SMART motor™

USER'S GUIDE

Class 5 SmartMotor™ Technology with COMBITRONIC™

Revision 5.23

ANIMATICS®

Defining the Future in Motion Control

Animatics SmartMotor™ Class 5 User's Guide.

Please let us know if you find any errors or omissions in this manual so that we can improve it for future readers. Such notifications should be sent by e-mail with the words "User's Guide" in the subject line sent to: **techwriter@animatics.com**. Thank you in advance for your contribution.

Contact Us:

Animatics Corporation
3200 Patrick Henry Drive
Santa Clara, CA 95054
USA
Tel: 1 (408) 748-8721
Fax: 1 (408) 748-8725
www.animatics.com

Table of Contents

Table of Contents

Continued from preceding page

Table of Contents

Continued from preceding page

Table of Contents

Continued from preceding page

What makes the Animatics SmartMotor by far the most powerful Integrated Motor in the industry is its unique ability to control an entire machine. The combination of programmability, networking, I/O and servo performance is unparalleled, though exactly what you should expect from the Pioneer of Integration.

While priced similarly to other Integrated Servos, the SmartMotor brings true and additional savings to the machine builder by eradicating other expensive and complicated elements in the machine, including PLCs, Sensors, I/O Blocks, Cabinets, etc.

Competitive products deliver only compactness. The SmartMotor is more than a product. It is the natural *by-product* of a **design philosophy**:

1) Reduce development time - shorten Time-To-Market

2) Lower machine production cost

3) Simplify machine, machine build, and support

When you come to believe the above three directives are truly born out in the Animatics SmartMotor product line, you will find that an accurate calculation of the benefit of choosing the SmartMotor does not arise from the comparative acquisition costs. The greatest benefit of using the SmartMotor is that it lets you trump your competition by getting to market weeks or potentially months faster.

The **SmartMotor™** is an entire servo control system built inside of a servo motor. It includes a controller, an amplifier and an encoder. All that is required for it to operate is power, and either an internal program, or serial commands from outside (or both). To make the SmartMotor move, the program or serial host must state a target position, a maximum velocity at which to travel to that target, and a maximum acceleration. Once these three parameters are set, and the two limit inputs are properly grounded or deactivated, a "**Go**" command will start the motion profile.

Motion Control Functions

The controller portion of the SmartMotor performs many functions. When the motor is set to "servo" (hold its position), its windings are charged with current only so much as is necessary to keep the programmed position, either at rest, or over time during motion. This power level is controlled by the "PID filter" and updated 8,000 times per second by default and as fast as 16,000 for maximum performance.

Optional SmartMotor™ cable (CBLSM1-10)

Trajectory generation is also done by the controller, to exacting precision. Position, velocity and acceleration can be changed at any time, even during an existing move. To reach a target position, the SmartMotor will accelerate at the programmed acceleration until it reaches the programmed maximum velocity, whereupon it will travel at that velocity. When it approaches the target position, it will decelerate at the last programmed deceleration rate such that the moment it comes to rest, it will be at the programmed target position. Profiles can be programmed that are all acceleration and deceleration with no slew. The PID control will direct the amplifier to give the motor as much current as required to stay on the trajectory, based on loading. If there is not enough power to move the load and stay on trajectory, there will be a position error and the motor will stop, unless programmed otherwise. The amount of power the SmartMotor requires is entirely dependent upon the load it must move and the motion profile.

Optional PS24V8A or PS48V6A power supply

In addition to the ability to create trajectories, the SmartMotor can position in ratio to incoming encoder or step & direction signals, it can interpolate its position between points in a CAM table, and it can perform complex contours when coordinated by a host computer with custom software, or one of Animatics' standard software programs, including SMNC, Animatics' G-Code based CNC motion control software.

PID control and trajectory generation (or following) are the controller's top priority. Regardless of what else may be processing or happening, these functions will be performed at the full and precise PID rate.

System Control Functions

The SmartMotor's controller can also be programmed in a language similar to BASIC. This capability creates infinite flexibility and in many applications eliminates the need for a PLC (Programmable Logic Controller).

SmartMotors have numerous I/O incorporating multiple functions. Clever programs can define interactions between the I/O, the SmartMotor's shaft motion

and also other peripherals like sensors, light curtains, bar code readers, etc., even other SmartMotors.

Communication Functions

The SmartMotor comes standard with one RS-232 and one RS-485 communication port. These ports can be used to connect SmartMotors together, and/or to a host computer or PLC. In addition to these networks, SmartMotors have the option of being available with a number of industry standard control networks such as CANopen, DeviceNet, Profibus, USB, Ethernet, and others. These other networks can be used for communication between SmartMotors, between a group of SmartMotors and a host, and in many instances allow a SmartMotor to master out to network based I/O expansion modules.

Each industrial network imposes standards for operation and the SmartMotors are designed to conform to those particular standards where industrial fieldbuses are used.

For communication over the SmartMotor's native RS-232 or RS-485 ports, several hundred unique commands are interpreted from incoming ASCII text at a default 9,600 baud. The baud rate is configurable within a user program on SMI. There is no hardware or software handshaking. Commands are simply buffered and interpreted as they come in. Requests can be made of the SmartMotor for data or system status as needed. The commands used in an internal program are the same as those interpreted over the serial channels, except that the program has additional commands for decision making and program flow.

Commands arriving over the serial channels have priority over internal program commands. As a command comes in over the serial channel, it is serviced "next" and then execution is returned to the SmartMotor's program, if it exists and is running. If a request is made for data, such as a request for position: "**RPA**", for example, the current position is output in the form of ASCII text to the main channel, regardless of whether the request was made over the main channel serial network, or by internal program. If a request for data arrives from the secondary serial channel, or other serial network, however, the data is reported to that channel. The SmartMotor uses both spaces and carriage returns as delimiters.

Closeup of optional CAN connector

Like many industrial controls, a string of more than 100 SmartMotors can be connected as CANopen slaves to an external master when equipped with an optional CAN connector. The SmartMotor is the only integrated motor in the world, however, to use Animatics' COMBITRONIC© technology to seamlessly connect all SmartMotors in a chain, allowing any node to be a Master *and* a Slave to every other. These additional communications can coexist with mastered CANopen or DeviceNet communications over the same bus, as well as communications over the main RS-232 and RS-485 ports.

I/O Functions

The SmartMotor's I/O (Input/Output) ports are extremely flexible and provide a variety of digital and analog input and output capability. Each I/O point has

a corresponding pre-assigned variable name within the programming environment and can be read from, or written to, by placing it on the right or left side of an equation, respectively. There are 7 standard I/O and they are 0-5V and are un-isolated.

The SmartMotor has the option of also being equipped with an additional 10 points of isolated 24V I/O. This I/O is sourcing, universal discrete input or output, and 10-bit analog input.

The analog input function is always available with SmartMotor I/O, no matter how the I/O point is configured.

Closeup of optional isolated 24V I/O connector

Most newcomers to Integrated Motors and even most manufacturers of Integrated Motors believe the benefits to be limited to general compactness and simplicity.

For Animatics, the thought to move the control and drive electronics into the servomotors themselves arose from an exercise. The exercise was to, for a moment, look away from the upfront acquisition costs of automation components and instead look at the higher level costs of developing, manufacturing, selling and supporting automated equipment. Then, from that perspective and with a clean sheet of paper, conceive of an automation product family that would be designed to fundamentally address these higher level, and higher value, business issues.

The result was the conclusion that by reshaping the automation components, how they come together, and what they do, we could decisively mine several times the actual product costs directly out of the machine builder's overall expenses.

Simply buying SmartMotors instead of old-style control systems, however, only gets the machine builder half-way through the total potential benefits. Completely capturing all of the savings and opportunity SmartMotors can bring arises from adopting the design vision that gave rise to the SmartMotor, described as follows:

1) Reduce development time - shorten Time-To-Market

The old way of designing automated equipment is to start with a large cabinet, specify and lay out a large array of components within the cabinet and then design all of the requisite wiring. Then add rows of DIN-Rail mount termination points and data or signal converters and the end result is a monumental and time-consuming design and procurement project.

By contrast, a SmartMotor-based machine involves a simple network of motors deployed with predominantly off-the-shelf cables. With Animatics' COMBITRONIC technology linking the SmartMotors together, sensors can be cabled directly to the nearest SmartMotors, unless micro-second response times with motion are required, where the sensors would best be wired directly to the related motor.

Analog AND digital sensors can be read, and air valves and indicator lights can be directly controlled by SmartMotors without channeling all signals through a cabinet.

Historically, equipment with physical motion is deployed with a PLC presiding over I/O and motion. Motion is accomplished through separate controls and amplifiers, or "smart" amplifiers. Where I/O and motion are tightly coupled, days and even weeks are spent getting the PLC to communicate with the motion axes, and/or getting the motion axes to communicate with each other. The unified programming environment the SmartMotor technology delivers eliminates this extended development time, along with the throughput compromises that typically come with it.

Shortened development time accelerates revenue generation and beats others to market.

2) Lower machine production cost

Fewer components simply cost less. Machines automated with SmartMotors often enjoy smaller footprints and are lighter in weight, reducing other material costs, crating and shipping costs.

3) Simplify machine, machine build, and support

Can't debug without a cabinet full of screw terminals? Untrue, since the SmartMotor can read the analog value of every I/O point, regardless of how that I/O point is configured. The software itself can check, automatically and in real time, every point for short circuits, open circuits and failed outputs. Every SmartMotor knows its voltage, its current, and its temperature.

In the best of situations, the machine builder can find one model of SmartMotor that can work in most or all positions and specially label all standard and custom cables.

Debugging is no longer done by a factory expert flown in after delay and great expense to open the cabinet while the machine is down. Typically, the machine user will call, and the machine itself made it clear that there is a problem with a certain sensor, or a certain cable, or a certain motor. Any of these components can be replaced by the customer themselves, the downtime quickly eliminated, and the debugging transplanted back to the original factory, to be conducted at leisure.

Spares on hand can be used to eliminate downtime. This is facilitated by each motor having exactly the same program, with simply different portions of the same program executing in each axis. Ideally, the program would set the SmartMotor address and determine what portion of the program to execute, by looking at an analog input, or a set of digital inputs that are unique at each motor by virtue of wiring. One spare can be placed in any position and know exactly what to do.

Many Animatics customers have reported "never" having to directly service a machine in the field for a motion-related problem since deploying the SmartMotor. One factory paid field service call can destroy the profit on the sale of a machine, and prolonged downtime is a killer.

By contrast, an empowered user experiencing near zero downtime can be a valued customer for life.

Follow these design principles and you will likely find that the SmartMotors are actually paying you.

In order to make the SmartMotor™ run, the following will be needed at a minimum:

1. A SmartMotor
2. A computer running MS Windows
3. A DC power supply for the SmartMotor
4. A data cable to connect the SmartMotor to the computer's serial port or serial adapter.
5. Host level SMI software to communicate with the SmartMotor

The first time user of the SM1700 through SM3400 series motors should purchase the Animatics SMDEVPACK-D. It includes the CBLSM1-10 data and power cable, the SMI software, the manual and a connector kit.

The CBLSM1-10 cable (right) is also available separately.

Animatics also has the following unregulated linear DC power supplies available for the Class 5 SmartMotor: PS24VAG-110 (24 Volt, 8 Amp) and PS42V6A G-110 (42 Volt, 6 Amp). 48V power supplies and protective shunts are also available. For any particular SmartMotor, more torque and speed is available with higher voltage.

When relying on torque/speed curves, pay close attention to the voltage on which they are based. Also, special care must be taken when near the upper voltage limit or in vertical applications that can back-drive the SmartMotor. Gravity influenced applications can turn the SmartMotor into a generator and back-drive the power supply voltage above the safe limit for the SmartMotor. Many vertical applications require a shunt to protect the SmartMotor from damage. Larger open frame power supplies are also available and may be more suitable for cabinet mounting.

Optional SmartMotor™ cable (CBLSM1-10)

Optional PS24V8A or PS48V6A power supply

CAUTION

Many vertical applications require a shunt to protect the SmartMotor from damage

Connecting an SM23205D SmartMotor using a CBLSM1-10 cable assembly and PS24V8A power supply

Software Installation

Follow standard procedures for software installation using the Animatics SMI CD-ROM or files downloaded from Animatics web site at www.animatics.com. After the software is installed, be sure to restart your computer before running the SMI program. With the SMI Software loaded and your SmartMotor connected as shown on the next page, you are ready to start making motion. Turn the SmartMotor's power on and start the SMI Program.

A Quick Look at the SmartMotor Interface

The SmartMotor Interface (SMI) software connects a SmartMotor or a series of SmartMotors to a computer or workstation and gives a user the capability to control and monitor the status of the motors directly from a standard computer. SMI also allows the user to write programs and download them into the SmartMotor's long-term memory.

If you are a first-time user, a simpler interface is available to help you get started. From the SMI main screen, go to the "Tools" menu and select "SmartMotor Playground".

*The **SmartMotor Playground** allows the user to immediately begin making motion without having to know anything about the programming.*

Now, click in the "Detect Motors" button in the upper-right portion of the screen. If your SmartMotor is not properly detected, use the utility to the upper left to select the more appropriate COM port. If you still are unsuccessful, it is likely that your computer is not configured properly for RS-232 communications. This problem should be corrected, or another computer substituted.

Within the SmartMotor Playground, you can experiment with the many different modes of operation. You might start by moving the position slider bar to the right and watching the motor follow. By selecting the "Terminal" tab, you can try different commands found later in this guide.

While the SmartMotor Playground is useful in testing the motor and learning about its capabilities, to develop an actual application you will need to click on the "Close" button at the bottom and launch the SMI development software.

Learning the SmartMotor Interface (SMI)

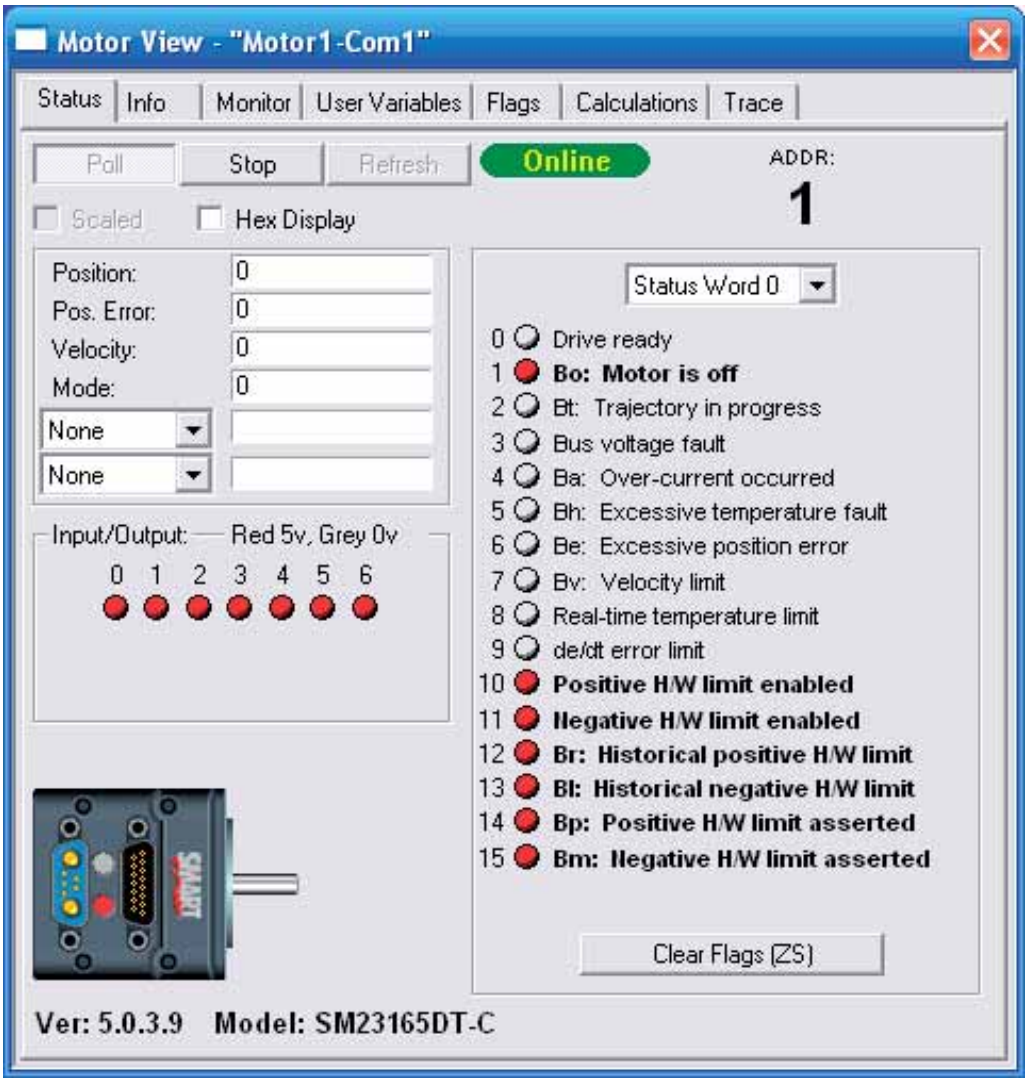
*Every SmartMotor has an ASCII interpreter built in. It is not necessary to use **SMI** to operate a SmartMotor.*

The SMI main screen shows a menu section across the top, a Configuration window on the left, a Terminal window in the center (colored blue) and an Information window on the bottom.

With your motor connected and on, click on the  button located mid-way on the toolbar. If everything is connected and working properly, the motor should be identified in the Information window. If the motor is not found, check your connections and make sure the serial port on your PC is operational.

Monitoring Motor Status

To see the status of the connected motor, go to the "Tools" menu, select "Motor View" and double click on the available motor. Once the Motor View window appears, press the "Poll" button. The SmartMotor™ requires limits to be con-



***Motor View** gives you a window into the status of a SmartMotor*

nected or disabled before the motor will operate. If you see limit errors and you want to move the motor without wiring the limits, you can redefine the Limit Inputs as General Inputs. Then reset the errors by issuing the following commands (in bold) in the Terminal window (be sure to use all caps and don't enter the comments to the right).

```
EIGN(2)     'Disable Left Limit
EIGN(3)     'Disable Right Limit
ZS          'Reset errors
```

Normally, when the motor is attached to an application that relies on proper limit operation, you would not make a habit of disabling them. If your motors are connected to an application and capable of causing damage or injury, it would be essential to properly install the limits before experimenting.

Limit Inputs must be either tied low, or disabled to achieve motion.

CAUTION

You may need to check the "Disable Hardware Limits" boxes and clear the error flags to get motion. ***DO NOT disable limits if this action creates a hazard.***

Initiating Motion

To get the motor to make a trajectory, enter the following into the Terminal window (the dark blue window in the middle of the SMI screen).

Acceleration, Velocity and Position *fully describe a trapezoidal motion profile*

```
ADT=100     'Set Target Acceleration/Deceleration
VT=1000000  'Set Target Velocity
PT=300000   'Set Target Position
G           'Go, Starts the move
```

After the final “G” command has been entered, the SmartMotor™ will accelerate up to speed, slow and then decelerate to a stop at the absolute target position. The progress can be seen in the Motor View window.

Writing a User Program

In addition to taking commands over the serial interface, the SmartMotor can run programs. To begin writing a program, press the  button on the left end of the toolbar and the SMI program editing window will open. This window is where SmartMotor programs are entered and edited.

CAUTION

The larger SmartMotors can shake and move suddenly and should be restrained for safety.

Enter the following program in the editing window. It’s only necessary to enter the boldface text. If you have no limits connected, you may need to add the Limit redefinition code used in the previous exercise to the top of the program. The text preceded by a single quote is a comment and is for information only. Comments and other text to the right of the single quotation mark do not get sent to the motor. Pay close attention to spaces and capitalization. The code is case sensitive and a space is a programming element:

```
EIGN(2)     'Disable Left Limit
EIGN(3)     'Disable Right Limit
ZS          'Reset errors
ADT=100     'Set Target Acceleration
VT=100000   'Set Target Velocity
PT=300000   'Set Target Position
G           'Go, Starts the move
TWAIT       'Wait for move to complete
PT=0        'Set buffered move back to home
G           'Start motion
END         'End program
```

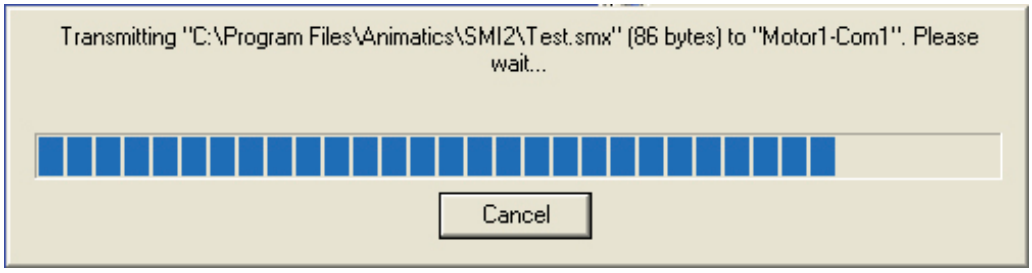
After the program has been entered, select “File” from the menu bar and “Save As” . . . from the drop down menu. In the Save File As window, give the new program a name such as “Test.sms” and click on the “Save” button.

Transmitting the Program to a SmartMotor

Before transmitting the program, press the “Stop” button in the Motor View window to stop the polling. To check the program and transmit it to the SmartMotor, click on the  button located on the tool bar. A small window will ask to which motor you want to download the program. Simply select the only motor presented. SMI compiles the program during this step as well, so errors

may be found in the file. If errors are found, make the necessary corrections and try again.

SMI transmits the compiled version of the program to the SmartMotor.



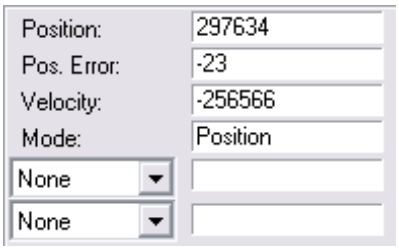
Finally, you will be presented with options relating to running the program. Simply select “Run”. If the motor makes only one move, that is probably because it was already at position 300,000. Press the button on the toolbar and the motor should make both moves.

Since the program ends before the return move is finished, you can try running the program during a return move and learn a bit about how programs and motion work within the SmartMotor.

To better see the motion the new program is producing, press the “Poll” button in the Motor View window and run the program.

With the program now downloaded into the SmartMotor, it is important to note that it will remain until replaced. This program will execute every time power is applied to the motor. To get the program to operate continuously, you will need to write a “loop”, described later on.

A program cannot be “erased”; it can only be replaced. To effectively replace a program with nothing, download a program with only one command: END.



Looking at the position error and feeling the motor shaft will determine if the motor requires additional tuning. For most applications these parameters will suffice, but if still greater precision is required, there is a section on tuning the PID filter later in this manual. Also, the “Tools” menu has a tuning utility that can be of further use. Whether you accept the preceding values or you come up with different ones on your own, you should consider putting the preceding commands at the top of your program, with the F command to put them to work. Alternatively, if you are operating a system with no programs in the motors, be sure to send the commands promptly after power-up or reset.

Tuning the Motor
The SmartMotor shows more than adequate performance with the same tuning parameters. This is largely due to the all-digital design.

Refer to the section on the PID filter for more information on tuning.

Many are surprised at the vast array of different parameters the SmartMotor finds stable. The SmartMotor is so much more forgiving than a traditional control because of its all-digital design. While traditional controls also boast very fast PID rates, the conventional analog input servo amplifier has several calculations worth of delay in the analog signaling, making them difficult to tune. By virtue of it’s all-inclusive design, the SmartMotor requires no analog circuitry or associated noise immunity circuitry, and so the amplifier portion conveys all of the responsiveness the controller can deliver.

All SmartMotor™ commands are grouped by function with the following notations:

#	Numerical integer value, constrained by command, e.g. 0,1,..22.
frm	Formula or number. e.g. 123 or a=1 or a=(2*3)-1.
exp	Simple expression or number. E.g. a+3 or al[1] or 5.
W	Option for addressing I/O 16 Bit Status Words.
msk	Is the mask value of which bits are to be affected when working with Integers can typically be passed into a command as an exp. Mainly for I/O and Status Word bit manipulations. See the Appendix for better understanding Binary Data.

Enter the commands below in the Terminal window and the SmartMotor will move:

```
EIGN(2)     'Disable left limit
EIGN(3)     'Disable right limit
ZS          'Reset errors
ADT=100     'Set target acceleration
VT=100000   'Set target velocity
PT=300000   'Set target position
G           'Go, starts the move
```

*A complete move requires the user to set a position, a velocity and an acceleration, followed by a "**Go**", or **G** command.*

On power-up the motor defaults to position mode. Once Target Acceleration-Deceleration (**ADT**) and Target Velocity (**VT**) are set, simply issue new Target Position (**PT**) commands, followed by a Go (**G**) command to execute moves to new absolute locations. The motor does not instantly go to the programmed position, but follows a trajectory to get there. The trajectory is bound by the maximum Target Velocity and Target Acceleration parameters. The result is a trapezoidal velocity profile, or a triangular profile if the maximum velocity is never met.

Position, Velocity, and Acceleration can be changed at any time during or between moves. The new parameters will only apply when a new **G** command is sent.

Basic Motion Parameters

ADT=frm Set Target Acceleration-Deceleration

Target Acceleration-Deceleration must be a positive integer within the range of 0 to 2,147,483,647. The default is zero, so a non-zero number must be entered to get motion. A typical value is 100. This command sets acceleration and deceleration of the motion profile to the value specified. This value can be changed at any time. The value set does not take effect until the next **G** command is executed. Native acceleration units are (counts/sample/sample)*65536 and the default sample period is 8.0 kHz.

AT=frm Set Target Acceleration Only

DT=frm Set Target Deceleration Only

The **AT**, and **DT** commands allow for setting different values for the acceleration and deceleration of the motion profile, respectively. The standard practice should be to use the **ADT** command instead unless separate values are needed. There is an override that will set **DT** automatically to **AT** if the motor is powered up and only **AT** is set, but this should be avoided by using the **ADT** command if **DT** is not going to be set.

To convert acceleration in revolutions per second^2 to units of **ADT**, **AT**, or **DT**, follow this formula:

ADT = Acceleration * ((enc. counts per rev.)/(sample rate^2)) * 65536

If the motor has a 4000 count encoder (sizes 17 and 23), multiply the desired acceleration, in rev/sec^2, by **4.096** to arrive at the number to set **ADT** to. With an 8000 count encoder (size 34) the multiplier is **8.192**. These factors assume a PID rate of 8.0 kHz, which is the default.

Note that **ADT**, **AT**, and **DT** allow even numbers only. When odd numbers are used, they will be rounded up. The default values are zero.

VT=frm Set Target Velocity

The **VT** command specifies a target velocity (specifies speed and direction) for velocity moves, or a slew speed for position moves. The value must be in the range -2,147,483,647 to 2,147,483,647. Note that in position moves, this value is the unsigned speed of the move and does not imply direction. The value set by the **VT** command only governs the calculated trajectory of **MP** and **MV** modes (position and velocity). In either of these modes, the PID compensator may need to 'catch up' if the actual position has fallen behind the trajectory position. In this case, the actual speed will exceed this target speed. The value defaults to zero so it must be set before any motion can take place. The new value does not take effect until the next **G** command is issued.

To convert velocity in revolutions per second to units of **VT**, follow this formula:

VT = Velocity * ((enc. counts per rev.)/(sample rate)) * 65536

If the motor has a 4000 count encoder (sizes 17 and 23), multiply the desired velocity in rev/sec by 32768 to arrive at the number to set **VT** to. With an 8000 count encoder (size 34), the multiplier is 65536. These factors assume a PID rate of 8.0 kHz, which is the default.

PT=frm Set Target Position (Absolute)

The **PT** command sets an absolute end position to move to when the motor is in Position Mode. The units are encoder counts and can be positive or negative in the range -2,147,483,648 to +2,147,483,647. It is not advisable to attempt to use absolute moves that would cross the rollover point of the most positive and most negative value. Also, absolute moves should not attempt to specify a move with a relative distance with that is more than 2,147,483,647. The end position can be set or changed at any time during or at the end of previous moves. SmartMotor™ sizes 17 and 23 resolve 4000 increments per revolution, while SmartMotor™ size 34 resolves 8000 increments per revolution.

The following program illustrates how variables can be used to set motion values to real-world units and have the working values scaled for motor units for a size 17 or 23 SmartMotor.

```
a=100       'Acceleration in rev/sec*sec
v=1         'Velocity in rev/sec
p=100       'Position in revs
GOSUB(10)   'Initiate motion
END         'End program
C10         'Motion routine
  ADT=a*4.096   'Set Target Acceleration
  VT=v*32768    'Set Target Velocity
  PT=p*4000     'Set Target Position
  G         'Start move
RETURN      'Return to call
```

If any errors exist, they must be cleared before the **G** *command will work. All errors can be cleared with the* **ZS** *command.*

PRT=frm Set Relative Target Position

The **PRT** command allows a relative distance move to be specified when the motor is in position mode. The number following is encoder counts to travel in the range -2,147,483,648 to +2,147,483,647. The relative distance will be added to the current trajectory position and not the actual position, either during or after a move. If a previous move is in still progress, then the current trajectory position will be added to at the point in time when **G** is commanded. Make sure a move has finished before commanding **G** again if the total distance traveled needs to directly correspond to the number of moves made.

G Go, Start Motion

The **G** command does more than just start motion. It can be used dynamically during motion to create elaborate profiles. Since the SmartMotor allows position, velocity and acceleration to change during motion “on-the-fly”, the **G** command can be used to replace the current move with a new one. All faults must be cleared before the **G** command will work, as indicated by the ‘drive ready’ status bit. Faults can be cleared by correcting the fault situation and then issuing the **ZS** command.

S Abruptly Stop Motion in Progress

If the **S** command is issued while a move is in progress it will cause an immediate and abrupt stop with all the force the motor has to offer. After the stop, assuming there is no position error, the motor will still be servoing. The **S** command works in all modes.

X Decelerate to Stop

If the **X** command is issued while a move is in progress it will cause the motor to decelerate to a stop at the last entered deceleration value according to the **ADT**, **DT**, and **AT** commands. When the motor comes to rest it will servo in place until commanded to move again. The **X** command works in Position, Velocity and Torque Modes. It also applies to Follow and Cam Modes.

O=frm Set/Reset Origin to Any Position

The **O=** command (using the letter **O**, not the number zero) allows the host or program to declare the current position to a specific value, positive or negative, or 0 in the range -2,147,483,648 to +2,147,483,647. This command sets the commanded trajectory position to the value specified at that point in time. and the actual position is adjusted similarly. The **O=** command directly changes the motor's position register and can be used as a tool to avoid +/- 31 bit roll over Position Mode problems. If the SmartMotor runs in one direction for a very long time it will reach position -2,147,483,648 or +2,147,483,647, which will cause the position counter to change sign. While that is not an issue with Velocity Mode, it can create problems in absolute position moves or create confusing results when reading position.

OSH=frm Shift the Origin by Any Distance

The **OSH=** command will shift the origin by the amount described, which may be -2,147,483,648 to +2,147,483,647. This command is similar to **O=**, except that it specifies a relative shift. This can be useful in applications where the origin needs to be shifted during motion, without losing any position counts.

OFF Turn Motor Servo Off

The **OFF** command will turn off the motor's drive. When the drive is turned off, the 'PWR/SERVO' status LEDs will revert to flashing green. The motor will not free-wheel by default in the **OFF** state, since each SmartMotor has a safety feature that engages dynamic braking equivalent to the **MTB** command. This has the effect of causing a resistance to motion. To make a SmartMotor truly free-wheel when off, issue **BRKRLS** and be sure any faults are cleared.

MP Position Mode

Issuing the **MP** command puts the SmartMotor in Position Mode. Position Mode is the default mode of operation for the SmartMotor upon power-up. In Position Mode, the **PT**, **PRT**, **VT**, **ADT**, **AT**, and **DT** commands can be used to govern motion. At a minimum, **ADT**, **VT**, and (**PT** or **PRT**) must be issued.

MV Velocity Mode

Velocity Mode will allow continuous rotation of the motor shaft. In Velocity Mode, the programmed position using the **PT** or the **PRT** commands is ignored. Acceleration and velocity need to be specified using the **ADT** and the **VT** commands. After a **G** command is issued, the motor will accelerate up to the programmed velocity and continue at that velocity indefinitely. In Velocity Mode as in Position Mode, velocity and acceleration are changeable on-the-fly, at any time. Simply specify new values and enter another **G** command to trigger the change. In Velocity Mode the velocity can be entered as a negative number, unlike in Position Mode where the location of the target position determines velocity direction or sign. If the 32 bit register that holds position rolls over in Velocity Mode it will have no effect on the motion.

Velocity Mode calculates its trajectory as an ideal position over time and corrects the resulting measured position error instead of measuring velocity error. This is significant in that this mode will 'catch up' lost position just as Position Mode will if a disturbance causes a lagging position error.

MT Torque Mode

In Torque Mode, the motor will apply a PWM commutation effort to the motor proportional to the **T** command, and independent of position. If the motor model has a current-control commutation mode, then torque is controlled in proportion to the **T** command. Otherwise, torque will be dependant on the actual motor speed and bus voltage, eventually reaching an equilibrium speed. Nevertheless, for a locked rotor the torque will be largely proportional to the **T** value and bus voltage.

To run the motor in Torque Mode, use the **T** command and issue a **G** for the new torque value to take effect.

The internal encoder tracking will still take place and can be read by a host or program, but the value will be ignored for motion because the **PID** loop is inactive.

TS=frm Set Torque Slope

The **TS=** command will cause new torque settings to be reached gradually, rather than instantly. Values may be from -1 to +2,147,483,647. -1 disables the slope feature and causes new torque values to be reached immediately. A TS setting of 65536 will increase the output torque by one unit per PID sample period.

*You must issue a **G** for a new torque value to take effect in Torque Mode.*

T=frm Set Torque Value, -32767 to 32767

In Torque Mode, activated by the **MT** command, the drive duty cycle can be set with the **T=** command. The following number or variable must fall in the range between -32767 and 32767. The full scale value relates to full scale or maximum duty cycle. At a given speed there will be reasonable correlation between drive duty cycle and torque. With nothing loading the shaft, the **T=** command will dictate open-loop speed. A **G** must be entered after the **T=** for the new value to take effect.

The following example will increase torque up to 8000 units, one unit every PID sample period.

```
MT         ' Select torque mode.
T=8000     ' Final torque after the TS ramp that we want.
TS=65536   ' Increase the torque by 1 unit of T per PID sample.
G          ' Begin move
```

BRKRLS **Brake Release**
BRKENG **Brake Engage**
BRKSRV **Automatically Release Brake Only When Servo Active**
BRKTRJ **Automatically Release Brake Only When in Trajectory**

The SmartMotor is available with power safe brakes. These brakes will apply a force to keep the shaft from rotating should the SmartMotor lose power. Issuing the **BRKRLS** command will release the brake and **BRKENG** will engage it. There are two other commands that initiate automated operating modes for the brake. The command **BRKSRV** engages the brake automatically, should the motor stop servoing and no longer hold position for any reason. This event might be due to loss of power or just a position error, limit fault, or over-temperature fault.

Finally, the **BRKTRJ** command will engage the brake in response to all of the previously mentioned events, plus any time the motor is not performing a trajectory. In this mode the motor will be off and the brake will be holding it in position, perfectly still, rather than the motor servoing when it is at rest. As soon as another trajectory is started, the brake will release. The time it takes for the brake to engage and release is on the order of only a few milliseconds.

The brakes used in the SmartMotor are zero-backlash devices with extremely long life spans. It is well within their capabilities to operate interactively within an application. Care should be taken not to create a situation where the brake will be set repeatedly during motion or it will reduce the brake's life.

Where a SmartMotor is not equipped with a physical brake, it will simulate the braking with its Mode Torque Brake or **MTB** feature which causes a faulted motor to still experience strong resistance to shaft motion. **MTB** only works when power is applied to the SmartMotor and is no substitute for an actual brake where safety is an issue.

CAUTION

The **MTB** *feature only works when power is applied to the SmartMotor and is not a substitute for a physical brake where safety is an issue.*

EOBK(exp) Re-route Brake Signal to I/O

When the automated brake functions are desired for an external brake, this command can be used to choose a specified I/O port. This corresponds to the same I/O pin numbering used by other I/O commands. These commands re-route the internal brake signal to the respective I/O pins. The brake signal is active high to engage the brake to the shaft on the pulled-up 5 Volt I/O. On the 24 Volt I/O, the default state is off (0 Volts), therefore the brake engages the shaft when the 24 Volt signal is low. The **EOBK(-1)** command removes the brake function from any external I/O. Only one pin can be used as the brake pin at any one time so each command supersedes the other.

MTB Mode Torque Brake

Mode Torque Brake is the default state upon power-up. It causes the motor control circuits to tie the 3 phases of the motor together as a form of dynamic braking. Upon a fault, or the **OFF** command, instead of the motor coasting to a stop, it will abruptly stop. This is not done by servoing the motor to a stop, but by simply shorting all of the coils to ground. If there is a constant torque on the motor, it will allow only very slow movement of the shaft.

The **MTB** command immediately activates dynamic braking independently of the Brake Mode. **MTB** while the motor is running will turn off the motor drive and enable dynamic braking, even if **BRKRLS** has been issued. To remove the effect of the **MTB** command, either issue the **OFF** command, or issue a motion command.

BRKENG can engage dynamic braking unconditionally as well. The opposite of this command is **BRKRLS**; **OFF** will not remove the effect of **BRKENG**.

To allow a motor to freewheel, issue **OFF**. To ensure that the **MTB** command is not active, command **BRKRLS** and the dynamic braking will release. Finally, make sure to clear any fault or choose a fault action of freewheel since faults can also activate dynamic braking.

Status Word 6, Bit 11 reports if dynamic braking is active or not, including as a result of the **MTB** command, the **BRKENG** command, or a fault action.

Commutation Modes

The following commands allow selection of different Commutation Modes. Since the SmartMotor uses a Brushless motor it does not have the mechanical commutator that a brushed motor has to switch the current to the next optimal coil as the rotor swings around. To cause shaft rotation in a brushless motor, the control electronics have to see where the shaft is, and decide which coils to deliver the current to next.

The most typical way to determine the orientation of the rotor is with small magnetic sensing devices called "Hall Sensors". The process of shifting which coils get the current based on shaft rotation is called "Commutation". There are may ways of commutating a motor and the best choice may vary by application.

***MDE**, **MDS**, and **MDC** require angle match before they will take effect. This means the SmartMotor's factory calibration is valid and the index mark of the internal encoder has been seen since startup. Until then, the SmartMotor will operate in default **MDT**.*

MDT Mode Drive Trapezoidal

Trapezoidal commutation uses only the hall sensors (default). This is the most simple method and is ready upon boot up under all circumstances and is very effective despite minor inaccuracies typical in the mechanical placement of the sensors.

MDE Mode Drive Enhanced

This driving method is exactly the same as basic Trapezoidal commutation using Hall Sensors, except that it also uses the internal encoder to add accuracy to the commutation trigger points. This idealized trapezoidal commutation mode offers the greatest motor torque and speed, but can exhibit minor ticking sounds at low rates as the current shifts abruptly from one coil to the next. Since **MDE** uses the encoder, it requires angle match (the first sighting of the encoder index) before it will engage.

MDS Mode Drive Sine

This is Sinusoidal (sine) Commutation, Voltage Mode. It offers smoother commutation compared to Trapezoidal Modes by shifting current more gradually

from one coil to the next, but at a small (10-20%) cost to the motor's torque and speed performance. Since **MDS** uses the encoder, it requires angle match (the first sighting of the encoder index) before it will engage. **MDS** offers beautifully smooth and quiet rotation at low speeds and is the best ergonomic choice.

MDC Mode Drive Current

Available only in the IP-65 versions of Class 5 SmartMotors, this sinusoidal (sine) commutation method, augmented with digital current control, offers the best possible performance without sacrificing quiet operation. It is definitely the best choice on every level where the capability is available. Since **MDC** uses the encoder, it requires angle match (the first sighting of the encoder index) before it will engage.

Status Word 6 contains bits that indicate what commutation mode is currently active. Note that a command for a mode may not take effect until the angle match requirement is met. These Status Bits can be used to test for this success.

Status Word 6:

Bit 0 Trap-hall Mode

Bit 1 Trap-encoder (enhanced) Mode

Bit 2 Sine Voltage Mode

Bit 3 Sine Current (vector) Mode

MDB Trajectory Overshoot Braking (TOB) option.

This command should be used after entering **MDT** or **MDE** to enable TOB action. This option reverts to off when one of the above choices of commutation is made. This option is off by default. Status Word 6, Bit 9 indicates if this mode is active.

MINV(0), MINV(1) Invert Motion Direction

The **MINV(1)** command will invert the direction convention of the SmartMotor and **MINV(0)** will restore the default.

Synchronized Motion using COMBITRONIC™

All SmartMotors that are equipped with the CAN port option come with Combitronic capability, which is basically the unification of all SmartMotors on a CAN network. With Combitronic technology comes the ability to perform multiple axis synchronized motion. The following is the command set that makes multi-axis synchronized linear moves simple.

PTS(), PRTS() Position Target Synchronized abs. and rel.

These commands allow the user to identify two or three axis positions (pos*n*) and associated axis CAN addresses (axis*n*) to cause a synchronized multi-axis move where the combined path velocity is controlled.

PTS(pos1;axis1,pos2;axis2[,pos3;axis3])

In addition to the three axis limitation, the programmer must be mindful of the overall limit of 64 characters per line of code in the SmartMotor. Using variables in place of explicit positions is more space efficient. The **PTS()** command processes the positions as absolute, whereas the **PRTS()** command treats them as relative. After a **PTS()** or **PRTS()** command, the combined distance is stored in the **PTSD** variable and the combined axis move time is stored in the **PTST** variable, in (ms), in the event these may be useful to the programmer. **PTSD** and **PTST** can be used in a program or read over the serial channel by the **RPTSD** and **RPTST** commands. The **PTS()** command first goes out to the Combitronic network and gathers the last target positions in order to calculate the relative motion necessary to get to the next absolute position. It is extremely important that prior to a synchronized move being calculated with the **PTS()** or **PRTS()** commands, the previous target positions are accurate and uncorrupted by origin shifts. Then, it is equally important that the synchronized move NOT be started before each axis reaches its previous target positions.

VTS= Velocity Target for Synchronized Move

The motion along a synchronized move is defined along the path. The **VTS** command is specific to defining the combined velocity of all contributing axes. If the move were to occur in an X-Y plane, for example, the velocity set by VTS would not pertain to the X axis or the Y axis, but rather to the combined motion, in the direction of motion.

ADTS=, ATS=, DTS= Accel. Targets for Synchronized Move

Like the velocity parameter, ADTS pertains to the combined path motion. The **PTS()** command scales the path velocity and accelerations set by the **VTS=** and **ADTS=** commands so that each axis will reach its constant velocity portions at exactly the same time, creating combined, straight-line motion. The **ADTS=** command sets both acceleration and deceleration, whereas **ATS=** and **DTS=** allow the programmer to set separate acceleration and deceleration where desired.

GS Start Synchronized Move

To start a synchronized motion profile, use the **GS** command. It will, behind the scenes, issue **G** commands to all axes involved in the previous **PTS()** or **PRTS()** command. It is important to be sure all motors are at their previous targets before issuing the GS command. Otherwise, the motion will not be synchronized.

TSWAIT Wait for Synchronized Move to complete

After a **GS** command has been issued to start a synchronized move, the **TSWAIT** command can be used to pause program execution until the move has been completed. A standard **TWAIT** command would not work where the motor issuing the **PTS()** and **GS** commands had a zero length contribution to the total move. That is why the **TSWAIT** command was specially created.

```
ADTS=100      'Set target synchronized acceleration
VTS=100000    'Set target synchronized velocity
PTS(30000;1,40000;2)      'Set target positions, axes 1 & 2
GS            'Go, starts the synchronized move
TSWAIT        'Optional wait for synch. move to complete
```

The previous example is a synchronized move in its simplest form. The code could be written in either motor 1 or 2 and it would work the same.

The **TSWAIT** command merely pauses program execution (except for interrupt routines). It may be desirable to continue running the program while waiting. In that event, the program can loop around the Synchronized Move Status Bit, which is status **word 7**, **bit 15,** accessible by variable **B(7,15)**. So the following While Loop code example is equivalent to the **TSWAIT** command, except that more code can be added within the loop for execution during the wait.

```
WHILE B(7,15)==1    'While synchronized move in process
  ...
LOOP        'Loop back
```

The next code example adds subroutine efficiency, the efficiency of setting up the "next" move while the "existing" move is ongoing, and adds an error check before continuing to issue synchronized move commands.

```
ADTS=100      'Set target synchronized acceleration
VTS=100000    'Set target synchronized velocity
WHILE Bt:1|Bt:2|Bt:3      'Loop while motion in any axis
  WAIT=10     'Allow time for other CAN communications
LOOP          'Loop back
x=1000 y=2000 z=3500 GOSUB10 'Put positions into variables
x=2200 y=1800 z=1200 GOSUB10 'Put positions into variables
x=1500 y=2600 z=2500 GOSUB10 'Put positions into variables
x=-120 y=1000 z=1500 GOSUB10 'Put positions into variables
x=0 y=0 z=0 GOSUB10          'Put positions into variables
END         'End Program

C10             'Place label
  PTS(x;1,y;2,z;3) 'Set next positions, axes 1, 2 & 3
                   'and do this while the previous move
                   'is in progress
  WHILE B(7,15)==1 'While synchronized move in process
    'If one motor faults, stop all and end program -
    IF B(0,0):1==0 MTB:0 END    '*note
    IF B(0,0):2==0 MTB:0 END    '*note
    IF B(0,0):3==0 MTB:0 END    '*note
  LOOP          'Loop back
  GS            'Go, starts the synchronized move
```

```
RETURN          'Return to call

'*note: Managing faults is better done by using interrupts
'in other motors, taught later in this guide.
```

There is a note in the preceding example program stating that a better job can be done of detecting and reacting to errors by using interrupts. This is true because the example as written causes a considerable amount of unnecessary communications over the Combitronic interface. By loading interrupt routines in each SmartMotor that constantly monitor for drive status, each motor can be made responsible for reporting a local error. By this means, it is no longer necessary to poll each motor. The motor controlling the synchronized motion can simply do a quick check for reports right before issuing the next **GS** command. This will be shown in greater detail later in the Interrupt Programming section.

Some gantry type multiple-axis machines have two motors operating the same axis of motion. In this instance, an axis address can be repeated as shown in the following example where the "x" axis is driven by motors 1 and 2.

```
PTS(x;1;2,y;3,z;4) 'Set next positions, axes x, x', y & z
```

In this case, the position, velocity and acceleration data sent to axis 1 will be identically sent to axis 2, both axes being the basic "x" axis.

PTSS(), PRTSS() Pos. Target Synch. abs. and rel., Supplemental

The **PTSS()** and **PRTSS()** commands allow supplemental axis moves to be added and synchronized with the previous **PTS()** or **PRTS()** commanded motion. Issue these additional axis commands after a **PTS()** or **PRTS()** command, but before the next **GS**. The commands allow the user to identify one axis position (pos*n*) and associated axis CAN addresses (axis*n*) at a time.

PTSS(pos*n*;axis*n*)

The supplemental axis motions will start at exactly the same time as the main **PTS()** or **PRTS()** motion with the next **GS**, they will transition from their accelerations to their slew velocities at exactly the same time, and decelerate and stop at exactly the same times.

Moves too short to ever reach the **VTS=** velocity will execute a triangular, rather than trapezoidal, profile, but still be synchronized.

What is different about the **PTSS()** move from a **PTS()** member is that the supplemental axis moves do not reduce the primary profile velocity in an effort to hold the total motion to a total combined velocity set by **VTS**. Likewise for Acceleration.

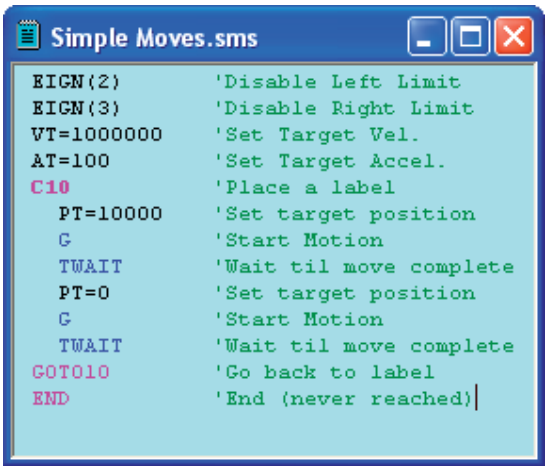
The combined motion of the **PTS()** move will be controlled to the **VTS** limit, and then **PTSS()** moves will simply align with that combined motion.

```
ADTS=100      'Set target synchronized acceleration
VTS=100000    'Set target synchronized velocity
x=1000 y=2000 z=3500 a=100 b=200
PTS(x;1,y;2,z;3)    'Set next positions, axes 1, 2 & 3
PTSS(a;4)           'Set supplemental position, axes 4
PTSS(b;5)           'Set supplemental position, axes 5
GS                  'Go, starts the synchronized move
```

Note that if the supplemental axis move is longer than the **PTS()** move, the supplemental axis velocity will exceed the limit set by **VTS=**.

Program commands are like chores, whether it is to turn on an output, set a velocity or start a move. A program is a list of these chores. When a programmed SmartMotor is powered-up or is reset with the **Z** command, it will execute its program from top to bottom, with or without a host PC connected. This section covers the commands that control the program itself.

SmartMotor programs are written in the SMI software editor opened by selecting “File” - “New”. The simple program example to the right shows an infinite loop. It will cause the motor to move back and forth forever.



Programs execute at rates of thousands of lines per second.

The following are commands that can be used in your program to control how it flows and how it makes decisions:

RUN Execute Stored User Program

If the SmartMotor is reset with a **Z** command or at power-up, all previous variables and mode changes will be erased for a fresh start and the program will begin to execute from the top. Alternatively, the **RUN** command can be used to start the program, in which case the state of the motor is unchanged and its program will be invoked.

RUN? Halt Program If No RUN Issued

The **RUN?** command prevents further execution of code until the **RUN** command is received over the serial channel. Code will execute on power-up to the point of reaching **RUN?**. When **RUN** is issued via the serial port, the CPU will at that point, execute all code from the top-down, jump over the **RUN?** command to the next line of code and continue executing to the end of the program.

```
PRINT(“Boot-Up”,#13)    'Message always prints
RUN?                    'Pgm. stops here on reset or power up
PRINT(“Run Issued”,#13)'This runs if RUN received
END
```

The above code will print only the first message after a **Z** command or upon power-up, but print both messages when a **RUN** command is received over the serial line.

The **RUN?** command placed at the top of your program during development can protect you from accidentally locking up your SmartMotor with a bad program.

Once the program is running, there are a variety of commands that can redirect program flow and most of those can do so based on certain conditions. How these conditional decisions are set up determines what the programmed SmartMotor will do, and exactly how “smart” it will actually be.

GOTO#, GOTO(exp), C# Redirect Program Flow, Place a Label

The most basic command for redirecting program flow, without inherent conditions, is **GOTO#** or **GOTO(exp)**, used in conjunction with the label **C#**. A label consists of the letter **C** followed by a number (#) between 0 and 999 and is inserted in the program as a place marker. If a label, **C1** for example, is placed in a program and that same number is placed at the end of a **GOTO** command, **GOTO1** example, the program flow will be redirected to label **C1** and the program will proceed from there.

```
C10             'Place label
  IF IN(0)==0   'Check Input 0
    GOSUB20     'If Input 0 low, call Subroutine 20
  ENDIF         'End check Input 0
  IF IN(1)==0   'Check Input 1
    a=30        'as example for below
    GOSUB(a)    'If Input 1 low, call Subroutine 30
  ENDIF         'End check Input 1
GOTO(10)        'Will loop back to C10
```

As many as a thousand labels can be used in a program (0 - 999), but, the more **GOTO** commands used, the harder the code will be to debug or read. Try using only one and use it to create the infinite loop necessary to keep the program running indefinitely, as some embedded programs do. Put a **C10** label near the beginning of the program, but after the initialization code and a **GOTO10** at the end. Then every time the **GOTO10** is reached the program will loop back to label **C10** and start over from that point until the **GOTO10** is reached, again, which will start the process at **C10** again, and so on. This will make the program run continuously without ending. Any program can be written with only one **GOTO**. It might be a little harder, but it will tend to force better program organization. Organize your program using the **GOSUB** command instead, and it will be much easier to read and support.

CAUTION

Calling subroutines from the host can crash the stack if not done carefully.

GOSUB#,GOSUB(exp),RETURN Execute a Subroutine and Return

Just like the **GOTO#** command, the **GOSUB#** command, used in conjunction with a **C#** label, will redirect program execution to the location of the label. But, unlike the **GOTO#** command, the **C#** label needs to eventually be followed by a **RETURN** command to return the program execution to the location of the original **GOSUB#** command that initiated the redirection. There may be many sections of a program that need to perform the same basic group of commands. By encapsulating these commands between a **C#** label and a **RETURN**, they become a subroutine and may be called any time from anywhere with a **GOSUB# or GOSUB(exp)**, rather than being repeated in their totality, over and over again. There can be as many as one thousand different subroutines (0 - 999) and they can be accessed as many times as the application requires.

By pulling sections of code out of a main loop and encapsulating them into subroutines, the main code can also be easier to read. Organizing code into multiple subroutines is a good practice.

```
C10             'Place label
  IF IN(0)==0   'Check Input 0
    GOSUB20     'If Input 0 low, call Subroutine 20
  ENDIF         'End check Input 0
  IF IN(1)==0   'Check Input 1
    a=30        'as example for below
    GOSUB(a)    'If Input 1 low, call Subroutine 30
  ENDIF         'End check Input 1
GOTO(10)        'Will loop back to C10
```

IF, ENDIF Conditional Test

Once the execution of the code reaches the **IF** command, the code between that **IF** and the following **ENDIF** will execute only when the condition directly following the **IF** command is true. For example:

```
a=IN(0)     'Variable ‘a’ set 0,1
a=a+IN(1)   'Variable ‘a’ 0,1,2
IF a==1     'Use double = test
  b=1       'Set ‘b’ to one
ENDIF       'End IF
```

Variable **b** will only get set to one if variable **a** is equal to one. If **a** is not equal to one, then the program will continue to execute using the command following the **ENDIF** command.

Notice also that the SmartMotor language uses a single equal sign (**=**) to make an assignment, such as where variable **a** is set to equal the logical state of input **0**. Alternatively, a double equal (**==**) is used as a test, to query whether **a** is equal to 1 without making any change to **a**. These are two different functions. Having two different syntaxes has other benefits.

ELSE, ELSEIF

The **ELSE** and **ELSEIF** commands can be used to add flexibility to the **IF** statement. If it were necessary to execute different code for each possible state of variable **a**, the program could be written as follows:

```
a=IN(0)         'Variable ‘a’ set 0,1
a=a+IN(1)       'Variable ‘a’ 0,1,2
IF a==0         'Use double ‘=’ test
  b=1           'Set ‘b’ to one
ELSEIF a==1
  c=1           'Set ‘c’ to one
ELSEIF a==2
  c=2           'Set ‘c’ to two
ELSE            'If not 0 or 1
  d=1           'Set ‘d’ to one
ENDIF           'End IF
```

There can be many **ELSEIF** statements, but at most one **ELSE**. If the **ELSE** is used, it needs to be the last statement in the structure before the **ENDIF**. There can also be **IF** structures inside **IF** structures. That's called "nesting" and there is no practical limit to the number of **IF** structures that can nest within one another.

The commands that can conditionally direct program flow based on a test, such as the **IF**, where the test may be **a==1**, can have far more elaborate tests inclusive of virtually any number of operators and operands. The result of a comparison test is zero if "false", and one if "true". For example:

```
IF ABS(EA-5)>x      'A numeric test
                    'placing further commands here
ENDIF
IF (a<b)&(c<d)      'A logical test using bit-wise AND
                    'placing further commands here
ENDIF
IF (a==b)|(c!=d)    'A logical test using bit-wise OR
                    'placing further commands here
ENDIF
```

Complex logical tests involving bit-wise AND, OR and Exclusive OR only depend on whether the result of an operation is zero or one. Any test for zero or not zero must be made explicitly.

```
IF (a<b)&c          'This should be avoided and replaced by
IF (a<b)&(c!=0)     'an explicit test of c not zero
```

WHILE, LOOP

The most basic looping function is a **WHILE** command. The **WHILE** is followed by an expression that determines whether the code between the **WHILE** and the following **LOOP** command will execute or be passed over. While the expression is true, the code will execute. An expression is true when it is non-zero. If the expression results in a "zero" then it is false. The following are valid **WHILE** structures:

```
WHILE 1     '1 is always true
  OS(0)     'Set output to 1
  OR(0)     'Set output to 0
LOOP        'Will loop forever

a=1         'Initialize variable 'a'
WHILE a     'Starts out true
  a=0       'Set 'a' to 0
LOOP        'This never loops back
a=0         'Initialize variable 'a'
WHILE a<10  'a starts less
  a=a+1     'a grows by 1
LOOP        'Will loop back 10 times
```

The task or tasks within the **WHILE** loop will execute as long as the loop condition remains true.

The **BREAK** command can be used to break out of a **WHILE** loop, although that somewhat compromises the elegance of a **WHILE** statement's single test point, making the code a little harder to follow. The **BREAK** command should be used sparingly or preferably not at all in the context of a **WHILE**.

If it's necessary for a portion of code to execute only once based on a certain condition then use the **IF** command.

SWITCH, CASE, DEFAULT, BREAK, ENDS

Long, drawn out **IF** structures can be cumbersome, and burden the program visually. In these instances it can be better to use the **SWITCH** structure.

The following code would accomplish the same thing as the **ELSEIF** program example:

```
a=IN(0)    'Variable 'a' set 0,1
a=a+IN(1)  'Variable 'a' 0,1,2
SWITCH a   'Begin SWITCH
  CASE 0
    b=1    'Set 'b' to one
  BREAK
  CASE 1
    c=1    'Set 'c' to one
  BREAK
  CASE 2
    c=2    'Set 'c' to two
  BREAK
  DEFAULT  'If not 0 or 1
    d=1    'Set 'd' to one
BREAK
ENDS       'End SWITCH
```

Just as a rotary switch directs electricity, the **SWITCH** structure directs the flow of the program. The **BREAK** statement then jumps the code execution to the code following the associated **ENDS** command. The **DEFAULT** command covers every condition other than those listed. It is optional.

TWAIT Wait for Trajectory to Finish

The **TWAIT** command pauses program execution while the motor is moving. Either the controlled end of a trajectory, or the abrupt end of a trajectory due to an error, will terminate the **TWAIT** waiting period. If there were a succession of move commands without this command, or similar waiting code between them, the commands would overtake each other because the program advances, even while moves are taking place. The following program has the same effect as the **TWAIT** command, but has the added virtue of allowing other things to be programmed during the wait, instead of just waiting. Such things would be inserted between the two commands.

```
WHILE Bt   'While trajectory
  ...
LOOP       'Loop back
```

WAIT=frm Wait, Pause Program Execution for Time in Milliseconds

There will probably be circumstances where the program execution needs to be paused for a specific period of time. The **WAIT** command will pause the program for the specified number of milliseconds. **WAIT=**1000, for example, would wait one second. The following code would be the same as **WAIT=**1000, only it would allow code to execute during the wait if it were placed between the **WHILE** and the **LOOP**.

```
CLK=0                 'Reset CLK to 0
WHILE CLK<1000        'CLK will grow
  ...
LOOP                  'Loop back
```

STACK Reset the GOSUB Return Stack

The **STACK** is where information is held with regard to the nesting of subroutines (nesting is when one or more subroutines exist within others). In the event program flow is directed out of one or more nested subroutines, without executing the included **RETURN** commands, the stack will be corrupted. The **STACK** command resets the stack with zero recorded nesting. Use it with care and try to build the program without requiring the **STACK** command.

One possible use of the **STACK** command might be if the program used one or more nested subroutines and an emergency occurred, the program or operator could issue the **STACK** command and then a **GOTO** command which would send the program back to a label at the beginning. Using this method instead of the **RESET** command would retain the states of the variables and allow further specific action to resolve the emergency.

```
C1              'Subroutine C1
STACK           'Clear the nesting stack
RUN             'Begin the program, retaining variables
RETURN          'Never reached, but necessary for comp.
```

END End Program Execution

If it is necessary to stop a program, use an **END** command and execution will stop at that point. An **END** command can also be sent by the host to intervene and stop a program running within the motor. The SmartMotor program is never erased until a new program is downloaded. To erase the program in a SmartMotor, download only the **END** command as if it were a new program. That will be the only command that is left in the SmartMotor until a new program is downloaded. To compile properly, every program needs an **END** somewhere, even if it is never reached. If the program needs to run continuously, the **END** statement has to be outside the main loop.

Interrupt Programming

ITR(), ITRE, ITRD, EITR(), DITR(), RETURNI Interrupt Commands

A SmartMotor can be configured to execute a routine upon the change of a status bit using the Interrupt **ITR()** function. There are dozens of different bits of information available in the SmartMotor that are held in groups of 16 bits called “Status Words”. **ITR()** can tell the SmartMotor to execute a subroutine upon the change of any one of these status bits in any Status Word. When the status bit changes, that subroutine will execute at that instant from wherever the normal program happens to be. A program of some sort must be running for the interrupt routine to execute.

Interrupt subroutines are ended with the **RETURNI** command to distinguish them from ordinary subroutines. After the interrupt code execution reaches the **RETURNI** command, it will go back to the program exactly where it was interrupted. An interrupt subroutine must not be called directly with a **GOSUB** command. The **ITR()** function has five parameters:

ITR(Int #, Status Word, Bit #, Bit State, Label #)

Int #:	Interrupt number: there can be eight: 0 to 7
Status Word:	0-8,12,13,16 and 17 (at time of this printing)
Bit #:	0 to 15
Bit State:	The state that causes the interrupt, 0 or 1
Label #:	Subroutine label number to be executed, 0 to 999

For an interrupt to work it must be enabled at two levels. Individually enable an interrupt with the **EITR()** command with the interrupt number, 0 to 7, in the parentheses. Then enable all interrupts with the **ITRE** command. Similarly, individual interrupts can be disabled with the **DITR()** command, and all interrupts can be disabled with the **ITRD** command.

The **STACK** and **END** commands clear the tracking of subroutine nesting, and disable all interrupts. In the following program example, interrupt number zero is set to look at Status Word 3, Bit 15, which is “Velocity Target Reached”. When this status bit switches to 1, subroutine 20 will execute, issuing an X command, stopping the motor. Every time the motor reaches its target velocity, it will immediately decelerate to a stop, causing it to forever accelerate and decelerate without ever spending any time at rest, or at the target velocity.

```
EIGN(2)             'Disable Left Limit
EIGN(3)             'Disable Right Limit
ZS                  'Clear faults
VT=700000           'Set Target Velocity.
ADT=100             'Set Target Accl and Decel.
MV                  'Set Mode Velocity
ITR(0,3,15,1,20)    'Set Interrupt
EITR(0)             'Enable Interrupt zero
ITRE                'Enable All Interrupts
G                   'Start motion
C10                 'Place a label
GOTO10              'Loop..., req. for int. operation
END                 'End (never reached)
```

CAUTION

*The **STACK** and **END** commands disable all interrupts.*

```
C20                 'Interrupt Subroutine
  X                 'Decelerate to stop
  TWAIT             'Hold program until motor reaches stop
  G                 'Restart velocity motion
RETURNI             'Return to infinite loop
```

TMR(exp1,exp2) Timers

The **TMR()** function controls four timers, the states of which can be found in the first four bits of Status Word 4. If it is desired that the interrupt routine execute after a certain period of time, the **TMR()** function can be used. The **TMR()** function has two parameters where:

exp1 Is the Timer #. There are four timers: 0 to 3
exp2 Is the Value in milliseconds, to count down to zero

While the timer is counting down, the corresponding status bit in Status Word 4 will be one. When it reaches zero, the status bit will revert to zero. This bit change can be made to trigger a subroutine using the **ITR()** function.

```
EIGN(2)          'Disable Left Limit
EIGN(3)          'Disable Right Limit
ZS               'Clear faults
MP               'Set Position Mode
VT=500000        'Set Target Velocity.
AT=300           'Set Target Acceleration.
DT=100           'Set Target Deceleration.
TMR(0,1000)      'Set Timer 0 to 1s
ITR(0,4,0,0,20)'Set Interrupt
EITR(0)          'Enable Interrupt
ITRE             'Enable all Interrupts
p=0              'Initialize variable p
O=0              'Set commanded and actual pos. to zero
C10              'Place a label
  IF PA>47000   'Just before 12 moves
    DITR(0)     'Disable Interrupt
    TWAIT       'Wait till reaches 48000
    p=0         'Reset variable p
    PT=p       'Set Target Position
    G           'Start motion
    TWAIT       'Wait for move to complete
    EITR(0)     'Re-enable Interrupt
    TMR(0,1000)     'Re-start timer
  ENDIF
GOTO10           'Go back to label
END              'End (never reached)

C20              'Interrupt Subroutine Label
  TMR(0,1000)   'Re-start timer
  p=p+4000      'Increment variable p
  PT=p          'Set Target Position
  G             'Start Motion
RETURNI         'Return to main loop
```

Error Handling

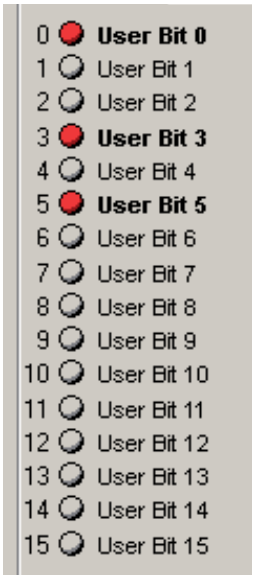
In many multiple-axis applications, if there is a fault in one axis, it is desirable to stop all motion in the machine. An effective way to accomplish this is to place the following example code into every motor. At such time as any axis experiences a drive-disable, interrupt routine **C0** will execute. **C0** will then immediately broadcast a global **MTB**, Mode Torque Brake to stop all axes. After that, the motor calling for the shutdown will place its address in the user accessible mode bits of Status Word zero.

```
EIGN(W,0,12)        'Another way to disable Travel Limits
ZS                  'Clear faults
ITR(0,0,0,0,0)      'Set Int 0 for: stat word 0, bit 0,
                    'shift to 0, to call C0
EITR(0)             'Enable Interrupt 0
ITRE                'Global Interupt Enable
PAUSE               'Pause to prevent "END" from disabling
                    'Interrupt, no change to stack
END

C0                  'Faut handler
  MTB:0             'Motor will turn off with Dynamic
                    'breaking, tell other motors to stop.
  US(0):0           'Set User Status Bit 0 to 1 (Status
                    'Word 12 bit zero)
  US(ADDR):0        'Set User Status Bit "address" to 1
                    '(Status Word 12 Bit "address")
RETURNI
```

After all motors have been stopped, appropriate recovery actions can be taken.

Monitor Status Word 12 in SMI Motor View to see the results of an axis fault.



PAUSE Suspending Program Execution

The **PAUSE** command when executed will suspend the program execution until the **RESUME** command is received. It will not affect the present state of the Interrupt Handler i.e.; if the Interrupt Handler is enabled it will still be enabled after a **PAUSE** and its execution has no affect on the interrupt/sub-routine stack. There is a separate stack for **PAUSE** and will go 10 “resumes“ deep that allow for PAUSES over the communications and within User Program Interrupt routines.

RESUME Resuming from a PAUSE

The **RESUME** command when executed will start program execution from the location following the **PAUSE** command. It is intended to be issued externally over communications and is not allowed to be compiled within a program.

The **RESUME** command does not differentiate where the **PAUSE** came from. i.e. If you have a **PAUSE** in the main program and a **PAUSE** in an interrupt, whichever **PAUSE** is active at the time will be resumed.

Variables are data holders that can be set and changed within the program or over one of the communication channels. Although most of the variables are 32-bit signed integers, there are also eight floating-point variables. All variables are represented by lower-case text. They are stored in volatile memory, meaning that they are lost when power is removed and default to zero upon power-up. If they need to be saved, you must store them in EEPROM, non-volatile memory using the **VST** command.

There are three sets of integer variables, each containing twenty-six 32-bit signed integers and referenced by `a,b,c,…,x,y,z, aa,bb,cc,…,xx,yy,zz` and `aaa,bbb,ccc,…,xxx,yyy,zzz`. There is an additional set of fifty-one 32-bit signed integers in array form, `al[i],i=0…50`. The eight floating point variables are also in array form and referenced by af[i],i=0…7.

a = # Set variable a to a numerical value

a = frm Set variable a to value of a variable or formula

All variables can be used in mathematical expressions assuming *standard hierarchical rules* and using any of the mathematical operations

+ Addition

- Subtraction

***** Multiplication

/ Division

Together with the logical operations

< Less than

> Greater than

== Equal to

!= Not Equal to

<= Less than or Equal to

>= Greater than or Equal to

The following additional integer operations are also supported:

^ Raise to an integer power <= 4

& Bit wise AND (see appendix A)

| Bit wise OR (see appendix A)

!| Bit wise Exclusive OR (see appendix A)

% Modulo

SQRT(x) **I**nteger Square Root (x = integer, where x>=0)

ABS(x) Integer Absolute Value (x = integer)

The following floating point functions are also supported:

FSQRT(x)	Floating Point Square Root(x=float where x >= 0)
FABS(x)	Floating Point Absolute Value (x = float)
SIN(x)	Floating Point Sine (x = float in degrees)
COS(x)	Floating Point Cosine (x = float in degrees)
TAN(x)	Floating Point Tangent (x = float in degrees)
ASIN(x)	Floating Point Arc Sine in degrees on [-90,90], (x = float on the interval [-1,1])
ACOS(x)	Floating Point Arc Cosine in degrees on [0,180], (x = float on the interval [-1,1])
ATAN(x)	Floating Point Arc Tangent in degrees on [-90,90] , (x = float)
PI	Floating Point representation of PI = 3.14159265359…

In any operation, if the input is an integer then the result remains an integer. A result is promoted to a float once the operation includes a float. Functions that require a floating point argument implicitly promote integer arguments to float. In converting a floating point number to an integer result, the floating point number is truncated toward zero. Although the floating point variables and their standard binary operations conform to IEEE-754 double precision, the floating point square root and trigonometric functions only produce IEEE-754 single precision results. Here are some examples:

```
a=(b+c/d)*e       'Standard hierarchical rules apply
a=2^3             'a=8
c=123%12          'Modulo, c=3, remainder of 123/12
b=(-10<a)&(a<10) 'b=0 if “a” not in range,b=1 otherwise
x=ABS(EA)         'Set x to the abs value of pos error
r=SQRT(a)         'if a=64, r=8,  if a=63, r=7
b=af[0]           'if af[0]=1.8,b=1,if af[0]=-1.8,b=-1
af[0]=SIN(57.3)   'Set float var af[0]to sine 57.3 degrees

af[7]=ATAN(af[6])*180/PI      'Set af[7] to arctan result
                              'converted to radians
af[4]=af[3]*(af[1]/af[2]-1)   'Standard hierarchical
                              'rules apply
af[0]=(a+b)/2+3.0             'if a=8 and b=1,
af[0]=7.0af[0]=(a+b)/2.0+3.0  'if a=8 and b=1,
af[0]=7.5af[5]=FSQRT(a)       'if a=63, af[5]=7.937253952
```

An array variable is one that has a numeric index component that allows for the selection of which variable a program is to access. This memory space is highly flexible because it can hold 51 thirty-two bit integers, or 102 sixteen bit integers, or 204 eight bit integers (all signed). The array variables take the following form:

ab[i]=frm Set var. to a signed 8-bit value where index i=0...203

aw[i]=frm Set var. to a signed 16-bit value where index i=0...101

al[i]=frm Set var. to a signed 32-bit value where index i=0...50

The index `i` may be a number, a variable, or an expression.
The same array space can be accessed with any combination of variable types and can be viewed simply as the union of the data type arrays. Just keep in mind how much space each variable takes. We can even go so far as to say that one type of variable can be written and another read from the same space. For example, if the first four eight bit integers are assigned as follows:

```
ab[0]=0
ab[1]=0
ab[2]=1
ab[3]=0
```

they would occupy the same memory space as the first single 32-bit number or the first pair of 16-bit numbers. The order is least significant to most with `ab[3]` being the most significant. Because of the way binary numbers work, this would make the 32 bit variable `al[0]` equal to 65,536, as well as the 16-bit variables `aw[0]` equal to 0 and `aw[1]` equal to 1.

A common use of the array variable type is to set up what is called a buffer. In many applications, the SmartMotor will be tasked with inputting data about an array of objects and to do processing on that data in the same order, but not necessarily at the same time. Under those circumstances it may be necessary to “buffer” or “store” that data while the SmartMotor processes it at the proper times.

To set up a buffer the programmer should allocate a block of memory to it, assign a variable to an input pointer, and another to an output pointer. Both pointers would start out as zero and every time data was put into the buffer the input pointer would increment. Every time the data was used, the output buffer would likewise increment. Every time one of the pointers is incriminated, it would be checked for exceeding the allocated memory space and rolled back to zero in that event, where it would continue to increment as data came in. This is a first-in, first-out or “FIFO” circular buffer. Be sure there is enough memory allocated so that the input pointer never overruns the output pointer.

Every SmartMotor has its own little solid-state disk drive for long term storage of data. It is based on EEPROM technology and can be written to, and read from, more than a million times.

EPTR = expression **Set EEPROM Pointer in Bytes, 0-32,767**

To read or write into this memory space, it is necessary to properly locate the pointer. This is accomplished by setting **EPTR** equal to the offset in bytes. EE locations above **EPTR** equal to 32,339 contain important motor information and are read only.

VST(variable,index) **Store Variables**

To store a series of variables, use the **VST** command. In the "variable" space of the command, put the name of the variable and in the "index" space, put the total number of sequential variables that need to be stored. Enter a one if just the variable specified needs to be stored. The actual sizes of the variables will be recognized automatically. Do not put the **VST** command in a tight program loop or you will likely exceed the 1M write cycles, damaging the EEPROM.

*Keep the **VST** command out of tight loops to avoid exceeding the 1M write cycle limit of the EEPROM.*

VLD(variable,index) **Load Variables**

To load variables, starting at the pointer, use the **VLD** command. In the "variable" space of the command, put the name of the variable and in the "index" space, put the number of sequential variables to be loaded. Again, the actual sizes of the variables will be recognized automatically.

The SmartMotor System Status has been broken up into 16-bit Status Words (see Appendix for the actual break down of each Status Word). Many status bits are predefined and offer information about the state of the SmartMotor operating system or the motor itself. However, there are Status Words that contain User bits and have been set aside for use by the programmer and his or her specific application.

Status Bits may not be cleared or reset if the condition which has set it still exists, for example the **Bh** bit.

Status Bits can be used to cause interrupts within an application program. The state of a Status Bit can also be tested by **IF** and **WHILE** instructions. Therefore Status Bits can determine the flow or path of execution of an application program.

The following are instructions that are to be used in retrieving and manipulating Status Words and Bits.

=**W**(exp)	Gets the 16-bit Status Word.
=**B**(exp1,exp2)	Gets Status Bit, exp1=Word#,exp2=Bit#.
Z(exp1,exp2)	Clears Status Bit, exp1=Word#,exp2=Bit#.
=B@	Gets a Status Bit thru direct addressing, where @ is replaced with a lower case alpha character.
Z@	Clears a Status Bit thru direct addressing, where @ is replaced with a lower case alpha character.
ZS	Is a clear of a defined set of status bits, its intent is to clear the faults of a motor allowing motion to continue from a **G**.

There are many system and motor Status Bits to govern the application program and motor behavior; to follow is a list of the very useful directly addressed Status Bits. It is important to note that the following list is not the full complement of accessible Status Bits for a program to utilize, and it is necessary to refer to Appendix F.

General System Directly Addressed Status Bits:

Bk	Program check sum/EEPROM failure
Bs	Syntax error occurred

General Motor Directly Addressed Status Bits:

Bo	Motor off
Bt	Trajectory in progress
Bw	Position wraparound occurred
Bv	Velocity limit reached.
Ba	Over current state occurred
Bh	Excessive temperature
Be	Excessive position error

Motor Hardware Limits Directly Addressed Status Bits:

Bm	Real time negative hardware limit, “Left” limit.
Bp	Real time positive hardware limit, “Right” limit.
Bl	Historical negative hardware limit, “Left” limit.
Br	Historical positive hardware limit , “Right” limit.

Motor Software Limits Directly Addressed Status Bits:

Bms	Real time negative hardware limit, “Left” limit.
Bps	Real time positive hardware limit, “Right” limit.
Bls	Historical negative software limit, “CCW” limit.
Brs	Historical positive software limit, “CW” limit.

Motor Index/Capture Directly Addressed Status Bits:

Bi(0)	Rising index/capture available on the internal motor encoder.
Bi(1)	Rising Index/capture report available on the external encoder.
Bj(0)	Falling index/capture value available on the internal motor Encoder.
Bj(1)	Falling index/capture value available on the external encoder.
Bx(0)	Hardware index/capture input level internal motor encoder
Bx(1)	Hardware index/capture input level external encoder

If action is taken based on some of the error flags, the flag will need to be reset in order to look out for the next occurrence, or in some cases depending on how the code is written, in order to keep from acting over and over again on the same occurrence.

Za	Reset over current state occurred
Zh	Reset excessive temperature
Ze	Reset excessive position error
Zl	Reset historical left limit occurred
Zr	Reset historical right limit occurred
Zls	Reset historical left limit occurred
Zrs	Reset historical right limit occurred
Zs	Reset syntax error occurred
ZS	Resets all above Z@ status flags, to include **Bi**(#) and **Bj**(#) statuses.

An example of where one would use a System Status Bit would be to replace the **TWAIT** command. The **TWAIT** command pauses program execution until motion is complete but interrupt subroutines will still take place. To avoid a routine to rest on the **TWAIT**, a routine could be written that does much more. To start with, the following code example would perform the same function as **TWAIT**:

```
WHILE Bt        'While trajectory
LOOP            'Loop back
```

Alternatively, the above routine could be augmented with code that took specific action in the event of an index signal as is shown in the following example:

```
EIGN(W,0)           'Set all I/O to be general inputs
a=0
ZS                  'Clear all faults
Ai(0)               'Arm motor's capture register
MV VT=1000 ADT=10   'setup velocity mode, very slow
G                   'start motion
WHILE Bt            'While trajectory
  IF Bi(0)==0       'Check index captured of encoder
    GOSUB(1)        'call subroutine
  ELSE
    X
  ENDIF             'end checking
LOOP                'Loop back
OFF
END
                    'SUB 1: Increment a every 1 second
C1
  IF B(4,0)==0      'check Timer 0 status
    a=a+1           'updating a every second
    TMR(0,1000)     'set Timer 0 counting
  ENDIF
RETURN
END
```

Timer Status Bits

Timer Status Bits are true while a timer is actively counting. Timers have resolution of 1 millisecond.

TMR(0,1000)	Sets Timer Status bit 0 true for 1 second.
=**TMR**(3)	Get the value of Timer 3.

Interrupt Status Bits

Interrupt Status Bits are true if an interrupt is enabled. It is important to note the interrupts need to be configured before being enabled for proper operation. Please refer to the Program Flow section of this manual for interrupt configuration and their use, the below commands are examples which would directly affect the state of the Interrupt Status Bits:

ITRE	Enable interrupts handler, sets Interrupt Status Bit 15.
ITRD	Disable the interrupt handler, clears Interrupt Status Bit 15.
EITR(0)	Enable the highest priority interrupt, sets Interrupt Status Bit 0.
DITR(0)	Disable the highest priority interrupt, clears Interrupt Status Bit 0.

I/O Status

Typically, to get an I/O port logical status the programmer would use the **IN**() instructions (see appropriate section), for zero-based addressing of I/O Ports. As with any status of the SmartMotor you can also retrieve the I/O Port status using **W**() and **B**() Status Word commands, but not change their state.

User Status Bits

User Bits allow for the programmer to keep track of events or status within an application program. There functions are to be defined by the application program of the SmartMotor. User Bits are address individually starting at 0 (zero-based). Likewise the User Bits words are addressed starting at 0 (zero-based).

A power feature of user bits is their ability to be addressed over networks such as COMBITRONIC or CANopen. This can also give a hosting application the ability to cause a SmartMotor to run an interrupt routine.

__G__ also resets several system state flags.

The User Bits can be addressed as words also, with or without a mask to define which bits are affected. Below are examples of commands that directly effect the user bits:

US(0)	SET User Bit 0
US(W,0,a)	SET first three User Bits when a=7
UR(19)	RESET User Bit 3 in second User Bit Status Word
UR(W,0)	RESET all User Bits in first User Status Word
UR(W,1,7)	RESET User bits 16, 17 and 18
UO(0)=a&b	Sets User Bit to 1 if the bit-wise operation result is odd else sets it to 0
UO(W,1,7)=a	Sets User bits 16,17,and 18 to the value of the lower three bits in **a**

Multiple Trajectory Support Status Bits

The SmartMotor System supports the ability to have multiple trajectory generators operating at the same time. The outputs of the generators can be manipulated to affect the SmartMotor in a combination of ways, which are discussed in other section within this manual. These trajectory generator Status Bits help the programmer properly control the use of these generators from the application program or over a network.

The following example exercises the trajectory Status Bits of Word 7 in a Standard Class 5 SmartMotor:

```
EIGN(W,0)
ZS
MFA(1000)   'setup Gearing Profile
MFR(2)      'Gearing Profile for Trajectory Gen.(TG)2
O=0         'Establish actual position to Zero
PT=0        'setup target Position to Zero
VT=1000     'setup Target Velocity
ADT=100     'setup accel and decel
MP(1)       'Position Mode for trajectory generator(TG)1
G(2)        'Start TG 2, TG2 in Progress is ON
G(1)        'Start TG 1, PC=PT so TG 1 in Progress is OFF
PT=10000
G(1)        'Until PC=PT TG1 in Progress is ON
TWAIT(1)
OFF
END
```

Cam Status Bits

The Class 5 SmartMotor supports cams running in either spline and/or Linear Interpolated Position Modes. Each individual Cam Segment can be interpolated in one of these two modes. While the cam is being executed, the Cam Segment Mode bits can be interrogated to determine which mode is presently being used for that segment. The programmer can also turn ON and OFF Cam User Bits which are defined when each segment is written into Cam Memory via the **CTW**() command. Cam User Bits offer a periodic signal based on the phase of a cam, and they can be programmed to come ON or OFF within any given section of the cam. They function much like the industry standard Programmable Limit Switch (PLS). In a Standard Class 5 Motor these bits reside in Status Word 8. The following example will exercise each Cam User Bit during the programmed cam profile.

```
EIGN(W,0)
ZS
CTA(7,0,0)           'Add table into RAM al[0]-al[8]
CTW(0,0,1)           'Add 1st point, Cam UserBit 0 ON
CTW(1000,4000,1)     'Add 2nd point, Cam UserBit 0 ON
CTW(3000,8000,2)     'Add 3rd point, Cam UserBit 1 ON
CTW(4000,12000,132)  'Add 4th, Spline Mode, Cam Bit 2 ON
CTW(1000,16000,136)  'Add 5th, Spline Mode, Cam Bit 3 ON
CTW(-2000,20000,16)  'Add 6th point. Cam Bit 4 ON
CTW(0,24000,32)      'Add 7th point. Cam Bit 5 ON

MC                   'select Cam Mode
SRC(2)               'Use the virtual master encoder.
MCE(0)               'Force Linear interpolation.
MCW(0,0)             'Use table 0 in RAM from point 0.
MFMUL=1              'Simple 1:1 ratio from virtual enc.
MFDIV=1              'Simple 1:1 ratio from virtual enc.
MFA(0) MFD(0)        'Disable virtual enc. ramp-up/ramp-
                     'down sections.
MFSLEW(24000,1)      'Table is 6 segments *4000 encoder
                     'counts each. Specify the second
                     'argument as a 1 to force this
                     'number as the output total of the
                     'virtual master encoder into the cam.
MFSDC(-1,0)          'Disable virtual master (Gearing)
                     'repeat.
G                    'Begin move.
END                  'Obligatory END
```

Interpolation Status Bits

The Class 5 SmartMotor supports Interpolated Position Modes (IP Modes) from data sent over a CANopen network. The same bits supported for cams also exist as separate set of Status Bits when operating in IP Mode. In a standard Class 5 motor, these bits reside in Status Word 8. Assuming Animatics' own SMNC Multi-Axis Contouring Software is used, there is built-in support for these Status Bits.

Motion Mode Status

The Class 5 SmartMotor supports many different motion modes. Keeping them straight can present a challenge.

RMODE, RMODE(arg) Report Motion Mode

The **RMODE** command will report the current active motion mode. Insert an argument to specify move generators 1 or 2. The value returned has the following meanings:

- **7** CANopen Interpolation
- **4** Torque
- **3** Velocity
- **1** Position
- **0** Null, (move generator inactive)
- **-2** Quadrature Follow
- **-3** Step/Direction Follow
- **-4** Cam
- **-5** Mixed

What sets the SmartMotor apart from other Integrated Motors is its ability to go beyond the control of motion, to control an entire machine. The extensive and enormously flexible I/O is a cornerstone to that capability.

Whether it is the standard 7 points of 5V I/O located in the 15-pin D-Sub connector, or the optional 10 points of Isolated 24V I/O located in a circular, M-12 connector, each point of I/O can be used configured as a Digital Input, or Digital Output. Regardless of the I/O setting, the Analog value can also be read. The 5V I/O is push-pull, while the 24V I/O is Sourcing for machine safety reasons.

All I/O are organized into 16-bit Status Words starting at Status Word 16 of the controller, but within I/O commands it is word 0 (zero). The I/O ports are initially inputs at power-up; once the state is set using a discrete output command it then controls the state of the I/O pin.

On-board I/O in any standard Class 5 motor will be in the first I/O Status Word 0. The I/O can be addressed in commands zero based starting with the first I/O number as 0 (zero). There can be as many as 16 On-board I/O ports. Individual motor specifications need to be reviewed to determine the number and physical nature of the I/O. The physical nature of the I/O will address the voltage levels and isolation characteristics of each I/O point.

Expanded I/O in a Class 5 motor will start at I/O Status Word 1, and the first expanded I/O number is then 16. Again, individual motor specifications shall determine the number and physical nature of the expanded I/O.

For all commands listed below:

x can be any variable a to zzz to include arrays also.

exp is the I/O Bit Number or Status Word Number. This can be passed in from a variable.

msk bit-wise mask to screen out certain bits

W refers to "Word", or 16 bits of information

Discrete Input Commands

x=IN(exp)	Gets the state of an I/O bit & puts it in a variable.
x=IN(W,exp)	Gets the state of an I/O word & puts it in a variable.
x=IN(W,exp,msk)	Gets the state of an I/O word after applying a mask.

*For the 5V I/O in the D-Sub connector, **exp** can be 0 - 6 for I/Os 0 - 6. For the 24V I/O, exp is 16 - 25 for the ten I/Os 16 - 25.*

Discrete Output Commands

OS(exp)	SET a single output to logic 1 or ON.
OS(W,exp,msk)	SET multiple outputs at once, applying a bit mask first.
OR(#)	RESET a single output to logic 0 or OFF.
OR(W,exp,msk)	RESET multiple outputs at once, applying a bit mask first.
OUT(exp)=frm	If the bit in expression to the right of the “=” is odd then set I/O ON, else when even or zero turn it OFF.
OUT(W,exp)=frm	Set the I/O group to a value to the right of the “=”.
OUT(W,exp,msk)=frm	Set the I/O group with mask.

Output Condition

=**OC(exp)**	Individual output status; bit is 1 if output is ON and 0 if OFF.
=**OC(W,exp)**	Get output status within a word

Note: Inputs return OFF even if external condition is logic 1 above in the OC() commands.

Output Fault Status Reports 24Volt I/O Only

=**OF(exp)**	Returns the present fault state for that I/O, where: 0 = no Fault , 1 = over current, 2 = possible shorted.
=**OF(S,exp)**	Returns the bit mask of present Faulted I/O points. Where # is the 16-bit word number, 0 is the Controller I/O Status Word 16. If the value is ever greater than zero, then the I/O fault status flag (Controller Status Word 3) is set.
=**OF(L,exp)**	Returns the bit mask of Latched Faulted I/O points. Where # is the 16-bit word number, A read of a 16-bit word will attempt to clear the I/O words latch.

Setting an I/O point to be General Use Input Config.

EIGN(exp)	Sets a given I/O Port to or back to an input with no function attached. In other words, to remove the function of travel limit from I/O port 2, execute the instruction **EIGN(2)**.
EIGN(W,exp)	Set all I/O in a given I/O word back to Input.
EIGN(W,exp,msk)	Set all I/O in a given I/O word back to Input if mask bit is set.

The following example shows multiple I/O functions:

```
EIGN(W,0)          'deactivate default on-board I/O functions

ab[10]=W(16)       'read the status of on-board I/O.
ab[11]=IN(W,0)     'Same as above, so ab[10]=ab[11] assuming
                   'I/O states didn't change.
a=0
WHILE a<4
  ab[a]=IN(a)      'get first 4 I/O states into ab[0]-ab[3]
  a=a+1
LOOP
a=0
```

```
WHILE a<4
  OS(a+4)        'turn ON I/O Ports 4 thru 7.
  a=a+1
LOOP

a=1
OUT(W,1)=aw[0]  'set expansion I/0 to value in aw[0]
OR(W,1,a)       'reset only I/O 16
END

EIGN(W,0)       'remove default on-board I/O functions

ab[10]=W(16)    'read the status of on-board I/O via
                'controllers status word.
ab[11]=IN(W,0)  'same as above, so ab[10]=ab[11] assuming
                'I/O states didn't change.
a=0
WHILE a<4
  ab[a]=IN(a)  'get first 4 I/O states into ab[0] thru
ab[3]
  a=a+1
LOOP
a=0

WHILE a<4
  OS(a+4)      'turn ON I/O ports 4 thru 7.
  a=a+1
LOOP

a=1
OUT(W,1)=aw[0] 'set expansion I/0 to value in aw[0]
OR(W,1,a)      'reset only I/O 16
END
```

Analog Functions of I/O Ports

An I/O port's analog value can be monitored with the following commands and the 24V I/O of a SmartMotor offer more flexibility then the 5V I/O as shown below. All scaled readings are in millivolts. The analog reads can help diagnose wiring issues external to the SmartMotor, as an example; while Ports 4 and 5 are being used as RS-485 the signal bias could be monitored, or say a 5V I/O pin is being driven as an output the analog read can help find a short.

Read Push-Pull 5V I/O

INA(A,exp)	Raw analog read: 10-bit res. 0-32736=0-5VDC
INA(V1,exp)	Scaled voltage reading in millivolts directly where 3456 would be 3.456VDC

*For the 5V I/O in the D-Sub connector, **exp** can be 0 - 6 for I/Os 0 - 6. For the 24V I/O, exp is 16 - 25 for the ten I/Os 16 - 25.*

24Volt I/O Sourcing

INA(A,exp)	Raw analog read: 10-Bit res. 0-32736=0-41.25VDC
INA(V,exp)	Scaled read 24000 to 0, where 15500 is 15.5Volts.
INA(V1,exp)	Scaled read 5000 to 0, where 550 is 0.55Volts.

With the 24V I/O, the V1 and V2 settings focus the 10bits of resolution on the finer voltage spans of 5V and 0.6V respectively.

INA(V2,exp)	Scaled read 600 to 0, where 60 is 0.06Volts.
INA(S,exp)	Sourcing voltage for the I/O port(when output pin).
INA(T,exp)	I/O chip temperature.

Special Functions of I/O Ports

The On-board I/O ports have special functions attached as follows:

Ports 0 and 1	External Encoder Inputs, Brake Output.
Ports 2 and 3	Travel Limit Inputs, Brake Output.
Ports 4 and 5	Communications, Brake Output.
Port 6	Go function, Capture Input, Brake Output.

The Brake output function is some what unique and can be pointed to any one of the On-board I/O or expanded I/O ports.

I/O Ports 0 and 1 – External Encoder Function Commands

Ports 0 and 1 can be wired to a external encoder, it should be noted that for proper counting, the commands **OS()**, **OR()**, and **OUT()** should be avoided for ports 0 and 1. Below are the supporting configuration commands further information can be obtained from the ENCODER AND PULSE TRAIN section.

MF0	Set Enc. Counter to zero, put it in Quadrature Count Mode.
MS0	Set Enc. Counter to zero, put it in Step and Direction Count Mode (default count mode).

I/O Ports 2 and 3 – Travel Limit Inputs

Ports 2 and 3 are defaulted to travel limit inputs. They can be changed to a general purpose I/O points by using the **EIGN()** commands, and then returned to the travel limit function with the following commands:

EILN	Set I/O 2 as negative over travel limit.
EILP	Set I/O 3 as positive over travel limit.

Note that the secondary RS-485 port is non-isolated and not properly biased by the two internal 5k Ohm pullups. It is suitable to talk to a bar code reader or light curtain, but not to cascade motors because of the heavy biasing and ground bounce resulting from variable shaft loading.

I/O Ports 4 and 5 – Communications

Ports 4 and 5 maybe configured to a second communications channel. The main communications channel is 0 and Ports 4 and 5 are associated to commands for communications across channel 1. Below are the supporting configuration commands examples further information can be obtained from the Communications section.

OCHN(IIC,1,N,200000,1,8,D)	Set I/O 4 and 5 for I2C mode.
CCHN(IIC,1)	Close the I2C channel.
OCHN(RS4,1,N,38400,1,8,D)	Set I/O 4 and 5 to RS485 mode.
CCHN(RS4,1)	Close the RS485 channel.

Note: These functions are not supported on the Class 5 IP65 rated motors with On-board 24Volt I/O.

I/O Port 6 - Go Command, Capture Input

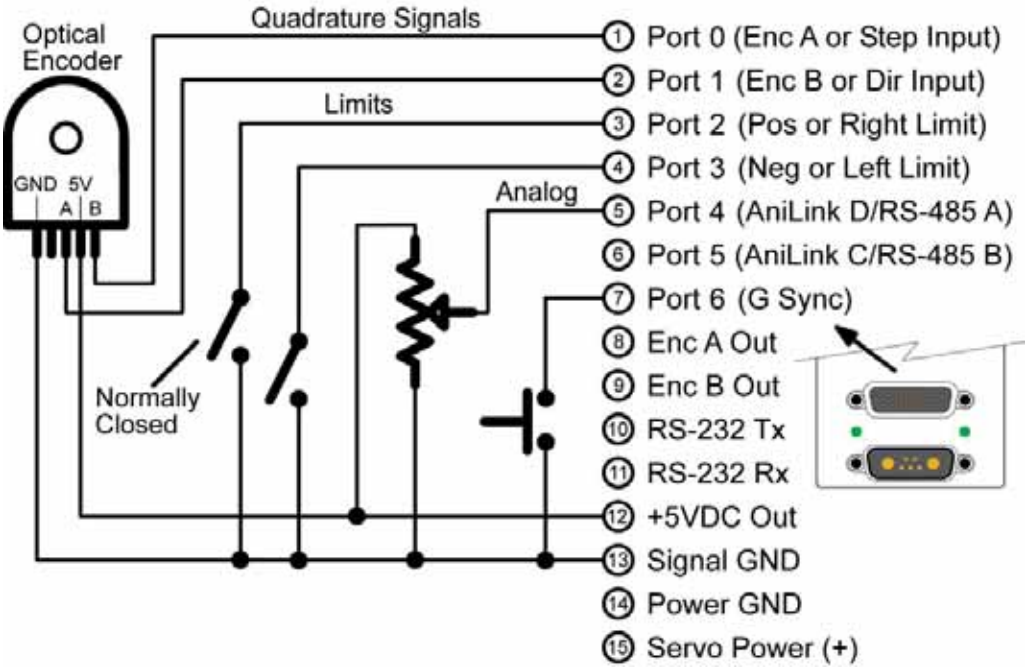
EISM(6) Issues G when specified I/O goes low.
EIRE Index/Registration input capture of the external encoder count (default setting).
EIRI Index/Registration input capture of the internal motor encoder count.

I/O Brake Output Commands

The Brake output function can be configured to any I/O port including the expanded I/O ports where exp is the bit number.

EOBK(exp) Configure a given output to control an external brake.
EOBK(-1) Remove the brake function from the I/O port.

I/O Connection Examples



Encoders, ports, switches and buttons are easy to connect directly to the SmartMotor's I/O pins.

I²C I/O Expansion

I/O ports 4 and 5 can perform as an I²C port. I²C is an "Inter-IC-Communication" scheme that is extremely simple, and yet very powerful insofar as there are dozens of low-cost I²C devices on the market that message over I²C and deliver so many resources. I²C chips include I/O expanders, analog input and output, non-volitile memory, temperature sensors, etc. There is no lower cost way of expanding the functionality of a SmartMotor than by using the I²C resource. Learn more about this capability at the end of the next chapter on Communications.

AniLink, using I²C protocol offers easy digital and analog I/O expansion.

Simply buy I²C chips like the PCF8574A, and the PCF8591.

Functions of I/O Ports

"D" Type SmartMotor Connector Pinouts:

PIN	Main Power	Specifications:	Notes:	Diagram:
1	I/O – 6 "G" command or GP	25mAmp Sink or Source 10Bit 0-5VDC A/D	Redundant connection on I/O connector	**7W2 Combo D-sub Connector**
2	+5VDC Out	50mAmps Max.	Unisolated	
3	RS-232 Transmit	Com(0)	115.2KBaud Max	
4	RS-232 Receive	Com(0)	115.2KBaud Max	
5	Ground	Ground	Unisolated	
A1	Main DC Power Positive	+12.5V min., 48V Max.		
A2	Ground	Main Power Ground		
PIN	**5V I/O Connector**	**Specifications:**	**Notes:**	**Diagram:**
1	I/O – 0 GP or Enc. A or Step Input	25mAmp Sink/Source	1.5MHz max Enc. in.	**DB-15 D-sub Connector**
2	I/O – 1 GP or Enc. B or Dir. Input	25mAmp Sink/Source	1.5MHz max Enc. in.	
3	I/O – 2 Positive Over Travel/GP	25mAmp Sink/Source		
4	I/O – 3 Negative Over Travel/GP	25mAmp Sink/Source		
5	I/O – 4 GP or RS-485 A Com(1)	25mAmp Sink/Source	115.2KBaud Max	
6	I/O – 5 GP or RS-485 B Com(1)	25mAmp Sink/Source	115.2KBaud Max	
7	I/O – 6 "G" command or GP	25mAmp Sink/Source	Redundant connection on Main Pwr Connector	(* All 7 I/O also have 10Bit 0-5VDC A/D)
8	Phase A Encoder Output	25mAmp Sink/Source		
9	Phase B Encoder Output	25mAmp Sink/Source		
10	RS-232 Transmit Com(0)		115.2KBaud Max	
11	RS-232 Receive Com(0)		115.2KBaud Max	
12	+5VDC Out	50mAmps Max		
13	Ground			
14	Ground			
15	Main Power: +12.5VDC to +48VDC	If -DE Option, Control Power separate from Main Power	With -DE option, this becomes separate control power input.	
PIN	**Isolated 24V I/O Connector**	**Specifications:**	**Notes:**	**Diagram:**
1	I/O – 16 GP	150mAmps Max.		**M12, 12-Pin Female End View**
2	I/O – 17 GP	150mAmps Max.		
3	I/O – 18 GP	150mAmps Max.		
4	I/O – 19 GP	150mAmps Max.		
5	I/O – 20 GP	300mAmps Max.		
6	I/O – 21 GP	300mAmps Max.		
7	I/O – 22 GP	300mAmps Max.		
8	I/O – 23 GP	300mAmps Max.		
9	I/O – 24 GP	300mAmps Max.		
10	I/O – 25 GP	300mAmps Max.		
11	+24Volts Input	+12.5V min., 48V Max.	Isolated	
12	GND-I/O	24V I/O ground	Isolated	
PIN	**CAN Connector**	**Specifications:**	**Notes:**	**Diagram:**
1	NC	NC		**M12, 5-Pin Female End View**
2	NC	NC		
3	GND_CAN	CAN ground	Isolated	
4	CAN-H	1M Baud max		
5	CAN-L	1M Baud max		

"M" Type SmartMotor Connector Pinouts:

PIN	Main Power	Specifications:	Notes:	Diagram:
1	Control Power In	18V Min., 32V Max.	Also supplies I/O	M16, 4 Pin Male
2	Chassis			
3	Control, Com, I/O and Amplifier Ground	Common Ground	Unisolated	
4	Amplifier Power In	+12.5V Min., 48V Max.	Powers Amplifier Only	

PIN	Communications Connector	Specifications:	Notes:	Diagram:
1	Control, Com, I/O and Amp Ground	Common Ground	Unisolated	M12, 8-Pin Female End View
2	RS-485 B, Channel 0	115.2KBaud Max	Unisolated	
3	RS-485 A, Channel 0	115.2KBaud Max	Unisolated	
4	Encoder A+ Input/Output	1.5MHz max. as Enc or Step input	Configurable as Encoder Output	
5	Encoder B- Input/Output	1.5MHz max. as Enc or Direction input	Configurable as Encoder Output	
6	Encoder A- Input/Output	1.5MHz max. as Enc or Step input	Configurable as Encoder Output	
7	+5V Out	250mA Max.		
8	Encoder B+ Input/Output	1.5MHz max. as Enc or Direction input	Configurable as Encoder Output	

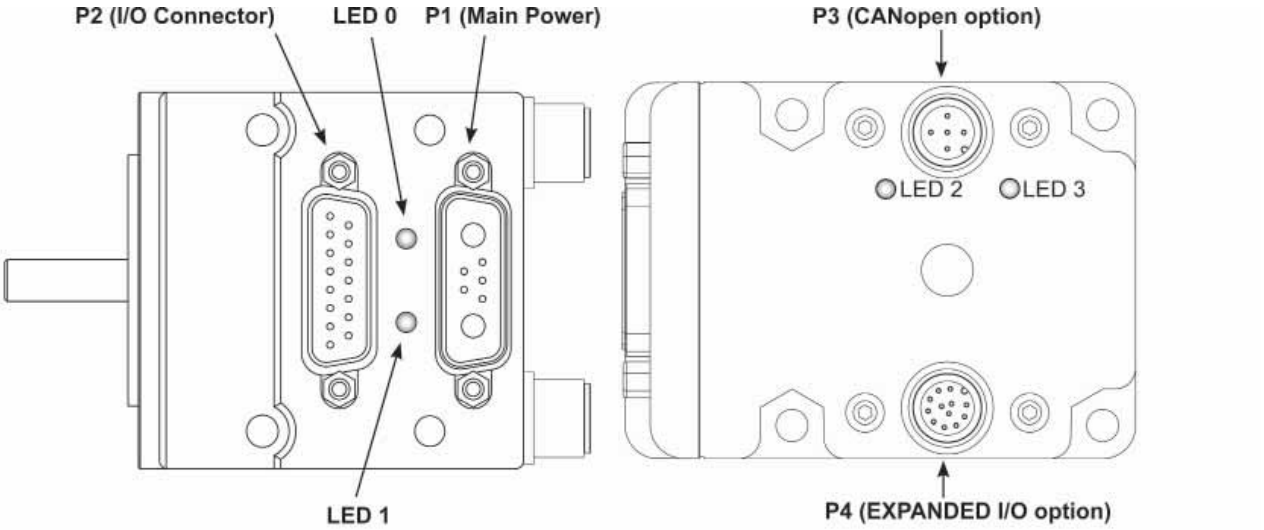
PIN	24V I/O Connector	Specifications:	Notes:	Diagram:
1	I/O – 0 GP	150mAmps Max.	Configurable	M12, 12-Pin Female End View
2	I/O – 1 GP	150mAmps Max.	Configurable	
3	I/O – 4 GP	150mAmps Max.	Configurable	
4	I/O – 5 GP	150mAmps Max.	Configurable	
5	I/O – 6 GP or Go	150mAmps Max.	Configurable	
6	I/O – 7 GP	150mAmps Max.	Configurable	
7	I/O – 8 GP or External Brake	300mAmps Max.	Configurable	
8	I/O – 9 GP	300mAmps Max.	Configurable	
9	I/O – 11 GP No Fault Output	150mAmps Max.	Configurable	
10	I/O – 12 GP Drive Enable Input	150mAmps Max.	Configurable	
11	+24Volts Out	18V Min., 32V Max.		
12	Ground	Common Ground	Unisolated	

PIN	Limit Connector	Specifications:	Notes:	Diagram:
1	+24Volts Out		From Control Pwr In	M12, 5-Pin Female End View
2	-Limit (Input 3)	150mAmps Max.	Configurable	
3	Ground	Common Ground	Unisolated	
4	+Limit (input 2)	150mAmps Max.	Configurable	
5	Input 10	1M Baud max	Configurable	

PIN	CAN Connector	Specifications:	Notes:	Diagram:
1	NC	NC		M12, 5-Pin Female End View
2	NC/Optional Voltage Monitor	NC	DeviceNet Option	
3	Ground	Common Ground	Unisolated	
4	CAN-H	1M Baud max	Unisolated	
5	CAN-L	1M Baud max	Unisolated	

Functions of I/O Ports

LED Functionality:



LED Status Power-up:
with no program
the travel limit inputs are not grounded:
LED0 will be solid RED indicating the motor is in a fault state due travel limit fault.
LED1 will be off

LED Status Power-up:
with no program
and the travel limits are hard wired to ground:
LED0 will be solid red for 500mseconds and then begin flashing Green.
LED1 will be off

LED Status Power-up:
with a program that only disables travel limits and nothing else
LED0 will be solid red for 500mseconds and then begin flashing Green.
LED1 will be off

LED0: Drive Status

OFF	:No Power
Solid Green	:Drive On
Flashing Green	:Drive Off
Flashing Red	:Watchdog Fault
Solid Red	:Major Fault
Alt. Red/Green	:In Boot Load, Needs Firmware

LED1: Trajectory Status

OFF	:Not Busy
Solid Green	:Drive On, Trajectory In Progress

LED2 CAN Bus Network Fault (Red LED)

Off	:No Error
Single Flash.	:At least One Error exceeded Limit
Double Flash	:Heartbeat or Guard Error
Solid	:Busy Off State

LED3: CAN Bus Network Status (Green LED)

Blinking	:Pre-Operational State, (during boot-up)
Solid	:Normal Operation
Single	:Device is in Stopped State

There are basically 5 ways to communicate to a SmartMotor:

1) By using I/O

2) By using I^2C or I^2C devices (AniLink Port)

3) By messaging over RS-485 (AniLink Port)

4) By messaging over RS-232

5) By messaging over CAN - Combitronic, CANopen, DeviceNet, etc.

When using I^2C, the SmartMotor is always the bus master. You cannot communicate between SmartMotors via I^2C.

In early generation SmartMotors, RS-232 was the most common way to talk to one or more SmartMotors. In high axis-count applications both electrical isolation as well as RS-485 conversion to the main RS-232 port was utilized. The pins implementing the SmartMotor's native RS-485 port share I/O functions and are neither isolated, nor properly biased for high axis-count applications. The native RS-485 port is used for linking a small number of SmartMotors, or for talking to intelligent peripherals, such as Light Curtains, Bar Code Scanners, etc.

Today, in applications utilizing more than one SmartMotor, the best choice for communications is to link the SmartMotors together over their optional CAN ports, and then talking to the group through any of the RS-232 or RS-485 ports of any of the motors on the chain. The SmartMotor's new COMBITRONIC© communications over CAN unify all SmartMotor data and functions in a group, making any single motor look like a multi-axis controller from the perspective of any of the RS-232 or RS-485 ports.

Multiple SmartMotors can be connected over optional CAN ports and share resources as though they were one giant, multi-axis controller using Animatics COMBITRONIC© Technology.

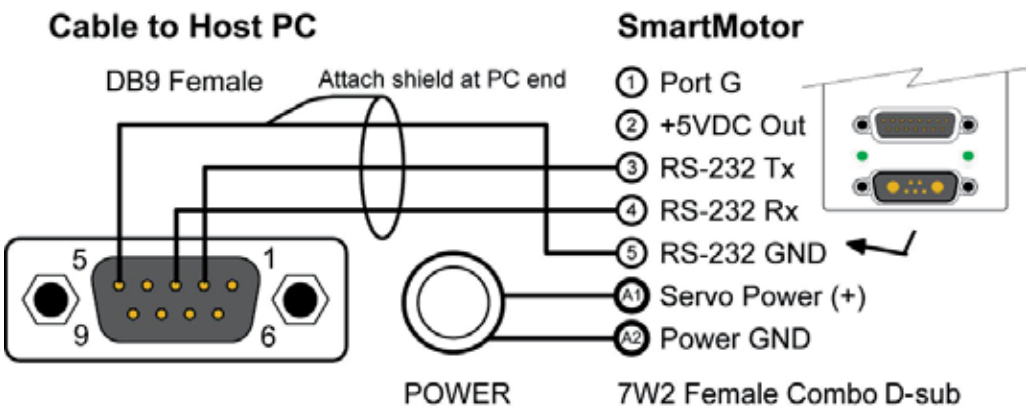
Connecting to a host

While there are a variety of options, the default mode for communicating with a standard Class 5 SmartMotor is serial RS-232 for the main port (except for the Class 5 IP whose main port is RS-485).

To maximize the flexibility of the SmartMotor, all serial communication ports are fully programmable with regard to bit-rate and protocol.

The Class 5 IP65 rated motors, have only one RS-485 and one CAN port.

There is a sixteen-byte input buffer for the primary port and another for the secondary RS-485 port where it exists. These buffers ensure that no arriving information is ever lost, although when either port is in data mode, it is the



*The **CBLSM1-10** makes quick work of connecting to your first RS-232 based SmartMotor.*

Be sure to use shielded cable to connect RS-232 ports together, with the shield ground connected to ground (pin 5) of the PC end only.

responsibility of the user program within the SmartMotor to keep up with the incoming data.

By default, the primary channel, which shares a connector with the incoming power in some versions, is set up as a command port with the following default characteristics:

	Default:	**Other Options:**
Type:	RS-232	RS-485
Parity:	None	Odd or Even
Bit Rate:	9600	2400 to 115200
Stop Bits:	1	0 or 2
Data Bits:	8	7
Mode:	Command	Data
Echo:	Off	On

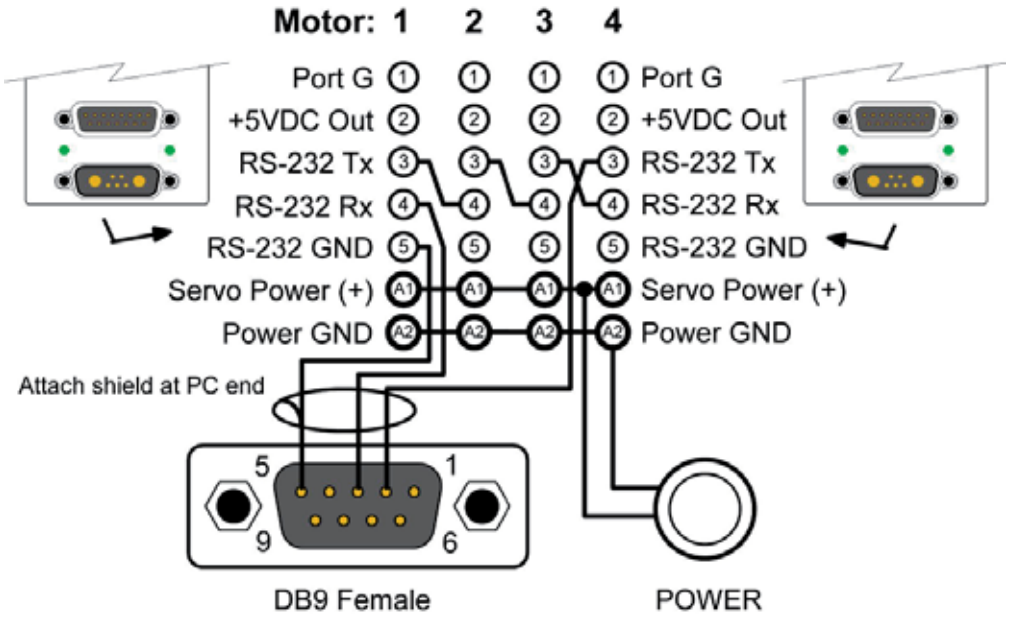
If the cable used is not provided by Animatics, make sure the SmartMotor's power and RS-232 connections are correct.

Buffers on both sides mean there is no need for any handshaking protocol when commanding the SmartMotor. Most commands execute in less time than it would take to receive the next one. Be careful to allow processes time to complete, particularly relatively slow processes like printing to a connected LCD display or executing a full subroutine. Since the **EEPROM** long term memory is slow to write, the terminal software does employ two way communications to regulate the download of a new program.

Daisy Chaining multiple SmartMotors over RS-232

Multiple SmartMotors can be connected to a single RS-232 port as shown. For low-power motors, size SM23165D and smaller, as many as 120 motors could be cascaded using the daisy-chaining technique for RS-232. For independent motion, however, each motor must be programmed with a unique address. In

You can create your own RS-232 daisy chain cable or purchase Add-A-Motor cables from Animatics.



a multiple motor system the programmer has the choice of putting a host computer in control or having the first motor in the chain be in control of the rest.

Fully molded Add-A-Motor cables make quick work of daisy-chaining multiple motors over an RS-232 network.

ADDR= Set Motor to New Address

The **ADDR=** command causes a SmartMotor to respond exclusively to commands addressed to it. It is separate and independent of what may be the motor's CAN address. The range of address numbers is from 1 to 120. Once each motor in a chain has a unique address, each individual motor will communicate normally after its address is sent at least once over the chain. To send an address, add 128 to its value and output the binary result over the communication link. This puts the value above the ASCII character set, quickly and easily differentiating it from all other commands or data. The address needs to be sent only once until the host computer, or motor, wants to change it to something else. Sending out an address zero (128) will cause all motors to listen and is a great way to send global data such as a **G** for starting simultaneous

motion in a chain. Once set, the address features work the same for RS-232 and RS-485 communications.

Unlike the RS-485 star topology, the consecutive nature of the RS-232 daisy-chain creates the opportunity for the chain to be independently addressed entirely from the host, rather than by having a uniquely addressed program in each motor. Setting up a system this way adds simplicity because the program in each motor can be exactly the same. If the **RUN?** command is the first in each of the motor's programs, the programs will not start upon power up. Addressing can be worked out by the host prior to the programs being started later by the host sending the **RUN** command globally.

*The SmartMotor can be made to automatically **ECHO** received characters to the next SmartMotor in a daisy-chain.*

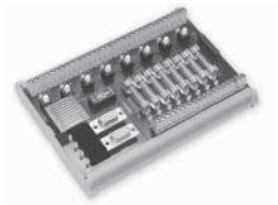
SLEEP, SLEEP1 Assert sleep mode

WAKE, WAKE1 De-assert SLEEP

Telling a motor to sleep causes it to ignore all commands except the **WAKE** command. This feature can often be useful, particularly when establishing unique addresses in a chain of motors. The **1** at the end of the commands specify the AniLink RS-485 port.

CAUTION

Large size 23 or size 34 SmartMotors draw so much power that reliable communications often require isolated communications. For such applications, consider using the Animatics DIN Rail RS-232 fanout:



Animatics offers adapters converting RS-232 to RS-485 and either to USB.

ECHO, ECHO1 **ECHO input**

ECHO_OFF, ECHO_OFF1 **De-assert ECHO**

The **ECHO** and **ECHO_OFF** commands toggle the echoing of data input. Because the motors do not echo character input by default, consecutive commands can be presented, configuring them with unique addresses, one at a time. If the host computer or controller sent out the following command sequence, each motor would have a unique and consecutive address.

If a daisy chain of SmartMotors have been powered off and back on, the following commands can be entered into the SmartMotor Interface to address the motors (0 equals 128, 1 equals 129, etc.). Some delay should be inserted between commands when sending them from a host computer.

0SADDR1
1ECHO
1SLEEP
0SADDR2
2ECHO
2SLEEP
0SADDR3
3ECHO
0WAKE

Commanded by a user program in the first motor, instead of a host, the same daisy chain could be addressed with the following sequence:

```
SADDR1                          'Address the first motor
ECHO                            'Echo for host data
PRINT(#128,“SADDR2”,#13)        '0SADDR2
WAIT=10                         'Allow time
PRINT(#130,“ECHO”,#13)          '2ECHO
WAIT=10
PRINT(#130,“SLEEP”,#13)         '2SLEEP
WAIT=10
PRINT(#128,“SADDR3”,#13)        '0SADDR3
WAIT=10
PRINT(#131,“ECHO”,#13)          '3ECHO
WAIT=10
PRINT(#128,“WAKE”,#13)          '0WAKE
WAIT=10
```

Communicating over RS-485

Multiple SmartMotors can be connected to a single host port by connecting their RS-485 A signals together and B signals together and then connecting them to an RS-485 port or an adapter to RS-232 or USB. Adapters provided by Animatics have built-in biasing resistors, but extensive networks should add bias at the very last motor in the chain. The RS-485 signals of the SmartMotor share I/O functions and are not properly biased for more than just a few SmartMotors. Proper cabling would include a shielded twisted pair for transmission.

The two communications ports RS-232 and RS-485 have enormous flexibility. To select from the vast array of options, use the **OCHN** command.

OCHN(type,channel,parity,bit rate,stop bits,data bits,mode)

	Options:	
type:	RS2, RS4	RS-232 or RS-485
channel:	0, 1 or 2	0=Main, 1=AniLink
parity:	N, O or E	None, Odd or Even
bit rate:	2400,4800,9600,19200,38400,57600,115200 baud	
stop bits:	0, 1 or 2	
data bits:	7 or 8	
mode:	C or D	Command or Data

Here is an example of the **OCHN** command:

```
OCHN(RS4,0,N,38400,1,8,D)
```

If the primary communication channel (0) is opened as an RS-485 port, it will assume the RS-485 adapter is connected to it. If that is the case then pin G in the same connector is assigned the task of directing the adapter to be in Transmit or Receive Mode in accordance with the motor's communication activity and will no longer be useful as an I/O port to the outside.

CCHN(type,channel) Close a communications channel

Use the **CCHN** command to close a communications port when desired.

BAUD#, BAUD(#)=frm Set BAUD rate of main port

The **BAUD** command presents a convenient way of changing only the bit rate of the main channel. The number can be from 2,400 to 115,200 bps.

PRINT( ), PRINT1( ) Print to RS-232 or AniLink channel

A variety of data formats can exist within the parentheses of the **PRINT( )** command. A text string is marked as such by enclosing it between double quotation marks. Variables can be placed between the parentheses as well as two variables separated by one operator. To send out a specific byte value, prefix the value with the # sign and represent the value with as many as three decimal digits ranging from 0 to 255. Multiple types of data can be sent in a single **PRINT( )** statement by separating the entries with commas. Do not use spaces outside of text strings because the SmartMotor uses spaces as delimiters along with carriage returns and line feeds.

The following are all valid print statements and will transmit data through the main RS-232 channel:

```
PRINT("Hello World")    'text
PRINT(a*b)              'exp.
PRINT(#32)              'data
PRINT("A",a,a*b,#13)    'all
```

PRINT1 Prints to the AniLink port with RS-485 protocol.

The Main RS-232 ports of the SmartMotors can be converted to RS-485 and isolated using Animatics adapters.

SILENT, SILENT1 **Suppress PRINT() outputs**

TALK, TALK1 **De-assert Silent Mode**

The **SILENT** mode causes all **PRINT( )** output to be suppressed. This is useful when talking to a chain of motors from a host, when the chain would otherwise be talking within itself because of programs executing that contain **PRINT( )** commands. The **TALK** and **TALK1** commands restore print messaging.

a=CHN(#) **Communication Error Flags**

Where # can be 0 or 1 for COM Channel 0 or 1.

The **CHN(#)** variables hold binary coded information about the historical errors experienced by the two communications channels. The information is as follows:

Bit	Value	Meaning
0	1	Buffer overflow
1	2	Framing error
2	4	Command scan error
3	8	Parity error

A subroutine that printed the errors to an LCD display would look like the following:

```
C9
  IF CHN(0)          'If CHN0 != 0
    IF CHN(0)&1
      PRINT("BUFFER OVERFLOW")
   ENDIF
   IF CHN(0)&2
      PRINT("FRAMING ERROR")
    ENDIF
    IF CHN(0)&4
      PRINT("COMMAND SCAN ERROR")
    ENDIF
    IF CHN(0)&8
      PRINT("PARITY ERROR")
    ENDIF
      Z(2,0)         'Reset CHN0 errors
  ENDIF
RETURN
```

a=ADDR **Motor's Self Address**

If the motor's address (**ADDR**) is set by an external source, it may still be useful for the program in the motor to know to what address it is set. When a motor is set to an address, the **ADDR** variable will reflect that address from 1 to 120.

Getting data from RS-232/RS-485 port using Data Mode

If a Communications port is in Command Mode, then the motor will simply respond to arriving commands it recognizes. If the port is opened in Data

Mode, however, then incoming data will start to fill the 16-byte buffer until it is retrieved with the **GETCHR** command.

a=LEN	Number of characters in RS-232 buffer
a=LEN1	Number of characters in RS-485 buffer
a=GETCHR	Get character from RS-232 buffer
a=GETCHR1	Get character from RS-485 buffer

The buffer is a standard **FIFO** (**F**irst **I**n **F**irst **O**ut) buffer. This means that if the letter **A** is the first character the buffer receives, then it will be the first byte offered to the **GETCHR** command. The buffer exists to make sure that no data is lost, even if the program is not retrieving the data at just the right time.

The **GETCHR** buffer will stop accepting characters if the buffer overflows, and **RLEN** will stop incrementing. Also, the overflow bit will be set for that serial channel. When the buffer is empty, **GETCHR** will return a value of (negative 1.) If **GETCHR** is assigned to a byte ab[], then the value gets cast from the range -1 to +255 to the signed range -128 to +127. This causes -1 (empty buffer) to have the same value as char 255, since 255 gets cast to -1. It is recommended to assign **GETCHR** to a word or long to perform comparisons.

The **LEN** variable holds the number of characters in the buffer. A program must see that the **LEN** is greater than zero before issuing a command like **a=GETCHR**. Likewise, it is necessary to arrange the application so that, overall, data will be pulled out of the buffer as fast as it comes in.

The ability to configure the communication ports for any protocol as well as to both transmit and receive data allows the SmartMotor to interface to a vast array of RS-232 and RS-485 devices. Some of the typical devices that would interface with SmartMotors over the communication interface are:

Other SmartMotors
Bar Code Readers
Light Curtains
Terminals
Printers

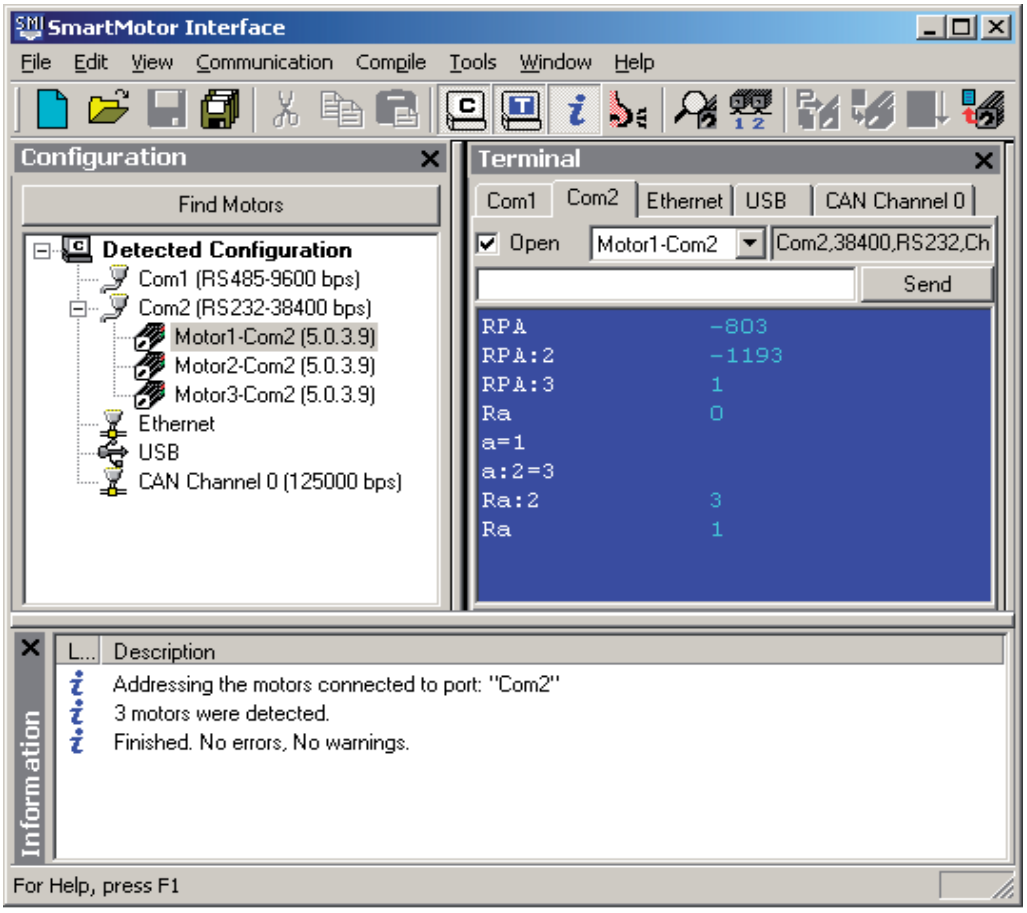
The following is an example program that repeatedly transmits a message to an external device (in this case another SmartMotor) and then takes a number back from the device as a series of ASCII letter digits, each ranging from 0 to 9. A carriage return character will mark the end of the received data. The program will use that data as a position to move to.

```
AT=500                      'Preset acceleration.
VT=1000000                  'Preset velocity.
PT=0                        'Zero out position.
O=0                         'Declare origin
G                           'Servo in place
OCHN(RS2,0,N,9600,1,8,D)
PRINT("RPA",#13)
C0
  IF LEN                    'Check for chars
   a=GETCHR                 'Get char
```

```
      IF a==13                  'If carriage return
         G                      'Start motion
         PT=0                   'Reset buffered P to zero
         PRINT("RP",#13)        'Next
       ELSE
         PT=PT*10               'Shift buffered P
         a=a-48                 'Adjust for ASCII
         PT=PT+a                'Build buffered P
       ENDIF
     ENDIF
  GOTO(0)                       'Loop forever
```

The ASCII code for zero is 48. The other nine digits count up from there so the ASCII code can be converted to a useful number by subtracting the value of 0 (ASCII 48). The example assumes that the most significant digits will be returned first. Any time it sees a new digit, it multiplies the previous quantity by 10 to shift it over and then adds the new digit as the least significant. Once a carriage return is seen (ASCII 13), motion starts. After motion is started, **P** (Position) is reset to zero in preparation for building up again. **P** is buffered so it will not do anything until the **G** command is issued.

The SmartMotor has a wealth of data that can be retrieved over the Combitronic, RS-232, and RS-485 ports simply by asking. Data and status reporting commands can be tested by issuing these report commands from any hosting application. Using SMI Terminal window as the host, (see the example below), the command is shown on the left and the SmartMotor's response is shown in the



middle. The SMI host software uses these commands to implement the Motor View window and Monitor View tools. Data that does not have direct report commands can be retrieved either of two ways, by embedding the variable in a PRINT command, or by setting a variable equal to the parameter and then reporting the variable.

The report commands are listed alphabetically in Appendix C.

It is important to note that Combitronic reports will only work if the CAN network is wired to each motor, and CAN addresses and baud rate configured. Unique addresses should be assigned to each motor with the **CADDR** command. All motors on the same CAN network must be configured to the same baud rate with the **CBAUD** command for proper operation.

CAN Communications

The Class 5 SmartMotor will support different protocols over the CAN port if equipped. CANopen and DeviceNet are popular industrial networks that utilize CAN. If a master is communicating to a group of SmartMotors as slaves under either of these standard protocols, the Combitronic protocol can still function, undetected by the CANopen or DeviceNet master. A CAN network must have all devices set to the same baud rate to operate. The following commands setup and support the CAN port of the motor.

CADDR=frm

Where Number may be from 1 to 127. The setting is stored in the EE, but to take effect the user must cycle power to the motor.

CBAUD=frm

Where frm may be one of the following: 1000000; 800000; 500000; 250000; 125000; 100000; 50000; 20000; 10000

=CAN

The reports a bit map of errors that can happen over the CAN bus where:

Bit	Description
0	CAN bus Pwr Okay(DeviceNet Option)
1	Device is OFFLINE, DeviceNet DupMac Error, bus-OFF
2	reserved
3	reserved
4	User attempted to do Combitronic read from broadcast address
5	Combitronic debug, internal issue.
6	Timeout (Combitronic Client)
7	Combitronic server ran out of buffer slots.
8	Errors reached warning level
9	Receive Errors reached warning level
10	Transmit Errors reached warning level
11	Receive Passive Error
12	Transmit Passive Error

13 Bus Off Error
14 RX buffer 1 overflowed
15 RX buffer 0 overflowed

CANCTL(action, value)

Commands execute based on the action argument to control CAN functions.

Action = 0: Timeout motor action. What the motor does when a timeout in DeviceNet occurs. (DeviceNet firmware only.) Value is 0-9.

0=servo off

1=smooth stop

2=hard stop

3=motor reset

4=no action

5 to 9= **GOSUB**(value) (routines C5-C9)

Action = 1: Reset the CAN MAC and all errors. Resets CANopen stack or DeviceNet stack depending on firmware type. Value is ignored.

Action = 2: Reset the activity of the CANopen clock sync via the high-resolution timestamp. Value is ignored.

Action = 3: Reset the CANopen interpolation buffer via user command. Value is ignored.

Action = 4: Force entry into CANopen Interpolation Mode via user command. Value is 7 to force Interpolation Mode.

Action = 5: Set timeout for Combitronic. Value is in milliseconds, and defaults to 30 for 30 milliseconds.

COMBITRONIC™ Communications

The single most unique feature of SmartMotors are their ability to communicate with each other and share resources using Animatics' own Combitronic technology.

Equipped with an optional CAN port, SmartMotors can not only act as slaves to a CANopen master, but they can also talk to each other. Even while on the same CAN bus, Combitronic communications occur undetected by a CANopen master.

If you have multiple SmartMotor servos, and even if you don't care to communicate with the group over CANopen, you will certainly still want to acquire the CAN port option and let them communicate with each other using the Combitronic feature.

For the greatest possible simplicity, Combitronic communications occur automatically, simply by reference. Take virtually any SmartMotor parameter, add a

colon, followed by a number representing the address of another SmartMotor on the same CAN bus, and that parameter will belong to that SmartMotor. For example, imagine you have three SmartMotors linked together and set with addresses 1, 2 and 3. The following line of code, written in SmartMotor number 2 for example, would set a target position in that same SmartMotor:

```
PT=4000    'Set Target Position in local motor
```

This line of code below, written in SmartMotor number 2, or any of the three motors for that matter, would set a target position in SmartMotor number 3:

```
PT:3=4000 'Set Target Position in motor 3
```

The Combitronic global address for all SmartMotors is zero, so the following line of code, written in ANY SmartMotor would set the target position in ALL SmartMotors, at the same time:

```
PT:0=4000 'Set Target Position in all motors
```

The following line of code could be written in motor number 1 and set variable **a** in motor number 2 equal to an I/O of motor number 3:

```
a:2=IN(0):3   'Set variable in 2 to I/O of 3
```

The possibilities are as endless as is the associated convenience and efficiency. The Combitronic technology creates a true parallel processing environment, but with unparalleled simplicity. The speed of program execution within the SmartMotor, combined with the speed of the Combitronic communications, and the vastness of the SmartMotor's programming language often result in the elimination of the need for a PLC (programmable logic controller). Sensors and valves can be connected to the closest SmartMotor in the machine and be available to the program of any SmartMotor on the network. HMIs (human-machine-interfaces) can connect to any one or more of the SmartMotor's RS-232 or RS-485 ports and provide visibility into the entire network. The size and complexity of the machine collapses to the point where in many cases, there is no longer even a cabinet. The machine builder is spared the traditional bulk, the failure modes, the wiring time and complexity, and the COST of separate servo controllers, servo amplifiers, and PLCs.

Bear in mind that while Combitronic communications are very fast, program execution is also very fast, so if a tight loop is written with a Combitronic transaction inside, you will flood the CAN bus with data, which in the extreme, can slow all operations of all SmartMotors on the chain.

It is simple to avoid this problem. If, for example, motor 1 needs to "poll" the state of an input on motor 2, then instead of writing a tight loop with a Combitronic command in it, simply write a tight loop in motor 2 that would exercise a Combitronic transmission ONLY when that input changes state. The Combitronic command issued in motor 2 could be to set a variable in motor 1 in the event of the input state change. The program in motor A could then simply poll its own internal variable. This way, the actual polling activity is not hammering the CAN bus.

A key to powerful programing in SmartMotors is to exploit the parallel processing for throughput without needlessly wasting throughput through unnecessary polling over the Combitronic interface.

CAUTION

Tight loops with Combitronic commands can flood the CAN bus with data and impare the function of a SmartMotor network. Structure programs to tread lightly on the CAN infrastructure for the best performance.

It is interesting to note that global Combitronic transmissions are especially fast because they do not involve node responses at the protocol level. This fact can be leveraged to speed applications by having certain motors globally broadcast low-frequency, but relevant state changes. For example, if a machine had a "door" and that door could be Open or Closed. The motor performing that function could set every motor's variable "d" equal to 1 when the door is open and 0 when the door is closed like this: **`d:0=1`** and **`d:0=0`**.

Each program in each motor can simply be checking its own variable "d" for the status of the door. By this technique, the programmer has created a new type of variable, a "global" variable.

A further clever way to program a network of SmartMotors is to write ONE program and download that same program to all motors. Then, have the program first look to the motor's CAN address, and execute only that portion of the master program that pertains to that motor address. This makes supporting a large network much easier because there is only one program. Make sure "global" variables as created in the previous example are all unique.

One final step can be taken to further simplify the support of a SmartMotor based machine, and that is to allocate a small group of I/O, or the analog value of an input, to be unique in each motor position by virtue of the wiring leading to that motor. Then have the program set its CAN address in accordance with that unique input status. With this technique, a spare SmartMotor with the master program could replace any failed unit in the system without any special configuration. Even its own address would be automatically set.

CANopen - Can Bus Protocol

CANopen is an industrial CAN bus protocol supported in the standard Class 5 firmware. The protocol supports CIA 402 profile for drives and motion devices. The hosting controller can use an EDS file supplied by Animatics that will easily allow for the control of the SmartMotor over the CANopen network. One of the more powerful features of the CIA 402 profile is Interpolation Mode which is supported by both the Class 5 SmartMotor and Animatics' own coordinated motion software SMNC, and Integrated Motion DLL. The Integrated Motion DLL by itself can offer a host application developer the means to control SmartMotors using CANopen.

DeviceNet - Can Bus Protocol

DeviceNet is an industrial CAN bus protocol supported in the SmartMotor with optional firmware. The protocol supports CIP (Common Industrial Protocol) Profile for a Position Controller. The hosting controller can use an EDS file supplied by Animatics that will easily allow for the control of the SmartMotor using DeviceNet.

I^2C Communications

Maybe the best kept secret of the SmartMotor is its open I2C communications capabilities and the profound potential they have to expand SmartMotor capabilties.

The I2C port is comprised of two signals, the two referred to as ports 4 and 5. These pins are most often shared with the SmartMotor's RS-485 port and a choice must be made between I^2C and RS-485 communications.

OCHN(IIC,1,N,baud,1,8,D)

IIC – the literal syntax IIC to tell what kind of communication this is.

1 – the literal value 1, since this is the location of that port.

N – literal, not relevant to IIC

baud – the bit rate for communication with the IIC device.

1 – literal, not relevant to IIC

8 – literal, not relevant to IIC

D – literal: always in data mode for IIC communication.

CCHN(IIC,1) - closes the channel.

PRINT1(arg1, arg2, … ,arg_n)

Where arg is:

IIS – Start or restart an IIC command. For IIC devices that require a restart, simple call the IIS command a second time within a print.

IIP – stoP an IIC command.

IIGn – get n bytes from the IIC channel (requires the previous commands to have provided whatever addressing or command is required for the device to start sending. The G argument will provide the right number of clock intervals to 'pump' the IIC device to get the data.

RGETCHR1, Var=GETCHR1

Gets the data returned from the IIC device (if available). The data is always in unsigned byte values. So it is advised to assign the data to a 16 or 32-bit register first in order to test for special cases.

For example, the value will be 0-255 for normal data representing all possible values for the byte. If the value from the GETCHR1 command is -1, that indicates the buffer was empty.

RLEN1, Var=LEN1

Gets the amount of bytes in the receive buffer.

There are I^2C devices that perform dozens of purposes including non-volitile memory, high resolution AtoD and DtoA, analog and digital I/O expansion to name just a few. The following example shows how one would utilize a small EEPROM memory device known as the 24FC512. Only the initialization part runs at power-up. Thereafter, subroutines 100 and 200 can be called to write

or read data into the EEPROM.

```
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Class 5 I2C EEPROM Test 00
' Sept 10, 2009
' I2C test for 24FC512 EEPROM on Personality Module
' Address 1010 001 x
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
SADDR1
ECHO
C0
OFF OCHN(IIC,1,N,200000,1,8,D)  'Init I/Os 4 and 5 as IIC port
PRINT(#13,"IIC Port Initialized",#13)
PRINT(#13)
END

C100 'Write variable a at pointer p
  al[0]=a
  al[1]=p
  PRINT(#13)
  PRINT("Load ",al[0]," at pos ",p,#13)
  PRINT1(IIS,#160,#ab[5],#ab[4],#ab[3],#ab[2],#ab[1],#ab[0],I
IP)
  PRINT("Load bytes: ",ab[3],", ",ab[2],", ",ab[1],",
",ab[0],#13)
  PRINT(#13)
RETURN

C200 'Read into variable a at pointer p
  al[1]=p
  PRINT1(IIS,#160,#ab[5],#ab[4],IIP)  'Write memory pointer
  WAIT=1  'Must have small wait to give the write time it needs
  PRINT1(IIS,#161,IIG4,IIP)  'Setup to read four bytes
  WAIT=1  'Must have small wait to give the write time it needs
  ab[3]=GETCHR1
  ab[2]=GETCHR1
  ab[1]=GETCHR1
  ab[0]=GETCHR1
  a=al[0]
  PRINT(#13)
  PRINT("Read bytes: ",ab[3],", ",ab[2],", ",ab[1],",
",ab[0],#13)
  PRINT("Read ",a," at pos ",p,#13)
  PRINT(#13)
RETURN
```

The SmartMotor includes a very high quality, high performance brushless servomotor with extremely powerful rare earth magnets and a stator (the outside, stationary part): a densely wound, multi-slotted electromagnet.

The PID filter is off during Torque Mode.

Controlling the position of a brushless servo's rotor with only electromagnetism working as a lever is like pulling a sled with a rubber band. Accurate control would seem impossible.

The parameters that make it all work are found in the PID (Proportional, Integral, Derivative) control section. These are the three fundamental coefficients to a mathematical algorithm that intelligently recalculates and delivers the power needed by the motor 8,000 times per second. The input to the PID control is the instantaneous desired position minus the actual position, be it at rest, or part of an ongoing trajectory. This difference is called the position error.

The Proportional parameter of the PID control creates a simple spring constant. The further the shaft is rotated away from its target position, the more power is delivered to return it. With this as the only parameter, the motor shaft would respond just as the end of a spring would if it was grabbed and twisted.

While the derivative term usually acts to dampen instability, this is not the true definition of the term. It is possible to cause instability by setting the derivative term too high.

If the spring is twisted and let go, it will vibrate wildly. This sort of vibration is hazardous to most mechanisms. In this scenario, a shock absorber is added to dampen the vibrations, which is the equivalent of what the Derivative parameter does. If a person sat on the fender of a car, it would dip down because of the additional weight based on the constant of the car's spring. It would not be known if the shocks were good or bad. However, if someone jumped up and down on the bumper, it would quickly become apparent whether the shock absorbers were working or not. That's because they are not activated by position but rather by speed. The Derivative parameter steals power away as a function of the rate of change of the overall PID control output. The parameter gets its name from the fact that the derivative of position is speed. Electronically stealing power based on the magnitude of the motor shaft's vibration has the same effect as putting a shock absorber in the system, and the algorithm never goes bad.

Even with the two parameters working, a situation can arise that will cause the servo to leave its target created by "dead weight". If a constant torque is applied to the end of the shaft, the shaft will comply until the deflection causes the Proportional parameter to rise to the equivalent torque. There is no speed so the Derivative parameter has no effect. As long as the torque is there, the motor's shaft will be off of its intended target.

That's where the Integral parameter comes in. The Integral parameter mounts an opposing force that is a function of time. As time passes and there is a deflection present, the Integral parameter will add a little force to bring it back on target with each PID cycle. There is also a separate parameter (KL) used to limit the Integral parameter's scope of what it can do so as not to overreact.

Each of these parameters has its own scaling factor to tailor the overall performance of the PID control to the specific load conditions of any one particular application.

The scaling factors are as follows:

KP	Proportional
KI	Integral
KD	Derivative
KL	Integral Limit

Tuning the PID Control

Refer to the section: SMI Advanced Functions to learn more about the SMI Tuner and how it can help tune the SmartMotor.

The task of tuning the PID control is complicated by the fact that the parameters are so interdependent. A change in one can shift the optimal settings of the others. The automatic utility makes all of the settings easy, but it still may be necessary to know how to tune a servo.

When tuning the motor it is useful to have the status monitor running which will monitor various bits of information that will reflect the motors performance.

In most cases, it is unnecessary to tune SmartMotors. They are factory tuned, and stable in virtually any application.

KP=exp	Set **KP**, proportional coefficient
KI=exp	Set **KI**, time-error coefficient
KD=exp	Set **KD**, damping coefficient
KS=exp	Set **KS**, derivative coefficient
KL=exp	Set **KL**, time-error coefficient limit
F	Update **PID** control

The main objective in tuning a servo is to get **KP** as high as possible, while maintaining stability. The higher the **KP**, the stiffer the system and the more under control it is. A good start is to simply query what the beginning point is (**RKP**) and then start increasing it 10% to 20% at a time. It is a good idea to start with **KI** equal to zero. Keep in mind that the new settings do not take effect until the **F** command is issued. Each time **KP** is raised, try to physically destabilize the system by bumping or twisting it or have a program loop cycling that invokes abrupt motions. As long as the motor always settles to a quiet rest, keep raising **KP**. Of course if the SMI Tuning Utility is being used, it will employ a step function and show more precisely what the reaction is.

As soon as the SmartMotor starts to find it difficult to maintain stability, find the appropriate derivative compensation. Move **KD** up and down until the value is found that gives the quickest stability. If **KD** is too high, there will be a grinding sound. It is not really grinding, but it is a sign to go the other way. A good tune is not only stable, but reasonably quiet. The level of noise immunity in the **KD** term is controlled by **KS**.

The derivative term **KD** requires estimating the derivative of the position error. While the simplest method is a backward difference, **KS**=0, this is inherently noisy. The choices of **KS**=1, 2 and 3 provide increasing levels of noise immunity at the expense of slightly increasing latency in the estimation. Since higher latency will typically result in lower achievable PID loop gains, choose the best compromise between smoothness and tracking performance. The default set-

ting is **KS=1**.

After optimizing **KD**, it may be possible to raise **KP** a little more. Keep going back and forth until there's nothing left to improve the stiffness of the system. After that it's time to take a look at **KI**.

KI, in most cases, is used to compensate for friction; without it the SmartMotor will never exactly reach the target. Begin with **KI** equal to zero and **KL** equal to 1000. Move the motor off target and start increasing **KI** and **KL**. Keep **KL** at least ten times **KI** during this phase.

Continue to increase **KI** until the motor always reaches its target, and once that happens add about 30% to **KI** and start bringing down **KL** until it hampers the ability for the **KI** term to close the position precisely to target. Once that point is reached, increase **KL** by about 30% as well. The Integral term needs to be strong enough to overcome friction, but the limit needs to be set so that an unruly amount of power will not be delivered if the mechanism were to jam or simply find itself against one of its ends of travel.

EL=expression Set Maximum Position Error

The difference between where the motor shaft is supposed to be and where it is actually positioned, is appropriately called the "position error". The magnitude and sign of the error are delivered to the motor in the form of torque after it is put through the PID control. The higher the error, the more out of control the motor will become. Therefore, it is often useful to put a limit on the allowable error after which time the motor will be turned off. Which is why the **EL** command is exists. It defaults to 1,000, but can be set from 0 to 262,143. This feature can be disabled by setting **EL**=-1.

There are still more parameters that can be utilized to reduce the position error of a dynamic application. Most of the forces that aggravate a PID loop through the execution of a motion trajectory are unpredictable, but there are some that can be predicted and further eliminated preemptively.

__KV__ and __KA__ have no effect in Sleep Modes or Follow Modes.

KG=expression Set KG, Gravity Offset Term

The simplest of these is gravity. Why burden the PID loop with the effects of gravity in a vertical load application if it can simply be weeded out? If in a particular application motion would occur with the power off due to gravity, a constant offset can be incorporated into the PID control to balance the system. **KG** is the term. **KG** can range from -8,388,608 to 8,388,607. To tune **KG**, simply make changes to **KG** until the load equally favors upward and downward motion.

KV=expression Set KV, Velocity Feed Forward

Another predictable cause of position error is the natural latency of the **PID** loop itself. At higher speeds, because the calculation takes a finite amount of time, the result is somewhat "old news". The higher the speed, the more the actual motor position will slightly lag the trajectory calculated position. This can be programmed out with the **KV** term. **KV** can range from zero to 65,535. Typical values range in the low hundreds. To tune **KV** simply run the motor at a constant speed, if the application will allow, and increase **KV** until the error

gets reduced to near zero and stays there. The error can be seen in real time by activating the Monitor Status window in the SMI program.

KA=expression Set KA, Acceleration Feed Forward

Force equals mass times acceleration. If the SmartMotor is accelerating a mass, it will be exerting a force during that acceleration. This force will disappear immediately upon reaching the cruising speed. This momentary torque during acceleration is also predictable and need not aggravate the **PID** control. Its effects can be programmed out with the **KA** term. **KA** can range from zero to 65,535. It is a little more difficult to tune **KA**, especially with hardware attached. The objective is to arrive at a value that will close the position error during the acceleration and deceleration phases. It is better to tune **KA** with **KI** set to zero because **KI** will address this constant force in another way. It is best to have **KA** address 100% of the forces due to acceleration, and leave the **KI** term to adjust for friction.

The PID rate of the SmartMotor can be slowed down.

PID1	Set highest **PID** update rate, 16 KHz
PID2	Divide highest **PID** update rate by 2, default of 8 KHz
PID4	Divide highest **PID** update rate by 4, 4 KHz
PID8	Divide highest **PID** update rate by 8, 2 KHz

A reduction in the PID rate can result in an increase in the SmartMotor™ application program execution rate.

The trajectory and **PID** control calculations occur within the SmartMotor at the "sample rate" selected by the **PIDn** command. Although 16 KHz is available, 8 KHz corresponding to **PID2** is the default, providing a reasonable compromise between very good control and the SmartMotor application program execution rate. This program execution rate can be increased by reducing the PID rate using PID4 and PID8 in applications where the lower "sample rate" still results in satisfactory control.

*Providing proper care is taken to keep the PID filter stable, the **PID#** command can be issued on-the-fly.*

If the **PID** rate is lowered, keep in mind the "sample rate" is the basis for velocity values, acceleration values and the **PID** coefficients. If the rate is cut in half, expect to do the following to keep all else the same:

Double Velocity

Increase Acceleration by a factor of 4

Current Limit Control

AMPS=expression Set Current Limit, 0 to 1023

In some applications, if the motor is misapplied at full power, the attached mechanism could be damaged. It can be useful to reduce the maximum amount of current available thus limiting the torque the motor can produce. Use the **AMPS** command with a number, variable, or expression within the range of 0 to 1023, where the value 1023 corresponds to the maximum commanded torque to the motor. Current is controlled by limiting the maximum PWM duty cycle which for this reason will reduce the maximum speed of the motor as well. The **AMPS** command has no effect in Torque Mode.

Follow Mode (Electronic Gearing)

Follow Mode allows a motor to follow a standard TTL quadrature external encoder input signal, or internal clock, at a user defined ratio.

By default, Follow Mode runs continuously at a ratio of 1:1 in terms of input counts to distance moved.

The user can freely select either the external encoder or fixed rate internal clock as the input source. The fixed rate internal clock runs at 8000 counts per second by default, but can be influenced by the PID commands.

SRC(exp) Select the input source used in Follow and Cam Modes.

The **SRC()** command can allow the SmartMotor to use the many advanced following and camming functions even without an external encoder input. Values for exp:

1 external encoder (**-1** inverts direction)

2 time-base at PID rate (**-2** inverts direction)

MFR Config. A & B inptus to Quadrature Mode and sel. Follow Mode.

The Mode Follow Ratio command configures the A and B inputs of the motor to be read as standard quadrature inputs and puts the SmartMotor in Follow Mode.

MSR Config. A & B inputs to Step/Dir. Mode and sel. Follow Mode.

The Mode Step Ratio command configures the A and B inputs of the motor to be read as standard Step and Direction inputs and puts the SmartMotor in Follow Mode.

MF0 Exit Follow Mode

MS0 Exit Step/Direction Follow Mode

Be careful if using the **ENC1** command that you do not inadvertently change the operation of the encoder with one of these commands. If it is required to use an alternate encoder source for Follow Mode *and* at the same time use **ENC1**, then choose the version of the above commands to match your encoder type for the **ENC1** command.

MFMUL=, MFDIV= Set Follow Mode Ratio

The internal mathematics work best by describing the follow ratio in terms of a fraction of two integers. Choose **MFMUL=** and **MFDIV=** to create the following ratio if it is not 1:1. Note that **MFR** or **MSR** must be issued after any changes to **MFMUL=** or **MFDIV=**

```
MFMUL=frm       'Multiplier applied to follow ratio.
MFDIV=frm       'Divisor applied to follow ratio.
```

MFA(exp1[,exp2]) Ascend ramp to sync. ratio from ratio of 0.

Exp1 Setting from 0 to 2,147,483,647. Set to 0 to disable. By default, it is disabled.

Exp2 Is optional and specifies the meaning of **exp1**. Values of **exp2**: 0 for designating input units (master units) and to 1 for designating distance traveled (slave units).

MFD(exp1[,exp2]) Descend ramp from ratio to ratio of 0

Exp1 Setting from 0 to 2,147,483,647. Set to 0 (default) to disable.

Exp2 Is optional and specifies the meaning of **exp1**. Values of **exp2**: 0 for designating input units (master units), 1 for designating distance traveled (slave units).

*Changed MF values do not take effect until after the next **G** command.*

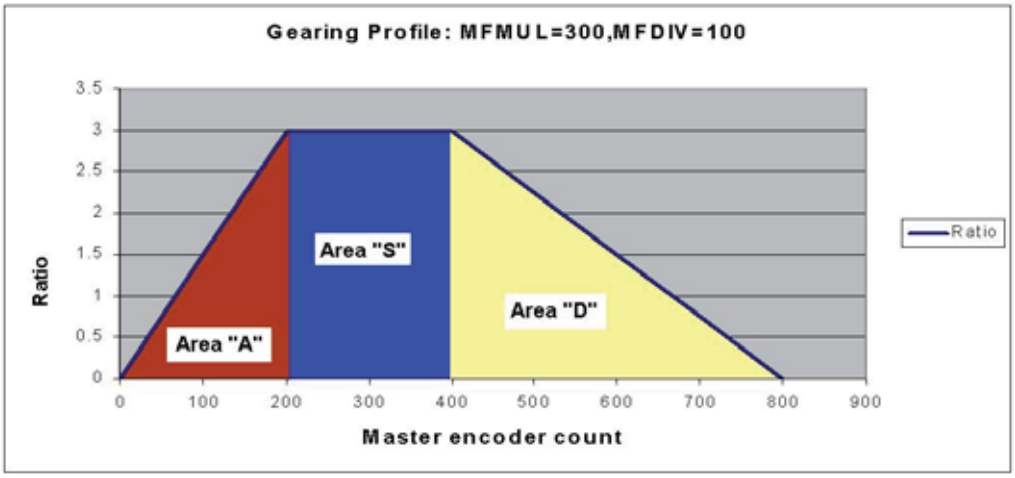
MFSLEW(exp1[,exp2]) Slew at ratio for a fixed distance

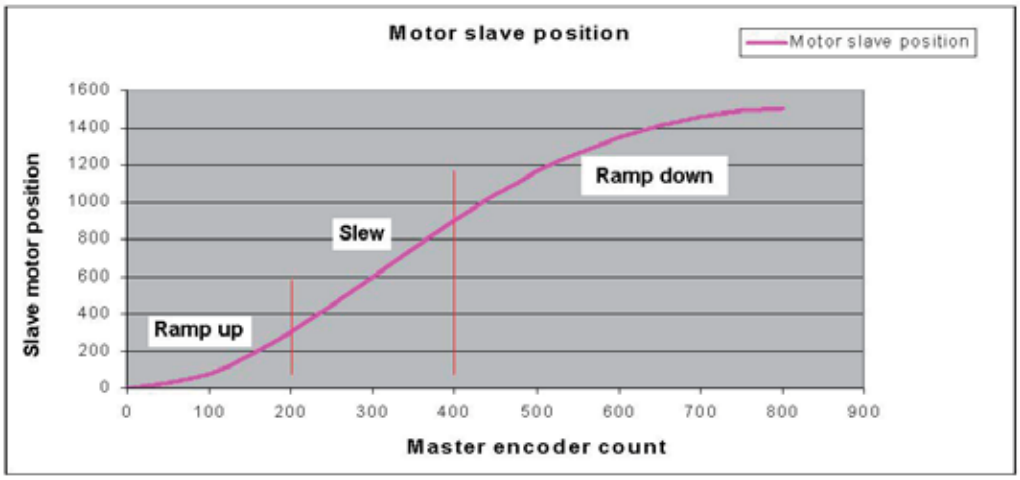
Exp1 Setting from -1 to 2,147,483,647. Set to -1 (default) to disable. When disabled, Follow Mode runs at ratio continuously.

Exp2 Is optional and specifies the meaning of **exp1**. Values of **exp2**: 0 for designating input units (master units), 1 for designating distance traveled (slave units).

The following example shows a profile driven by an incoming encoder signal where in addition to following the incoming encoder, the SmartMotor will perform an acceleration into the following relationship, and after a prescribed distance, perform a deceleration back to rest. In the first graph below, the master encoder signal is along the horizontal axis of the graph, and the gear ratio is the vertical axis of the graph. This demonstrates that 'area under the curve' is the slave position. the second graphs shows the slave position as a function of master position. In this example, **MFA**, and **MFD** are commanded in slave units, and **MFSLEW** is commanded in master units. These 3 commands can accept either master or slave units according to the second argument as a 0 or 1, respectively. The firmware automatically calculates the move accordingly.

```
MFMUL=300
MFDIV=100
MFA(300,1)      'Slave moves 300 counts over ascend
MFD(600,1)      'Slave moves 200 counts over descend
MFSLEW(200,1)   'Slave maintains sync ratio for 300 counts
MFR
G
```





MFSDC(exp1,exp2) Dwell at 0 ratio for input distance

Exp1 Set from 0 to 2,147,483,647 to specify the number of master counts the slave dwells at zero ratio. Set to -1 (default) to dis able. When disabled, Follow Mode runs at ratio continuously.

Exp2 Set to 0 to repeat the gearing profile in the same direction. Set to 1 to repeat the gearing profile in the opposite direction.

A setting of 0 for **Exp 2** is typical for feeding labels in label applications and a setting of 1 is typical for traverse-and-takeup spool winding applications.

The next example demonstrates the use of **MFSDC**. It is a spool winding program that will perform a following profile accross the spool, dwell at the end for a specific span of input distance and then reverse the profile back to the original end of the spool for another dwell. The motion will go back and forth with dwells at each end until another **MFSDC** command is issued with **Exp1** equal to **-1** followed by a **G** command, or an **X** or **S** is issued.

CAUTION

*Any value other than **-1** for **MFSDC Exp1** command will cause the motion profile to continuously dwell and repeat. Reissue the command with **Exp1** equal to **-1** to stop the repetitive motion.*

*Note that the **G** command will assume the cycle starts at one end of the spool.*

```
a=1000       'ascend and descend distance in slave counts
b=200000     'spool width in slave counts
c=4000       'one rev of spool in master counts
s=b-(a*2)    'calculate MFSLEW distance
m=1000       'gear ratio multiplier
d=1000       'gear ratio divisor
MFMUL=m      'set ratios for gearing
MFDIV=d
MFA(a,1)     'set ascend into ratio distance
MFD(a,1)     'set descend out of ratio distance
MFSLEW(s,1) 'set slew dis. btwn the accel and decel points
MFSDC(c,1)  'set dwell for "c" cnts, auto rev. after dwell
MFR          'set mode to electronic gear ratio
G            'start follwing the external master encoder
```

Cam Mode

The Class 5 SmartMotor supports motion profiles based on data stored in a cam table. The cam table can reside in EEprom memory, or in the user array. Multiple tables can be created in the EEprom non-volatile storage. The motor position is interpolated between each data point. This interpolation can be specified as linear, spline that is not periodic, and spline that is periodic.

Cam Mode has the ability to apply sophisticated shaping and selection of the encoder input source using Follow Mode. Cam mode uses **MFMUL** and **MFDIV** to set the ratio incoming master counts. Through the use of the **SRC** command either the external encoder or a fixed-rate 'virtual encoder' can be used as the input source to the cam. This fixed-rate encoder also works through the Follow Mode, so the actual rate into the cam can be set. The speed of the virtual encoder is equivalent to the PID rate. In other words for a Class 5 motor at its default PID rate of 8000 Hz, the virtual encoder will see 8000 encoder counts per second.

Cam tables can be written to EEprom memory that stays when power is removed, or to the variable data space, which goes away with loss of power, but is more dynamically flexible.

Cam Mode Commands

CTE(exp) Erase tables in EE memory starting at the value specified.

To erase all EE tables, choose **CTE(1)**. By choosing a number higher than 1, lower table numbers can be preserved. If for example there were 3 tables stored, **CTE(2)** would erase table 2 and 3, but not table 1. **CTE(0)** is not defined.

CTA(exp1,exp2[,exp3]) Add a Cam table.

The **CTA** command is what you use to setup a table to either EEprom memory, or the data variable space in preparation for writing the table with teh **CTW** command.

Exp1 Specifies the number of points in the table.

Exp2 Specifies the encoder distance between each point. If exp2 is set to 0, then the distance is specified per data record.

Exp3 Is optional and specifies if this is a table in user variables, or EE. By default if exp3 is omitted, then EE is chosen. If exp3 is a 0, then the user array location is chosen (al[0] through al[50].) Only one table can exist in the user variables. Up to 10 tables (numbered 1 through 10) can exist in EE location.

CTW(exp1[,exp2][,exp3]) Write a Cam table

The **CTW** command writes to the table addressed by the most recent **CTA** command. **CTW** writes to either the EE stored tables, or the user array stored tables.

*More typically the actual Cam Table would not be part of the program that executes the mode. **SMI** tools are available to facilitate Cam Table generation.*

Exp1 Is the position coordinate of the motor for that data point. Please set the first point in the table to 0 to avoid confusion. When the table is run, the currently commanded motor position seamlessly becomes the starting point of the table. By keeping the first point of the table at 0, it is easier to realize that all of the data points are relative to this starting point.

Exp2 Is not required if this is a fixed-length cam table (specified in the **CTA** command.) If this cam table was specified as variable length in the **CTA** command, then exp2 is required for each data point. Exp2 represents the absolute distance of the encoder source beginning from the start of the table. For reasons similar to exp1, exp2 should also be 0 for the first data point specified.

Exp3 Is completely optional; this is used to specify additional information about the segment between the specified point and the previous point. The details of this are not covered here, and omitting this point will provide reasonable defaults.

When loading Cam tables, it is important to be mindful of the table capacity. When a cam table is stored in user array memory (al[0]-al[50]), 52 points can be stored as fixed-length segments. 35 points are possible when variable length segments are used.

When tables are written to EEprom memory, significantly more data can be written. For fixed length cams, there is space for at least 750 points. For variable length cam segments, at least 500 points can be written.

MCE(arg) Cam table interpolation mode

The **MCE(arg)** command sets up the Cam function and defines the behavior based on the following arguments:

0 Force linear motion for all sections

1 Spline mode with non-periodic data at ends of table

2 Spline mode with periodic data wrapped at ends of table

MCW(arg1,arg2) Cam table starting point

The **MCW()** command determines where to start the Cam function with **arg1** being the table number and **arg2** being the starting record, being as early as record 0 to as late as the last record in the table.

RCP Read cam pointer

The **RCP** command will report the cam pointer and the **CP** variable can be used by the user program.

RCTT Read number of cam tables

The **RCTT** command will report the number of cam tables and the **CTT** variable can be used by the user program.

MC Enter Cam Mode

The **MC** command enters Cam Mode and must be issued before the **G** command.

Cam Example Program:

```
CTE(1)          'Erase all EE tables.
CTA(7,4000)     'Create a 7-point tbl. at each 4k enc. inc.
CTW(0)          'Add 1st point.
CTW(1000)       'Add 2nd pt. Go to point 1000 from start.
CTW(3000)       'Add 3rd pt. Go to point 3000 from start.
CTW(4000)       'Add 4th pt. Go to point 4000 from start.
CTW(1000)       'Add 5th pt. Go to point 1000 from start.
CTW(-2000)      'Add 6th pt. Go to point -2000 from start.
CTW(0)          'Add 7th point. Return to starting point.
                'Table has now been written to EE.
SRC(2)          'Use the virtual encoder.
MCE(0)          'Force linear interpolation.
MCW(1,0)        'Use table 1 from point 0.
MFMUL=1         'Simple 1:1 ratio from virtual encoder.
MFDIV=1         'Simple 1:1 ratio from virtual encoder.
MFA(0) MFD(0)   'Disable virtual encoder rampup/
                'rampdown sections.
MFSLEW(24000,1)'Table is 6 segments * 4000 encoder
                'counts each.
                'Specify the second argument as a 1 to
                'force this number as the output total of
                'the virtual encoder into the cam.
MFSDC(-1,0)     'Disable virtual encoder profile repeat.
MC              'Enter Cam Mode
G               'Begin move.
END
```

O(arg)=value Set move generator origin to value

The **O()=** command sets move generator origins based on the following arguments:

0 Set the origin of the global move generator (sets value of PA)

1 Set the origin of move generator 1 (sets value of PC(1))

2 Set the origin of move generator 2 (sets value of PC(2))

OSH(arg)=value Shift move generator origin to value

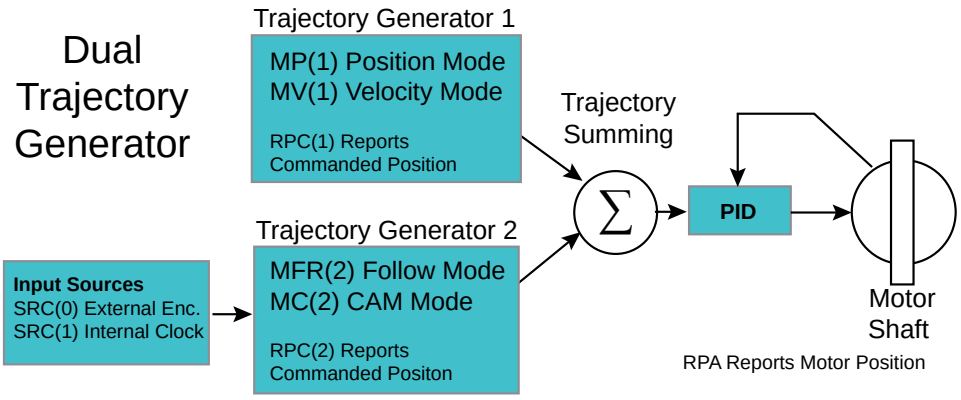
The **OSH()=** command shifts the move generator origins based on the following arguments:

0 Shift the origin of the global move generator (sets value of PA)

1 Shift the origin of move generator 1 (sets value of PC(1))

2 Shift the origin of move generator 2 (sets value of PC(2))

Multiple Trajectories



It is possible to create two trajectories that run concurrently. There are restrictions on which combinations of moves are possible. A combined move consists of a selection from column 1 and a selection from column 2. Note that Torque Mode cannot be combined with any other mode, selecting Torque Mode replaces any other mode currently running.

Trajectory 1 - PC(1)	Trajectory 2 - PC(2)
Position - MP(1)	
Velocity - MV(1)	
CANopen Interpolation	
	Follow - MFR(2)
	Cam - MC(2)
Torque Mode overrides all other modes	

Commands that are selectable to act on only one of these options or both can be provided with a parameter:

MP(exp)	exp = 1 only
MV(exp)	exp = 1 only
MFR(exp)	exp = 2 only
MSR(exp)	exp = 2 only
MC(exp)	exp = 2 only
G(exp)	exp = 1 or 2
X(exp)	exp = 1 or 2
S(exp)	exp = 1 or 2
TWAIT(exp)	exp = 1 or 2
=PC(exp)	exp = 1 or 2
RPC(exp)	exp = 1 or 2
O(exp)=	exp = 1 or 2
OSH(exp)=	exp = 1 or 2
RMODE(exp)	exp = 1 or 2

In the following program, the SmartMotor will move to its origin and then instantly start gearing to an external encoder. It will then perform a relative move on top of the gearing relationship, with the relative move governed by the **VT=** and **ADT=** limits.

```
MP(1)       'Choose position mode from column 1
MFR(2)      'Choose follow mode from col. 2 at same time
PT=0        'This command only relevant to position move
VT=100000   'This command only relevant to position move
ADT=10      'This command only relevant to position move
MFMUL=1     'This command only relevant to follow mode
MFDIV=1     'This command only relevant to follow mode
G(1)        'Position move starts
TWAIT(1)    'Wait for position move only
G(2)        'Start Follow Mode
WAIT=1000   'Wait for one second
PRT=1000    'Prepare for a relative move on top of
            'the Follow Mode
G(1)        'Execute the relative position move
TWAIT(1)    'Wait for position move to finish
PT=0        'Command position 0 of the position mode's
            'frame of reference
G(1)
S           'Stop all moves (follow and position.)
MP          'Position mode exclusively.  Note there is no
            'parentheses  to make only position mode
PT=0
G           'Motor returns to position 0 in terms of actual
            'position because this is only position mode
```

Reading Trajectory Information

=VT	Read back what has been set as the velocity target.
=PT	Read back what has been set as the position target.
=PRT	Read back what has been set as the position relative target.
=AT	Read back what has been set as the acceleration target.
=DT	Read back what has been set as the deceleration target.
=PC	Commanded position of the motor shaft as a result of motion trajec tory generation. This may include a sum of concurrent moves such as a follow mode move combined with a position move.
=PC(0)	equivalent to a=PC
=PC(1)	Reports the commanded position in trajectory generator 1's frame of reference.
=PC(2)	Reports the commanded position in trajectory generator 2's frame of reference.
=VC	Real-time commanded velocity from all trajectory generators.
=AC	Real-time commanded acceleration from all trajectory genera tors (negative indicates deceleration.)
=VA	Report the velocity of the motor shaft. The units are encoder counts per PID sample times 65,536. The factor of 65,536 allows for finer resolution of slower speeds. Please note that this finer resolution information below 65,536 is calculated via a filter since the only direct measurement is whole units of encoder counts per sample.
VAC(exp)	Controls the filter used to measure speed. Default value is 65,000. Higher values provide a smoother filter, at the cost of a longer settling time. The maximum value is 65,535. A value of 0 turns off this filtering.
=PA	Reads position of the motor shaft based on the encoder chosen by ENC1,ENC0 commands.
=CTR(0)	Position of the internal encoder. Unaffected by ENC1,ENC0 commands.
=CTR(1)	Accumulated counts from the external encoder. Unaffected by ENC1, ENC0 commands.

Modulo Position

The actual position of the motor can be reported through a modulo count in addition to the usual absolute count. This count allows for applications where position is cyclical such as a rotary table.

PML=frm	Set the modulo limit. The modulo counter will report between 0 and this value minus one. By default, this is 1,000. The mini-mum value is 1,000. This command resets the modulo count to 0.
=PML	Reads the modulo limit currently set.

PMT=frm Set a target for a position move in terms of a modulo posi tion. The motor will take the shortest path to reach this value. This means that the motor may move in either a clockwise or counter-clockwise direction, depending on which one produces the shortest motion in modulo terms.

=PMT Reads what was set for the modulo target.

=PMA Reads actual motor position in modulo count. This counter is affected by the **O=** and **OSH=** commands.

ENC0 Uses internal encoder for commanded motion, actual position reporting, modulo position reporting.

ENC1 Uses external encoder for commanded motion, actual position reporting, modulo position reporting. Be sure that the correct encoder type is selected with the **MF0** (for quadrature), or **MS0** (step and direction) is chosen.

Postion Error Limits

EL=frm Set position error limit.
=EL Read back position error limit.
=EA Read position error(Commanded - Actual Positions).

DE/Dt Limits

=DEA Read back the actual rate of change of the PID position error. This value is averaged over 4 consecutive PID cycles, and is in units of position error per PID cycle *65,536.
DEL=frm Set position error rate limit. This is useful for detecting obstructions to the motor's path, and faulting the motor sooner than position error limit alone would. This is in the same units as the **=DEA** command.
=DEL Read back the error rate limit.

Velocity Limits

VL=frm Set the velocity fault limit in revolutions per minute. When the motor exceeds this speed (traveling clockwise or counter clockwise), then the motor will fault.
=VL Read back the current setting of the limit in revolutions per minute.

Hardware Limits

External stimulus to limit motion, causes a motion fault if exceeded.
EILP Enable positive limit switch on I/O port.
EILN Enable negative limit switch on I/O port.

Software Limits

As an alternative to hardware limits connected to the limit inputs of the SmartMotor, software limits offer distinct advantages. Software limits are “virtual” limit switches that can interrupt motion with a fault in the event the actual position of the motor strays beyond the desired region of operation. The limit fault is directionally sensitive, so it will cause a fault if motion is commanded further in the direction of a limit once the limit has been past.

SLE Software Limits Enable
SLD Software Limits Disable

SLN=frm Sets left-negative limit.
SLP=frm Sets right-positive limit.
SLM(exp) Software limit mode. Determines if software limits result in a fault or not. 0 no fault, 1 causes fault when soft limit history asserted.

Fault Handling

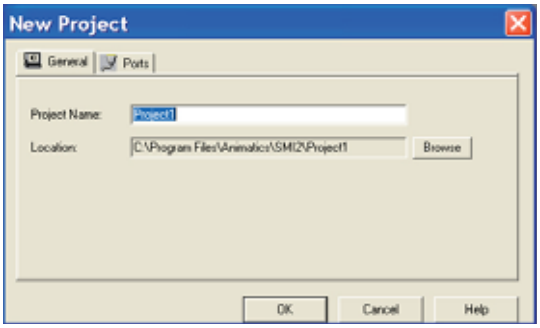
FSA(exp1,exp2) Fault stop action.

exp1: the type of fault to set a mode on:
0 – All types of fault.
1 – Hardware travel limits.
2+ - Reserved.

exp2: action to take
0 - Default action (MTB)
1 - Servo off
2 - X command

The Quick Start section of this guide describes the minimum SMI functionality necessary to talk to a SmartMotor as well as create, download and test SmartMotor programs. SMI as a whole, however, has much greater capability.

SMI Projects

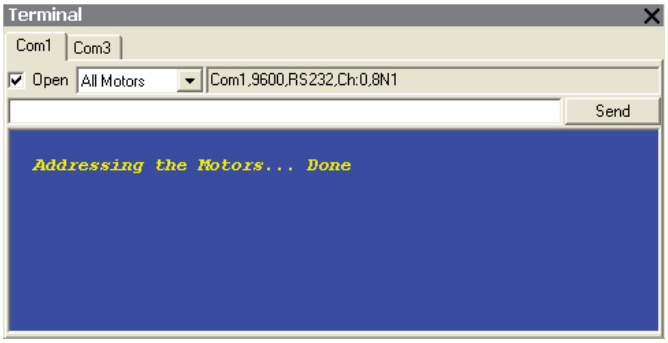


When working with multiple motors, programs or ports, creating a Project can be a great way of organizing and using all of the individual elements.

In applications with more than one SmartMotor and possibly more than one program or communications port, it is helpful to organize all of the elements as a Project, rather than deal with individual files. Projects can be created from the “File” menu. When starting a new project, you have the option of SMI2 exploring the network of motors and setting up the project automatically, or to do it manually by double clicking on the specific communication ports or motors exhibited in the Information window.

Terminal Window

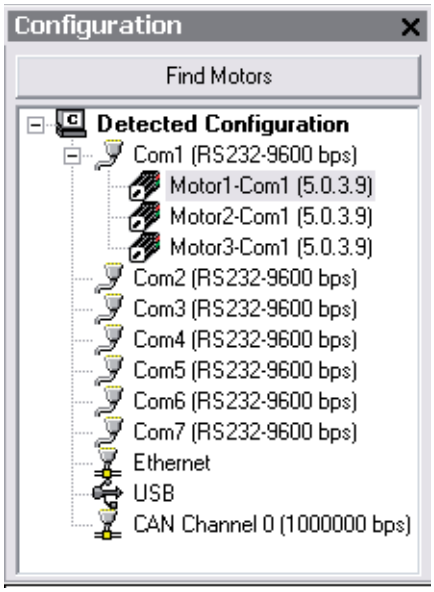
The Terminal window acts as a real time portal between you and the SmartMotor. By typing commands in the Terminal window, you can set up and execute trajectories, execute subroutines of downloaded programs and request data to be reported back.



Specific communication ports can be selected using the tabs. If multiple SmartMotors are on a single communication port are individually addressed, commands can be routed to any or all of them by making the appropriate selection from the pull-down menu just below the tabs. The SMI program will automatically send the appropriate codes to the network to route the data to the intended motors. Commands can be entered in the white text window or the blue screen. If data is flooding back from the motor, then the white text window will be more convenient. **PRINT** commands containing data can be sprinkled in programs to send data to the Terminal window as an aid in debugging. Data that has associated report commands like Position that is retrieved using **RPA** command can be easily reported by simply putting the report command in the program code. Be careful in tight loops because they can bombard the Terminal window with too much data. If a program is sending too much data to the Terminal window, try putting in a **WAIT=50** command in the program. The Terminal window has a scroll feature that allows the user to review history.

Configuration Window

The Configuration window is essential to keeping multiple SmartMotor systems organized.

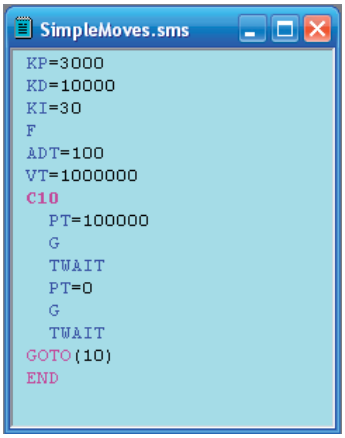


The Configuration window shows the current configuration and allows access to specific ports and motors. Press “Find Motors” to analyze your system, or right-click on an available port and either “detect motors” or “address motors” to find motors attached to that port. Once that is accomplished you can double-click on any port to get instant access to its properties. You can also double-click on any motor to immediately bring up the “Motor View” tool for that motor. By right-clicking the motor, you have immediate and convenient access to its properties along with various other tools.

The Configuration window is essential to keeping multiple SmartMotor systems organized, especially in the context of developing multiple programs and debugging their operation.

Program Editor

SmartMotor programs are written in the SMI Program Editor before being scanned for errors and downloaded to the motor. To get the Program Editor to appear, simply go to the “File” menu and select “New” or simply press the button on the toolbar. As you write your program, the editor will highlight commands it recognizes in different colors.



It is generally good practice to indent program loops by two spaces for readability. Comments are made invisible to the syntax scanner by preceding them with a single quotation mark.

Every program requires an **END**, even if the program is designed to run indefinitely and the **END** is never reached.

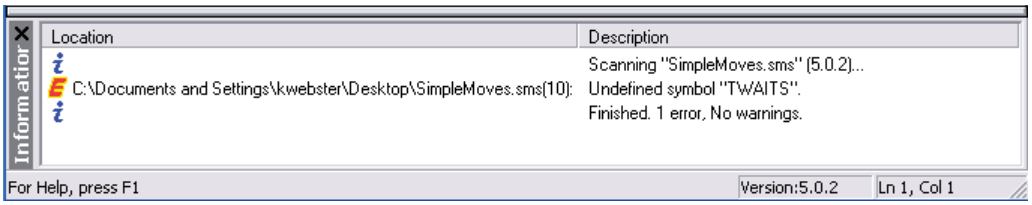
The first time you write a program, you must save it before you can download it to the motor. Every time a program is downloaded, it is automatically saved to that file name. This point is important to note as most Windows applications require an overt “save”. If you want to set aside a certain revision of the program, it should be copied and renamed, or you should simply save the continued work under a new name.

Once a program is complete, you can simply scan it for errors by pressing the button on the toolbar or scan and download it at one time by pressing the button. If errors are found, the download will be aborted and the problems will be identified in the Information window located at the bottom of the screen.

Programs are scanned using a language file which is related to different motor firmware versions. If compile and download is selected, the language file will

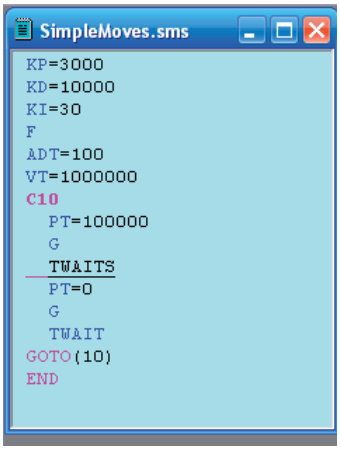
be chosen based on the version read from the motor. If “scan” is selected, the default language file will be used. To change the default language file, select “Compile” from the menu then Compiler default firmware version and then click on the desired firmware version.

Information Window



The Information window shows program status. When a program is scanned and errors are found, they are listed in the Information Window preceded by an.

Program errors can be located instantly by double-clicking on the error listed in the Information window.



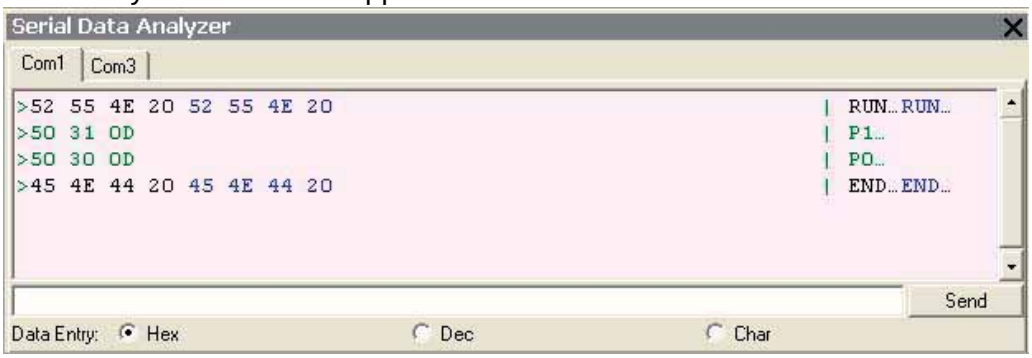
By double-clicking on the error in the Information window, the specific error will be located in the Program Editor and underlined. In the example below, the scanner does not recognize the command TWAITS. The correct command is **TWAIT**.

You can correct the error and press the button again. Once all errors are cleared, the program can be downloaded to the SmartMotor.

Warnings may appear in the Information window to alert you to potential problems, but warnings will not prevent the program from being downloaded to the SmartMotor. It is the programmer’s responsibility to determine the importance of addressing the warnings.

Serial Data Analyzer

The SMI Terminal window formats text and performs other housekeeping functions that are invisible to the user. For an exact picture of what data is being traded between the PC and the SmartMotor, press the button and the Serial Data Analyzer window will appear.



SMI can display the precise data being sent back and forth between the host and the SmartMotor, in multiple formats.

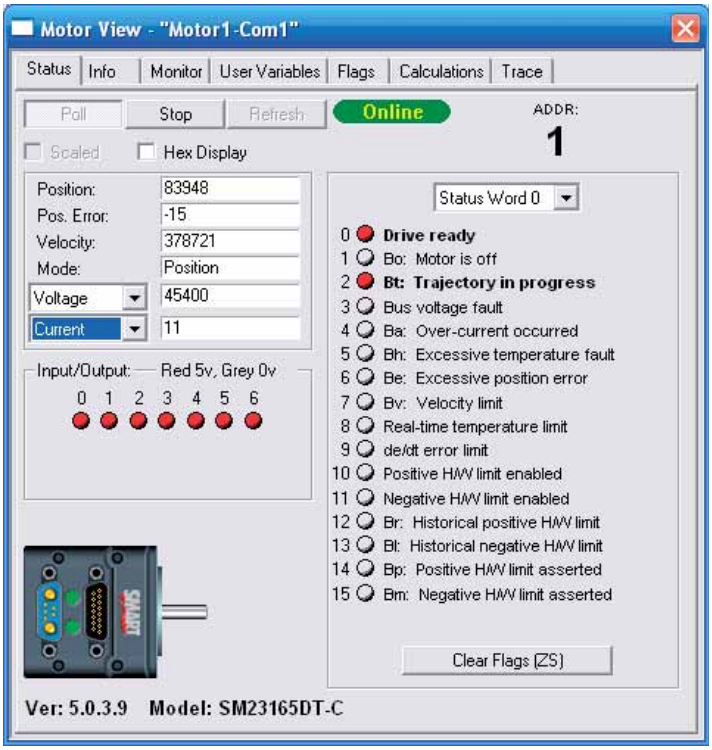
The Serial Data Analyzer window can display serial data in a variety of formats and can be a useful tool in debugging communications. For non-intrusive “sniffing” of data, a special cable can be configured to connect the host receive pin

and ground the data channel to be monitored.

Motor View

The SMI Motor View window enables the user to view multiple parameters related to the motor, in real time. It is most conveniently accessible by double-clicking the motor of interest in the Configuration window.

Motor View *provides a window into the inner workings of a SmartMotor, in real-time.*



Press the "Poll" button to initiate the real-time scanning of motor parameters.

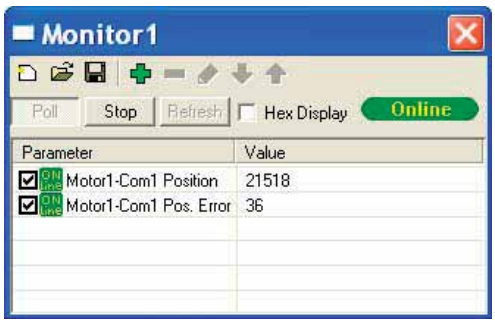
A program can be running in the motor while the Motor View window is polling as long as the program itself does not print text to the serial channel being used for the polling.

In addition to the standard items displayed, there are two fields that allow the user to select from a list of additional parameters to display. In the example here, Voltage and Current are polled. This information can be useful when setting up a system for the first time, or debugging a system in the field. Temperature is also useful to monitor in applications with demanding loads. All seven of the user-configurable onboard I/O points are shown. Any onboard I/O that is configured as an output can be toggled by clicking on the dot below the designating letter.

The SmartMotor has built-in provisions allowing them to be identified by the SMI software. If a motor is identified, a picture of it will appear in the lower left corner of the Motor View window. Tabs across the top offer a wealth of additional information.

Monitor Window

If you want maximum speed and you are interested in only a small number of very specific items, the SMI Monitor window allows you to create your own fully customized monitor. You can find the Monitor window by going to the "Tools" menu and selecting "Monitor View".



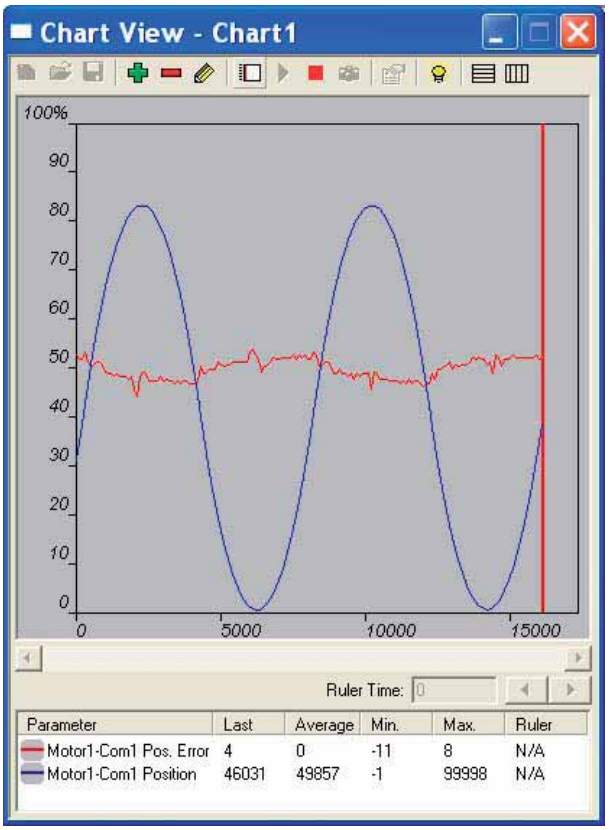
Polling items can be added by pressing the button. The “Add New Monitor Item” window will appear and offer special fields for every portion of the monitoring function.

To monitor items that do not have explicit report commands, fully custom items can be added by entering the specific commands appropriate to getting the data reported, like making a variable equal to the parameter and then reporting the variable, for example.

Sometimes, the best way to understand a data trend is by seeing it graphically. The SMI Chart View provides graphical access to any readable SmartMotor parameter.

Chart View

For graphical monitoring of data, go to the “Tools” Menu and select “Chart View”. Like the Monitor View window, polling items for Chart View can be added by pressing the button.



The Fields and Options are identical to those from the Monitor tool.

Adjustable upper and lower limits for each polled parameter allow them to be scaled to fit the space. The toolbar across the top provides multiple additional functions that are described by holding the cursor over them (without clicking).

Press the button to start the charting action.

While Chart View does not have an intrinsic printing function for a paper copy, Window’s standard “Print Screen” key can capture the graph to be pasted into any standard paint package. Not only is Chart View a very useful tool to see the behavior of the different motion parameters, but its graphical data can be a useful addition to written system reports.

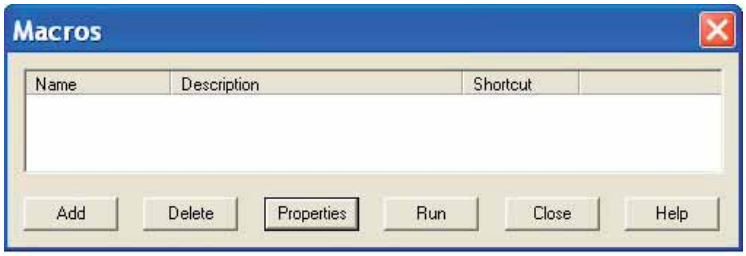
Macros

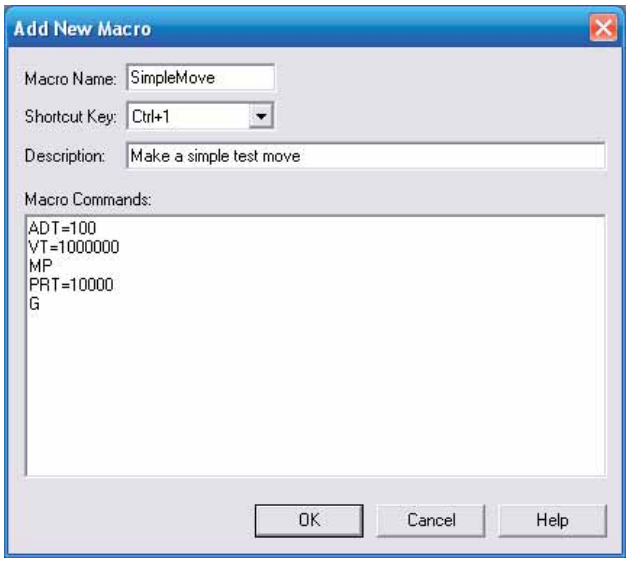
For the SMI user’s convenience, the programmer can associate a command or series of commands with a Ctrl-# key. This is done by selecting “Macro..” from the Tool Menu.

To add a Macro, start by pressing the “ADD” button in the Macro window.

Enter a name for the Macro, select a shortcutControl Key and provide a simple description of the Macro. Then type the command or commands in the window provided.

When this is complete, press the "OK" button. You will again be presented with the Macro window. Click once on the macro you have written and press the "RUN" button in the Macro window to test it.

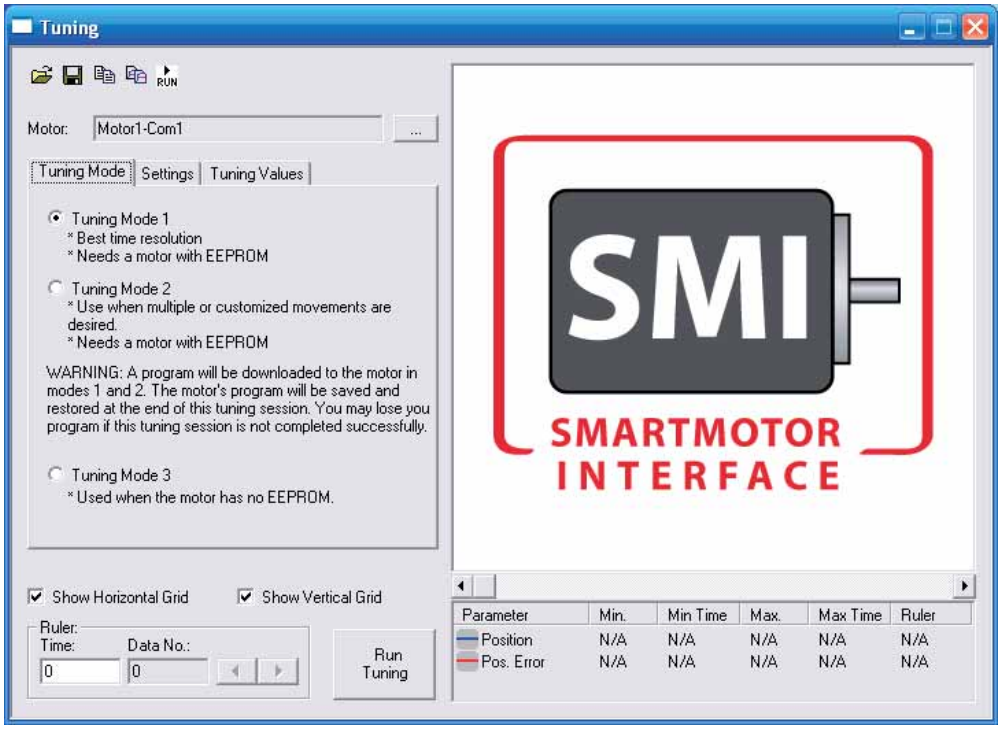




If you are happy with the results, you can press the "Close" button, whereas if you want to edit the Macro, press the "Properties" button instead. With this utility you can create multiple Macros to make the development of your products quicker and easier.

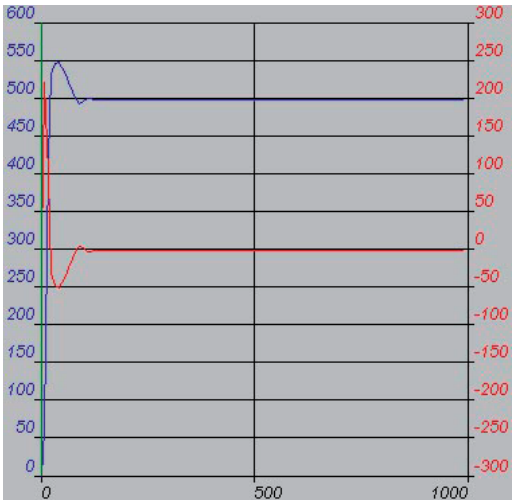
Tuner

Tuning a SmartMotor is far simpler than tuning traditional servos, but it can be even easier using the SMI Tuner to see the actual results of different tuning



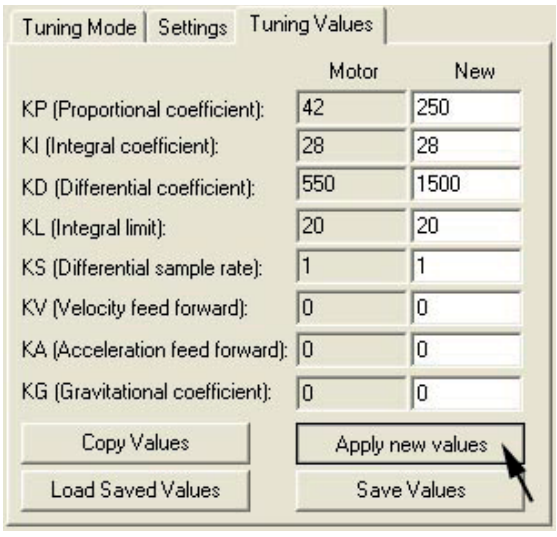
parameters. For a detailed description on how to tune a SmartMotor, refer to the section Tuning the PID Control.

To bring up the SMI Tuner, select “Tuner” from the SMI “Tools” menu.



The Tuner shows the step response of the SmartMotor, graphically. The step response is the SmartMotor’s actual reaction to the request for a small but instantaneous change in position. Rotor inertia prevents the SmartMotor from changing its position in zero time, but how valiant the effort is shows a lot about how well in-tune the motor is.

The Tuner will download a program that utilizes variables a, b, p, t, w and z. The program that was in the motor before tuning and the user variables will be restored after tuning. Before running the Tuner, be sure the motor, and what ever it is connected to, is free to move about 1000 encoder counts or more, and that the device is able to safely withstand an abrupt jolt. If that is the case, then press the “Run Tuning” button at the bottom of the Tuner window. If the SmartMotor is connected, is on, and still, you should see something like what is depicted to the right. The upper curve with the legend on the left is the SmartMotor’s actual position over time. Notice that it overshot its target position before settling in. Exercising the procedure outlined in the section on PID Tuning will stiffen the motor up and create less overshoot. Bear in mind that in a real-world application, there will be an acceleration profile, not a demand for instantaneous displacement, so significant overshoot will not exist. Nevertheless, it is useful to look at the worst case scenario of a step response.



To try a different set of tuning parameters, select the “Tuning Values” tab to the left of the graph area. You will see a list of the tuning parameters with two columns. The one on the left lists what is currently in the SmartMotor. The column to the right provides an area to make changes.

In this example, change KP to 3000 and KD to 10000, then click the “Apply new values” button. Now these new values are in the SmartMotor and we can execute the test of another step response by pressing the “Run tuner” button at the bottom of the Tuning window. The motor will jolt again and the results of the step response will overwrite the previous graph. Normally, this process involves repeated trials, again, exercising the procedure outlined in the section on The PID Filter.

Once you are happy with the results, the parameters that had the best results can be added to the top of your program in the SmartMotor, or in applications where there are no programs in the motors, sent by a host after each power-up. Whether from a host or in a program, the tuning parameters would be set using the tuning commands:

KP=3000

KI=30

KD=10000

KL=32767

F

SMI Options

The SMI Terminal Software can be customized in general by way of the “Options” choice in the “Tools” menu.

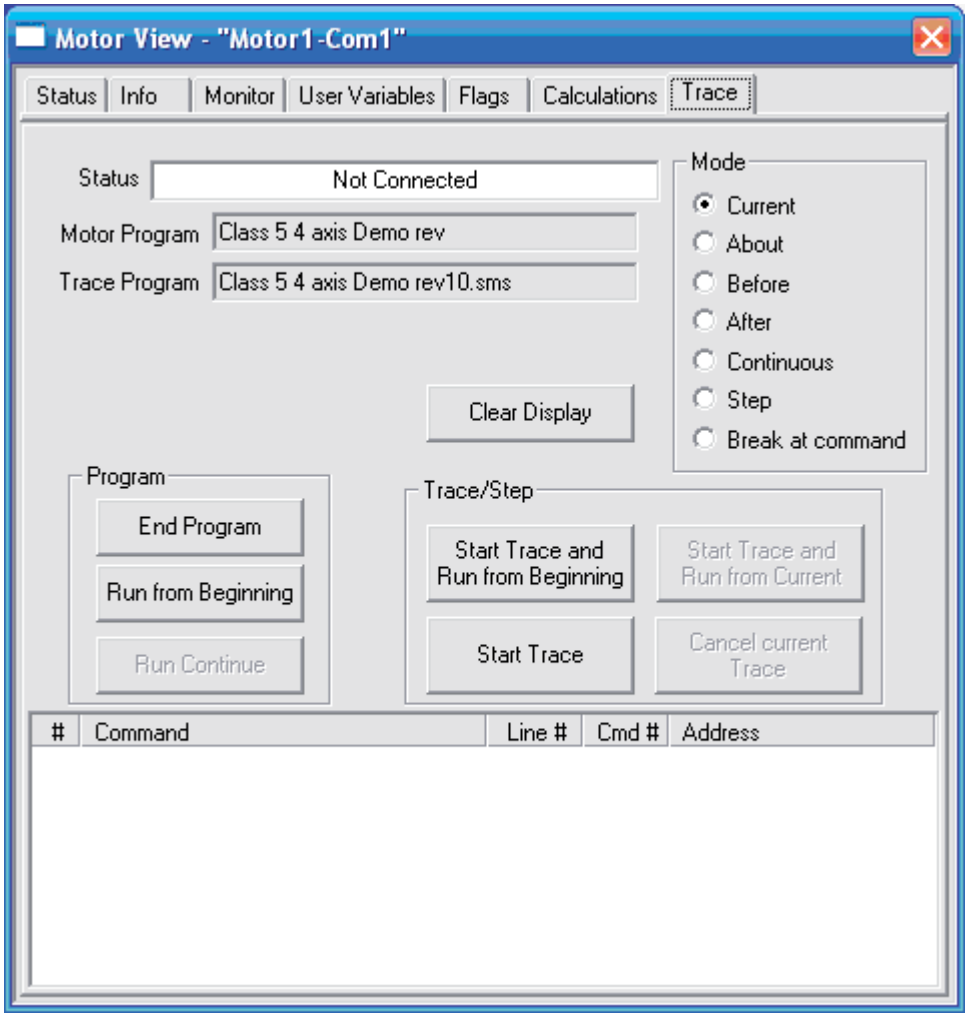
A key option to consider is the firmware version. Since different SmartMotor firmware versions have subtle differences, the program scanner needs to know which firmware is being utilized so it can know what are legal commands and what are commands that are unsupported.

Other adjustable options deal more with the issue of preferences.

SMI Help

The most complete and up-to-date information available for SMI functions is available within the program’s extensive “Help” facility. The easiest way to get instant access to help on any feature is by clicking on the  button in the main toolbar. After clicking on the  button, click on the item you want to learn about and information will be presented on that item.

SMI Trace Functions



First, you must select a program that matches the one currently loaded in the motor by double-clicking on it in the Editor window. This will cause the program to be scanned in order to generate the addresses that will be used while tracing.

The general procedure for tracing is as follows:

Select the desired Mode.

Double-click on desired line in the Editor window if needed.

Press the desired button in the Trace/Step box.

Remember the program must run before anything will happen.

The Status window shows the current state of the Trace program and Motor Program. This becomes active after a command is executed on the “Trace” tab and will remain active until the Motor View is closed.

Possible displays:

"Not Connected" – Not connected to motor

"Program running" or "Program Stopped" – If at a breakpoint or the program is stopped.

"Trace Active or Trace Inactive" – If there a trace currently in progress or waiting to hit a breakpoint in progress. If a trace is active it must be canceled before selecting a new Mode.

"At Break Point" – Program execution halted because of breakpoint or a step has been completed.

Motor Program window: Shows the name of the program contained in the motor.

Trace Program window: Shows the name of the program that was doubled-clicked.

"Clear Display" button: Clears the highlighted text in the editor window and remove any information in the Trace List window.

Program frame:

"End Program:" Stops program execution by writing **END** comand

"Run from Beginning:" Issue **RUN** command.

"Run Continue": Release firmware from current breakpoint. Only avaiable when at a breakpoint.

Mode Frame:

Trace selections: For any trace information to be retrieved from the motor the program must run. After a trace is completed or canceled it will not affect the execution state of the program in the motor.

"Current:" Captures the first 20 points encountered.

"About," "Before" and "After": Requires the user to select a line from the program in the Editor window by double-clicking on it. This will load the command, address, and line fields with information. Verify the information is correct. If running a trace with one of these modes, the user can select the "Cancel Current Trace" button.

"About:" captures 9 points before and 10 points after desired line.

"Before:" captures 20 points before the desired line.

"After:" captures 20 points following the desired line.

"Continuous:" Polls the motor for the commands that are executing. Because of bandwidth, every line executed by the prgram will not show up in the trace view or highlighted in the program.

"Trace/Step" frame options for the trace selections.

"Start Trace and Run from Beginning" button: Sets the trace information in motor and issues a **RUN** command.

"Start Trace" button: Sets the trace information in the motor.

"Start Trace and Run from Current" button: Available when at a break point. The trace information will be set in the motor and the program will continue from the current break point.

"Cancel Trace" button: Available when a trace is active to cancel the current trace.

Step:

Trace/Step frame options for the step selection:

"Step from Beginning" button: Sets a breakpoint in the motor and issues a RUN command. The program will execute the first line of code and stop.

"Step from Current" button: Sets a breakpoint in the motor. If the program is running, the motor will stop at the next command. If the program is at a breakpoint the motor will execute the next command and stop.

BREAK at Command:

Requires the user to select a line in the program by double-clicking on it in the Editor window. This will load the Command, Address, and Line fields with information. Verify the information is correct.

"Trace/Step" frame options for the BREAK At Command selections

"Set Breakpoint and Run from Beginning" button: Sets the breakpoint and runs the program from the beginning.

"Set Breakpoint" button: Sets a breakpoint in the motor.

"Set Breakpoint and Run from Current" button: If at a breakpoint this sets the new breakpoint and runs the program from the current location.

"Remove Breakpoint" button: If a breakpoint was set and not reached this button removes it.

Appendix A: The ASCII Character Set

ASCII is an acronym for American Standard Code for Information Interchange. It refers to the convention established to relate characters, symbols and functions to binary data. If a SmartMotor is asked its position over the RS-232 link, and it is at position 1, it will not return a byte of value one, but instead will return the ASCII code for 1 which is binary value 49. That is why it appears on a Terminal window as the numeral 1.

The ASCII character set is as follows:

0	NUL	35	#	70	F	105	i
1	SOH	36	$	71	G	106	j
2	STX	37	%	72	H	107	k
3	ETX	38	&	73	I	108	l
4	EOT	39	‘	74	J	109	m
5	ENQ	40	(	75	K	110	n
6	ACK	41	)	76	L	111	o
7	BEL	42	*	77	M	112	p
8	BS	43	+	78	N	113	q
9	HT	44	,	79	O	114	r
10	LF	45	-	80	P	115	s
11	VT	46	.	81	Q	116	t
12	FF	47	/	82	R	117	u
13	CR	48	0	83	S	118	v
14	SO	49	1	84	T	119	w
15	SI	50	2	85	U	120	x
16	DLE	51	3	86	V	121	y
17	DC1	52	4	87	W	122	z
18	DC2	53	5	88	X	123	{
19	DC3	54	6	89	Y	124	|
20	DC4	55	7	90	Z	125	}
21	NAK	56	8	91	[	126	~
22	SYN	57	9	92	\	127	Del
23	ETB	58	:	93	]		
24	CAN	59	;	94	^		
25	EM	60	<	95	_		
26	SUB	61	=	96	’		
27	ESC	62	>	97	a		
28	FC	63	?	98	b		
29	GS	64	@	99	c		
30	RS	65	A	100	d		
31	US	66	B	101	e		
32	SP	67	C	102	f		
33	!	68	D	103	g		
34	“	69	E	104	h		

The SmartMotor language allows the programmer to access data on the binary level. Understanding binary data is very easy and useful when programming the SmartMotor or any electronic device. What follows is an explanation of how binary data works.

All digital computer data is stored as binary information. A binary element is one that has only two states, commonly described as “on” and “off” or “one” and “zero.” A light switch is a binary element. It can either be “on” or “off.” A computer’s memory is nothing but a vast array of binary switches called “bits.”

The power of a computer comes from the speed and sophistication with which it manipulates these bits to accomplish higher tasks. The first step towards these higher goals is to organize these bits in such a way that they can describe things more complicated than “off” or “on.”

Different numbers of bits are used to make up different building blocks of data. They are most commonly described as follows:

Four bits	=	Nibble
Eight bits	=	Byte
Sixteen bits	=	Word
Thirty two bits	=	Long

One bit has two possible states, on or off. Every time a bit is added, the possible number of states is doubled. Two bits have four possible states. They are as follows:

00	off-off
01	off-on
10	on-off
11	on-on

A nibble has 16 possible states. A byte has 256, a word has 65536, and a long has billions of possible combinations.

Because a byte of information has 256 possible states, it can reflect a number from zero to 255. This is elegantly done by assigning each bit a value of twice the one before it, starting with one. Each bit value becomes as follows:

Bit	Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

If all their values are added together the result is 255. By leaving particular bits out, any sum between zero and 255 can be created. Look at the following example bytes and their decimal values:

Appendix B: Binary Data

Byte	Value
0 0 0 0 0 0 0 0	0
0 0 0 0 0 0 0 1	1
0 0 0 0 0 0 1 0	2
0 0 0 0 0 0 1 1	3
0 0 0 1 0 0 0 0	16
0 0 0 1 1 1 1 0	30
0 0 1 1 1 1 0 0	60
1 0 0 0 0 0 0 0	128
1 0 0 1 1 1 0 1	157
1 1 1 1 1 1 1 1	255

To make use of the limited memory available with micro controllers that can fit into a SmartMotor, there are occasions where every bit is used. One example is the Status Word 0. A single value can be uploaded from a SmartMotor and have coded into it, in binary, eight, sixteen or thirty-two independent bits of information. The following is the Status Word 0 and its 16 bits of coded information:

Name	Description	Bit	Value
	Drive ready	0	1
Bo	Motor OFF	1	2
Bt	Trajectory in progress	2	4
	Bus voltage fault	3	8
Ba	Over current	4	16
Bh	Excessive temperature fault	5	32
Be	Excessive position error fault	6	64
Bv	Velocity limit fault	7	128
	Real-time temperature limit	8	256
	Derivative of position error limit	9	512
	Hardware right (+) limit enabled	10	1024
	Hardware left (-) limit enabled	11	2048
Br	Historical right (+) limit fault	12	4096
Bl	Historical left (-) limit fault	13	8192
Bp	Real time right (+) limit	14	16384
Bm	Real time left (-) limit	15	32768

There are three useful mathematical operators that work on binary data, the “&” (bit-wise and), the ”|” (bit-wise or) and the “!|” (bit-wise exclusive or). The “&” compares the two operands (bytes, words or longs) and looks for what they have in common. The resulting data has ones only where there were ones in both operands. The “**|**” results in a one for each bit corresponding to a one in either operand. The “**!|**” produces a one for each bit when the corresponding bits in the two operands are different and a zero when they are the same. These operations are illustrated in the following examples:

A	B	A&B	A|B	A!|B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Knowing how the binary data works will enable shorter, and therefore faster, code to be written. The following are two code examples that are written to check if both limit inputs are high. One does this without taking advantage of a binary operator while the second shows how using a binary operator makes the code shorter, and therefore faster.

Example 1:

```
IF Bm           'Look for - limit high
  IF Bp         'Look for + limit high
    GOSUB100    'Execute subroutine
  ENDIF
ENDIF
```

Example 2:

```
IF (W(0)&49152)==49152 'Look at both limits, bits 14 & 15,
                'w/bit mask 49152 = 32768 + 16384
  GOSUB100      'Execute subroutine
ENDIF
```

Both examples will execute subroutine 100 if both limit inputs are high. Example two uses less code than example one and will run faster as a part of a larger program loop.

The next two examples show how the use of the “|” operator can improve program size and execution speed:

Example 3:

```
IF IN(0)        'Look for input 0
  GOSUB200      'Execute subroutine
ENDIF
IF IN(1)        'look for input 1
  GOSUB200      'Execute subroutine
ENDIF
```

Example 4:

```
IF IN(W,0,3)    'Look at both inputs 0 and 1
   GOSUB200     'Execute subroutine
ENDIF
```

Both examples 3 and 4 accomplish the same task with different levels of efficiency.

Legend

When a description of a command states that the command is assigned a value, this means that the command like **VA** would typically be used like this:

```
VA=2*a
```

When a description states that it 'reports' a value, then that command typically stands alone on a line, and prints results to the active serial interface:

```
RVA
```

When a description of a command states that a command 'Sets' a value, then that command typically is on the right-hand side of an equals:

```
b=ADT
```

Command descriptions below that do not specify one of these conditions are simply commands that complete an action and stand alone on a line.

Command List:

a...z	Get user variable
a=...z=	Set user variable
aa...zz	Get user variable
aa=...zz=	Set user variable
aaa...zzz	Get user variable
aaa=...zzz=	Set user variable
ab[index]	Get array variable 8 bit
ab[index]=...	Set array variable 8 bit
af[index]	Get float variable
af[index]=...	Set float variable
al[index]	Get array variable 32 bit
al[index]=...	Set array variable 32 bit
aw[index]	Get array variable 16 bit
aw[index]=...	Set array variable 16 bit
Ai(0)	Arm index rising edge of internal encoder
Ai(1)	Arm index rising edge of external encoder
Aij(0)	Arm index rising edge then falling edge internal encoder
Aij(1)	Arm index rising edge then falling edge external encoder

Appendix C: Commands

Aj(0)	Arm index falling edge of internal encoder
Aj(1)	Arm index falling edge of external encoder
Aji(0)	Arm index falling edge then rising edge internal encoder
Aji(1)	Arm index falling edge then rising edge external encoder
ABS(...)	Get integer absolute value
AC	Get commanded acceleration
ACOS(...)	Get arc-cosine in degrees
ADDR	Get motor's serial address
ADDR=...	Set serial address
ADT=...	Set acceleration and deceleration
ADTS=...	Set accel. and decel. for synchronized motion
AMPS	Get assigned max. drive PWM limit
AMPS=...	Set PWM drive signal limit
ASIN(...)	Get arc-sine in degrees
AT	Get target acceleration
AT=...	Set acceleration
ATS=...	Set acceleration for synchronized motion
ATAN()	Get arc-tangent in degrees
ATOF()	Get ASCII to float conversion
B()	Get status bit
Ba	Get over current Status Bit
BAUD(0)	Get baud rate of channel 0.
BAUD(1)	Get baud rate of channel 1.
BAUD#	Set baud rate of channel 0.
BAUD(0)=...	Set baud rate of channel 0.
BAUD(1)=...	Set baud rate of channel 1.
Be	Get excessive position error Status Bit
Bh	Get excessive temperature Status Bit
Bi(0)	Get index captured Status Bit (rising, internal encoder)
Bi(1)	Get index captured Status Bit (rising, external encoder)
Bj(0)	Get index captured Status Bit (falling, internal encoder)

Bj(1)	Get index captured Status Bit (falling, external encoder)
Bk	Get EEPROM data integrity Status Bit
Bl	Get historical hardware left/negative limit Status Bit
Bls	Get historical software left/negative limit Status Bit
Bm	Get left/negative hardware limit Status Bit
Bms	Get left/negative software limit Status Bit
Bo	Get motor off Status Bit
Bp	Get right/positive hardware limit Status Bit
Bps	Get right/positive software limit Status Bit
Br	Get historical right/positive hardware limit Status Bit
Brs	Get historical right/positive software limit Status Bit
Bs	Get syntax error Status Bit
Bt	Get trajectory in progress Status Bit
Bv	Get velocity error fault
Bw	Get encoder wrap around Status Bit
Bx(0)	Get real time internal index input Status Bit
Bx(1)	Get real time external index input Status Bit
BREAK	Program execution flow control.
BRKENG	Brake engage
BRKRLS	Brake release
BRKSRV	Brake without servo
BRKTRJ	Brake without trajectory
C#	Program subroutine label
CADDR	Get CAN address
CADDR=...	Set CAN address
CAN	Get CAN error
CANCTL(...)	Control network features
CASE #	Program flow instruction
CBAUD	Get CAN baudrate
CBAUD=...	Set CAN baudrate
CCHN()	Close a serial channel

Appendix C: Commands

CHN(0)	Get RS-232 communications error flags
CHN(1)	Get RS-485 communications error flags
CLK	Get 1 millisecond clock variable
CLK=...	Set 1 millisecond clock
COS(...)	Get cosign of an angle in degrees
CP	Get cam pointer
CTA(...)	Add cam table
CTE(...)	Erase cam table(s)
CTR(0)	Get primary encoder/step and direction counter
CTR(1)	Get second encoder/step and direction counter
CTT	Get number of cam tables in EE
CTW()	Write cam table point
DEA	Get de/dt actual
DEFAULT	Switch-case structure element
DEL	Get the setting for de/dt fault limit
DEL=...	Set the de/dt fault limit
DFS(...)	Get af[] variable in its raw 32-bit IEEE format.
DITR(...)	Disable 1 or more individual interrupts
DT	Get deceleration setting
DT=...	Set deceleration
DTS=...	Set deceleration for synchronized motion
EA	Get actual position error
ECHO	Echo input data back out main channel
ECHO_OFF	Stop echo main channel
ECHO1	Echo input data back out second channel
ECHO_OFF1	Stop echo second channel
EIGN(...)	Set one or more I/O pins to input
EILN	Activate negative hardware limit switch
EILP	Activate positive hardware limit switch
EIRE	Configure index capture pin to capture external encoder
EIRI	Configure index capture pin to capture internal encoder
EISM(6)	Confgure pin 6 to call G command
EITR(...)	Enable one or more interrupts

EL	Get position error fault limit
EL=...	Set position error fault limit
ELSE	If structure element
ELSEIF	Else structure element
ENC0	Select internal encoder for servo
ENC1	Select external encoder for servo
END	End program
ENDIF	End if statement
ENDS	End switch structure
EOBK(...)	Send brake signal to I/O output
EPTR	Get data EEPROM pointer
EPTR=...	Set data EEPROM pointer
ERRC	Get most recent command error code
ERRW	Get communication channel of most recent command error
F	Activate buffered PID settings
FABS(...)	Get floating-point absolute error
FSA(...)	Configure action upon fault
FSQRT(...)	Get floating point square root
FW	Get firmware version as 32-bit field
G	Start motion (GO)
G(...)	Start motion (GO) specific trajectory
GS	Start motion (GO) for synchronized move
GETCHR	Get character from main comm channel
GETCHR1	Get character from second comm channel
GOSUB(...)	Call a subroutine by literal number, or variable
GOSUB#	Call a subroutine
GOTO(...)	Goto a program label by literal number, or variable
GOTO#	Goto a program label
HEX(...)	Get a hex string into a variable
I(0) (capital i)	Get hardware index position variable (rising edge, internal encoder)
I(1) (capital i)	Get hardware index position variable (rising edge, external encoder)
IF ...	Conditional test

IN(...)	Get I/O input
INA(...)	Get analog input
ITR(...)	Configure user interrupt
ITRD	Global disable of user interrupts
ITRE	Global enable of user interrupts
J(0)	Get hardware index position variable (falling edge, internal encoder)
J(1)	Get hardware index position variable (falling edge, external encoder)
KA	Get the buffered PID setting for KA (acceleration feed-forward)
KA=...	Set the buffered PID setting for KA (acceleration feed-forward)
KC	Get the setting for KC
KC=...	Set KC
KCS	Get the setting for KCS
KCS=...	Set KCS
KD	Get the buffered PID setting for KD (Derivative term)
KD=...	Set the buffered PID setting for KD (Derivative term)
KG	Get the buffered PID setting for KG (gravity term)
KG=...	Set the buffered PID setting for KG (gravity term)
KI	Get the buffered PID setting for KI (integral term)
KI=	Set the buffered PID setting for KI (integral term)
KL	Get the buffered PID setting for KL (integral limit term)
KL=...	Set the buffered PID setting for KL (integral limit term)
KP	Get the buffered PID setting for KP (proportional term)
KP=	Set the buffered PID setting for KP (proportional term)
KS	Get the buffered PID setting for KS (derivative filter control)
KS=...	Set the buffered PID setting for KS (derivative filter control)

KV	Get the buffered PID setting for KV (velocity feed-forward)
KV=...	Set the buffered PID setting for KV (velocity feed-forward)
LEN	Main communications channel buffer fill level, data mode
LEN1	Second communications channel buffer fill level, data mode
LFS(...)	Get float value from 32-bit IEEE format
LOAD	Initiate program download to motor
LOCKP	Prevent program upload until new program is loaded
LOOP	While structure element
MC	Enable Cam Mode
MC(...)	Enable Cam Mode, additional trajectory
MCE(...)	Cam spline enable
MCW(...)	Cam starting point
MDB	TOB commutation enable
MDC	Sine current commutation mode
MDE	Trapezoidal encoder commutation mode
MDS	Sine voltage commutation mode
MDT	Trapezoidal hall commutation mode
MF0	Set CTR(1) to 0, and choose quadrature mode on external encoder
MFA(...)	Follow Mode Ascend ramp
MFD(...)	Follow Mode Decend ramp
MFDIV	Get Follow Mode divisor setting
MFDIV=...	Set Follow Mode divisor
MFMUL	Get Follow Mode multiplier setting
MFMUL=...	Set Follow Mode divisor
MFR	Choose Follow Mode with quadrature
MFR(...)	Choose Follow Mode with quadrature, additional trajectory
MFSDC(...)	Follow Mode stall-dwell-continue
MFSLEW(...)	Follow Mode Slew
MINV(...)	Invert commutation
MODE	Get Operating Mode

Appendix C: Commands

MODE(...)	Get Operating Mode, specific trajectory
MP	Enable Position Mode
MP(...)	Enable Position Mode, additional trajectory
MS0	Set CTR(1) to 0, and choose step/direction mode on external encoder
MSR	Choose Follow Mode with step/direction
MSR(...)	Choose Follow Mode with step/direction, additional trajectory
MT	Enable Torque Mode
MTB	Mode Torque Brake
MV	Enable Velocity Mode
MV(...)	Enable Velocity Mode, additional trajectory
O=...	Set Origin
O(...)=...	Set specific trajectory Origin
OC(...)	Get output condition (24 volt IO)
OCHN(...)	Open communications channel
OF(...)	Get output faults (24 volt IO)
OFF	Stop servoing the motor
OR(...)	Set 1 or more outputs to low
OS(...)	Set 1 or more outputs to high
OSH=...	Shift Origin
OSH(...)=...	Shift specific Origin
OUT(...)=...	Set 1 or more outputs to a specific state
PA	Get actual motor position
PAUSE	Pause program execution
PC	Get commanded motor position
PC(...)	Get commanded motor pos., specific trajectory
PI	Get the mathematical value pi
PID1	16,000 Hz PID rate
PID2	8,000 Hz PID rate (default)
PID4	4,000 Hz PID rate
PID8	2,000 Hz PID rate
PMA	Get actual position modulo
PML	Get position modulo limit setting
PML=...	Set position modulo limit

PMT	Get position modulo target (position move)
PMT=...	Set position modulo target (position move)
PRA	Get actual position relative to move start
PRC	Get commanded position relative to move start
PRINT(...)	Print data to main communications channel
PRINT1(...)	Print data to second communications channel
PRT	Get position relative target setting
PRT=...	Set position relative target
PRTS(...)	Set position target synchronized relative
PRTSS(...)	Set supplemental position target synchronized relative
PT	Get position target setting
PT=...	Set position target
PTS=(...)	Set position target synchronized absolute
PTSS=(...)	Set supplemental position target synchronized absolute
PTSD	Get synchronized move linear distance
PTST	Get synchronized move linear time
Ra...Rz	Report variables
Raa...Rzz	Report variables
Raaa...Rzzz	Report variables
Rab[index]	Report byte array variables (8-bit)
Raf[index]	Report float array variables
Ral[index]	Report long array variables (32-bit)
Raw[index]	Report word array variables (16-bit)
RABS(...)	Report integer absolute value
RAC	Report commanded acceleration
RACOS(...)	Report arc-cosine in degrees
RADDR	Report motor's serial address
RAMPS	Report assigned max. drive PWM limit
RANDOM	Get the next value from random generator e.g. a=RANDOM
RANDOM=...	Set the random generator seed
RASIN(...)	Report arc-sine in degrees
RAT	Report target acceleration
RATAN(...)	Report arc-tangent in degrees

RATOF(...)	Report ASCII to float conversion
RB(...)	Report Status Bit
RBa	Report over current Status Bit
RBAUD(0)	Report baud rate of channel 0.
RBAUD(1)	Report baud rate of channel 1.
RBe	Report excessive position error Status Bit
RBh	Report excessive temperature Status Bit
RBi(0)	Report index captured Status Bit (rising, internal encoder)
RBi(1)	Report index captured Status Bit (rising, external encoder)
RBj(0)	Report index captured Status Bit (falling, internal encoder)
RBj(1)	Report index captured Status Bit (falling, external encoder)
RBk	Report EEPROM data integrity Status Bit
RBl	Report historical hardware left/negative limit Status Bit
RBls	Report historical software left/negative limit Status Bit
RBm	Report left/negative hardware limit Status Bit
RBms	Report left/negative software limit Status Bit
RBo	Report motor off Status Bit
RBp	Report right/positive hardware limit Status Bit
RBps	Report right/positive software limit Status Bit
RBr	Report historical right/positive hardware limit Status Bit
RBrs	Report historical right/positive software limit Status Bit
RBs	Report syntax error Status Bit
RBt	Report trajectory in progress Status Bit
RBv	Report velocity error fault
RBw	Report encoder wrap around Status Bit
RBx(...)	Report real time index input Status Bit
RCADDR	Report CAN address
RCAN	Report CAN error
RCBAUD	Report CAN baudrate

RCHN(0)	Report RS-232 communications error flags
RCHN(1)	Report RS-485 communications error flags
RCKS	Report program checksum
RCLK	Report 1 millisecond clock variable
RCOS(...)	Report cosine of an angle in degrees
RCP	Report cam pointer
RCTR(0)	Report primary encoder/step and direction counter
RCTR(1)	Report second encoder/step and direction counter
RCTT	Report number of cam tables in EE
RDEA	Report DE/Dt actual
RDEL	Report the setting for DE/Dt fault limit
RDFS	Report af[] variable in its raw 32-bit IEEE format.
RDT	Report deceleration setting
REA	Report actual position error
REL	Report position error fault limit
REPTR	Report data EEPROM pointer
RERRC	Report most recent command error code
RERRW	Report communication channel of most recent command error
RES	Get encoder resolution. e.g. **a=RES**
RESUME	Continue program execution after a pause
RETURN	Return from subroutine
RETURNI	Return from interrupt routine
RFABS(...)	Report floating-point absolute error
RFSQRT(...)	Report floating point square root
RFW	Report firmware version as 32-bit field
RGETCHR	Report character from main communication channel
RGETCHR1	Report character from second communication channel
RHEX(...)	Report a hex string into a variable
RI(0)	Report hardware index position variable (rising edge, internal encoder)
RI(1)	Report hardware index position variable (rising edge, external encoder)

RIN(...)	Report I/O input
RINA(...)	Report analog input
RJ(0)	Report hardware index position variable (falling edge, internal encoder)
RJ(1)	Report hardware index position variable (falling edge, external encoder)
RKA	Report the buffered PID setting for **KA** (acceleration feed-forward)
RKC	Report the setting for **KC**
RKCS	Report the setting for **KCS**
RKD	Report the buffered PID setting for **KD** (derivative term)
RKG	Report the buffered PID setting for **KG** (gravity term)
RKI	Report the buffered PID setting for **KI** (integral term)
RKL	Report the buffered PID setting for **KL** (integral limit term)
RKP	Report the buffered PID setting for **KP** (proportional term)
RKS	Report the buffered PID setting for **KS** (derivative filter control)
RKV	Report the buffered PID setting for **KV** (velocity feed-forward)
RLEN	Main com channel buffer fill level, data mode
RLEN1	Second com channel buffer fill level, data mode
RLFS(...)	Report float value from 32-bit IEEE format
RMFDIV	Report Follow Mode divisor setting
RMFMUL	Report Follow Mode multiplier setting
RMODE	Report Operating Mode
RMODE(...)	Report Operating Mode, specific trajectory
ROC(...)	Report output condition (24 volt IO)
ROF(...)	Report output faults (24 volt IO)
RPA	Report actual motor position
RPC	Report commanded motor position
RPC(...)	Report commanded motor position, specific trajectory
RPI	Report the mathematical value pi

RPMA	Report actual position modulo
RPML	Report position modulo limit setting
RPMT	Report position modulo target (position move)
RPRA	Report actual position relative to move start
RPRC	Report commanded position relative to move start
RPRT	Report position relative target setting
RPT	Report position target setting
RPTSD	Report synchronized move linear distance
RPTST	Report synchronized move time (ms)
RRANDOM	Report the next value from random generator
RRES	Report encoder resolution.
RSAMP	Report sample rate (Hz)
RSIN(...)	Report sine of angle in degrees
RSLM	Report soft limit mode
RSLN	Report soft limit left/negative setting
RSLP	Report soft limit right/positive setting
RSP	Report sample rate and firmware string
RSP1	Report firmware compile date
RSP2	Report bootloader revision
RSQRT(...)	Report integer square root
RT	Report current requested torque
RTAN(...)	Report tangent of angle in degrees
RTEMP	Report temperature in degrees
RTH	Report temperature limit setting
RTHD	Report current limit timer setting
RTMR(...)	Report user timer
RTRQ	Report torque real-time
RTS	Report torque slope setting
RUIA	Report current
RUJA	Report voltage
RUN	Execute stored program
RUN?	End if the **RUN** command has not been issued since power up
RVA	Report actual velocity (filtered)

Appendix C: Commands

RVC	Report commanded velocity
RVL	Report velocity limit setting
RVT	Report target velocity
RW(...)	Report a specific status word
S	Stop move in progress abruptly
S(...)	Stop move in progress abruptly, specific trajectory
SADDR#	Set motor to new address
SAMP	Get sample rate (Hz)
SILENT	Suppress **PRINT** messages main channel
SILENT1	Suppress **PRINT** messages second channel
SIN(...)	Get sine of angle in degrees
SLD	Disable software limits
SLE	Enable software limits
SLEEP	Initiate Sleep Mode main channel
SLEEP1	Initiate Sleep Mode second channel
SLM	Get Soft Limit Mode. e.g. **a=SLM**
SLM(...)	Set Soft Limit Mode
SLN	Get left/negative software limit
SLN=...	Set left/negative software limit
SLP	Get right/positive software limit
SLP=...	Set right/positive software limit
SQRT(...)	Get integer square root
SRC(...)	Set follow and/or cam encoder source
STACK	Reset nesting stack tracking
STDOUT=...	Set where report commands are printed to
SWITCH ...	Program execution control
T	Get the target toque setting
T=	Set target torque
TALK	Enable **PRINT** messages on main channel
TALK1	Enable **PRINT** messages on main channel
TAN(...)	Get tangent of an angle in degrees
TEMP	Get temperature
TH	Get temperature limit setting
TH=...	Set temperature limit

THD	Get current limit timer setting
THD=...	Sets current limit timer delay
TMR(...)	Get a specific user timer value. e.g. a=TMR(0)
TMR(...) (as cmd)	Set a user timer. e.g. TMR(0,1000)
TRQ	Get torque real-time
TS	Get torque slope setting
TS=...	Set torque slope
TSWAIT	Pause program during a synchronized move
TWAIT	Pause program during a move
TWAIT(...)	Pause pgm. during a move, specific trajectory
UIA	Get motor current
UJA	Get bus voltage
UO(...)=...	Set one or more user Status Bits to specific values
UP	Upload user EEPROM program contents
UPLOAD	Upload user EEPROM readable program
UR(...)	Set one or more user status bits to a 0.
US(...)	Set one or more user status bits to a 1.
VA	Get actual velocity (filtered)
VAC(...)	Set velocity filter
VC	Get commanded velocity
VL	Get velocity limit setting
VL=...	Set velocity limit
VLD(...)	Sequentially load variables from data EEPROM
VST(...)	Sequentially store variables to data EEPROM
VT	Get velocity target setting
VT=...	Set velocity target
VTS=...	Set position target for synchronized motion
W(...)	Report a specific status word
WAIT=...	Suspends program for number of milliseconds
WAKE	Terminate Sleep Mode main channel
WAKE1	Terminate Sleep Mode second channel
WHILE...	Conditional program flow command
X	Slow motor motion to stop
X(...)	Slow motor motion to stop, specific trajectory
Z	Total system reset

Z(...)	Reset a particular Status Bit
Za	Reset current limit violation latch bit
Ze	Reset position error fault
Zh	Reset temperature fault
Zl	Reset historical left/neg. hardware limit latch bit
Zls	Reset historical left/neg. software limit latch bit
Zr	Reset historical right/pos. hardware limit latch bit
Zrs	Reset historical right/pos. software limit latch bit
Zs	Reset syntax error bit
ZS	Reset system latches to power-up state
Zv	Reset velocity error fault
Zw	Reset encoder wrap around event latch bit

Integer Array Memory:

Long (32-bit signed)	Word (16-bit signed)		Bytes (8-bit signed)			
al[n] where n is:	aw[n] where n is:		ab[n] where n is:			
	LSb	MSb	LSb	middle bytes		MSb
0	0	1	0	1	2	3
1	2	3	4	5	6	7
2	4	5	8	9	10	11
3	6	7	12	13	14	15
4	8	9	16	17	18	19
5	10	11	20	21	22	23
6	12	13	24	25	26	27
7	14	15	28	29	30	31
8	16	17	32	33	34	35
9	18	19	36	37	38	39
10	20	21	40	41	42	43
11	22	23	44	45	46	47
12	24	25	48	49	50	51
13	26	27	52	53	54	55
14	28	29	56	57	58	59
15	30	31	60	61	62	63
16	32	33	64	65	66	67
17	34	35	68	69	70	71
18	36	37	72	73	74	75
19	38	39	76	77	78	79
20	40	41	80	81	82	83
21	42	43	84	85	86	87
22	44	45	88	89	90	91
23	46	47	92	93	94	95
24	48	49	96	97	98	99
25	50	51	100	101	102	103
26	52	53	104	105	106	107
27	54	55	108	109	110	111
28	56	57	112	113	114	115
29	58	59	116	117	118	119
30	60	61	120	121	122	123
31	62	63	124	125	126	127
32	64	65	128	129	130	131
33	66	67	132	133	134	135
34	68	69	136	137	138	139
35	70	71	140	141	142	143
36	72	73	144	145	146	147
37	74	75	148	149	150	151
38	76	77	152	153	154	155
39	78	79	156	157	158	159
40	80	81	160	161	162	163
41	82	83	164	165	166	167
42	84	85	168	169	170	171
43	86	87	172	173	174	175
44	88	89	176	177	178	179
45	90	91	180	181	182	183
46	92	93	184	185	186	187
47	94	95	188	189	190	191
48	96	97	192	193	194	195
49	98	99	196	197	198	199
50	100	101	200	201	202	203
Overlapping is “little-endian” for byte and word order						

Overlapping Memory: aw[0] is the least significant word of al[0], likewise ab[0] is the least significant byte of aw[0] and al[0].

Appendix D: Data Variables Memory Map

Integer Variable Memory Non-Overlapping:

Name	Quantity	Type
a-z	26	32 bit signed
aa-zz	26	32 bit signed
aaa-zzz	26	32 bit signed
78 total letter variables		

Float Variable Memory:

Name	Quantity	Type
af[0]-af[7]	8	64 bit IEEE-754
8 total floating-point		

Moving Back and Forth

This is a simple program, used to set tuning parameters and create an infinite loop that causes the motor to move back and forth. Make note of the **TWAIT** commands used to pause program execution during the moves.

```
EIGN(W,0)       'Disable hardware limits
ZS              'Clear faults
ADT=100         'Set maximum acceleration
VT=1000000      'Set maximum velocity
MP              'Set Position Mode
C10             'Place a label
  PT=100000     'Set position
  G             'Start motion
  TWAIT         'Wait for move to complete
  PT=0          'Set position
  G             'Start motion
  TWAIT         'Wait for move to complete
GOTO(10)        'Loop back to label 10
END             'Obligatory END (never reached)
```

Double space indentation within conditional statements or loops make programs significantly more readable.

Moving Back and Forth with Watch

The following example is identical to the previous, except that instead of pausing program execution during the move with the **TWAIT**, a subroutine is used to monitor for excessive load during the moves. This is an important distinction insofar as most SmartMotor programs should have the ability to react to events during motion.

```
EIGN(W,0)       'Disable hardware limits
ZS              'Clear faults
ADT=100         'Set maximum acceleration
VT=100000       'Set maximum velocity
MP              'Set Position Mode
C1              'Place a label
  PT=100000     'Set position
  G             'Start motion
  GOSUB(10)     'Call wait subroutine
  PT=0          'Set position
  G             'Start motion
  GOSUB(10)     'Call wait subroutine
GOTO(1)         'Loop back to label 10
END             'Obligatory END (never reached)
' ****Subroutine
C10
  WHILE Bt      'Loop while trajectory in progress
    IF ABS(EA)>100 'Test for excessive load
      PRINT("Excessive Load",#13) 'Print warning
    ENDIF       'End test
  LOOP          'Loop back to While during motion
RETURN          'Loop back to label 10
```

Appendix E: Example Programs

Homing Against a Hard Stop

Because the SmartMotor has the capability of lowering its own power level and reading its position error, it can be programmed to gently feel for the end of travel as a means to develop a consistent home position subsequent to each power-up. The following program lowers the current limit, moves against a limit, looks for resistance, declares and moves to a home just 100 counts inside the hard limit. Machine reliability is heavily rooted in the process of eliminating potential sources of failure, and eliminating a home switch and its associated cable does well to leverage SmartMotor benefits toward increasing machine reliability.

SmartMotors present a unique opportunity to eliminate the failure mode of a faulty home switch or cable.

```
MDS              'Using Sine Mode commutation
KP=3200          'Increase stiffness from default
KD=10200         'Increase damping from default
F                'Activate new tuning parameters
AMPS=100         'Lower current limit to 10%
VT=-10000        'Set maximum velocity
ADT=100          'Set maximum acceleration
MV               'Set Velocity Mode
G                'Start motion
WHILE EA>-100    'Loop while position error is small
LOOP             'Loop back to WHILE
O=-100           'While pressed, declare home offset
S                'Abruptly stop trajectory
MP               'Switch to Position Mode
VT=20000         'Set higher maximum velocity
PT=0             'Set target position to be home
G                'Start motion
TWAIT            'Wait for motion to complete
AMPS=1000        'Restore current limit to maximum
END              'End Program
```

Homing to the Index

Each SmartMotor has an encoder with an index marker at one angle. This marker can be useful in establishing repeatable startup positions. The following example moves in the negative direction until the index marker is seen. It then decelerates to a stop and reverses until it aligns with the index marker.

```
EIGN(W,0)
O=0
ADT=100          'Set maximum acceleration
VT=10000         'Set maximum velocity
MP               'Set to Mode Position
PRT=20           'Move off in case on index
G                'Start motion
TWAIT            'Wait for motion to complete
i=I(0)           'Clear index flag by read
Ai(0)               'Arm the index register
PRT=-4000        'Set 1 rev, specific to motor
G                'Start motion
```

```
WHILE Bi(0)==0 'Wait for index flag to be true
LOOP           'Loop back to Wait
X              'Decelerate to stop
TWAIT          'Wait for motion to complete
PT=I(0)        'Set target position for Index
G              'Start motion
TWAIT          'Wait for motion to complete
O=0            'Declare current position home
END            'End program
```

Analog Velocity

This example causes the SmartMotor's velocity to track an analog input. Analog signals drift and dither, so a dead-band feature has been added to maintain a stable velocity when the operator is not changing the signal. There is also a wait feature to slow the speed of the loop.

```
EIGN(W,0)      'Disable hardware limits
KP=3020        'Increase stiffness from default
KD=10010       'Increase damping from default
F              'Activate new tuning parameters
ADT=100        'Set maximum acceleration
MV             'Set to Mode Velocity
d=10           'Analog dead band, 5000 = full scale
o=2500         'Offset to allow negative swings
m=40           'Multiplier for speed
w=10           'Time delay between reads
b=0            'Seed b
C10            'Label to create infinite loop
a=INA(V1,3)-o  'Take analog 5Volt FS reading
x=a-b          'Set x to determine change in input
IF x>d         'Check if change beyond deadband
  VT=b*m       'Multiplier for appropriate speed
  G            'Initiate new velocity
ELSEIF x<-d    'Check if change beyond deadband
  VT=b*m       'Multiplier for appropriate speed
  G            'Initiate new velocity
ENDIF          'End If statement
b=a            'Update b for prevention of hunting
WAIT=w         'Pause before next read
GOTO10         'Loop back to label
END            'Obligatory END (never reached)
```

Long Term Variable Storage

Each SmartMotor is equipped with a kind of solid-state disk drive called EEPROM reserved just for long term data storage and retrieval. Data stored in the EEPROM will remain even after power cycling, just like the SmartMotor's program itself. EEPROM has limitations however. It cannot be written to more than about one million times without being damaged. That may seem like a lot, but if a write command (VST) is used in a fast loop, this number can be exceeded in a short time. It is the responsibility of the programmer to see that the memory limitations are considered. The following example is a subroutine

to be called whenever there is a limit contact. It presumes that the memory locations were first seeded with zero.

*This example is a subroutine. It would be called with the command **GOSUB10**.*

```
C10             'Subroutine label
  EPTR=100      'Set EEPROM pointer in memory
  VLD (aa,2)    'Load 2 long variables from EEPROM
  IF Br         'If right limit, then...
    aa=aa+1     'Increment variable aa
    Zr          'Reset right limit state flag
  ENDIF
  IF Bl         'If left limit, then...
    bb=bb+1     'Increment variable bb
    Zl          'Reset left limit state flag
  ENDIF
  EPTR=100      'Reset EEPROM pointer in memory
  VST(aa,2)     'Store variables aa and bb
RETURN          'Return to subroutine call
```

Look for Errors and Print Them

This code example looks at different error status bits and prints appropriate error information to the RS-232 channel.

*This example is a subroutine. It would be called with the command **GOSUB10**.*

```
C10             'Subroutine label
  IF Be         'Check for position error
    PRINT("Position Error", #13)
  ENDIF
  IF Bh         'Check for over temp error
    PRINT("Over Temp Error",#13)
  ENDIF
  IF Ba         'Check for over current error
    PRINT("Over Current Error",#13)
  ENDIF
RETURN          'Return to subroutine call
```

Changing Speed upon Digital Input

SmartMotors have digital I/O that can be used for many purposes. In this example, a position move is started and the speed is increased by 50% if input A goes low.

```
EIGN(W,0)       'Disable hardware limit IO
KD=10010        'Changing KD term in tuning
F               'Accept new KD
O=0             'Reset origin
ADT=100         'Set maximum acceleration 'ATTENTION
VT=10000        'Set maximum velocity 'ATTENTION
PT=40000        'Set final position
MP              'Set Position Mode
G               'Start motion
WHILE Bt        'Loop while motion continues
IF IN(0)==0     'If input is low
```

```
  IF VT==10000 'Check VT so change happens once
    VT=12000   'Set new velocity 'ATTENTION
    G          'Initiate new velocity
  ENDIF
ENDIF
LOOP           'Loop back to WHILE
END
```

Pulse Output Upon a Given Position

It is often necessary to fire an output upon a certain position. There are many ways to do this with a SmartMotor. This example sets I/O B as an output while first making sure it comes up 1 by presetting the output value, then watches the encoder position until it exceeds 20000.

```
EIGN(W,0)          'Disable limits
ZS
ITR(0,4,0,0,1)     'ITR(int#,sw,bit,state,lbl)
ITRE
EITR(0)
OUT(1)=1           'Set I(0)/O B to output, high
ADT=100            'Set maximum acceleration
VT=100000          'Set maximum velocity
MP                 'Set Position Mode
'****Main Program Body
WHILE  1>0
  O=0              'Reset origin for move
  PT=40000         'Set final position
  G                'Start motion
  WHILE PA<20000   'Loop while motion continues
  LOOP             'Wait for desired position to pass
  OUT(1)=0         'Set output lo
  TMR(0,400)       'Use timer 0 for pulse width
  TWAIT
  WAIT=1000        'wait 1 second
  LOOP
END
'****Interrupt Subroutine
C1
OUT(1)=1           'set output high again
RETURNI
```

Stop Motion if Voltage Drops

The Voltage, Current and Temperature of a SmartMotor are always known and can be used within a program to react to changes. In this program, the SmartMotor begins a move and then stops motion if the voltage falls below 18.5 volts.

```
EIGN(W,0)      'Disable hardware limits
ZS             'Clear faults
MDS            'Sine Mode Commutation
ADT=100        'Set maximum acceleration
```

```
VT=100000       'Set maximum velocity
PT=1000000      'Set final position
MP              'Set Position Mode
G               'Start motion
WHILE Bt        'Loop while motion continues
  IF UJA<18500  'If voltage is below 18.5 Volts
    OFF         'Turn motor off
  ENDIF
LOOP            'Loop back to WHILE
END             'Obligatory END
```

Status Word: 0 SW(0) Primary Fault/Status Indicator					
Bit	**Value**	**Type**		**Clear**	**Description**
0	1	Indicator			Drive ready – no faults exist and enough bus voltage
1	2	Indicator	Bo		Motor is off
2	4	Indicator	Bt		Trajectory in progress
3	8	Fault			Servo bus voltage fault, set on regen fault, or while running with low bus.
4	16	Historical	Ba	Za	Peak over-current occurred
5	32	Fault	Bh		Excessive temperature, requires 5 deg C below TH setting and user clear of this bit.
6	64	Fault	Be	Ze	Excessive position error
7	128	Fault	Bv	Zv	Velocity limit
8	256	Indicator			Real-time temperature limit
9	512	Fault			First derivative (DE/Dt) of position error over limit
10	1024	Indicator			Hardware right (+) over travel limit enabled
11	2048	Indicator			Hardware left (-) over travel limit enabled
12	4096	Historical	Br	Zs	Right (+) over travel limit
13	8192	Historical	Bl	Zl	Left (-) over travel limit
14	16384	Indicator	Bp		Right (+) over travel limit active
15	32768	Indicator	Bm		Left (-) over travel limit active

Appendix F: Status Words

Status Word: 1	SW(1)	Index Registration and Software Travel Limits			
Bit	**Value**	**Type**			**Description**
0	1	Indicator			Arming Bit for Rise Capture of Encoder 0
1	2	Indicator			Arming Bit for Fall Capture of Encoder 0
2	4	Historical	Bi(0)		Rising Edge Capture on Encoder 0
3	8	Historical	Bj(0)		Falling Edge Capture on Encoder 0
4	16	Indicator			Arming Bit for Rise Capture of Encoder 1
5	32	Indicator			Arming Bit for Fall Capture of Encoder 1
6	64	Historical	Bi(1)		Rising Edge Capture on Encoder 1
7	128	Historical	Bj(1)		Falling Edge Capture on Encoder 1
8	256	Indicator	Bx(0)		Capture Input State 0
9	512	Indicator	Bx(1)		Capture Input State 1
10	1024	Indicator			Software Over Trave Limits Enabled
11	2048	Indicator			Software Over Trave Limit Mode: [0: Don't Stop] [1: Fault will occure MTB issued, Default]
12	4096	Historical	Brs	Zrs	Software Positive Over Travel Limit Occurred
13	8192	Historical	Bls	Zls	Software Negative Over Travel Limit Occurred
14	16384	Indicator	Bps		Software Positive Over Travel Limit Active
15	32768	Indicator	Bms		Software Negative Over Travel Limit Active

Status Word: 2 SW(2) Communications, Program and Memory					
Bit	Value	Type			Description
0	1	Indicator			Com Channel 0 (RS-232) General Error; Use RCHN(0) to get full status
1	2	Indicator			Com Channel 1 (RS-485) General Error; Use RCHN(1) to get full status
2	4	Indicator			Reserved
3	8	Indicator			Reserved
4	16	Indicator			CAN Bus Error; Use RCAN
5	32	Indicator			Reserved
6	64	Indicator			Ethernet Error
7	128	Indicator			I2C Running
8	256	Indicator			Reserved
9	512	Indicator			Animatics Data Block Checksum Error
10	1024	Indicator			Program RUNNING !
11	2048	Indicator			Trace In Progress (Currently in Alpha Test)
12	4096	Historical			EEPROM Write Buffer Overflow, Last Write to EEPROM exceeded buffer and was denied
13	8192	Indicator			EEPROM Busy (Write In Progress)
14	16384	Historical	Bs	Zs	Command Syntax Error NOTE !!!! See ERRC Command Errors Info
15	32768	Indicator	Bk		Main Program Checksum Error: Program is Corrupt and cannot run

Appendix F: Status Words

Status Word: 3	SW(3)	PID State, Brake, Move Generation Indicators			
Bit	Value	Type			Description
0	1	Historical			Position Error has Exceeded Software Limt
1	2	Indicator			Torque Saturation; Drive is Running at 100% PWM
2	4	Indicator			Voltage Saturation; Max Bus Voltage!
3	8	Historical	Bw	Zw	Wrap around Occurred; Position wrapped through +/- 2^31
4	16	Indicator			KG (Gravitational Offset Gain) enabled
5	32	Indicator			Shaft Direction
6	64	Indicator			PID Torque Direction
7	128	Historical		ZS	I/O Fault Latch
8	256	Indicator			Relative Move
9	512				
10	1024				
11	2048	Indicator			Modulo Rollover
12	4096	Indicator			Brake Asserted
13	8192	Indicator			Brake OK, Is Internally present or configured to an external I/O point
14	16384	Indicator			"G" command has been configured to an Input
15	32768	Indicator			Velocity Target Reached

Status Word: 4 SW(4) Interrupt Timers			
Bit	Value	Type	Description
0	1	Indicator	Timer 0 Running: (Not timed out yet)
1	2	Indicator	Timer 1 Running: (Not timed out yet)
2	4	Indicator	Timer 2 Running: (Not timed out yet)
3	8	Indicator	Timer 3 Running: (Not timed out yet)
4	16		
5	32		
6	64		
7	128		
8	256		
9	512		
10	1024		
11	2048		
12	4096		
13	8192		
14	16384		
15	32768		

Status Word: 5	SW(5)	Interrupt Status Indicators	
Bit	Value	Type	Description
0	1	Indicator	Interrupt 0 Enabled
1	2	Indicator	Interrupt 1 Enabled
2	4	Indicator	Interrupt 2 Enabled
3	8	Indicator	Interrupt 3 Enabled
4	16	Indicator	Interrupt 4 Enabled
5	32	Indicator	Interrupt 5 Enabled
6	64	Indicator	Interrupt 6 Enabled
7	128	Indicator	Interrupt 7 Enabled
8	256		
9	512		
10	1024		
11	2048		
12	4096		
13	8192		
14	16384		
15	32768	Indicator	Interrupt Event Scanner Enabled

Status Word: 6 SW(6) Drive Modes			
Bit	Value	Type	Description
0	1	Indicator	Running Standard Trapezoidal Mode (Direct From Hardware Commutation)
1	2	Indicator	Running Enhanced Trapezoidal Mode (Encoder Position Emulated Commutation)
2	4	Indicator	Running Sinusoidal Commutation
3	8	Indicator	Sine current mode
4	16		
5	32		
6	64		
7	128		
8	256	Indicator	Commutation Calibration OK (should =1 after first detection of internal index mark)
9	512	Indicator	TOB (Trajectory Overshoot Braking) Enabled
10	1024		Commutation Is Inverted (MINV(1) has been issued)
11	2048		MTB (Mode Torque Brake) is active
12	4096	Indicator	Encoder battery fault
13	8192	Indicator	Low bus indicator
14	16384	Historical	High bus latched ind.
15	32768	Indicator	Shunt Active

Status Word 7: SW(7) - Multiple Trajectory Support			
Bit	Value	Type	Description
0	1	Indicator	Trajectory 1 In Progress
1	2	Indicator	Trajectory 1 Accel Phase
2	4	Indicator	Trajectory 1 Slew Phase
3	8	Indicator	Trajectory 1 Decel Phase
4	16	Indicator	Reserved
5	32	Indicator	Reserved
6	64	Indicator	Reserved
7	128	Indicator	Reserved
8	256	Indicator	Trajectory 2 in Progress
9	512	Indicator	Ascend Segment
10	1024	Indicator	Slew Segment
11	2048	Indicator	Descend Segment
12	4096	Indicator	Dwell Segment
13	8192	Indicator	Reserved
14	16384	Indicator	Reserved
15	32768	Indicator	TSWAIT

Status Word 8: SW(8) – Cam Support			
Bit	Value	Type	Description
0	1	Indicator	Cam User Bit 0
1	2	Indicator	Cam User Bit 1
2	4	Indicator	Cam User Bit 2
3	8	Indicator	Cam User Bit 3
4	16	Indicator	Cam User Bit 4
5	32	Indicator	Cam User Bit 5
6	64	Indicator	Cam Segment Mode 0
7	128	Indicator	Cam Segment Mode 1
8	256	Indicator	Interpolation User Bit 0
9	512	Indicator	Interpolation User Bit 1
10	1024	Indicator	Interpolation User Bit 2
11	2048	Indicator	Interpolation User Bit 3
12	4096	Indicator	Interpolation User Bit 4
13	8192	Indicator	Interpolation User Bit 5
14	16384	Indicator	Interpolation Seg Mode 0
15	32768	Indicator	Interpolation Seg mode 1

Status Word 12: SW(12) – User Bits Word 0			
Bit	Value	Type	Description
0	1	Set/Reset	User Bit 0
1	2	Set/Reset	User Bit 1
2	4	Set/Reset	User Bit 2
3	8	Set/Reset	User Bit 3
4	16	Set/Reset	User Bit 4
5	32	Set/Reset	User Bit 5
6	64	Set/Reset	User Bit 6
7	128	Set/Reset	User Bit 7
8	256	Set/Reset	User Bit 8
9	512	Set/Reset	User Bit 9
10	1024	Set/Reset	User Bit 10
11	2048	Set/Reset	User Bit 11
12	4096	Set/Reset	User Bit 12
13	8192	Set/Reset	User Bit 13
14	16384	Set/Reset	User Bit 14
15	32768	Set/Reset	User Bit 15
Status Word 13: SW(13) – User Bits Word 1			
Bit	Value	Type	Description
0	1	Set/Reset	User Bit 16
1	2	Set/Reset	User Bit 17
2	4	Set/Reset	User Bit 18
3	8	Set/Reset	User Bit 19
4	16	Set/Reset	User Bit 20
5	32	Set/Reset	User Bit 21
6	64	Set/Reset	User Bit 22
7	128	Set/Reset	User Bit 23
8	256	Set/Reset	User Bit 24
9	512	Set/Reset	User Bit 25
10	1024	Set/Reset	User Bit 26
11	2048	Set/Reset	User Bit 27
12	4096	Set/Reset	User Bit 28
13	8192	Set/Reset	User Bit 29
14	16384	Set/Reset	User Bit 30
15	32768	Set/Reset	User Bit 31

Status Word: 16 SW(16) On Board Local I/O Status					
Bit	Value	I/O	Pin	Port	Description
0	1	0	1	A	
1	2	1	2	B	
2	4	2	3	C	
3	8	3	4	D	
4	16	4	5	E	
5	32	5	6	F	
6	64	6	7	G	
7	128	7	-	-	Internal I/O point may be set for purposes of triggering Interrupts
8	256				
9	512				
10	1024				
11	2048				
12	4096				
13	8192				
14	16384				
15	32768				

Appendix F: Status Words

Status Word: 17 SW(17) Expanded I/O Status				
Bit	Value	I/O	Pin	Description
0	1	16	1	
1	2	17	2	
2	4	18	3	
3	8	19	4	
4	16	20	5	
5	32	21	6	
6	64	22	7	
7	128	23	8	
8	256	24	9	
9	512	25	10	
10	1024			
11	2048			
12	4096			
13	8192			
14	16384			
15	32768			