QuickMotion Reference Guide

Doc. No. 951-530017-019

25 South Street
Hopkinton, MA 01748

Phone: 508.435.9595
Fax: 508.435.2373

Tuesday, June 18, 2013

ctc
Control Technology Corp.

Table of Contents

QuickMotion Reference Guide

Control Technology Corp.
25 South Street
Hopkinton, MA 01748
Phone: 508.435.9595 • Fax 508.435.2373

Document No. 951-530017-019

⚠ **WARNING:** Use of CTC Controllers and software is to be done only by experienced and qualified personnel who are responsible for the application and use of control equipment like the CTC controllers. These individuals must satisfy themselves that all necessary steps have been taken to assure that each application and use meets all performance and safety requirements, including any applicable laws, regulations, codes and/or standards. The information in this document is given as a general guide and all examples are for illustrative purposes only and are not intended for use in the actual application of CTC product. CTC products are not designed, sold, or marketed for use in any particular application or installation; this responsibility resides solely with the user. CTC does not assume any responsibility or liability, intellectual or otherwise for the use of CTC products.

The information in this document is current as of the following Hardware and Firmware revision levels. Some features may not be supported in earlier revisions. See www.ctc-control.com for the availability of firmware updates or contact CTC Technical Support.

Model Number	QuickBuilder Revision	Controller Firmware Revision	M3-40 Firmware Revision
5300	>=1.2.4410	>= 5.00.90R69.47	>= 1.45

1 Chapter 1: Introduction and Overview

This document provides details about adding motion control to a QuickBuilder project. QuickBuilder is CTC's integrated desktop development environment for the 5300 series automation controllers. The primary programming language used in QuickBuilder is QuickStep4 (QS4). The QuickStep multi-tasking state language was invented by CTC in the 1980s to simplify the process of programming high performance machine control applications. Over the years QuickStep has been continually refined, and now it has been extended with the addition of QuickMotion to be able to easily handle even the most demanding motion control applications in a very intuitive manner.

The focus of this document is the QuickMotion extension to QuickBuilder. It is assumed that the reader is already familiar with the QuickBuilder environment and the QuickStep programming language. This document should be used in conjunction with the QuickBuilder Reference Guide.

This document is valid for use with the following Model 5300 motion modules:

- M3-40A: 2 Axis Servo Module
- M3-40B: 3 axis Stepper/High-speed Counter Module (24V)
- M3-40C: 3 asix Stepper/High-speed Counter Module (5V)

Detailed data sheets for these motion modules may be found on CTC's website, www.ctc-control.com.

1.1 Guide to Symbols

Features that warrant caution or special consideration are denoted by a ⚠.

A command or statement that is supported by a given mode or block is denoted by a checked box ☑. Unsupported commands and statements are denoted by an empty checkbox ☐ .

1.2 Brief Overview of Motion Control

1.2.1 Servo Motor Applications

Background

A servo motor is used in a closed loop control system, where the controller has information about both the actual position and velocity of the motor as well as the desired position (or velocity). The controller then adjusts the motor's output to remove the difference between the actual and desired values. Because this system has information about the error, and the output (which is usually proportional to the motor power) increases as the error increases, it can require very little power when the error is small.

This means that the average power needed for a high performance application may be considerably less than the peak power, so smaller motors and drives may be used.

There is usually a Servo Drive module between the motion controller and the motor that accepts the control signal (torque or velocity command) from the motion controller (a low current signal in the range -10 Volts to +10

Volts) and converts it into the high power (depending on the motor, several amps of current at 24V to 200+V) signals required by the motor. The Servo Drive must usually be configured to match the Motor (or is designed specifically for the motor). The drive and the motor are often, but not necessarily, made by the same manufacturer. The motor may be a simple brush type DC motor (which is low cost but requires periodic replacement of brushes) or a Brushless DC or AC motor, which requires additional circuitry in the Servo Drive to handle electronic commutation and will generally require additional sensors and signals from the motor to the driver.

Controlling the Servo Motor

The Model 5300 automation controller can be used to control up to 64 axes of servo motors. To control motion, an M3-40A motion module is added to the system. The M3-40A module is a dual axis servo controller that can control 1 or 2 servo motor systems with Analog Torque or Velocity command and Quadrature encoder feedback.

Servo Command Output

The output of the Servo Controller is an Analog signal that can vary from -10V to 10V with 16 bit resolution. The analog output is used, via a servo amplifier, to control the current in a DC motor generating torque at the shaft. The amplifier may also handle other functions such as commutation for a brushless motor or it may use the analog input to control the velocity. Care must be taken in the wiring to minimize the possibility of errors being introduced into the signal by noise induced from any high power switching circuitry near to the system, since this will directly affect the quality of the control.

⚠ Shielded cabling must be used between the Servo Controller and the Servo Drive and the distance between them should be minimized.

Encoder Feedback

An encoder mounted to the motor generates two pulse signals (A, B) that are used by the M3-40A module to track the motor position. The M3-40A module can also accept a third encoder channel (the Z axis or Index) that can be used to identify a specific point in the motor rotation. This Z pulse is typically used for accurate homing of the motor.

The M3-40A encoder inputs accept a quadrature differential signal for the A and B encoder channels. The index pulse, or Z channel, is also accepted as a differential signal. The direction is counted positive, or clockwise (CW) when the A encoder phase leads the B encoder phase. Indicator LEDs for each servo axis on the module indicate the states of the A and B channels.

For some Brushless Servo systems, the Servo Drive also uses an encoder for information about the position and will provide a set of suitable encoder outputs for connection to the Servo Controller. In this case the power for the encoder is usually provided by the Servo Drive and it is not necessary to connect power for the encoder, but it is recommended that the controller's 5V return be connected to the common or return for the servo drive's encoder outputs. This limits the common mode voltage between the drive and controller and helps protect the encoder input circuits from damage caused by over voltage.

⚠ 5 VDC power is available from a dedicated 5V connector on the Model 5300 power supply modules. This connector also has a 5V return that is common to the controller's 24V return.

⚠ Shielded cabling should be used for the encoder wiring between the Servo Controller and the Servo Drive and the distance between them should be minimized.

⚠ *When the encoder output is provided by the Servo Drive, care must be taken that the signals are actually encoder signals and are not a simulated encoder generated by the Servo Drive from other signals. When the*

outputs are simulated encoder signals, there is generally a delay between the movement of the motor and the encoder signal generation. When this delay is small this is not a concern, but since the M3-40A updates the servo command at a rate of over 1250Hz, delays as small as 200µs can be significant.

The M3-40A module also has five high speed inputs and five high speed outputs that can be configured for a wide variety of functions via software. See IO Assignments later in this chapter and the Model 5300 module data sheets at http://www.ctc-control.com/customer/techinfo/idxdocs5300.asp for more details.

1.2.2 Stepper Motor Applications

Background

Stepper motors are typically used in open loop applications. A stepper motor has a fixed number of magnetic poles that determine how many steps the motor will move through during one revolution. Most stepping motors have 200 full steps that can be subdivided into smaller increments via microstepping technology built into the stepper drive. Microstepping drives can boost the number of steps per revolution to 50,000 or more providing smoother motion and more precise positioning.

Controlling the Stepper Motor

The Model 5300 automation controller can be used to control up to 64 axes of stepper motors. The motors are connected to a matched stepper drive, and then the stepper drive is commanded by the Model 5300 motion module. To control motion, an M3-40B or M3-40C stepper motion control module is added to the system. These modules are configured in QuickBuilder to match the steps per revolution of the stepper drive so that programming can be done in user units. The M3-40B/C module is a dual axis stepper controller that can control up to three stepper axes by putting out a step and direction command to the drive.

Encoder Feedback (optional)

Normally, stepper motor applications are designed to operate in an open-loop mode where there is no encoder feedback. However the M3-40B/C modules have one encoder input for each primary axis and they can be configured to monitor position via the encoder as a check on the commanded position. The encoder inputs accept a quadrature differential signal for the A and B encoder channels. The index pulse, or Z channel, is also accepted as a differential signal. The direction is counted positive, or clockwise (CW) when the A encoder phase leads the B encoder phase. Indicator LEDs for each servo axis on the module indicate the states of the A and B channels.

⚠ 5 VDC power for encoders is available from a dedicated 5V connector on the Model 5300 power supply modules. This connector also has a 5V return that is common to the controller's 24V return.

⚠ Shielded cabling should be used for the encoder wiring and the distance should be minimized.

⚠ *When the encoder output is provided by the Stepper Drive, care must be taken that the signals are actually encoder signals and are not a simulated encoder generated by the Stepper Drive from other signals. When the outputs are simulated encoder signals, there is generally a delay between the movement of the motor and the encoder signal generation. When this delay is small this is not a concern, but since the M3-40B/C updates the stepper command at a rate of over 1250Hz, delays as small as 200µs can be significant.*

The M3-40B/C modules also have five high speed inputs and five high speed outputs that can be configured for

a wide variety of functions via software. See IO Assignments later in this chapter and the Model 5300 module data sheets at http://www.ctc-control.com/customer/techinfo/idxdocs5300.asp for more details.

1.3 Brief Overview of M3-40 Motion Module Features

High performance motion control can be easily achieved with Blue Fusion Model 5300 automation controllers by adding one or more M3-40 motion modules. The M3-40 series modules are two axis motion control modules specifically designed for the Blue Fusion Model 5300 controller. They can be used to command motion on both servo and stepper motor drive systems. The M3-40 uses space saving design features that enable it to fit into a single rack slot in the Model 5300 controller. Up to 32 of the M3-40 modules can be installed into a single Model 5300 system, allowing for up to 64 axes of motion control.

Motion performance is maintained even as axes are added because each M3-40 has its own on-board processors that handle all motion related processing for two axes. CTC has fitted each dual axis module with a powerful RISC processor as well as CTC's new Motion Accelerator Chip (MAC). This gives the M3-40 modules the ability to run CTC's latest 64-bit floating point motion loops and handle local high-speed I/O events.

There are currently three M3-40 modules that can be used in the Model 5300 automation controller:

- M3-40A: 2 Axis Servo Module
- M3-40B: 3 Axis Stepper / High Speed Counter Module, 24V command
- M3-40C: 3 Axis Stepper / High Speed Counter Module, 5V command

Hardware Features

Each module is capable of controlling two axes of closed loop motion. The M3-40A can be connected to either stepper or servo drives. Each M3-40A axis has a precision 16-bit analog command signal that can command both torque and velocity mode drives, giving the designer great flexibility in motor and drive selection. Alternatively, each axis can also be set up to output step and direction signals to interface to stepper drives or intelligent servo indexers. The M3-40B and M3-40C do not have analog command capability and therefore are best suited for stepper applications.

All modules have two primary axes of control and most hardware and software functionality is divided accordingly. Each primary axis has encoder feedback inputs that operate at rates up to 17.5 MHz, accommodating even the fastest linear motors. Each primary axis has five fast user assignable inputs and five fast user assignable outputs. In addition, it is possible to configure two of the outputs on the M3-40 module (40A/B/C) to command a third open loop stepper. See the Alternative Stepper Output statement in the I/O Statements section of Chapter 4 for more on this topic.

Software Features

While the M3-40's hardware is impressive, its software capabilities are what really set it apart from the competition. The software has been designed to simplify and speed every step of the machine development process. To set up a motion axis, simply "drop" an axis object into the QuickBuilder Resource Manager. Then it can be easily configured using convenient user-units and other fill-in-the-blank properties. Dialog boxes and tuning wizards de-mystify the whole servo setup and tuning process.

CTC has taken a very modular approach to QuickBuilder's motion control capabilities. To create motion on an axis, one or more motion commands are placed in an object called a Motion Sequence Block (MSB). After

creation, that MSB can be used by any of the axes at any time. A simple example would be a homing MSB – write it once, and then use it on as many axes as desired.

To further simplify the motion programming process, CTC has created an extension to the QuickStep language within QuickBuilder called QuickMotion, which has more than 50 new commands and more than 100 specialized motion variables. QuickMotion makes programming motion applications very intuitive. For example, if one wanted to move an actuator 3.76 inches in 1.25 seconds the command would be:

Move to 3.76 ***in*** 1.25

Of course, 3.76 could just as easily be a variable or an expression that is calculated on the fly.

M3-40 Module Datasheets

Refer to Document No. 950-534001: Model M3-40A data sheet at http://www.ctc-control.com/customer/techinfo/docs/5300_950/950-534001.pdf for more detailed information on the M3-40A module.

Refer to Document No. 950-534002: Model M3-40B data sheet at http://www.ctc-control.com/customer/techinfo/docs/5300_950/950-534002.pdf for more detailed information on the M3-40B module.

Refer to Document No. 950-534003: Model M3-40C data sheet at http://www.ctc-control.com/customer/techinfo/docs/5300_950/950-534003.pdf for more detailed information on the M3-40C module.

1.3.1 Model M3-40 Motion Module Features

- Two axes of servo or stepper control per module
- Up to 64 axes per Model 5300 system
- Position loop update times of 800µs / 2 axes (as fast as 500µs under software selection)
- Encoder feedback up to 17.5 MHz
- 64-bit floating point loop control
- 16-bit analog command (M3-40A only)
- 5 user assignable inputs / axis
- 5 user assignable outputs / axis
- High speed registration capture
- High speed PLS outputs
- 48 user variables per axis

1.3.2 Special M3-40 I/O Functions

- **16 HS Counters (10 MHz):** All five inputs as well as the A, B, and Z signal pins on each axis connector can be configured as high-speed counters.
- **Period Measurement (0.1 µsec accuracy):** Two pairs of inputs on each axis can be set up to measure the time between activation of the first and second input in the pair. Ideal for high-speed measurement and frequency measurement.

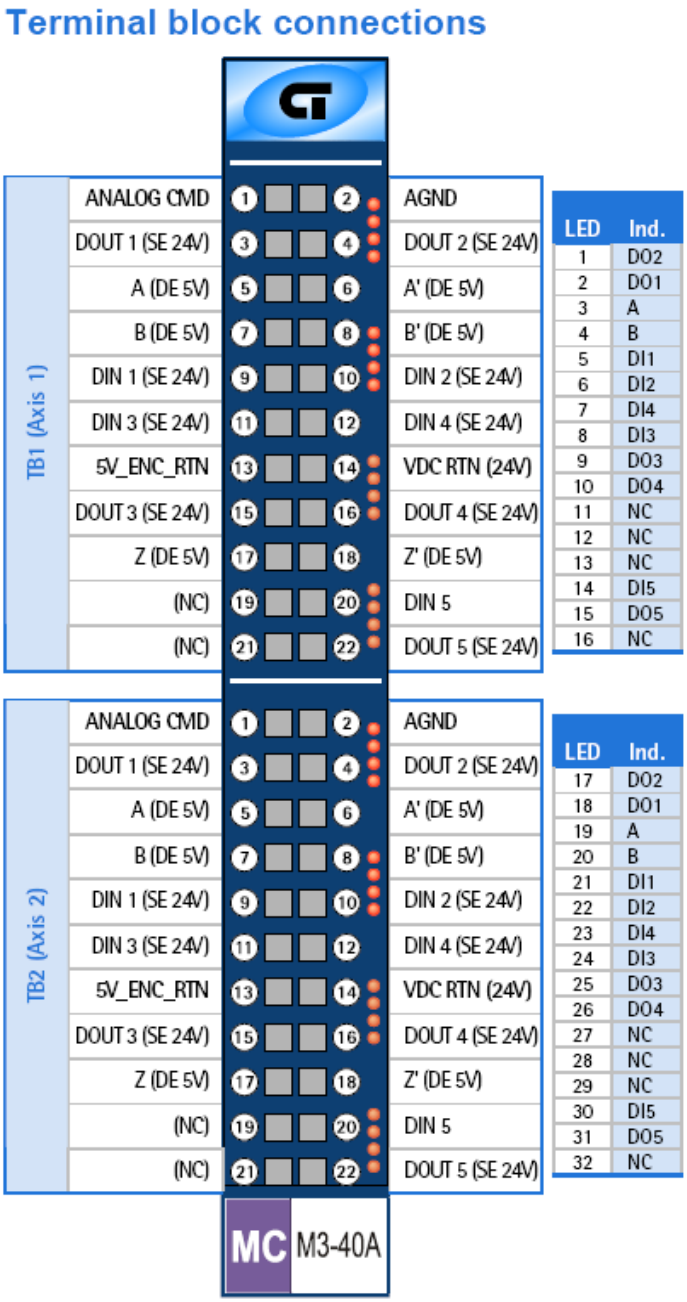
- **Frequency Outputs:** Three outputs on each axis can generate a programmable frequency up to 500 KHz.
- **Pulse Outputs:** All ten outputs can be pulsed for a programmable time value with an accuracy of 0.5 msec.
- **Programmable Limit Switch Outputs:** Three outputs on each axis can be configured to automatically turn on and off as a function of the encoder position. Up to sixteen on/off positions can be configured per axis. The on/off positions can be changed programmatically on-the-fly. This is especially useful to compensate for lead or lag time based on operating speed.

1.3.3 QuickBuilder Motion Control Features

- Axis objects configured in the Resource Manager
- New tuning wizard simplifies tuning
- Monitor motion parameters in multiple watch windows
- Use QuickScope to chart motion and I/O timing
- Simple English commands
- Over 100 new motion variables
- Full user-unit support
- Soft limits and hard limits
- Asynchronous event handlers

1.3.4 IO Assignments

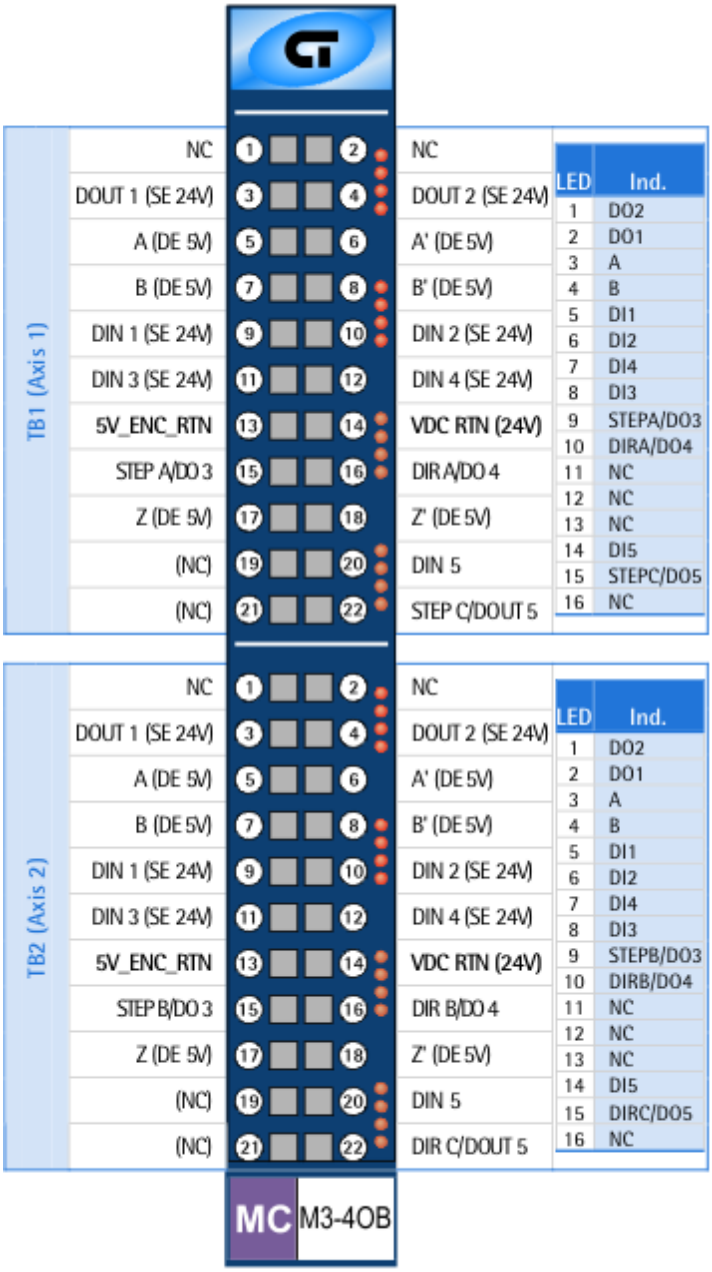
1.3.4.1 IO Assignments - M3-40A



⚠ Any two digital inputs can be configured in QuickBuilder to function as *registration inputs* 1 and 2. These digital inputs still function as general purpose inputs even when configured as *registration inputs.*

1.3.4.2 IO Assignments - M3-40B

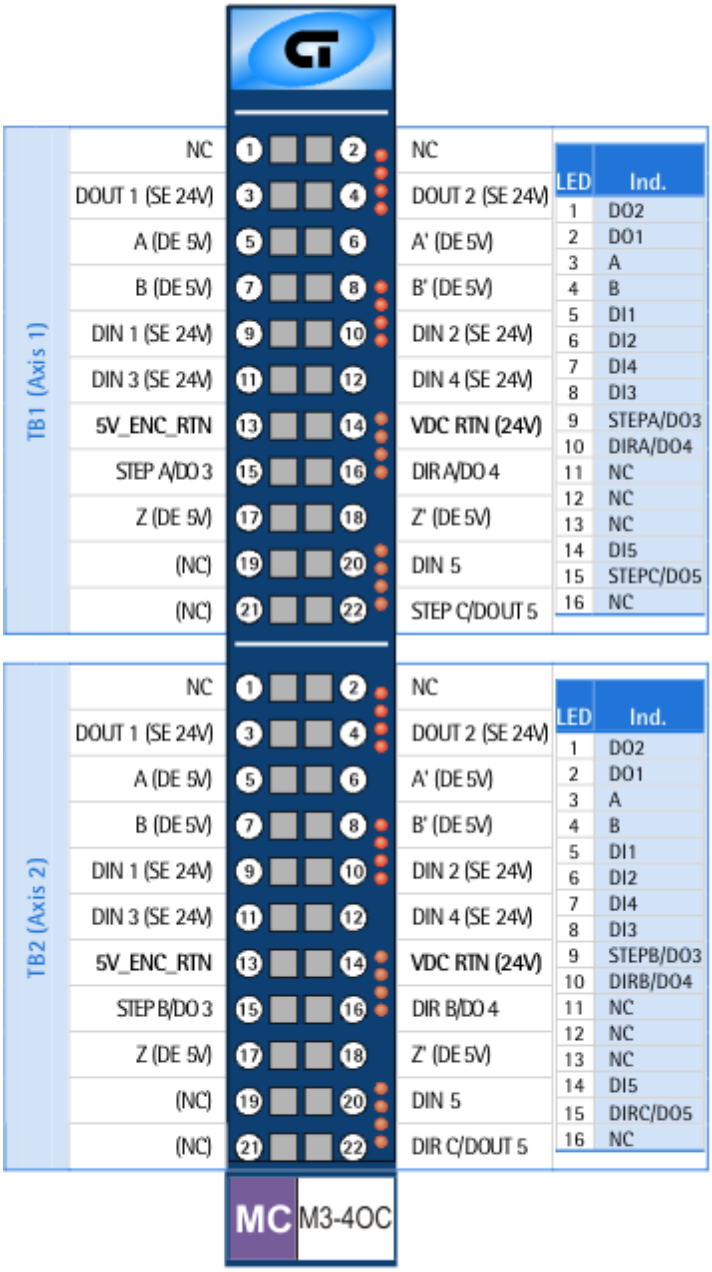
Terminal block connections



1. All step and dir connections are single-ended 24V.

1.3.4.3 IO Assignments - M3-40C

Terminal block connections



1. Step A/B and Dir A/B connections are single-ended 5V.
   Step/Dir C connections are single-ended 24V.

2 Chapter 2: Model 5300 Motion Architecture

The Model 5300 uses a powerful distributed architecture approach to solving machine control applications. The overall machine control program – called a QuickBuilder project – runs on the main CPU of the Model 5300 Automation Controller. It provides the primary guidance for the application and is in charge of communications with the outside world and the local Model 5300 I/O, motion, and specialty modules. The distributed nature of the Model 5300 design allows portions of the project to be passed to intelligent Model 5300 modules for local processing. This distribution of processing tasks and the overall coordination between modules and the main CPU is taken care of automatically by QuickBuilder.

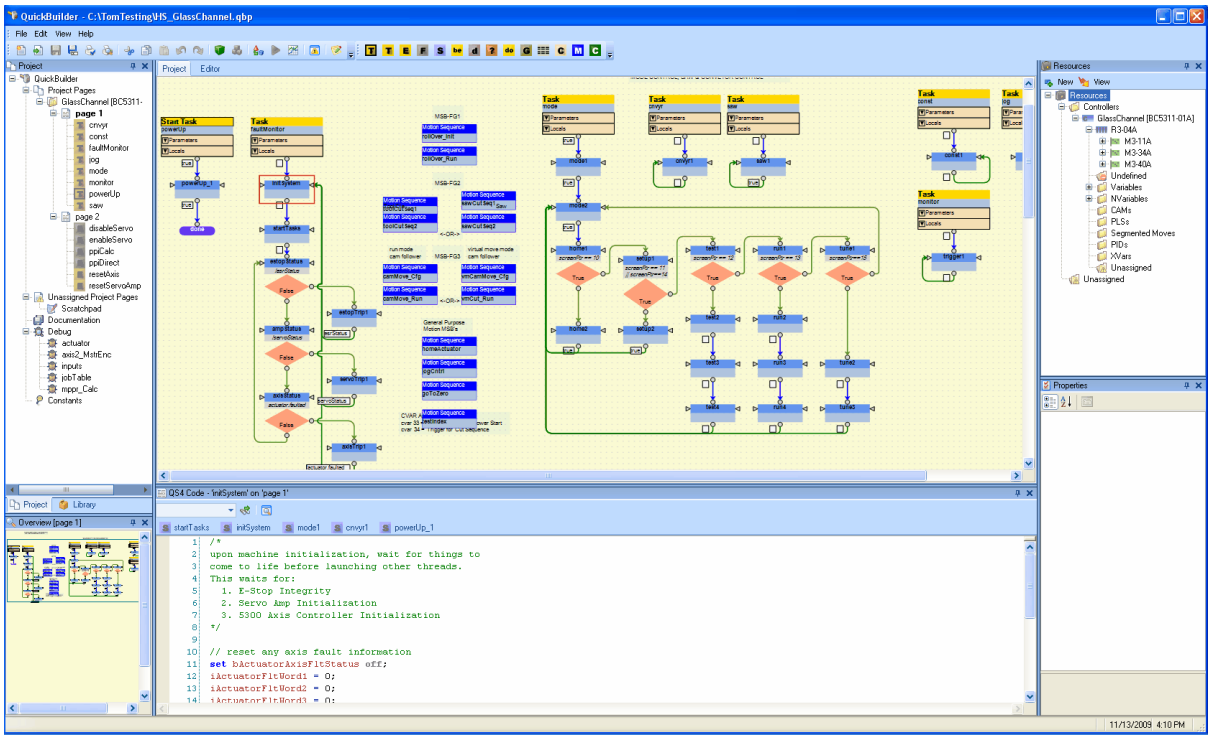
The result is a significant improvement in machine performance by off loading demanding processor-intensive functions like motion control tasks to specialized motion control processors on the Model 5300 Motion Modules. Even though this process takes place automatically, it's important for the automation engineer to have a basic understanding of the architecture of the Model 5300 controller and how it interacts with the QuickBuilder project.

Before we get into the details of how to add motion to a QuickBuilder project, we'll first review the major elements of the Model 5300's software architecture:

- QuickBuilder, the software application used to program Model 5300 controllers
- QuickStep, the programming language used in QuickBuilder
- QuickMotion, an extension to the QuickBuilder application that is tailored to handling motion control.

2.1 QuickBuilder

QuickBuilder is CTC's innovative graphical development environment built using the latest .NET technology, making it very intuitive to use. It combines all the aspects of an automation project into one easy to use desktop application. This holistic approach to solving automation projects leads to quicker machine startups and simpler understanding of even the most advanced automation tasks. The key to simplifying the automation process is to break the overall process down into the operating states of each of its elements.



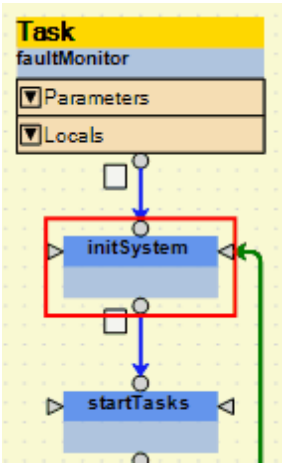
QuickBuilder desktop showing three tasks. A single step is highlighted in red.

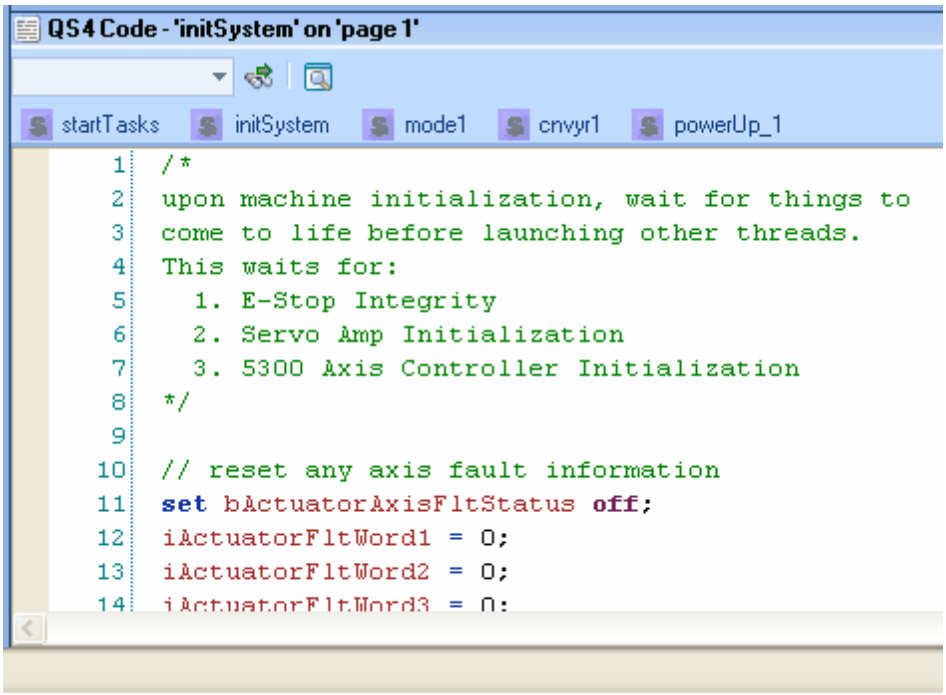
A QuickBuilder project is comprised of one or more tasks. Breaking the program into separate easily defined tasks greatly simplifies the programming process. A task contains multiple steps – where the steps represent a given operating state of the machine. Within the step are the actual instructions such as *wait for input*, *turn on output*, *move an actuator*, etc. It is also here at the instruction level that motion is initiated.

2.2 QuickStep

QuickStep is CTC's programming language used for the instructions within the steps. QuickStep was originally invented by CTC in the 1980's and has been proven in thousands of automation projects. Over the years CTC has continually refined and upgraded the language. The current version of QuickStep is QuickStep4 (QS4). The screen captures below show a highlighted step from the flow chart window that is automatically linked to the QS4 editor.

The use of QuickBuilder and QuickStep are covered in their respective manuals, and the user should be familiar with their use prior to starting a motion application.





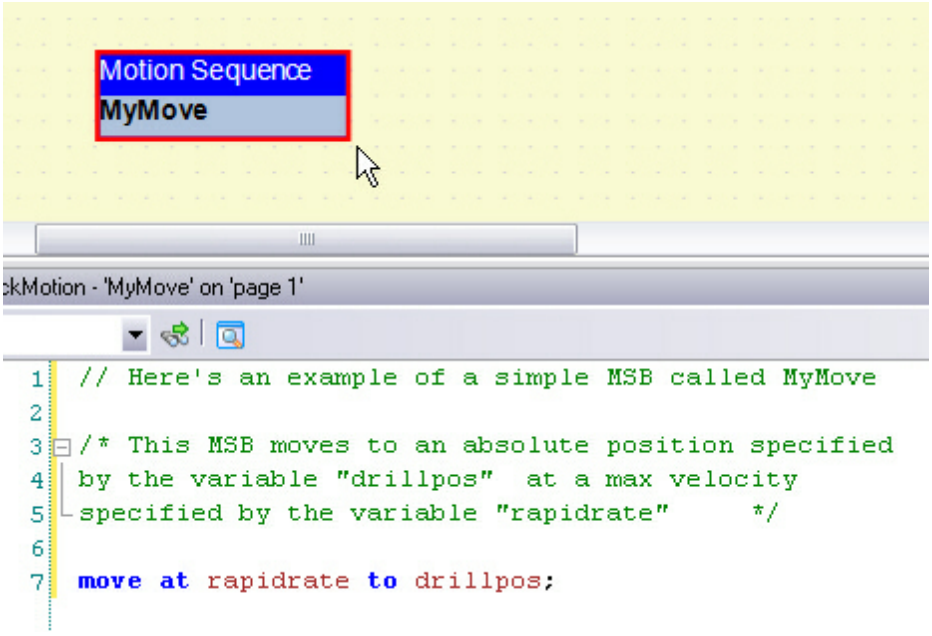
2.3 QuickMotion

QuickMotion is a specialized extension of QuickBuilder that is designed for motion control applications. It has been optimized to simplify the motion control process and to take advantage of the distributed architecture of Model 5300 automation controllers. QuickMotion instructions are entered into specialized tasks called Motion Sequence Blocks (MSBs).

The MSBs are coded within the QuickBuilder environment in the same way as steps are coded in QuickStep: Drag the MSB symbol onto the graphical desktop, give it a name, then use the editor to add the appropriate instructions. But there are two big differences:

1. an MSB is both a step and a task

2. a single MSB may be used by any number of axes.



By way of a practical example, think of the common motion control operation of homing an axis. In older control schemes, designers were either forced to write this homing code over and over in the program or call some generic homing routine hard coded by the motion control manufacturer. With QuickMotion, it is easy to create a customized homing MSB once, give it a name, and then use a QuickStep statement to start that MSB on any axis whenever an axis needs to be homed.

2.3.1 Adding Motion to the Blue Fusion 5300

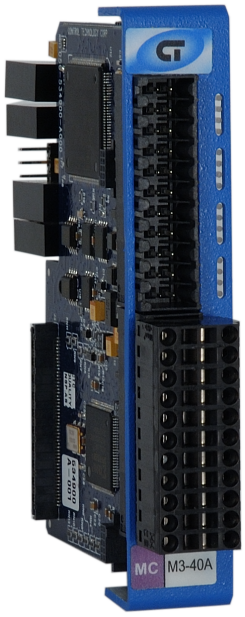
The main components used in Model 5300 motion control are:

- The Axis Module: The physical motion module in the rack
- The Axis Object: The QuickBuilder Resource representing an axis on that physical module.
- The MSB: The Motion Sequence Block, which contains one or more motion statements that execute on the Axis Module's CPU under the supervision of QuickStep on the main Model 5300 CPU.

2.3.1.1 The Axis Module

A Model 5300 axis module is inserted into the Model 5300 rack just like any other I/O module. CTC offers axis modules that can control one or more motion axes. Each motion module contains its own CPU and Motion Accelerator Chip (MAC), ensuring consistent high performance motion control regardless of the number of axes to be controlled.

M3-40A: Example of a Model 5300 Axis Module



2.3.1.2 The Axis Object

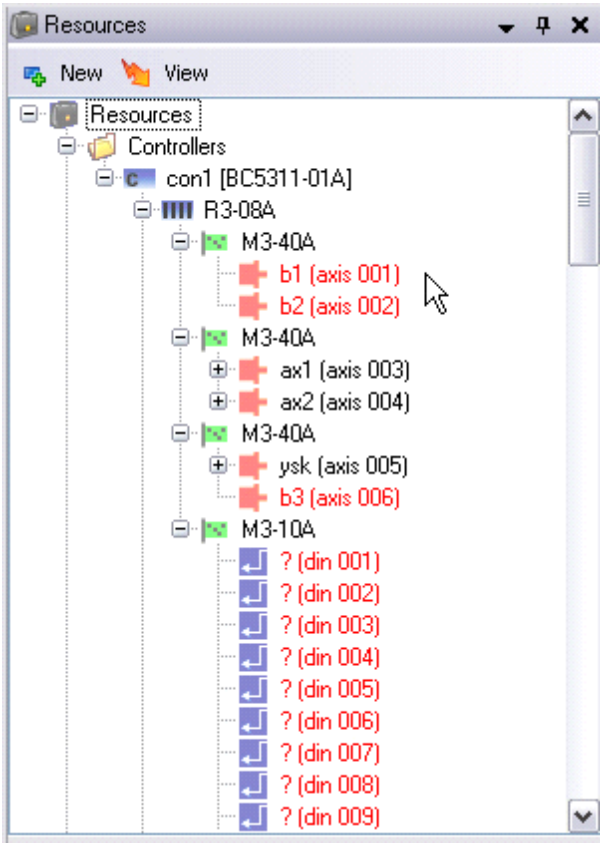
The *Axis* object represents a hardware-based axis associated with a servo or stepper drive. It is automatically created when a motion module is added to a rack in the Resource Manager. It typically consists of a controller module with various inputs and outputs that control the servo (or stepper) and usually feedback signals that are used to monitor position. Each axis can be commanded to perform some sequence of motion commands by the use of motion sequence blocks (MSBs). These MSBs appear in the QS4 program as stand-alone graphical elements.

Axis objects have many specialized properties that can be configured using the Property Inspector. Most of these properties can also be changed dynamically in the QuickBuilder project. Axis Objects have various inputs and outputs that control the servo (or stepper) and usually feedback signals that are used to monitor position.

When an MSB is selected, the programmed motion command sequence appears in the text editor window – the same window that is also used to edit QS4 code.

QuickStep4 can only start one motion sequence at a time for a given axis, but the active motion sequence can start other motion sequences (with some exceptions) that can run in parallel.

An MSB is not associated with any particular axis, which allows the same sequence to be reused many times for different axes.



2.3.1.3 The Motion Sequence Block

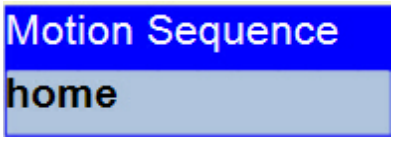
The Motion Sequence Block (MSB) element holds QuickMotion statement sequences. MSBs appear in the QuickBuilder project as stand-alone graphical elements. MSBs are not associated with any particular axis. This allows the same sequence to be reused many times for different axes, much like how a function works. MSBs are programmed using the QuickMotion language. An MSB may have only one QuickMotion statement, or it may have hundreds of statements.

The MSB is started on a given axis from QuickStep by using the *Start MSB* statement.

Once started, an MSB can start another MSB on its own that can run in parallel on the same axis. An MSB cannot start an MSB on another axis. This can only be done by QuickStep.

Up to 4 foreground MSBs can be running simultaneously. This limitation is imposed to guarantee high performance deterministic execution. A foreground MSB executes each of its statements at the loop update time of the Axis Module. This keeps them fast and in sync with the position loop.

In addition to the foreground MSBs, any number of background MSBs can be running simultaneously. The number of background MSBs is limited only by available memory on the Axis Module.



2.4 Controlling Motion from QuickStep

As mentioned earlier, QuickStep is in overall control of the project and as such, QuickStep has the ability to start and stop MSBs. There are actually only two Quickstep instructions pertaining to motion: *Start* and Stop.

- *Start*: Begins execution of the named motion sequence block (MSB) on the specified axis as a background MSB. This background MSB can then launch foreground MSBs on that axis. QuickStep can also directly launch foreground MSBs by using the FG option (start <axis> <msb> {optional FG/BG}, where FG is foreground and BG is background task.
- *Stop*: Stops execution of all foreground and background MSBs and thereby all motion.

In addition to these commands, QuickStep has extensive abilities to monitor and control MSBs on the axes while they are running via pre-defined and user-defined variables.

2.4.1 QS4 start Statement

This statement begins execution of the named motion sequence block (MSB) on the specified axis.

It is not an error to start another MSB when there is one already running for a given axis – however, if the named MSB is already running on a given axis, the start is effectively ignored.

```
start axis1 MSB1;       // start MSB1 on the axis called 'axis1', as
a background thread.

start axis1 MSB1 BG;    // start MSB1 on the axis called 'axis1', as
a background thread.

start axis1 MSB1 FG;    // start MSB1 on the axis called 'axis1', as
a foreground thread (run on each loop ticks, limited to 4).
```

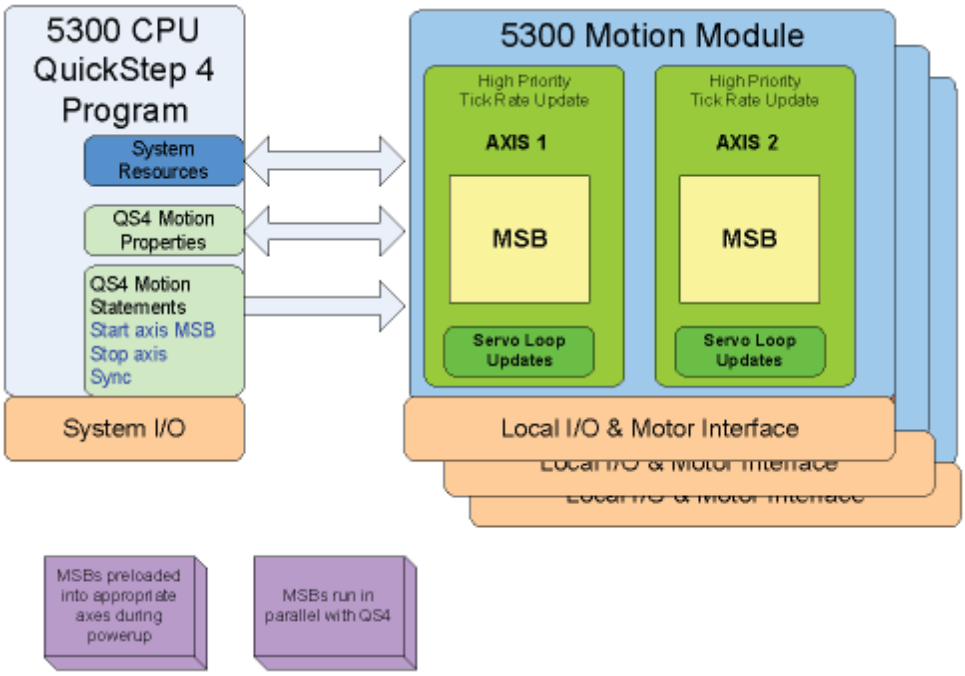
2.4.2 QS4 stop Statement

This statement stops execution of all MSBs on the named axis.

Example:

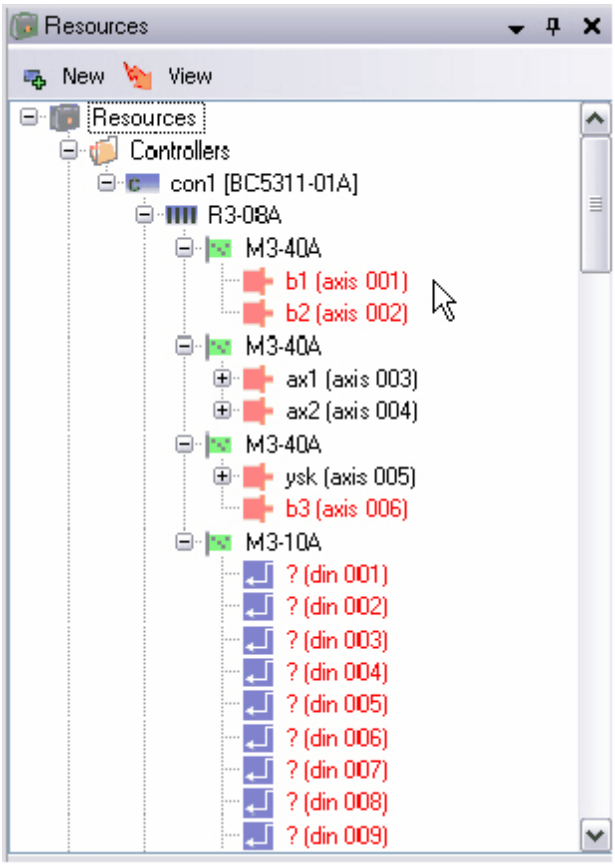
```
stop axis1;       // stop execution of all MSBs on 'axis1', this
stops immediately

stop axis1 slewed using 100; // stop execution of all MSBs on
'axis1', slewed stop at 100 user-units/sec/sec.
```

2.4.3 5300 Motion Architecture Summary Diagram



3 Chapter 3: QuickMotion Axis Setup

Adding a motion axis to a QuickBuilder project is very similar to adding any other resource. The first thing that needs to be done is to add the axis module to the appropriate rack in the controller. This is done by right clicking the rack and selecting the appropriate module. For this discussion we will be adding a third M3-40A module to our first 8-slot rack. As with other module types, axes are automatically numbered from left to right starting at the CPU. So in this case the two axes on the third module are numbered 5 and 6.

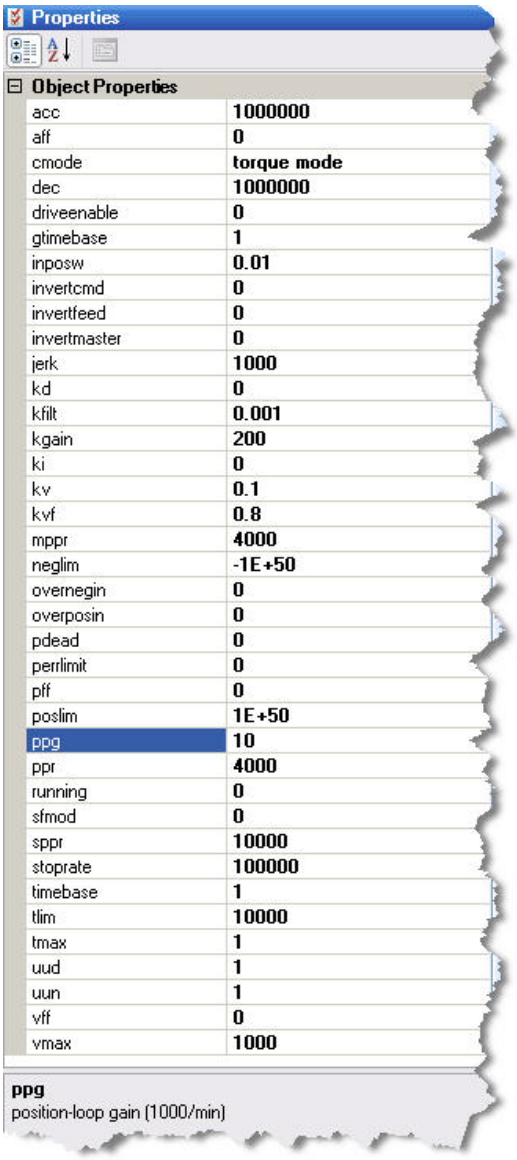


The axes first appear with question marks in their names, which must each be edited to a unique name. It is an error to have unnamed axes in a project. Right click and name the axis.

If the project changes, or the physical connection of the axes to the modules changes, axes can easily be rearranged in the Resource Manager. A single axis may be moved in the tree or a whole module can be moved as needed so that the named axes in the Resource Manager correspond to the actual wired axes.

After placing the *Axis* object in the proper place and naming it, the axis properties should be checked and updated as necessary. This is done in the property inspector window.

3.1 Axis Properties



When an *Axis* object is highlighted in QuickBuilder's Resource Manager, the following alphabetical property list for the axis is displayed. Required and Recommended properties to set up are reviewed below. Default values are given in [ ]. To learn more about these as well as the other properties, see the Variables Chapter later in this guide.

Required — When setting up an axis, the following properties must be set up in order for the Servo or Stepper Control module to properly interface with the connected motor and drive:

- **cmode:** Determines the command signal the controller sends out. Set to [Torque], Velocity, or Stepper.
- **tmax / vmax:** Depending on the drive type set, the maximum torque or velocity that will be realized by a 10V command from the controller. [1Nm / 1000RPM]
- **ppr:** The number of feedback counts per revolution [4000]
- **sppr:** When operating in stepper mode, this value must be set to correspond to the steps/rev of the controlled stepper drive.

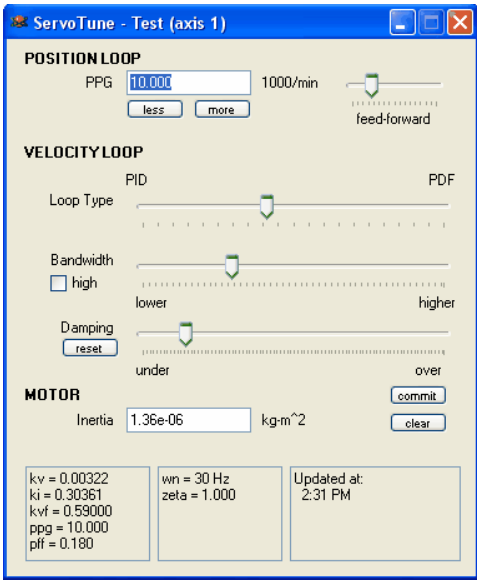
Recommended — Once the required properties have been entered, the axis can be tuned. However, it is recommended that the following properties also be checked and adjusted as necessary.

- **acc / dec:** Check that the acceleration / deceleration rates are appropriate. [10000000/10000000]
- **driveenable:** Set this to the output number that will be used to enable the servo drive. (Highly recommended that this be used. Use positive input number for true state=high; use negative number for true state=low.) [0=not used]

- **inposw:** The in-position window scaled in user units. This is used to determine when the drive has reached the commanded position. Use positive input number for true state=high; use negative number for true state=low. [0.01]
- **overnegin / overposin:** (Hardware over-travel limits) Set these to the input number to be used to signal positive and negative over-travel. Use positive input number for true state=high; use negative number for true state=low. [0=not used]
- **neglim / poslim:** (Software over-travel limits) Set these to the input number to be used to signal positive and negative overtravel. [-1E+50 / 1E+50]
- **perrlimit:** This is the maximum allowed following error in user units before a fault is generated. [0=disable checking]
- **uun/uud:** User-units numerator and denominator. This fraction is used to convert revolutions to user units. [1/1]

Other — Many of the other properties are either automatically adjusted by the tuning wizard or are used for more specialized functions. Refer to *Chapter 5: Variables* for more details.

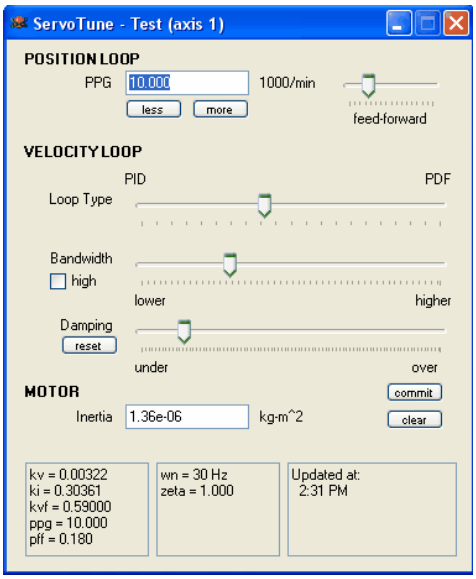
3.1.1 Basic Tuning



For basic tuning of an axis there is only one adjustment needed: adjust the Bandwidth slider until the desired performance is reached. Moving the slider to the right increases the servo loop bandwidth and hence the move performance. By checking the *high* box, the slider impact is doubled. If moved too far, the motor will become unstable and begin to emit a buzzing sound and vibration even with the motor at rest. If this occurs, move the slider back to the left until this condition is eliminated.

⚠ Note: Tuning parameters adjusted using the wizard are updated in volatile memory. To save them to the non-volatile memory of the controller it is necessary to download the project to the controller after tuning.

3.1.2 Fine Tuning



While the Basic Tuning method just discussed works well for most general purpose applications, higher performance applications or those with unusual loads or friction will typically require more adjustments. For best results in fine tuning an axis, it is useful to observe the velocity profile of the axis and how it responds to various adjustments to the tuning properties. This can be done by using QuickScope within QuickBuilder or by using an external oscilloscope to monitor the velocity output signal of the drive. The other wizard adjustment items are listed below:

PPG: This is the position loop proportional gain scaled in 1000/min units. This increases the response of the position loop and stiffness.

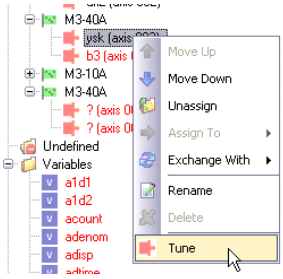
Feed-forward: This increases the position loop velocity feed-forward gain.

Loop Type: Adjust the loop type from 100% PID to 100% PDF structure.

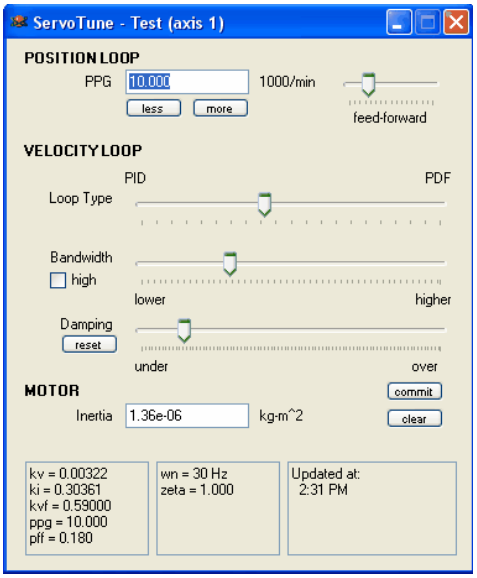
Damping: This has the effect similar to adding or removing friction from the system.

For advanced applications, all of these parameters with the exception of motor inertia can be changed programmatically or interactively through a QuickBuilder Watch Window. There are also several other tuning variables available for the experienced motion engineer. Refer to *Chapter 5: Variables* for details.

3.2 Tuning an axis

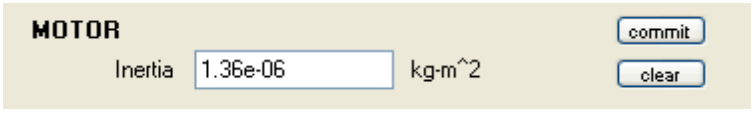


QuickBuilder simplifies the tuning process by utilization of an innovative new tuning wizard for each axis. To access the tuning wizard, simply right click on the axis and select *Tune*. Doing so will bring up a window like the one shown below. Note that each axis has its own tuning wizard window. Multiple windows may be active and displayed simultaneously.



To tune an axis with the wizard, the first step is to enter the motor inertia in the bottom box of the wizard. Once this is entered, the wizard is set up to critically damp the motor. Since the wizard adjusts tuning parameters in real time, the best way to use it is to set up a safe repeating move for the axis and then make adjustments in the wizard to optimize the motion profile.

Once tuning has been configured it may be save to the axis non-volatile memory but clicking on the 'commit' button. To remove tuning parameters from non-volatile motion board storage click the 'clear' button. By default the tuning parameters are saved with the QuickBuilder program and re-written each time the project is loaded. Committing the parameters to the motion board will override those in the program. This will set the nonvolatile axis variable to 1 when active.

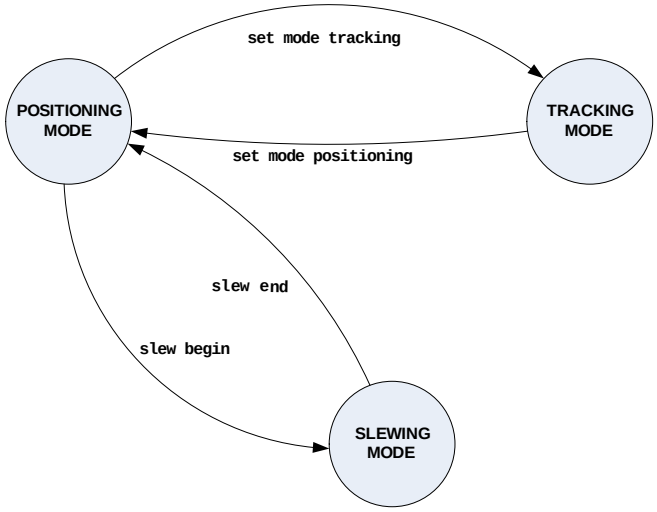


4 Chapter 4: QuickMotion Programming

This chapter covers the QuickMotion commands and their usage with MSBs.

⚠ These statements cannot be used in a QuickStep step! The MSB statement set has been created to simplify the motion programming process and make powerful motion control applications accessible to a wide range of users. These statements are optimized for high performance execution on the Model 5300 motion modules. In addition to the motion statement set, CTC has provided over 100 pre-defined motion variables that greatly simplify development. The motion related variables are covered within their own chapter, later in this guide.

4.1 Operating Modes



Positioning

In this mode, the axis is able to perform absolute and incremental time-based motion, including *SegmentedMoves* and time-based CAMs.

The axis must have completed any pending positioning operations before changing to a different operating mode.

Slewing

In this mode, the axis generates a series of interpolated positions based upon a constant (but alterable) velocity.

The axis must be stopped by using slew end in order to perform any positioning operations.

Tracking

The axis is able to perform position-tracking in this mode. This includes following, gearing and position-based

CAMs. The axis must complete all pending tracking operations before changing to a different operating mode.

4.2 Expressions

In QuickMotion, expressions consist of *variables*, *constants*, and *operators*. Variables are listed in *Chapter 6: Variables*.

The following *operators*, listed in order of grouped precedence, are available in QuickMotion:

```
( )    parenthesis
||     logical-or
&&     logical-and
|      bitwise-or
&      bitwise-and
!=     not-equal
==     equal
<=     less-than-or-equal
<      less-than
>      greater-than
>=     greater-than-or-equal
+      add
-      subtract
*      multiply
/      divide
%      modulo
!      logical-not
~      bitwise-not
-      negate
```

4.3 Utility Statements

Summary:

stop { slewed using *rate* **}**
drive enable
drive disable
delay *time* **ms**
variable = expression
zero feedback position
zero target position
zero following error
reset
if *condition* **then** *variable* **=** *expression*
wait until *condition*

Stop	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
stop { slewed using variable }				
parameters				
variable optional rate at which to stop the axis in user unit				

```
stop;              // stop the axis
stop slewed using rate;       // stop the axis by slewing to 0 at
specified rate
```

In positioning mode:

Non-slewed – This statement immediately aborts the present motion operation as well as halts the target position generator from updating the target position (*tpos*), thereby (eventually) stopping motion. This form of *stop* may not be desirable in all cases (such as when the axis has excessive following error), since the target position may be greatly different than the feedback position and the feedback position will still seek the target position.

Slewed – This statement first copies the current feedback position (*fpos*) into the target position (*tpos*) and then generates a controlled deceleration by *slewing* to zero velocity using the rate specified.

If the axis is in *slewing* mode, a *slew end* is issued thereby placing the axis in *positioning* mode. The optional stop mode *slewed* is ignored in this mode.

If the axis is in *tracking* mode, the numerator of the gear ratio is set to zero – but the axis remains in *tracking* mode. The optional stop mode *slewed* is ignored in this mode.

Enable/Disable Drive	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☐ FG MSB
syntax				
drive enable				
drive disable				

Enables or disables the drive associated with the axis, thereby allowing motion to occur. If the *driveenable* variable has been set to an output number, that output is automatically turned on when the drive enable command is encountered or turned off when the drive disable command is encountered. In some cases, the motor may slowly decelerate to a zero velocity when disabling.

```
drive enable;           // enable the drive for this axis
drive disable;          // disable the drive for this axis
```

⚠ Invoking the drive enable command sets the target position (*tpos*) to the feedback position (*fpos*).

Time Delay	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
delay *time* **ms**				
parameters				
`time`	an expression representing time in milliseconds			

This statement delays execution of the active MSB for the specified number of milliseconds.

```
delay 2500 ms;                  // delay for 2.5 seconds
```

Timeout Initialization	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
set timeout *ticks*				
parameters				
`ticks`	Number of ticks until a timeout occurs causing any active ‘on timeout’ event handlers to take action.			

This command initializes a private msb timer which is decremented on every tick if the ‘on timeout’ command is active. To disable execute an ‘on timeout ignore’ command. The timeout value must be set after every timeout, it acts as a down counter, invoking the event handler when 0 is reached. .

```
set timeout 100;  // Set timeout to 100 ticks
```

Assignment	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`variable = expression`				
parameters				
`variable`	a variable to change the value of			
`expression`	an expression			

The value of the specified expression is evaluated and stored to the named variable.

```
//calculate a new value for result
result = 34.857 * oldresult;
```

Zero Feedback Position	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`zero feedback position`				

Zeros the target position, but maintains following error (fposc = fposc - (ppr * tpos) then tpos = 0). Operates the same as *zero target position*.

```
//set the current position as zero
zero feedback position;
```

Zero Target Position	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`zero target position`				

Zeros the target position, but maintains following error. Operates the same as *zero feedback position*, but is more readable in stepper mode.

```
//set the current position as zero
zero target position;
```

Zero Following Error	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`zero following error`				

This statement zeros the feedback position (fpos/fposc) and target position (tpos), thereby removing any following error.

```
// relax the system by zeroing the following error
zero following error;
```

Unless you are current limiting and driving into a hard stop (or similar application), there is no reason to use "zero following error" (and it's probably wrong in most applications to use it). Zero position feedback is what should normally be used. Remember that following error is maintained when zeroing the position

feedback and 99.99% of the time that is what is desired. Think of it like this:

```
tpos = 1.000

fpos = 0.999
```

After "zero feedback position":

```
tpos = 0.000

fpos = -0.001
```

You don't want to lose that 0.001 of error, but you still want to call wherever you are zero — that is generally the case. Because tpos (the target to seek) runs the show, that is what you want to be precisely zero. All motion is relative/absolute to the target position, NOT the feedback position, as that wouldn't make sense.

Zero following error is used, for example, in nut-driving applications where one limits the torque, drives to an unreachable position (because as the nut is torqued, the torque limit is hit), and then watches for current limit and then zeroes the following error — thus, removing the torque, etc.

Reset Faults	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
reset				

Resets any fault (if possible to).

```
reset;      // reset axis faults
```

If/Assignment	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				

if *condition* **then** *variable* = *expression*	
parameters	
condition	a Boolean test condition
variable	a variable
expression	an expression

This statement evaluates the specified *condition*. If *true*, the *expression* is evaluated and *variable* is set to the resulting value. If *false*, MSB program flow continues at the next MSB statement.

```
// if the position error for the axis exceeds
// 0.25 set a variable 'fault' to 2
if perr > .25 then fault = 2;
```

Wait Until	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
wait until *condition*				
parameters				
condition	condition to test			

This statement waits for until the specified *condition* is *true*.

```
// wait here until chamber temp exceeds min
wait until temp > 32.849;
```

4.4 Program Flow Statements

Summary:

[*label*]
start *MSB mode*
end { and start *MSB mode* **}**
abort *MSB*
goto *label*
if *condition* **goto** *label*
on *asynchevent asynchhandler*

Statement Label	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
[*label***]**				

A label within an MSB is used as a marker for the destination of a *goto* or similar statement.

It is often required to *iterate* or *branch* depending on the state of some external input/output or internal condition – a label is used to mark the destination.

```
// this label is called Top
[Top]
```

Start	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
start *MSB mode*				
parameters				
MSB	the name of the MSB to start			
mode	**FG** start as a high-priority (tick) MSB **BG** start as a low-priority (non-tick) MSB			

This statement activates an MSB – if the MSB is already active, this statement is effectively ignored.

Up to 4 foreground (FG) MSBs may be running simultaneously.

There is no logical limit to the number of active background (BG) MSBs.

```
// start the MSB called PressCap and run as a foreground MSB
start PressCap FG;
```

End	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
end { and start *MSB mode* **}**				
parameters				
`MSB`	the name of the MSB to start			
`mode`	**FG** start as a high-priority (tick) MSB **BG** start as a low-priority (non-tick) MSB			

This statement ends execution of this MSB. An optional MSB can be specified to start after this one completes.

An *end* or *goto* statement should be the last statement in any MSB.

```
// this is the end of the MSB
end;

// end the current MSB and then start the MSB called WeldCap
// as a foreground MSB
end and start WeldCap FG;
```

Abort	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
abort *MSB*				
parameters				
`MSB`	the name of the MSB to abort (stop) the execution of			

This statement ends execution of another MSB. If the named MSB is not active, the statement is effectively ignored.

```
// kill only the WeldCap MSB
abort WeldCap;
```

Goto	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
goto *label*				
parameters				
`label`	the name of the label to branch to			

This statement changes program flow to the specified label.

```
// jump to the MSB label called Top
goto Top;
```

If/Goto	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
if *condition* **goto** *label*				
parameters				
`condition`	a Boolean test condition			
`label`	the name of the label to branch to			

This statement evaluates the specified *condition*. If *true*, MSB program flow continues at the specified *label*. If *false*, MSB program flow continues at the next MSB statement.

```
// If the axis's input1 is on goto the label MakeMove
if din1 goto MakeMove;
```

Asynchronous Event Handling	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
on *asynchevent asynchhandler*				
parameters				
`asynchevent`	One of the following: **riseof** *n* Rise of specified general purpose input. **fallof** *n* Fall of specified general purpose input. **hardfault** When a non-recoverable fault occurs. **capture** Capture of specified input trigger. **pls** *output* PLS output 1 to 5 activated. **timeout** 'timerticks' variable decrements to 0 (use 'set timeout' to initialize msb private value).			
`asynchhandler`	One of the following: **ignore** Cancel asynchronous event monitoring. **start** *MSB {FG/BG}{arm}* Starts the specified MSB in BG (background) mode unless FG is			

	specified, if capture then optional {arm} at end of statement. **goto** *label {arm}* Branch on event, if capture then optional {arm} at end of statement.

This statement controls asynchronous event handling.

If *asynchhandler* is set to *start…*, then an MSB is started automatically when the specified event occurs. If the *MSB* is already active when the event occurs, a second instance is *not* started. If not specified background mode is used (BG).

If *asynchhandler* is set to goto…, then a branch to that label occurs upon the event, within the same MSB.

If *asynchhandler* is set to cancel, then no operation will occur upon event. Each event is unique to a specific MSB although only one MSB may monitor a capture or specific pls output event.

If *asynchevent* is set to timeout then the 'set timeout <ticks>' command must be set for down counting to begin (500uS/tick). Branching based upon a timeout will occur regardless of motor operations and it is up to the MSB to properly recover and/or stop motors.

Example 'on timeout':

```
x = 0;
y = 0;
set timeout 5000 * 2;          // 5 second timeout
on timeout goto timedout;
[top]
// x will increment for 5 seconds and then a branch to [timedout] will occur
x = x + 1;
delay 100 ms;
goto top;
[timedout]
// y will increment after 5 seconds and continue forever
y = y+1;
delay 100 ms;
goto timedout;
```

4.5 Set Statements

Summary:

Set Loop Period	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
set loopperiod *value*				
parameters				
value	The desired loop time in uS, default value is .0008 (800uS). The minimum is 500uS.			

This statement sets motion interrupt loop period. The current loop period and rate are available via the axis 'loopperiod' and 'looprate' variables (looptime group). The period selected should be evenly divisible for accuracy. Thus .0005 has a rate of 2000 ticks/second, .0008 is 1250 ticks/second (1/.008). Setting one axis sets the other and it is recommended to only change the loop time at initialization, prior to the 'drive enable' command.

```
set loopperiod 800;     // Set loopperiod to the default, 800 uS
                        // (not needed since powerup default
```

Set Positioning Mode	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
set mode positioning				

Sets the operating mode of the axis to *positioning*.

```
set mode positioning;   // switch to positioning mode
```

Set Tracking Mode	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB

syntax

```
set mode tracking
```

Sets the operating mode of the axis to *tracking*.

```
set mode tracking;      // switch to tracking mode
```

Set/Offset Target/Feedback Position(s)	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB

syntax

```
set target position value
set feedback position value
set target position counts vcounts
set feedback position counts vcounts
offset position value
offset position counts vcounts
```

parameters

`value`	new or offset for the named position (user-units)
`vcounts`	new or offset for the named position (counts)

These statements modify the target and/or feedback positions. The new value (or offset) may be specified in user-units, or in feedback counts (by use of the keyword counts). The first two forms set the target or feedback position to a specific absolute value in user-units. The third and fourth forms set the target or feedback position to a specific absolute value in counts. The last two forms modify the target and feedback positions simultaneously by adding the specified offset to both.

⚠ Following error is maintained when these statements are executed.

⚠ The axis must not be active (i.e. actively generating a target position by use of a move statement) when any of these statements are executed.

```
// set the feedback position (fpos) to 2.149
set feedback position 2.149;

// offset both the target and feedback positions by 1100 counts
offset position counts 1100;
```

Set simulated feedback	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB

syntax

set simulated feedback *on/off*	
parameters	
on/off	off - normal operation, feedback from encoder. on - feedback simulated and from tposc on each servo loop.

Enables or disables simulated feedback, setting fposc to originate from the encoder (off) or tposc (on). 'tposc' is the incremental amount to move on the next servo loop. Thus when simulated the desired increment will be achieved on each loop. This command is useful for both test purposes and when using a virtual master. The simulated axis can publish its master position across the controller backplane, based upon its moves. See the 'Virtual Master' section. This command is also useful during open loop stepper operation when using the pls functionality.

```
set simulated feedback on; // this will cause fposc to = tposc after each loop period, drive must not be enabled
```

Set Master Encoder Source	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB

syntax		
set master *mode* { **using** *global* }		
parameters		
mode	**feedback1**	master position sourced from axis 1 feedback
	feedback2	master position sourced from axis 2 feedback
	target1	master position sourced from axis 1 target
	target2	master position sourced from axis 2 target
	feedbackZ	master position comes from axis 1&2 Z-inputs
	virtual	master position on this axis is to be calculated as a virtual source, reference 'move master at' for setup (master axis).
	common	master position from controller backplane as determined by variant register 36827 (slave axis)
global	**global**	(optional) This position information is made public to the controller backplane. Distributed to 'common' nodes as determined by variant register 36827.

This statement sets the source of the axis master encoder. The default source for MSBs executing on the first axis is *feedback2*. This means the first axis is using the second axis as the master. This command executes independently on each access thus to change axis 1 to be the master a 'set master feedback1' must be executed by MSB's on both axis. The default for the second axis is *feedback1*.

The source *feedbackZ* is derived by using the first axis' Z-channel input as the "A"-channel for the master encoder and the second axis' Z-channel input as the "B"-channel for the master encoder.

For an axis to make its master public the 'using global' option is used. This allows the axis to publish its position information to other axis that executes the 'set master common' command.

Variant register 36827 is used to define how global master information is distributed amongst slaves. The variant is a 4 row, 3 column array with the first 4 rows defining possible global master sources to reference and the columns referenced as follows:

[0] – enabled position information updates (every 4 mS to all slaves), set 1 to enable, 0 to disable.

[1] – master axis whose position information is to be distributed to slaves, 1 to N where N is all the axis in a controller rack. Note that the master axis MSB must have executed the 'set master global' command.

[2] – 32 bit field with each bit representing a slave axis to whom the master axis information is to be distributed. Bit 0 would be axis 1, Bit 31 is axis 32.

4.6 Common bits and variables

Summary:

set common bit *number state*
wait common bit *number state*
set common var *number value*
wait common var *number range*

Common bits and common vars are used to communicate state information:

a. between QuickMotion based modules
b. between QuickMotion and QuickStep 4
c. between axes on a single module such as an M3-40A

There are 256 common bits and 256 common vars. Common bits are Boolean, and common vars are bytes and therefore have values from 0 through 255.

The common bits are globally shared between all QuickMotion modules as well as QuickStep 4. Any changes made to common bits are "seen" by all QuickMotion modules and the main CPU running QuickStep 4.

The first 32 common vars are *overlaid* on top of the 256 common bits – changes made to a common var may alter up to 8 common bits.

The remaining 244 common vars are *module-local* – changes are only seen local to the module. This is useful to communicate state information between axes on a two-axis QuickMotion module such as an M3-40A.

A user may decide whether to use just common bits or just common vars or even a combination of the two depending on the application.

From QS4, common bits are accessed via the $CBITS[ ] system variable and common variables are accessed via the $CVARS[ ] system variable.

There are several QuickMotion instructions that deal with common bits and vars:

- *set common bit*
- *wait common bit*
- *set common var*
- *wait common var*

Within QuickMotion, common bits and vars can be used in expressions through the notation:

`cbit[n]` where n is 0 through 255

`cvar[n]` where n is 0 through 255

For example:

```
[top]
if cbit[10] goto op10;
if cbit[11] goto op11;
if cbit[12] goto op12;
goto top;

[op10]
move to 1.0;
wait for in position;
goto top;

[op11]
move to 0.0;
wait for in position;
goto top;

[op12]
move to 2.25;
wait for in position;
goto top;
```

Set Common Bit	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`set common bit` *`number state`*				
parameters				
`number`	bit number (0-255)			
`state`	**`true`** or **`false`**			

This statement sets the specified “common bit” to the given state.

Wait For Common Bit	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`wait common bit` *`number state`*				
parameters				
`number`	bit number (0-255)			
`state`	**`true`** or **`false`**			

This statement waits until the specified “common bit” is at the desired state.

Set Common Var	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`set common var` *`number value`*				
parameters				
`number`	variable number (0-255)			
`value`	an integer value (0-255)			

This statement sets the specified “common state variable” to the given value.

Wait For Common Var	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`wait common var` *`number range`*				
parameters				
`number`	variable number (0-255)			
`range`	`x`	a value of *x*		
	`x-y`	a value of *x* through *y* inclusive		
	`! x`	a value other than *x*		
	`! x-y`	a value outside the range of *x* through *y*		

This statement waits until the specified “common state variable” is within/outside the given range.

4.7 I/O Statements

Summary:

Set Output(s)	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
setout *outputlist*				
parameters				
outputlist	a comma delimited list of outputs to set			

This statement sets one or more outputs to the *on* state.

The output number can be 1-5 (dual axis mode) or 1-10 (1½ axis mode).

```
setout 2;          // turns on the second output on the module
setout 1, 3;       // turns on the first and third outputs
```

Clear Output(s)	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
clrout *outputlist*				
parameters				
outputlist	a comma delimited list of outputs to clear			

This statement sets one or more outputs to the *off* state.

The output number can be 1-5 (dual axis mode) or 1-10 (1½ axis mode).

```
clrout 2;          // turns off the second output on the module
```

```
clrout 1, 3;      // turns off the first and third outputs
```

Pulse Output	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
pulse *output* **for** *n*				
parameters				
output	the output to pulse 1-5 (dual axis mode) 1-10 (1½ axis mode)			
n	the time to pulse the output, an expression as milliseconds			

This statement causes the specified *output* to pulse for the specified duration. If the output is already *on* when this statement executes, the output state is unchanged – however it will be turned *off* after the specified time.

If another statement changes the state of the output to off before the allotted duration, the generation of the pulse is aborted.

The generated pulse is accurate within ½ of a millisecond.

```
// turns on the 2nd output on the module for 500ms
pulse 2 for 500;
```

PLS Define	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
pls *output* **using** *reference definitions*				
parameters				
output	the output (1-5) to control via a PLS			
reference	the encoder count scaled reference variable to compare to: **fposc** Feedback position of axis msb **mposc1 - mposc5** Master position counters #1 through #5 **mposc** Master position counter **smodc** Slave position (modulo) **smark** Slave marked position **tmc1 tmc2** Temporary master counters #1 & #2 **tsc1 tsc2** Temporary slave counters #1 & #2 **sdc** Slave decrement counter **fposc1** Feedback position of axis 1 (fposcA)			

	fposc2 Feedback position of axis 2 (fposcB) **tmodc** Temporary master counter mod mmc **sfposc** Secondary feedback position of axis
definitions	a comma-separated list of up to 16 PLS definitions: **on** x **to** y Turn output on when the reference is within the bounds specified by x through y (may be expressions)

The first statement defines or redefines a PLS (software-based programmable limit switch) associated with a given output. A definition over-writes the previous definition for an output (if one was defined already).

⚠ When a PLS is defined/re-defined it will be disabled and will not compute the state for the output. To enable a PLS after it is defined/re-defined, a *pls on* statement must be issued:

```
// define a PLS for output #1
//   output will be on when fposc is within 10-200 or 400-430
pls 1 using fposc on 10 to 200, on 400 to 430;

// enable the PLS for output #1
pls 1 on;
```

⚠ When using open loop stepper tposc is not available for PLS thus issue the command 'set simulated feedback on' to have tposc copied to fposc, on each control loop, allowing the use of this command.

PLS Enable/Disable	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
pls *output state*				
parameters				
output	the output to control via a PLS 1-5 (dual axis mode) 1-10 (1½ axis mode)			
state	**on** or **off**			

This statement enables (“on”) or disables (“off”) a PLS for an output.

On - Enables the pls functionality initialized for a particular output with the PLS Define statement.

Off – Disables the pls functionality initialized for a particular output with the PLS Define statement.

⚠ If the output is on when a PLS is disabled, it will remain on – unless the user re-enables the PLS (to re-compute the PLS output), or they *clrout* the output.

Wait For Input	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`wait for` *`transition`* **`of`** *`input`* `{` **`or`** *`condition`* `}`				
parameters				
`transition`	**`rise`** or **`fall`**			
`input`	the general purpose input to wait upon 1-5 (dual axis mode) 1-10 (1½ axis mode)			
`condition`	an optional exit condition			

This statement waits for the specified *transition* of the specified general purpose *input* to occur.

The MSB will not continue execution until the transition occurs – unless there was a *condition* specified and the condition evaluated to *true*.

```
// delay execution of MSB until input1 transitions from off to on
wait for rise of 1;
```

⚠ When this statement is used with the optional exit condition and the statement is part of a BG MSB, it is possible to miss transitions of the general purpose input. Therefore, the optional exit condition form should be used with care in BG MSBs.

Generate Pulses	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`generate output` *`output`* **`rate`** *`freq`*				
parameters				
`output`	1-10			
`freq`	the frequency (in Hz) to generate pulses; rounded to an integer			

This statement begins or ends generation of pulses using a specific output. If pulses are being generated on an output, then *setout*, *clrout* and *pulse output* commands given to the same output have the following behavior:

setout	no pulse generation occurs; the output will be active
clrout	pulse generation occurs for non-zero *freq*s
pulse output	no pulse generation occurs until the pulse output completes

When a frequency of 0 is specified, no pulse generation occurs. This effectively turns the output back into a general-purpose output.

The minimum frequency that can be generated is 1 Hz. The *maximum* frequency that can be generated is well over 500 kHz.

The accuracy of the generated signal varies by frequency (lower frequencies are more accurate). The following table summarizes the accuracy for several frequencies:

<100 Hz	+/- 0.001 Hz
500 Hz	+/- 0.005 Hz
1 kHz	+/- 0.02 Hz
2 kHz	+/- 0.1 Hz
5 kHz	+/- 0.5 Hz
10 kHz	+/- 2 Hz
20 kHz	+/- 8 Hz
50 kHz	+/- 50 Hz
100 kHz	+/- 200 Hz
250 kHz	+/- 1.5 kHz
500 kHz	+/- 5 kHz

⚠ Due to a hardware limitation, this statement is only usable with outputs 3, 4, 5 (Axis 1) and outputs 3, 4, 5 (Axis 2). The use of outputs other than those listed will be ignored.

⚠ The number of generated pulses cannot be controlled – only the frequency of the generated pulses.

Generate Steps	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`generate` *`n`* **`steps on`** *`pair`*				
parameters				
`n`	the number of steps to generate in 500 µsec			
`pair`	the step/direction pair to output steps on: 1. axis 1 step/direction pair (M3-40A/B/C outputs 3&4) 2. axis 2 step/direction pair (M3-40A/B/C outputs 3&4) 3. alternate step/direction pair (outputs 5 on each axis)			

This statement generates step and direction pulses on the specified step and direction pair.

If the expression *n* evaluates to a negative number, then the direction will be negative.

All of the pulses will be emitted in the next 500µs loop period.

⚠ Any *setout*, *pulse* or *generate output* used in parallel with this command will cause erroneous step/dir pulses to be emitted. One should not use these commands in conjunction with *generate steps*.

⚠ This command when used with *cmode* set to *stepper* mode will command additional pulses out the step/dir outputs.

Counter read, write, offset	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB

syntax

```
variable = ctr[n]
ctr[n] = expression
ctr[n] = offset
```

parameters

`n`	the counter number (0 through 7)
`variable`	the variable to store the current value of the counter to
`expression`	a new value for the counter
`offset`	an offset for the counter (subtracted from the current counter value)

These specialized forms of the *assignment* statement give read/write/offset access to the axis counters.

On the M3-40A, -40B, and -40C, there are 8 counters/axis that accumulate off-to-on transitions of the following:

ctr[0] digital input 1
ctr[1] digital input 2
ctr[2] digital input 3
ctr[3] digital input 4
ctr[4] digital input 5
ctr[5] 'A' channel input (non-quadrature)
ctr[6] 'B' channel input (non-quadrature)
ctr[7] 'Z' channel input (non-quadrature)

The first form of the statement stores the current counter value in a variable.

The second form of the statement changes the current counter value.

The third form of the statement offsets the current counter value.

The first and third forms are often used together:

```
totalcounts = 0;
```

```
[top]
// wait until input #1 rises
wait for rise of 1;
// get the current counter value
x = ctr[7];
// accumulate
totalcounts = totalcounts + x;
// offset so no counts are missed
ctr[7] -= x;
goto top;
```

Alternate Stepper Output	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
generate alternate *mode*				
parameters				
mode	**on** generate stepper outputs on alternate pins **off** generate stepper outputs on standard pins			

On a M3-40A/B/C, (and when in stepper mode in the case of the M3-40A), the step and direction outputs are normally output on axis (TBx) pin pairs (15, 16).

These cards also allow a third-axis to be controlled by temporarily outputting step and direction pulses on TB1 pin 22 (step) and TB2 pin 22 (direction).

To output on this alternate pair, the command *generate alternate on* should be issued. To output on the standard pair, the command *generate alternate off* should be issued.

⚠ One needs to be careful as only the destination of the step and direction signals change – the axis still believes that motion is being commanded on the primary axis (and thus updates its idea of where the absolute stepper position is). Therefore, it is good practice to zero the target position (*zero target position*) before switching to or from this alternate mode:

```
// move my axis 30 revs
zero target position;
generate alternate off;
move at 5 for 30 using 10,10;
wait for in position;

// move axis #3 20 revs
zero target position;
generate alternate on;
move at 10 for 20 using 10,10;
wait for in position;

// move me again 10 revs
zero target position;
generate alternate off;
move at 5 for 10 using 10,10;
```

```
wait for in position;
```

4.8 Simple Motion

Summary:

move to *position* **{ using** *acc, dec* **}**
move at *maxvelocity* **to** *position* **{ using** *acc, dec* **}**
move trap to *position* **using** *rate*
move in *time* **to** *position* **{mode** *n* **}**
move for *displacement* **{ using** *acc, dec* **}**
move at *maxvelocity* **for** *displacement* **{ using** *acc, dec* **}**
move trap for *displacement* **using** *rate*
move in time for displacement {mode n }
wait for in position
new endposition *position* **using** *rate*
new endposition relative *displacement* **using** *rate*
slew begin
slew at *velocity* **in** *time*
slew for *displacement*
slew end

Move Absolute, Triangular	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
move to *position* **{ using** *acc, dec* **}**				
parameters				
`position`	absolute end position, user-units			
`acc`	acceleration rate, user-units/sec/sec			
`dec`	deceleration rate, user-units/sec/sec			

This statement generates a triangular move to the specified end *position*. If the parameters *acc* and *dec* are omitted, then the default rates are used.

⚠ Linear acceleration and deceleration is used (as programmed in the axis *acc* and *dec* properties) unless the property *jerk_a_req/jerk_d_req* is set to a non-zero value in which case an S-curve type profile is generated.

Note: The specified *position* may also be specified as **ZPULSE_POS** or **ZPULSE_NEG**, meaning the next encoder Z-pulse in the positive or negative directions, respectively.

⚠ **ZPULSE_POS** or **ZPULSE_NEG** should only be used with absolute move commands.

```
/* Move to the absolute position specified by the variable
drillpos using default acceleration and deceleration rates. */

move to drillpos;
```

```
/* Move in the positive direction to the Z pulse using default
acceleration and deceleration rates. */

move to ZPULSE_POS;
```

Move Absolute, Speed-limited	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
move at *maxvelocity* **to** *position* **{ using** *acc*, *dec* **}**				
parameters				
`maxvelocity`	unsigned maximum velocity, user-units/sec			
`position`	absolute end position, user-units			
`acc`	acceleration rate, user-units/sec/sec			
`dec`	deceleration rate, user-units/sec/sec			

This statement generates a trapezoidal move to the specified end *position*. If it is not possible to reach the specified maximum velocity *maxvelocity*, then a triangular move is generated. If the parameters *acc* and *dec* are omitted, then the default rates are used.

⚠ Linear acceleration and deceleration is used (as programmed in the axis *acc* and *dec* properties) unless the property *jerk_a_req/jerk_d_req* is set to a non-zero value in which case an S-curve type profile is generated.

Note: The specified *position* may also be specified as **ZPULSE_POS** or **ZPULSE_NEG**, meaning the next encoder Z-pulse in the positive or negative directions, respectively.

⚠ **ZPULSE_POS** or **ZPULSE_NEG** should only be used with absolute move commands.

```
/* Move to the absolute position specified by the variable
drillpos using default acceleration and deceleration rates and
the rapidrate variable for a max velocity. */

move at rapidrate to drillpos;
```

Move Absolute, Trapezoidal	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
move trap to *position* **using** *rate*				
parameters				
`position`	absolute end position, user-units			
`rate`	acceleration/deceleration rate, user-units/sec/sec			

This statement generates a 1/3-1/3-1/3 trapezoidal move (1/3 of the time accelerating, 1/3 constant velocity, 1/3 decelerating) to the specified end *position*. The acceleration and deceleration rate must be specified.

⚠ Linear acceleration and deceleration is used (as programmed in the axis *acc* and *dec* properties) unless the property *jerk_a_req/jerk_d_req* is set to a non-zero value in which case an S-curve type profile is generated.

Note: The specified *position* may also be specified as **ZPULSE_POS** or **ZPULSE_NEG**, meaning the next encoder Z-pulse in the positive or negative directions, respectively.

⚠ **ZPULSE_POS** or **ZPULSE_NEG** should only be used with absolute move commands.

```
/* Move to the absolute position specified by the variable
drillpos using the variable rapidacc to set acceleration and
deceleration rates. The velocity used will be based on
the calculation to achieve a trap move. */

move trap to drillpos using rapidacc;
```

Move Absolute, Time-limited	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
move in *time* **to** *position* {**mode** *n* }				
parameters				
`time`	time, sec			
`position`	absolute end position, user-units			
`n`	acc / dec ramp multiplier			

This statement generates a 1/3-1/3-1/3 trapezoidal move to the specified end *position* in the specified *time*. The optional *mode* feature decreases the amount of time spent on acceleration and deceleration. The *n* parameter must be a positive, non-zero integer. By increasing the value of *n*, the acceleration and deceleration times are equally reduced, allowing more time at constant speed.

⚠ Linear acceleration and deceleration is used (as programmed in the axis *acc* and *dec* properties) unless the property *jerk_a_req/jerk_d_req* is set to a non-zero value in which case an S-curve type profile is generated.

Note: The specified *position* may also be specified as **ZPULSE_POS** or **ZPULSE_NEG**, meaning the next encoder Z-pulse in the positive or negative directions, respectively.

⚠ **ZPULSE_POS** or **ZPULSE_NEG** should only be used with absolute move commands.

```
/* Move to the absolute position specified by the variable
drillpos setting the calculated velocity, accel and decel rates
to make a trapezoidal move in the time specified by the variable
movetime. */
```

```
move in movetime to drillpos;
```

Move Incremental, Triangular	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
move for *displacement* **{ using** *acc, dec* **}**				
parameters				
`displacement`	incremental position, user-units			
`acc`	acceleration rate, user-units/sec/sec			
`dec`	deceleration rate, user-units/sec/sec			

This statement generates a triangular move for a specified *displacement*. If the parameters *acc* and *dec* are omitted, then the default rates are used.

⚠ Linear acceleration and deceleration is used (as programmed in the axis *acc* and *dec* properties) unless the property *jerk_a_req/jerk_d_req* is set to a non-zero value in which case an S-curve type profile is generated.

⚠ **ZPULSE_POS** or **ZPULSE_NEG** should not be used with incremental move commands.

```
/* Move an incremental distance specified by the variable
spanmove using default acceleration and deceleration rates */

move for spanmove;
```

Move Incremental, Speed-limited	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
move at *maxvelocity* **for** *displacement* **{ using** *acc, dec* **}**				
parameters				
`maxvelocity`	unsigned maximum velocity, user-units/sec			
`displacement`	incremental position, user-units			
`acc`	acceleration rate, user-units/sec/sec			
`dec`	deceleration rate, user-units/sec/sec			

This statement generates a trapezoidal move for a specified *displacement*. If it is not possible to reach the specified maximum velocity *maxvelocity*, then a triangular move is generated. If the parameters *acc* and *dec* are omitted, then the default rates are used.

⚠ Linear acceleration and deceleration is used (as programmed in the axis *acc* and *dec* properties) unless the property *jerk_a_req/jerk_d_req* is set to a non-zero value in which case an S-curve type profile is generated.

⚠ **ZPULSE_POS** or **ZPULSE_NEG** should not be used with incremental move commands.

```
/* Move an incremental distance specified by the variable
spanmove using default acceleration and deceleration rates and
using the variable slowspeed as a max velocity. */

move at slowspeed for spanmove;
```

Move Incremental, Trapezoidal	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
move trap for *displacement* **using** *rate*				
parameters				
displacement	incremental position, user-units			
rate	acceleration/deceleration rate, user-units/sec/sec			

This statement generates a 1/3-1/3-1/3 trapezoidal move (1/3 of the time accelerating, 1/3 constant velocity, 1/3 decelerating) for a specified *displacement*. The acceleration and deceleration *rate* must be specified.

⚠ Linear acceleration and deceleration is used (as programmed in the axis *acc* and *dec* properties) unless the property *jerk_a_req/jerk_d_req* is set to a non-zero value in which case an S-curve type profile is generated.

⚠ **ZPULSE_POS** or **ZPULSE_NEG** should not be used with incremental move commands.

```
/* Move the incremental distance specified by the variable offset
using the variable rapidacc to set acceleration and deceleration
rates. The velocity used will be based on the calculation to
achieve a trap move. */

move trap for offset using rapidacc;
```

Move Incremental, Time-limited	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
move in *time* **for** *displacement* {**mode** n }				
parameters				

`time`	time, sec
`displacement`	incremental position, user-units
`n`	acc / dec ramp multiplier

This statement generates a 1/3-1/3-1/3 trapezoidal move for a specified *displacement* in the specified *time*. The optional *mode* feature decreases the amount of time spent on acceleration and deceleration. The *n* parameter must be a positive, non-zero integer. By increasing the value of *n*, the acceleration and deceleration times are equally reduced, allowing more time at constant speed.

⚠ Linear acceleration and deceleration is used (as programmed in the axis *acc* and *dec* properties) unless the property *jerk_a_req/jerk_d_req* is set to a non-zero value in which case an S-curve type profile is generated.

⚠ **ZPULSE_POS** or **ZPULSE_NEG** should not be used with incremental move commands.

```
/* Move the incremental distance specified by the variable offset
setting the calculated velocity, accel and decel rates to make a
trapezoidal move in the time specified by the variable
movetime. */

move in movetime for offset;
```

Wait For In Position	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`wait for in position`				

This statement temporarily stops the execution of the active MSB until the target generator has reached its final value and the position error, perr is within the programmed in-position window.

```
// Move speed-limited
move at slowspeed for spanmove;

// Wait till motor is within the programmed in-position window
wait for in position;

// Turn on output 1 for 1 second
pulse 1 for 1000 ms;
```

Set New End Position	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`new endposition` *`position`* **`using`** *`rate`* **`new endposition relative`** *`displacement`* **`using`** *`rate`*				

This statement modifies the end position for the active *move* command. If there is no active *move*, then this statement is effectively ignored. The first form of this statement changes the end position to a new *absolute* position. The second form of this statement changes the end position *relative to the current position*. Using a *displacement* of 0 effectively stops motion *here* without generating a fault (unlike the *stop* command). Both statements require a *rate* to be specified. This *rate* is used as the acceleration/deceleration rate for the modified profile.

The *newvel* variable may be set to a nonzero value in order to specify a velocity. A trapezoidal move will be done whenever possible, if the end position does not allow for that then a triangular move will result. S-curve is not supported when using *newvel* although you may start out with an S-curve move and it will change to a trapezoidal or triangular with the new target and if *newvel* is nonzero, velocity.

⚠ Linear acceleration and deceleration is used (as programmed in the axis *acc* and *dec* properties) unless the property *jerk_a_req/jerk_d_req* is set to a non-zero value in which case an S-curve type profile is generated.

⚠ **ZPULSE_POS** or **ZPULSE_NEG** should not be used with this motion command.

Example 1: After din1 is activated change the end position to -3 mm.

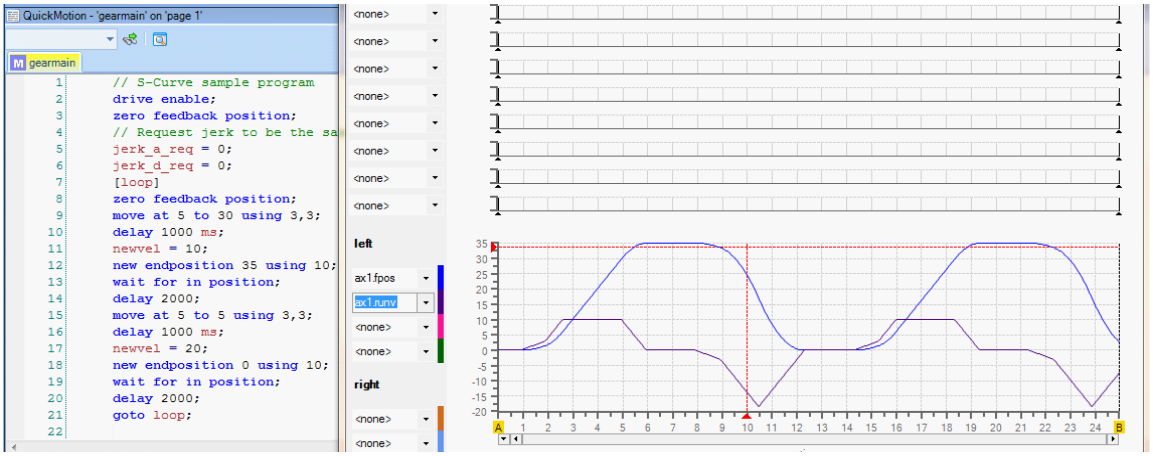
```
/* This example demonstrates how a move can be modified
on-the-fly by using the new endposition command */

[top]
zero feedback position;

// start moving to 25 mm
move at 5 to 25;

// if din1 is activated during the move, change the end
// position of the move to -3 mm
wait for rise of 1;
new endposition -3 using 10;

wait for in position;
delay 3000;
goto top;
```

Example 2: The move will be terminated 3mm after din1 is activated. Speed is only changed when it is time to decel to the new end position.

```
/* This example demonstrates how a move can be modified
on-the-fly by using the new endposition command.  */

[top]
zero feedback position;

move at 5 to 25;
wait for rise of 1;

new endposition relative 3 using 10;
wait for in position;

delay 3000;
```

```
goto top;
```

Example 3: Change target from 30 to 35, acceleration from 3 to 10 and velocity from 5 to 10 during the acceleration phase of the move.



Slew (begin)	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
slew begin				

This statement changes the operating mode of the axis to *slewing*.

```
slew begin;       // change from position mode to slew mode
```

Slew At	☐ Positioning	☑ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
slew at *velocity* **in** *time*				
parameters				
`velocity`	new slew velocity, user-units/sec			
`time`	time, sec			

This statement alters the current slew *velocity*. The velocity is changed smoothly over the specified *time*. For an immediate speed change, specify 0.0 for *time*.

```
// change from position mode to slew mode
slew begin;
```

```
// change from current speed to feedrate in 0.5 seconds
slew at feedrate in 0.5;
```

Slew For	☐ Positioning	☑ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
slew for *displacement*				
parameters				
displacement	ending relative slew position, user-units			

This statement alters the current slew velocity over time (to a slew velocity of 0.0) such that some *displacement* is consumed. If the current slew velocity is 0.0, then this statement is ignored.

The *displacement* should be unsigned, as the sign of the current slew velocity is used to sign the *displacement*.

```
// change from position mode to slew mode
slew begin;

// change from current speed to feedrate in 2 seconds
slew at feedrate in 2;

// delay execution of MSB until input3 transitions from off to on
wait for rise of 3;

// slew to a stop in the distance specified by the variable
// registrationoffset
slew for registrationoffset;
```

Slew (end)	☐ Positioning	☑ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
slew end				

This statement changes the operating mode of the axis to positioning. A zero-speed slew (in 0.0 *time*) is first generated if the axis is currently slewing at a non-zero velocity.

```
// change from position mode to slew mode
slew begin;

// change from current speed to slowjog in 0.5 seconds
slew at slowjog in 0.5;

// delay execution of MSB until input1 transitions from on to off
wait for fall of 1;
```

```
// stop motion and return to position mode
slew end;
```

4.9 Gearing

Summary:

gear at *numerator : denominator*
gear at *numerator : denominator* ***in*** *counts*
gear at *numerator : denominator* **in** *counts* **after** *acounts*
gear for *slavecounts* **in** *mastercounts*
gear for *slavecounts* **in** *mastercounts* **after** *acounts*
offset slave by *slavecounts* **in** *time*
wait master *counts*
wait slave *counts*
wait *source* **within** *start , end*
wait *source* **outside** *start , end*
zero *masslv* **counters**

Gear At	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
gear at *numerator* **:** *denominator*				
parameters				
`numerator`	new gear ratio numerator			
`denominator`	new gear ratio denominator			

This statement *instantaneously* changes the gear ratio of the slaved axis to the specified values.

Gear At In	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
gear at *numerator* **:** *denominator* **in** *counts*				
gear at *numerator* **:** *denominator* **in** *counts* **after** *acounts*				
parameters				
`numerator`	new gear ratio numerator			
`denominator`	new gear ratio denominator			
`counts`	counts of the master encoder			
`acounts`	counts of the master encoder to "wait for" before applying the gear/at...in...			

This statement changes the gear ratio of the slaved axis to the specified values over some number of master *counts*. An optional *after* condition can be applied to delay applciation of the gear/at/in.

Gear For In	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`gear for` *`slavecounts`* **`in`** *`mastercounts`*				
`gear for` *`slavecounts`* **`in`** *`mastercounts`* **`after`** *`acounts`*				
parameters				
`slavecounts`	counts of the axis encoder			
`mastercounts`	counts of the master encoder			
`acounts`	counts of the master encoder to "wait for" before applying the gear/for...in...			

This statement temporarily modifies the gear ratio of the slave axis such that a *slavecounts* correction (offset) occurs over a master-feedback displacement of *mastercounts*. The *slavecounts* correction may be positive or negative. An optional after condition can be applied to delay application of the gear/for/in.

Offset Slave Position	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`offset slave by` *`slavecounts`* **`in`** *`time`*				
parameters				
`slavecounts`	counts of the axis encoder			
`time`	time, sec			

This statement offsets the position (and therefore phase) of the axis such that a *slavecounts* correction (the offset) occurs over a period of *time*. The *slavecounts* correction may be positive or negative.

Wait for Counts of Master	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`wait master` *`counts`*				
parameters				
`counts`	counts of the master			

This statement waits until the specified number of master *encoder* counts has been generated.

Wait for Counts of Slave	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				

wait slave *counts*	
parameters	
counts	counts of the axis (slave)

This statement waits until the specified number of axis (slave, target-position) counts has been generated.

Wait Within	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
wait *source* **within** *start , end*				
parameters				
source	**master1, master2, master3, master4** or **slave**			
start	a modulo starting bound			
end	a modulo ending bound			

This statement waits for the *modulo position* (either *mposc1-4* or *sposc*) to lie within the specified bounds.

Wait Outside	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
wait *source* **outside** *start , end*				
parameters				
source	**master1, master2, master3, master4** or **slave**			
start	a modulo starting bound			
end	a modulo ending bound			

This statement waits for the *modulo position* (either *mposc1-4* or *sposc*) to lie outside the specified bounds.

Clear Temporary Gearing Counters	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
zero *masslv* **counters**				
parameters				
masslv	**master** clears tmc1 and tmc2 **slave** clears tsc1 and tsc2			

This statement atomically clears the temporary master or slave counters.

4.10 Position Capture & Registration

Summary:

set capture *transition* **of input** *input* { **gate input** *gateinput gatestate* }
set capwin range *start, end* **using** *reference* { *arm* }
wait capture { **if limit of** *limit* **goto** *limitlabel* }

Set Capture	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
set capture *transition* **of input** *input* { **gate input** *gateinput gatestate* }				
parameters				
`transition`	*rise*, *fall* or *edge* (any)			
`input`	the input# (1-10, representing all the inputs on the M3-40A card)			
`gateinput`	the input# (1-10, representing all the inputs on the M3-40A card)			
`gatestate`	**on** or **off**			

This statement initializes the parameters to be used for all captures on this axis, specifying the input (*capInput*) to use and the optional gated input. If gating is specified, then the specified gating input (*capGate*) must be at the specified gating state (*capGateState*).

The following variables are computed and available after a successful capture:

capposc	capture position in encoder counts
cappos	capture position in user units
capTriggered	flag set to 1 when capture occurs

Note: *capposc* and *cappos* are only valid when *capTriggered* is a 1. Once armed *capposc*/*cappos* will reflect the value latched when the capture input goes active but is not necessarily within the defined capture window. *capTriggered* verifies the capture window against the latched *capposc*/*cappos*, prior to setting.

If more than one running MSB on an M3-40A card arms the *same* input for capture, unexpected capture results may occur.

Only one input may be armed for capture at a time *per axis*. If another input is presently armed when this command is issued, the other input is effectively *disarmed*.

Set Capture Window	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
set capwin range *start, end* **using** *reference* { *arm* }				

parameters	
`start`	Start window position to compare against *reference*. *Reference* >= *start*.
`end`	End window position to compare against *reference*. If equals *start* then no window exists and capture will occur based on input. *Reference* <= *end*.
`reference`	the encoder count scaled reference variable to compare to: **fposc** feedback position **mposc1 - mposc5** master position counters #1 through #5 **mposc** master position counter **smodc** slave position (modulo) **smark** slave marked position **tmc1 tmc2** temporary master counters #1 & #2 **tsc1 tsc2** temporary slave counters #1 & #2 **sdc** slave decrement counter **fposc1** feedback position of axis 1 (fposcA) **fposc2** feedback position of axis 2 (fposcB) **tmodc** temporary master counter mod mmc **sfposc** secondary feedback position of axis
`arm`	If included will arm the capture, if not arm will need to be done by a Wait or On command.

This statement initializes a window to be monitored for valid captures to occur, anything outside this window is considered invalid and ignored. If the capture occurs outside this window it will automatically be re-armed within the loop period (default 800 uS). If 'arm' is specified this statement will automatically arm the capture prior to completing this instruction. The *capwinStart* variable is the start of range and the *capwinEnd* variable is the end of range, inclusive.

⚠ When using open loop stepper tposc is not available thus issue the command 'set simulated feedback on' to have tposc copied to fposc, on each control loop, allowing the use of this command.

Wait Capture	☐ Positioning	☐ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
`wait capture { if limit of` *`limit`* **`goto`** *`limitlabel`* **`}`**				
parameters				
`limit`	optional master encoder count limit			
`limitlabel`	optional label to branch to if `limit` is reached			

This statement waits for the capture and arms the capture input. If the capture occurs the next statement in the MSB is executed. A maximum limit of counts prior to exiting (*capLimit*/*capLimitflag*) can be set. This limit references the 'reference' set by 'set capwin' and the sign must be adjusted accordingly. The *capWait* variable will be set to 1 while the 'wait capture' is active, 0 if not.

If a limit (*capLimit*) is specified, then the statement will branch to the specified goto *limitlabel* after that number of master encoder counts has passed.

4.11 S-Curve

S-Curve support is optionally available for the move commands, from a stopped position. When using timed commands the distance, acceleration, and velocity will be calculated for the given time and then translated to an S-Curve move. The time will not be the same as the non S-Curve move but all other parameters will be, including position. Variables of interest are:

'runv' - velocity fed to the PID algorithm internal use only, read only.

'jerk_a/jerk_d' - acceleration/deceleration actual jerk, read only .

'jerk_a_req/jerk_d_req' – requested acceleration/deceleration jerk in units/sec3, read/write. Set to 1 for automatic calculation.

'sign' – nonzero for S-Curve move, 1 for CCW rotation and -1 for CW rotation, read only.

The minimum jerk that can be used is calculated by the formula ($a_{max} * a_{max}$)/v_{max}), applied independently to the requested acceleration and deceleration jerk. The maximum velocity is the same as the non S-Curve move and defined by the expression:

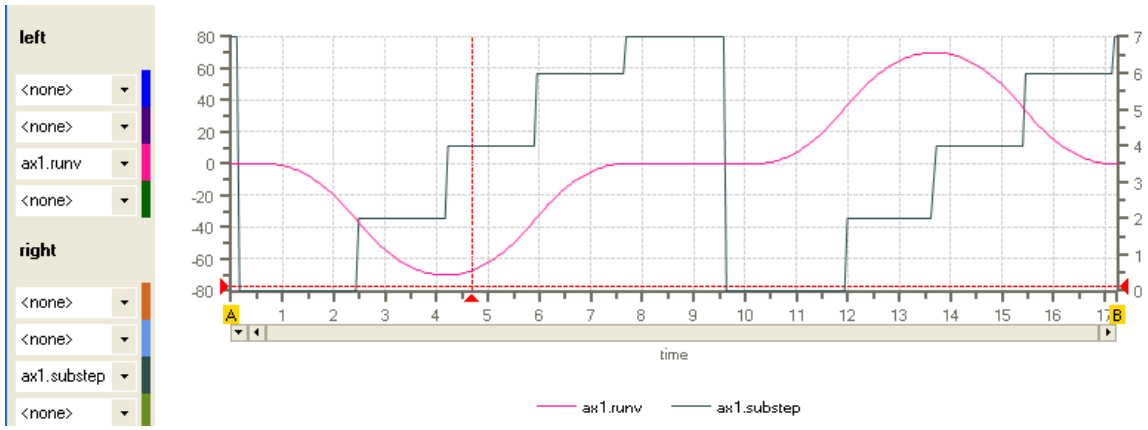
```
sign = 1;
if (delta < 0) {
   sign = -1;
   delta = -delta;
}
a_max = sign * acceleration;
d_max = -sign * deceleration;
V_max = sqrt(2.0 * a_max * d_max * sign * delta / (d_max - a_max));
```

If 'jerk_a_req/jerk_d_req' is 0 then a normal move will be attempted. If only one is set then 'jerk_a/jerk_d' will be set equal. If the requested jerk is greater than the minimum then it will be used. The variables 'jerk_a/jerk_d' are the actual jerk used for the move. Also note that S-Curve uses twice the acceleration and deceleration specified by the non S-Curve move request.

Below shows a sample S-Curve where the jerk is set to 1, thereby having the motion card calculate the optimum jerk and the velocity set to a large number so that the motion card will calculate the maximum velocity possible for the move. Note that the step graph is the segment (substep), 1 to 7, of the S-Curve, with 0 being segment 1. If for some reason the proper velocity or distance can not be attained by the parameters given, the non S-Curve curve move will be used. The end position can not be changed and a slew stop will do the non S-Curve stop. In the example below the motion card will calculate the maximum velocity and optimum jerk. Note there is little or no segment 4 (constant velocity). Also linear segments 2 and 6 are 0.

```
1  // S-Curve sample program
2  drive enable;
3  zero feedback position;
4  // Request jerk to be the same for acc/dec.
5  jerk_a_req = 1;
6  jerk_d_req = 1;
7  [loop]
8  move at 1000 for 246 using 20, 20;    // turn CCW
9  wait for in position;
10 delay 1000;
11 move at 1000 for -246 using 20, 20;   // turn CW
12 wait for in position;
13 delay 2000;
14 goto loop;
```

Resulting motion S-Curve using QuickScope:



⚠ 800uS is the default loop period. If 500uS is desired use the 'set loopperiod .0005' command prior to drive enable.

5 Chapter 5: Camming and Data Tables

Camming tables in QuickMotion are two-dimensional arrays of floating-point data. There are 6 tables available for use, numbered 0 through 5, each having up to 2000 rows and always 2 columns. These columns are named "x" and "y". Although their primary use is to hold data for *spline*- and *CAM*-based motion, they can be used to hold arbitrary data such as positions for recipe-based motion. Although limited to 6 tables, these tables can also be swapped out dynamically and refreshed with new data when loaded from the controller file system.

Spline tables use the "x" column as time and the "y" column as a *relative* position. *CAM* tables use the "x" column as a *relative master* position and the "y" column as a *relative slave* position.

Since *spline* and *CAM* tables use *relative* position data, the first point pair in these tables must be 0.0, 0.0 (time/master-position of 0, position/slave-position of 0). The exception to this is with CAM tables where the y component can be non-zero in newer firmware revisions, thereby establishing an offset. In addition, for any tables used for *spline* and *CAM* operations, all "x" values must be increasing, that is: a given row's "x" must be greater than the previous row's "x". Also, the minimum number of rows (pairs) in these tables is 3.

⚠ It is recommended that CAM tables and instructions be used whenever possible. Significant enhancements have been made to camming which have currently not been carried forward to splines. Some of this consists of the ability to start on non-zero y column values, ability to start anywhere within a table, and forward and reverse table traversing.

Points in a *spline* or *CAM* table are also referred to as *knots*, as they represent critical loci that must be passed through when interpolation occurs.

For example, in the following *spline* table:

0.0	0.0
1.5	2.0
2.0	2.5
3.0	3.0
4.0	2.0
5.0	0.0

there are 6 knots. Since this is a *spline* table, the last 5 knots are interpreted as follows:

At time = 1.5 seconds, the position of the axis should be 2.0 user-units beyond where the axis started this spline move.
At time = 2.0 seconds, the position of the axis should be 2.5 user-units beyond where the axis started this spline move.
At time = 3.0 seconds, the position of the axis should be 3.0 user-units beyond where the axis started this spline move.
At time = 4.0 seconds, the position of the axis should be 2.0 user-units beyond where the axis started this spline move.
At time = 5.0 seconds, the position of the axis should be back where the axis started this spline move.

The position of the axis between these "knots" is determined by the interpolation method specified by the QM code when the table is *started*.

The three available interpolation methods in QM for *spline* (and *CAM* tables) are:

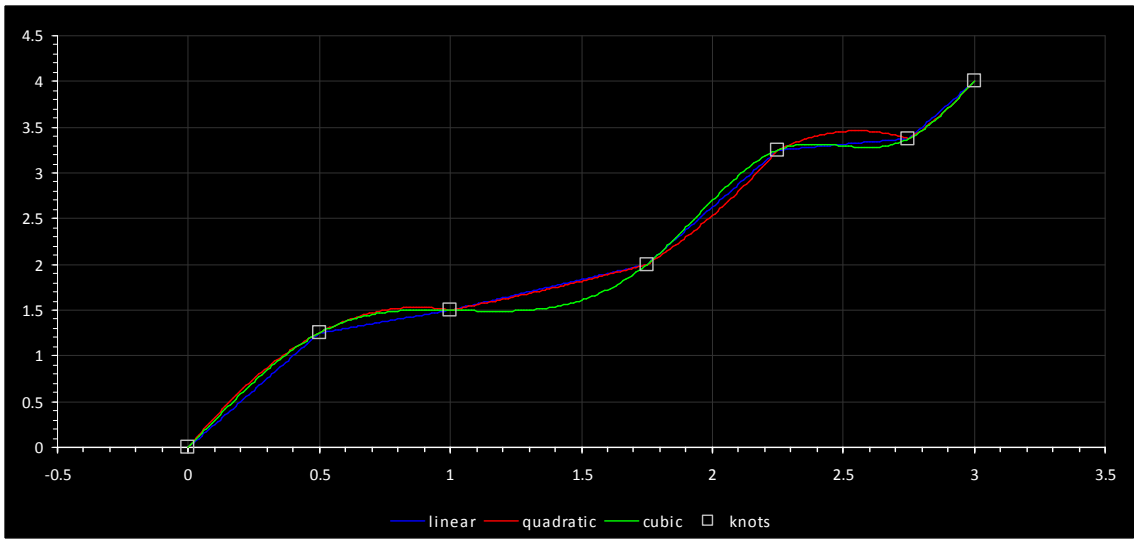
linear a straight-line joins each knot

quadratic a piecewise 2nd degree polynomial is fitted between this knot and the next; the first derivative of the first point is forced to 0.

cubic a piecewise 3rd degree polynomial is fitted between this knot and the next two knots; the first and second derivatives of the first point is forced to 0.

The following graph shows different interpolation methods using the following table of knots:

0.000	0.000
0.500	1.250
1.000	1.500
1.750	2.000
2.250	3.250
2.750	3.375
3.000	4.000



CAM tables are interpreted similar to their time-based *spline* counterparts. For example, in the following *CAM* table:

0.0	0.0
2.5	1.0
4.0	-1.0
5.0	0.0

the last 3 knots are interpreted as follows:

At a relative master position of 2.5 user-units, this (slave) axis should be 1.0 user-units beyond where it started.
At a relative master position of 4.0 user-units, this (slave) axis should be 1.0 user-units before where it started.
At a relative master position of 5.0 user-units, this (slave) axis should be where it started.

The master position is kept in the QM variable, `mpos` and is scaled to user-units by dividing by the axis parameter *`mppr`*. No other scaling occurs (i.e. *`uun`* and *`uud`* are not utilized). A raw (counts) variable is also available in *`mposc`*.

⚠ Unlike splines, Cam tables may start on a non-zero relative position (y). This position is used as an offset.

⚠ 'activeCAM_row' may be set to any desired row upon which mpos will be initialized to that which is the 'x' value of that row, allowing the table to start in that position.

⚠ 'invertmaster' variable is by default set to 0, meaning the cam table is traversed moving from row 0 to N. If 'invertmaster' is set to 1 the cam table position will begin at the end of the table and traverse N to 0. 'activeCAM_row' determines the start position, initialized by the precompute command either to the end or start of the table based upon 'invertmaster'. Prior to a 'table start' command 'activeCAM_row' can be changed to a different start position. 'invertmaster', when set causes mpos to decrement on positive master pulses, thus the reverse traversing of the table.

⚠ 'camming_invertend' variable is by default set to 0, meaning follow the logic described above for 'invertmaster'. If you wish to invert the logic of the 'invertmaster', with regards to camming table positioning only, set this flag. The 'invertmaster' variable will still control whether mpos is added or subtracted from based upon the master but 'caming_invertend', if set, allows you to start at the other end of the camming table. The direction you traverse the camming table is important since if you are at the start of the table and go slightly negative you will hold position but if you go past the end of the table the command will be considered completed.

invertmaster	camming_invertend	
0	0	master difference added to mpos, assume moving from beginning of camming table to end (thus go beyond end COMPLETE).
0	1	master difference added to mpos, assume moving from end of camming table to beginning (thus go beyond beginning COMPLETE).
1	0	master difference subtracted from mpos, assume moving from end of camming table to beginning (thus go beyond beginning COMPLETE).
1	1	master difference subtracted from mpos, assume moving from beginning of camming table to end (thus go beyond end COMPLETE).

5.1 Loading Tables

Summary:

table *n* **clear**
table *n* **addpair** *xexpression , yexpression*
table *n* **addseries** *pairs*
table *n* **copy from** *rowOffset1* **to table** *m rowOffset2 numRows*
table *n* **loadoffset** *rowOffsetFile, numPairs,rowOffsetTable*
table *n* **loadseries** *source fileNumber*

In order to use a table, it must be loaded with point pairs. There are several QM statements which facilitate loading of tables. These statements allow tables to be loaded either directly from within program code, thus static data, or dynamically from binary table files which reside on the controller file system. The commands that effect table loading are:

Table Clear	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
table *n* **clear**				
parameters				
n	the table to clear			

Clears a table of all of its points, thus setting the number of data points to 0, within a table.

```
table 1 clear;
```

There can be no active motion command when this statement is issued.

Table Add Pair	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
table *n* **addpair** *xexpression , yexpression*				
parameters				
n	the table to add a point pair to			
xexpression	an expression which when evaluated will be utilized as the value in the “x” column			
yexpression	an expression which when evaluated will be utilized as the value in the “y” column			

This statement adds a point pair to a table. This statement is used when the table is computed at MSB runtime since the pair is computed by two expressions.

```
table 2 addpair 3.75 + ztime, q + zoffset;
```

⚠ There can be no active motion command when this statement is issued.

⚠ An error will occur if there are already 2000 rows in the table.

Table Add Series	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
table *n* **addseries** *pairs*				
parameters				
n	the table to add a point pair to			
pairs	a series of one or more pairs (in the form of x,y), colon-delimited			

This statement adds constant point pairs to a table.

```
// add 4 point pairs to table 1

table 1 addseries 0.0,0.0 : 1.0,1.5 : 2.0,1.75 : 3.0,2.0;
```

⚠ There can be no active motion command when this statement is issued.

⚠ An error will occur if adding these pairs will result in a table with more than 2000 rows.

Table Copy	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
table *n* **copy from** *rowOffset1* **to table** *m rowOffset2 numRows*				
parameters				
n	The table which is source of the copy.			
rowOffset1	The source table row offset, 0 is no offset.			
m	The table which is destination of the copy.			
rowOffset2	The destination table row offset, 0 is no offset, -1 is append.			
numRows	The number of rows to copy, 0 is all.			

This statement allows for one table to be copy or appended to another table. The destination table does not need to exist. The offsets can be used to merge table data.

```
table 1 copy from 0 to table 2 0 0;     // Copy all of table 1 to 2
```

Table Loadoffset	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
table *n* **loadoffset** *rowOffsetFile, numPairs,rowOffsetTable*				
parameters				
n	The table to set the offset information on.			
rowOffsetFile	The file row offset to begin transfer on, 0 is no offset.			
numPairs	The number of cam file pairs to transfer, 0 is all.			
rowOffsetTable	The cam table row offset to begin storing file at, 0 is start.			

This statement works in conjunction with the 'loadseries' command, setting the offsets to be used. The offsets can be used to merge table data. This command only initializes parameters for 'loadseries' and does not directly effect the table.

```
table 1 loadoffset 0 0 0; // Default, transfer all from start .
```

Table Loadseries	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
table n **loadseries** *source fileNumber*				
parameters				
n	The table to load the cam file into.			
source	The location on disk where file will be found, 'flash' or 'ram' (/_system/Datatables or /RAMDISK/Datatables).			
fileNumber	The file to transfer, 'camtable#.tbl', where # is any valid positive number. A variable may be referenced as well.			

This statement requests a cam file to be transferred from the controllers file system. The file is transferred to the MSB for local storage and must be precomputed prior to operation. The 'loadoffset' parameters are referenced for this command as to where within the file and table to begin the transfer.

```
table 1 loadseries ram 1; // Load file 'camtable1.tbl'
```

The file format of 'camtable#.tbl' consists of a binary file of 32 bit float pairs with a file record structure as:

float rows[NUMROWS][2];

Where NUMROWS is the number of cam file pair entries, with the first starting at 0, 0. The same rules as the 'addseries' command exists. Each float is stored in little endian format with X being the first float.

As an example of a 3 row table with the values of:

0, 0
250, 25.67

500, 50.48

The binary data within a file would consist of 24 bytes, 4 bytes per entry in little endian and IEEE-754 floating point format. Below is byte representation of the required file, in hex:

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x7A 0x43 0x29 0x5C 0xCD 0x41
0x00 0x00 0xFA 0x43 0x85 0xEB 0x49 0x42

IEEE-754 conversion calculator is available at:

http://babbage.cs.qc.cuny.edu/IEEE-754/Decimal.html

For example, 50.48, is 0x4249EB85, reversed when stored in the file since in little endian format (low byte first).

5.2 Using Tables for Spline/CAM

Summary:

table *n* **continue**
table *n* **precompute**
table *n* **start** *imethod tscale , rpscale , repeatcount*
table *n* **start** *imethod* **cam** *mpscale , spscale , repeatcount*
stop table

Table Continue	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
table *n* **continue**				
parameters				
n	The table to continue that was previously stopped.			

Continues a cam table that was stopped by the 'stop table' command. Note that this command should only be used if the master position stopped at the beginning of the next table row position otherwise any row that is currently being executed during a 'stop table' (immediate stop) will be re-executed. The master position allows for a slewed stop. Upon execution of 'stop table', with the servo not moving, would then save all the camming information so that you can exit camming, jog into position, and then continue with the same camming table from where left off with the 'table *n* continue' command.

```
table 1 continue;
```

⚠There can be no active motion command when this statement is issued.

⚠ In many cases this command is no longer needed since the 'activeCAM_row' can be set prior to 'table start'. Only supported in camming mode, not spline.

Table Pre-compute for Spline/CAM	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☐ FG MSB
syntax				
table *n* **precompute**				
parameters				
n	the table to pre-compute			

This statement readies a table for use by a *spline/CAM* motion. After points have been added to a table, there are a series of computations that need to occur before the table can be utilized for spline and CAM motion operations. This statement causes those computations to occur.

There is no need to issue this command if a table is being utilized simply for data (i.e. for *tbln*, *tblx* or *tbly*

operations).

Failure to *precompute* a table before *starting* the table will cause a *hard fault*.

⚠ It takes roughly 250ms to *precompute* a 1000 row table.

⚠ The table must contain at least 3 points and all 'x' column values must be *increasing* or an error will occur. The first point in the table must be 0.0,0.0 otherwise an error will occur.

⚠ In camming mode 'invertmaster' set to 0 will cause 'precompute' to initialize the table to move from start to end, if set then from end to start, with positive master position motion.

```
// prepare the table for CAM use

table 1 precompute;
```

Table Start Spline Motion	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
table *n* **start** *imethod tscale , rpscale , repeatcount*				
parameters				
n	the table to utilize for motion			
imethod	**linear** uses linear interpolation **quadratic** uses 2nd order interpolation **cubic** uses 3rd order interpolation			
tscale	time scale factor: the values in the "x" (time) column in the table are effectively divided by this number			
rpscale	relative position scale factor: the values in the "y" (relative position) column in the table are effectively multiplied by this number			
repeatcount	the number of times to "run through" the table (0=forever)			

This statement starts *spline* motion using the specified table.

The current (target) position is used as the starting relative-position for the motion.

The table must be *ready* for use (i.e. a *table precompute* operation has been successfully completed on the table).

⚠ "In position" will be *true* only when the table has completed all required repeats. If 0 (forever) is specified for the `repeatcount`, then "In position" will never be true unless a `stop table` is issued.

⚠ The scale factor *`tscale`* must evaluate to a value greater than 0 otherwise an error will occur.

⚠ The scale factor *rpscale* must evaluate to a value other than 0 otherwise an error will occur.

⚠ The scale factor *rpscale* is not affected by uun or uud. The generated position is also unaffected by uun or uud.

```
[top]

zero feedback position;

table 1 start quadratic 1.0, 1.0, 0;  // run through the table
forever, 'table stop' command will cause the background motion
command to stop.
```

Table Start CAM Motion	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB

syntax

table *n* **start** *imethod* **cam** *mpscale* , *spscale* , *repeatcount*

parameters

n	the table to utilize for motion
imethod	**linear** uses linear interpolation **quadratic** uses 2nd order interpolation **cubic** uses 3rd order interpolation
mpscale	master-position scale factor: the values in the "x" (master-position) column in the table are effectively divided by this number
spscale	relative slave-position scale factor: the values in the "y" (relative slave position) column in the table are effectively multiplied by this number
repeatcount	the number of times to "run through" the table (0=forever)

This statement starts *CAM* motion using the specified table.

The master position (*mpos*, *mposc*) is not cleared when this statement is executed, the *activeCAM_row* will be used to offset into the cam table and that position will become *mpos/mposc*. *'table precompute'* will set the *activeCAM_row* to 0.

The current (target) position is used as the starting relative slave-position for the motion.

The table must be ready for use (i.e. a *table precompute* operation has been successfully completed on the table).

⚠ If the master position "*backs up*" past 0 (its initial position) and the "*repeats left to do*" counter is *greater than* 1, then the "*repeats left to do*" counter is decremented and the master-position wraps. If the repeatcount was specified as 0 (forever), then the master-position will always wrap.

⚠ "In position" will be *true* only when the table has completed all required repeats. If 0 (forever) is specified for the *repeatcount*, then "In position" will never be true unless a *stop table* is issued.

⚠ The scale factor `mpscale` must evaluate to a value greater than 0 otherwise an error will occur.

⚠ The scale factor `spscale` must evaluate to a value other than 0 otherwise an error will occur. The scale factor `rpscale` is not affected by uun or uud. The generated position is also unaffected by `uun` or `uud`.

```
[top]

zero feedback position;

table 1 start quadratic cam 1.0, 1.0, 1;
```

Table Stop	☑ Positioning	☐ Slewing	☐ Tracking	☑ BG MSB ☑ FG MSB
syntax				
stop table				

This statement stops *spline* or *CAM* motion. If in *CAM* motion then the current table state is saved in case a 'table continue' command is executed.

If there is no active `table start` (*spline*/*CAM*) motion, then this command is effectively ignored and no fault occurs.

Unlike other `stop` statements, `stop table` will never generate a *hard fault*.

```
wait until mpos > 9;

stop table;
```

5.3 Accessing Table Data

As mentioned earlier, tables may also be used to store/retrieve data. In QM, there are special array access operators called `tblx[]`, `tbly[]` and `tbln[]` that allow the user to retrieve information from a table.

`tblx[]` and `tbly[]` retrieve the "x" value (`tblx`) or "y" value (`tbly`) from a given row in a table. `tbln[]` retrieves the total number of rows in a table.

Their syntax is as follows:

```
tblx[table#, row]

tbly[table#, row]

tbln[table#]
```

⚠ Any attempt to read a value outside the bounds of the table will result in a value of 0.

They can be used in any QM expression, as show in the example QM code below:

```
// use table 1 for "move in time" pairs

// x will hold the move time (although it can hold anything we want)

// y holds an absolute position

// note that 'x' values in the table don't have to be

//   in increasing form as they do for spline/cam

//   since we are using the table just as data

// also note that there is no 'precompute' as the table is

//   just being used for data and not spline/cam

table 1 clear;

table 1 addseries 1.0,1.0 : 0.5,1.5 : 2.0,3.75 : 1.0,6.0;

[top]

zero feedback position;

// set index to 0 (indexes into tables are 0-based)

i = 0;
```

```
// grab how many pairs are in table 1

n = tbln[1];

[loop]

// done?

if i >= n goto top;

// grab data

move in tblx[1,i] to tbly[1,i];

wait for in position;

delay 1000;

// increment index

i = i + 1;

goto loop;
```

5.3.1 Diagnosing Table Issues

When table data is loaded by an MSB it can be difficult to determine if it is correct from a diagnostic viewpoint. Diagnostic variables exist that can be monitored to allow a user to walk through the table to visually or programmatically verify data from QuickBuilder. A Quickbuilder Debug Window can be used to view the Diagnostic Variables listed below:

Diagnostic Variables	Description	Type
debugTable	Cam table to view, from 0 to 5, representing table 1 to 6 since 0 based.	read-write
debugTableRows	Number of rows presently in the selected cam table, debugTable.	read-only
debugTableRow	Current row number to view in the selected cam table, debugTable.	read-write
debugTableX	X value for selected debugTableRow.	read-only
debugTableY	Y value for selected debugTableRow.	read-only

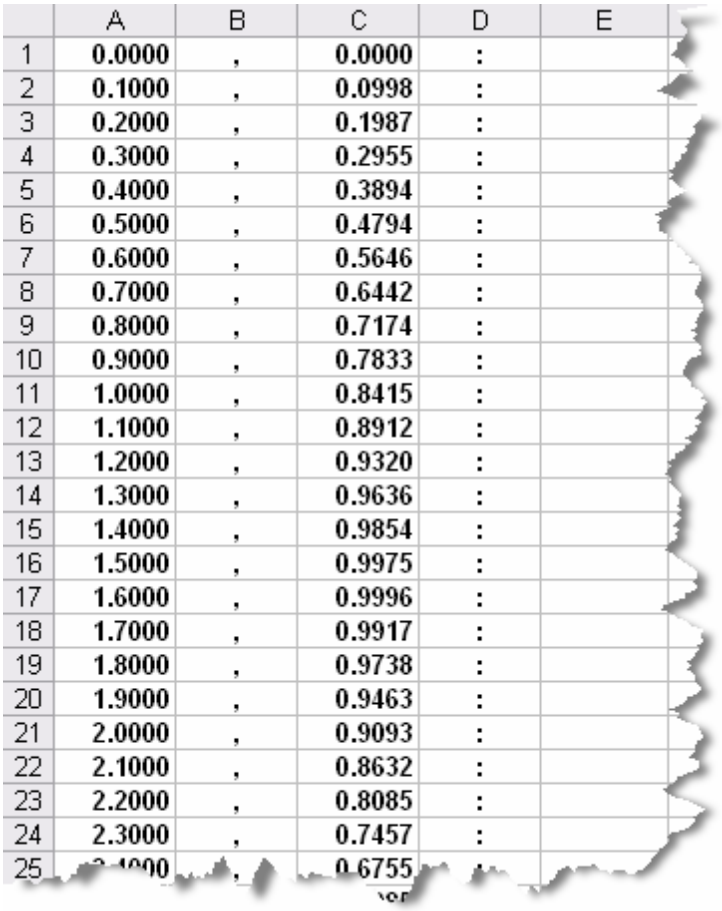
⚠ The above variable are only available from a QuickBuilder Debug Window, when executing an MSB use the

tblx, tbly, and tbln commands discussed in the previous section.

5.4 Microsoft Excel as Table Data

It is relatively simple to use data from Microsoft Excel as table data.

One can easily create four columns with *x-data*, a column containing a comma (“,”), *y-data* and lastly a column containing a colon (“:”) as follows:



	A	B	C	D	E
1	0.0000	,	0.0000	:	
2	0.1000	,	0.0998	:	
3	0.2000	,	0.1987	:	
4	0.3000	,	0.2955	:	
5	0.4000	,	0.3894	:	
6	0.5000	,	0.4794	:	
7	0.6000	,	0.5646	:	
8	0.7000	,	0.6442	:	
9	0.8000	,	0.7174	:	
10	0.9000	,	0.7833	:	
11	1.0000	,	0.8415	:	
12	1.1000	,	0.8912	:	
13	1.2000	,	0.9320	:	
14	1.3000	,	0.9636	:	
15	1.4000	,	0.9854	:	
16	1.5000	,	0.9975	:	
17	1.6000	,	0.9996	:	
18	1.7000	,	0.9917	:	
19	1.8000	,	0.9738	:	
20	1.9000	,	0.9463	:	
21	2.0000	,	0.9093	:	
22	2.1000	,	0.8632	:	
23	2.2000	,	0.8085	:	
24	2.3000	,	0.7457	:	
25	2.4000	,	0.6755	:	

Since the QuickBuilder editor allows free-form lines, the data can simply be copied and pasted into QM code in an MSB such as:

```
table 1 clear;

table 1 addseries

// paste Excel cells here
;
```

⚠ The last colon (“:”) in the last row will need to be removed using this method.

⚠ Refer to the loadseries command to dynamically load tables from a binary file stored on the controller's file system.

5.5 Virtual Master

At times a virtual master is required. This can be done in one of two ways:

1. Use the `move master` command to generate background pulses on the selected access based on timer loop tick counts. This allows the same access to operate normally, as an axis.

2. Setup a simulated axis, which runs as though it was receiving real encoder input, although it is not. The master position can then be published so others can track to it. Just about all motion commands are valid during a simulation, including s-curve.

Once a method of generating a master position is determined it can then be published across the backplane of the controller using variant 36827 (see Virtual Master Broadcasting).

Move Master Position	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
move master at *rate* **for** *limit* { **using** *ramp* }				
move master at *rate* **forever** *limit* { **using** *ramp* }				
parameters				
rate	The pulse rate of the encoder in pulses added per position loop period (800us default on the M3-40A).			
limit	The total number of pulses to generate.			
ramp	Optional ramp added or subtracted from rate at the position loop period (800us on the M4-40A).			

This statement virtually "moves" the master encoder by changing its "position" at the specified *rate* for a certain number of generated pulses (first form, specified by *limit*). If the *limit* is set to 0 then just the *rate* will change, dynamically, using any specified *ramp*, from the current rate.

To generate a continuous stream of pulses, use the second form. *move_master_ramp* and *move_master_rate* variables can be referenced to check current settings of virtual master.

Reference **`set master`** *`mode { `* **`using`** *`global }`* `for additional information.`

Example:

```
// stop the virtual axis
move master at 0 forever using 1;
// set the virtual access global
set master virtual using global;    // this will place the master information in dual ported memory for broadcast to slave axis
// start the virtual axis
move master at 100000 forever using 1;
```

One of the drawbacks of the 'move master at' command is that it is based on the loop period and not user units. This can make things difficult to program. The benefit is the axis is available for motion. If an axis can be reserved and dedicated as a master then a simulated feedback can be used. In essence the axis becomes a virtual axis, responding to commands as it would on a real axis:

```
set simulated feedback on; // this will cause fposc to = tposc after each loop period, drive must not be enabled
```

5.5.1 Broadcasting

When the 'set master virtual using global' command is given, mposc delta counts (mposc - last_mposc) are placed in a dual ported memory for broadcast by the controller, across the backplane. This allows muliple axis to follow a master. Up to 4 masters are currently support with a broadcast update rate of 4ms. Prior to broadcasting, the controller must be initialized with the proper master/slave information

A special variant table located at 36827 supplies the interface to control virtual broadcasting. In Quickbuilder define a table of type 'int' with an override of 36827.

Once created the variable table columns are defined as follows where masterNum is 0 to 3 (up to 4 masters):

MasterArray[masterNum][0] - 0 disables broadcasting, non-zero enables.

MasterArray[masterNum][1] - Master axis number, 1 to 32.

MasterArray[masterNum][2] - Slave axis bit positions, up to 32 supported (16 2 axis cards). Which slaves to replicate master information to.

Example:

```
MasterArray[0][0] = 0;  // disable any running master

MasterArray[0][1] = 1;  // Axis number 1 will be a master
```

```
MasterArray[0][2] = 14;  // Axis 2, 3, and 4 are slaves referencing master:  0x0000000E
```

5.6 Segmented Moves and Examples

Summary:

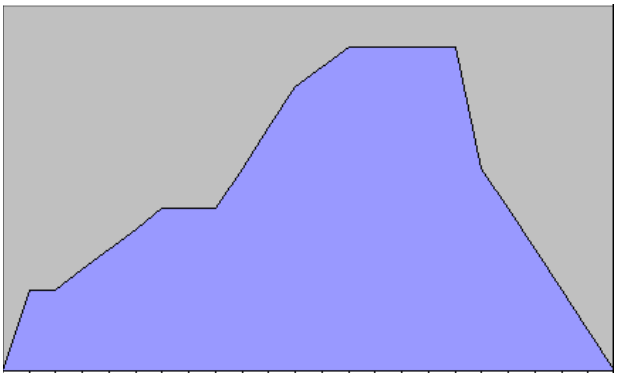
segmove table clear
segmove table accdec to vel using rate
segmove table accdec to vel for displacement
segmove table slew until position
segmove table stop at position using rate
segmove table start relative

Topics:

- Concept
- Commands
- Examples

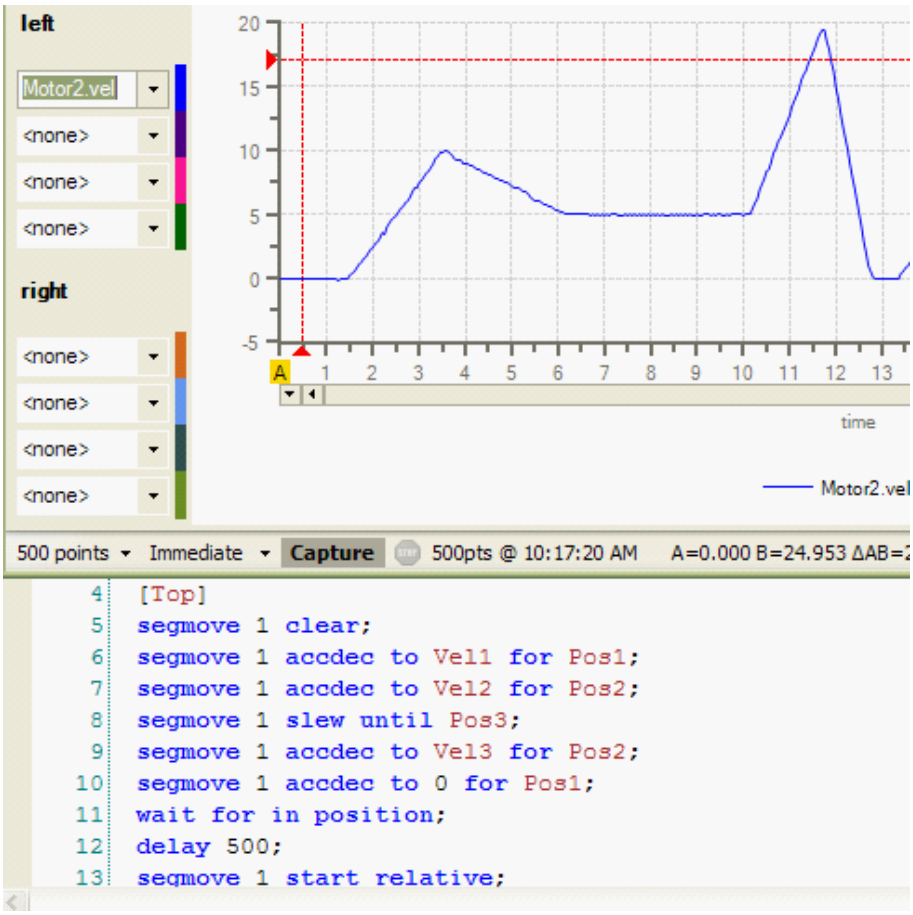
5.6.1 Concept

A segmented move is a precompiled move with multiple distances, acceleration, and velocities tied together. Below is the velocity profile of an example segmented move.



Up to 16 Segmented Move “tables” can be defined with up to 20 segments each residing within them. Once a segment table has been defined and then started, you can redefine that same table while it runs without affecting the segment table in progress.

Below is an example of a segmented move with 5 segments using table 1. You define each acceleration or deceleration ramp and each constant velocity ramp as a separate segment.



5.6.2 Commands

Creating and running a table is easy and uses the following procedure:

1. Clear the Table.
2. Create up to 20 acc/dec and constant velocity segments.
3. Start the Table.

Clear Segmented Move Table	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
segmove *table* **clear**				
parameters				
table	what table to clear: 1 to 16			

This command clears any existing table information
Example: **segmove** 1 **clear**;

Add Segmented Move to Table at rate	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB

				☑ FG MSB
syntax				
segmove *table* **accdec to** *vel* **using** *rate*				
parameters				
table	Which table to add to: 1 to 16			
vel	Velocity, user-units/sec			
rate	Acceleration/deceleration rate, user-units/sec/sec			

This command adds an acc/dec segment from the current velocity to the new <vel> at the specified <rate>.
Example: `segmove 1 accdec to Vel1 using Acc1;`

Add Segmented Move to Table over displacement	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
segmove *table* **accdec to** *vel* **for** *displacement*				
parameters				
table	Which table to add to: 1 to 16			
vel	Velocity, user-units/sec			
displacement	Incremental position, user-units			

This command add an acc/dec segment from the current velocity to the new <vel> over some <displacement>. Note this is an incremental acc/dec segment.
Example: `segmove 1 accdec to Vel2 for Pos2;`

Add Segmented Move to Table (slew)	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
segmove *table* **slew until** *position*				
parameters				
table	Which table to add to: 1 to 16			
position	Velocity, user-units/sec			

This command adds a constant velocity segment until reaching some specified <position>. This is an absolute position from the start of the profile. Prior segments in table must represent movement before this command is accepted, otherwise a fault will occur as table is built
Example:
`segmove 1 slew until Pos3;`

Add Segmented Move to Table (stop)	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB

				☑ FG MSB
syntax				
segmove *table* **stop at** *position* **using** *rate*				
parameters				
table	Which table to add to: 1 to 16			
position	Position to stop at, user-units			
rate	Acceleration/deceleration rate, user-units/sec/sec			

This command stops motion at the specified *position*, with a given *rate*. This will cause motion to stop at an absolute position at a specified deceleration rate. Prior segments in table must represent movement before this command is accepted, otherwise a fault will occur as table is built.
Example:

```
segmove 1 stop at Position using Accel;
```

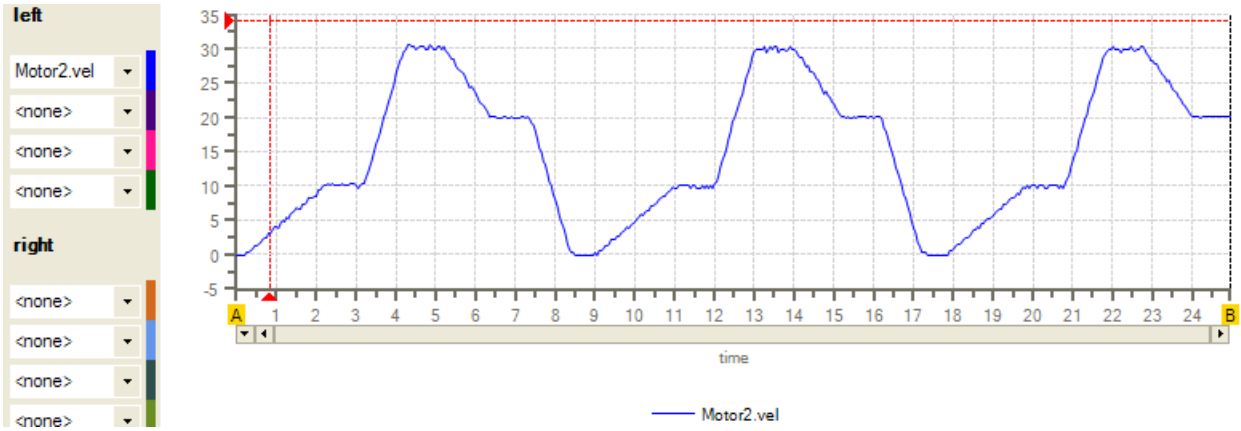
Add Segmented Move to Table (relative)	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
segmove *table* **start relative**				
parameters				
table	Which table to add to: 1 to 16			

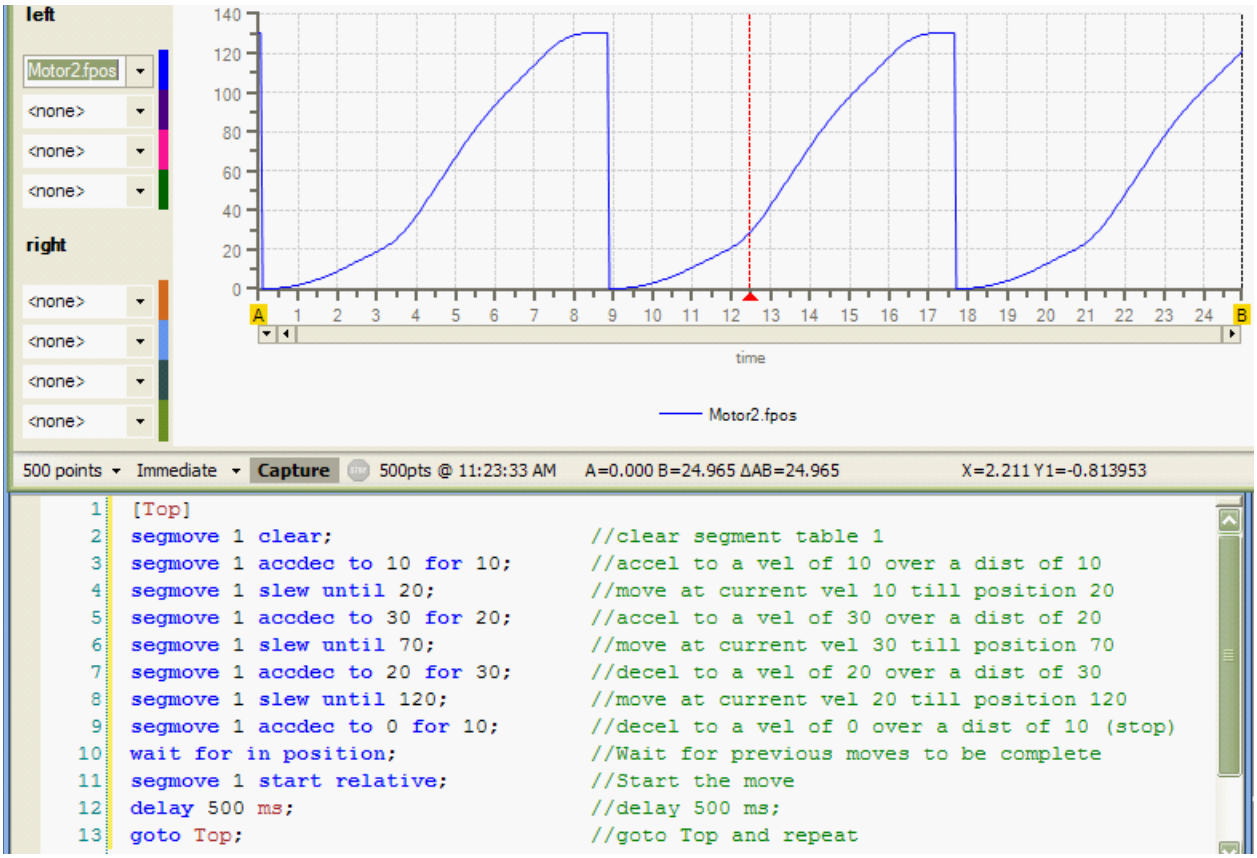
This command starts a relative segmented move – a "zero feedback position" occurs automatically upon executing this command.
Example:

```
segmove 1 start relative;
```

5.6.3 Examples

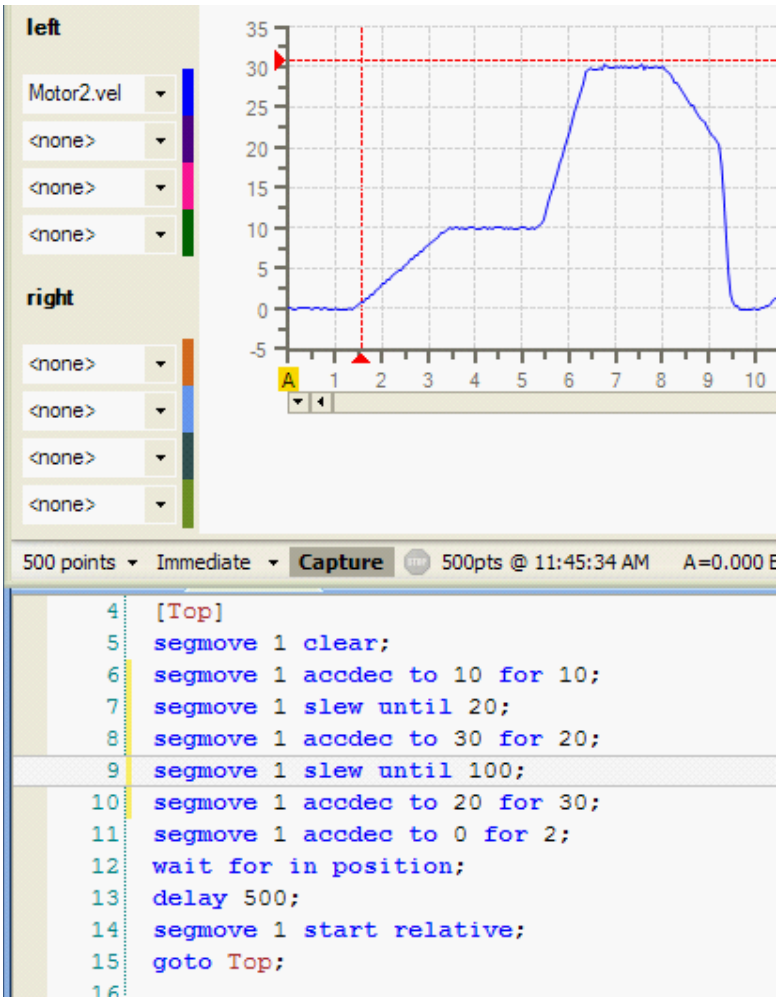
The following pages include screen shots of move profiles and the code used to create them.



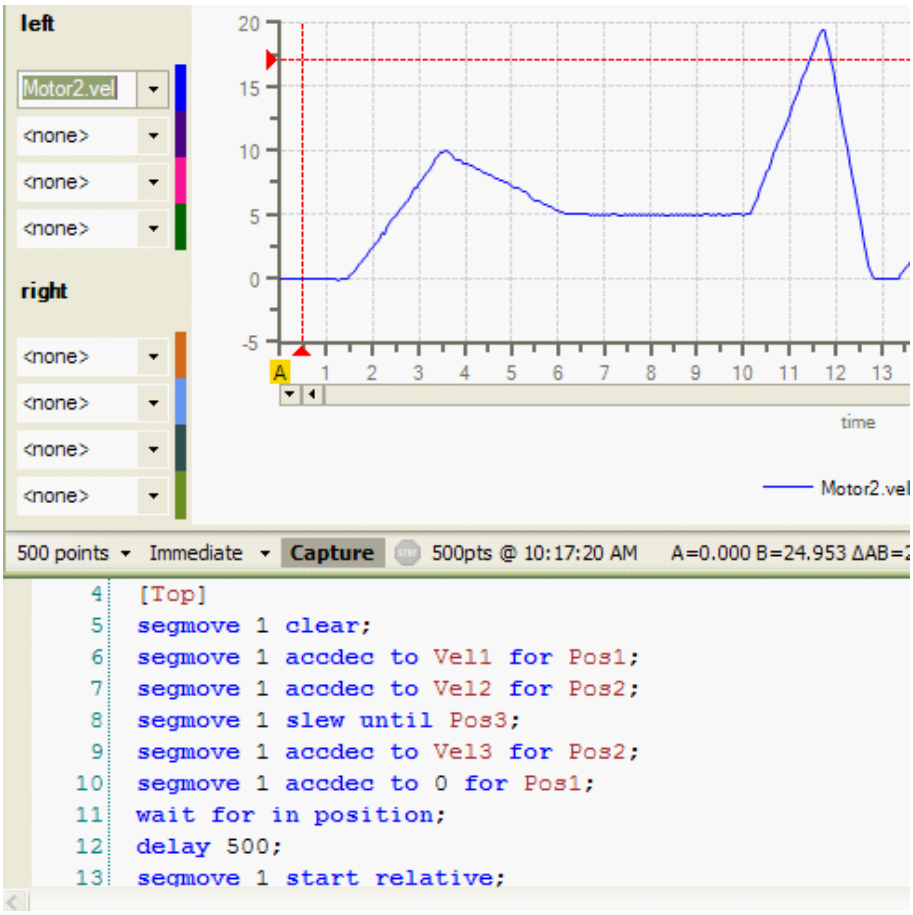



```
[Top]
segmove 1 clear;                    //clear segment table 1
segmove 1 accdec to 10 for 10;      //accel to a vel of 10 over a dist of
10
segmove 1 slew until 20;            //move at current vel 10 till position
20
segmove 1 accdec to 30 for 20;      //accel to a vel of 30 over a dist of
20
segmove 1 slew until 70;            //move at current vel 30 till position
70
segmove 1 accdec to 20 for 30;      //decel to a vel of 20 over a dist of
30
segmove 1 slew until 120;          //move at current vel 20 till position
120
segmove 1 accdec to 0 for 10;      //decel to a vel of 0 over a dist of 10
(stop)
wait for in position;               //Wait for previous moves to be
complete
segmove 1 start relative;           //Start the move
delay 500 ms;                       //delay 500 ms;
goto Top;                           //goto Top and repeat
```


```
[Top]
segmove 1 clear;                    //clear segment table 1
segmove 1 accdec to 10 using 10;    //accel to a vel of 10 using accel of
10
segmove 1 slew until 35;            //move at current vel 10 till position
35
segmove 1 accdec to 40 using 200;   //accel to a vel of 40 using accel of
200
segmove 1 slew until 70;            //move at current vel 40 till position
70
segmove 1 accdec to 5 using 30;     //decel to a vel of 5 using accel of 30
segmove 1 slew until 120;           //move at current vel 5 till position
120
segmove 1 accdec to 0 using 60;     //decel to a vel of 0 using accel of 60
(stop)
wait for in position;               //Wait for previous moves to be
complete
segmove 1 start relative;           //Start the move
delay 500 ms;                       //delay 500 ms;
goto Top;                           //goto Top and repeat
```




```
[Top]
segmove 1 clear;
segmove 1 accdec to 10 for 10;
segmove 1 slew until 20;
segmove 1 accdec to 30 for 20;
segmove 1 slew until 100;
segmove 1 accdec to 20 for 30;
segmove 1 accdec to 0 for 2;
wait for in position;
delay 500;
segmove 1 start relative;
goto Top;
```

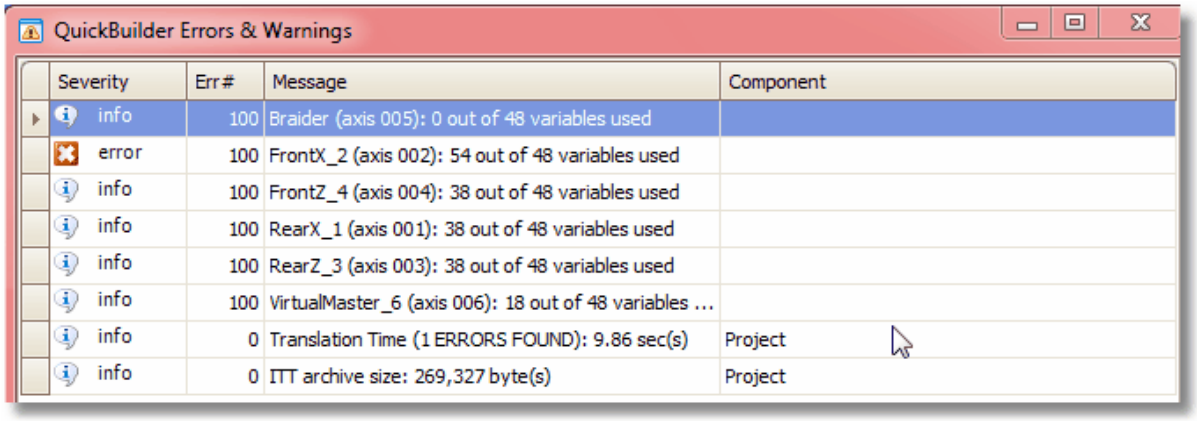



```
[Top]
segmove 1 clear;
segmove 1 accdec to Vel1 for Pos1;
segmove 1 accdec to Vel2 for Pos2;
segmove 1 slew until Pos3;
segmove 1 accdec to Vel1 for Pos2;
segmove 1 accdec to 0 for Pos1;
wait for in position;
delay 500;
segmove 1 start relative;
```

6 Chapter 6: Variables

6.1 User-defined Variables

When entering code into an MSB, variables are automatically defined as they are typed in. A total of 48 user variables per axis are allowed. Each variable is automatically created as a double-precision floating point variable. When you translate a QuickBuilder program the number of variables used on each axis will be displayed, should too many be referenced an error will be flagged:



Variables used in an MSB are automatically assigned to each axis that uses that particular MSB.

For example in the following example two MSB variables are used: *speed* and *dist*.

```
// move at assigned speed for assigned distance
[top]
move at speed for dist;
wait for in position;
delay 500;
goto top;
end;
```

In this sample project, this is the only MSB that QuickStep is calling for the axis named *ax1*, the *speed* and *dist* variables are automatically added to the *ax1* object at the time of translation in QuickBuilder. Note that they will not be shown in the resource tree under an axis object until the project is translated, because it is only at this point that QuickBuilder knows which axes are using which MSBs.

Because a single MSB can be used by more than one axis, the actual variable name has the axis name pre-pended to it so that it can be uniquely accessed by QuickStep (QS4). So in the above example, the variable *dist* used on *ax1* has a name of *ax1.dist* that is used to access it from QuickStep. If that same MSB were to be used on *ax2* as well the *dist* variable *ax2.dist* would be used in QuickStep.

Note that axis prefixes are only used at the QuickStep level and not within MSBs. The MSB cannot directly access a variable from an MSB running on a different axis. If this information is needed, it can be obtained by first assigning the desired MSB variable to a QuickStep variable. This QuickStep variable can then be assigned to a variable in the other MSB. The methodology for reading and writing variables between QuickStep and an MSB is shown below.

QS4 Example code:

```
// QS4 Sample code showing how to update variables between
// QuickStep and MSBs

// set the MSB variable x for Axis1 DrillPosition
// where, DrillPosition is a QuickStep variable
Axis1.x = DrillPosition;

// set the MSB variable speed for Axis1 to 5
Axis1.Speed = 5;

// set the QuickStep variable AxOneTarget
// to the MSB variable Axis1.Target
AxOneTarget = Axis1.Target
```

Axis1 MSB Example code:

```
// MSB Sample code showing how to use updated variables
// between QuickStep and MSBs

Halfspd = Speed/2;

/* Make a trap move to the DrillPosition specified in the QS4 step at
half speed */
move at Halfspd to x;

wait for in position;   // Wait for move to complete
pulse 1 for 1000;       // Turn the drill output on for 1 sec

/* Move to Target at the speed specified above. */

move at Speed to Target;
```

6.2 QuickMotion Pre-defined Variables

In addition to user-defined variables, there are approximately 100 pre-defined variables for an axis in the QuickMotion language.

Many of these variables correspond to properties in the QS4 world.

The pre-defined variables are organized on the following pages into tables by function. The functional groups are:

- **Status Variables** – These are read-only variables that give information as to the status of a given axis, such as fault code, in position, over-travel reached, etc.
- **Control Variables** – These are a mix of read-only and read-write variables used to set general control conditions for the axis and how it interfaces with the drive. Some of these can only be adjusted before the axis is enabled.
- **Tuning Variables** – These variables are all read-write and they are used to adjust the control loop characteristics. These values can be adjusted while the axis is running either by using the tuning wizard in QuickBuilder, or by directly changing the value of the variable.
- **Feedback Variables** – These variables are a mix of read only and read-write that set the characteristics of the encoder feedback. This is where the counts per revolution and the user unit conversions are set.
- **IO Variables** – These variables are used to read the status of the Axis I/O; change the status of outputs, and assign special functionality to an I/O point such as input to be used for positive over travel.
- **Tracking Variables** - This is a large set of variables used to set up electronic gearing and registration type applications. These variables greatly simplify these types of applications from a programming perspective, plus they dramatically improve performance.
- **Capture Variables** - These variables are used for registration/capture routines.
- **Diagnostic Variables** - These variables are useful in monitoring low level functionality internal to QuickMotion.
- **Quickstep Variables** - These variables are used when programming in Quickstep rather than QuickBuilder.
- **Fault Variables** - These variables are used to analyze axis fault conditions.

Status Variables	Description	Type
activeCAM_row	Active cam row presently executing in cam table.	read-only
camRequest	1 requesting cam file from controller disk, 0 idle, else error code.	read-only
capStatus	Capture status, bit 8 (axis 1), bit 9 (axis 2). 1 = active.	read-only
enabled	Holds the state of *drive enable*.	read-only
fault1 fault2 (not used)	Fault status words, reference Chapter 8.	read-only

Status Variables	Description	Type
fault3 (not used) fault4 (not used)		
faulted	Set to *true* when a fault has occurred.	read-only
inpos	Holds the state of *in position*. *In Position* is *true* when the target generator is *inactive* and when the position error (*perr*) is within bounds set by *inposw*.	read-only
overpos	Set to *true* when target position (t*pos*) >= *poslim* or when the associated hardware positive overtravel limit is *active*.	read-only
overneg	Set to *true* when target position (*tpos*) <= *neglim* or when the associated hardware negative overtravel limit is *active*.	read-only
overtrq	Set to *true* when the commanded torque *trqc* has been clamped to the torque limit (either *tmax* or *tlim*).	read-only

pstate	Current axis motion state:
	enum PSTATE {
	IDLE, *// ready to run*
	RUNNING, *// processing sub-steps*
	COMPLETE, *// done running, awaiting IDLE*
	STOP, *// stop*
	SLEWSTOP, *// slewed stop*
	SLEWING, *// slewing*
	PRESPLINE, *// pre 'SPLINE' move*
	PRECAM, *// pre 'CAM' move*
	CONT_CAM, *// continue 'CAM' move that was stopped*
	INSPLINE, *// in 'SPLINE' move*
	INCAM, *// in 'CAM' move*
	TABLESTOP, *// stop table*
	TRACKING, *// geared*
	PRETRACKING *// initialization for TRACKING (geared) mode*
	};
		read-only

time	A settable, accurate time counter (sec). This is a floating point variable with precision of the loop period (800uS default).	read-write
zpulse*	Set to *true* when the Z-pulse has been seen. *Note: Currently this does not work properly and only detects the first Z-pulse. Workaround is to watch for a change in ztheta as an indication of Z-pulse.	read-only

Control Variables	Description	Type
acc	Default acceleration rate for absolute and incremental motion. Scaled in user-units/sec/sec.	read-write
cmode	Control mode – controls the structure of the position/velocity loops. *Torque (0)* – Control loop outputs a torque command (velocity loop is active). *Velocity (1)* – Control loop outputs a velocity command to the drive (velocity loop is inactive). *Stepper (2)* – Control loop outputs step and direction pulses to the drive (velocity loop is inactive). *Open loop (16)* – Or this with *Torque* mode for open loop, direct dac control. Write directly to trqc, -10 to 10 (float) representing volts. This variable cannot be changed while the axis is *enabled*	read-write
dec	Default deceleration rate for absolute and incremental motion. Scaled in user-units/sec/sec.	read-write
gtimebase	A global timebase variable that affects both axes. This variable in conjunction with the per-axis *timebase* sets the effective per-axis natural time base of the target generator. This parameter should only be set through a reference to the first axis in an MSB or from a QS4 program.	read-write
jerk_a_req	Requested acceleration jerk (default 0), units/sec^3. Jerk (S-curve generation) for absolute and incremental motion (scaled in user-units/sec/sec/sec). If set to zero (0.0), then S-curve generation is disabled. Set to -1 for automatic calculation based on move.	read-write
jerk_d_req	Requested deceleration jerk (default 0), units/sec^3. Jerk (S-curve generation) for absolute and incremental motion (scaled in user-units/sec/sec/sec). If set to zero (0.0), then S-curve generation is disabled. Set to -1 for automatic calculation based on move.	read-write
jerk_a	Actual acceleration jerk used, units/sec^3.	read-only
jerk_d	Actual deceleration jerk used, units/sec^3.	read-only
newvel	New velocity is used in conjunction with the 'new endposition' command to request a different velocity than is current. If 0 then is ignored.	read-write
sppr	Steps/rev to output when in stepper mode (when *cmode* is *Stepper*)	read-write

Control Variables	Description	Type
stoprate	Rate at which to do a slewed stop (uu/sec/sec)	read-write
theta	Motor angle	read-only
time	Incremented by loop period each interrupt.	read-write
timebase	Used to override the natural time base of the target generator. When set to 1.0 (the default value), the target generator's "time" is un-scaled. When set to a value between 0.0 and 1.0, the target generator's "time" is slowed-down, effectively generating lower velocities. When set to 0.0, motion stops. *Changing the timebase only effects commanded motions, it does not alter other commands such as delay.*	read-write
tlim	Torque limit (Nm) – torque command limit.	read-write
tmax	Scale factor – maximum torque (Nm) that is generated at the motor when the control loop commands 10V to the drive. This is set using the *property inspector* and cannot be changed in QM code. Consult the connected motor and drive specifications to properly set this value. This property is valid when *cmode* is *Torque*.	read-write
vmax	Scale factor – velocity generated when the control loop commands 10V to the drive. Scaled in RPM (rotational) or linear-units/min (linear). This is set using the *property inspector* and cannot be changed in QM code. Consult the connected motor and drive specifications to properly set this value. This property is valid when *cmode* is *Velocity*.	read-write
ztheta	Motor angle of Z.	read-only

Tuning Variables	Description	Type
_highBW	Internal use only	read-write
_inertia	Internal use only (tuning inertia)	read-write

Tuning Variables	Description	Type
_wn	Internal use only (tuning wn)	read-write
_zeta	Internal use only (tuning zeta)	read-write
aff	Velocity-loop acceleration feed-forward gain. Scaled as Nm/(rev/sec)/sec or Nm/(linear-unit/sec)/sec of commanded velocity.	read-write
kd	Velocity-loop derivative gain (D). Scaled as Nm-sec/(rev/sec)or Nm-sec/(linear-unit/sec) of velocity error.	read-write
kfilt	A compensation value for the Kalman-filter used in the velocity estimator.	read-write
kgain	A compensation value for the Kalman-filter used in the velocity estimator.	read-write
ki	Velocity-loop integral gain (I). Scaled as Nm/(rev/sec)/sec or Nm/(linear-unit/sec)/sec of velocity error.	read-write
kv	Velocity-loop proportional gain (P). Scaled as Nm/(rev/sec) or Nm/(linear-unit/sec).	read-write
kvf	Velocity-loop factor (0.0-1.0). When set to 1.0, the velocity loop is a classic PID structure. When set to 0.0, the velocity loop is a classic PDF structure. When set to a value in between, the velocity loop is a combination of both.	read-write
nonvolatile	Writing a 1 will cause tuning parameters to originate from nonvolatile serial flash instead of the defaults used in the property window when the program was first created. 0 clears this feature (one per axis).	read-write
pdead	Position-loop dead-band (user-units).	read-write
pff	Position-loop velocity feed-forward gain (0.0 – 1.0).	read-write
ppg	Position-loop proportional gain. Scaled as 1000/min.	read-write
vff	Velocity-loop velocity feed-forward gain.	read-write

Tuning Variables	Description	Type
	Scaled as Nm/(rev/sec) or Nm/(linear-unit/sec) of commanded velocity.	

Feedback Variables	Description	Type
camming_invertend	Inverts logic of invertmaster with regards to camming table start position and assumed direction traversing. By default 0, follow invertmaster, 1 to do opposite of invertmaster (!invertmaster).	read-write
encoderZ	Z encoder input.	read-only
encoderZ3	Combination of Axis 1 and Axis 2 Z inputs A/B	read-only
fpos	The feedback position scaled in user-units.	read-only
fposc	The feedback position scaled in encoder counts.	read-only
gratio	Present gear ratio.	read-only
inposw	*In Position* window. Controls when the axis is deemed *in position*. Scaled in user-units.	read-write
invertcmd	Whether to invert the sign of the command output: 0 = no inversion 1 = invert	read-write
invertfeed	Whether to invert the way the feedback encoder counts: 0 = count normally 1 = count inverted	read-write
invertmaster	Whether to invert the way the master encoder counts: 0 = count normally 1 = count inverted	read-write
mppr	Master encoder counts per revolution (or per *linear unit* for linear feedback devices).	read-write
neglim	Negative over-travel limit, scaled in user-units.	read-write
perr	The position error (scaled in user-units).	read-only
perrlimit	The limit before a *following-error* fault is generated. The variable is scaled in user-units. 0 = disable *following-error* fault check	read-write

Feedback Variables	Description	Type
ppr	Feedback encoder counts per revolution (or per *linear unit* for linear feedback devices).	read-write
poslim	Positive over-travel limit, scaled in user-units.	read-write
runv	Calculated run velocity fed to PID algorithm.	read-write
sign	Nonzero for SCurve move, 1 for CCW, -1 for CW	read-only
stepsout	Stepper pulses output.	read-only
substep	Segmented move current step on. Trapazoidal is 0 to 2, S-Curve is 0 to 6.	read-only
sfmod	The secondary position modulus. Used to control when *sfposc* wraps around to 0.	read-write
sfpos	Secondary feedback position (in revolutions). sfpos = sfposc * (1/ppr)	read-only
sfposc	A secondary feedback position (scaled in counts). A separately maintained feedback position similar to *fposc* with the exception that the position will "wrap" (modulo) at 0 and at *sfmod* (unless *sfmod* is set to 0).	read-only
tpos	The target position scaled in user-units.	read-only
tposc	The target position scaled in encoder counts.	read-only
trqc	The commanded torque value (Nm). Note that if in torque mode and cmode open loop bit set (bit 5) then this becomes DAC analog output -10 to 10V, floating point.	read-only
uud	User-units conversion factor (*denominator*). Motion commands are divided by this value (after multiplying by uun) to scale user-units to revolutions (or *linear unit* or linear feedback devices).	read-write
uun	User-units conversion factor (*numerator*). Motion commands are multiplied by this value (then divided by uud) to scale user-units to revolutions (or *linear unit* or linear feedback devices).	read-write

Feedback Variables	Description	Type
vcmd	Commanded velocity (in rev/sec or linear-units/sec).	read-only
vel	Feedback velocity (in rev/sec or linear-units/sec).	read-only
verr	Velocity error (in rev/sec or linear-units/sec).	read-only
zfpos	Holds the last feedback position before it was modified by a "*zero feedback position*" or "*zero following error*" statement.	read-only
ZPULSE_POS	The next Z-pulse location in the positive direction; user-units	read-only
ZPULSE_NEG	The next Z-pulse location in the negative direction; user-units	read-only
ztpos	Holds the last target position before it was modified by a "*zero feedback position*" or "*zero following error*" statement.	read-only

IO Variables	Description	Type
ctr0-ctr7	Axis counters (64-bit). These variables count off-to-on transitions of the eight axis-related inputs (5 digital inputs, A, B and Z). **variable** **input (M3-40A/B/C)** ctr0 din1 ctr1 din2 ctr2 din3 ctr3 din4 ctr4 din5 ctr5 A-encoder channel ctr6 B-encoder channel ctr7 Z-encoder channel	read-write
din1 – din5	The state of digital inputs 1 through 5.	read-only
din6 – din10	The state of digital inputs 6 through 10. *Valid only when the module is in 1½ axis mode*	read-only
dout1 – dout5	The state of digital outputs 1 through 5.	read-only
dout6 – dout10	The state of digital outputs 6 through 10. *Valid only when the module is in 1½ axis mode*	read-only
dins	The state of digital inputs 1 through 5 (or 10 if in 1 ½ axis mode) as a single integer. *Each input has its own binary value starting with 1 for din1, 2 for din2, 4 for din3, 8 for din4, etc. As an example, if din3 and din5 were both on, dins would equal 20.*	read-only

IO Variables	Description	Type
douts	The state of digital outputs 1 through 5 (or 10 if in 1 ½ axis mode) as a single integer. *Each output has its own binary value starting with 1 for dout1, 2 for dout2, 4 for dout3, 8 for dout4, etc. As an example, if dout3 and dout5 were both on, douts would equal 20.*	read-only
driveenable	The digital output number to use for "drive enable." Positive input number for true state=high Negative number for true state=low 0 = use no output *When an output is assigned for use as drive enable, all set/clear operations to that output are ignored.*	read-write
overposin	The digital input number to use for positive over-travel. Positive input number for true state=high Negative number for true state=low 0 = disable positive over-travel checking	read-write
overnegin	The digital input number to use for negative over-travel. Positive input number for true state=high Negative number for true state=low 0 = disable negative over-travel checking	read-write
running	The digital output number to use for "MSB active" (running). 0 = use no output *When an output is assigned for use as "MSB active" (running), all set/clear operations to that output are ignored.*	read-write

Tracking Variables	Description	Type
antibackup	Whether or not to allow the slave to generate geared pulses in response to negative displacements of the master 0 = allow generated pulses in all cases 1 = accumulate negative displacements of the master and generate geared slave pulses when accumulated total > 0	read-write
mcinv	The bit-oriented variable controls when *mposc1-5* are cleared. Bit 0, the least-significant bit, controls *mposc1*. Bit 1, the next significant bit controls *mposc2*, etc. Bit 16 to bit 20 controls whether mposc# is cleared upon entering *tracking* mode. If set cleared. Bit 21 to bit 22 controls whether tsc1/tsc2 is cleared upon entering *tracking* mode. If set cleared.	read-write
mdelta1	Master position delta, counts This variable holds the displacement that occurred in the master encoder *between* position captures. Essentially this is the last value of	read-only

Tracking Variables	Description	Type
	mposc1 before *mposc1* is cleared due to a position capture. Cleared upon entering *tracking* mode.	
mdelta2	Master position delta, counts This variable holds the displacement the occurred in the master encoder *between* position captures. Essentially this is the last value of *mposc2* before *mposc2* is cleared due to a position capture. Cleared upon entering *tracking* mode	read-only
mdelta3	Master position delta, counts This variable holds the displacement the occurred in the master encoder *between* position captures. Essentially this is the last value of *mposc3* before *mposc3* is cleared due to a position capture. Cleared upon entering *tracking* mode	read-only
mdelta4	Master position delta, counts This variable holds the displacement the occurred in the master encoder *between* position captures. Essentially this is the last value of *mposc4* before *mposc4* is cleared due to a position capture. Cleared upon entering *tracking* mode	read-only
mdelta5	Master position delta, counts This variable holds the displacement the occurred in the master encoder *between* position captures. Essentially this is the last value of *mposc5* before *mposc5* is cleared due to a position capture. Cleared upon entering *tracking* mode	read-only
mmc	Master position modulus (0=functionality disabled) This variable is used as a *modulus* for the variables *mposc1-5 and tmod*. Whenever updated, *mmc* is applied by formula: *mposc1* = *mposc1* **mod** *mmc* *mposc2* = *mposc2* **mod** *mmc* *mposc3* = *mposc3* **mod** *mmc* *mposc4* = *mposc4* **mod** *mmc* *mposc5* = *mposc5* **mod** *mmc* *tmodc* = *tmodc* **mod** *mmc*	read-write

Tracking Variables	Description	Type
mposc	Master position, counts This variable is cleared when the mode is changed to *tracking*. This variable is unaffected by *mmc* or changes to *mposc1-5*. This counter rolls over at 65536 times the value of *mppr* when in *tracking* mode.	read-write
mposc1	Master position, counts (modulo by *mmc*) This variable is cleared when the mode is changed to *tracking*. This variable is cleared when input #1 makes an *off-to-on* transition unless the 0-bit in *mcinv* is set in which case this variable is cleared when the input makes an *on-to-off* transition. Bit 16 in *mcinv* set will disable clearing upon entering *tracking* mode.	read-only
mposc2	Master position, counts (modulo by *mmc*) This variable is cleared when the mode is changed to *tracking*. This variable is cleared when input #2 makes an *off-to-on* transition unless the 1-bit in *mcinv* is set in which case this variable is cleared when the input makes an *on-to-off* transition. Bit 17 in *mcinv* set will disable clearing upon entering *tracking* mode.	read-only
mposc3	Master position, counts (modulo by *mmc*) This variable is cleared when the mode is changed to *tracking*. This variable is cleared when input #3 makes an *off-to-on* transition unless the 2-bit in *mcinv* is set in which case this variable is cleared when the input makes an *on-to-off* transition. Bit 18 in *mcinv* set will disable clearing upon entering *tracking* mode.	read-only
mposc4	Master position, counts (modulo by *mmc*) This variable is cleared when the mode is changed to *tracking*. This variable is cleared when input #4 makes an *off-to-on* transition unless the 3-bit in *mcinv* is set in which case this variable is cleared when the input makes an *on-to-off* transition. Bit 19 in *mcinv* set will disable clearing upon entering *tracking* mode.	read-only
mposc5	Master position, counts (modulo by *mmc*) This variable is cleared when the mode is changed to *tracking*. This variable is cleared when input #5 makes an *off-to-on* transition unless the 4-bit in *mcinv* is set in which case this variable is cleared	read-only

Tracking Variables	Description	Type
	when the input makes an *on-to-off* transition. Bit 20 in *mcinv* set will disable clearing upon entering *tracking* mode	
move_master_counts	Move master counts target if not forever.	read-only
move_master_rate_target	Move master rate target setting (virtual master).	read-only
move_master_ramp	Move master ramp setting (virtual master)	read-only
move_master_rate	Move master rate setting (virtual master)	read-only
mpgai	Master position during *gear…at…in*, counts This variable holds the number of consumed master position counts during the last *gear…at…in* statement	read-only
mpgfi	Master position during *gear…for…in*, counts This variable holds the number of consumed master position counts during the last *gear…for…in* statement	read-only
sdc	Slave decrement counter This counter decrements for every output slave count whereas *tsc1* and *tsc2* increment.	read-write
spgai	Slave position during *gear…at…in*, counts This variable holds the number of consumed slave position counts during the last *gear…at…in* statement.	read-only
spgfi	Slave position during *gear…for…in*, counts This variable holds the number of consumed slave position counts during the last *gear…for…in* statement.	read-only
smodc	Slave position counter. This variable is cleared when the mode is changed to *tracking*.	read-only
smod	Slave position modulus This variable is used as a *modulus* for the variable smodc. Whenever *smodc* is updated, *smod* is applied by formula: `smodc = smodc mod smod`	read-write
smark	Slave modulo marker position, counts When an input transistions in conjunction with the bits specified in	read-only

Tracking Variables	Description	Type
	smarkrise and *smarkfall*, this variable is computed by formula: `smark = sphase – smodc`	
smarkrise	This bit-oriented variable controls when *smark* is calculated. When the input corresponding to a set bit in *smarkrise* makes an off-to-on transition, *smark* is calculated. Bit 0, the least significant bit, represents input #1, etc.	read-write
smarkfall	This bit-oriented variable controls when *smark* is calculated. When the input corresponding to a set bit in *smarkfall* makes an on-to-off transition, *smark* is calculated. Bit 0, the least significant bit, represents input #1, etc.	read-write
sphase	Slave marker position phase, counts Used to offset *smark*.	read-only
tmc1	Temporary master position counter 1 This variable is auxiliary, settable counter that tracks master position. These variables can be zeroed atomically by *zero master counters*. Cleared when changing mode to TRACKING.	read-write
tmc2	Temporary master position counter 2 This variable is auxiliary, settable counter that tracks master position. These variables can be zeroed atomically by *zero master counters*. Cleared when changing mode to TRACKING.	read-write
tmodc	Temporary master position counter mod mmc. User cleared only.	read-write
tsc1	Temporary slave position, counter 1 This variable is auxiliary, settable counter that tracks slave position.	read-write
tsc1rise	This bit-oriented variable controls when *tsc1* is cleared. When the input corresponding to a set bit in *tsc1rise* makes an off-to-on transition, *tsc1* is cleared. Bit 0, the least significant bit, represents input #1, etc.	read-write
tsc1fall	This bit-oriented variable controls when *tsc1* is cleared. When the input corresponding to a set bit in *tsc1fall* makes an on-to-off transition, *tsc1* is cleared. Bit 0, the least significant bit, represents input #1, etc.	read-write
tsc2	Temporary slave position, counter 2 This variable is auxiliary, settable counter that tracks slave position.	read-write

Tracking Variables	Description	Type
tsc2rise	This bit-oriented variable controls when *tsc2* is cleared. When the input corresponding to a set bit in *tsc2rise* makes an off-to-on transition, *tsc2* is cleared. Bit 0, the least significant bit, represents input #1, etc.	read-write
tsc2fall	This bit-oriented variable controls when *tsc2* is cleared. When the input corresponding to a set bit in *tsc2fall* makes an on-to-off transition, *tsc2* is cleared. Bit 0, the least significant bit, represents input #1, etc.	read-write
vmdelta	Virtual master delta counts.	read-only

Capture Variables	Description	Type
capArmed	Capture armed, non zero. If capture is armed and this variable is cleared any capture will be ignored (equivalent to disabling capture).	read-write
capEdge	Edge to monitor for capture as set by the 'set capture' instruction. 2 – any edge, 1 – rising edge, 0 – falling edge.	read-only
capGate	Capture input used to gate the trigger input, if -1 then always gated.	read-write
capGateState	Gate active as on or off, 0 it waiting for gate to be high, 1 if waiting for gate to be low.	read-only
capInput	Capture input to be used as a trigger.	read-only
capLimit	Capture limit value	read_write
capLimitflag	Set if capture limit occured	read-only
cappos	Capture position in user units. capposc * 1/ppr.	read-only
capposc	Capture position in counts. This will be fposc + encoder offset if not in Tracking mode. If Tracking then sfposc + encoder offset. Latched when defined capture input goes active.	read-only
capStatus	Capture status, bit 8 (axis 1), bit 9 (axis 2). 1 = active.	read-only
capTriggered	Capture occurred flag, non zero. capposc/cappos contains the latched position when occurred.	read-write
capWait	If 1, an MSB is waiting on a 'wait capture' instruction, else 0.	read-only
capwaitBranch	Capture MSB offset branch value.	read-only
capwinEnd	End of capture range as set by the 'set capwin' instruction. If same as capwinStart then no window is active.	read-write
capwinStart	Start of capture range as set by the 'set capwin' instruction. If same as capwinEnd then no window is active.	read-write
capOffset	Amount to move after a capture occurs, fposc + capOffset = new end position. If 0 then move continues as was originally instructed.	read-write

capwinType	Capture window type (0-17): **fposc** (0) feedback position **mposc1 - mposc5** (1-6) master position counters #1 through #5 **mposc** (7) master position counter **smodc** (8) slave position (modulo) **smark** (9) slave marked position **tmc1 tmc2** (10/11) temporary master counters #1 & #2 **tsc1 tsc2** (12/13) temporary slave counters #1 & #2 **sdc** (14) slave decrement counter **fposc1** (15) feedback position of axis 1 (fposcA) **fposc2** (16) feedback position of axis 2 (fposcB) **tmodc** (17) temporary master counter mod mmc **sfposc** (18) secondary feedback position of axis	read-only
msource	Master source setting: 0x01 – feedback1 0x02 – feedback2 0x03 - feedbackz 0x04 – target1 0x05 – target2 0x06 – common 0x07 - virtual Bit OR of above: 0x80 - global	read-only

Diagnostic Variables	Description	Type
activeBG_MSBs	Number of active background MSB's running on axis.	read-only
activeFG_MSBs	Number of active foreground MSB's running on axis.	read-only
debugTable	Cam table to view, from 0 to 5	read-write
debugTableRows	Number of rows presently in the selected cam table, debugTable.	read-only
debugTableRow	Current row number to view in the selected cam table, debugTable.	read-write
debugTableX	X value for selected debugTableRow.	read-only
debugTableY	Y value for selected debugTableRow.	read-only
lastOverall	Last full loop time of all axis in uS.	read-only
loopperiod	Periodic motion loop interrupt time in uS.	read-only
looprate	Number of motion loop interrupts/second.	read-only

minLoopTime	Minimum actual individual axis loop execution time (uS) reached.	read-only
maxLoopTime	Maximum actual individual axis loop execution time (uS) reached.	read-only
minOverall	Minimum actual axis loop execution time (uS) reached for all axis.	read-only
maxOverall	Maximum actual axis loop execution time (uS) reached for all axis.	read-only
overflowFlag	Motion loop overflow flag, set to 1 if loop time exceeded while in the loop (800uS default, reference 'set loopperiod')	read-only

6.3 Host Register Access

The Host Read/Write commands are used to directly access all the main controller's registers, including variant storage. These registers consist of, but are not limited to:

- Analog I/O
- Digital I/O
- Data tables
- Volatile and non-volatile Variant scalar, vector and tables
- Generic integer registers
- Non-volatile register
- Communications

Reference the Quickstep Register Guide for a summary of available registers:

http://www.ctc-control.com/customer/techinfo/docs/5300_951/951-530006.pdf

Summary:

host read *variable, register {, row, column}*
host write *variable, register {, row, column}*

Host Read	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
host read *variable, register {, row, column }*				
parameters				
variable	Local user variable or axis property to have 'register' stored to it.			
register	Main controller QuickBuilder register as defined in the Model 5300 Quick Reference Register Guide. May be constant or variable access.			
row	Optional row used only for variant register table access. May be constant or variable access.			
column	Optional column used only for variant register table access. May be constant or variable access.			

This statement pauses execution of the MSB while the contents of a QuickBuilder register is retrieved from the main processor. The register value is then stored into the local 'variable' or axis 'property'. The data type will automatically be converted to that of the local storage. Both integer based registers and variant vectors and tables are supported. When reading a variant, one cell at a time in the table (if any) is read. If no row or column is specified, 0 is assumed.

```
// Read the controller tick timer referencing a variable
// and store to 'userVar'
reg = 13002;
host read userVar, reg;
// Read the controller tick timer using constant register
// number and store to 'userVar'
host read userVar, 13002;
```

Host Write	☑ Positioning	☑ Slewing	☑ Tracking	☑ BG MSB ☑ FG MSB
syntax				
host write *variable, register* {, *row, column* }				
parameters				
variable	Local user variable or axis property to store to controller 'register'.			
register	Main controller QuickBuilder register as defined in the Model 5300 Quick Reference Register Guide. May be constant or variable access.			
row	Optional row used only for variant register table access. May be constant or variable access.			
column	Optional column used only for variant register table access. May be constant or variable access.			

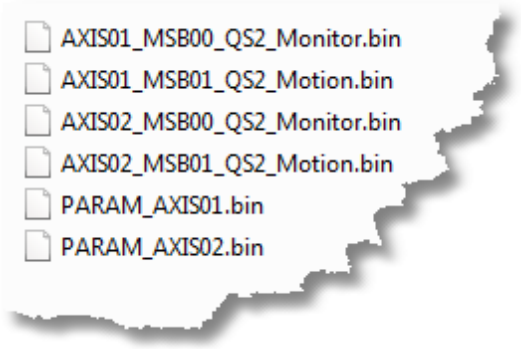
This statement pauses execution of the MSB while the contents of a local 'variable' or axis 'property' is written to a QuickBuilder register on the main processor. The data type will automatically be converted to that of the QuickBuilder register, thus double will be converted to integer, etc. Both integer based registers and variant vectors and tables are supported. When writing a variant, one cell at a time in the table (if any) is written. If no row or column is specified, 0 is assumed.

```
// Clear the controller tick timer, register 13002,
// referencing the contents of 'userVar'.
userVar = 0;
reg = 13002;
host write userVar, reg;
// Clear the controller tick timer, register 13002,
// using a constant value.
host write 0, 13002;
```

7 Chapter 7: Quickstep Support

QuickMotion has been designed for high integration with the QuickBuilder language and include such features as program interaction and user units. A legacy product, Quickstep, uses a register interface for motion control. This interface is not as tightly coupled but there is a large existing code base thus an MSB emulation mode has been created which allows the M3-40 module to appear as a 2219 motion control card, used on the 2700 series controllers, or 5140, within the 5100/5200 controller family.

Register emulation is always available from a read only perspective. In order to fully support the emulation mode a special MSB has been created and must be loaded. This MSB is the output of a QuickBuilder project where initial parameters and any minor MSB customization can be made. To simplify initial use a fully compiled project is available that can be loaded into a 5300 controller for Quickstep program support, QS2MSB. This project is available from the download portion of CTCs' web site. In the example provided, 2 axis, are supported. To support more axis simply add the card to the QuickBuilder project as well as 'start axis' references, or if QuickBuilder is not available, you may simply copy and rename the files with the appropriate axis names. Note the files which are available after compiling QS2MSB, within the controller sub-directory:



These files consist of the binary output, generated by QuickBuilder, for MSB's (AXIS??_MSB??_msbname.bin) and their respective configuration parameters (PARAM_AXIS??.bin). They must be placed in the controller '/_system/Programs/Motion' subdirectory. Upon power up or reset the M3-40A module will automatically look in this directory and if the files are present then an auto-boot sequence will begin. Namely, the files will be loaded into the M3-40 card and automatically executed. If a servo has been tuned and parameters saved to the card, the PARAM file settings will be ignored and only the MSB binary file will be loaded.

Once loaded and running most legacy Quickstep motion applications will run, unchanged.

⚠ Note that the emulation will appear similar to a 2700, 2219 module. Any extended features available within the 5100/5200 controller 5140 module are not currently supported.

7.1 Registers

Most of the motion registers supported by the 2700/5100/5200 controller are available within the 5300, regardless of whether the emulation mode is run or not. If emulation is not running then write operations are not supported. These registers consist of:

<table>
<tr><th colspan="2">Motion Registers Grouped by function then axis</th></tr>
<tr><td></td><td>The 5300 firmware is designed to access up to 16 axes. For the 14XXX register values below substitute the axis number for 'ax' to get the correct register. Axis #1 = 1; For example the position of axis #1 is stored in register 14001.</td></tr>
<tr><td>140ax</td><td>Position (counts), R only [QuickBuilder reference = fposc]</td></tr>
<tr><td>141ax</td><td>Error (counts), R only [QuickBuilder reference = perr * ppr * (uun/uud)]</td></tr>
<tr><td>142ax</td><td>Velocity (counts / sec), R only [QuickBuilder reference = vel * ppr * (uun/uud)]</td></tr>
<tr><td>143ax</td><td>Status, R only:
<table>
<tr><th>Status</th><th>Description</th></tr>
<tr><td>0</td><td>Axis not initialized</td></tr>
<tr><td>1</td><td>Stopped and ready</td></tr>
<tr><td>2</td><td>Motion imminent: waiting for start</td></tr>
<tr><td>3</td><td>Accelerating</td></tr>
<tr><td>4</td><td>At max speed.</td></tr>
<tr><td>5</td><td>Decelerating to new max speed</td></tr>
<tr><td>6</td><td>Decelerating to stop</td></tr>
<tr><td>7</td><td>Soft stop</td></tr>
<tr><td>8</td><td>Registration move (armed, not moving)</td></tr>
<tr><td>9</td><td>Home</td></tr>
<tr><td>10</td><td>Following (not used)</td></tr>
<tr><td>128-255</td><td>Error (not used)</td></tr>
</table></td></tr>
<tr><td>144ax</td><td>Integral Error (count-seconds), R only (not supported)</td></tr>
<tr><td>145ax</td><td>Velocity Feedforward [QuickBuilder reference =QS2_VAR_NEW_VEL_FEEDFORWARD]</td></tr>
<tr><td>146ax</td><td>Deceleration (counts/sec^2) [QuickBuilder reference = QS2_NEW_DECELERATION]</td></tr>
<tr><td>147ax</td><td>Dedicated Inputs, R only:
This is a bit map of the input signals
<table>
<tr><th>Bit Number</th><th>Description</th><th>Bit Number</th><th>Description</th></tr>
<tr><td>0 (lsb)</td><td>Reg. (not supported)</td><td>4</td><td>Rev EOT</td></tr>
<tr><td>1</td><td>Home</td><td>5</td><td>Fwd EOT</td></tr>
</table></td></tr>
</table>

	<table><tr><td>2</td><td>Start</td><td>6</td><td>Z/Index (not supported)</td></tr><tr><td>3</td><td>Kill</td><td>7</td><td>Not Used</td></tr></table>
148ax	Acceleration Feedforward [QuickBuilder reference = QS2_VAR_NEW_ACC_FEEDFORWARD]
149ax	Analog Output, R/W - 32,767 = -10.000V; 32,767 = 10.000V, [QuickBuilder reference = rint (dac_mv * 3.2767)]

Motion Registers Grouped by axis then function

	For the 15xxx, 16xxx, 17xxx register values below substitute the axis number for 'bx' to get the correct register. Axis #1 = 0; For example the position of axis #1 is stored in register 15000.
15bx0	Position (counts), R only [QuickBuilder reference = fposc]
15bx1	Error (counts), R only [QuickBuilder reference = perr * ppr * (uun/uud)]
15bx2	Velocity (counts / sec), R only [QuickBuilder reference = vel * ppr * (uun/uud)]
15bx3	Status, R only: <table><tr><th>Value</th><th>Description</th><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Axis not initialized</td><td>6</td><td>Decelerating to stop</td></tr><tr><td>1</td><td>Stopped and ready</td><td>7</td><td>Soft stop (not used)</td></tr><tr><td>2</td><td>Motion imminent: waiting for start</td><td>8</td><td>Registration move (armed, not moving)</td></tr><tr><td>3</td><td>Accelerating</td><td>9</td><td>Home</td></tr><tr><td>4</td><td>At MAX speed</td><td>10</td><td>Following (not used)</td></tr><tr><td>5</td><td>Decelerating to new MAX speed</td><td>128-255</td><td>Error (not used)</td></tr></table>
15bx4	Integral Error (count-seconds), R only (not supported)
15bx5	Velocity Feedforward, ~~**also used to specify the Output in Direct mode**~~ [QuickBuilder reference = QS2_VAR_NEW_VEL_FEEDFORWARD], output in direct mode not supported, use Analog Output instead (15bx9).
15bx6	Deceleration (counts/sec^2), R only [QuickBuilder reference = QS2_NEW_DECELERATION]
15bx7	Dedicated Inputs, R only: This is a bit map of the input signals

<table>
<tr><td></td><td>
<table>
<tr><th>Bit Number</th><th>Description</th><th>Bit Number</th><th>Description</th></tr>
<tr><td>0 (lsb)</td><td>Reg. (not supported)</td><td>4</td><td>Rev EOT</td></tr>
<tr><td>1</td><td>Home</td><td>5</td><td>Fwd EOT</td></tr>
<tr><td>2</td><td>Start</td><td>6</td><td>Z/Index (not supported)</td></tr>
<tr><td>3</td><td>Kill</td><td>7</td><td>Not Used</td></tr>
</table>
</td></tr>
<tr><td>15bx8</td><td>Acceleration Feedforward [QuickBuilder reference = QS2_VAR_NEW_ACC_FEEDFORWARD]</td></tr>
<tr><td>15bx9</td><td>Analog Output, R/W -32,767 = -10.000V; 32,767 = 10.000V [QuickBuilder reference = rint(dac_mv * 3.2767)]. On 2219 this is read only and Velocity Feedforward is written to for Analog Output.</td></tr>
<tr><td>16bx0</td><td>Reg. Start, R/W – Position at which the registration will be enabled [QuickBuilder reference = QS2_CAP_WINSTART]</td></tr>
<tr><td>16bx1</td><td>Reg. Window, R/W – The range that the registration will be enabled [QuickBuilder reference = QS2_CAP_WINEND_REL]</td></tr>
<tr><td>16bx2</td><td>Reg. Position, R only – The position at which the registration was detected, when Reg status is 1 [QuickBuilder reference = capposc]</td></tr>
<tr><td>16bx3</td><td>Reg. Offset, R/W – The distance to be moved after the registration input [QuickBuilder reference = QS2_CAP_WINOFFSET]</td></tr>
<tr><td>16bx4</td><td>Reg. Status, R/W – 0 = Armed (write 0 to arm), 1 = Detected, can only set to 0 [QuickBuilder reference = QS2_REG_STATUS]</td></tr>
<tr><td>16bx5</td><td>Numerator, R/W – For following the master axis [QuickBuilder reference = QS2_VAR_MTN]</td></tr>
<tr><td>16bx6</td><td>Denominator, R/W – For following the master axis [QuickBuilder reference = QS2_VAR_MTD]</td></tr>
<tr><td>16bx7</td><td>Leader Position, R only – Only valid when following a master axis (not supported)</td></tr>
<tr><td>16bx8</td><td>Leader Velocity, R only – Only valid when following a master axis (not supported)</td></tr>
<tr><td>16bx9</td><td>Reserved</td></tr>
<tr><td>17bx0</td><td>Firmware Revision, R only</td></tr>
<tr><td>17bx1</td><td>Filter & Mode, R/W:
~~In Direct mode the Feedforward Velocity g~~ain specifies the output value (0 to ~~32767) with a value of 32767 = 10V (sign depends on the Filter type)~~.
[QuickBuilder reference = QS2_FILTER_MODE]
<table>
<tr><th>Description</th><th>Value</th></tr>
<tr><td>Lower 3 bits (0x07)
Filter type</td><td>0 or 3 = PID
1 = + Direct (CW) 2 = - Direct (CCW)
4 = PAVff 5 = PAV
6 = Stepper Resolution 7 = Initialize Encoder</td></tr>
<tr><td>Bits 4 & 5
Accel/Decel Type</td><td>0 = Linear 1 = S Curve
~~2 = Parabolic~~ ~~3 = Inverse Parabolic~~</td></tr>
<tr><td>Bit 7 (0x80)</td><td>0=Trajectory Following</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td></td><td>1 (value 128) = Encoder Following</td></tr>
</table>

Note: Initialize Encoder Resolution, filter type 7 is only a temporary mode that can be applied anytime there is no motion. It is recommended this be done prior to initial motion. This can be used to override the default ppr, mppr, and sppr. Upon writing this value the following registers will be initialized as follows, thus set accordingly prior to execution:

ppr = QS2_VAR_NEW_VEL_FEEDFORWARD
mppr = QS2_VAR_NEW_ACC_FEEDFORWARD
sppr = QS2_NEW_DECELERATION

After changing the above variables set them back to their previous values and set the Filter Mode to the proper mode for motion desired.
</td></tr>
<tr><td>17bx2</td><td>Input Polarity, R/W:
This is a bit map that controls the active level of the input signals,. When the bit is 0 then the input is active when it is On; if the bit is 0 then the input is active Off.
<table>
<tr><th>Bit Number</th><th>Description</th><th>Bit Number</th><th>Description</th></tr>
<tr><td>0 (lsb)</td><td>Reg. (not supported)</td><td>4</td><td>Rev EOT</td></tr>
<tr><td>1</td><td>Home</td><td>5</td><td>Fwd EOT</td></tr>
<tr><td>2</td><td>Start</td><td>6</td><td>Z/Index (not supported)</td></tr>
<tr><td>3</td><td>Kill</td><td>7</td><td>Not Used</td></tr>
</table>
</td></tr>
<tr><td>17bx3</td><td>Home Direction, R/W [QuickBuilder reference = QS2_HOME]

<table>
<tr><th>Direction</th><th>Description</th></tr>
<tr><td>CCW</td><td>0 or –1 = Home & Index
-2 = Home Only
-3 = Index Only (not supported)</td></tr>
<tr><td>CW</td><td>1 = Home & Index
2 = Home Only
3 = Index Only (not supported)</td></tr>
</table>
</td></tr>
<tr><td>17bx4</td><td>Options, R only (not supported)</td></tr>
<tr><td>17bx5</td><td>Reserved</td></tr>
<tr><td>17bx6</td><td>Maximum Following Error, R/W default = 30000 [QuickBuilder reference = perrlimit * ppr]]</td></tr>
<tr><td>17bx7</td><td>Speed Limit, R/W – overrides maximum velocity, default = 4194303 steps/sec (not supported)</td></tr>
<tr><td>17bx8</td><td>Maximum Position, R/W – Used as a Software EOT when it is larger than the Minimum Position (not supported)</td></tr>
<tr><td>17bx9</td><td>Minimum Position, R/W – Used as a Software EOT when it is smaller than the Maximum Position (not supported)</td></tr>
</table>

7.2 Quickstep Variables

Quickstep Variables	Description	Type
QS2_Status	Axis status as defined by register 143xx.	read-write
QS2_Cmd	Command to be processed by the MSB, emulates 2700 2219 module.	read-write
QS2_Overrides	Override commands that can be processed by the MSB during motion without a fault.	read-write
QS2_Holding	Holding command to be processed by the MSB.	read-write
QS2_Params	Parameter command to be processed by the MSB. Written once any 'VAR_NEW' variables are updated.	read-write
QS2_VAR_NEW_ACCELERATION	X value for selected debugTableRow.	read-write
QS2_VAR_NEW_MAX_SPEED	Y value for selected debugTableRow.	read-write
QS2_VAR_NEW_PROPORTIONAL	Requested new kd, processed by MSB as required. This is also written to by the QS2 profile statement.	read-write
QS2_VAR_NEW_INTEGRAL	Requested new ki, processed by MSB as required. This is also written to by the QS2 profile statement.	read-write
QS2_VAR_NEW_DIFFERENTIAL	Requested new kd, processed by MSB as required. This is also written to by the QS2 profile statement.	read-write
QS2_VAR_NEW_VEL_FEEDFORWARD	Processed by MSB application.	read-write
QS2_VAR_NEW_HOLDING_MODE	Processed by MSB application.	read-write
QS2_VAR_NEW_DECELERATION	Processed by MSB application.	read-write
QS2_VAR_NEW_FORCE_POSITION	Processed by MSB application.	read-write
QS2_VAR_NEW_FORCE_CUMULATIVE	Processed by MSB application. Not currently used.	read-write
QS2_CAP_WINSTART	Processed by MSB application.	read-write
QS2_CAP_WINEND_REL	Processed by MSB application.	read-write
QS2_CAP_WINOFFSET	Processed by MSB application.	read-write

Quickstep Variables	Description	Type
QS2_VAR_NEW_ACC_FEEDFORWARD	Processed by MSB application.	read-write
QS2_HOME	Processed by MSB application.	read-write
QS2_VAR_MTN	Processed by MSB application.	read-write
QS2_VAR_MTD	Processed by MSB application.	read-write
QS2_LAST_CMD	Last QS2_Cmd processed.	read-write
QS2_CMD_CNT	Number of QS2_Cmd's processed.	read-write
QS2_OVERRIDE_CNT	Number of override commands processed.	read-write
QS2_HOLDING_CNT	Number of holding commands processed.	read-write
QS2_PARAM_CNT	Number of parameter commands processed.	read-write
QS2_MSB_STATE	General scratch storage used by the MSB to write program execution state information.	read-write
QS2_FILTER_MODE	Reference register 17bx1 for mode settings.	read-write
QS2_TMP1	General scratch storage used by the MSB as needed (integer).	read-write
QS2_TMP2	General scratch storage used by the MSB as needed (integer).	read-write
QS2_TMP3	General scratch storage used by the MSB as needed (integer).	read-write
QS2_TMP4	General scratch storage used by the MSB as needed (integer)	read-write
QS2_REG_STATUS	Registration status, write a 0 to arm, else read a 1 if detected.	read-write

7.3 Input Mapping

M3-40A inputs are monitored by the QS2MSB MSB program and when executing the DIN inputs are monitored similar to a 2219. The inputs are mapped as:

DIN1 - START
DIN2 - REGISTRATION INPUT
DIN3 - FWD LIMIT
DIN4 - REV LIMIT
DIN5 - HOME

8 Chapter 8: Fault Codes & MSB Debugging

Should an error occur several registers contain information which can be helpful in detecting what caused the problem. These registers exist for each axis:

Fault Variables	Description	Type
fault1 fault2 (not used) fault3 (not used) fault4 (not used)	Fault status words, reference Chapter 8.	read-only
faulted	0 = no fault, 1 = faulted.	read-only
faultFunction	Code 0 to N which represents internal function where fault occurred. Thus far defines as: FGTick - 1 runMSB - 2 processMSB - 3 process_motion_command - 4 DP_MGRTask_Axis1 - 5 DP_MGRTask_Axis2 - 6 processVFC - 7	read-only
faultMSB	The MSB number from 0 to 31 which has faulted.	read-only
faultMSBLine	The line number as referenced to source code MSB where the fault occurred. Note that the source must be in sync with what is executing for this to be correct.	read-only
faultMSBOffset	Absolute byte offset into MSB binary opcode where was executing when fault occurred. Internal use.	read-only
faultOpcode	MSB opcode that was being executed when fault occurred. Internal use.	read-only

8.1 Fault Codes

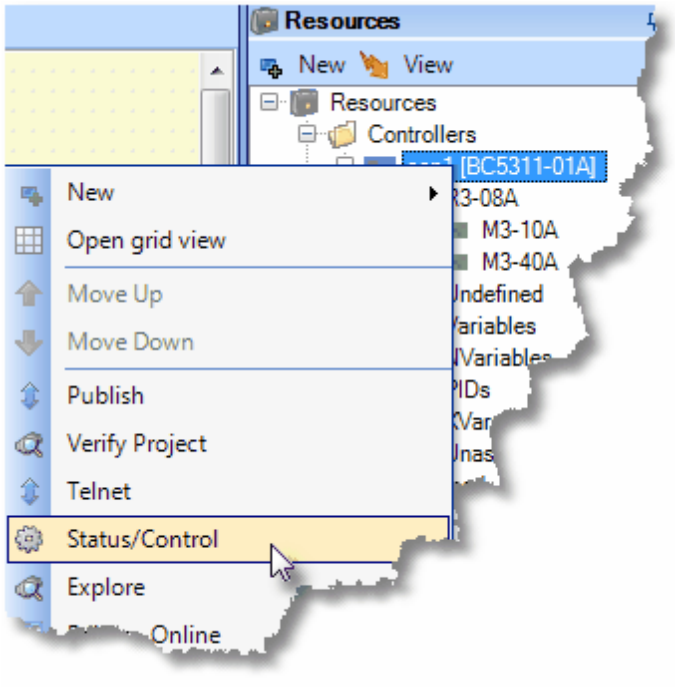
⚠ Note that firmware prior to V1.40 (M3-40A) use outdated fault codes. Changes were made to enhance diagnostic abilities.

Fault Codes	Description	Code Value
MF_GENERICFAULT	Generic Motion Fault.	0
MF_INVALIDTIME	Negative or Zero 'time' specified in MOVE.	1
MF_INVALIDVEL	Negative or Zero 'velocity' specified in MOVE.	2
MF_INVALIDACC	Negative or Zero 'acc' specified in MOVE.	3
MF_INVALIDDEC	Negative or Zero 'dec' specified in MOVE.	4
MF_INVALIDRATE	Negative or Zero 'rate' specified in MOVE.	5
MF_ONLYINBG	QM command only allowed in BG MSB.	6
MF_MOTIONACTIVE	MOVE attempted while MOVE in progress.	7
MF_UNIMPLEMENTED	MOVE attempted while MOVE in progress.	8
MF_WRONGMODE	In wrong mode (positioning/tracking/slewing.	9
MF_FGMSBLIMIT	FG MSB limit reached.	10
MF_NOTINSLEW	Not in SLEW mode.	11
MF_FOLLOWERR	Following error limit reached.	12
MF_BADINPUTNO	Invalid input number specified.	13
MF_NOTENABLED	Not enabled.	14
MF_BADARGUMENT1	Bad argument1/parameter.	15
MF_INVALID_TBL_OP	Invalid 'table' operation.	16
MF_NOTINTRACK	Not in 'tracking' mode.	17

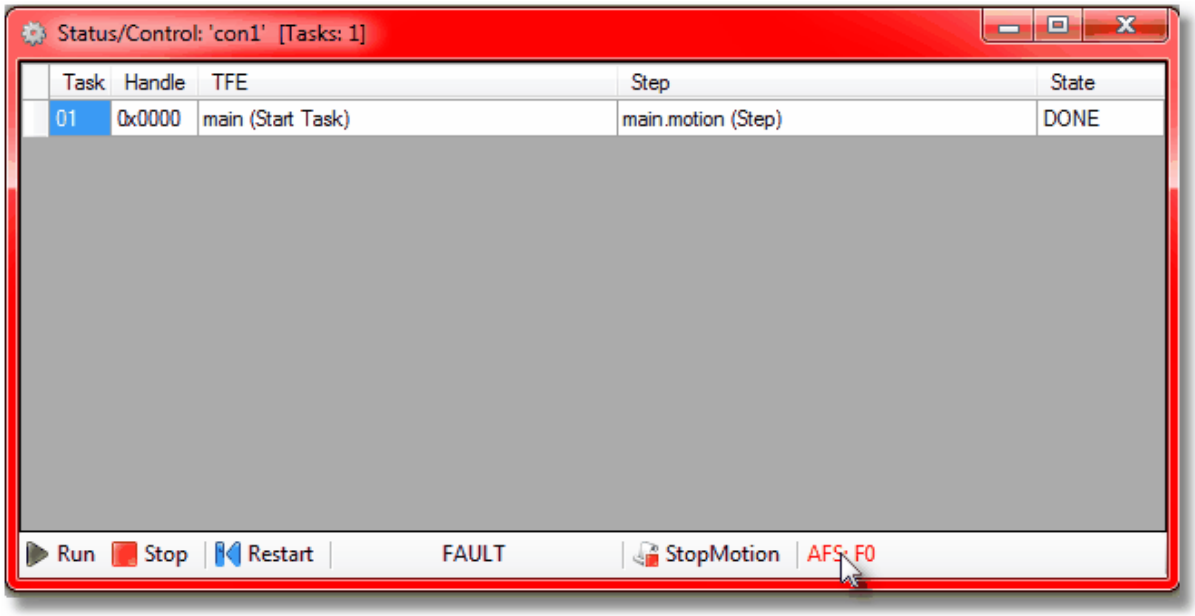
Fault Codes	Description	Code Value
MF_CANTCONSUME	Illegal state for 'consume'.	18
MF_SESGMOVE_ERROR	'Segmented Move' error.	19
MF_SESGMOVE_SIZE	'Segment size' error, too many.	20
MF_NOCAMFILE	Requested CAM file not found.	21
MF_REMOTE_READ	Read of controller register failed.	22
MF_REMOTE_WRITE	Write of controller register failed.	23
MF_NOMSBFILE	MSB file does not exist on flash disk.	24
MF_BADARGUMENT2	Bad argument2/parameter.	25
MF_BADARGUMENT3	Bad argument3/parameter.	26
MF_BADARGUMENT4	Bad argument4/parameter.	27

8.2 MSB Status/Control Monitor Fault Processing

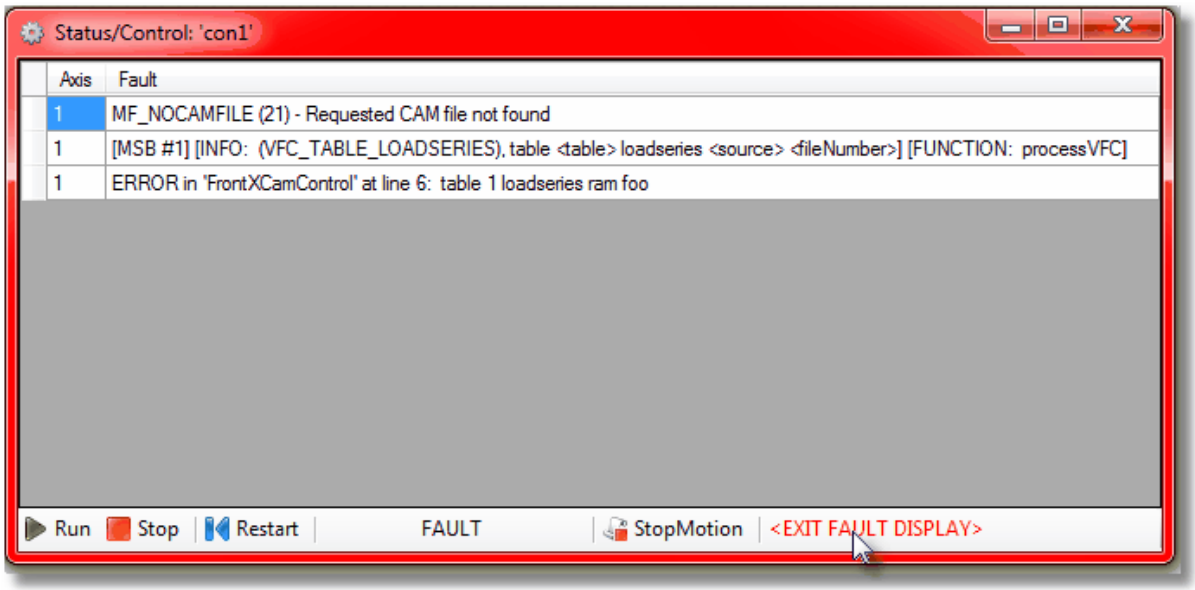
There are a number of features within QuickBuilder to enable the debugging of QuickMotion MSB's. This can be either during normal operation or should a fault occur. A fault is indicate by a flashing FLT LED on the controller CPU. To observe a QuickMotion fault the Status/Control monitor can be viewed:



Once the Status/Control window appears observe and click the AFS text. Note that each character represents an axis, with the first on the far left. In the example below a 0 means the axis is OK, F that there is a fault. Below shows a fault on axis 1 since it is 'F'.



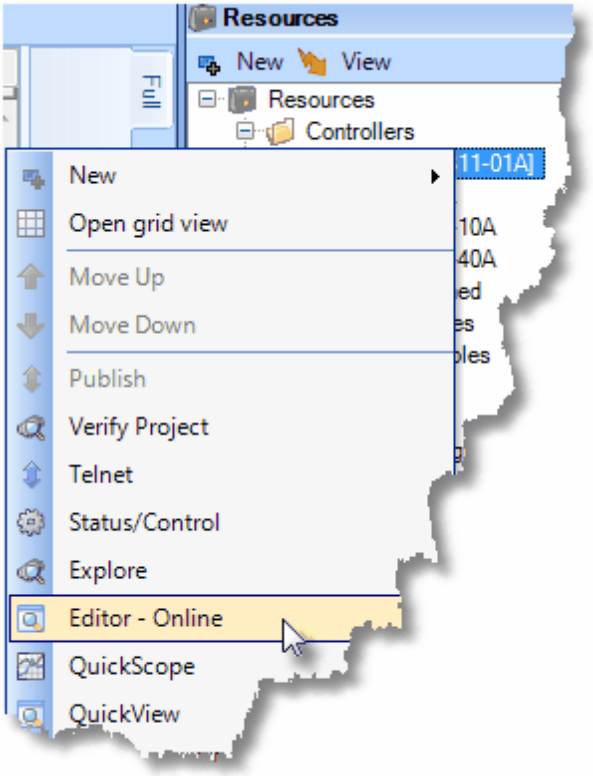
Once clicked detailed information about the fault will be shown, if available:



Note that the error occurred at line 6 of the source code of the FrontXCamControl MSB. In referencing that MSB we can see the line listed, 'table 1 loadseries ram foo' as being the problem. In this case there was no camtable209 file present within the controller flash disk.:

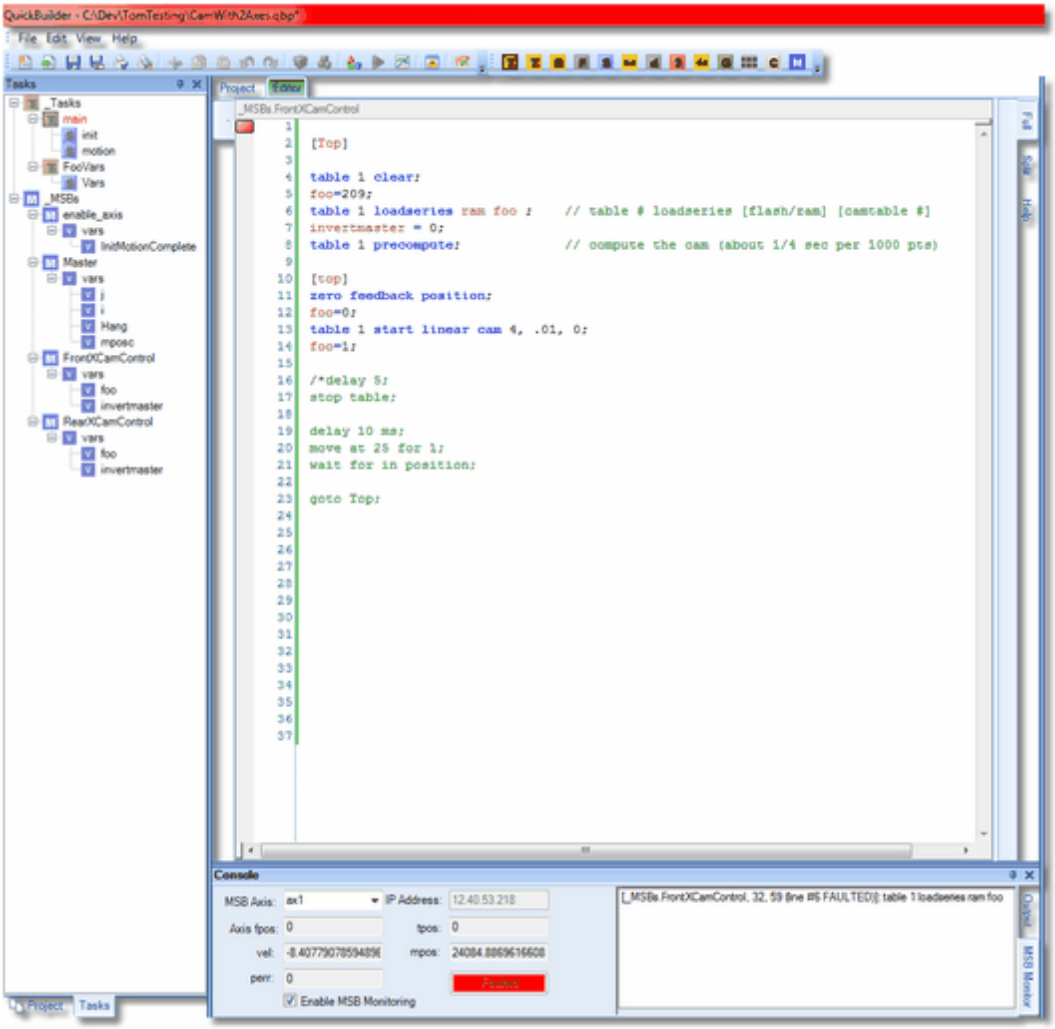
8.3 MSB Monitor

QuickBuilder offers a MSB Monitor when online in the Editor mode.



This monitor periodically (about every second) refreshes axis information for display. Current fpos, mpos, vel, tpos and perr are available as well as the instruction and state of MSB's that are executing. A pull down combo box lists all available axis, that selected is what will be automatically refreshed.

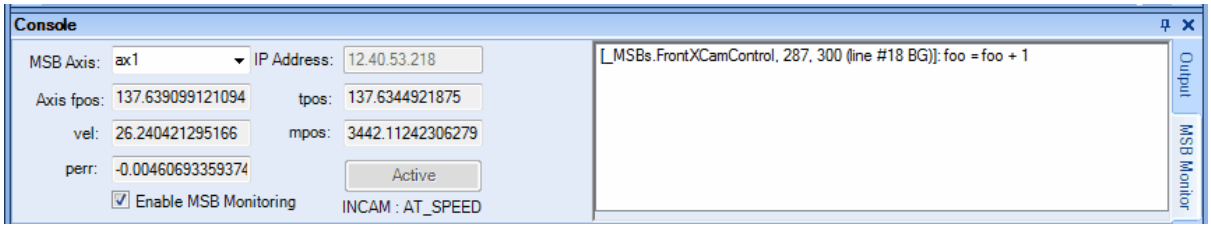
If the axis is faulted, using the example from the 'MSB Status/Control Monitor Fault Processing' section, the following will appear:



⚠ Note that the 'Enable MSB Monitoring' check box must be checked for monitoring to be active. Also the Editor tab should be green to indicate online debug mode.

⚠ Double clicking on the MSB line appearing in the list box will automatically make that code and line current in the Editor.

In situations where a fault had not occurred multiple MSB's would appear executing, as well as their line number and axis motion status:



9 Appendix: Sample Code

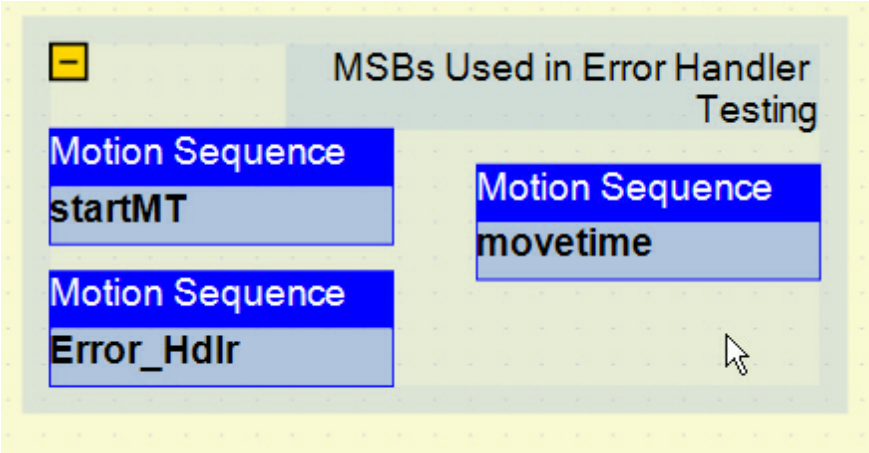
⚠**WARNING:** The following examples are offered for training purposes only and are not intended to perform any actual real-world application or function.

```
// ---------------  Pause Motion MSB  -------------------------

/* This MSB will pause motion by moving the timebase to 0
and then back to 1 based on switch 3 position.
Note: that changes to the timebase variable only impact
the actual motion commands other MSB commands such as
delay are not altered. */

[top]
      wait for rise of 3;           // wait for rising edge of input3
      timebase=0;                   // set the motion timebase to zero
      wait for fall of 3;           // wait for falling edge of input3
      timebase=1;                   // put timebase back to 100%
      goto top;                     // repeat

end;
// ---------------  Jog MSB  -------------------------

/* This MSB performs a simple jogging routine
The variable JogSpeed is passed to this MSB to set
the jog velocity.  If switch 1 is on a positive Jog
is activated, if switch 5 is on a negative Jog is activated;
if neither 1 or 5 are on zero speed is commanded, and the
motor stops. */

JogSpeed=1;                         // set a default jog speed

slew begin;                         // witch to slewing mode

[loop]
      // check the switches
      if !din1 && !din5 then speed=0;
      if din1 then speed = JogSpeed;
      if din5 then speed = -JogSpeed;

      slew at speed in 0.5;         // slew to speed in .5 sec
      delay 510;                    // wait 510 ms until at speed

      if !din2 goto loop;           // as long as input 2 is off loop

      slew end;                     // return to position mode

end;
```

```
// ---------------  Home via Z MSB  --------------------------


// add a move to switch code here if needed

foundz = 0;
// set zdir to 1 to search in the positive dir
// set zdir to -1 to search in the negative dir
zdir = -1;

// check if we know where the Z-pulse is
if zpulse goto knownz;

// dont know where z is, so...
// move positive for +/- 2 revs looking for it
zero feedback position;
move in 0.25 for zdir*2;

[searchloop]
// a z while moving?
// check the zpulse variable (1 if a z pulse has been seen)
if zpulse goto foundmid;
// done?
if !inpos goto searchloop;
// no z, stop and quit
stop;
end;

// found a z mid move, so stop
[foundmid]
new endposition relative 0 using 10000;
wait for in position;

// move to z
[knownz]

// find the Z that is closest
if zdir > 0 goto posz;

[negz]
move in 0.125 to ZPULSE_NEG;
goto exit;

[posz]
move in 0.125 to ZPULSE_POS;
goto exit;

[exit]
wait for in position;
zero feedback position;
```

```
foundz = 1;
end;

// ---------------  Error Handler Example  -------------------------
```

The following MSBs illustrate how an error handler can be used on a motion axis. The code for each is given below with a brief explanation.



startMT MSB: **This MSB starts the asynchronous event handler. In this case Error_Hdlr will automatically be called whenever there is a hardfault.**

```
on hardfault start Error_Hdlr;      //start an asynchronous event to
                                    monitor for a hardfault error
start movetime FG;                        //now go do some motion
```

Error_Hdlr: **This MSB contains the error handler code. The comment at the end gives a listing of valid error codes**

```
// Error handler example MSB

// Check for a fault
if fault1 == 0 goto genfault; //Generic Motion Fault
if fault1 == 1 goto invtime;  //Negative or Zero 'time' specified in MOVE
if fault1 == 2 goto invvel;   //Negative or Zero 'velocity' specified in
MOVE
                                                  //(etc)continue on to trap
all errors if you want

goto unknown;                                 // If you get here there's no
known fault code

[genfault]                                    //routine for general fault goes
here
//(put code here)
```

```
[invtime]
// Invalid Time Fault:  Trigger this error by setting move time to 0 in the
movetime MSB
      setout 5;                    //turn on output to signal error
      delay 5000;                  // wait 5 sec
      clrout 5;                    // turn off output 5
      delay 1000;                  // wait 1 sec
      reset;                             // reset all faults

      delay 1000;                  // wait 1 sec
      start movetime FG;      //re-start the MSB.  Hint if you don't change
                                   the movetime
      //you'll end up right back here in six seconds.

[invvel]

      //(code)

[unknown]

      //(code)

end;
```

movetime MSB: **This MSB contains a simple motion routine used to trigger a hardfault**

```
zero feedback position;

xm=1;             //default mode setting

// Do a repeating forward and back move
[top]

      time=0;                                          //reset timer
      move in time2 for dist mode xm;     //move forward
      wait for in position;

      move in time2 for -dist mode xm; //move back
      wait for in position;
      movetime=time;                    //update movetime

goto top;
```

10 Appendix: Command Hyperlinks

Statements:

Utility Statements:

Set Statements:

Program Flow Statements:

```
[label]
start MSB mode
end { and start MSB mode }
abort MSB
goto label
if condition goto label
on asynchevent asynchhandler
```

Common bits and variables Statements:

```
set common bit number state
wait common bit number state
set common var number value
wait common var number range
```

I/O Statements:

```
setout outputlist
clrout outputlist
pulse output for n
pls output using reference definitions
pls output state
wait for transition of input { or condition }
generate output output rate freq
generate n steps on pair
variable = ctr[n]
ctr[n] = expression
ctr[n] = offset
generate alternate mode
```

Simple Motion Statements:

```
move to position { using acc, dec }
move at maxvelocity to position { using acc, dec }
move trap to position using rate
move in time to position {mode n }
move for displacement { using acc, dec }
move at maxvelocity for displacement { using acc, dec }
move trap for displacement using rate
move in time for displacement {mode n }
wait for in position
new endposition position using rate
new endposition relative displacement using rate
slew begin
slew at velocity in time
```

```
slew for displacement
slew end
```

Gearing Statements:

```
gear at numerator : denominator
gear at numerator : denominator in counts
gear at numerator : denominator in counts after acounts
gear for slavecounts in mastercounts
gear for slavecounts in mastercounts after acounts
offset slave by slavecounts in time
wait master counts
wait slave counts
wait source within start , end
wait source outside start , end
zero masslv counters
```

Position and Capture Statements:

```
set capture transition of input input { gate input gateinput gatestate }
set capwin range start, end using reference { arm }
wait capture { if limit of limit goto limitlabel }
```

Loading Tables Statements:

```
table n clear
table n addpair xexpression , yexpression
table n addseries pairs
table n copy from rowOffset1 to table m rowOffset2 numRows
table n loadoffset rowOffsetFile, numPairs,rowOffsetTable
table n loadseries source fileNumber
```

Spline/CAM Statements:

```
table n continue
table n precompute
table n start imethod tscale , rpscale , repeatcount
table n start imethod cam mpscale , spscale , repeatcount
stop table
```

Virtual Master Statements:

```
move master at rate for limit { using ramp }
```

Segmented Move Statements:

```
segmove table clear
```

segmove table accdec to vel using rate
segmove table accdec to vel for displacement
segmove table slew until position
segmove table stop at position using rate
segmove table start relative

Host Register Statements:

host read *variable, register {, row, column}*
host write *variable, register {, row, column}*

Index

- A -

- C -

- D -

- E -

- F -

- I -

- K -

- M -

- P -

- Q -

- R -

- S -

- T -

- V -