Settings

Use the ADC Settings item to configure the analog inputs (ADCs) on the motion controller. You can set the ADC voltage range and enable or disable the channel, as shown in Figure 4-22.

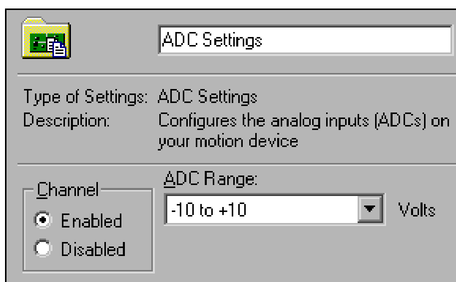


Figure 4-22. ADC Settings

Encoder Settings

Use the Encoder Settings item to enable or disable the encoders, configure phase and index line polarity, and change the filter frequency, as shown in Figure 4-23.

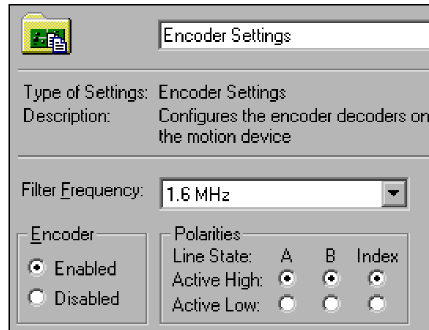


Figure 4-23. Encoder Settings

PWM Settings

Use PWM Settings to configure the PWM (Pulse Width Modulation) outputs on the motion controller. You can enable or disable the PWM outputs, change the clock frequency, and configure the duty cycle, as shown in Figure 4-24.

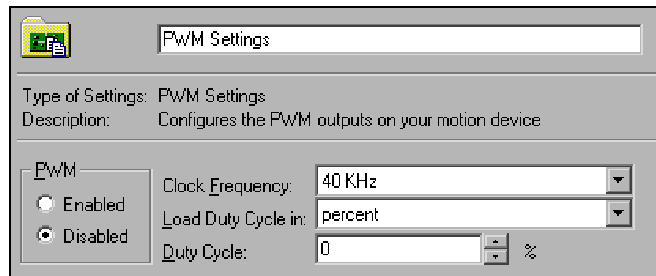


Figure 4-24. PWM Settings

Synchronization Settings

Use Synchronization Settings to configure the RTSI lines, Breakpoints, home and index phases, and other sources, as shown in Figure 4-25.

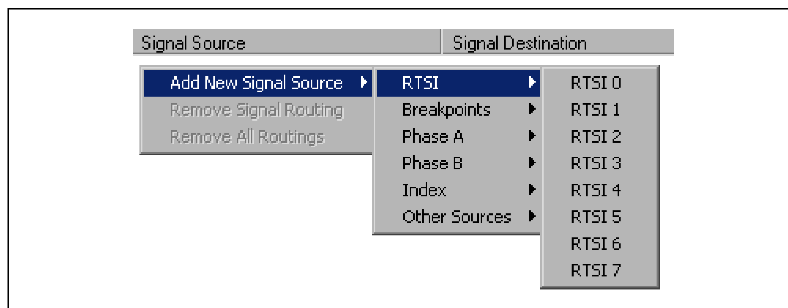


Figure 4-25. Signal Sources

For example, you can configure the source for a RTSI line. Once you have selected a signal source, route it to a destination, as shown in Figure 4-26.

Signal Source	Signal Destination
Breakpoint 2	RTSI 4
Index 2	RTSI 3
RTSI 2	High-Speed Capture 3

Figure 4-26. Signal Sources Routed to Destinations

Interactive

MAX for motion control offers 1D and 2D interactive panels that allow you to test the motion system after you have configured it. These panels serve as a scope where you can view the graphs of the position and velocity of the moves as well as interactively move the motors.

Values you set in the interactive panels are not saved.

1D Interactive

Use the 1D Interactive panel to test the setup and performance of any one axis at a time. Control the basic move parameters from the main tab, as shown in Figure 4-27, and the advanced move parameters from the Advanced tab, as shown in Figure 4-28.

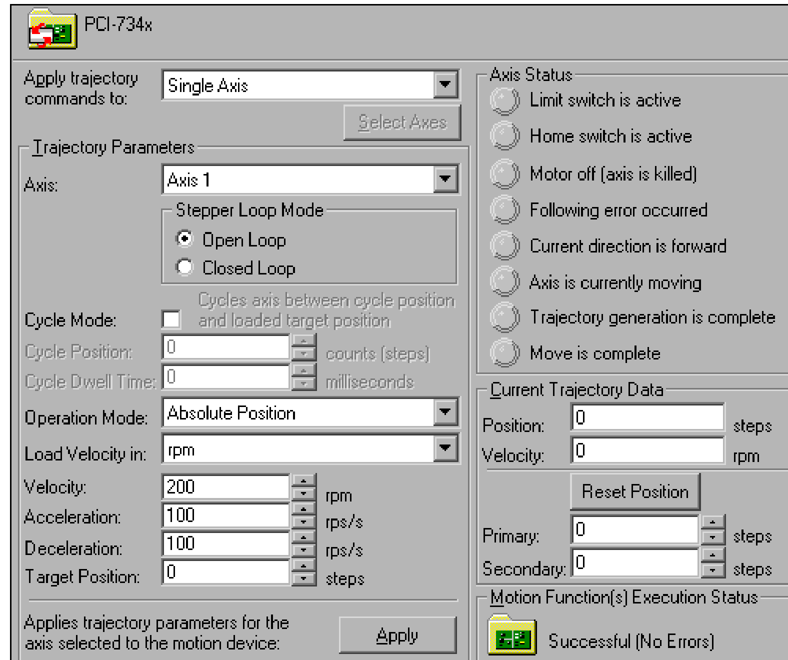


Figure 4-27. 1D Interactive Main Tab

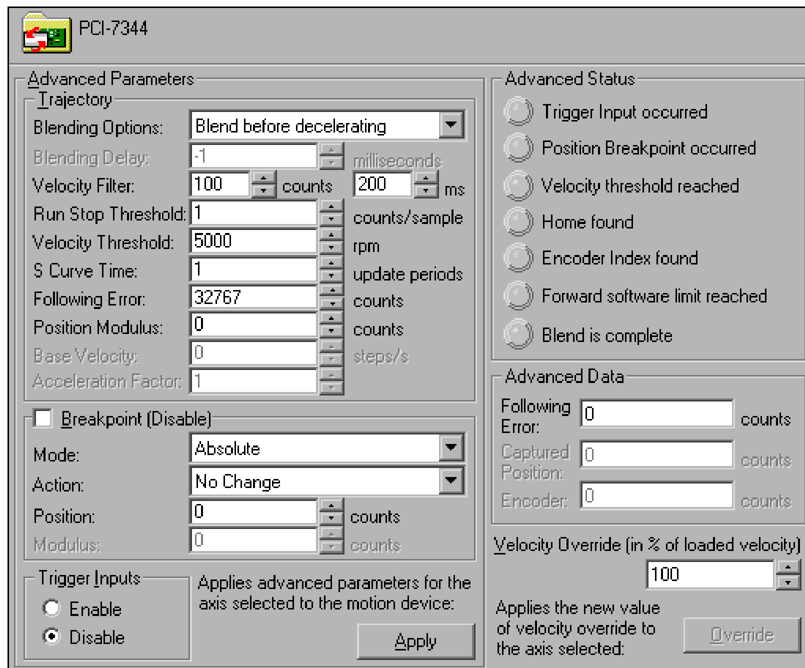


Figure 4-28. 1D Interactive Advanced Tab

Clicking the **Apply** button found on either tab sends the parameters to the motion controller, but does not save the parameters to the database.

Use the Misc. Plots tab to view the results of the motion, as shown in Figure 4-29.

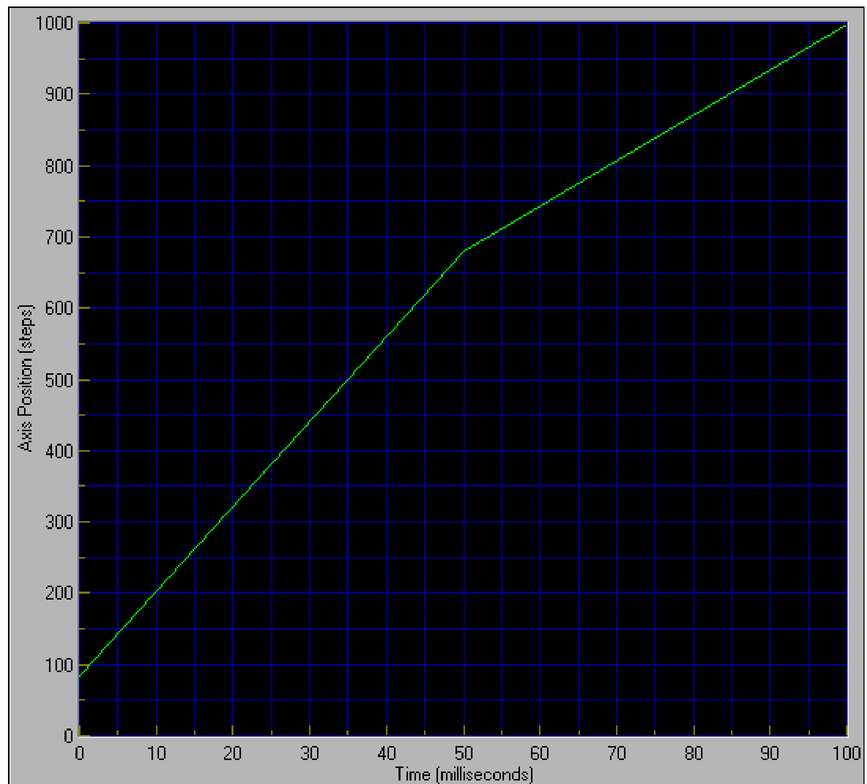


Figure 4-29. Miscellaneous Plots Tab

2D Interactive

Use the 2D Interactive panel to test the setup and performance of any two axes together. Use the position and velocity graphs on the Position and Misc. Plots tabs to view the motion.

Calibration (734*x* and 735*x*)

Motion controllers require calibration to maximize the performance and accuracy of servo motor systems. Use servo tuning to calibrate your servo motors.

Tuning the 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 4-30.

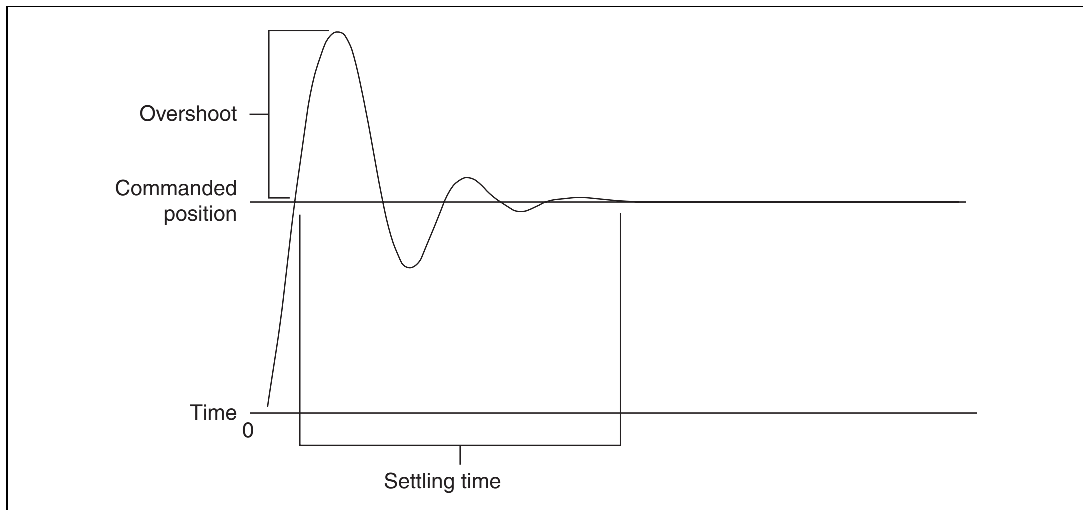


Figure 4-30. Properly Tuned Servo Motor Behavior

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.

Click the **Servo Tune** item under Calibration in the MAX navigation tree to change control loop and response settings. Refer to the *MAX for Motion Online Help* for more information about servo tuning and step-by-step instructions for tuning the servo motors.

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.

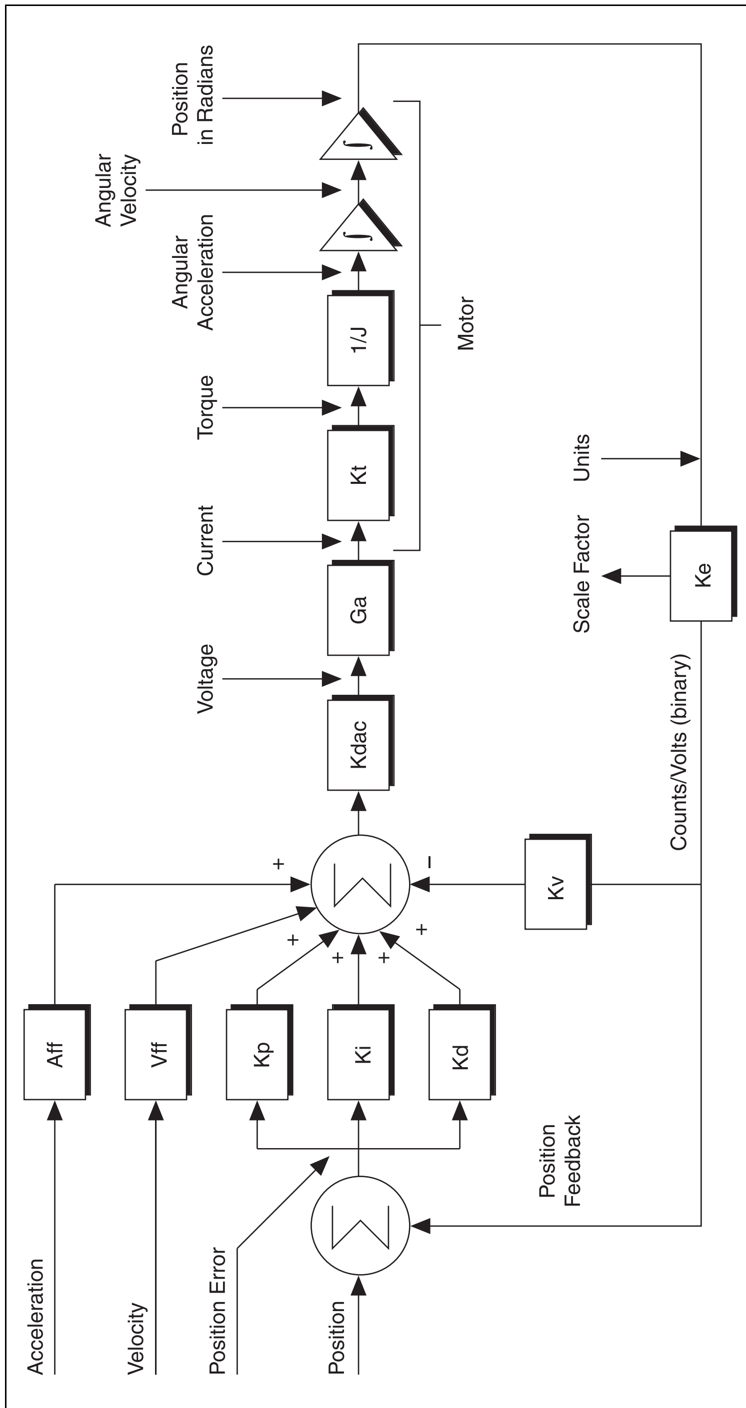


Figure 4-31. NI-Motion Servo PID Loop

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 for 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 secondary feedback encoder for velocity feedback if the axis is configure for it. The velocity feedback gain (Kv) is used to scale this velocity feedback before it is added to the other components in the 16-bit DAC command output.

Velocity feedback gain (Kv) is similar to derivative gain (Kd) except that it scales the velocity estimated from the secondary feedback resource 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.

Using 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.

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.

Using 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 'D' term estimates motor velocity by differentiating the 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 4-32, 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.

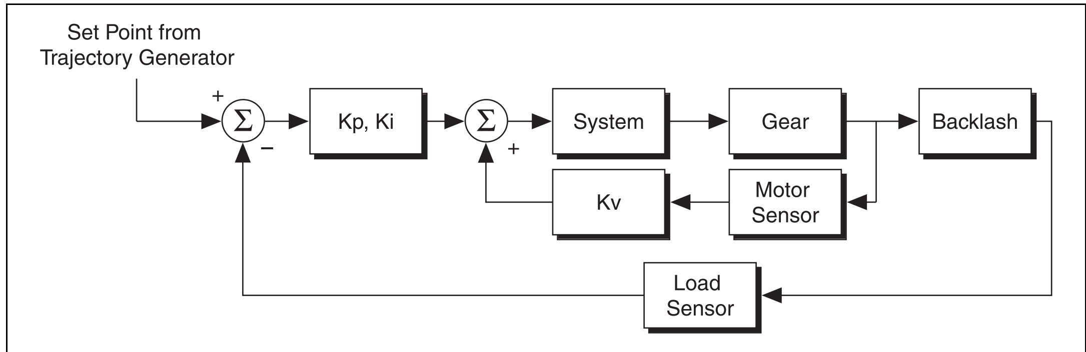


Figure 4-32. Dual Loop Feedback

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 Kv (velocity feedback) gain instead of the Kd (derivative gain) to dampen and stabilize the system, as shown in Figure 4-33.

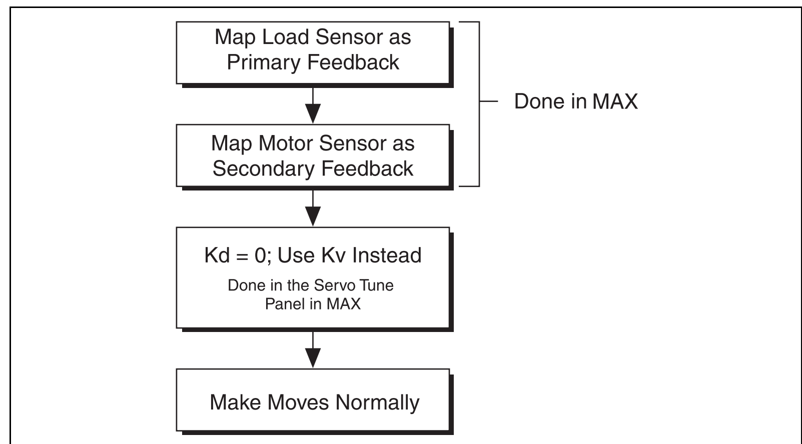


Figure 4-33. Dual Loop Feedback Algorithm

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 (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 4-34.

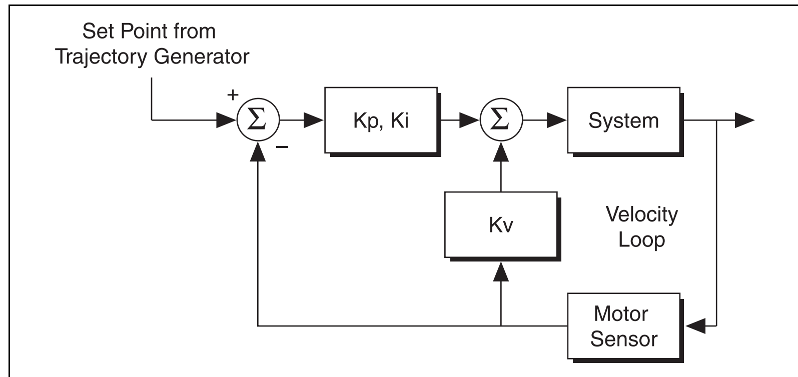


Figure 4-34. Velocity Feedback

You can configure the same position sensor to be the primary and secondary feedback for an axis. Once this is done, you can 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 the secondary feedback resource 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 4-35.

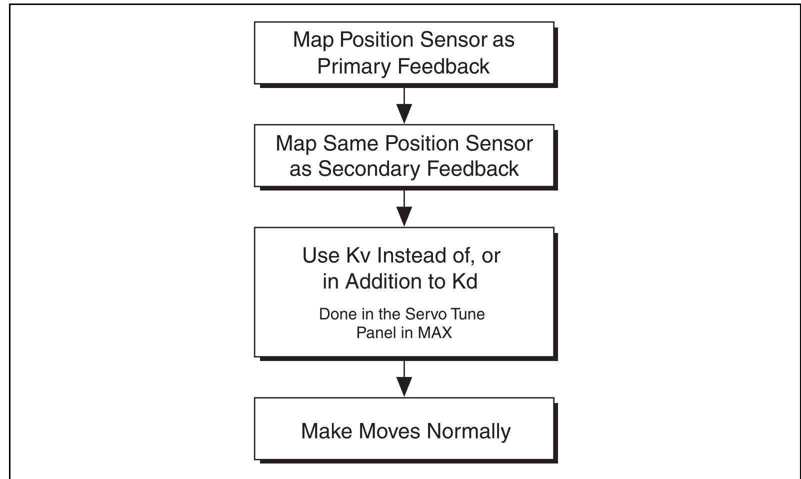


Figure 4-35. Alternate Dual-Loop Feedback Algorithm

NI Motion Controllers with Velocity Amplifiers

You can use the NI motion controller with velocity amplifiers in PIVff mode. In this mode, the Kd and Kv gains are set to zero.

Velocity amplifiers close the velocity loop using a tachometer on the amplifier itself, as shown in Figure 4-36. 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.

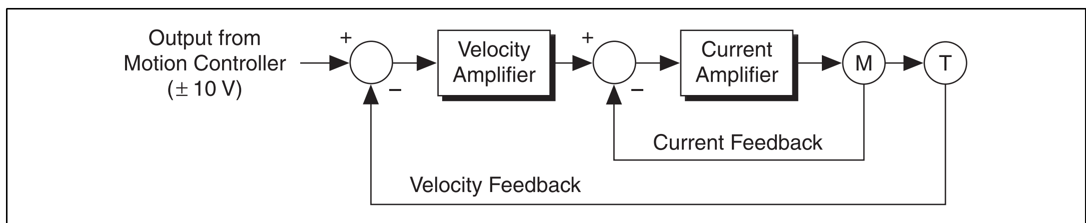


Figure 4-36. NI Motion Controllers with Velocity Amplifiers

Figure 4-37 describes how to use NI motion controllers with velocity amplifiers.

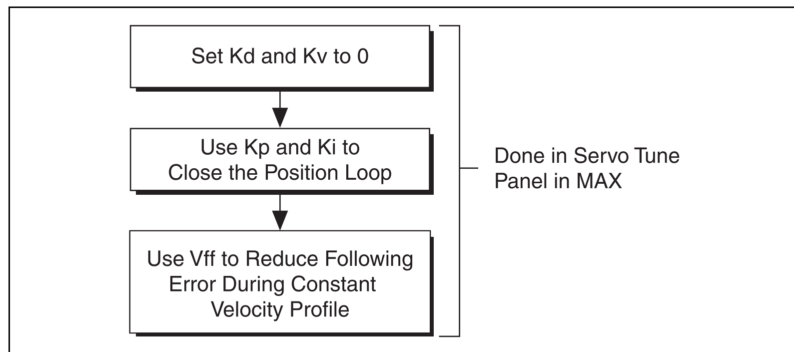


Figure 4-37. NI Motion Controllers with Velocity Amplifiers Algorithm

Velocity feedforward is typically used when operating in this mode. 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 the recommended solution.

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 velocity feedforward is not required.

Sinusoidal Commutation for Brushless Servo Motion Control

Sinusoidal commutation allows you to use less expensive servo motor drives with NI motion controllers that support this feature.

Phase Initialization

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:

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

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.

Initializing the Controller Programmatically

You can initialize the motion controller from within a LabVIEW, Visual BASIC, or C/C++ program, in addition to manual initialization in Measurement & Automation Explorer. Programmatic initialization involves adding the Initialize Controller function or VI to the beginning of your code. Initializing the controller causes the initialization settings that are saved in MAX to be sent to the motion controller.

Using the Motion Controller with RT

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:

- NI PXI chassis with an available PXI slot
- NI PXI Motion controller
- Host computer
- LabVIEW Real-Time Module
- NI-Motion 6.0 or later

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 NI-Motion 6.0 or later onto the host system.
2. Launch MAX.
3. Expand the **Remote Systems** tree.
4. Highlight the system on which to install NI-Motion.
5. Select the **Software** tab.

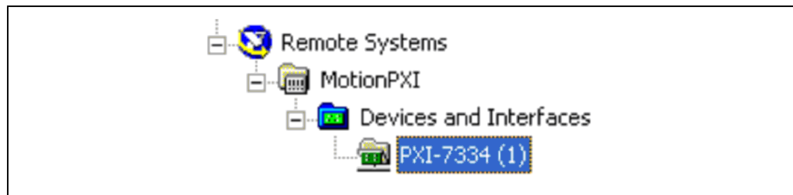


Figure 4-38. Remote Systems Tree

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 motion system 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.

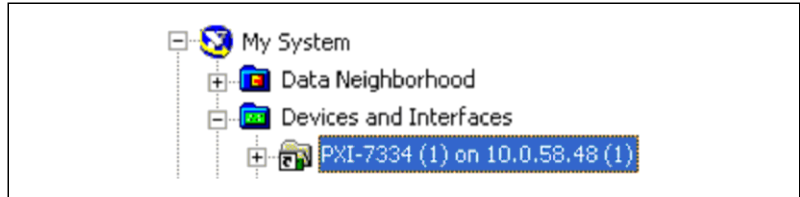


Figure 4-39. Remote System Mapped to a Local Board ID

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.

5

Testing the Motion System

It is important to test the motion control system setup before building and programming your application. Do the following procedures to test the motion system. Perform each procedure once for each motor individually.

Testing the Encoders

If the motion system uses servo motors or closed-loop stepper motors, you must test the encoders. If the motion system uses open-loop stepper motors, skip this procedure. Complete the following steps to test the encoders.

1. Power off the motors and drive. If you have a National Instruments nuDrive or MID drive, do not turn on the **Enable** switch.
2. Ensure that the encoders are powered on.
3. Launch MAX and select the motion controller that is connected to the system.
4. Initialize the motion controller by clicking **Initialize**. This step assumes that you have correctly configured the Initialization Settings.
5. Open the **1D Interactive** panel.
6. If you are testing a stepper motor, skip to step 8. If you are testing a servo motor, select the main tab on the **1D Interactive** panel.
7. Move the motor manually in the forward direction. You should see the encoder count increment in the position indicator. Now move the motor manually in the reverse direction. You should see the encoder count decrement. You now have tested the servo motor encoder.

   If the encoders failed this test, skip to step 9. Otherwise, skip to step 10.
8. If you are testing a stepper motor, ensure you have closed-loop selected as the stepper loop mode on the **1D Interactive** main panel. Select the **Advanced** panel and make sure the encoder position increments in the encoder indicator.

9. If you see the encoder decrement instead of increment, swap encoder cable A with $\overline{A}$ or B with $\overline{B}$, but not both. If you have single-ended encoders, swap channel A with B.
10. Repeat steps 1 through 7 for the remaining encoders.

You have now tested the encoders.

Testing the Motors

Complete the following steps to test the motors:

1. Disconnect power to the motors and drive. If you have a National Instruments nuDrive or MID drive, do not turn on the **Enable** switch.
2. Verify all the wiring between the controller and motor and between the drive and the motor. If you have servo motors, you may need to verify the direction of the voltage. If you have stepper motors, you may need to verify the polarity of the step and direction lines. Refer to the motor drive documentation for polarity information.
3. Launch MAX and select the motion controller that is connected to the system.
4. Initialize the motion controller by clicking **Initialize**. This step assumes that you have correctly configured the Initialization Settings and saved them by clicking **Apply**.
5. Open the **1D Interactive** panel, and click **Kill** to allow the motors to spin freely.

Tip The motor could run out of control if the motion system is set up incorrectly. Set the following error to a low number in MAX to prevent damage to the motion system.

6. Power on the drive and motor.
7. Click **Halt**. This should energize the motor and freeze it into position. If this happens, the controller is correctly sending information to your drive. For servo motors, this may cause the motor to buzz or oscillate. If this happens, use the servo tune panels to tune the motors. For stepper motors, execute a move to identify correct behavior.
8. Repeat steps 1 through 7 for the remaining motors.

You have now tested the motors.

Troubleshooting the Motors

Complete the following steps if the motors do not move as expected when you test them.

Check the Motion Controller

1. Ensure that the cable is not loose on either end.
2. Check polarities of the inhibits to make sure they match the hardware settings on the UMI or drive.
3. For servo motors, check the following error status to ensure they are not tripping out on following error immediately after the start of a move. This could happen if the encoders or motors are wired incorrectly. Before turning on the amplifier, check that the encoders work. You should be able to see the encoder counts increment and decrement on the **1D Interactive** panel in MAX if you move the motor manually.

Check the Drive

1. Check the amplifier (drive) to ensure it does not have any faults.
2. Ensure that the drive is enabled. Most drives have a reset or enable switch.
3. If you are using a Universal Motion Interface (UMI), ensure that 5 V is being supplied for encoder power.
4. If you are using brushless servo motors and the drive is handling the commutation, ensure that the Hall effect sensors and encoders are connected correctly to the drive. Refer to the drive manual for details.
5. Most servo drives need to be configured. If the servo drive comes with tuning or configuration software, ensure you have installed it and configured the drive for torque (current) or velocity mode.
   The NI motion controller always closes the position loop, so the position loop should never be closed by the drive. The velocity loop can be optionally closed on the drive (velocity amplifier). Make sure that the gains for the drive are properly adjusted for the motor you are using.
6. If you are using stepper motors, ensure that the step and direction lines are connected according to the requirements of the drive. In some cases, a pull-up resistor is required.

Testing Limit and Home Switches

Complete the following steps to test the limit and home switches:

1. Disconnect power to the motors and drive. If you have a National Instruments nuDrive or MID drive, do not turn on the **Enable** switch.
2. Launch MAX and select the motion controller that is connected to the system.
3. Initialize the motion controller by clicking **Initialize**. This step assumes that you have correctly configured the Initialization Settings and saved them by clicking **Apply**.
4. Ensure that the switches are correctly powered.
5. Open the **1D Interactive** panel in MAX.
6. Move the motor manually until it hits the limit or home switch. When the limit or home switch is active, LEDs on the main tab of the **1D Interactive** panel change from gray to red. As you move the motor away from the limit or home switches, these LEDs turn back to gray.
7. Repeat steps 1 through 6 for each motor that is attached to limit or home switches.

You have now tested the limit and home switches.

1D Interactive Test

Complete the following steps to test a 1D move in MAX:

1. Launch MAX by double-clicking the MAX icon.
2. Select the controller by clicking once on the folder icon next to the name.
3. Click **Initialize**.
4. Click **+** next to the motion controller in the MAX navigation tree to expand the controller options.
5. Expand the **Interactive** folder icon.
6. Click once on **1D Interactive**.
7. Change the number in **Target Position** to a number that is different from the one under **Current Trajectory Data**.

8. Click **Apply**.
9. Click **Start**. The motor should move to the target position.
10. Repeat steps 1 through 9 for each remaining axis by changing the axis value.

You have now tested the motion system.

Part III

Programming NI-Motion

You can use the C/C++ functions and LabVIEW VIs included with NI-Motion to exercise great control over your motion control application. Although it is not feasible to document every possible combination of functions and VIs, Part III covers the most important 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 your 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 C Reference Help* or the *NI-Motion VI Help* for detailed information on specific functions or VIs.

Part III covers the following topics:

- *Straight-Line Moves*
- *Arc Moves*
- *Contoured Moves*
- *Reference Moves*
- *Blending Your Moves*
- *Electronic Gearing*
- *Acquiring Time-Sampled Position and Velocity Data*
- *Synchronization*

- *Torque Control*
- *Onboard Programs*

What You Need to Know about Moves

This section 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 values of move constraints.

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

Trapezoidal

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 III-1.

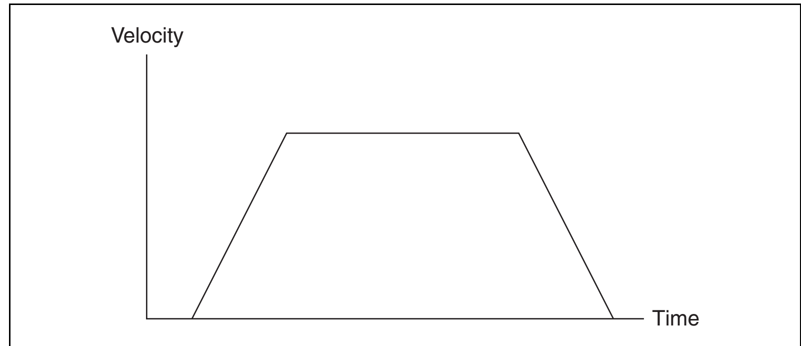


Figure III-1. Trapezoidal Move Profile

S-Curve

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

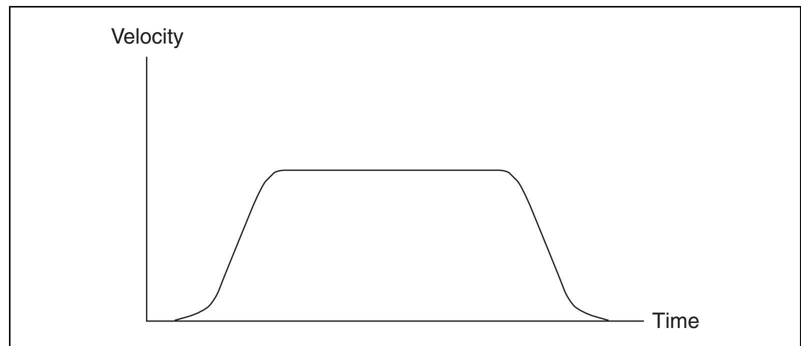


Figure III-2. S-Curve Move Profile

Basic Moves

There are four basic move types:

- Reference Move—Used to initialize the axes to a known physical reference such as a home switch or encoder index
- Straight-Line Move—Used to move from point A to point B in a straight line. The move can be based on position or velocity

- Arc Move—Used to move from point A to point B in an arc or helix
- Contoured Move—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 your chosen 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 to follow the points you have defined.

Coordinate Space (Vector Space)

With the exception of the arc move, you can execute all the 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 III-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 III-3.

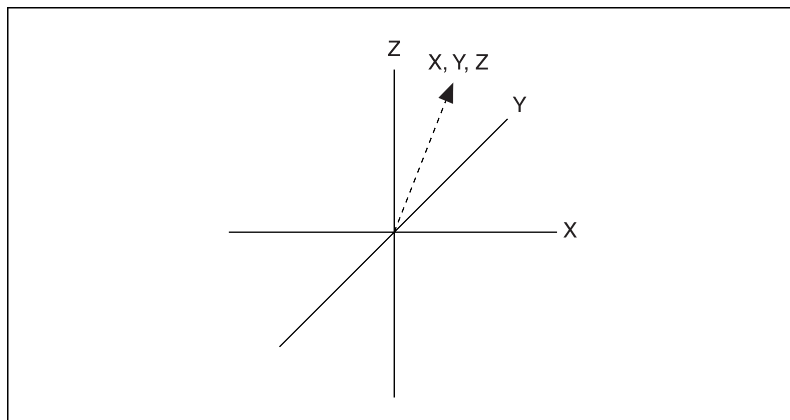


Figure III-3. 3D Coordinate Space

Use the Configure Vector Space function to configure a coordinate space. This function creates a logical mapping of axes and treats them as part of a coordinate space. Then the move generated by the trajectory generator is done on the vector and all the move constraints are treated 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 end the moves of the axes together, you must calculate the appropriate move constraints for each axis that brings it to the end of its travel with the others. In coordinate spaces, this is calculated automatically.

Trajectory Parameters

Trajectory parameters give you great control over 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 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 being used. 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 is more coarse.

There are two other factors that affect the way trajectory parameters are loaded to the NI motion controller versus how they are used by the trajectory generators: floating-point versus fixed-point parameter representation, and time base.

Floating-Point versus Fixed-Point

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. This is important when working with onboard variables, inputs, and return vectors. It also has a small effect on parameter range and resolution.

Time Base

The second factor is the 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.

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

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

where Vmin = 1/65,536 count/sample or step/sample,

Ts = sample period in seconds per sample,

60 = number of seconds in a minute, and

R = counts/steps per revolution.

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

$$\left(\frac{1}{65,536}\right) \times 4,000 \times \left(\frac{60}{2,000}\right) = 0.00183105 \text{ RPM}$$

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

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

where Vmax = 20 MHz for servos,

8 MHz for steppers on a 735*x* controller,

4 MHz for steppers on a 734*x* or 743*x* controller, and

R = counts/steps per revolution,

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

$$\text{velocity} \le (65536 \times \text{deceleration}) - \text{acceleration}$$

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

From the example, the maximum RPM is:

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

RPM values stored in onboard variables are in double-precision IEEE format (f64). For information about the number of variables required to hold an RPM value, refer to Onboard Variables, Input, and Return Vectors.

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:

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

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

Ts = sample period in seconds per sample.

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

$$\left(\frac{1}{65{,}536}\right) \times 16{,}000 = 0.244 \text{ counts or steps/s}$$

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:

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

where Vmax = 20 MHz for servos,
8 MHz for steppers on a 735*x* controller,
4 MHz for steppers on a 734*x* or 743*x* controller,

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

$$\text{velocity} \le (65536 \times \text{deceleration}) - \text{acceleration}$$

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

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

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

where Amin = 1/65,536 counts/sample^2 or steps/sample^2, and
Ts = sample period in seconds per sample.

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

$$\left(\frac{1}{65,536}\right) \times \left(\frac{4,000^2}{2,000}\right) = 0.122070 \text{ counts/s}$$

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

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

where Amax = 32 counts/sample

Ts = sample period in seconds per sample,

and is constrained according to the following equations:

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

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

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

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

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

Ts = sample period in seconds per sample, and

R = counts or steps per revolution.

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

$$\left(\frac{1}{65{,}536}\right) \times \left(\frac{4{,}000^2}{2{,}000}\right) = 0.122070 \text{ RPS/s}$$

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

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

where Amax = 32 counts/sample

Ts = sample period in seconds per sample,

R = counts or steps per revolution,

and is constrained according to the following equations:

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

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

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

Velocity Override in Percent

Whereas the Load Velocity Override function takes a single-precision floating-point (f32) data value from 0 to 150%, velocity override is internally implemented as a velocity scale factor of 0 to 384 with an implicit fixed denominator of 256. This is done for the sake of calculation speed—the division is a simple 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 your commanded maximum move constraints. In such instances, NI-Motion adjusts your move constraints lower to reach your commanded position.

Arc Angles in Degrees

The Load Circular Arc, Load Helical Arc, and Load Spherical Arc functions take their 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° and

R = remainder.

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

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

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

The minimum angular increment is therefore:

$$\left(\frac{1}{65{,}536}\right) \times 45° = 0.000687°$$

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.

Arc Move Limitations

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

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

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

and

$$1677216 \geq \frac{(V \times P^2 \times 83443)}{R \times I} \geq 16$$

where V = Velocity in counts/s,

P = PID sample rate in seconds,

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

R = Radius in counts.

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

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

and

$$65536 \geq \frac{A \times P^3 \times 83443}{R \times I^2} \geq 1$$

where P = PID sample rate in seconds,

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

R = Radius in counts, and

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

Timing Your Loops

National Instruments recommends the following loop timings for NI-Motion applications:

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 only detect flicker 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 with in 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 will provide enough resolution for typical applications. Consider how accurate your 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, suppose you are performing a scan routine on an object where a user places the object under the scan area and clicks a **Scan** button. To synchronize the motion with the acquisition, you create periodic breakpoints every 10 counts to trigger a data acquisition over RTSI. In this example, the loop only needs 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. Therefore, since 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 move complete as well.

6

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 desired target position to generate the move trajectory. For example, if the motor is currently at position zero, and the target position is 100, then a position-based move creates a trajectory that moves 100 counts or steps.

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

- Start position—The 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

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.

Algorithm

Do the following to start a move:

1. Load Target Position—Specifies the end position
2. Load the move constraints—Loads the velocity, acceleration, deceleration and jerk values
3. 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 relative position to the starting position. Notice that, although you can

update any parameter while the move is in progress, the new parameter is used only after a subsequent Start or Blend Motion.

Tip It is necessary to load the move constraints only if they need to be different from what was previously loaded.

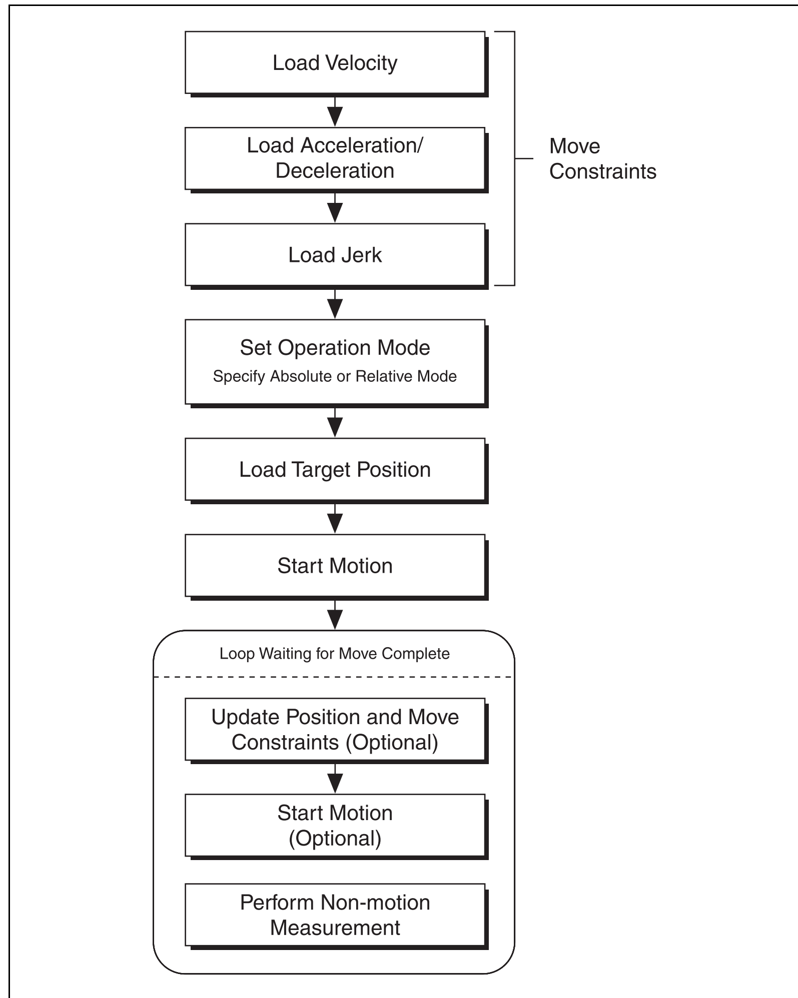


Figure 6-1. Position-Based Straight-Line Move Algorithm

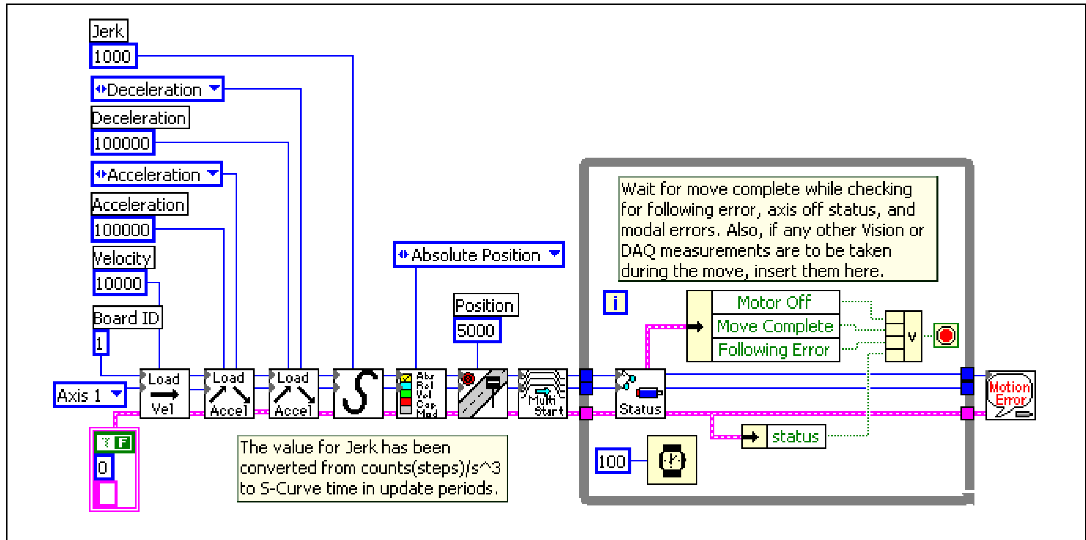


Figure 6-2. D Straight-Line Move in LabVIEW

NI-Motion VIs for Figure 6-2, in order from left to right:

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

Figure 6-3. 2D Straight-Line Move in LabVIEW

NI-Motion VIs for Figure 6-3, in order from left to right:

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

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 more complete and can compile.

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 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
}
```

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 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
}
```

Velocity-Based Straight-Line Moves (Jogging/Velocity Profiling)

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

You can move 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 6-4 is a generic algorithm applicable to both C/C++ and VI code.

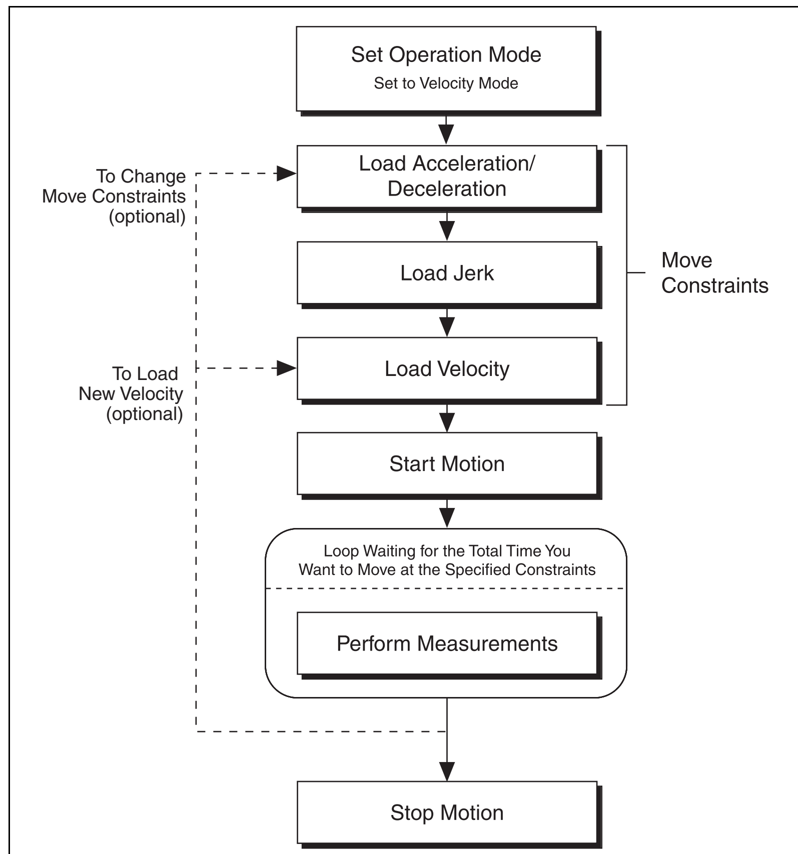


Figure 6-4. Velocity-Based Straight-Line Move Algorithm

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 6-5.

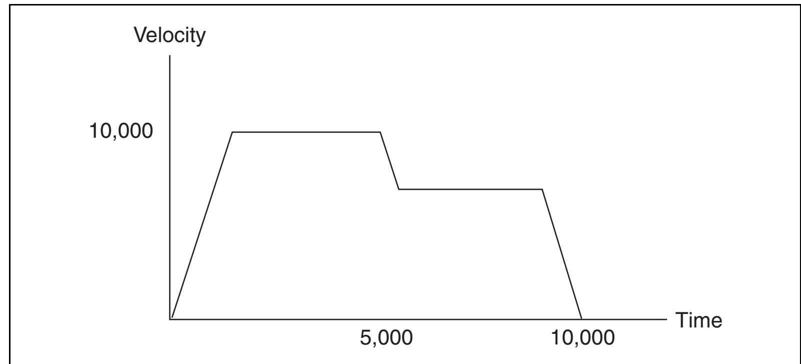


Figure 6-5. Velocity Profile

LabVIEW Code

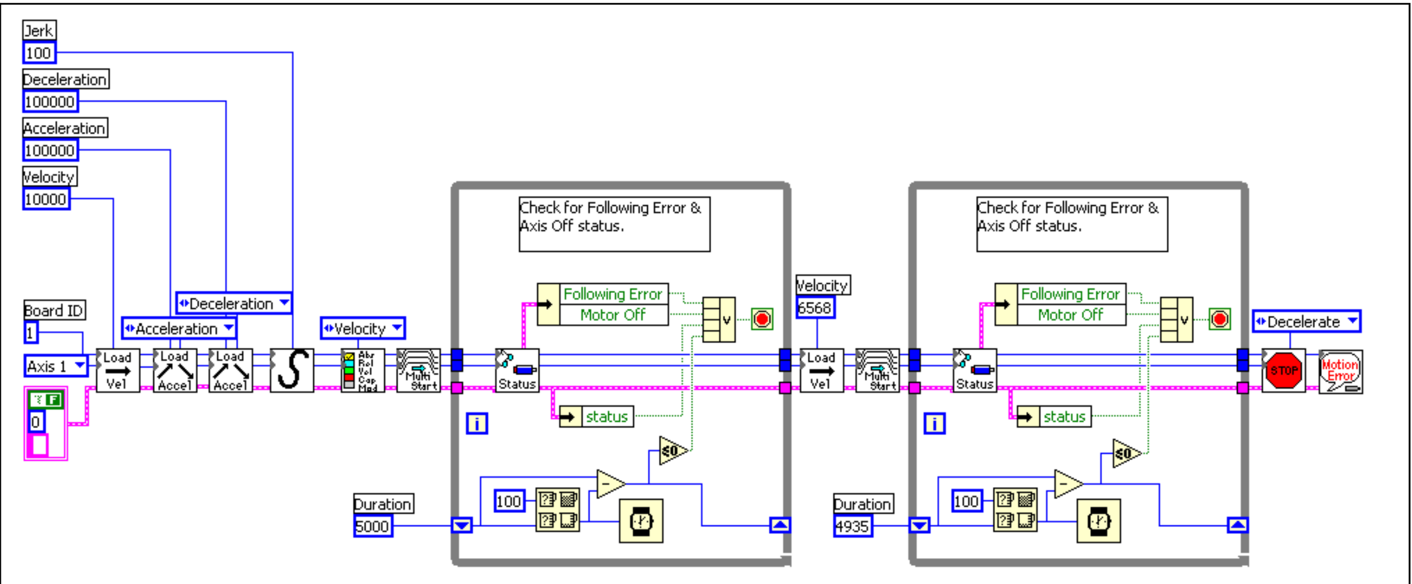


Figure 6-6. Velocity-Based Straight-Line Move in LabVIEW

NI-Motion VIs for Figure 6-6, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

Velocity Profiling Using Velocity Override

Velocity profiling using the Load Velocity Override function provides another way to shift from one velocity to another while executing moves. However, instead of explicitly stating the velocity you want to change to, you indicate the new velocity in terms of a percentage of the originally loaded velocity.

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

The transition between velocities obeys all other move constraints.

Algorithm

Figure 6-7 is a generic algorithm applicable to both C/C++ and VI code.

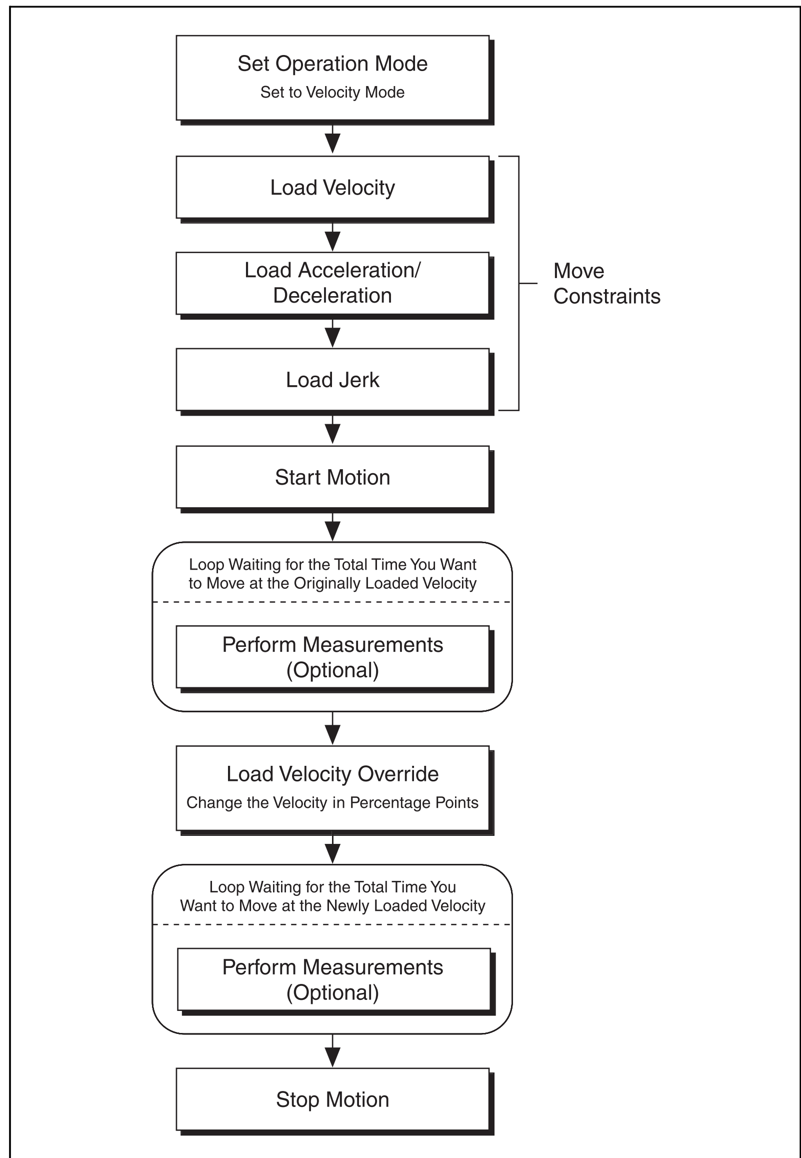


Figure 6-7. Velocity Override Algorithm

LabVIEW Code

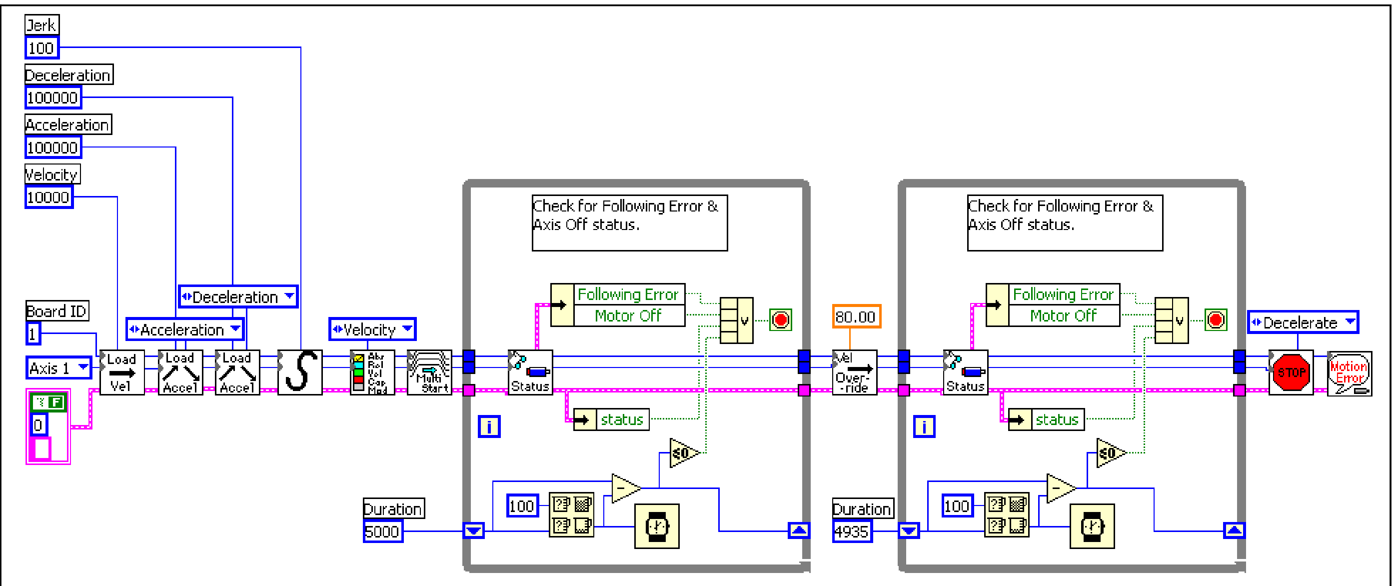


Figure 6-8. Velocity-Based Move Using Velocity Override in LabVIEW

NI-Motion VIs for Figure 6-8, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

7

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 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 7-1.

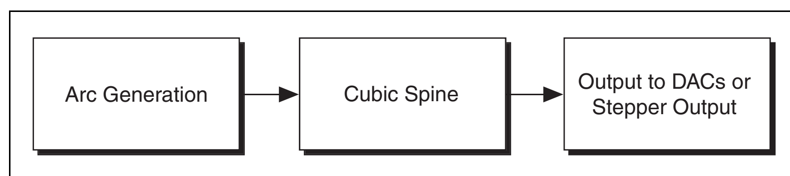


Figure 7-1. Arc Move Data Path

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 and, like all coordinate space moves, uses the values of move constraints—maximum velocity, maximum acceleration, and maximum deceleration.

Jerk is not used to calculating the profile for arc moves. While moving in an arc you can only use trapezoidal profiles.

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

- Radius—The radius of an arc is the distance from the center of the arc to its edge.

- Start Angle—The start angle orients the arc on its plane using the starting point as an axis to spin around. Since 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 7-2 shows the effect of changing the start angle from 0° to 180°.

1 Original Arc	2 Arc with 180° Start Angle

Figure 7-2. Rotating Start Angle

- Travel 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 7-3.

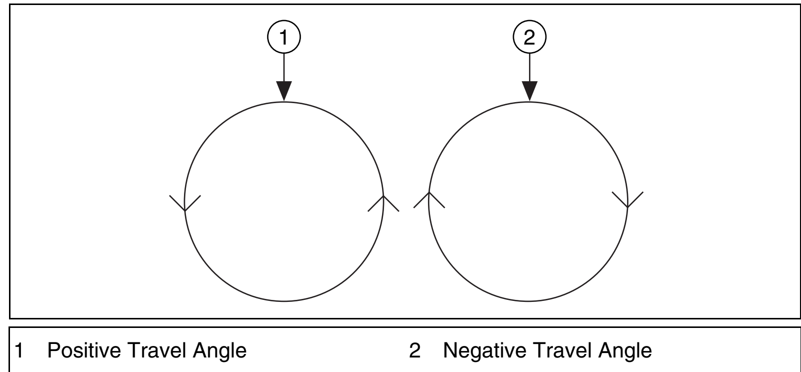


Figure 7-3. Positive and Negative Travel Angles

Algorithm

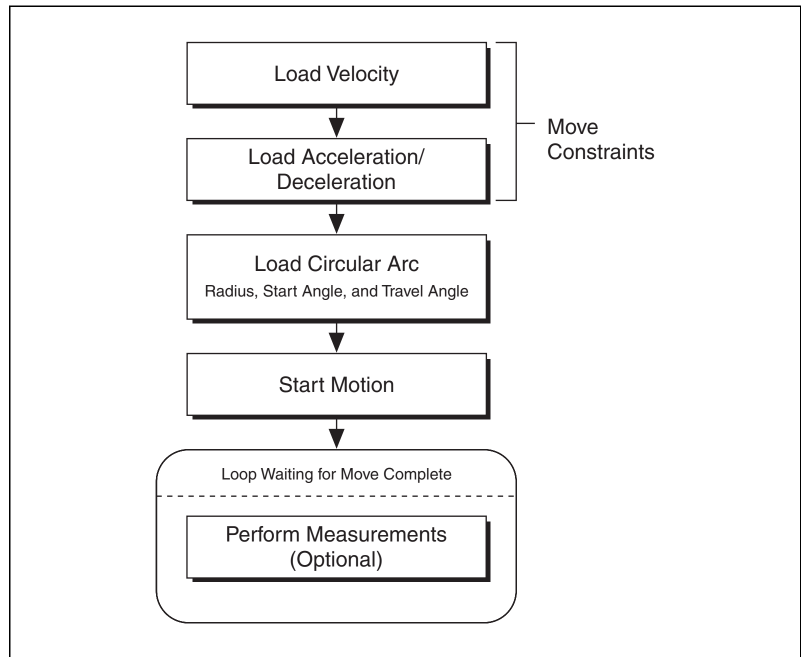


Figure 7-4. Circular Arc Move Algorithm

LabVIEW Code

Figure 7-5. Circular Arc Move in LabVIEW

NI-Motion VIs for Figure 7-5, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

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 7-6 and 7-7.

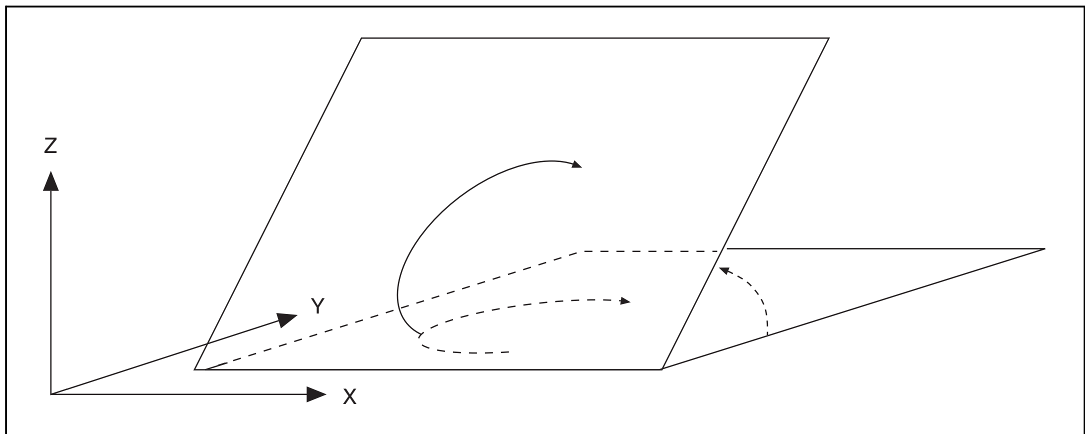


Figure 7-6. Changing Pitch by Rotating the X Axis

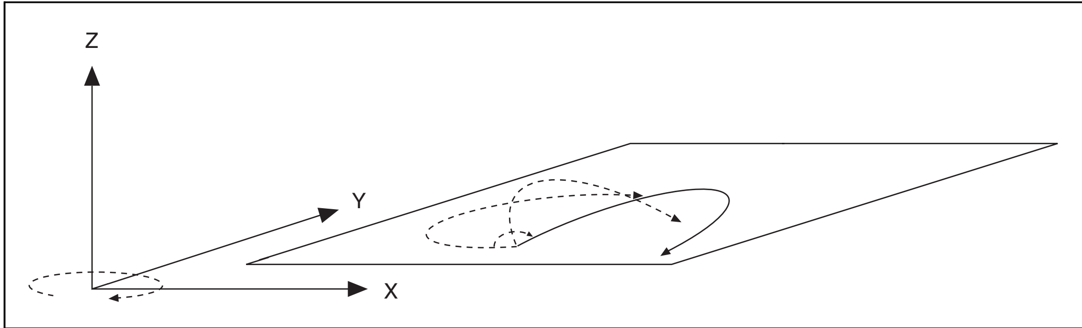


Figure 7-7. Changing Yaw by Rotating the Z Axis

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 with respect 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 in case of a spherical arc that defines the arc in two dimensions.

Algorithm

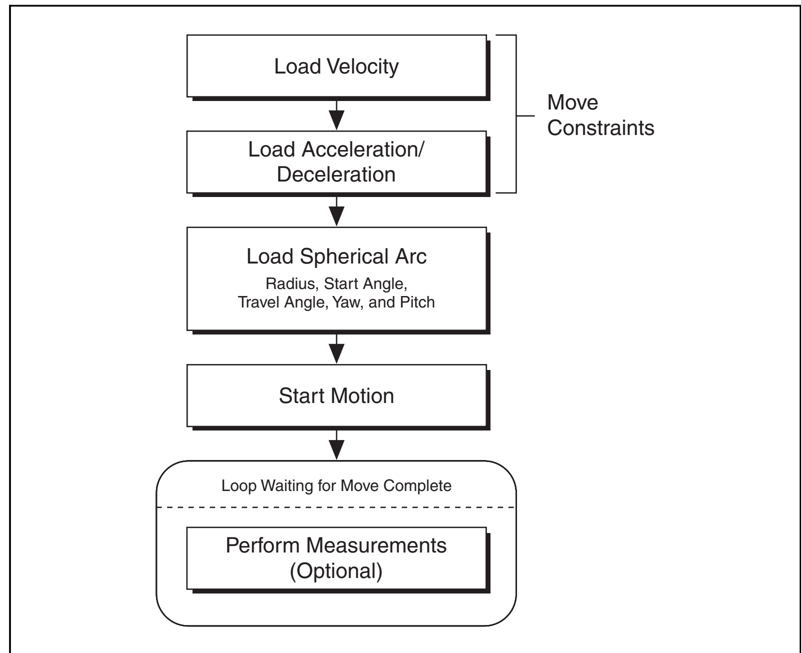


Figure 7-8. Spherical Arc Algorithm

LabVIEW Code

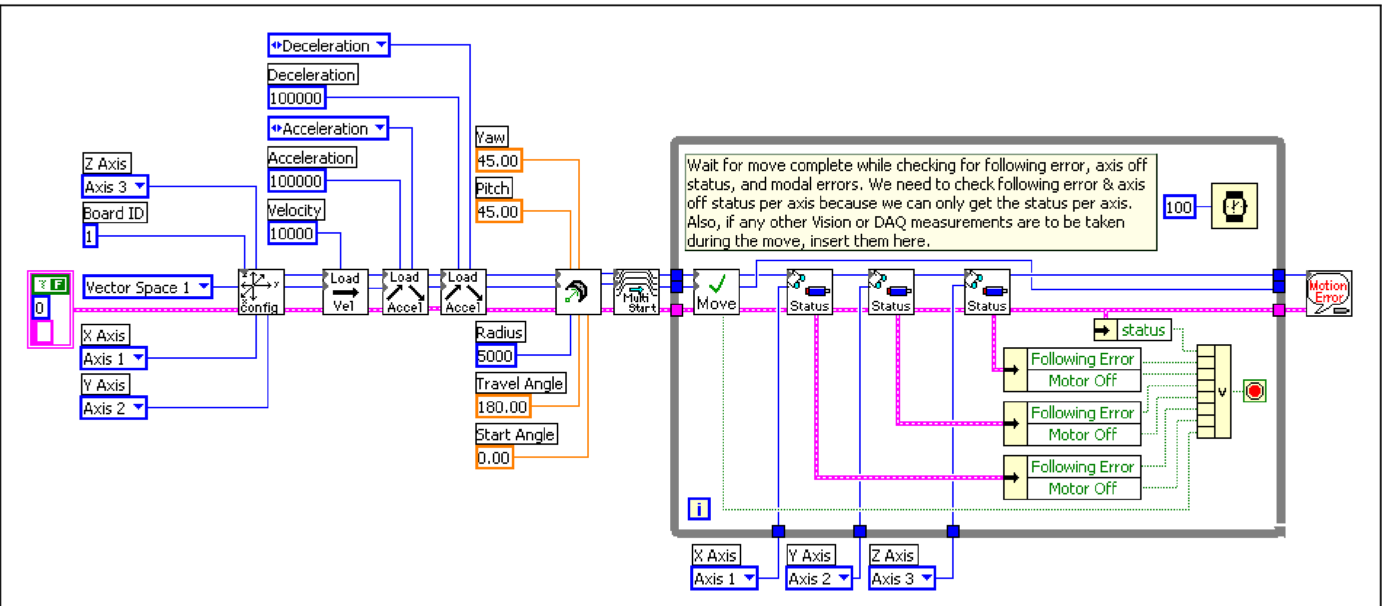


Figure 7-9. Spherical Arc Move in LabVIEW

NI-Motion VIs for Figure 7-9, in order from left to right:

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

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 more complete and can compile.

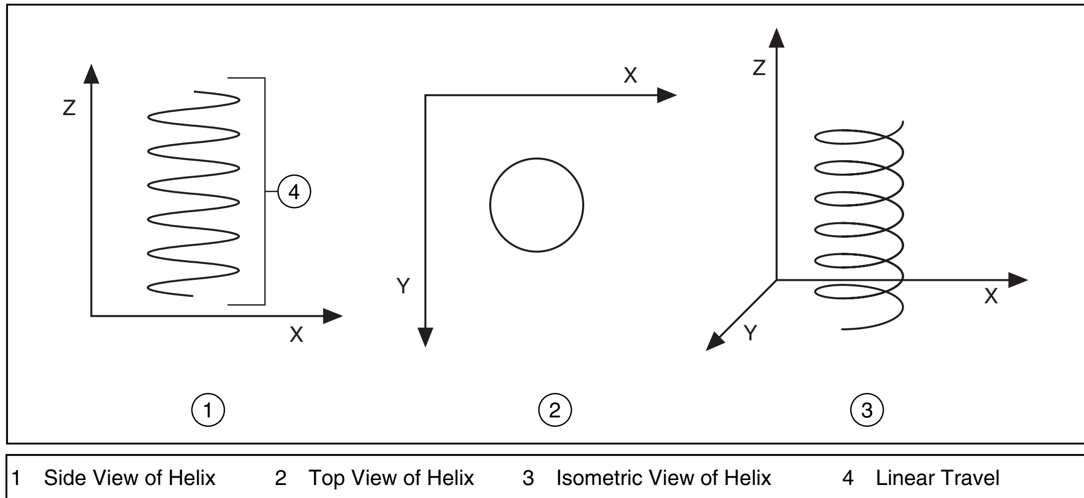
```
// Main Function
void main(void)
{
    u8 boardID;// Board identification number
    u8 vectorSpace;// Vector space number
    u16 csr = 0;// Communication status register
    u16 axisStatus;// Axis status
    u16 status;
    u16 moveComplete;

    //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 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
}
```

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 7-10.



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

Figure 7-10. Helical Arc

Algorithm

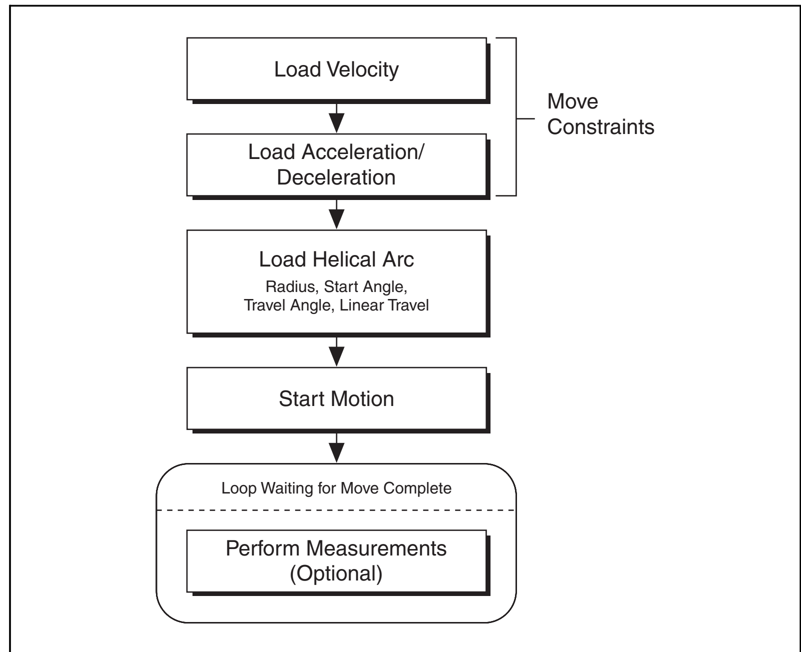


Figure 7-11. Helical Arc Algorithm

LabVIEW Code

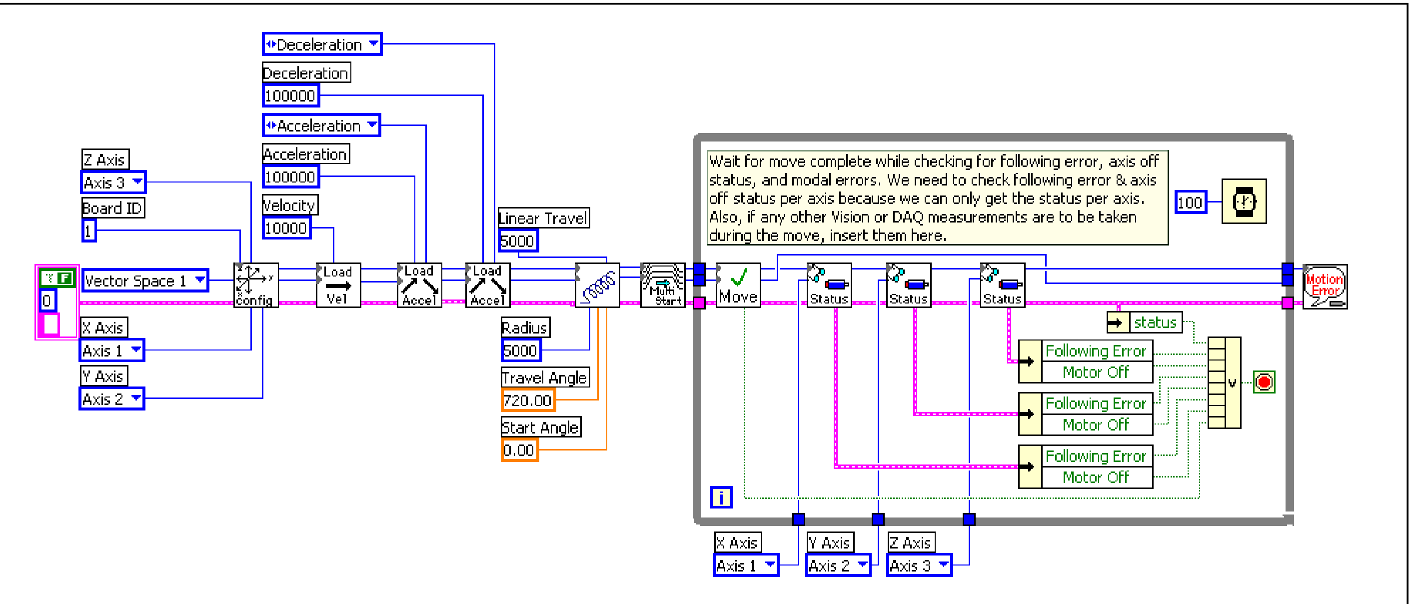


Figure 7-12. Helical Arc Move in LabVIEW

NI-Motion VIs for Figure 7-12, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

8 Contoured Moves

A contoured move causes an axis or a coordinate space of axes to move in a pattern that you define. The trajectory generator on the motion controller is not used during a contoured move. The controller instead 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 8-1.

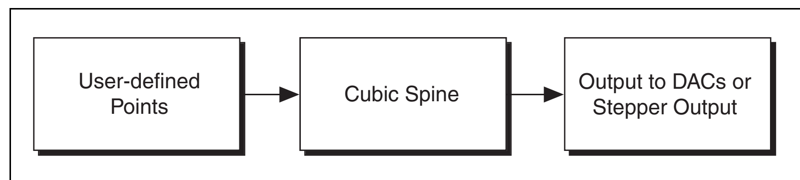


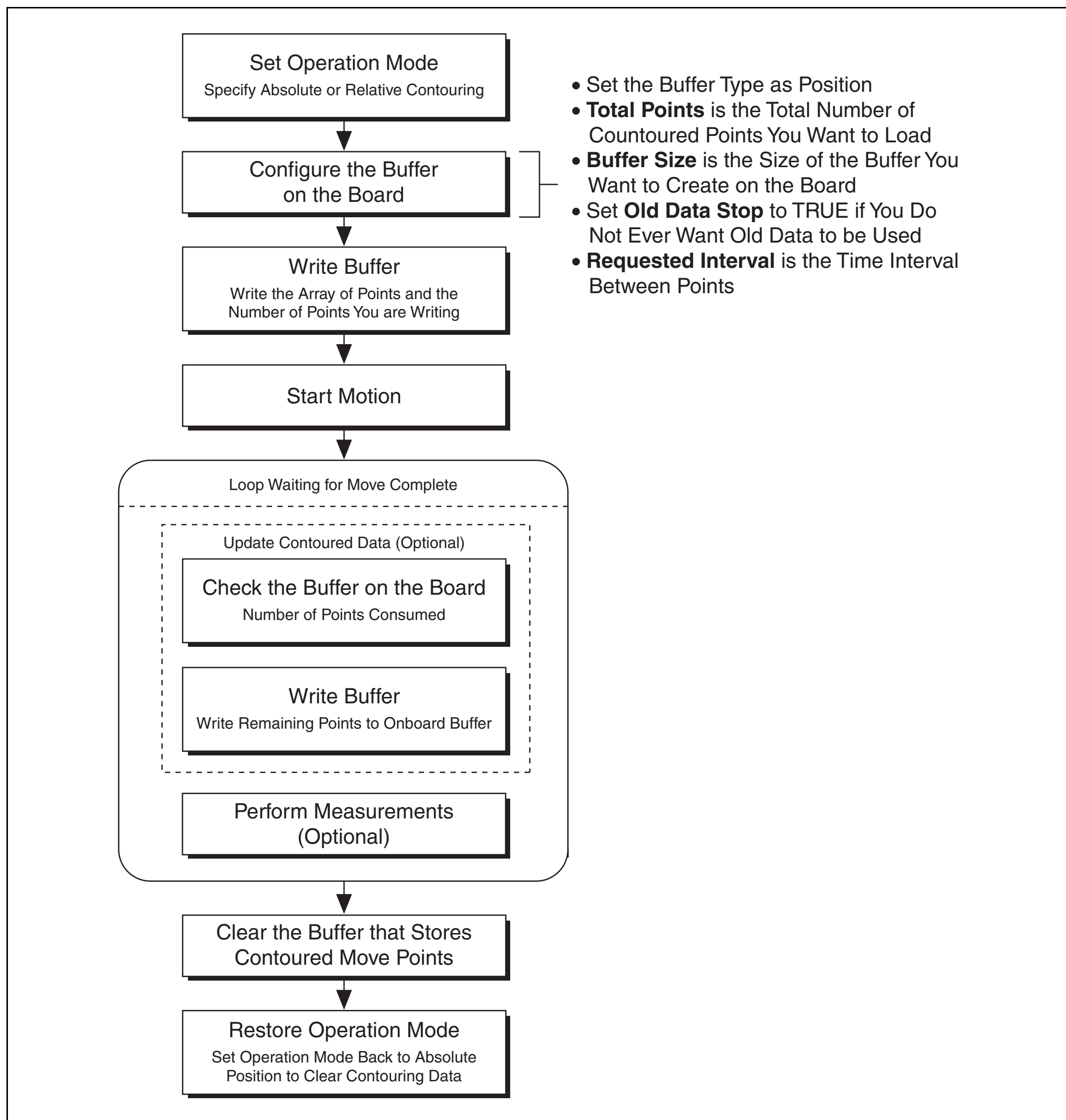
Figure 8-1. Contoured Move Data Path

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 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 take a user-defined trajectory and re-map it based on your desired move constraints, preserving move characteristics and move geometry.

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 is similar to the way arc moves work. Depending on the operation mode you configure, 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 8-2. Absolute Contouring Buffer Values

1	2	3	4	4	4	4	3	2	1

Figure 8-3. Relative Contouring Buffer Values

LabVIEW Code

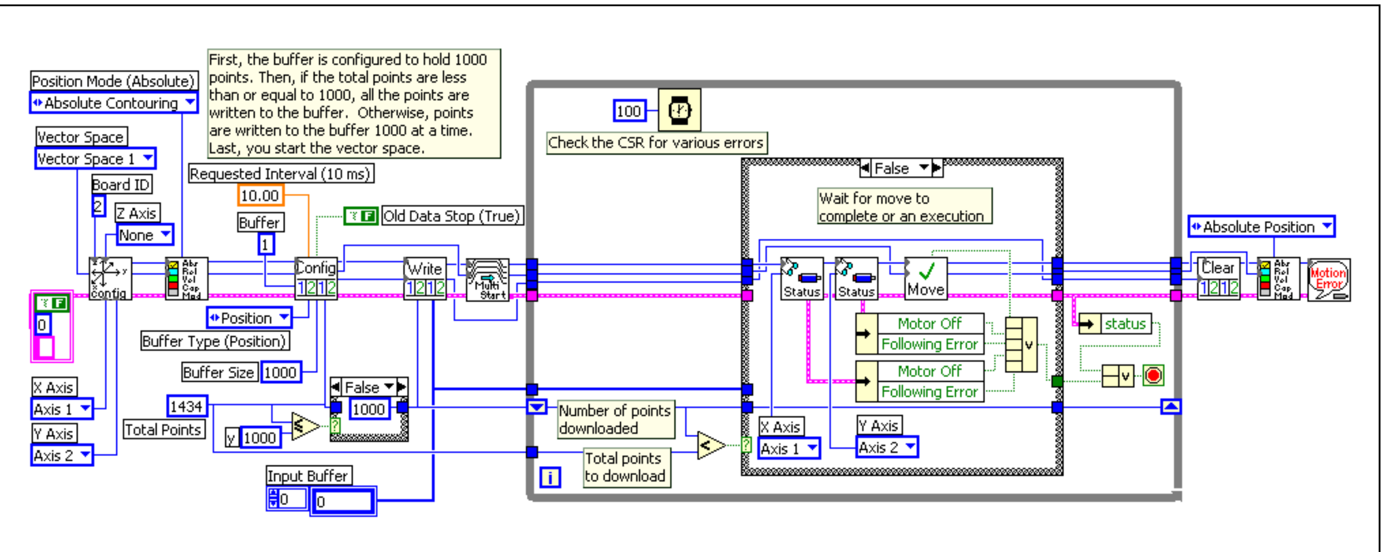


Figure 8-4. Contoured Move in LabVIEW

NI-Motion VIs for Figure 8-4, in order from left to right:

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

1 Check Buffer	2 Write Buffer

Figure 8-5. 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 more complete and can compile.

```
// 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 controller can
                         // really contour
    i32* downloadData = NULL;// The temporary array that is created to
                              // download the points to the 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 controller out of
    // contouring mode
    err = flex_set_op_mode(boardID, vectorSpace,
                           NIMC_ABSOLUTE_POSITION);
    CheckError;
    // Free the buffer allocated on the controller memory
    err = flex_clear_buffer(boardID, 1/*buffer number*/);
    CheckError;
    return;// Exit the Application
    //////////////////////
    // Error Handling
    //
    nimcHandleError; //NIMCCATCHTHIS:
    // Check to see if there were any Modal Errors
    if (csr & NIMC_MODAL_ERROR_MSG){
        do{
            // Get the command ID, resource ID, 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
}
```

9

Reference Moves

Use reference moves to move the axes to a known starting location and orientation. Reference tools 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 C Reference Help* for more information about reference move options.

Algorithm

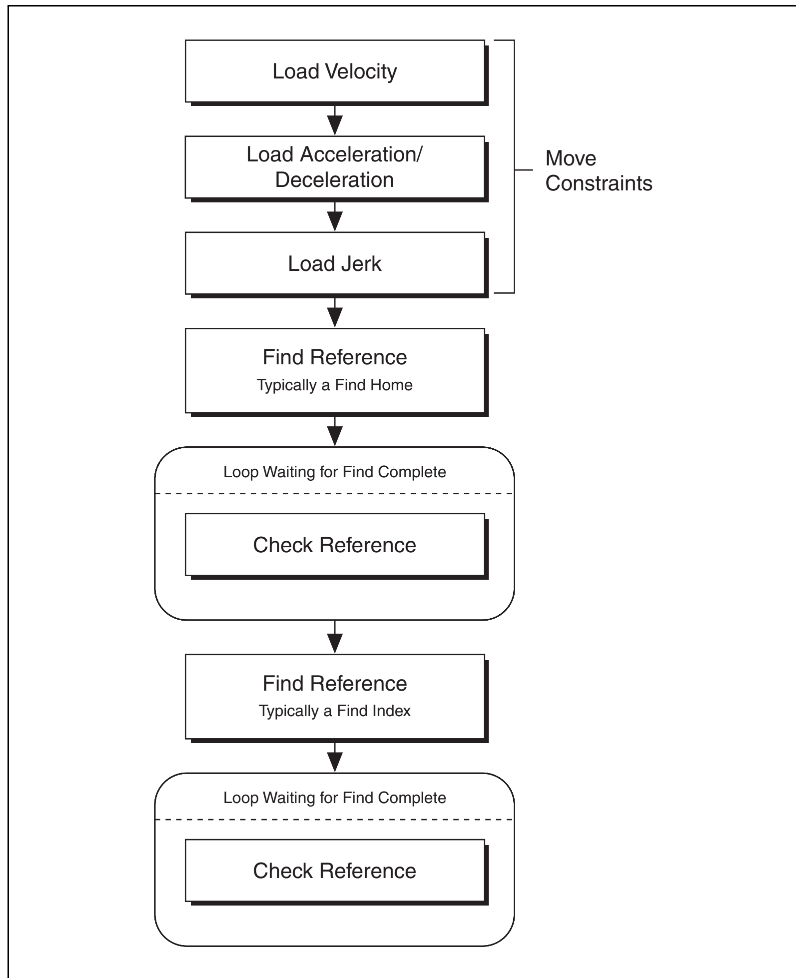


Figure 9-1. Find Reference Move Algorithm

LabVIEW Code

Figure 9-2. Find Reference Move in LabVIEW

NI-Motion VIs for Figure 9-2, in order from left to right:

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

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 more complete and can compile.

```
// 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 board is at power up reset condition
    err = flex_read_csr_rtn(boardID, &csr);
    CheckError;

    if (csr & NIMC_POWER_UP_RESET ){
        printf("\nThe FlexMotion board is in the reset condition.
        Please initialize the board before ");
        printf("running this example.  The
        \"flex_initialize_controller\" function will initialze 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;

   //Dislay 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 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
}
```

10

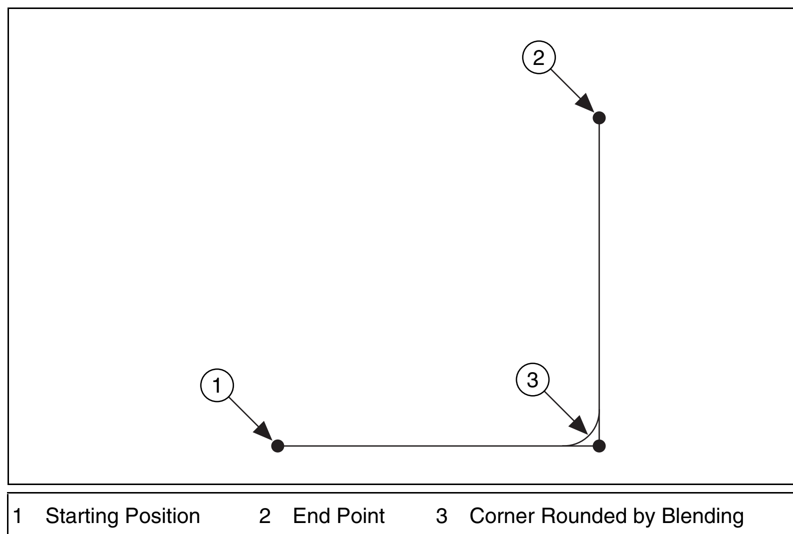
Blending Your Moves

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. This 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 10-1.



1 Starting Position	2 End Point	3 Corner Rounded by Blending

Figure 10-1. Two Blended Straight-Line Moves

The controller 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 points where you can start the second move in a blend: superimposing the two moves by starting the second move as the first move starts to decelerate, starting the second move after the first profile is complete, and starting the second move after the first profile is complete plus an added delay. Refer to the *Move Profiles* section of Part III, *Programming NI-Motion*, for more information about move profiles.

Superimpose Two Moves

This is the most common way blending is used. In this case, the controller tries to maintain continuous motion by superimposing the two move segments being blended such that the second move segment starts its profile while the first move is decelerating

The second move segment starts while the first one is still in progress, as shown in Figure 10-2.

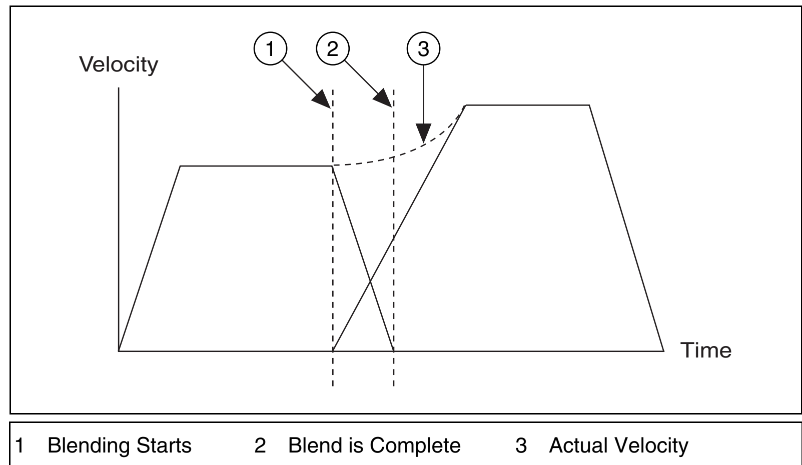


Figure 10-2. Superimposing Two Moves

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

This type of blending causes the first move segment to come to a complete stop before starting the profile of the second segment, as shown in Figure 10-3.

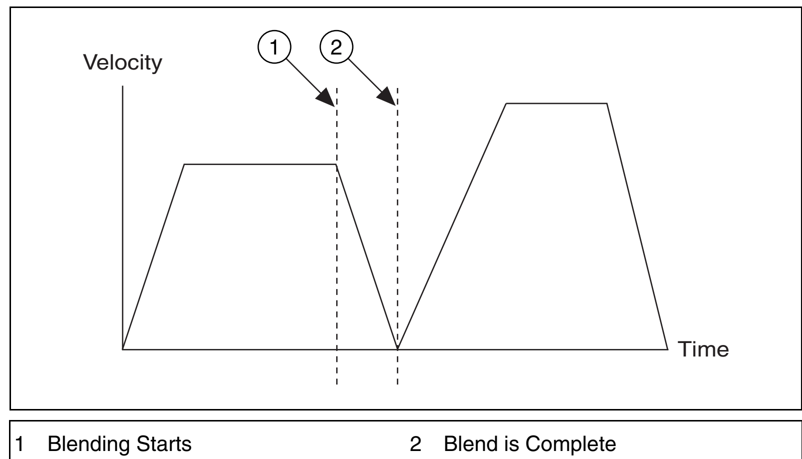


Figure 10-3. Blending after Move Complete

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

Blend after Delay

You can specify a delay between the two blended move segments, as shown in Figure 10-4.

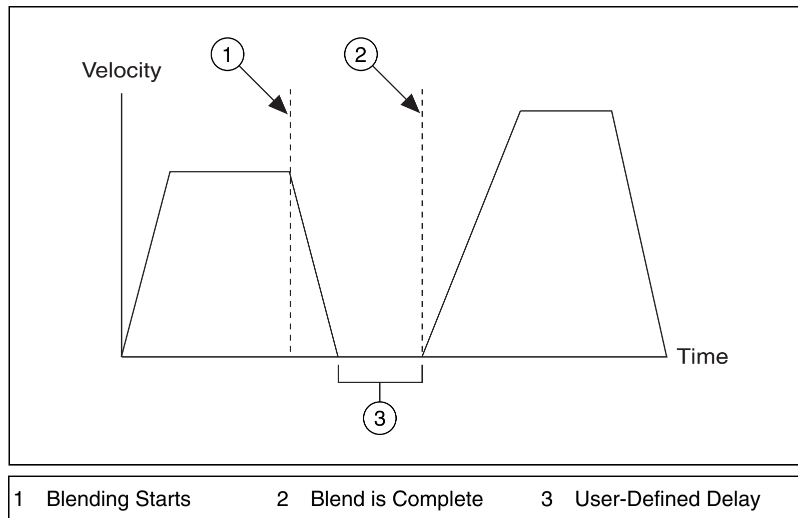


Figure 10-4. Blending after a Delay

Blending in this manner is useful if you want to start two move segments after a very 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.

Algorithm

Figure 10-5 diagrams a generic algorithm for blending moves.

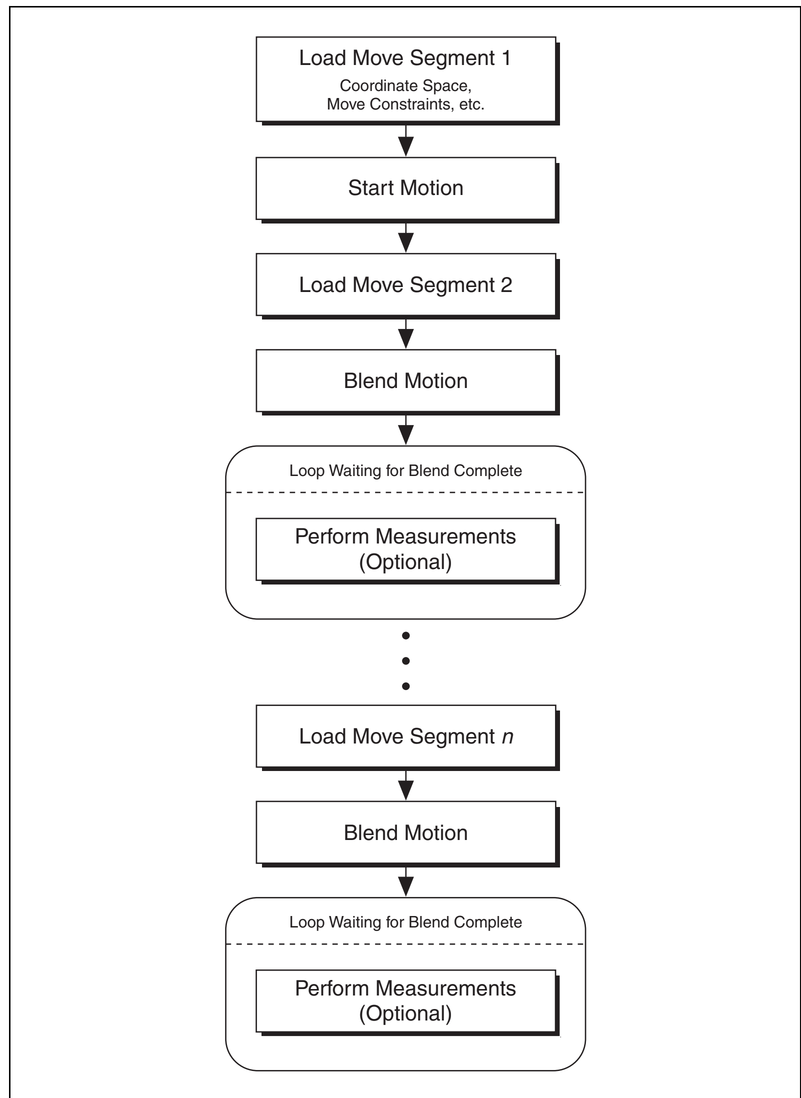


Figure 10-5. Blending Algorithm

LabVIEW Code

Figure 10-6. Blended Straight-Line Move and Arc Move in LabVIEW

NI-Motion VIs for Figure 10-6, in order from left to right:

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

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 more complete and can compile.

```
// 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 scurve 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 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 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

Electronic Gearing

Use electronic gearing to link the movement of one or more slave axes to the movement of a designated master axis. The movement of the slave axes may be at a higher or lower gear ratio than the master axis. For example, every turn of the master axis can cause a slave axis to turn twice.

Gearing

Electronic gearing allows one slave motor to be driven in proportion to the position and movement of 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 obvious is the flexibility, as you can change the gear ratios at any time. The second advantage is that you can superimpose motion over a geared axis. The superimposed move is always added to the move profile of the slave axis. This ability allows for on-the-fly synchronization of the slave axis.

Algorithm

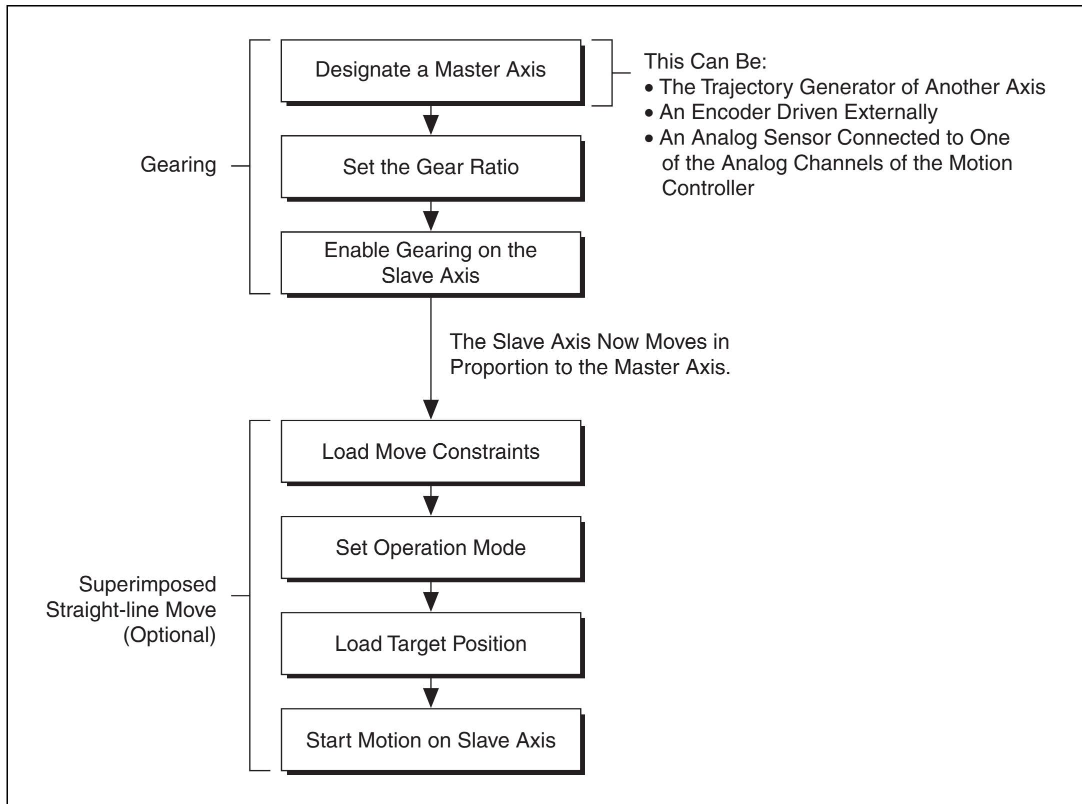


Figure 11-1. Electronic Gearing Algorithm

The motion controller can treat loaded gear ratios as absolute or relative. An absolute gear ratio uses the position of the master axis when gearing is enabled as a fixed offset as long as gearing remains enabled. The motion controller uses the following formula to calculate the position of the slave axis for absolute gearing:

$$\text{Slave axis position} = (\text{Master axis position} - \text{Master offset}) \times \text{Gear ratio}$$

A relative gear ratio allows you to change the gear ratio on the fly. Both the master and slave axes use their respective offsets at the time gearing is enabled. The motion controller uses the following formula to calculate the position of the slave axis for relative gearing:

$$\text{Slave axis position} = (\text{Master axis position} - \text{Master offset}) \times \text{Gear ratio} + \text{Slave offset}$$

Gearing can be enabled or disabled on the fly for the slave axis.

When you execute the Enable Gearing function, the positions of the slave and its master are recorded as their absolute gearing reference. From then on, as long as the gear ratio remains absolute, every incremental change of the master position is multiplied by the absolute gear ratio and applied to the slave axis or axes.

If you select and load a relative gear ratio after gearing is enabled, the position of the master is recorded as its relative reference point, and every incremental change from this reference point is multiplied by the relative gear ratio and applied to the slave axis or axes.

Caution Although you can change an absolute gear ratio on the fly, be aware that the slave axis will jump with full torque to the position defined by the new ratio even when the master position has not changed.

Gear Master

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

When the master axis is an open-loop stepper, the slave axis is geared to the trajectory output of the master axis.

If the master axis is a closed-loop stepper or servo, the slave is geared to the feedback device of the master axis.

When an axis is geared to an encoder resource, the slave axis tracks the position of the encoder itself.

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

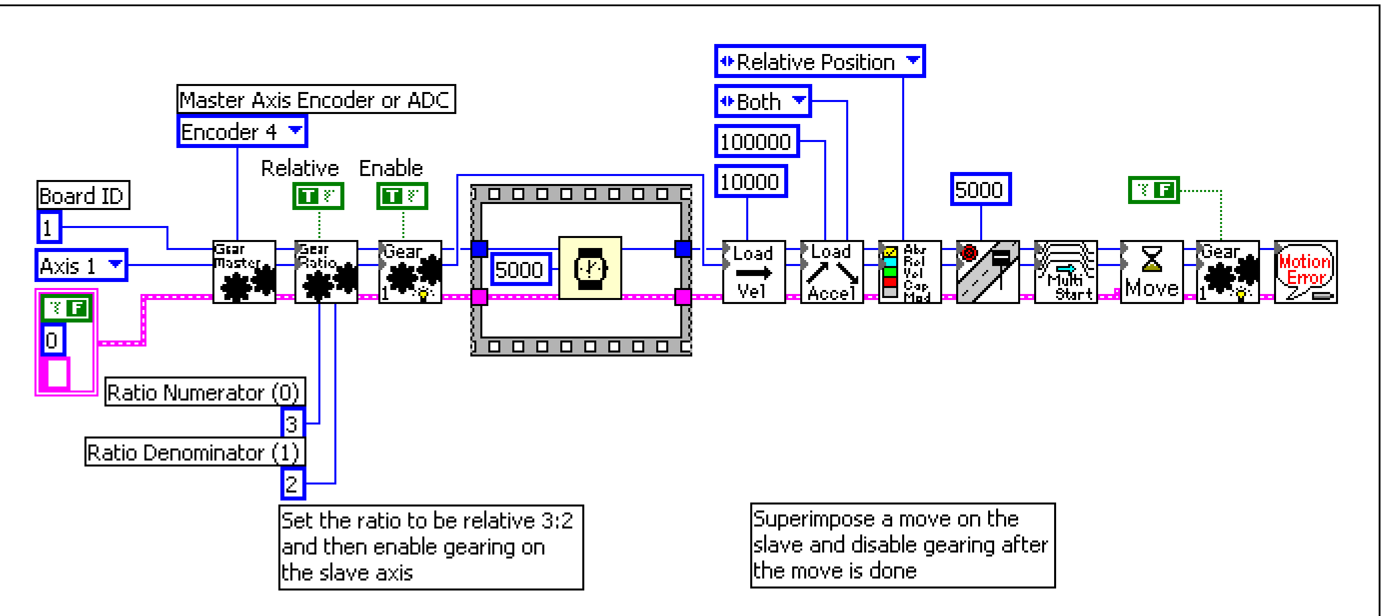


Figure 11-2. Tracking an Encoder Using Electronic Gearing with Superimposed Move

NI-Motion VIs for Figure 11-2, in order from left to right:

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

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 more complete and can compile.

```
// Main Function
void main(void)
{
    u8 boardID; // Board identification number
    u8 slaveAxis; // Slave axis number
    u8 master; // Gear master
    u16 csr = 0; // Communication status register
    u16 moveComplete;

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

    //////////////////////////////
    // Set the board ID
    boardID = 1;
    // Set the axis number
    slaveAxis = 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 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
}
```

12

Acquiring Time-Sampled Position and Velocity Data

NI motion controllers can acquire a buffer of position and velocity data that is firmware-timed. Once 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 controller. You can read data in from this buffer asynchronously from the host computer, as shown in Figure 12-1.

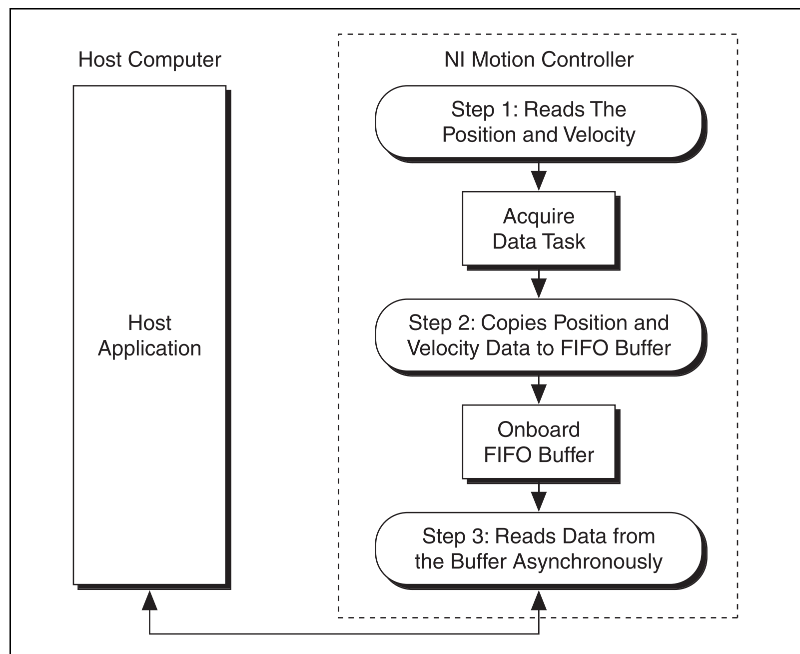


Figure 12-1. Acquire Data Path

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

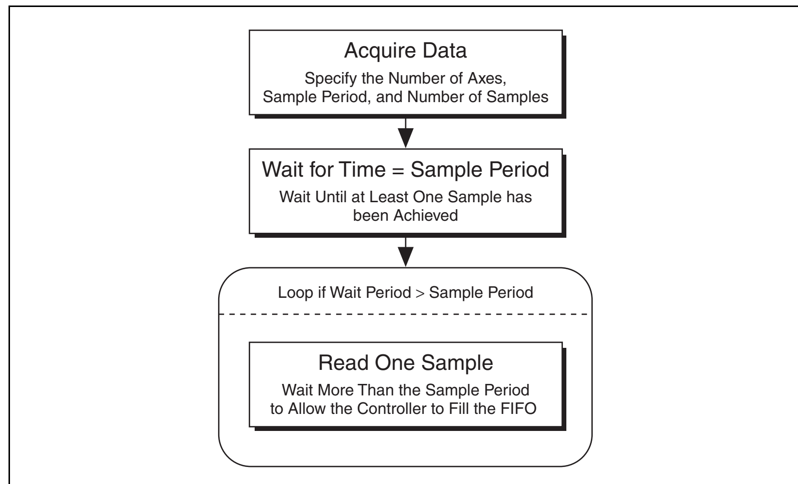


Figure 12-2. Acquire Data Algorithm

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 12-3 acquires data for two axes, 200 samples, and three milliseconds apart.

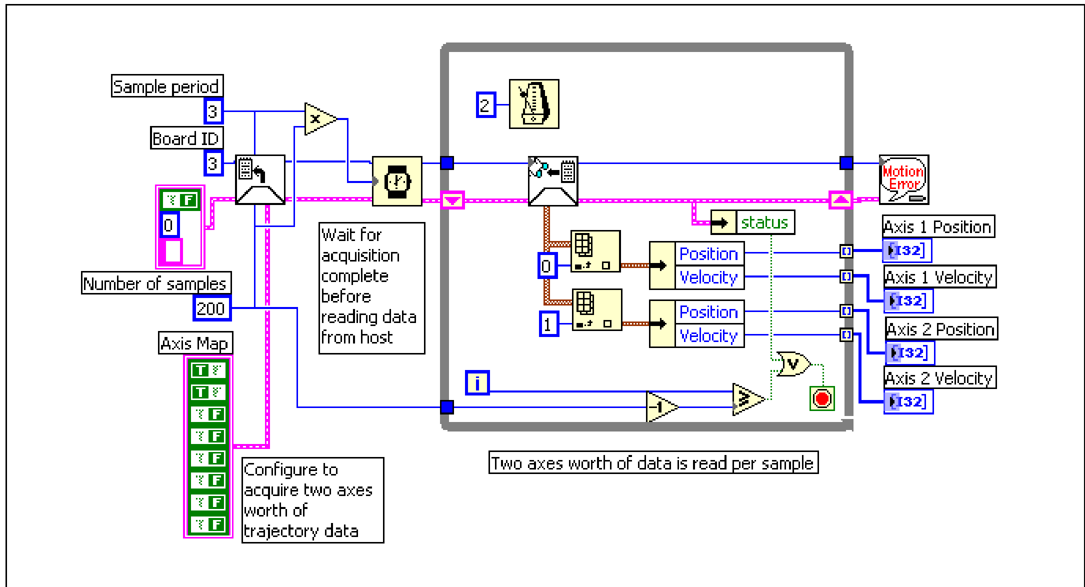


Figure 12-3. Acquire Data Using LabVIEW

NI-Motion VIs for Figure 12-3, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

13 Synchronization

NI motion controllers synchronize with data acquisition (DAQ) and image acquisition (IMAQ) devices using breakpoints and high-speed captures.

Timing and triggering with NI-Motion is always related to either position or velocity. It is important to synchronize position and velocity information with the external world so you can coordinate your measurements with your moves. You can program the motion controller to trigger another device at desired 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 via RTSI or externally using a pin on the Motion IO connector. When triggered, the motion controller can latch the current position of the encoder, which can be read and recorded.

Table 13-1 shows the availability of breakpoint modes on each NI motion controller.

Table 13-1. Breakpoint Modes on NI Motion Controllers

Breakpoint Mode	NI-735*x*	NI-734*x* and NI-733*x*
Absolute*	Y	Y
Relative*	Y	Y
Periodic	Y	N

Table 13-1. Breakpoint Modes on NI Motion Controllers (Continued)

Breakpoint Mode	NI-735*x*	NI-734*x* and NI-733*x*
Modulus	N	Y
Buffered	Y	N
* Available in buffered and single operation for NI-735*x* and in single operation only for all other controllers		

Absolute Breakpoints

Absolute position breakpoints allow you to trigger external activities as the motors reach specified positions. For example, suppose an image acquisition (IMAQ) device needs to capture an image from a certain position while the device under test is in continuous motion. The motion controller needs to be able to trigger the IMAQ 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 re-enabling is done for you automatically by the motion controller.

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 system. For example, if you have a very small breakpoint window, the jitter in the motion 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 (735*x* 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 a higher bandwidth.

Algorithm

Figure 13-1 shows the basic algorithm for implementing buffered breakpoints.

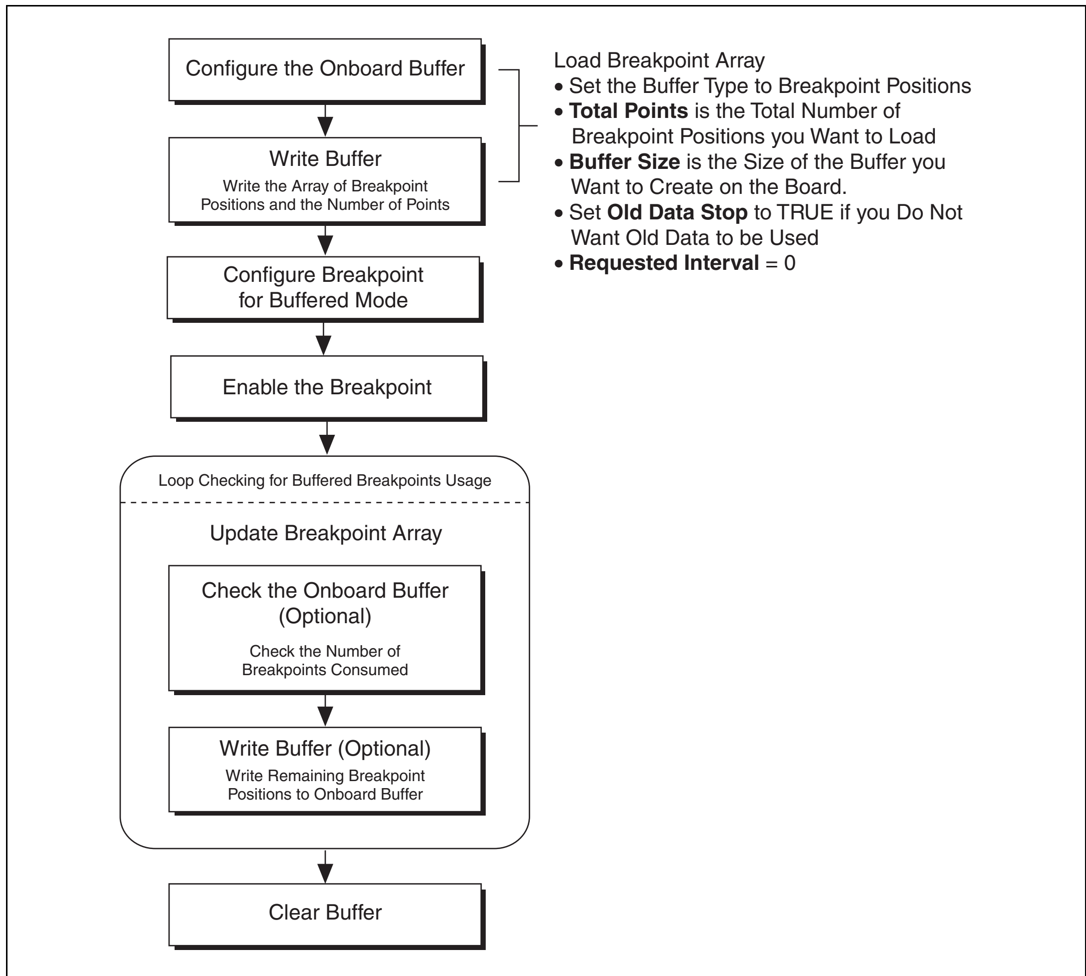


Figure 13-1. Buffered Breakpoint Algorithm

LabVIEW Code

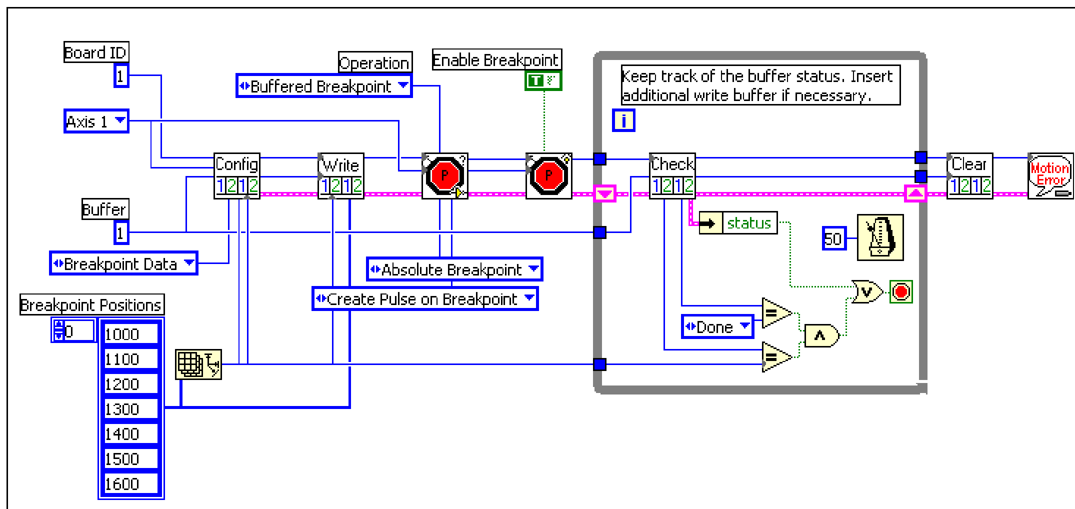


Figure 13-2. Buffered Position Breakpoint in LabVIEW

NI-Motion VIs for Figure 13-2, in order from left to right:

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

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 more complete and can compile.

```
// 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 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

}
```

Single Position Breakpoints

Single position breakpoints execute one breakpoint per enabling.

Algorithm

Figure 13-3 shows the basic algorithm for implementing position breakpoints.

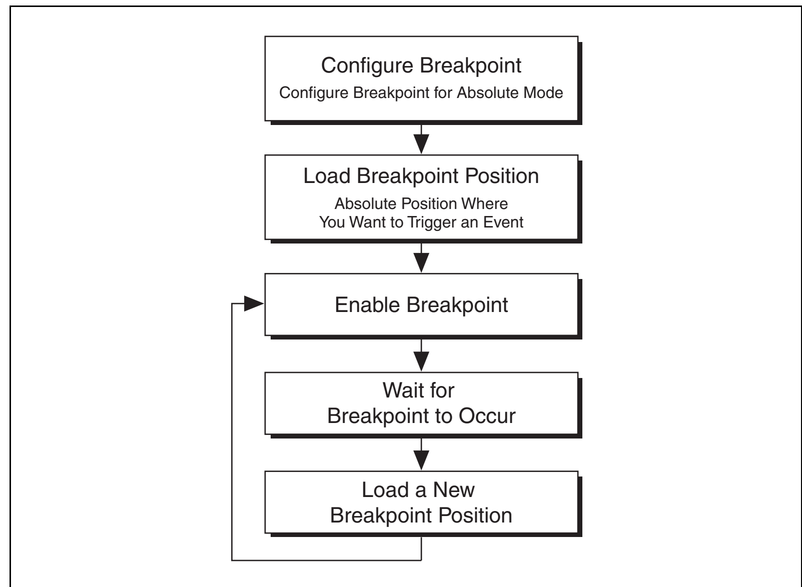


Figure 13-3. Single Position Breakpoint Algorithm

LabVIEW Code

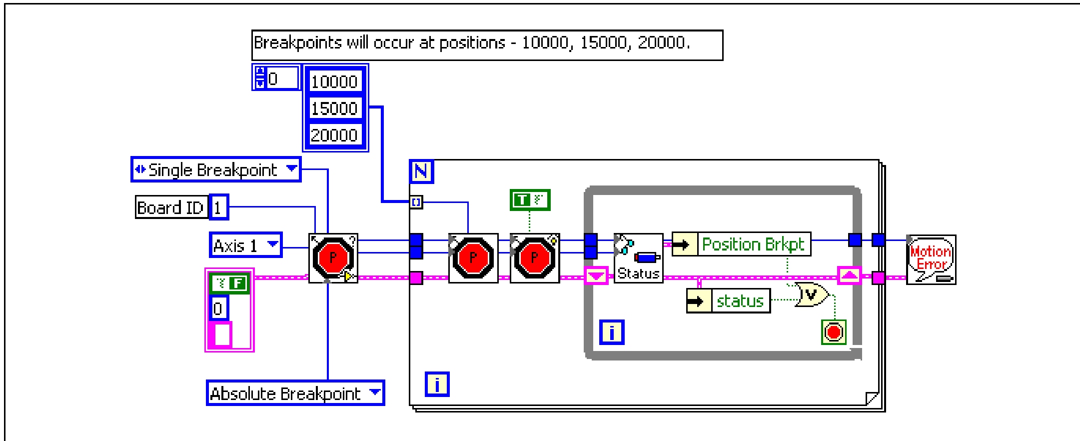


Figure 13-4. Single Position Breakpoint in LabVIEW

NI-Motion VIs for Figure 13-4, in order from left to right:

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

Refer to Figure 13-5 for an example of how to route this breakpoint via RTSI.

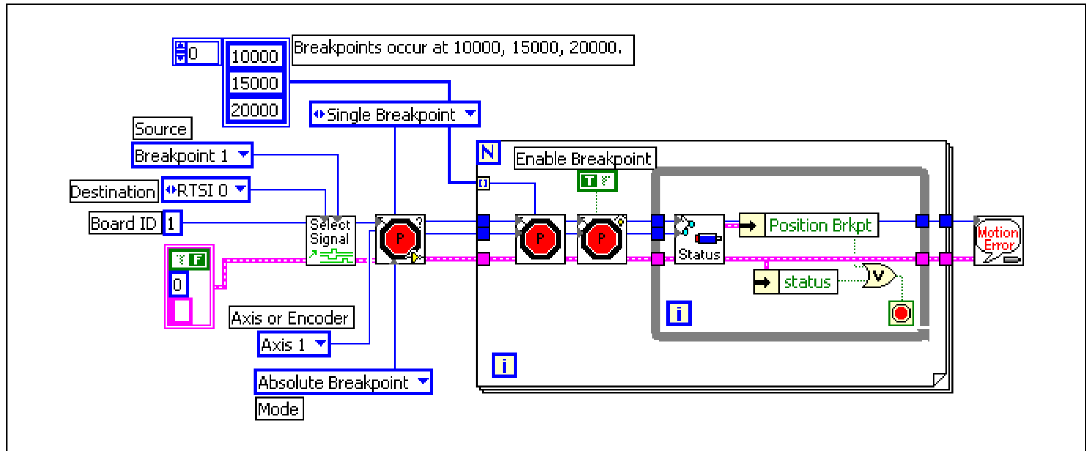


Figure 13-5. Single Position Breakpoint With RTSI Using LabVIEW

NI-Motion VIs for Figure 13-5, in order from left to right:

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

Once 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 more complete and can compile.

```
// 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 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
}
```

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, suppose you are creating a motion 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 you do not care what the current position is at any given time. You only need the axis to move a

specific number of counts from wherever it is and then generate a breakpoint, wherever that may be.

Note All breakpoints can be affected by jitter in the motion system. For example, if you have a very small breakpoint window, the jitter in the motion 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.

Algorithm

Figure 13-6 shows the basic algorithm for relative breakpoints.

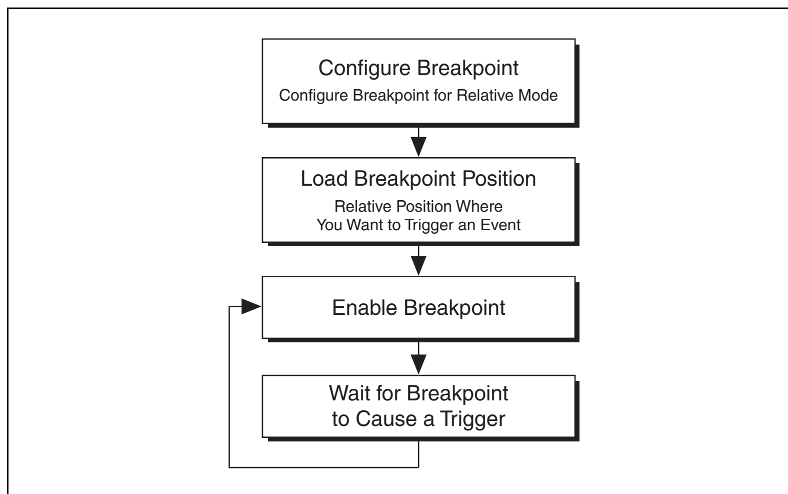


Figure 13-6. Relative Position Breakpoints Algorithm

Notice that relative break points 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 (735x 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. You would need to modify the examples to incorporate them into a motion application.

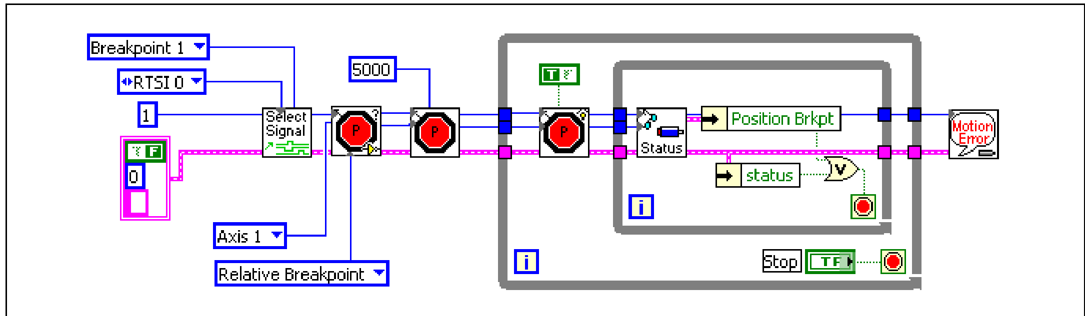


Figure 13-7. Relative Position Breakpoint with RTSI Using LabVIEW

NI-Motion VIs for Figure 13-7, in order from left to right:

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

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 more complete and can compile.

```
// 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 - 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 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
}
```

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 of creating periodically occurring breakpoints using NI-Motion functions, depending on which motion controller you have. For the 735*x* controller, use periodic breakpoints. For 734*x* and 733*x* controllers, use modulo breakpoints.

Note All breakpoints can be affected by jitter in the motion system. For example, if you have a very small breakpoint window, the jitter in the motion 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 (735*x* only)

Periodic breakpoints require that you specify an initial breakpoint and an ongoing repeat period. Once 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 13-8.

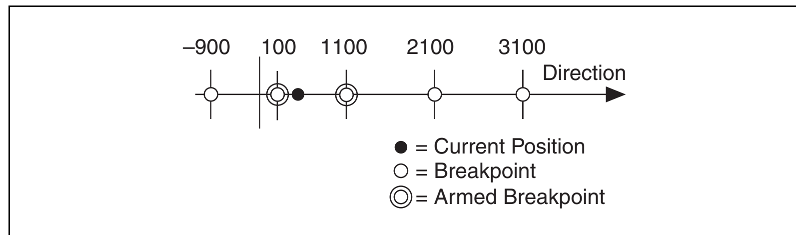


Figure 13-8. Periodic Breakpoint Every 1,000 Counts/Steps

Algorithm

Figure 13-9 shows the basic algorithm for periodic breakpoints.

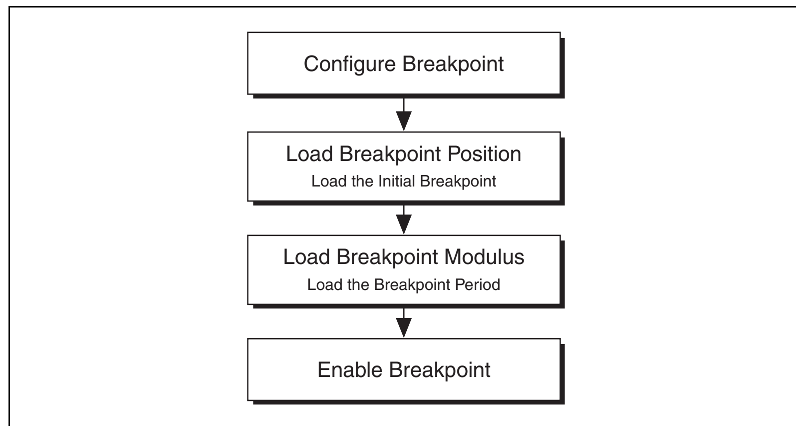


Figure 13-9. Periodic Breakpoint Algorithm

LabVIEW Code

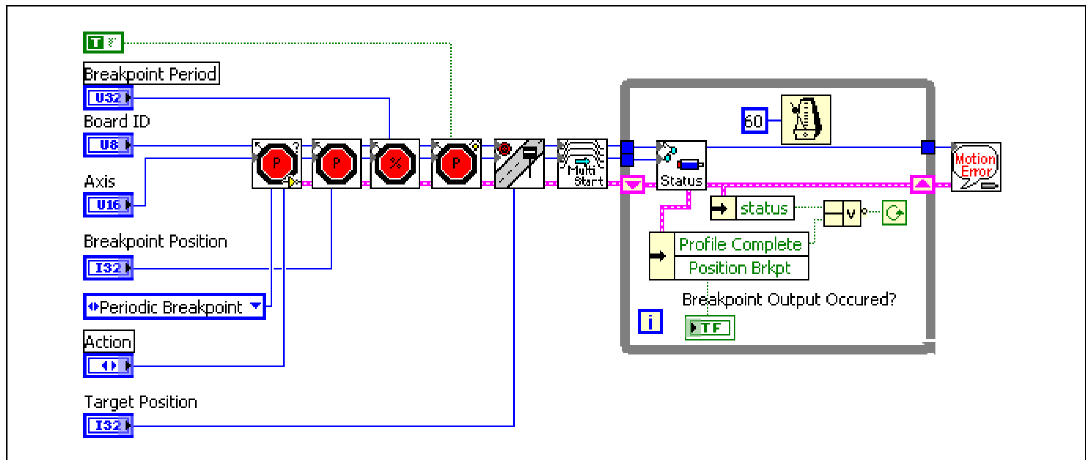


Figure 13-10. Relative Position Breakpoint with RTSI Using LabVIEW

NI-Motion VIs for Figure 13-7, in order from left to right:

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

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 more complete and can compile.

```
// 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 an axis number: ");
scanf("%u",&axis);

//Get the Target Position
printf("Enter a target position: ");
scanf("%ld",&targetPos);

//Get the Break Point Position
printf("Enter a break point position: ");
scanf("%ld",&bpPos);

//Get the Break Point Period
printf("Enter a break point period: ");
scanf("%ld",&bpPer);

//Configure the break point to be absolute
err = flex_configure_breakpoint(boardID, axis,
                              NIMC_PERIODIC_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,&axisStatus);
    CheckError;

    err = flex_read_pos_rtn(boardID,axis,&currentPos);
    CheckError;

    //Check the breakpoint bit
    bpStatus = !((axisStatus & NIMC_POS_BREAKPOINT_BIT)==0);

    //Check the profile complete bit
    profileStatus = !((axisStatus & NIMC_PROFILE_COMPLETE_BIT) ==
                      0);
    printf("Current Position=%10d Breakpoint Status=%d Profile
            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_SPACE1,
                        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 ID, 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
}
```

Modulo Breakpoints (734*x* and 733*x* only)

Modulo breakpoints use a breakpoint "window" to define 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 13-11.

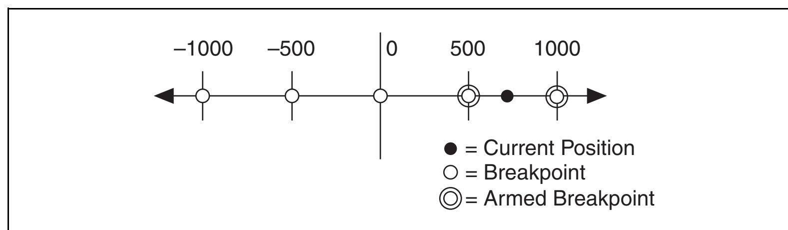


Figure 13-11. Breakpoint Modulus of 500

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 13-12.

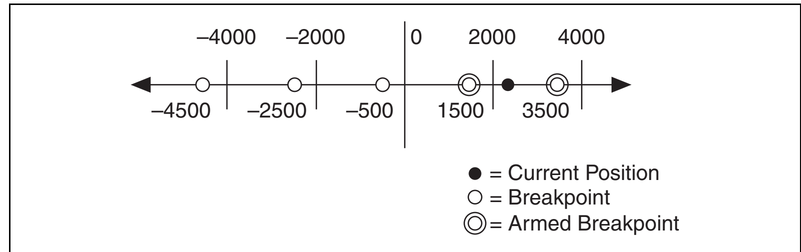


Figure 13-12. Breakpoint Modulus of 2,000 with an Offset of 500

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

Algorithm

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

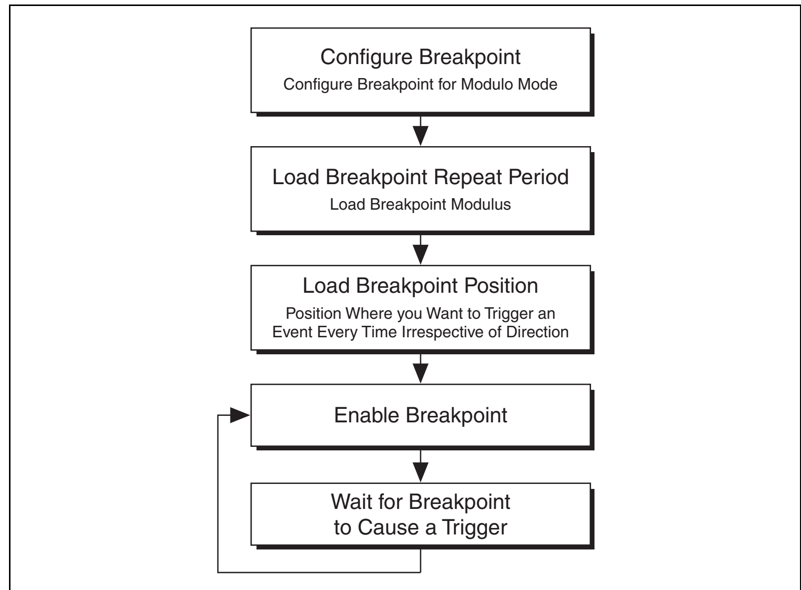


Figure 13-13. Modulo Breakpoints Algorithm

LabVIEW Code

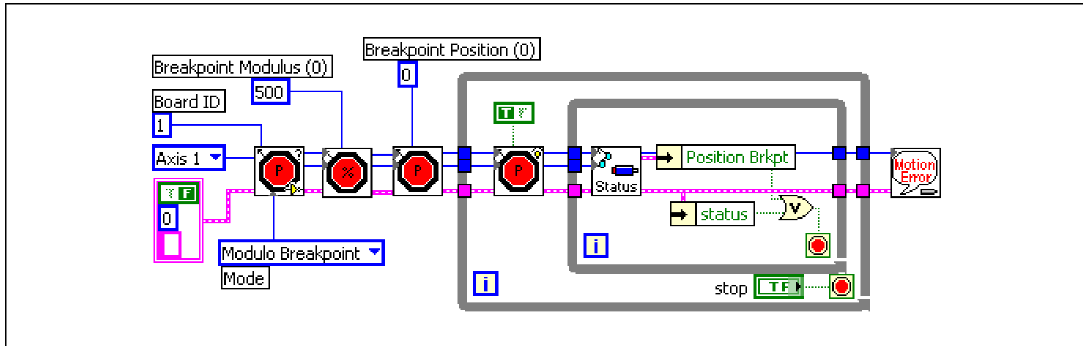


Figure 13-14. Modulo Breakpoint Using LabVIEW

NI-Motion VIs for Figure 13-14, in order from left to right:

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

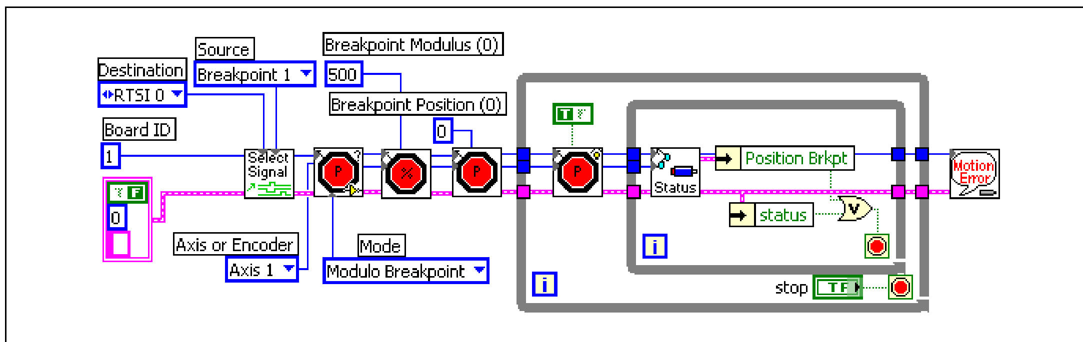


Figure 13-15. Modulo Breakpoint With RTSI Using LabVIEW

NI-Motion VIs for Figure 13-15, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

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 (735*x* 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, enabling better frequency than the non-buffered high-speed capture method.

Algorithm

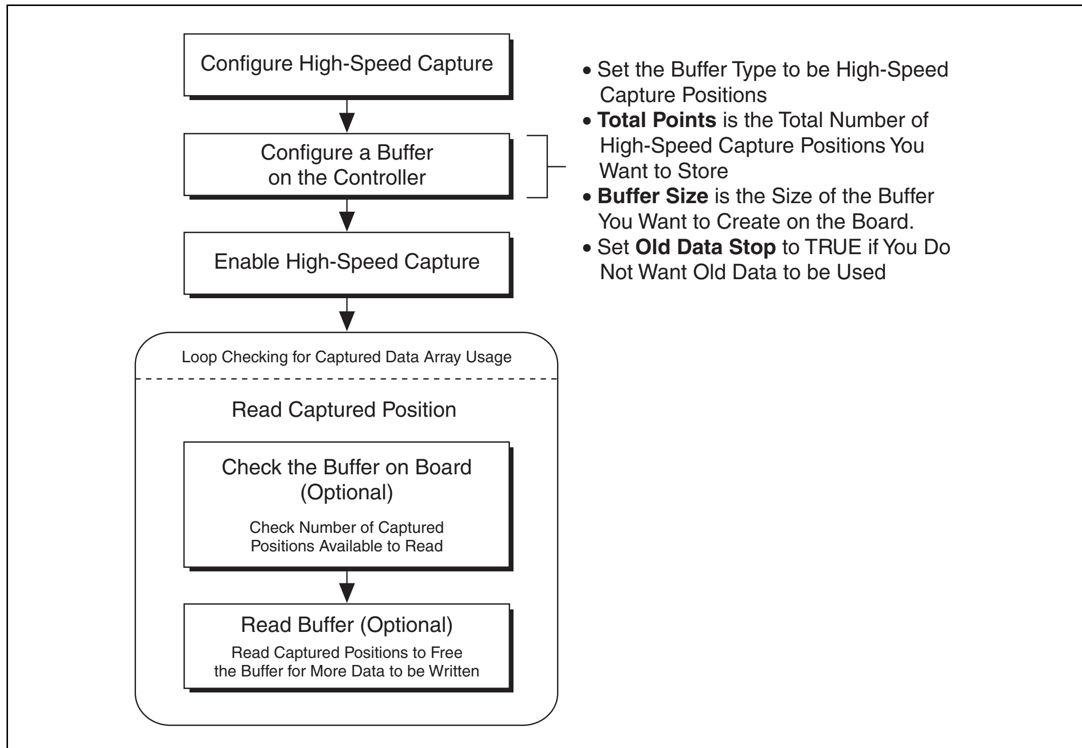


Figure 13-16. Buffered High-Speed Capture Algorithm

LabVIEW Code

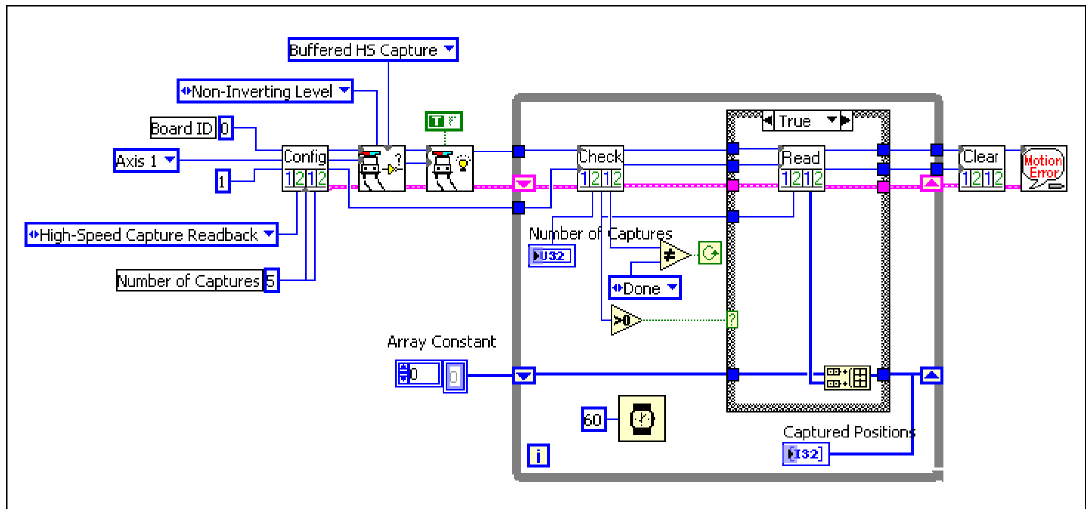


Figure 13-17. Buffered High-Speed Capture in LabVIEW

NI-Motion VIs for Figure 13-17, in order from left to right:

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

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 more complete and can compile.

```
// 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 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[currentDataPoint] = readBuffer[i];
        printf("capture pos %d\n",
               capturedPositions[currentDataPoint]);
        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 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 board
            err = flex_read_error_msg_rtn(boardID, &commandID,
                                          &resourceID, &errorCode);

            nimcDisplayError(errorCode,commandID,resourceID);

            //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 after it triggers.

Algorithm

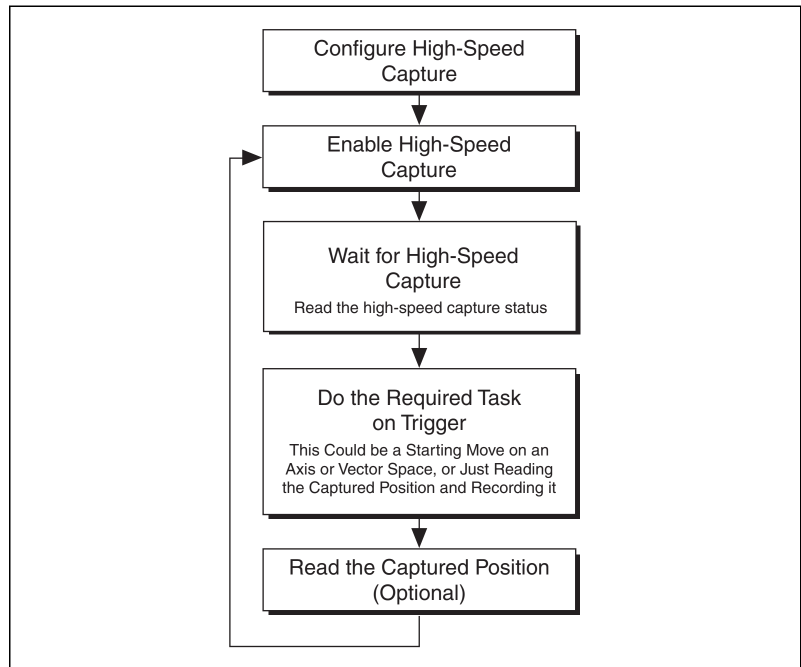


Figure 13-18. High-Speed Capture Algorithm

LabVIEW Code

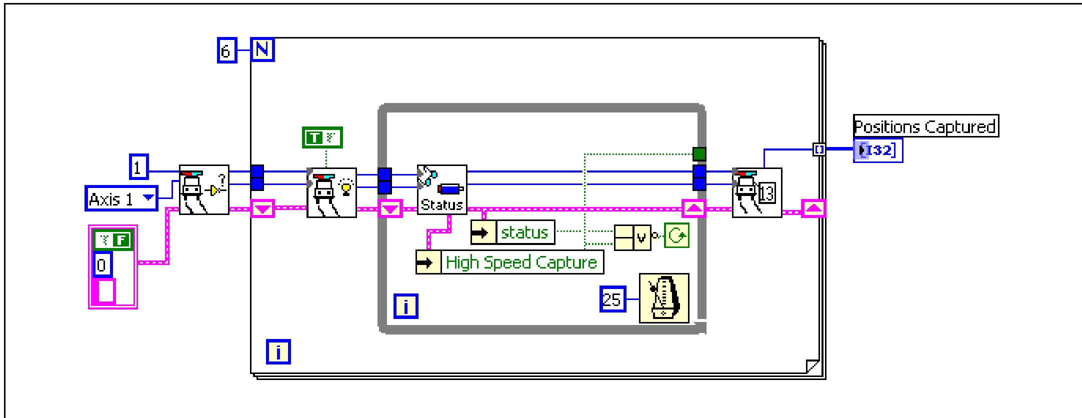


Figure 13-19. High-Speed Capture Using LabVIEW

NI-Motion VIs for Figure 13-19, in order from left to right:

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

To trigger the high-speed capture from a RTSI line, connect the destination in Select Signal to be the High Speed Capture Line, as shown in Figure 13-20.

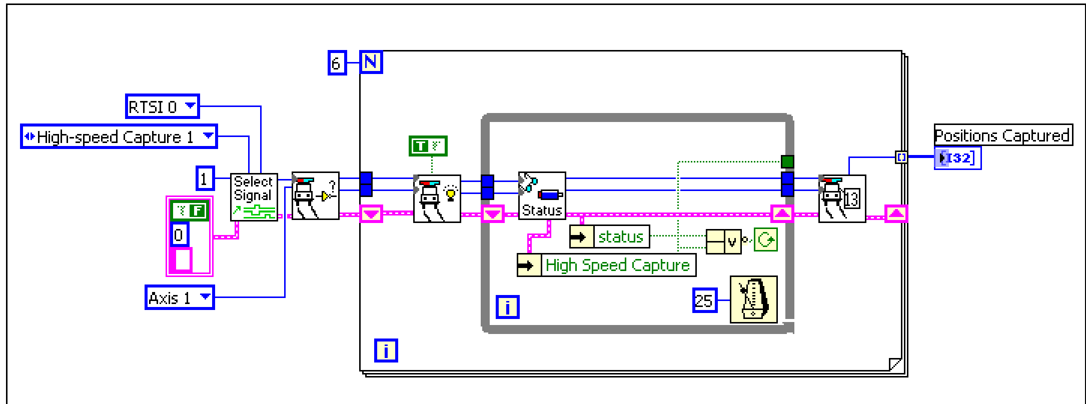


Figure 13-20. High-Speed Capture with RTSI Using LabVIEW

NI-Motion VIs for Figure 13-20, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

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 acquisition (DAQ) devices and image acquisition (IMAQ) 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 data acquisition, image acquisition, digital I/O, or other NI motion boards 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 through 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 via the RTSI bus to trigger other devices. You also can configure DAQ and IMAQ devices to trigger the NI motion controllers via the RTSI bus using the high-speed capture (trigger input) capability.

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 DAQ or IMAQ.

As shown in Figure 13-21, the I/O reaction task automatically re-enables the breakpoints or high-speed captures on the NI-735*x* motion controller. On NI-734*x* motion controllers, you must write an onboard program or use the host to perform the same re-enabling tasks.

Figure 13-21. 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 13-22.

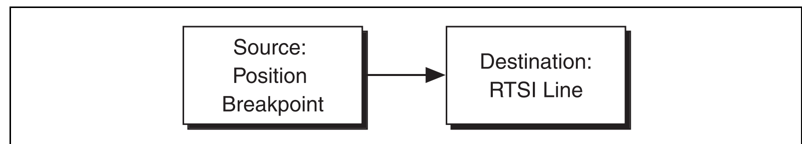


Figure 13-22. Position Breakpoint Using RTSI

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 13-23.

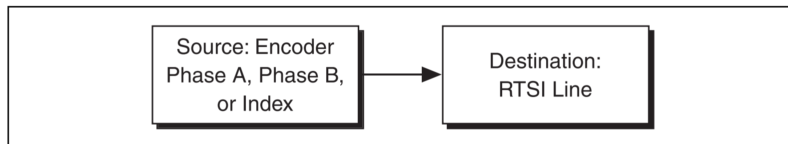


Figure 13-23. Encoder Pulses Using RTSI

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 13-24.

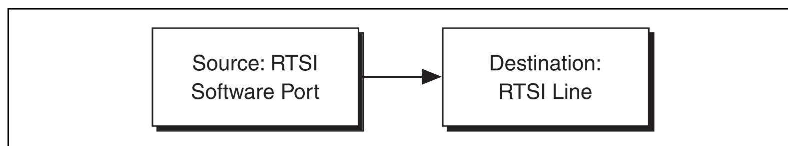


Figure 13-24. Software Trigger Using RTSI

High-Speed Capture Input Using RTSI

When the RTSI line receives the trigger from DAQ or IMAQ, the corresponding high-speed capture occurs, as shown in Figure 13-25.

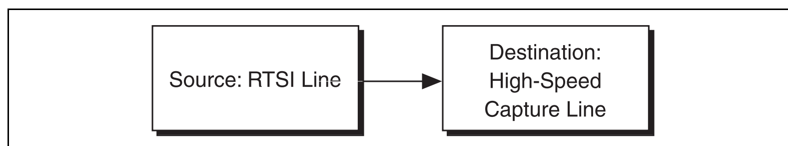


Figure 13-25. High-Speed Capture Input Using RTSI

14

Torque Control

To maintain constant torque or force, the sensor that returns the feedback to the 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 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 monitoring force.

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.

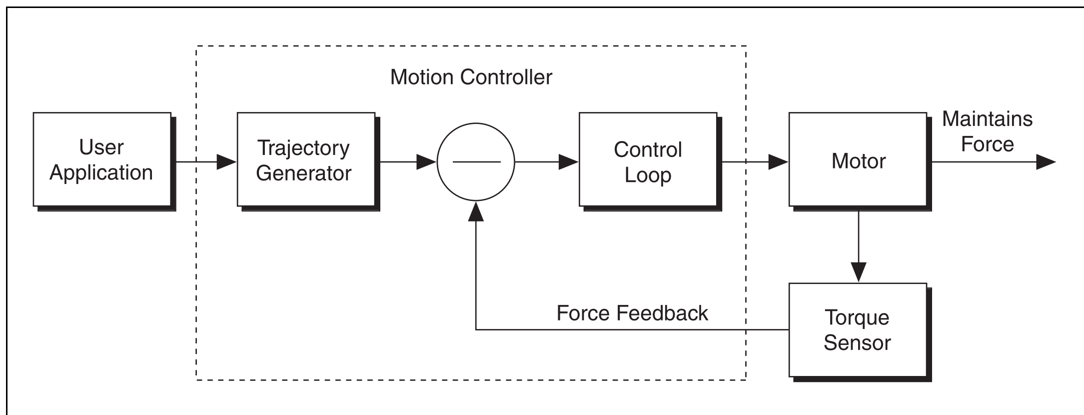


Figure 14-1. Torque Control Using Analog Feedback Flowchart

Tuning the control loop with a force sensor (analog feedback) should be the same 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 to 5 V, –5 to +5 V, 0 to 10 V and –10 to +10 V. As long as you are dealing in counts for entering the values of position, velocity, acceleration, and deceleration, you do not have to worry entering the counts/revolution value for the axis.

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

Algorithm

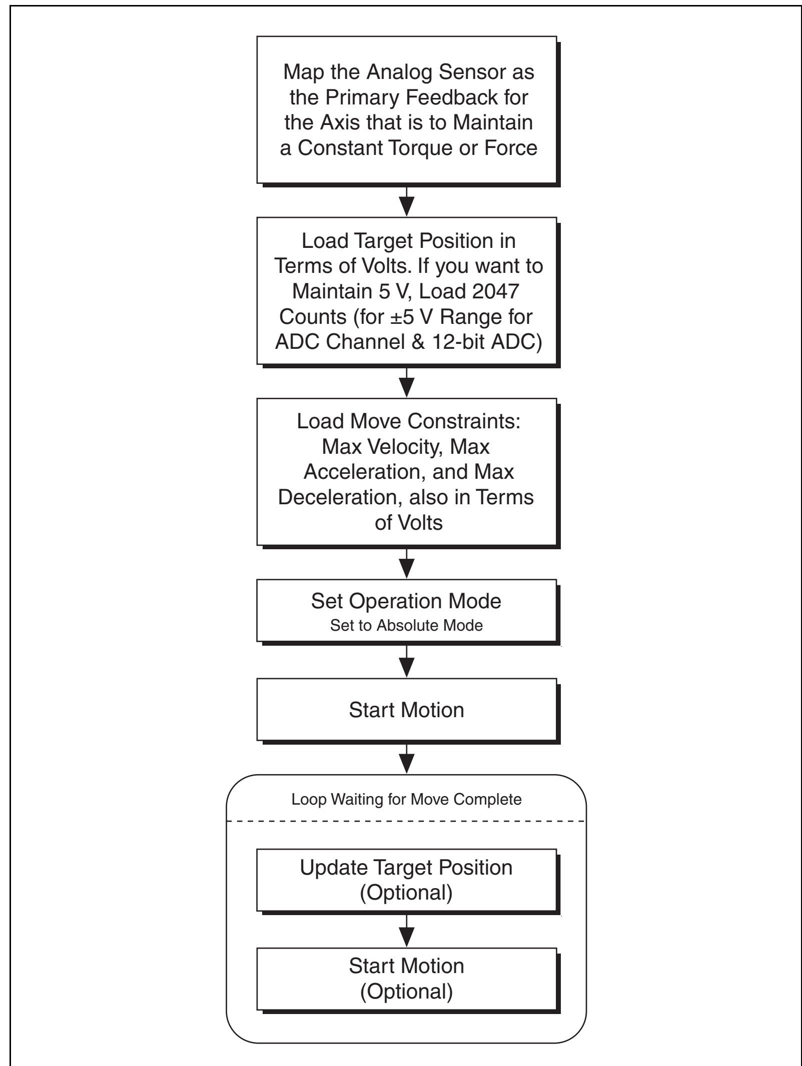


Figure 14-2. Torque Control Using Analog Feedback Algorithm

LabVIEW Code

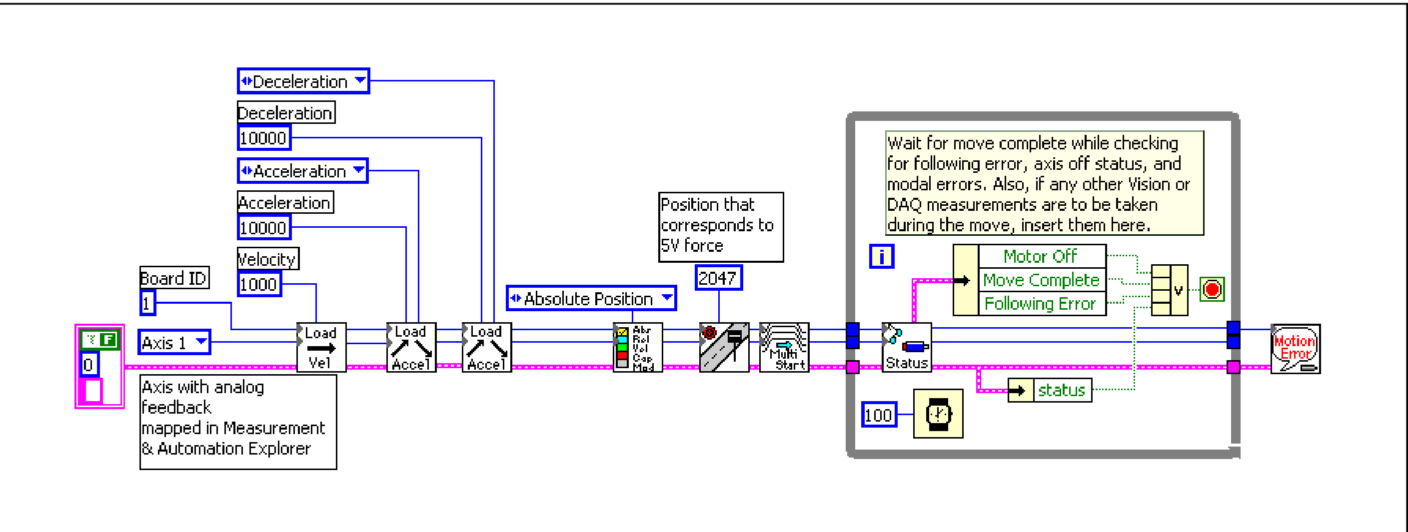


Figure 14-3. Torque Control Using Analog Feedback Using LabVIEW

NI-Motion VIs for Figure 14-3, in order from left to right:

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

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 more complete and can compile.

```
// 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 - 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 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 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
}
```

Monitoring Force

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

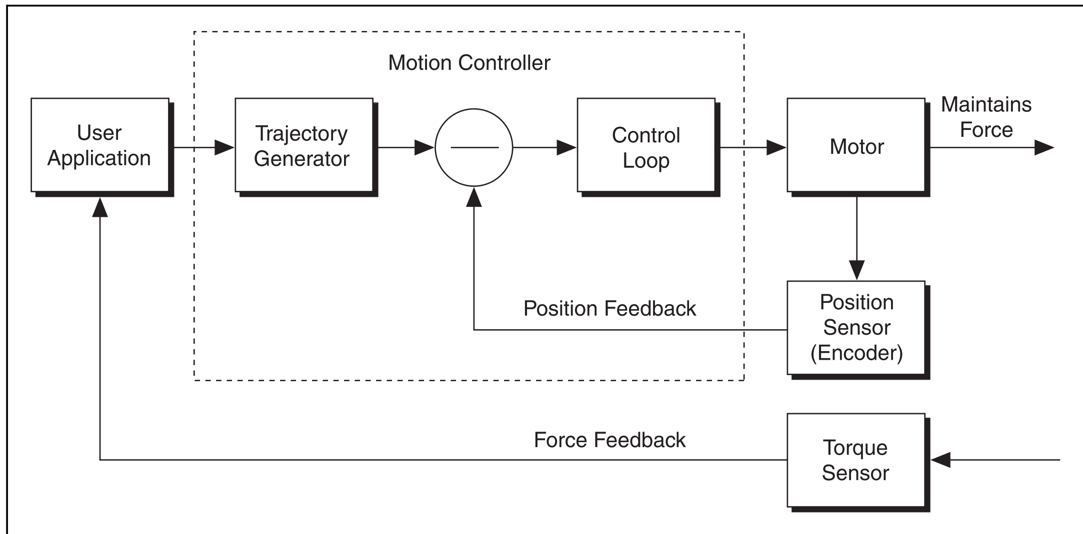


Figure 14-4. Torque Control Using Analog Feedback Flowchart

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

Algorithm

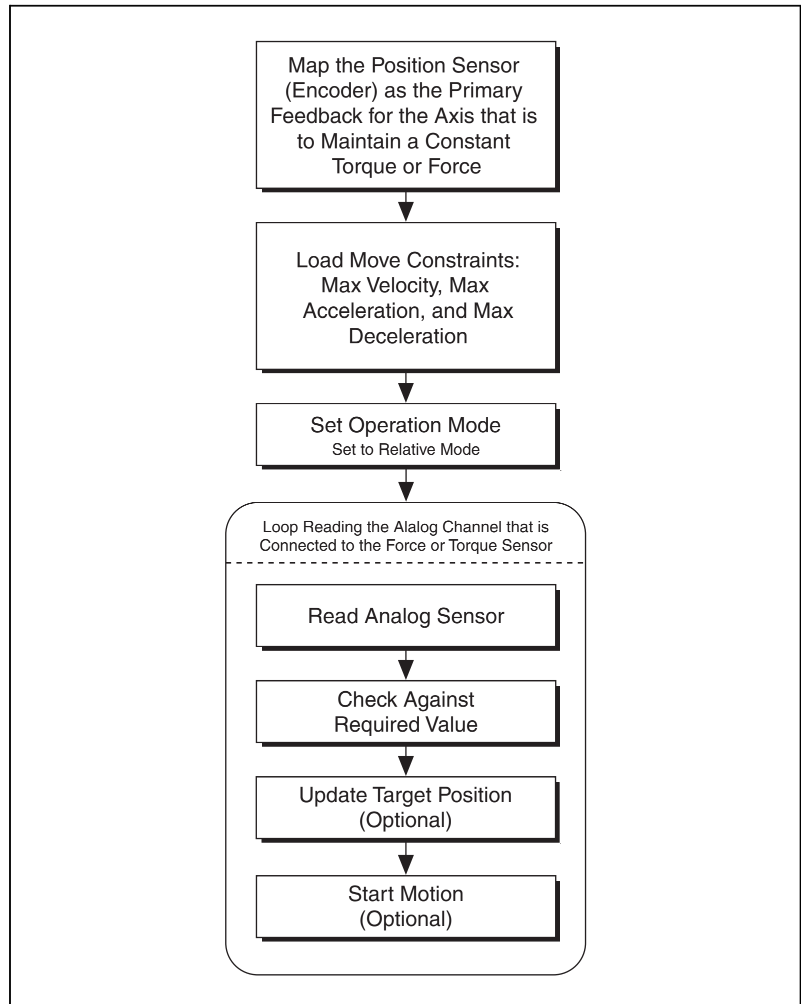


Figure 14-5. Torque Control Using Monitoring Force Algorithm

LabVIEW Code

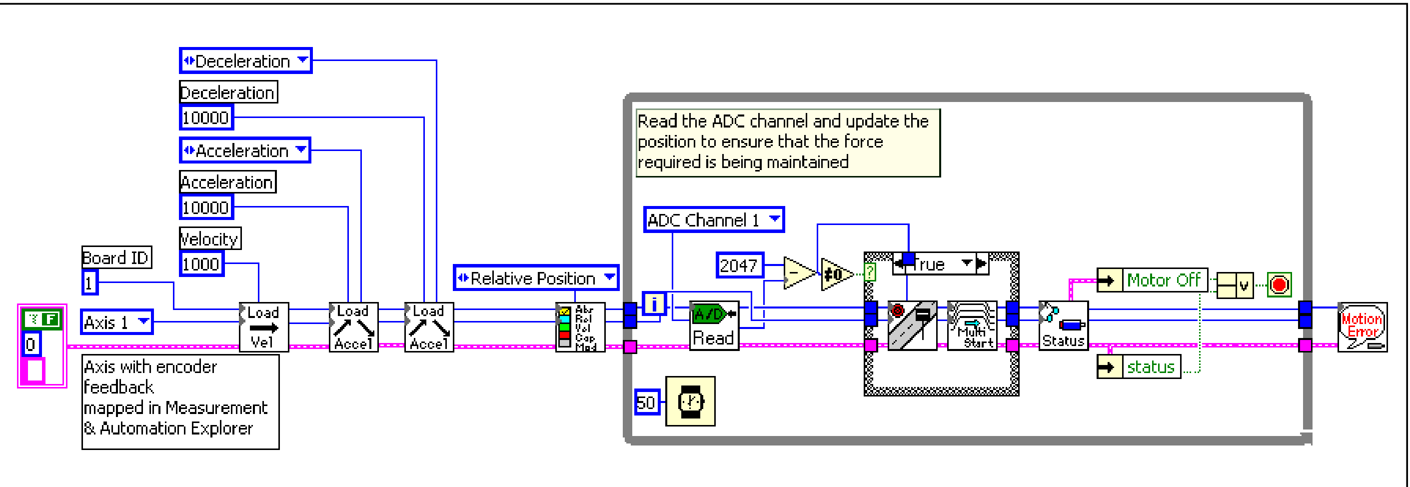


Figure 14-6. Torque Control Using Monitoring Force Using LabVIEW

NI-Motion VIs for Figure 14-6, in order from left to right:

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

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 more complete and can compile.

```
// 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
    // corrsponds 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 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
}
```

Speed Control Based on Analog Value

Consider 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 then 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.

Algorithm

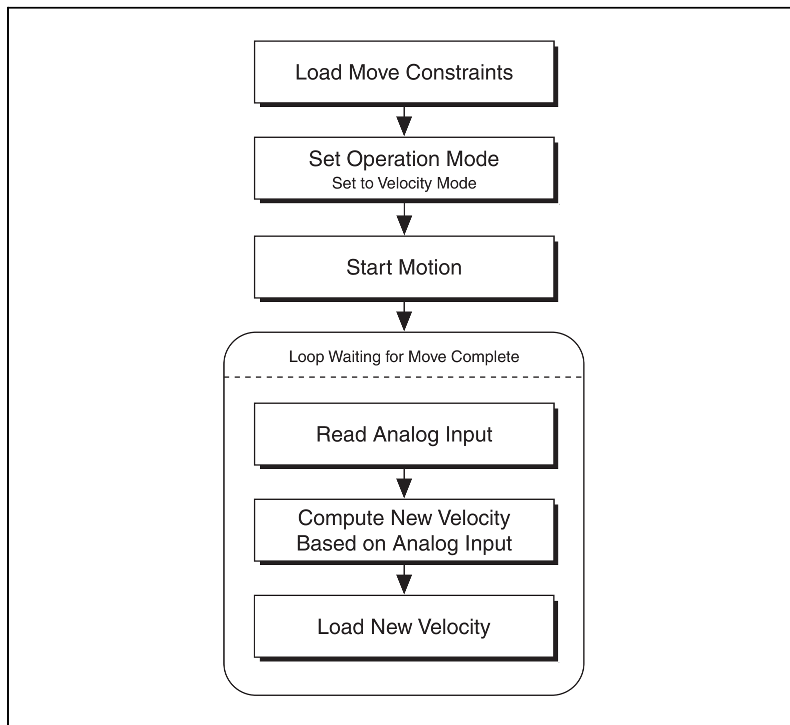


Figure 14-7. Speed Control Based on Analog Feedback Algorithm

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

LabVIEW Code

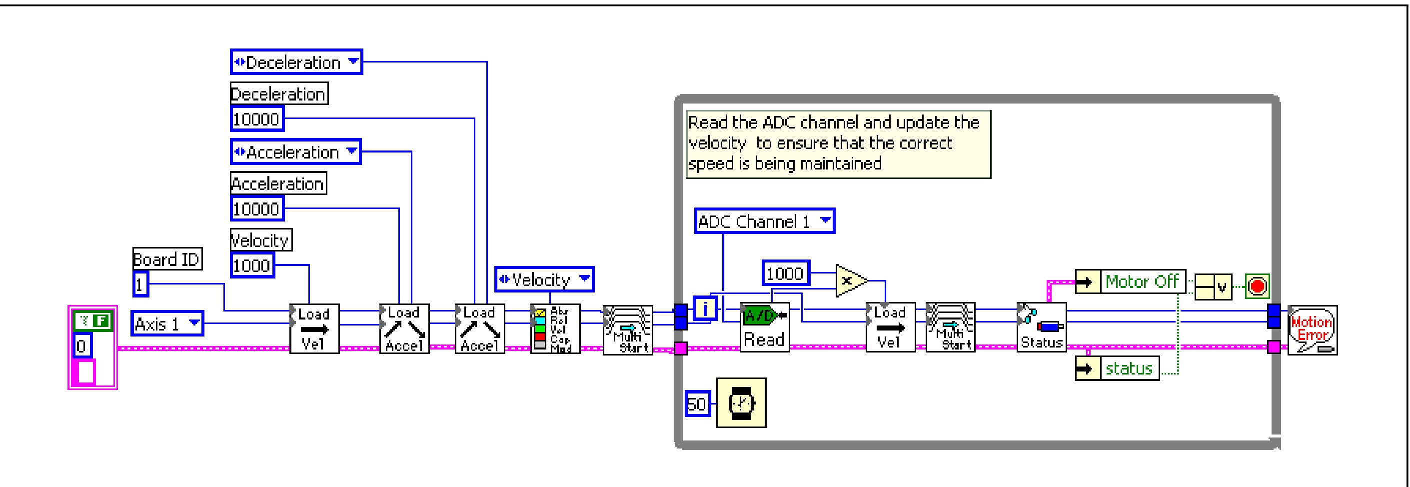


Figure 14-8. Speed Control Based on Analog Feedback Using LabVIEW

NI-Motion VIs for Figure 14-8, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

15 Onboard Programs

You can use the real-time operating system on the NI motion controller to run custom programs. This allows you to offload certain motion-specific tasks from the host processor and onto the motion controller. Using onboard variables, which are global data on the board, simple 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.

Even though running an onboard program seems very deterministic, onboard programs have the least priority in a preemptive multitasking environment running on the embedded microprocessor. This is 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 15-1. The onboard programs and housekeeping tasks are time-sliced among themselves.

For greater control and determinism for the motion system, National Instruments offers the LabVIEW RT motion control system, consisting of a PXI chassis, PXI motion controller or controllers, LabVIEW RT, and NI-Motion driver software.

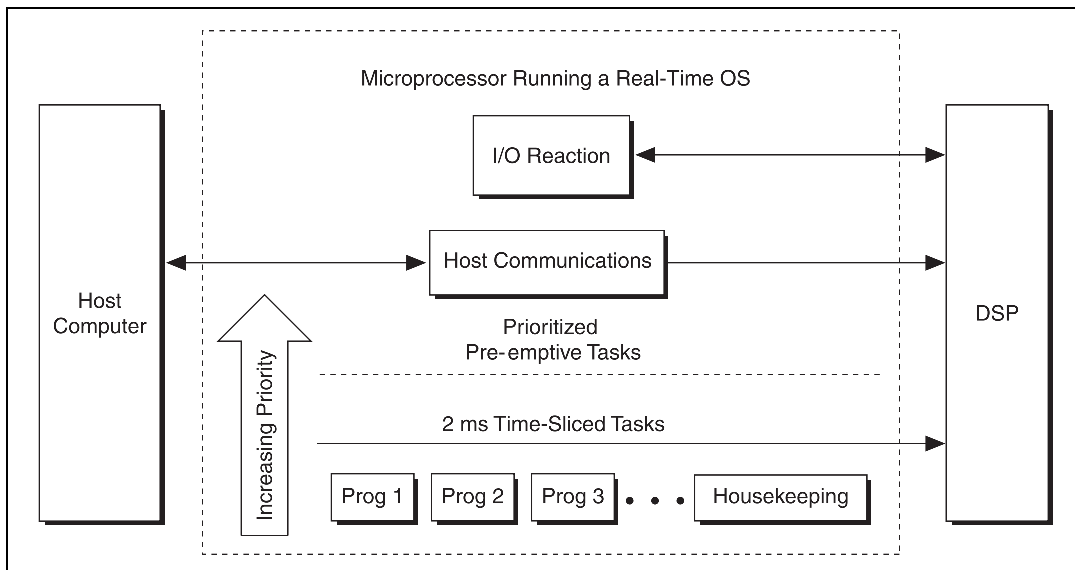
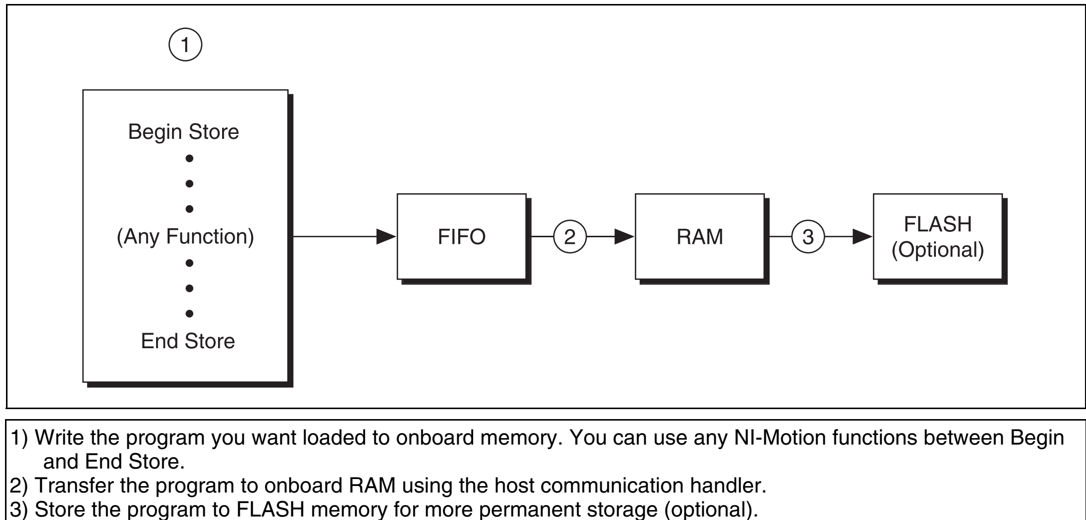


Figure 15-1. Onboard Program Priority

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 Your Loops* section of Part III, *Programming NI-Motion*, for more information about programming delays in the loops.

Writing Onboard Programs

Almost all NI-Motion functions that execute on the host can run onboard. You can store up to 32 onboard programs on the controller. These onboard programs remain on the controller until it is reset. If you want the onboard programs to persist through a reset of the controller, save them to FLASH, as shown in Figure 15-2.



1) Write the program you want loaded to 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 15-2. Writing Onboard Programs

Algorithm

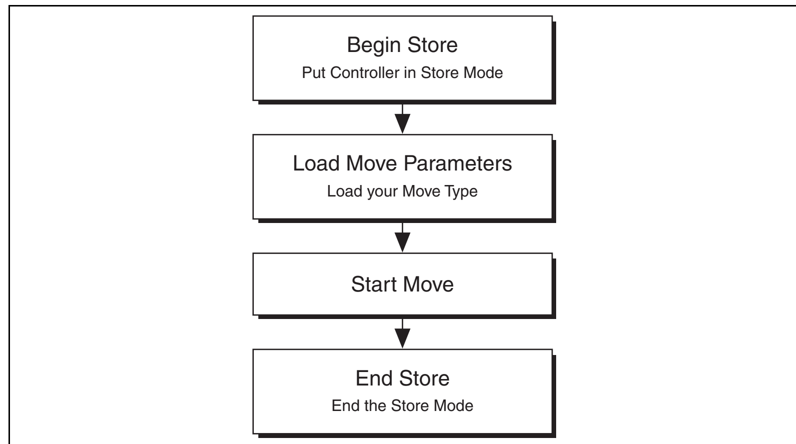


Figure 15-3. Basic Onboard Program Algorithm

LabVIEW Code

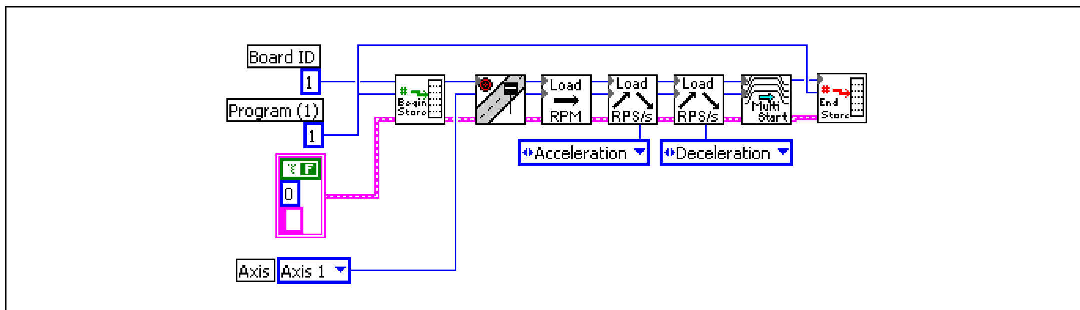


Figure 15-4. Simple Onboard Program in LabVIEW

NI-Motion VIs for Figure 15-4, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

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, it is necessary to 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 controller to automatically run an onboard program upon powering up the motion system. You also can call an

onboard program from another onboard program using the Run Program function.

Recursively calling an onboard program generates an error.

Stopping an Onboard Program

Stop Program ends the execution of previously run onboard programs.

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

It is important to notice that stopping an onboard program is different from stopping motion. 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 axis or axes from moving.

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

Like 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 pauses automatically also 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 in the onboard programs that affect its execution. 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 wakes up the onboard program to continue executing the next function in its sequence, as shown in Figure 15-5.

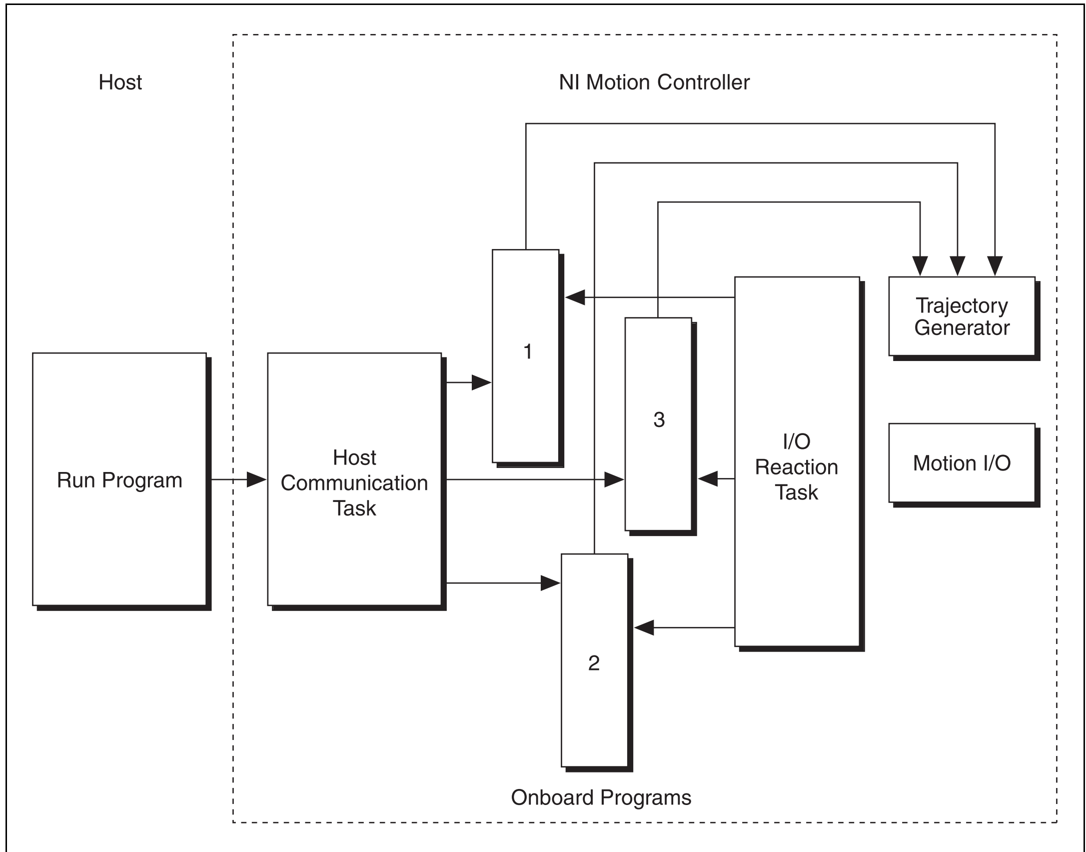


Figure 15-5. Executing Onboard Programs

Algorithm

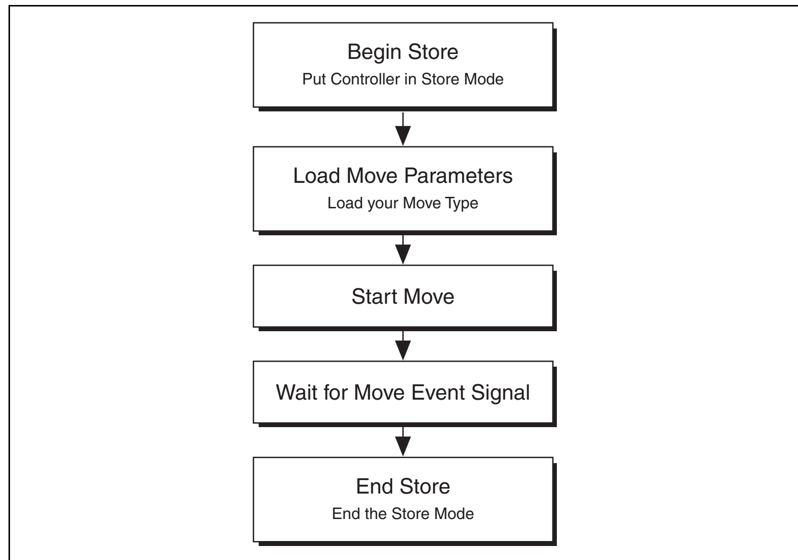


Figure 15-6. Onboard Program Conditional Execution Algorithm

LabVIEW Code

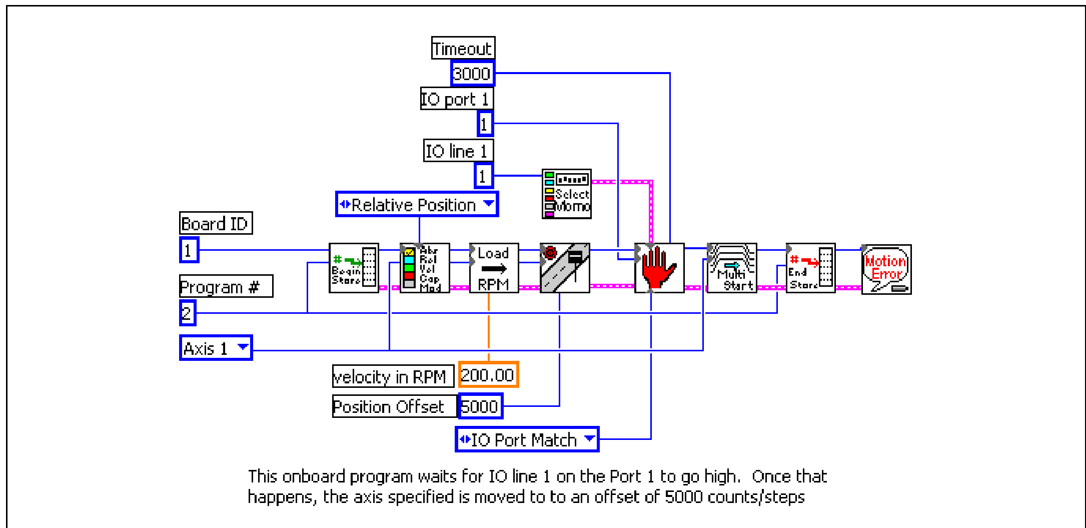


Figure 15-7. Onboard Program Waiting for an I/O Line to Go Active

NI-Motion VIs for Figure 15-7, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

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 being loaded by functions that are executing within an onboard program. You also can synchronize execution or data between the host computer and the motion controller. This memory is statically allocated.

For example, you may want to update the velocity of an axis based on the analog voltage read from an ADC channel.

Algorithm

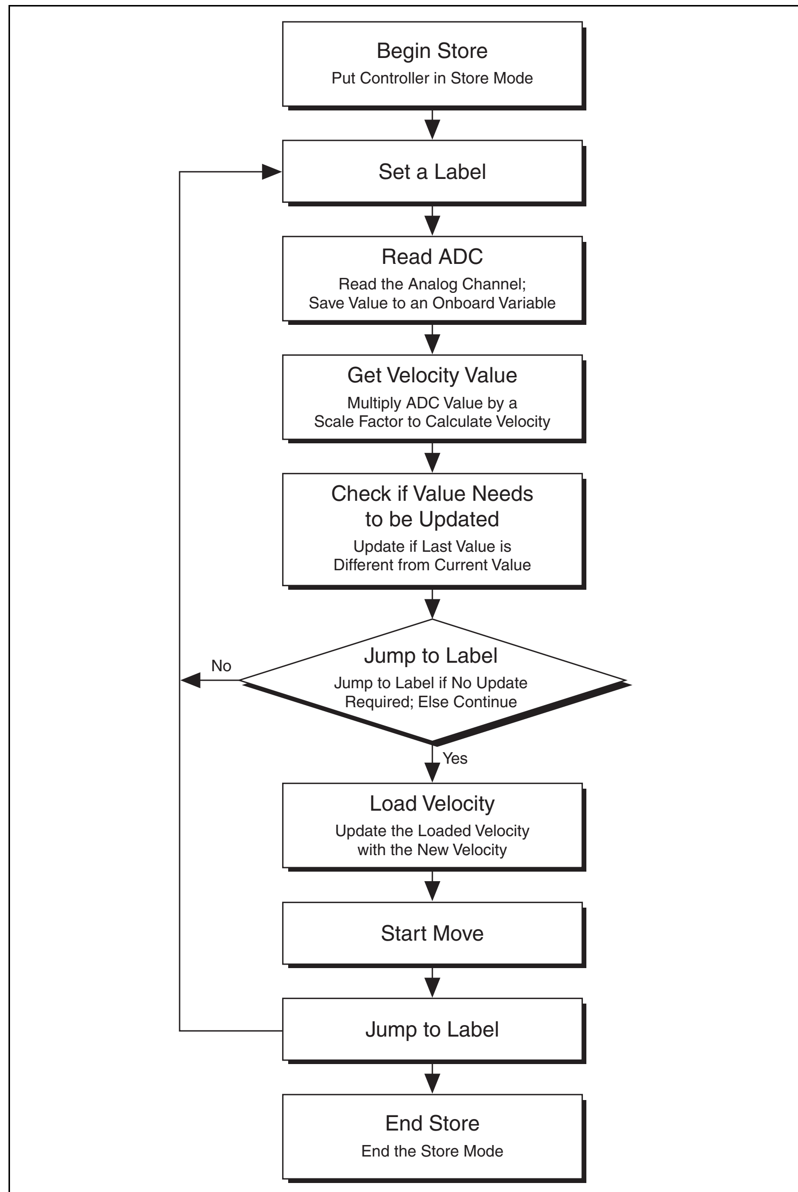


Figure 15-8. Updating Velocity Based on ADC Channel Algorithm

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

LabVIEW Code

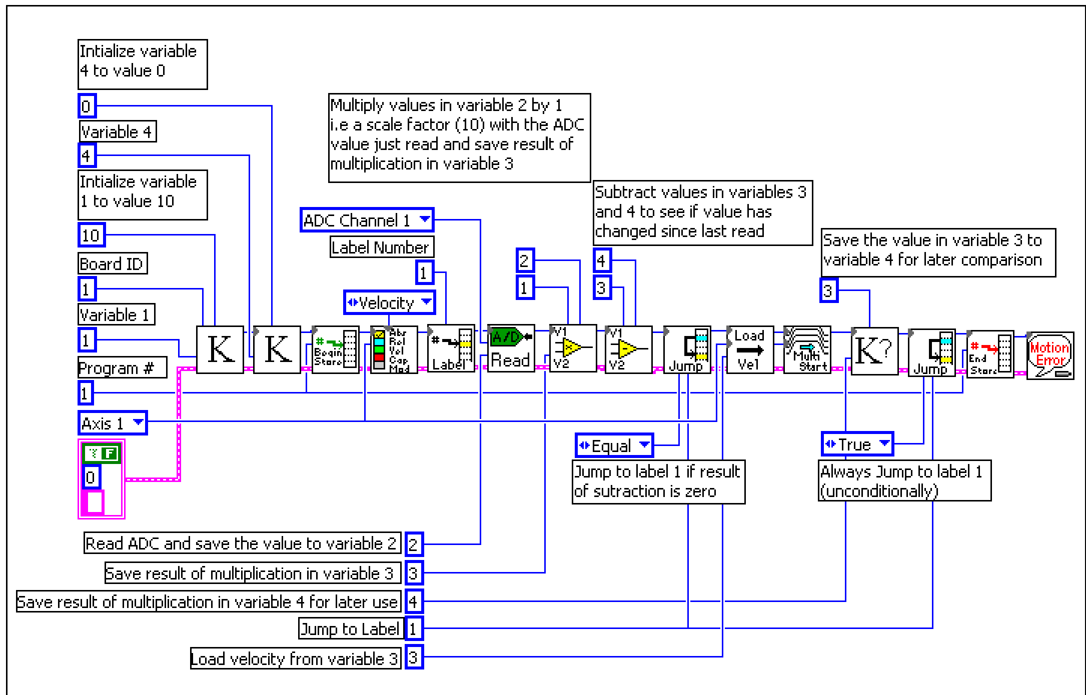


Figure 15-9. Updating Velocity Based on ADC Channel in LabVIEW

NI-Motion VIs for Figure 15-4, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

Branching Onboard Programs

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

Algorithm

Figure 15-10 shows an onboard program waiting for an I/O line to go active before starting a move.

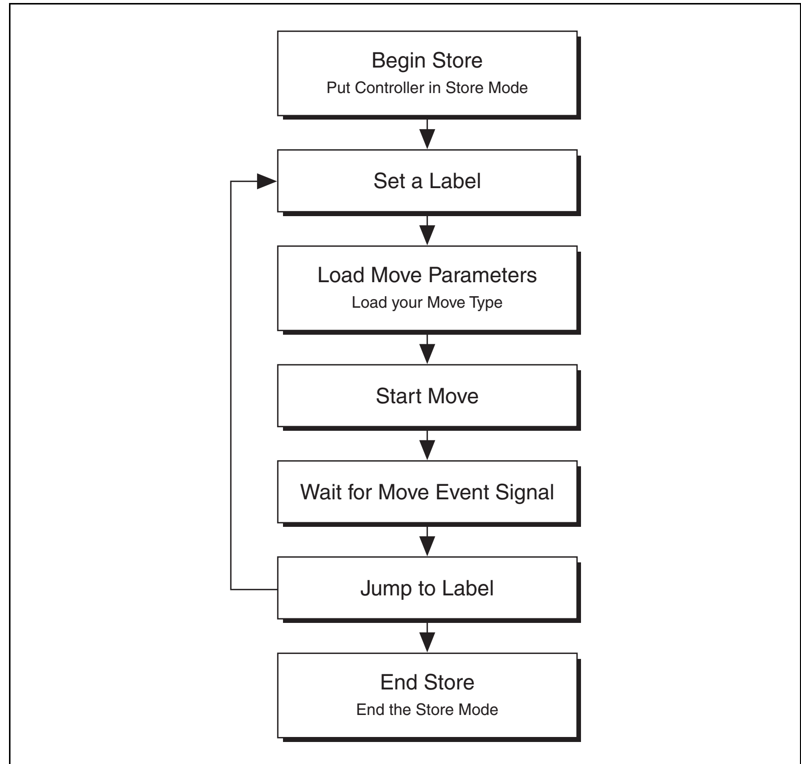


Figure 15-10. Using Labels with Onboard Programs

LabVIEW Code

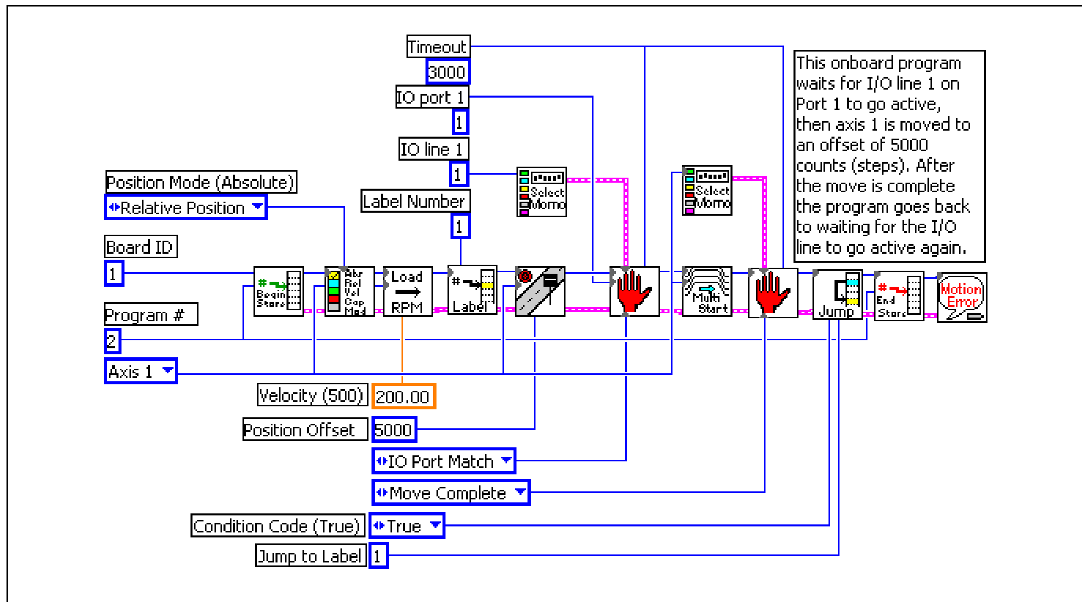


Figure 15-11. Continuously Executing Onboard Program in LabVIEW

NI-Motion VIs for Figure 15-4, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

Math Operations

Math operations are always performed on values stored in onboard variables. There is a level of indirection here that requires care. All math operations work on onboard variables and set a global condition that the Jump to Label function uses to decide whether or not to jump 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 example above, 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, they use the value loaded in variables 1 to 0x78 and interpret them as variables from which to load the value. This creates two levels of indirection.

Making the return vector of the Read Position function point to 0x81 causes the position variable to end up in the variable number contained in onboard variable 1, as shown in Figure 15-12.

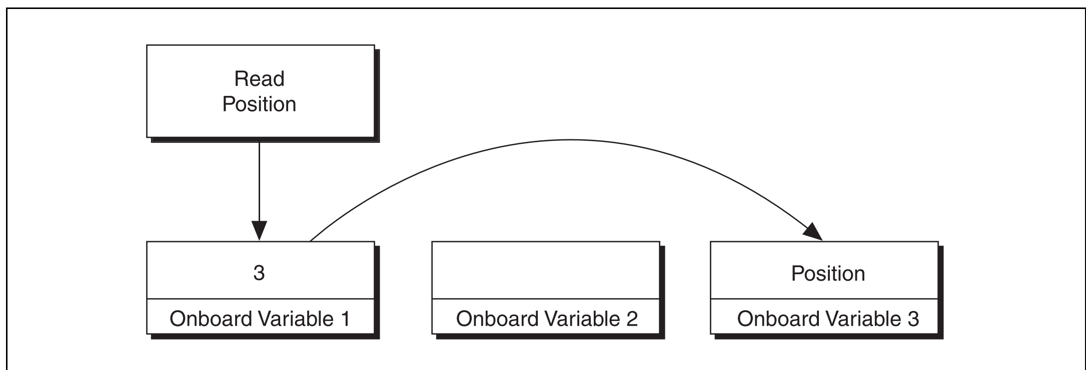


Figure 15-12. Reading an Indirect Variable

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

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 15-13.

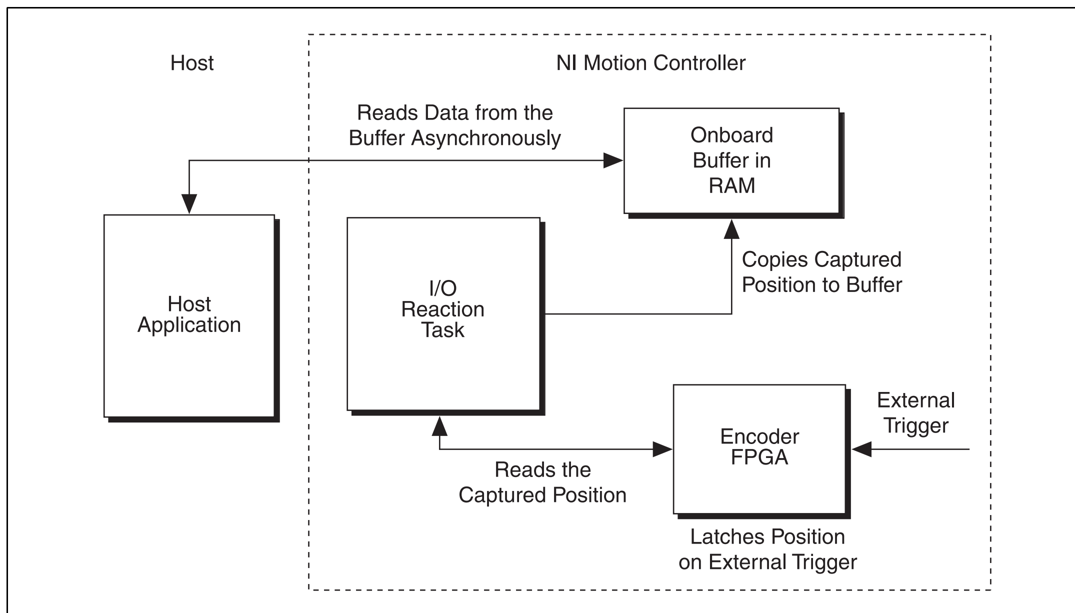


Figure 15-13. Onboard Buffer Data Flow

Buffers are created from a dynamic pool of memory, so you should free the memory whenever 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 15-14 shows the algorithm for using onboard buffers to store data.

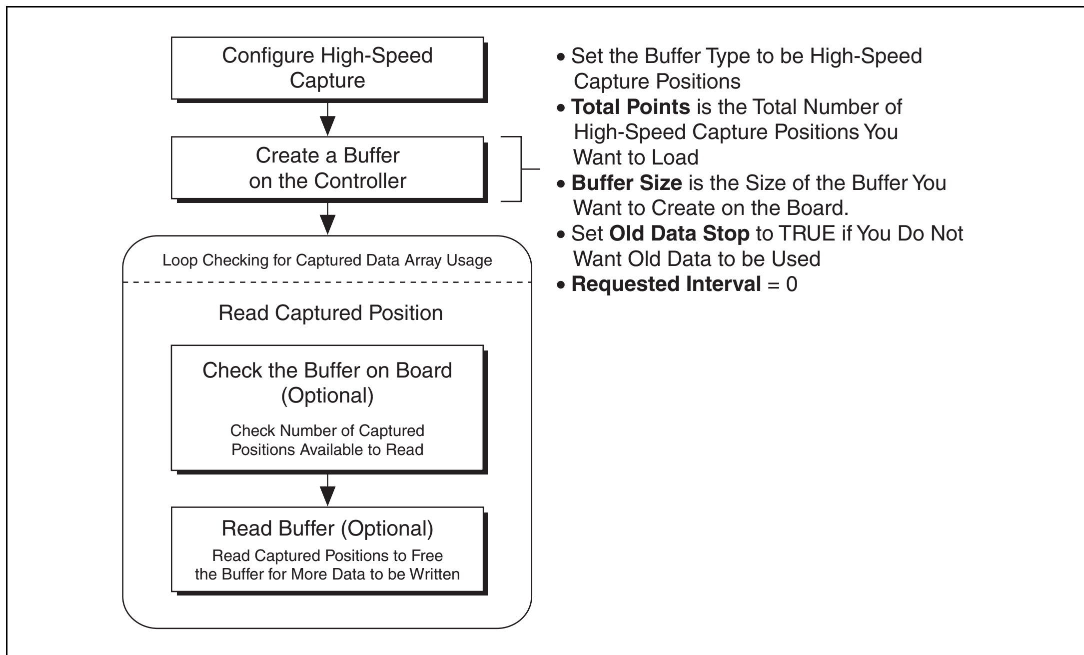


Figure 15-14. Onboard Buffer Algorithm

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 15-15.

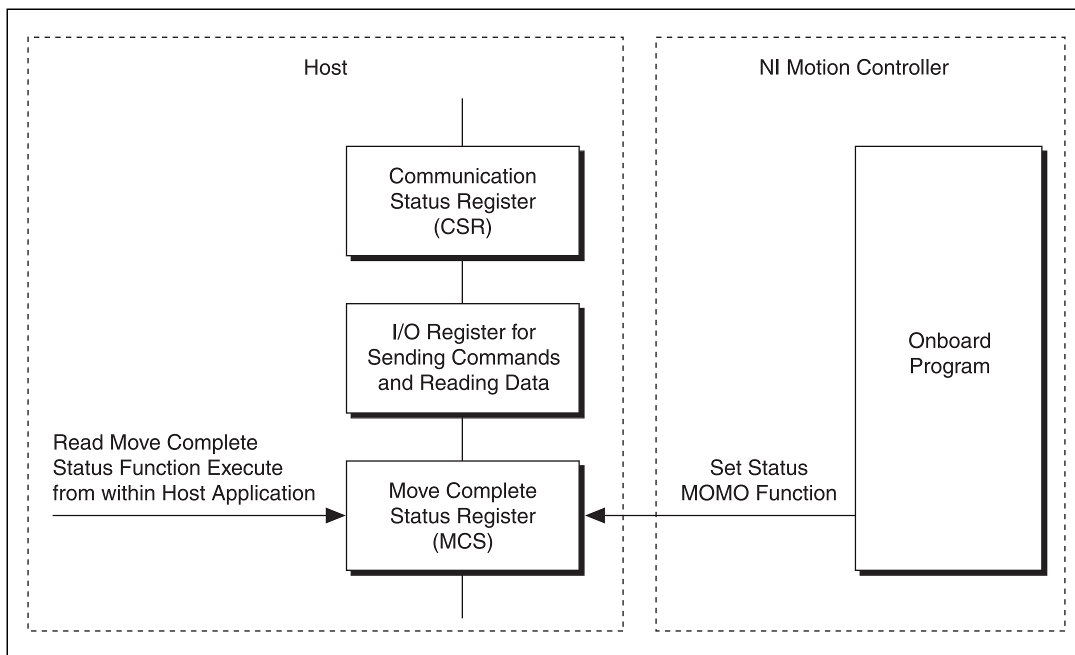


Figure 15-15. Synchronizing Host Applications with Onboard Programs

For example, consider a host application that reads an onboard variable that has been updated by an onboard program. Use the algorithm in Figure 15-16 to synchronize the host application with the onboard program.

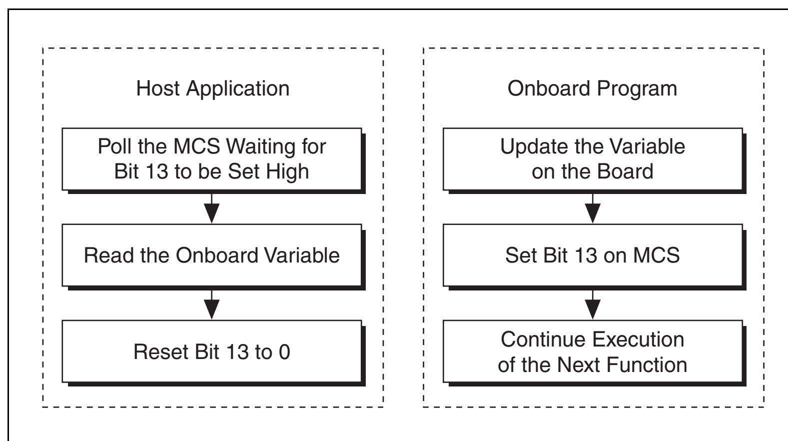


Figure 15-16. Synchronization Algorithm

LabVIEW Code

This example causes axis 1 to move 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 15-17 shows the code that runs as an onboard program.

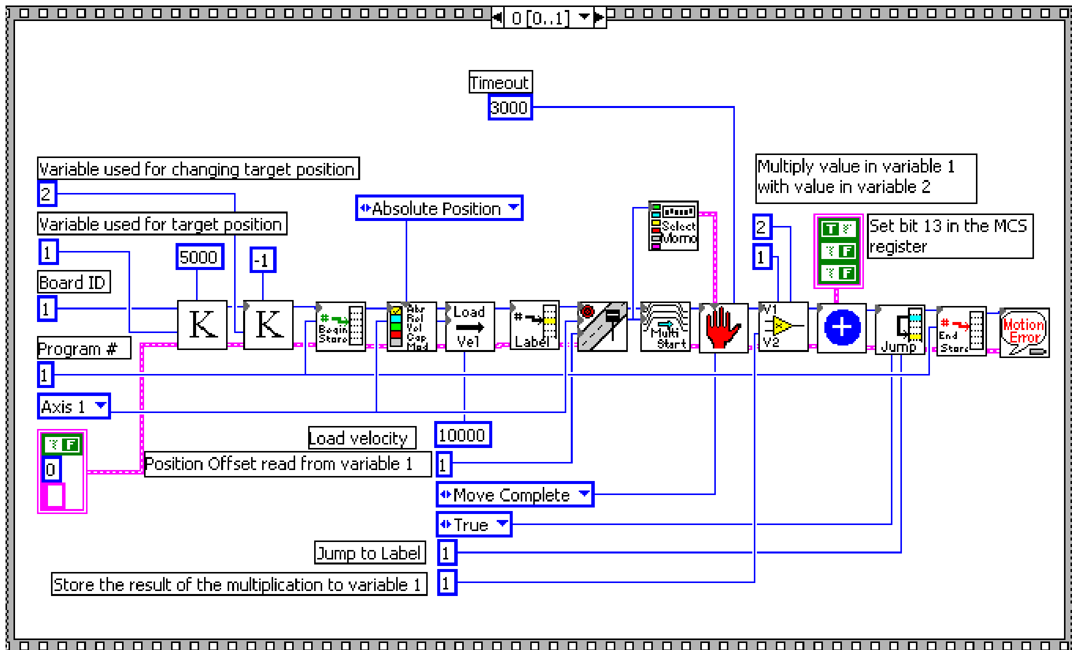


Figure 15-17. Synchronization Onboard Code in LabVIEW

NI-Motion VIs for Figure 15-17, in order from left to right:

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

Figure 15-18 shows the code that runs on the host.

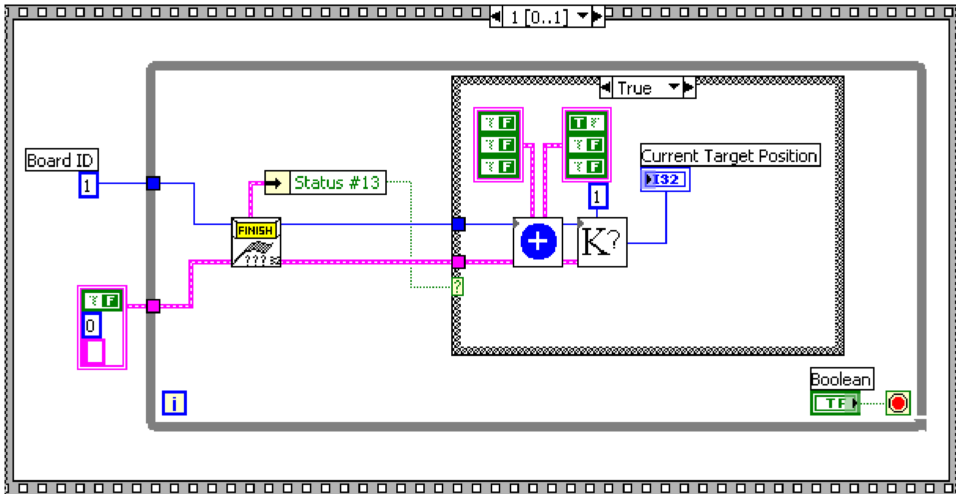


Figure 15-18. Synchronization Host Code in LabVIEW

NI-Motion VIs for Figure 15-18, in order from left to right:

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

Note As the host is polling a register on the controller, it is not invoking the Host Communication Task on the real-time operating system on the controller. Therefore, the onboard programs executing are not preempted, but instead run reliably and 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 more complete and can compile.

```
// 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 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
}
```

Onboard Subroutines

You can create subroutines to run as different onboard programs and execute them from within an onboard program. This way you can cause different onboard programs to execute from within the main onboard program.

Algorithm

Figure 15-19 shows an onboard program algorithm that checks the I/O line state to determine which onboard subroutine to execute.

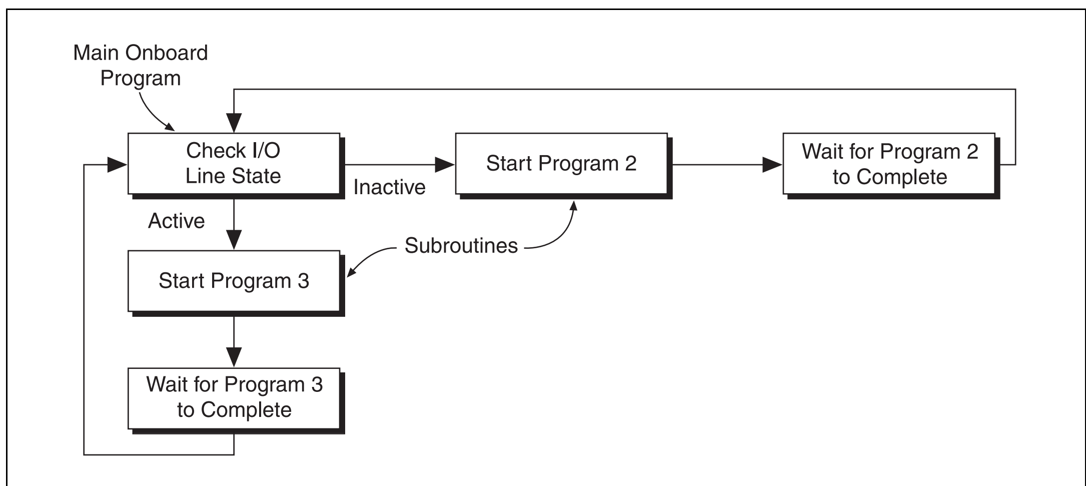


Figure 15-19. Onboard Subroutine Algorithm

If the I/O line is active, the main onboard program calls an onboard subroutine that causes the motor to rotate clockwise. If the I/O line is inactive, the main onboard program calls an onboard subroutine that causes the motor to rotate counter-clockwise.

LabVIEW Code

Figure 15-20 shows the main onboard program used to determine the subroutine call.

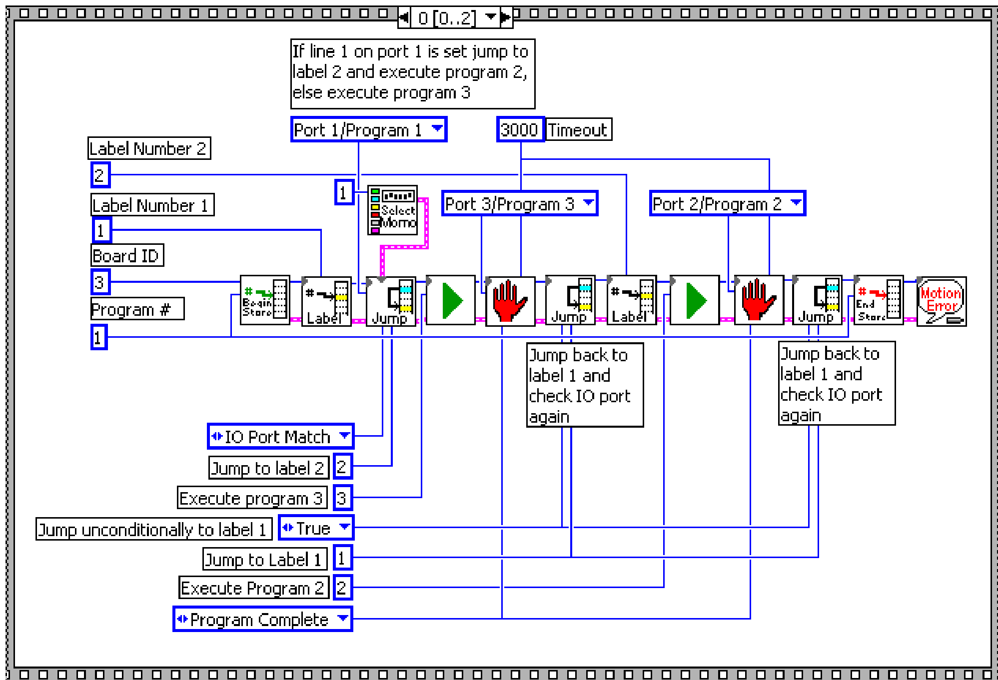


Figure 15-20. Onboard Subroutine Call Using LabVIEW

NI-Motion VIs for Figure 15-20, in order from left to right:

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

Figure 15-21 shows the subroutine that causes the motor to rotate clockwise.

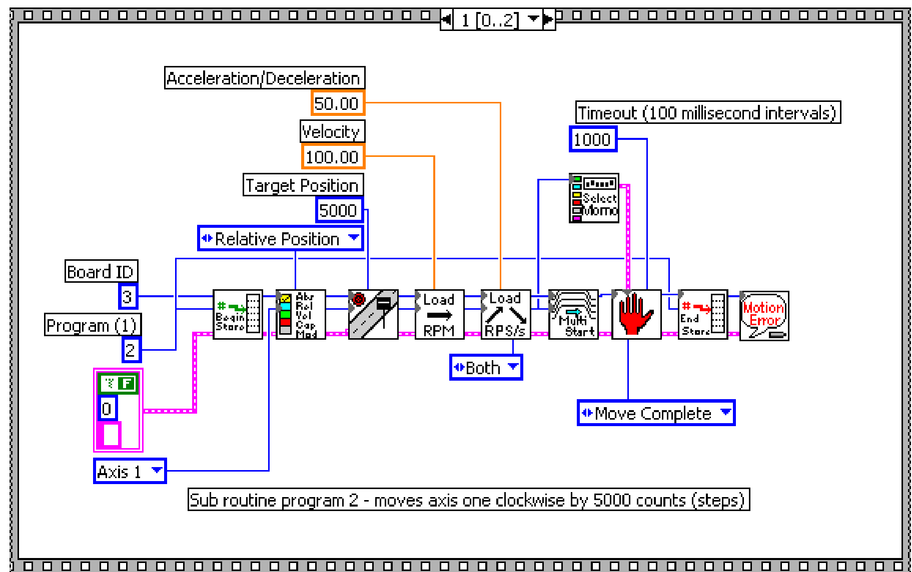


Figure 15-21. Clockwise Subroutine Using LabVIEW

NI-Motion VIs for Figure 15-21, in order from left to right:

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

Figure 15-22 shows the subroutine that causes the motor to rotate counter-clockwise.

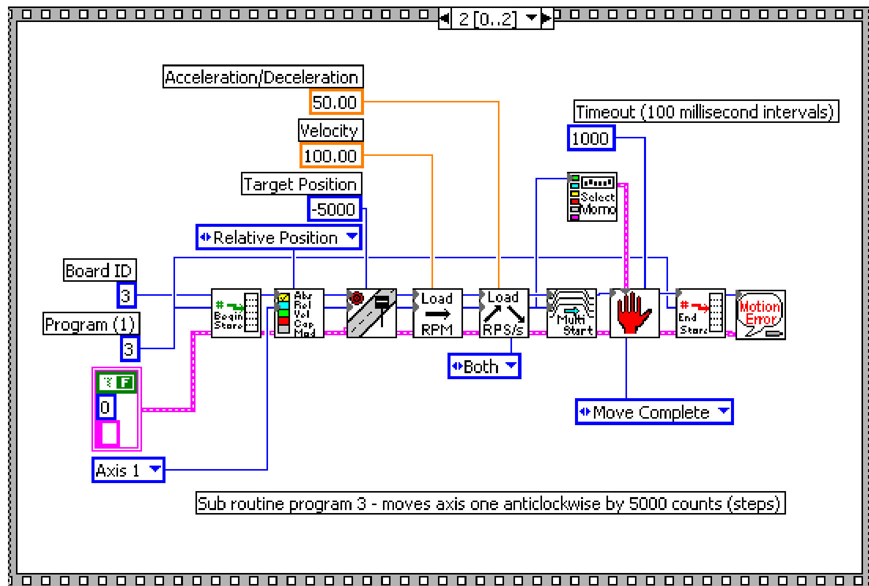


Figure 15-22. Counter-Clockwise Subroutine Using LabVIEW

NI-Motion VIs for Figure 15-22, in order from left to right:

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

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 more complete and can compile.

```
// 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 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
}
```

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 controller comes out of 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 controller executes the onboard program configured to automatically start. The auto-start requires no host interaction once it is set up.

Automatically starting the onboard programs is very useful if you need 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. This is 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 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*

16

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 and 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 means it is less likely a defect can go undetected. Shortening the scan time means the cycle time is lower, and your production or testing runs faster.

There are three ways to do this. The first method is to move the stage in a raster by connecting several straight-line move segments. The second method is to use blending to make the scan a single continuous move. The third method is to 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 will also increase the cycle time.

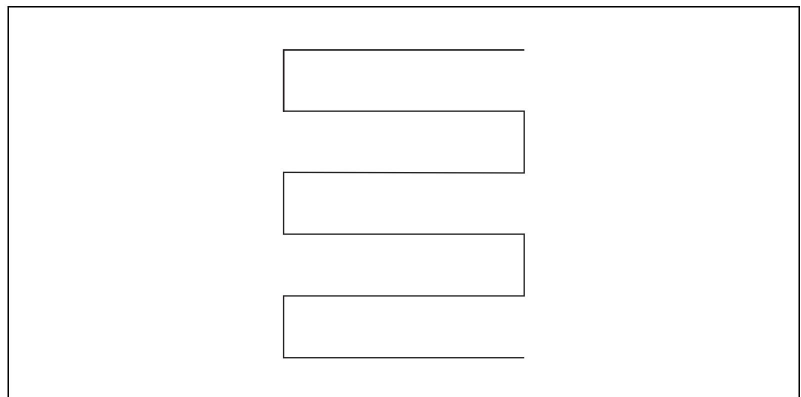


Figure 16-1. Raster Scanning Path

Algorithm

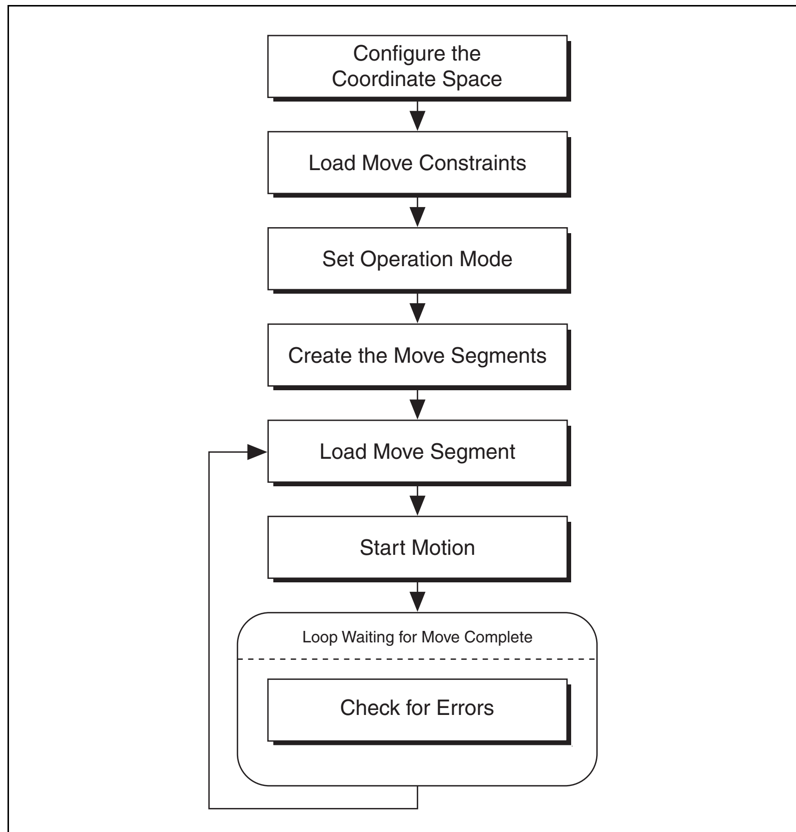


Figure 16-2. Raster Scanning Using Straight Lines Algorithm

In this example, the motors come to a stop after every segment of the move, so the cycle time is longer than other methods.

LabVIEW Code

Figure 16-3. Scanning Using LabVIEW

NI-Motion VIs for Figure 16-3, in order from left to right:

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

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 more complete and can compile.

```
# 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 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
}
```

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 16-4 shows the path of the blended move segments.

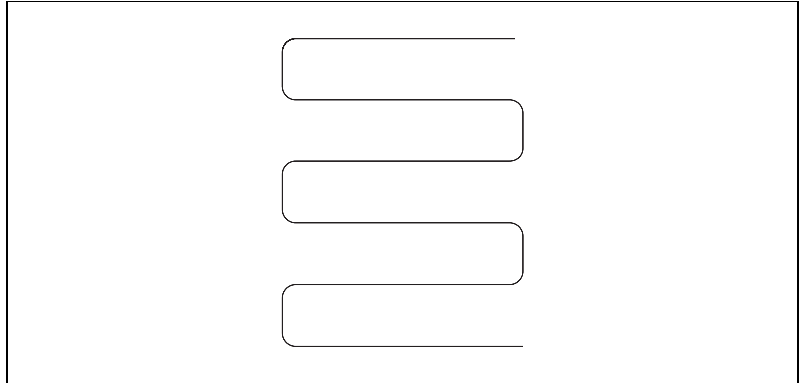


Figure 16-4. Blended Raster Scanning Path

Refer to Chapter 10, *Blending Your Moves*, for more information on using blending with NI-Motion.

Algorithm

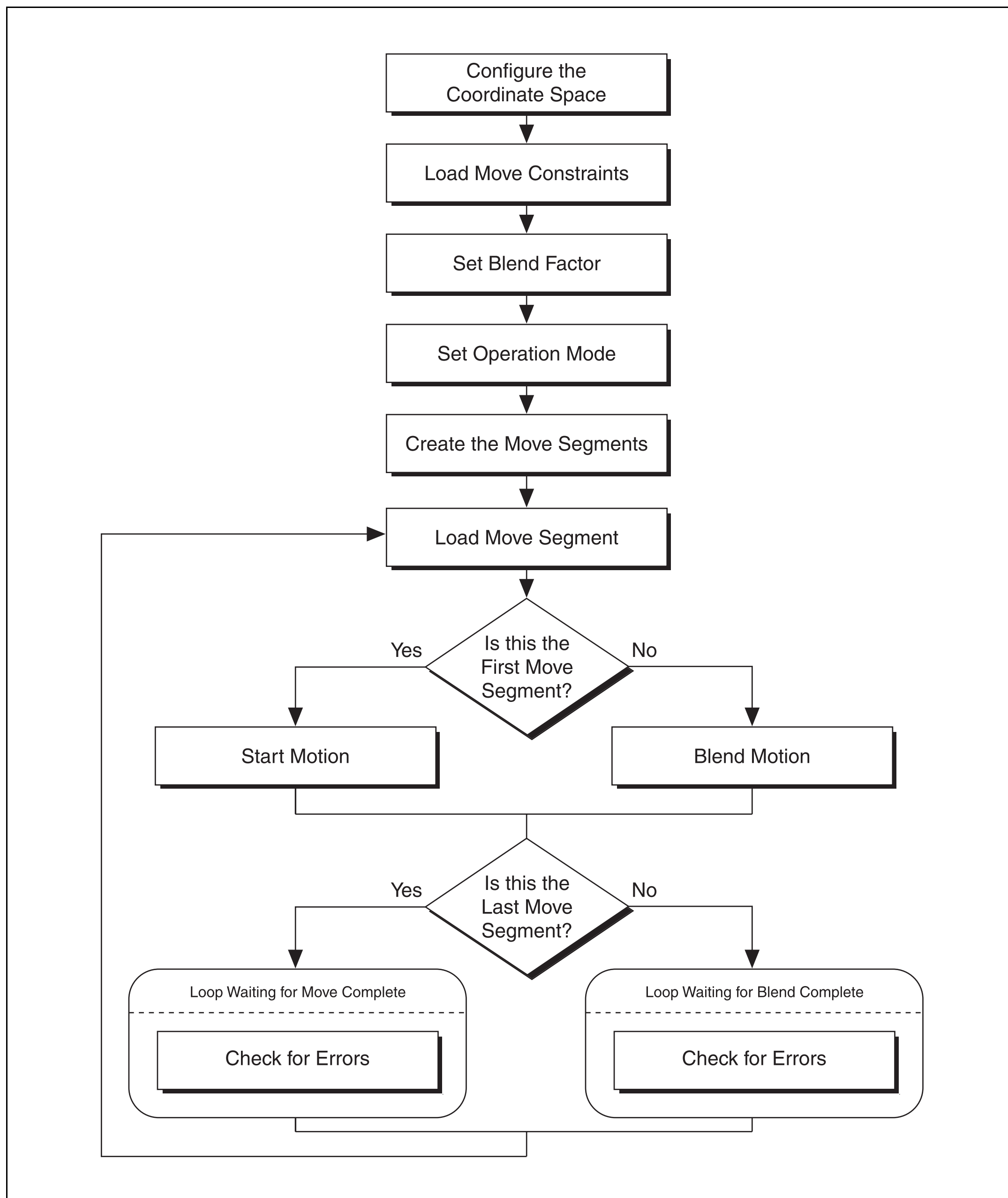


Figure 16-5. Raster Scanning Using Blended Straight Lines Algorithm

LabVIEW Code

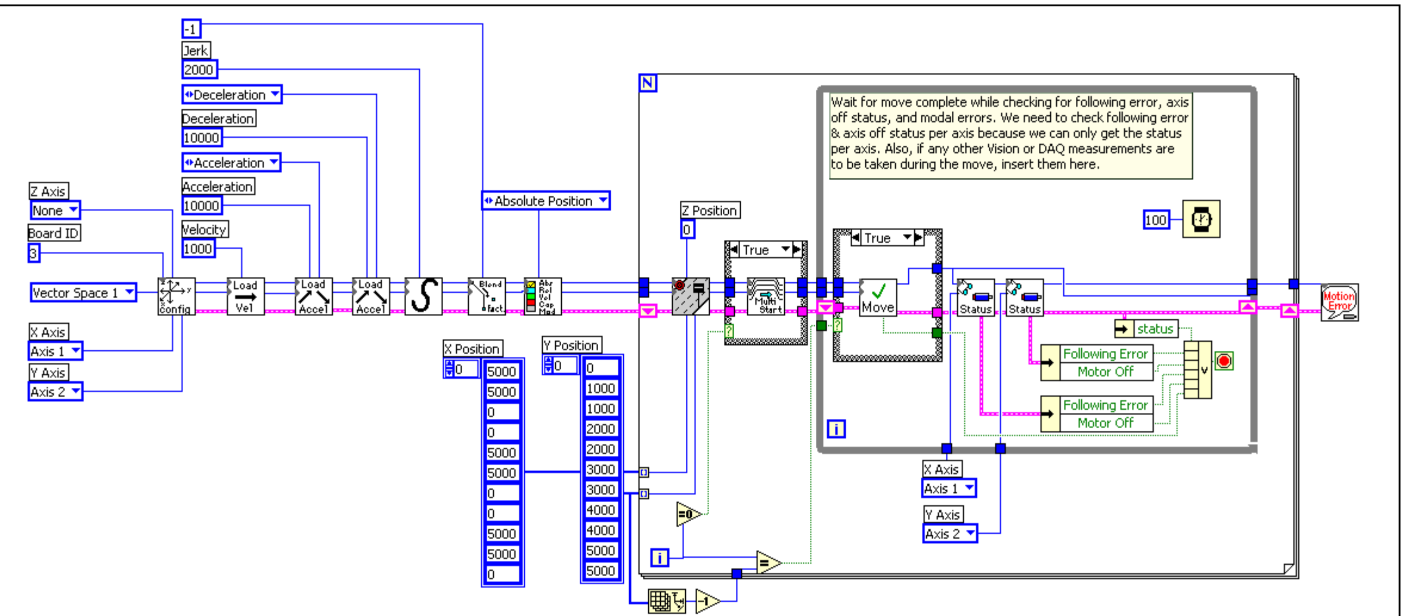


Figure 16-6. Scanning Using Blending

NI-Motion VIs for Figure 16-6, in order from left to right:

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

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 more complete and can compile.

```
#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(boardID,
                                             vectorSpace, 0,
                                             &complete);
       CheckError;
      }else{
       // 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(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 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
}
```

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 16-7 shows the scanning path used in the example that follows.

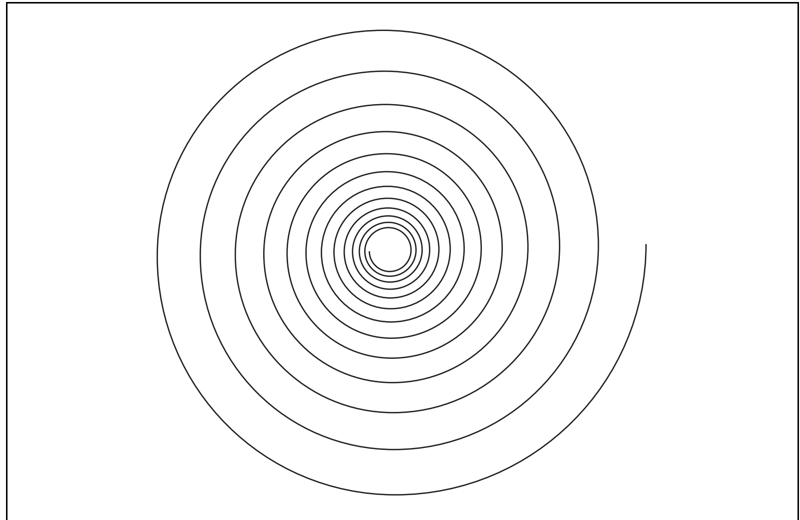


Figure 16-7. User-Defined Scanning Path

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 8, *Contoured Moves*, for more information on using contoured moves with NI-Motion.

Algorithm

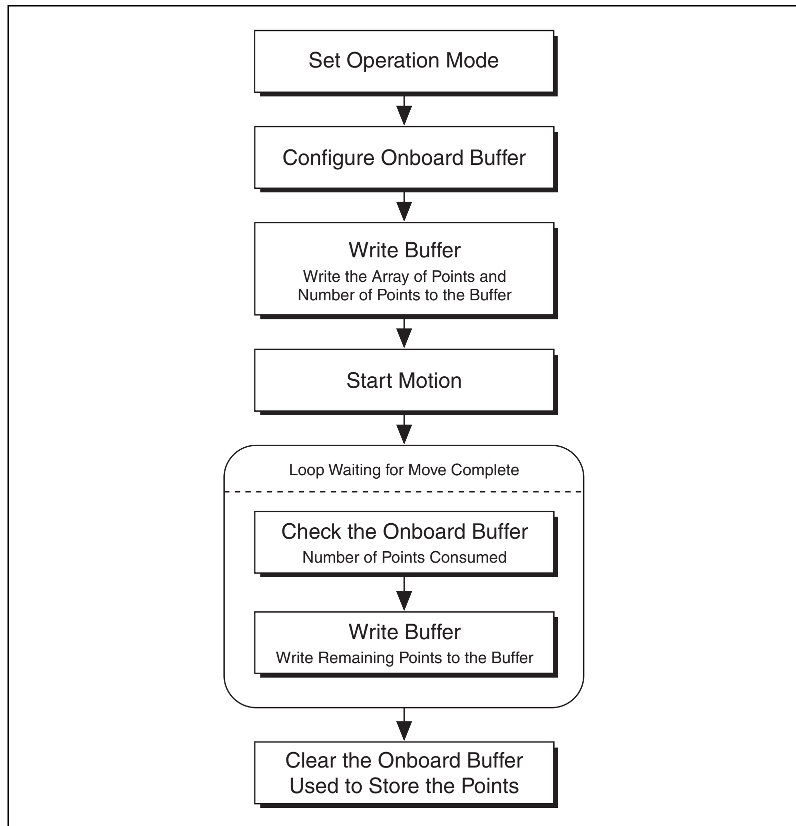


Figure 16-8. User-Defined Scanning Path Algorithm

LabVIEW Code

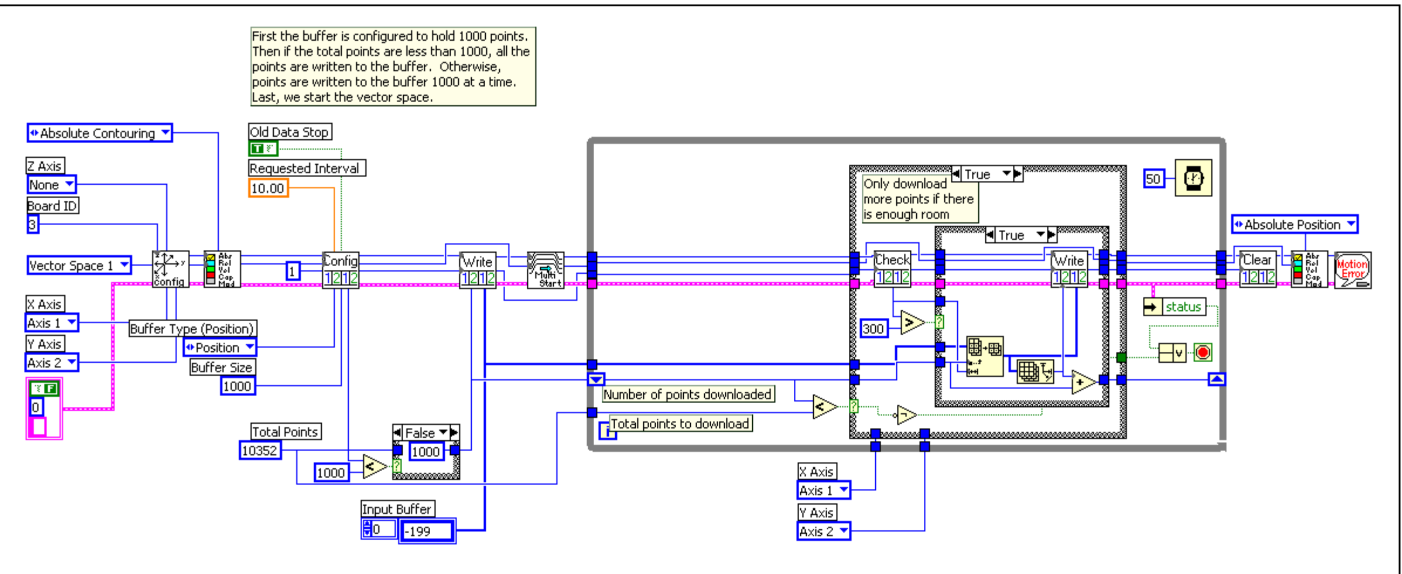


Figure 16-9. Scanning Using Contouring

NI-Motion VIs for Figure 16-9, in order from left to right:

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

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 more complete and can compile.

```
// 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 controller can
                       //really contour
    i32* downloadData = NULL;// The temporary array that is created to
                             //download the points to the 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 controller out of
   //contouring mode
```

```
    err = flex_set_op_mode(boardID, vectorSpace,
                           NIMC_ABSOLUTE_POSITION);
    CheckError;

    // Free the buffer allocated on the controller memory
    err = flex_clear_buffer(boardID, 1/*buffer number*/);
    CheckError;
    return;// Exit the Application

    // Error Handling
    nimcHandleError; //NIMCCATCHTHIS:

    // Check to see if there were any Modal Errors
    if (csr & NIMC_MODAL_ERROR_MSG){
        do{

            //Get the command ID, resource ID, 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
}
```

17 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. As the web material can stretch under 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 once per cycle at the required cutting location. The NI motion controller reads this mark via a sensor and performs the necessary correction.

To make this example simpler, it is assumed that the length of the cut is equal to the circumference of the knife. This means that under ideal conditions the mark should be read when the blade is at position A, as shown in Figure 17-1. Therefore, the motor should move one revolution without any correction before causing the cut.

Solution

The rotary knife is electronically geared to the web with a gear ratio of 1:1. This ensures that at the time of cut, the speed of the web and the knife is the same. This is essential 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 NI motion controller. As the length of the labels to be cut can vary due to elasticity of the web material, the mark can be read before the blade is at position A or after it is at position A. Therefore, the application must correct the position where the blade of the rotary knife should be when the high-speed capture occurs. However, 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 a position B, as shown in Figure 17-1.

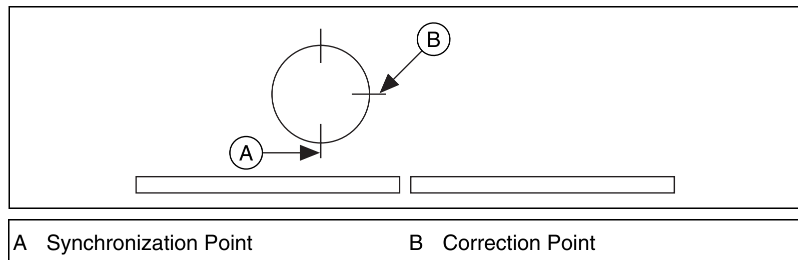


Figure 17-1. Rotating Knife

Algorithm

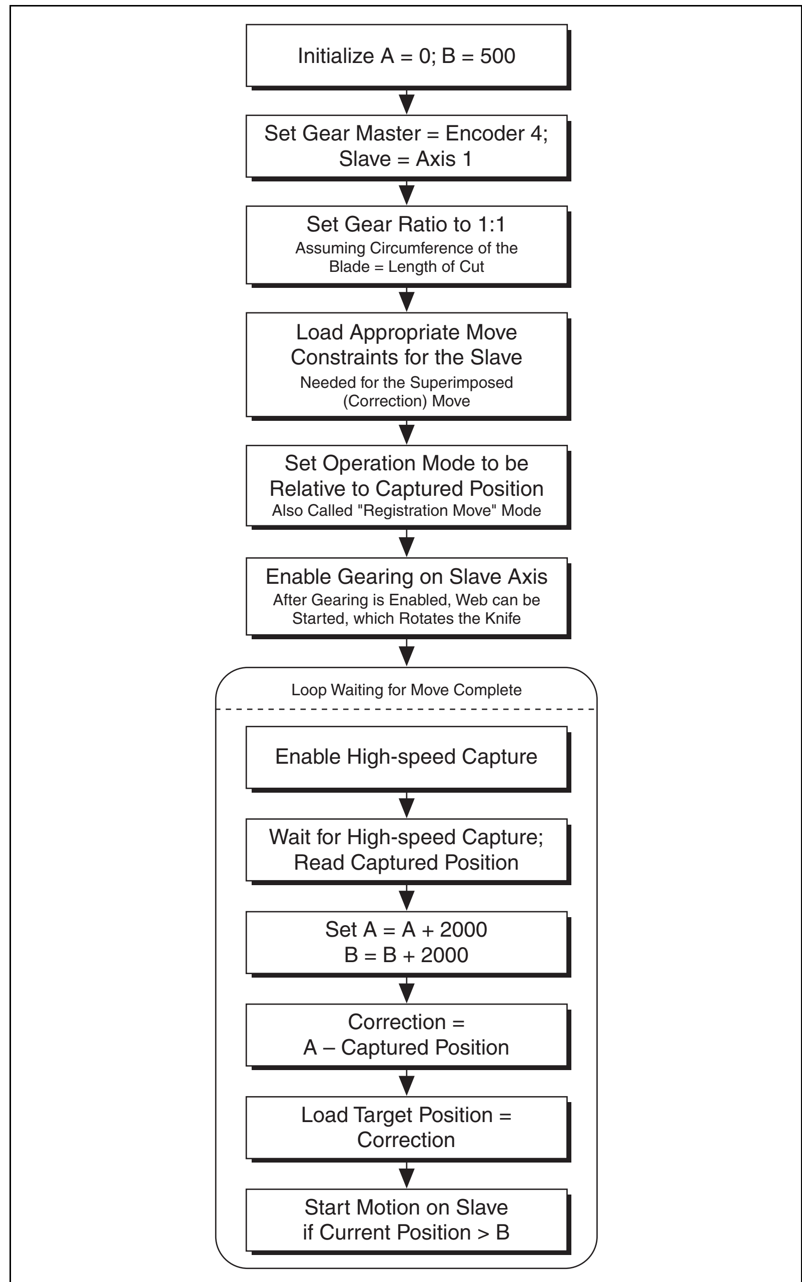


Figure 17-2. Rotating Knife Application Algorithm

LabVIEW Code

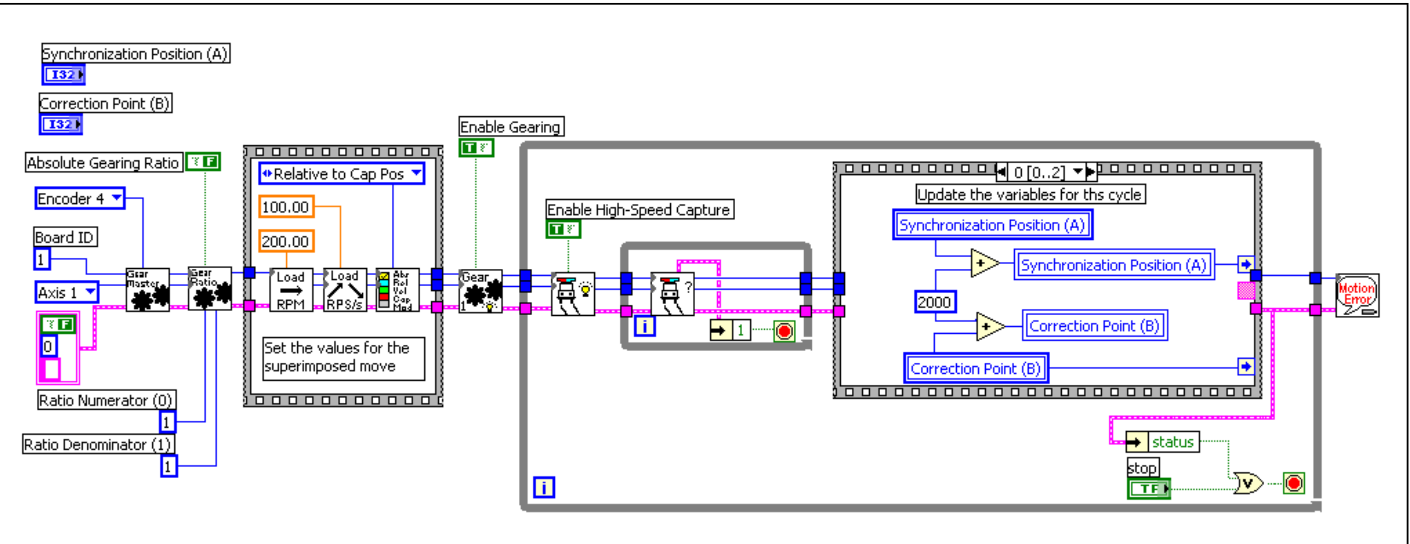


Figure 17-3. Rotating Knife Application Using LabVIEW

NI-Motion VIs for Figure 17-3, in order from left to right:

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

Figures 17-4 and 17-4 show the remaining cases for the block diagram in Figure 17-3.

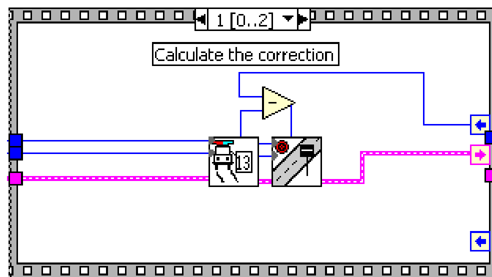


Figure 17-4. Figure 17-3 Sequence Structure 1

NI-Motion VIs for Figure 17-4, in order from left to right:
1) Read Captured Position 2) Load Target Position

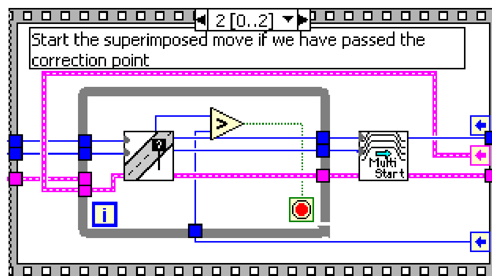


Figure 17-5. Figure 17-3 Sequence Structure 2

NI-Motion VIs for Figure 17-5, in order from left to right:
1) Read Position 2) Start Motion

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 more complete and can compile.

```
// 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 cyle
        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 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
}
```

A

NI-Motion Issues

This appendix contains general programming tips and information designed to help you develop your NI-Motion application.

Handling Following Error (Position Error)

Following Error, sometimes referred to as Position Error, is the difference between the commanded position and the actual position reported by the feedback device. This difference is used by the PID control to calculate the command output. It also can be used to determine how far off the calculated trajectory the motion system is at any given moment.

The amount of following error present is different for each motion system. Depending on how well a system is tuned, the following error could be greater or smaller. Following error may increase during the acceleration and deceleration periods in the motion profile. The steepness of the acceleration or deceleration affects the following error, as do the size of the motors and the system load.

The amount of following error that is acceptable depends also on each motion system and its requirements. A system performing coarse motion between two positions may not be adversely affected by a larger following error, whereas a precision motion system or cutting system may have tolerances that require a very low following error.

You can use following error to protect the physical system. When the motion controller detects following error greater than the limit you specify, it triggers an event that stops all motion.

For example, suppose that under normal operation the following error on a motion system remains under 10 counts. If following error exceeds 10 counts, it could mean that something is blocking the motion or that the system needs maintenance.

NI motion controllers set the command voltage to 0 and set the inhibit line to the active state. If the inhibits are connected to the drive, the drive is disabled. This allows the motors to spin freely. You also can connect the

inhibit lines to an external brake to stop further motion or to lock the system in place.

When you first connect a system, set following error to a small number to protect the system from a runaway motor. Choose a following error small enough that a runaway motor cannot hit a physical limit of the system.

You may want to use a larger maximum following error while you are tuning the system. This gives you greater flexibility that may be necessary while making adjustments to the system.

Once the motion system is tuned, set the following error according to the final system requirements. Always keep safety in mind when setting following error.

B

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 include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at ni.com/support. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
  - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting ni.com/ask. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit ni.com/training for self-paced tutorials, videos, and interactive CDs. 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, NI Alliance Program 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

Symbol	Prefix	Value
p	pico	10^{-12}
n	nano	10^{-9}
μ	micro	10^{-6}
m	milli	10^{-3}
k	kilo	10^{3}
M	mega	10^{6}
G	giga	10^{9}
T	tera	10^{12}

Numbers/Symbols

°	degrees
/	per
%	percent
±	plus or minus
+	positive of, or plus
–	negative of, or minus
+5 V	+5 VDC source signal

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	a 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	a signal is active when its value is high (1)
active-low	a 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	the 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	a number system with a base of 2
buffer	temporary storage for acquired or generated data (software)

bus — the 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 direction of rotation of the motor

closed-loop — a motion 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 direction of motor rotation

D

DAC — Digital-to-Analog Converter

DAQ — Data Acquisition

digital I/O port — a group of digital input/output signals

DLL — dynamic link library—provides the API for the motion control boards

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 board

E

encoder	a device that translates mechanical motion into electrical signals; used for monitoring position or velocity in a closed-loop system
encoder resolution	the number of encoder lines between consecutive encoder indexes (marker or Z-bit). If the encoder does not have an index output the encoder resolution can be referred to as lines per revolution.

F

F	Farad
FIFO	First-In, First-Out
filtering	a type of signal conditioning that filters unwanted signals from the signal being measured
filter parameters	indicates the control loop parameter gains (PID gains) for a given axis
flash ROM	a type of electrically reprogrammable read-only memory
following error trip point	the 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 is done by 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 is done by 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)	a 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 which 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	the polarity of a switch (limit switch, home switch, and so on) in *active* state. If these switches are active-low they are said to 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 physical end of travel is being approached and that the motion should stop

M

m — meters

MCS — Move Complete Status

microstep — the 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 — an undesirable electrical signal—noise comes from external sources such as the AC power line, motors, generators, transformers, fluorescent lights, soldering irons, CRT displays, computers, electrical storms, welders, radio transmitters, and internal sources such as semiconductors, resistors, and capacitors. Noise corrupts signals you are trying to send or receive.

noninverting — the polarity of a limit switch, home switch, and so on, in *active* state. If these switches are active-high, they are said to have non-inverting polarity.

O

open collector — a method of output capable of sinking current, but not sourcing current

open-loop — refers to a motion control system where no external sensors (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) a communications connection on a computer or a remote controller; (2) a digital port, consisting of eight lines of digital input and/or output
position breakpoint	position breakpoint for an encoder can be set in absolute or relative quadrature counts. When the encoder reaches a position breakpoint, the associated breakpoint output immediately transitions.
PWM	Pulse Width Modulation—a 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	the 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	a 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	a method of output capable of sinking and sourcing current
trapezoidal profile	a typical motion trajectory, where a motor accelerates up to the programmed velocity using the programmed acceleration, traverses at the programmed velocity, 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	a timer task that shuts down (resets) the motion control board if any serious error occurs
word	the standard number of bits that a processor or memory manipulates at one time, typically 8-bit, 16-bit, or 32-bit

Index

Numerics

1D Interactive test, 5-4
733*x*, 3-1
734*x*, 3-1

A

absolute contouring, 8-3
acceleration feedforward, 4-29
acceleration in counts/s^2, III-8
acceleration in RPS/s, III-9
acquiring data
    algorithm, 12-2
    C/C++ code, 12-4
    data path, 12-1
    LabVIEW code, 12-3
actuator, 3-4
ADC settings, 4-18
algorithms
    position-based straight-line move, 6-2
    velocity-based straight-line move, 6-10
amplifier, 3-3
Amplifier Gain, 4-29
analog feedback
    algorithm, 14-3
    C/C++ code, 14-5
    flowchart, 14-1
    LabVIEW code, 14-4
analog I/O, 3-6
application notes, 1-5
applications
    rotating knife, 17-1
        algorithm, 17-3
        C/C++ code, 17-5
        LabVIEW code, 17-4
        solution, 17-2
    scanning, 16-1
        blending move segments, 16-7
            algorithm, 16-8
            C/C++ code, 16-10
            LabVIEW code, 16-9
        connecting move segments, 16-1
            algorithm, 16-2
            C/C++ code, 16-4
            LabVIEW code, 16-3
        user-defined scan path, 16-13
            algorithm, 16-14
            C/C++ code, 16-16
            LabVIEW code, 16-15
arc angles in degrees, III-10
arc move, III-4
    circular, 7-1
        algorithm, 7-3
        C/C++ code, 7-5
        LabVIEW code, 7-4
    helical, 7-14
        algorithm, 7-15
        C/C++ code, 7-17
        LabVIEW code, 7-16
    spherical, 7-7
        algorithm, 7-9
        C/C++ code, 7-11
        LabVIEW code, 7-10
arc moves, 7-1
automatically starting onboard programs, 15-41
axis configuration, 4-8

B

blending, 10-1
    after delay, 10-4
    after first move, 10-3
    algorithm, 10-5

- C/C++ code, 10-7
- LabVIEW code, 10-6
- superimposing, 10-2

blending moves, 10-1
branching onboard programs
- algorithm, 15-19
- C/C++ code, 15-21
- LabVIEW code, 15-20

breakout box, 3-4
breakpoint and trigger, 4-10
breakpoints, 3-7
breakpoints using RTSI, 13-37
breakpoints. *See* synchronization, 13-1
buffers
- onboard
  - algorithm, 15-25
  - data flow, 15-24

C

C/C++ code
- position-based straight-line move, 6-5
- velocity profiling using velocity override, 6-19

calibration, 4-24
changing a time slice, 15-41
check reference, 9-1
circular arc move, 7-1
- algorithm, 7-3
- C/C++ code, 7-5
- LabVIEW code, 7-4

clockwise/counter-clockwise mode, 3-2
commutation frequency, 4-36
commutation, sinusoidal, 4-34
- commutation frequency, 4-36
- determining counts per electrical cycle, 4-35
- phase initialization
  - direct set, 4-35
  - Hall effect sensors, 4-35
  - shake and wake, 4-35
- troubleshooting Hall effect sensors, 4-36

conditional execution of onboard programs, 15-8
- algorithm, 15-10
- C/C++ code, 15-11
- LabVIEW code, 15-11

configuration, 4-1
- ADC settings, 4-18
- axis configuration, 4-8
- breakpoint and trigger, 4-10
- calibration, 4-24
- control loop, 4-10
- current settings, 4-6
- device resources, 4-4
- digital I/O settings, 4-17
- encoder settings, 4-19
- find reference settings, 4-14
  - home, 4-14
  - index, 4-15
  - limits, 4-16
  - run sequence, 4-15
- gearing settings, 4-18
- initialization preferences, 4-7
- initialization settings, 4-4
- interactive, 4-20
- miscellaneous tab, 4-12
- motion I/O, 4-9
- move criteria, 4-14
- PWM settings, 4-19
- static friction, 4-11
- synchronization settings, 4-20
- trajectory settings, 4-13
- tuning, 4-24

contacting National Instruments, B-1
contoured move, III-4, 8-1
- absolute versus relative, 8-3
- algorithm, 8-2
- C/C++ code, 8-6
- data path, 8-1
- LabVIEW code, 8-4

contoured moves, 8-1

control loop, 1-4, 4-10, 4-25
acceleration feedforward, 4-29
derivative gain, 4-28
dual loop feedback, 4-30
algorithm, 4-31
Ga, 4-29
integral gain, 4-27
Kdac, 4-29
Kt, 4-30
proportional gain, 4-27
velocity feedback, 4-28, 4-31
algorithm, 4-33
velocity amplifiers, 4-33
velocity feedforward, 4-28
control loop (figure), 4-26
controller, 3-1
servo control, 3-2
step generation, 3-2
microstepping, 3-3
controlling torque, 14-1
coordinate space, III-4
counts per electrical cycle, 4-35
creating NI-Motion applications, 2-1
generic steps diagram, 2-2
I/O diagram, 2-3
current settings, 4-6
customer
education, B-1
professional services, B-1
technical support, B-1
CW/CCW mode, 3-2

D

data, acquiring time-sampled position and velocity
algorithm, 12-2
C/C++ code, 12-4
data path, 12-1
LabVIEW code, 12-3
derivative gain, 4-28
device resources, 4-4
diagnostic resources, B-1
digital I/O, 3-6
digital I/O settings, 4-17
Digital to Analog Converter gain, 4-29
direct set, 4-35
documentation, 1-5
online library, B-1
drive, 3-3
drivers
instrument, B-1
software, B-1
dual loop feedback, 4-30
algorithm, 4-31

E

electrical cycle, counts per, 4-35
electronic gearing. *See* gearing
encoders, 3-5
pulses using RTSI, 13-38
settings, 4-19
testing the encoders, 5-1
event polling, III-13
example code, B-1
examples, 1-5
default installation directory, 1-6
export settings, 4-4

F

feedback, 3-5
dual loop, 4-30
algorithm, 4-31
velocity, 4-31
algorithm, 4-33
velocity amplifiers, 4-33
find home, 9-1
find index, 9-1
find reference, 9-1

find reference settings
- home, 4-14
- index, 4-15
- limits, 4-16
- run sequence, 4-15

fixed point, III-5
floating point, III-5
floating point versus fixed point, III-5
forward limit, 9-1
frequency, commutation, 4-36
frequently asked questions, B-1

G

Ga, 4-29
gear ratio, 11-1
gearing, 11-1
- algorithm, 11-2
- C/C++ code, 11-5
- LabVIEW code, 11-4

gearing settings, 4-18
graphing data, III-12

H

Hall effect sensors, 4-35
Hall effect sensors, troubleshooting, 4-36
helical arc move, 7-14
- algorithm, 7-15
- C/C++ code, 7-17
- LabVIEW code, 7-16

help
- application notes, 1-5
- Motion Hardware Advisor, 1-6
- NI Developer Zone, 1-6
- professional services, B-1
- technical support, B-1

high-speed capture, 3-8
high-speed capture input using RTSI, 13-38
high-speed capture. *See* synchronization, 13-1
home, 3-7, 9-1
home switch, 3-7

I

import settings, 4-5
index, 9-1
indirect variables, onboard programs, 15-23
inhibit output, 3-8
initialization preferences, 4-7
initialization settings, 4-4
initialization, programmatic, 4-37
input/output. *See* synchronization, 13-1
instrument drivers, B-1
integral gain, 4-27
interactive panels, 4-20
introduction, I-1
- actuator, 3-4
- amplifier, 3-3
- analog I/O, 3-6
- breakout box, 3-4
- configuring the system, 4-1
  - ADC settings, 4-18
  - axis configuration, 4-8
  - breakpoint and trigger, 4-10
  - calibration, 4-24
  - control loop, 4-10
  - current settings, 4-6
  - digital I/O settings, 4-17
  - encoder settings, 4-19
  - find reference settings, 4-14
    - index, 4-15
    - limits, 4-16
    - run sequence, 4-15
  - gearing settings, 4-18
  - initialization preferences, 4-7
  - interactive, 4-20
  - miscellaneous tab, 4-12
  - motion I/O, 4-9
  - move criteria, 4-14
  - PWM settings, 4-19

real-time, 4-37
static friction, 4-11
synchronization settings, 4-20
trajectory settings, 4-13
tuning the motors, 4-24
configuring your system
device resources, 4-4
find reference settings
home, 4-14
initialization settings, 4-4
control loop, 4-25
acceleration feedforward, 4-29
derivative gain, 4-28
dual loop feedback, 4-30
algorithm, 4-31
Ga, 4-29
integral gain, 4-27
Kdac, 4-29
Kt, 4-30
proportional gain, 4-27
velocity feedback, 4-28, 4-31
algorithm, 4-33
velocity amplifiers, 4-33
velocity feedforward, 4-28
control loop (figure), 4-26
creating NI-Motion applications, 2-1
generic steps diagram, 2-2
I/O diagram, 2-3
digital I/O, 3-6
documentation, 1-5
drive, 3-3
encoders, 3-5
examples, 1-5
feedback, 3-5
mechanics, 3-6
motion control components, 3-1
motion controller, 3-1
servo control, 3-2
step generation, 3-2
microstepping, 3-3
motion I/O, 3-6
breakpoints, 3-7
high-speed capture, 3-8
home, 3-7
inhibit output, 3-8
limits, 3-6
motor, 3-4
NI motion controller architecture
control loop, 1-4
functional architecture, 1-4
functional architecture diagram, 1-5
motion I/O, 1-4
physical architecture, 1-2
supervisory control, 1-4
trajectory generator, 1-4
NI-Motion, 1-1
NI-Motion, architecture, 1-1
software/hardware interaction, 1-2
stage, 3-6
UMI, 3-4

J

jogging, 6-9, 6-16

K

Kdac, 4-29
KnowledgeBase, B-1
Kt, 4-30

L

LabVIEW code
position-based straight-line move, 6-3
velocity profiling using velocity override, 6-18
velocity-based straight-line move, 6-12
limit switch, 3-6
limits, 3-6, 9-1
looping onboard programs

algorithm, 15-19
C/C++ code, 15-21
LabVIEW code, 15-20
loops, timing
event polling, III-13
graphing data, III-12
status display, III-12

M

master axis, 11-1, 11-3
math operations, onboard programs, 15-23
MAX configuration, 4-1
ADC settings, 4-18
axis configuration, 4-8
breakpoint and trigger, 4-10
calibration, 4-24
control loop, 4-10
current settings, 4-6
device resources, 4-4
digital I/O settings, 4-17
encoder settings, 4-19
find reference settings
home, 4-14
index, 4-15
limits, 4-16
run sequence, 4-15
gearing settings, 4-18
initialization preferences, 4-7
initialization settings, 4-4
interactive, 4-20
miscellaneous tab, 4-12
motion I/O, 4-9
move criteria, 4-14
PWM settings, 4-19
real-time, 4-37
static friction, 4-11
synchronization settings, 4-20
trajectory settings, 4-13
tuning, 4-24
mechanics, 3-6
miscellaneous tab, 4-12
monitoring force
algorithm, 14-9
C/C++ code, 14-11
flowchart, 14-8
LabVIEW code, 14-10
motion control components, 3-1
motion controller, 3-1
servo control, 3-2
step generation, 3-2
microstepping, 3-3
Motion Hardware Advisor, 1-6
motion I/O, 1-4, 4-9
breakpoints, 3-7
high-speed capture, 3-8
home, 3-7
inhibit output, 3-8
limits, 3-6
motor, 3-4
motor drive, 3-3
move constraints, III-2
move criteria, 4-14
move profile
move constraints, III-2
s-curve, III-3
trapezoidal, III-2
moves
arc move, III-4
circular, 7-1
algorithm, 7-3
C/C++ code, 7-5
LabVIEW code, 7-4
helical, 7-14
algorithm, 7-15
C/C++ code, 7-17
LabVIEW code, 7-16
spherical, 7-7
algorithm, 7-9
C/C++ code, 7-11
LabVIEW code, 7-10

arc moves, 7-1
blending, 10-1
after delay, 10-4
after first move, 10-3
algorithm, 10-5
C/C++ code, 10-7
LabVIEW code, 10-6
superimposing, 10-2
contoured move, III-4
absolute versus relative, 8-3
algorithm, 8-2
C/C++ code, 8-6
data path, 8-1
LabVIEW code, 8-4
contoured moves, 8-1
gearing, 11-1
algorithm, 11-2
C/C++ code, 11-5
LabVIEW code, 11-4
reference move, III-3
algorithm, 9-2
C++ code, 9-4
check reference, 9-1
find reference, 9-1
LabVIEW code, 9-3
wait reference, 9-1
reference moves, 9-1
straight-line move, III-3
position-based, 6-1
algorithm, 6-2
C/C++ code, 6-5
LabVIEW code, 6-3
velocity profiling using velocity override, 6-16
algorithm, 6-17
C/C++ code, 6-19
LabVIEW code, 6-18
velocity-based, 6-9
algorithm, 6-10
LabVIEW code, 6-12
straight-line moves, 6-1
multi-dimension moves, III-4

N

National Instruments
customer education, B-1
professional services, B-1
system integration services, B-1
technical support, B-1
worldwide offices, B-1
NI Developer Zone, 1-6
NI motion controller
control loop, 1-4
functional architecture, 1-4
functional architecture diagram, 1-5
motion I/O, 1-4
physical architecture, 1-2
supervisory control, 1-4
trajectory generator, 1-4
NIDZ, 1-6
NI-Motion architecture, 1-1
NI-Motion, introduction, 1-1

O

onboard buffers
algorithm, 15-25
data flow, 15-24
onboard programs, 15-1
algorithm, 15-4
automatically starting, 15-41
branching
algorithm, 15-19
C/C++ code, 15-21
LabVIEW code, 15-20
buffers
algorithm, 15-25
data flow, 15-24
changing a time slice, 15-41
conditional execution, 15-8

algorithm, 15-10
C/C++ code, 15-11
LabVIEW code, 15-11
description, 15-1
indirect variables, 15-23
looping
algorithm, 15-19
C/C++ code, 15-21
LabVIEW code, 15-20
math operations, 15-23
pausing, 15-7
automatic, 15-7
single-stepping, 15-8
priority, 15-2
resuming, 15-7
running, 15-6
simple C/C++ code, 15-5
simple LabVIEW code, 15-4
stopping, 15-7
subroutines
algorithm, 15-33
C/C++ code, 15-37
LabVIEW code, 15-34
synchronizing host applications with, 15-25
algorithm, 15-26
C/C++ code, 15-29
data flow, 15-26
LabVIEW code, 15-27
using onboard memory and data, 15-13
algorithm, 15-14
C/C++ code, 15-16
LabVIEW code, 15-15
writing, 15-3
online technical support, B-1
output. *See* synchronization, 13-1

P

pausing onboard programs, 15-7
automatic, 15-7
single-stepping, 15-8
phase initialization
direct set, 4-35
Hall effect sensors, 4-35
shake and wake, 4-35
phone technical support, B-1
position breakpoints using RTSI, 13-37
professional services, B-1
programmatic initialization, 4-37
programming examples, B-1
programs, onboard. *See* onboard programs
proportional gain, 4-27
PWM settings, 4-19

R

radius, 7-1
ratio, gear, 11-1
real-time, using NI motion controllers with, 4-37
reference move, III-3
algorithm, 9-2
C/C++ code, 9-4
LabVIEW code, 9-3
reference moves, 9-1
relative contouring, 8-3
resuming onboard programs, 15-7
reverse limit, 9-1
rotating knife application, 17-1
algorithm, 17-3
C/C++ code, 17-5
LabVIEW code, 17-4
solution, 17-2
RTSI
encoder pulses, 13-38
hardware implementation, 13-36
high-speed capture input, 13-38
software trigger, 13-38
RTSI, using breakpoints with, 13-37
run sequence, 9-1
running onboard programs, 15-6

S

scanning
- blending move segments, 16-7
  - algorithm, 16-8
  - C/C++ code, 16-10
  - LabVIEW code, 16-9
- connecting move segments, 16-1
  - algorithm, 16-2
  - C/C++ code, 16-4
  - LabVIEW code, 16-3
- user-defined scan path, 16-13
  - algorithm, 16-14
  - C/C++ code, 16-16
  - LabVIEW code, 16-15

s-curve, III-3
servo control, sinusoidal commutation, 3-2
servo tuning, 4-24
- control loop, 4-25
  - acceleration feedforward, 4-29
  - derivative gain, 4-28
  - dual loop feedback, 4-30
    - algorithm, 4-31
  - Ga, 4-29
  - integral gain, 4-27
  - Kdac, 4-29
  - Kt, 4-30
  - proportional gain, 4-27
  - velocity feedback, 4-28, 4-31
    - algorithm, 4-33
    - velocity amplifiers, 4-33
  - velocity feedforward, 4-28
- control loop (figure), 4-26

shake and wake, 4-35
single-stepping onboard programs, 15-8
sinusoidal commutation, 3-2, 4-34
- commutation frequency, 4-36
- determining counts per electrical cycle, 4-35
- phase initialization
  - direct set, 4-35
  - Hall effect sensors, 4-35
  - shake and wake, 4-35
- troubleshooting Hall effect sensors, 4-36

slave axis, 11-1, 11-3
software drivers, B-1
software trigger using RTSI, 13-38
software/hardware interaction, 1-2
speed control, 14-13
- algorithm, 14-14
- C/C++ code, 14-16
- LabVIEW code, 14-15

spherical arc move, 7-7
- algorithm, 7-9
- C/C++ code, 7-11
- LabVIEW code, 7-10

stage, 3-6
start angle, 7-2
static friction, 4-11
status display, III-12
step and direction mode, 3-2
step generation, 3-2
- microstepping
  - microstepping, 3-3
- resolution, III-5

stepper
- CW/CCW mode, 3-2
- resolution, III-5
- step and direction mode, 3-2

stiction, 4-11
stopping onboard programs, 15-7
straight-line move, III-3
- position-based, 6-1
  - algorithm, 6-2
  - C/C++ code, 6-5
  - LabVIEW code, 6-3
- velocity profiling using velocity override, 6-16
  - algorithm, 6-17
  - C/C++ code, 6-19
  - LabVIEW code, 6-18

velocity-based, 6-9
algorithm, 6-10
LabVIEW code, 6-12
straight-line moves, 6-1
subroutines, onboard
algorithm, 15-33
C/C++ code, 15-37
LabVIEW code, 15-34
supervisory control, 1-4
support, technical, B-1
synchronization, 13-1
breakpoint
modes, 13-1
breakpoints
absolute, 13-2
buffered, 13-2
algorithm, 13-3
C/C++ code, 13-4
LabVIEW code, 13-4
modulo, 13-20
algorithm, 13-21
C/C++ code, 13-23
LabVIEW code, 13-22
periodic
algorithm, 13-16
C/C++ code, 13-17
LabVIEW code, 13-17
relative position, 13-11
algorithm, 13-12
C/C++ code, 13-13
LabVIEW code, 13-13
single
algorithm, 13-7
C/C++ code, 13-9
LabVIEW code, 13-8
LabVIEW code with RTSI, 13-9
high-speed capture, 13-25
buffered, 13-25
algorithm, 13-26
C/C++ code, 13-27
LabVIEW code, 13-27
non-buffered
algorithm, 13-31
C/C++ code, 13-33
LabVIEW code, 13-32
single
algorithm, 13-31
C/C++ code, 13-33
LabVIEW code, 13-32
RTSI
encoder pulses, 13-38
hardware implementation, 13-36
high-speed capture input, 13-38
software trigger, 13-38
RTSI, using breakpoints with, 13-37
synchronization settings, 4-20
synchronizing host applications with onboard programs, 15-25
algorithm, 15-26
C/C++ code, 15-29
data flow, 15-26
LabVIEW code, 15-27
system integration services, B-1

T

technical support, B-1
telephone technical support, B-1
testing the limit and home switches, 5-4
testing the motion system
1D Interactive test, 5-4
encoders, 5-1
limit and home switches, 5-4
motors, 5-2
troubleshooting, 5-3
testing the motors, 5-2
troubleshooting, 5-3
time base, III-6
time slice, changing, 15-41
timing your loops
event polling, III-13

graphing data, III-12
status display, III-12
torque constant, 4-30
torque control
analog feedback
algorithm, 14-3
C/C++ code, 14-5
flowchart, 14-1
LabVIEW code, 14-4
monitoring force, 14-8
algorithm, 14-9
C/C++ code, 14-11
flowchart, 14-8
LabVIEW code, 14-10
training, customer, B-1
trajectory generator, 1-4
trajectory parameters
acceleration in counts/s^2, III-8
acceleration in RPS/s, III-9
arc angles in degrees, III-10
fixed point, III-5
floating point, III-5
time base, III-6
velocity in RPM, III-6
velocity in steps/counts per second, III-7
velocity override in percent, III-10
trajectory settings, 4-13
trapezoidal, III-2
travel angle, 7-2
trigger inputs, 3-8
troubleshooting resources, B-1
tuning the motors, 4-24

U

UMI, 3-4
using onboard memory and data, 15-13
algorithm, 15-14
C/C++ code, 15-16
LabVIEW code, 15-15

V

variables, indirect, 15-23
vector space, III-4
velocity
counts/steps per second, III-7
RPM, III-6
velocity feedback, 4-28, 4-31
algorithm, 4-33
velocity amplifiers, 4-33
velocity feedforward, 4-28
velocity override in percent, III-10
velocity profiling, 6-9, 6-16

W

wait reference, 9-1
Web
professional services, B-1
technical support, B-1
worldwide technical support, B-1