



The



Series 4 User's Guide

©2001, 2002, 2003, 2004, 2005, 2007, 2008 Animatics Corp.
All rights reserved

Animatics SmartMotor™ Series 4 User's Guide, Revision 5.

This manual, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The content of this manual is furnished for informational use only, is subject to change without notice and should not be construed as a commitment by Animatics Corporation. Animatics Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear herein.

Except as permitted by such license, no part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Animatics Corporation.

Animatics, the Animatics logo, SmartMotor and the SmartMotor logo are all trademarks of Animatics Corporation. Windows, Windows 95/98, Windows 2000, Windows NT and XP are all trademarks of Microsoft Corporation.

Animatics' SmartMotor is patented by US Patent # 5,912,541.

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

Animatics is Defining the Future in Motion Control!

TABLE OF CONTENTS

SMARTMOTOR THEORY OF OPERATION	7
Motion Control Functions	7
System Control Functions	7
Communication Functions	8
I/O Functions	8
 QUICK START	 9
Software Installation	10
SmartMotor Background	10
A quick look at the SmartMotor Interface	11
 PROGRAMMING TABLE OF CONTENTS	 17
Creating Motion	21
Program Flow	31
Variables	39
Reporting Commands	43
Encoder and Pulse Train Following	47
System State Flags	55
Inputs and Outputs	57
Communications	69
The PID Filter	79
 SMI ADVANCED FUNCTIONS	 85
SMI Software	85
SMI Projects	85
Terminal Window	85
Configuration Window	86
Program Editor	86
Information Window	87
Serial Data Analyzer	87
Motor View	88

TABLE OF CONTENTS

Continued from preceding page

Monitor Window	88
Chart View	89
Macros	89
Tuner	90
SMI Options	92
SMI Help	92
APPENDIX A	95
ASCII Character Set	95
APPENDIX B	97
Binary Data	97
APPENDIX C	101
Commands	101
APPENDIX D	111
Data Variables Memory Map	111
APPENDIX E	113
Example Programs	113
Moving back and forth	113
Moving back and forth with watch	113
Homing against a hard stop	114
Homing to the index	114
Analog Velocity	115
Long term variable storage	116
Look for errors and print them	116
Changing speed upon digital input	116
Pulse output upon a given position	117

TABLE OF CONTENTS

Stop motion if voltage drops	117
Measuring command execution time	118
Custom parser with checksum	119
APPENDIX F	125
F= Commands	125

This page has been intentionally left blank.

SMARTMOTOR THEORY OF OPERATION

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, a "Go" signal, or statement 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 more than 4,000 times per second for maximum performance.

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 "acceleration" rate such that the moment it comes to rest, it will be at the programmed target position. The PID filter 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.

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, CNC 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, can eliminate the need for a PLC (Programmable Logic Controller).

SmartMotors have numerous I/O incorporating multiple functions. Clever

*Optional
SmartMotor™ cable
(CBLSM1-10)*

*Optional PS24V8A or
PS48V6A power
supply*

*Many vertical
applications require
a SHUNT to protect
the SmartMotor
from damage*

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

SmartMotors come standard with RS-232 and/or RS-485 communication ports. These ports can be used to connect SmartMotors together, and/or to a host computer or PLC. In addition to these networks, SmartMotors are also available with a number of industry standard control networks such as CANopen, DeviceNET, Profibus, USB, Ethernet, Ethernet/IP, 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 fieldbusses 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. 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: "**RP**", 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.

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.

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

1. A SmartMotor™
2. A computer running MS Windows 95/98, 2000, NT or XP
3. A DC power supply for those SmartMotors that require DC voltage.
4. A data cable to connect the SmartMotor to the computer's serial port or serial adapter.
5. Host level software to communicate with the SmartMotor

The first time user of the SM1700 through SM3400 series motors should purchase the Animatics SMDEVPACK. 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 DC power supplies available for Series 4 SmartMotors: PS24V8A (24 Volt, 8 Amp) and PS42V6A (42 Volt, 6 Amp). ServoStep SmartMotors operate up to 75VDC. They can use any of the power supplies, plus higher voltage supplies. For any particular motor, more Torque and Speed is available with higher voltage.

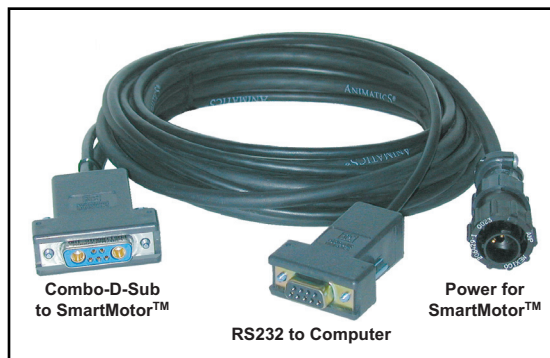


When relying on Torque/Speed curves, pay close attention to the voltage on which they are based. Also, special care must be taken when near the upper voltage limit or in vertical applications that can back-drive the SmartMotor. Gravity influenced applications can turn the SmartMotor into a generator 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.

For the AC SmartMotors, SM4200 through SM5600 series, Animatics offers:

- | | |
|-------------|---|
| CBLSMA1-10 | 10' communication cable |
| CBLAC110-10 | 10' 110 volt AC single phase power cord |
| CBLAC200-10 | 10' 208-230 volt AC 3 phase power cord |



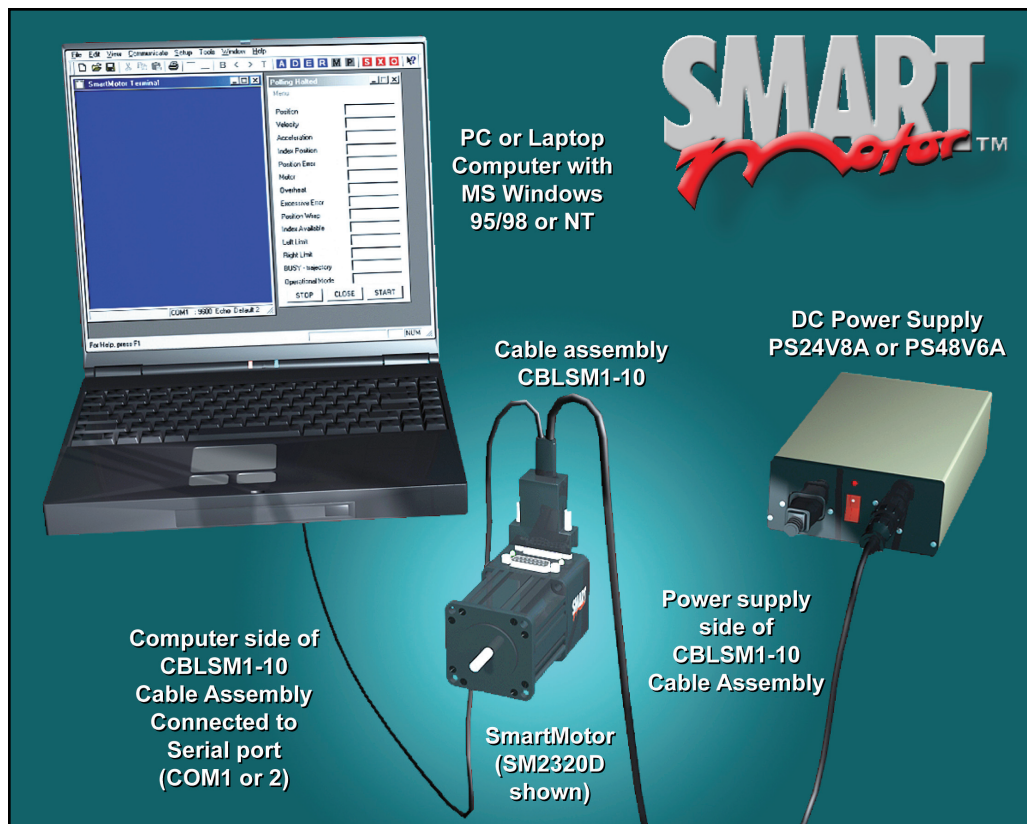
*Optional
SmartMotor™ cable
(CBLSM1-10)*

*Optional PS24V8A or
PS42V6A power
supply*

***Many vertical
applications require
a SHUNT to protect
the SmartMotor
from damage***

QUICK START

Connecting a SM2320D SmartMotor using a CBLSM1-10 cable assembly and PS24V8A power supply



SOFTWARE INSTALLATION

Follow standard procedures for software installation using either the Animatics SMI CD-ROM or files downloaded from the Animatics Website at www.animatics.com.

After the software is installed, be sure to reset your computer before running the SMI program.

With the SMI Software loaded and your SmartMotor connected as shown above, you are ready to start making motion. Turn the SmartMotor's power on and start the SMI Program.

SMARTMOTOR BACKGROUND

The SmartMotor is an entire Servo Control System in a single component. Of course, it's shaft position, velocity and acceleration are programmable but there is much more. The SmartMotor also has analog and digital I/O and can be programmed to operate by itself in a language similar to Basic. The same commands one would use to program a SmartMotor can be sent to it over RS-232, or RS-485, depending on your product selection. These commands, explained later in this guide, can be sent using most any host terminal software, but the SMI "SmartMotor Interface" program does this and much more.

A QUICK LOOK AT THE SMARTMOTOR INTERFACE

The 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 the ability to write programs and download them into the SmartMotor's long-term memory.

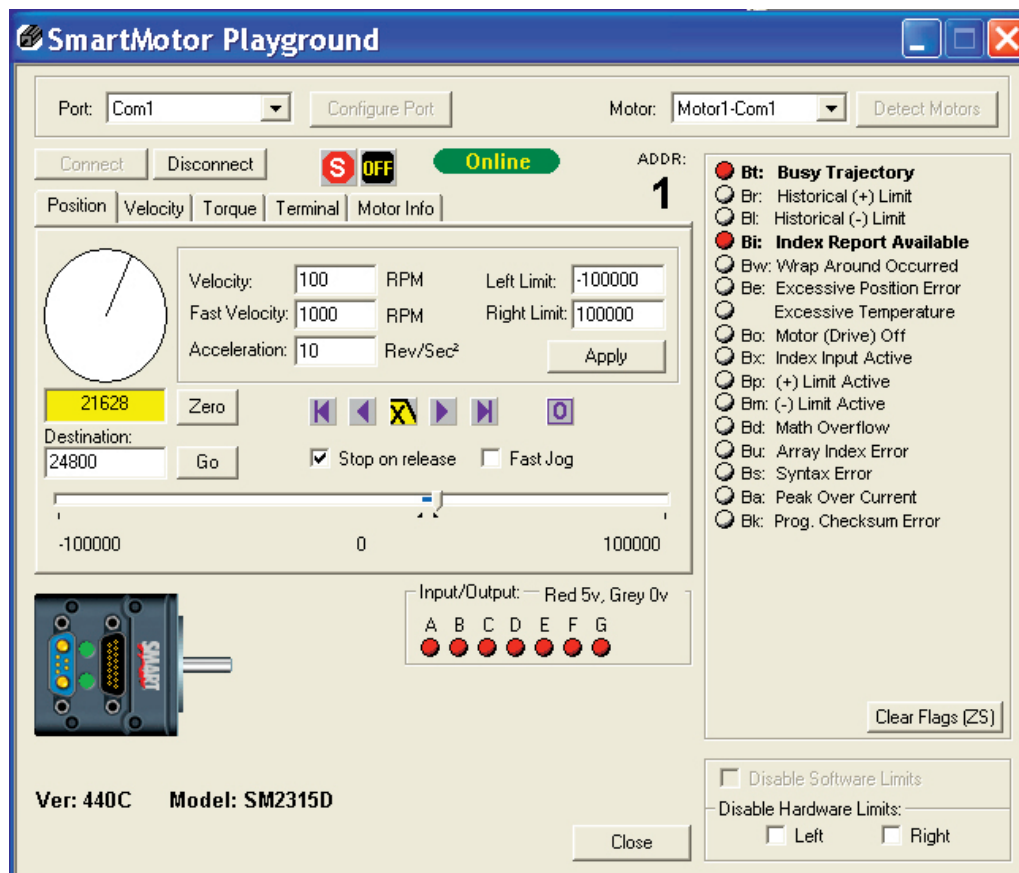
For the benefit of the first-time user, the SMI software starts with the "SmartMotor Playground". If you are using a ServoStep or other RS-485 based SmartMotor, start by clicking on the "Configure Port" button and select "RS-485".

Now, click in the "Detect Motors" button in the upper-right. If your SmartMotor is not properly detected, use the utility to the upper left to select the more appropriate COM port. If you still have no success, it is likely that your computer is not configured properly for RS-232 communications. This problem should be cor-

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

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

*If you are using a SmartMotor with **PLUS** firmware or a ServoStep, 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 hazzard.*



rected, 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 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

QUICK START

MotorView gives you a window into the status of a SmartMotor

PLUS and ServoStep Firmware require the Limit Inputs to be either tied low, or disabled to achieve motion.

Acceleration, Velocity and Position fully describe a trapezoidal motion profile

"Close" button at the bottom and launch the SMI development software.

LEARNING THE SMARTMOTOR INTERFACE (SMI)

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

With your motor connected and on, click on the purple **A** 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

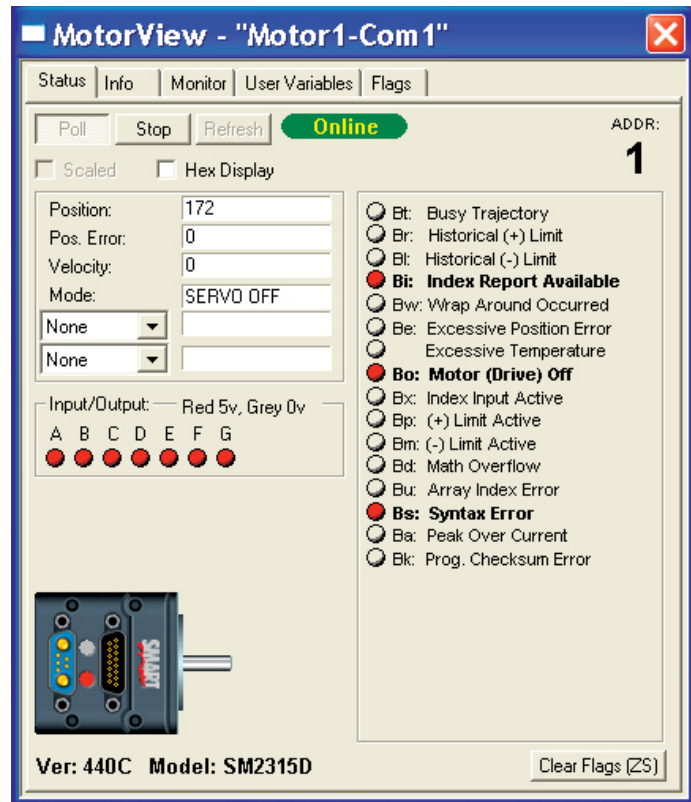
To see the status of the connected motor, go to the **"Tools"** menu, select **"Motor View"** and double click on the available motor. Once the MotorView box appears, press the **"Poll"** button.

SmartMotors with PLUS Firmware and Servo-Step require limits to be connected before the motor will operate. If you see limit errors, and you want to move the motor anyway, you don't have to install limits. Instead, you can re-define the Limit Inputs as

General Inputs, and 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).

```
UCI    'Configure Port C (limit) as general input
UDI    'Configure Port D (limit) as general input
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.




Initiating motion

To get the motor to make a trajectory, enter the following into the **Terminal**.

```
A=100      'sets the Acceleration
V=1000000  'sets the maximum Velocity
P=300000   'sets the target Absolute Position
G          'Go, initiates motor movement
```

After the final **G** command has been entered, the SmartMotor will accelerate up to speed, slew and then decelerate to a stop at the absolute target position. The progress can be seen in the **MotorView**.

Writing a user program


In addition to taking commands over the serial interface, SmartMotors 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.

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:

```
A=100      'Set buffered acceleration
V=1000000  'Set buffered velocity
P=300000   'Set buffered relative move
G          'Start Motion
TWAIT     'Wait for move to complete
P=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

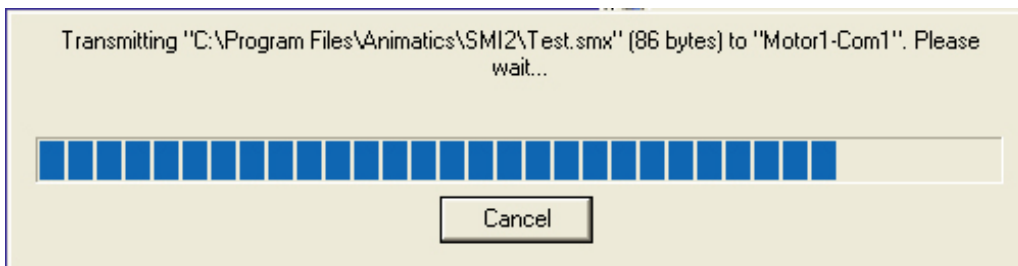
Before transmitting the program, press the **STOP** button in the MotorView window. To check the program and transmit it to the SmartMotor, click on the  button located on the tool bar. A small window will ask what motor you want to download to. Simply select the only motor presented. SMI2 compiles the program during this step as well, so if errors may be found in the file. If errors are found, make the necessary corrections and try again.


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

1000000 Scaled Counts/Sample= about 1860 RPM for SM2300 series motors, about 930 RPM for series SM3400, 4200 and 5600 motors, and about 465 RPM for ServoStep motors.

QUICK START

SMI2 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 300000. Press the **RUN** () 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 **MotorView** 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.

Tuning the Motor
Most SmartMotors show more than adequate performance with the same tuning parameters. This is largely due to the all-digital design.

Position:	126904
Pos. Error:	355
Velocity:	1000000
Mode:	ABSOLUTE
	
	

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 show that the motion, so far, is a bit sloppy. That is because the motor's **PID** Filter is tuned by default to be stable in almost

any start-up environment. Try issuing the following commands in the **Terminal** and run the program again:



```

KP=200      `Increase Proportional Gain (P) (Stiffness)
KD=600      `Increase Derivative Gain (D) (Dampening)
F           `Update PID Filter
```

The motor shaft position should feel and appear much stiffer now. More can be done, however, to make the shaft settle faster and be more accurate. Issue the following commands to increase what is called the "Integral Gain":

```

KI=100      `Increase Integral
Gain (I)
KL=100      `Increase I Limit
F           `Update PID filter
```

Position:	167846
Pos. Error:	25
Velocity:	1000000
Mode:	ABSOLUTE
	
	

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

By running the program with the **MotorView** on, you will see improved results. Note the lower Position Error. For most applications, these parameters will suffice, but if still greater precision is required, more can be found on the topic of tuning later in this manual in the section on tuning. Also, the Tools menu has a **Tuning** utility that can be further useful. 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.

Many are surprised at the vast array of different parameters the SmartMotor finds stable. SmartMotors are so much more forgiving than traditional controls because of their 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 its 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.

This page has been intentionally left blank.

PROGRAMMING TABLE OF CONTENTS

CREATING MOTION 21

A=exp	Set absolute acceleration	21
V=exp	Set maximum permitted velocity	22
P=exp	Set absolute position for move	22
D=exp	Set relative distance for position move	22
G	Go, start motion	23
S	Abruptly stop motion in progress	23
X	Decelerate to stop	23
O=exp	Set/reset origin to any position	23
OFF	Turn motor servo off	23
MP	Position mode	24
MV	Velocity mode	24
MT	Torque mode	24
T=exp	Set torque value	25
MD	Contouring mode	25
MD50	Drive Mode	29
BRK...	Brake Commands	29
MTC, G, I	Re-route brake signal	30
MTB	Mode Torque Brake	30

PROGRAM FLOW 31

RUN	Execute stored user program	31
RUN?	Halt program if no RUN issued	31
GOTO#	Redirect program flow	31
C#	Subroutine label	31
END	End program execution	32
GOSUB#	Execute a subroutine	32
RETURN	Return from subroutine	32
F=32, F=64	Interrupt subroutines	33
IF, ENDIF	Conditional test	34
ELSE, ELSEIF	Conditional alternate test	34

PROGRAMMING TABLE OF CONTENTS

WHILE, LOOP	Conditional loop	35
SWITCH, CASE, DEFAULT, BREAK, ENDS		36
TWAIT	Wait during trajectory	37
WAIT=exp	Wait (exp) sample periods	37
STACK	Reset the GOSUB return stack	37
VARIABLES		39
Arrays		39
Storage of Variables		41
EPTR=exp	Set EEPROM pointer	41
VST(var,index)	Store variables	41
VLD(var,index)	Load variables	41
Fixed or Pre-assigned variables		41
Variable Space Restrictions		42
REPORTING COMMANDS		42
Report to Host Commands		42
ENCODER AND PULSE TRAIN FOLLOWING		47
MF1, MF2, MF4	Mode Follow	47
MF0, MS0		47
MFDIV=exp	Set Ratio divisor	47
MF MUL=exp	Set Ratio multiplier	47
MFR	Calculate Mode Follow Ratio	48
MSR	Calculate Mode Step Ratio	48
MC	Mode Cam	48
BASE=exp	Base Length	48
SIZE=exp	Number of Cam Data Entries	48
CI	Cam Initialize	49
CX	Value of Current Cam Index	49
F=16	Cam Relative Position Mode	49
F=128	Cam Modulo	51

PROGRAMMING TABLE OF CONTENTS

ENC0, ENC1	Encoder Select	53
SYSTEM STATE FLAGS		55
Reset System State Flags		56
INPUTS AND OUTPUTS		57
The Main RS-232 port		58
The G port		58
Counter Functions of ports A and B		59
General I/O functions of ports A and B		59
The AnaLink port (using I²C protocol)		60
The AnaLink port (using RS-485 protocol)		60
The AnaLink port as general I/O		60
External RS-485 I/O		61
AnaLink I/O modules		62
I/O Connection Examles		63
Motor Connector Pin Identifications		64
Motor Connector Locator		65
Standard I/O Modules		66
Interfacing Standard I/O Modules		67
COMMUNICATIONS		69
Daisy Chaining RS-232		70
SADDR#	Set motor to new address	07
SLEEP, SLEEP1	Assert sleep mode	71
WAKE, WAKE1	De-assert SLEEP	71
ECHO, ECHO1	Echo input	71
ECHO_OFF, ECHO_OFF1	De-assert ECHO	71
Communicating over RS-485		72
OCHN		73
CCHN(type,channel)	Close a COM channel	74
BAUD#	Set Baud rate of main port	74

PRINT(), PRINT1()	74
SILENT, SILENT1 Assert silent mode	74
TALK, TALK1 De-assert silent mode	74
! Wait for RS-232 char. to be received	75
a=CHN0, a=CHN1 RS-485 COM error flags	75
a=ADDR Motor's self address	75
Getting data from a COM port	76
 THE PID FILTER	 79
PID Filter Control	79
Tuning the Filter	80
 CURRENT LIMIT CONTROL	 83

Enter the commands below in the **SmartMotor Terminal** window, following each command with a return, and the SmartMotor will start to move. Note that the ServoStep will make a series of minute motions upon power-up to calibrate the rotor position. This may take about a second and will be confined to a few degrees of motion. When this is complete the following commands can be issues:

Commands	Comments
UCI	`Dis. Limit (required w/Plus Firm.)
UDI	`Dis. Limit (required w/Plus Firm.)
ZS	`Reset Errs (required w/Plus Firm.)
A=100	`Set Maximum Acceleration
V=1000000	`Set Maximum Velocity
P=1000000	`Set Absolute Position
G	`Start move (Go)

On power-up the motor defaults to position mode. Once **Acceleration (A)** and **Velocity (V)** are set, simply issue new **Position (P)** commands, followed by a **G (Go)** 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 **Velocity** and **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.

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

#	Integer number
exp	Expression or signed integer
var	Variable
COM	Communication channel

A=exp Set absolute acceleration

Acceleration must be a positive integer within the range of 0 to 2,147,483,648. The default is zero forcing something to be entered to get motion. A typical value is 100. If left unchanged, while the motor is moving, this value will not only determine acceleration but also deceleration which will form a triangular or trapezoidal velocity motion profile. This value can be changed at any time. The value set does not get acted upon until the next **G** command is executed.

If the motor has a 2000 count encoder (sizes 17 and 23), multiply the desired

A complete move requires the user to set a Position, a Velocity and an Acceleration, followed by a Go.

PLUS and ServoStep Firmware requires the added overhead of disabling limits (if none are connected) and clearing errors before a G can be accepted.

Do not disable limits if such a thing could cause damage or injury.

For SM17 & SM23
 $A = \text{rev/sec}^2 * 7.91$

For SM34, 42 & 56
 $A = \text{rev/sec}^2 * 15.82$

For ServoStep
 $A = \text{rev/sec}^2 * 31.64$

CREATING MOTION

For SM17 & SM23

V=rev/sec * 32212

For SM34, 42 & 56

V=rev/sec * 64424

For ServoStep

V=rev/sec * 128848

For SM17 & SM23

P=rev * 2000

For SM34, 42 & 56

P=rev * 4000

For ServoStep

P=rev * 8000

acceleration, in rev/sec², by 7.91 to arrive at the number to set **A** to. With a 4000 count encoder (sizes 34, 42 and 56) the multiplier is 15.82. ServoSteps use 8000 count encoders, so for them the multiplier is 21.62. These constants are a function of the motors **PID** rate. If the **PID** rate is lowered, these constants must be raised proportionally.

V=exp Set maximum permitted velocity

Use the **V** command to set a limit on the velocity the motor can accelerate to. That limit becomes the slew rate for all trajectory based motion whether in position mode or velocity mode. 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. If the motor has a 2000 count encoder (sizes 17 and 23), multiply the desired velocity in rev/sec by 32212 to arrive at the number to set **V** to. With a 4000 count encoder (sizes 34, 42 & 56) the multiplier is 64424. ServoSteps use 8000 count encoders, so for them the multiplier is 128848. These constants are a function of the motors **PID** rate. If the **PID** rate is lowered, these constants will need to be raised.

P=exp Set absolute position for move

The **P=** command sets an absolute end position. The units are encoder counts and can be positive or negative. 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 2000 increments per revolution while SmartMotor sizes 34, 42 and 56 resolve 4000 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
GOSUB10         'Initiate motion
END             'End program
C10             'Motion routine
  A=a*8          'Set Acceleration
  V=v*32212      'Set Velocity
  P=p*2000       'Set Position
  G              'Start move
RETURN          'Return to call
```

D=exp Set relative distance for position move

The **D=** command allows a relative distance to be specified, instead of an absolute position. The number following is encoder counts and can be positive or negative.

The relative distance will be added to the current position, either during or after a move. It is added to the desired position rather than the actual position so as to avoid the accumulation of small errors due to the fact that any servo motor is seldom exactly where it should be at any instant in time.

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 trigger the next profile at any time. With PLUS or ServoStep Firmware, the **G** will not work until all errors are cleared. The **ZS** command clears all errors in these cases.

***G** also resets several system state flags*

*With PLUS and ServoStep Firmware, errors must be reset before **G** will function. **ZS** will reset all errors.*

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 both Position and Velocity 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 **A=** value. When the motor comes to rest it will servo in place until commanded to move again. The **X** command works in both Position and Velocity modes.

O=exp Set/Reset origin to any position

The **O=** command (using the letter **O**, not the number zero) allows the host or program not just to declare the current position zero, but to declare it to be any position, positive or negative. The exact position to be re-declared is the ideal position, not the actual position which may be changing slightly due to hunting or shaft loading. 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 which will cause the position counter to change sign. While that is not an issue with Velocity Mode, it can create problems in position mode.

OFF Turn motor servo off

The **OFF** command will stop the motor from servoing, much as a position error or limit fault would. When the servo is turned off, one of the status LEDs will revert from Green to Red. Motors with PLUS firmware have a different "off" state; they default to MTB (Mode Torque Brake). Rather than being free-wheeling in their Off state, they are extremely resistive. To make a PLUS SmartMotor free-wheel, issue **BRKRLS** immediately followed by **OFF**.

MP Position Mode

Position mode is the default mode of operation for the SmartMotor. If the mode were to be changed, the **MP** command would put it back into position mode. In position mode, the **P#** and **D#** commands will govern motion.

BINARY POSITION DATA TRANSFER

The **ASCII** based command string format, while convenient, is not the fastest way to communicate data. It can be burdensome when trajectory commands are sent to the motor. For that reason a special binary format has been established for the communication of trajectory critical data such as **Position**, **Velocity** and **Acceleration**. Using the binary format, these 32 bit parameters are sent as four bytes following a code byte that flags the data for a particular purpose. The code bytes are 252 for acceleration, 253 for velocity and 254 for position. As an example, the following byte values communicate A=53, V=-1 & P=2137483648.

A=53	252	000	000	000	053	032
V=-1	253	255	255	255	254	032
P=2137483648	254	127	255	255	255	032

For further expediency, the commands can be appended with the **G** command to start motion immediately. Two examples are as follows (the ASCII value for **G** is 71):

P=0 G	254	000	000	000	000	071	032
V=512 G	253	000	000	002	000	071	032

MV Velocity Mode

Velocity mode will allow continuous rotation of the motor shaft. In Velocity mode, the programmed position using the **P** or the **D** commands is ignored. Acceleration and velocity need to be specified using the **A=** and the **V=** 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.

MT Torque Mode

In torque mode the motor shaft will simply apply a torque independent of position. 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. To specify the amount of torque, use the **T=** command, followed by a number between -1023 and 1023.

T=exp Set torque value, -1023 to 1023

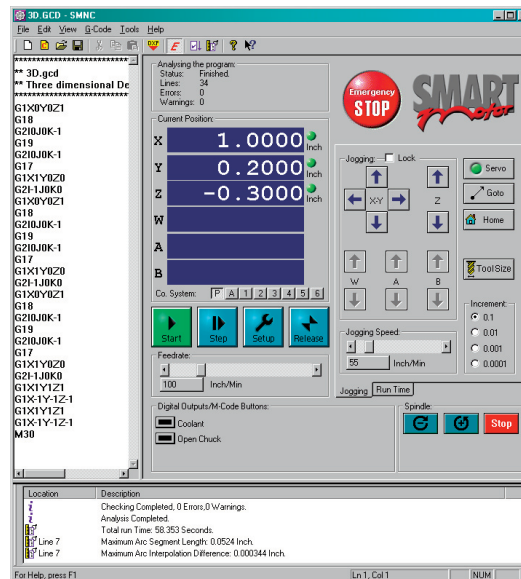
In torque mode, activated by the **MT** command, the drive duty cycle can be set with the **T=** command. The following number or variable must fall in the range between -1023 and 1023. The full scale value relates to full scale or maximum duty cycle. At a given speed there will be reasonable correlation between drive duty cycle and torque. With nothing loading the shaft, the **T=** command will dictate open-loop speed.

MD Contouring Mode (requires host)

SmartMotors with version 4.15 or greater firmware which includes all PLUS and ServoStep SmartMotors have the added ability to do multiple axis contouring. This firmware version became standard roughly mid-year 2001. The **Contouring Mode** is the foundation of the Animatics' G-Code interface that enables a P.C. and multiple SmartMotors to interpret G-Code files and do linear, circular and helical interpolation as well as unlimited multi-axis contouring.

The best way to take advantage of the SmartMotors contouring capability is to utilize the Animatics Provided Drivers that come with the SMI program. These drivers can free you from the additional work of implement the base level of the contouring functions in your own host level software. Still, if you wish to implement the function yourself, the following will detail how that is done.

The basic principle of operation takes advantage of the fact that each SmartMotor has a very accurate time base. Absolute position-time pairs of data get sent to the SmartMotor to fill buffers that facilitate continuous motion. The SmartMotor will adjust its own Velocity and Acceleration to be certain to arrive at the specified position at the exact specified time without slowing to a stop. As new position-time pairs arrive, the motor transitions smoothly from one profile to the next producing smooth, continuous motion. In a multiple axis configuration, different positions can be sent to different motors, with the same time intervals resulting in smooth, continuous multiple axis motion. The key is for the host to regulate the volume of data in each of the different motor's buffers. The position-time pairs of data are preceded with an identification byte and then four bytes for position and four for time. Time is in units of servo samples and is limited to 23 bits. Time is further constrained to be even powers of 2 (i.e. 1, 2, 4, 8, ..., 32768).



Contouring Mode is the foundation of Animatics' G-Code interface that enables a P.C. and multiple SmartMotors to interpret G-Code files and do multiple axis contouring.

CREATING MOTION

The coordinating host can send the **Q** command to solicit status information on the coordination process. Upon receiving the **Q** command, the SmartMotor will return status, clock and space available in the dedicated circular buffer. The response to **Q** takes two forms, one while the mode is running with trajectory in progress and no errors having occurred and another when the mode is not running. Both responses conform to the overall byte format of:

Q Response: 249 byte1 byte2 byte3 byte4

If the mode is running:

- byte1 bit 7 is set
- byte1 bits 6 through 0 return data slots available
- bytes 2, 3 & 4 return the 24 bit clock of the SmartMotor

If the mode is not running:

- byte1 bit 7 is clear
- byte1 bits 6 through 0 return status
- byte2 returns space available
- bytes 3 & 4 return the 12 lower bits of the 24 bit clock of the SmartMotor

As absolute position and time data is sent to the SmartMotor, differences are calculated what are referred to as "deltas". A delta is the difference between the latest value and the one just prior. Time deltas are limited to 16 bits while Position deltas are limited to 23 bits in size.

The Status Byte is constructed as follows:

bit0=1	MD mode pending a G
bit1=1	MD mode actually running
bit2=1	Invalid time delta > 16 bit received
bit3=1	Invalid position delta > 23 bits received
bit4=1	Internal program data space error
bit5=1	Host sent too much data (data buffer overflow)
bit6=1	Host sent too little data (data buffer underflow)

A trajectory terminates if an unacceptable position error occurs, if invalid data is received, if there is a data overflow or if there is a data underflow.

The host should send data pairs only when at least 3 empty data slots are available. **MD** responds to limit switches with an aborted trajectory. The **MD** mode uses KV feed forward for improved performance.

The byte flag that precedes and marks a position is of decimal value 250. The byte flag that precedes and marks a time is of decimal value 251.

The following is an example of the decimal byte values for a series of constant speed motion segments. Firmware versions 4.16 and higher do not need time values after the first two if the time delta is not changing. The byte transfers terminate with a carriage return (13).

Position	250 000 000 000 000 013	Position = 0
Time	251 000 000 000 000 013	Time = 0
Position	250 000 000 016 000 013	
Time	251 000 000 001 000 013	Time delta = 256
Position	250 000 000 032 000 013	
Position	250 000 000 048 000 013	
Position	250 000 000 052 000 013	Reduce position delta
Time	251 000 000 003 064 013	Reduce time delta
Position	250 000 000 056 000 013	
Position	250 000 000 060 000 013	
Position	250 000 000 064 000 013	

This example does not include addressing bytes. Refer to the section on Addressing SmartMotors to learn the most efficient way to address different motors.

What is not shown in the these codes are the addressing bytes that would be used to differentiate multiple motors on a network. As described ahead in this manual (see the **SADDR** command), a network of SmartMotors can be sorted out by sending a single address byte. When communicating to a particular motor, the address byte need only be sent once, until all of the communications to that particular motor are complete and another motor needs to be addressed. The byte patterns in the previous example would need to be preceded with an address byte (to a properly addressed motor) for multiple axis contouring. In the addressing scheme, there is a global address provision for sending data to all motors at once. By zeroing out the clocks before starting the contouring, the motors will by synchronized and single time values can then be sent to all motors at once, increasing overall bandwidth. Also, as mentioned earlier, SmartMotors with version 4.16 or higher do not need time data past the first two, if there is no change in the time delta.

Note that Time Data is the same for all motors and should be sent once to all motors at the same time, preceded by the global address byte (128).

The basis for contouring using this format is to keep the rate at which data is sent to each motor constant (and as fast as possible). That means that in order to accelerate axes, absolute positions need to be sent that invoke progressively larger position deltas, and to keep constant velocity, absolute positions need to be sent that are equidistant.

With all of the communications to send data and receive status, it would be outstanding to have a bandwidth on a two axis system of 64 samples, or 16ms. Typically, with a three or four axis system a bandwidth of 128 servo samples or 32ms is achievable. This would be at a baud rate of 38.4k. Keep in mind that during this time the SmartMotor is micro interpolating. The motion will be very smooth and continuous.

In contouring mode, all of the binary contouring data goes into the motor's

CREATING MOTION

buffers. While this is true, regular commands will still be recognized and they will operate normally. This will take some time, however, and it is up to the programmer to assure that the buffers never underflow due to neglect.

With Non-PLUS Firmware, contouring mode can be exited in only two ways. One way is to simply stop sending data, causing the buffers to underflow. This is effective, but leaves the motors off altogether. Alternatively, contouring mode can be terminated by sending a long time delta to the motor while holding position, estimating when the motor would be executing the long time delay and sending a **G** command, and likely a **P**=(last host mode position), to execute the next buffered mode.

PLUS and ServoStep Firmware afford more options related to termination of contour mode. Consecutive identical clock values may be sent to the SmartMotors to produce a zero time delta. When the host mode clock has reached the identical clock value, the host mode trajectory position is latched, the host mode trajectory velocity is latched and a "G-on-th-fly" is executed internally, using buffered mode, acceleration, velocity, position and relative distance. The position associated with the duplicated clock value is ignored. Any buffered mode (other than host mode) initiated by the G command will be initiated, including: **MV**, **MP** (w/P=#), **MP** (w/D=#), **MFR**, **MFN**, **MSR** and **MS**. As an example, the following code will cause the motor to servo in place at the final host mode trajectory position:

```
A=100      'Set Acceleration
V=10000    'Set Velocity
MD         'Init. buffered coordinate collection

<Load buffers with initial pos-time coordinates>

G          'Start the contouring motion

<Stream pos-time coordinates for contouring>

MP         'Buffer next mode after Host Mode

<Stream pos-time coordinates for contouring>

D=0        'Buffer next pos. move after Host Mode

<Stream pos-time coordinates for contouring>

MP         'Buffer next mode after Host Mode

<Send two identical time values to end Host Mode>
```

At this point the motors will come to a rest. If the motor is at speed when leaving Host Mode, then it will decelerate to rest and then return to the exact position where Host Mode was terminated. One could just as well exit Host Mode in Velocity Mode and maintain a constant speed.

MD50 Drive Mode (Removed in Plus & ServoStep Firmware)

The **MD50** command causes the SmartMotor to emulate a traditional servo and amplifier. In this mode, Port A is assigned to receive an analog command signal input where 0-5VDC commands -100% to +100% PWM in Torque Mode. 2.5VDC is the center point for zero torque. To balance the internal 5K Ohm pull-up resistor, it is suggested that a 5k external pull-down resistor be used. Understand that no I/O port can accommodate a negative voltage swing. Only 0-5VDC can be applied directly to the I/O pins. It is best to make use of an isolated analog signal conditioner to convert a +/-10VDC command signal to 0-5VDC. Otherwise, for a 0-10VDC (positive only) signal, a simple voltage divider could be used. The external command signal must have a complementary push-pull drive circuit capable of driving 10mAmps minimum for a good linear response from the motor.

```
UAI          'Assign Port A as an Input Port
MD50         'Set Drive Mode
```

BRAKE COMMANDS

BRKRLS Brake release

BRKENG Brake engage

BRKSRV Release brake when servo active, engage when not

BRKTRJ Release brake when running a trajectory, engage under all other conditions. Turns servo off when the brake is engaged

Many SmartMotors are 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 holding position for any reason. This might be due to loss of power or just a position error, limit fault, 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 SmartMotors 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. That will reduce the brake's life.

*When **BRKG** is used, do not issue the following commands:*

RS4

OCHN(RS4,...

UGI

UG=<value>

<variable>=UG

BRKC, BRKG, BRKI Re-route brake signal to I/O pin C or G (PLUS and ServoStep firmware only)

When the automated brake functions are desired for an external brake, commands **BRKC** and **BRKG** can be used. These commands re-route the internal brake signal to the respective I/O pins. The brake signal is active low. The **BRKI** command restores the brake function to the internal brake signal used with internally installed brakes. Only one pin can be used as the brake pin at any one time so each command supersedes the other.

MTB Mode Torque Brake (Available only in SmartMotors with PLUS firmware)

Mode Torque Brake is the default state of SmartMotors operating from PLUS firmware. It causes the amplifier to dynamically "brake" the motor when it is in it's off state. 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.

To deactivate **MTB**, issue **BRKRLS** immediately followed by **OFF**. The default **MTB** action is consistent with **BRKSRV** mode. When Status bit Bo (motor-off) is 0, **MTB** will be inactive, whereas when **Bo** is 1, **MTB** will be active. Basically, when the motor is off for any reason, **MTB** will be active, including upon power-up.

If **BRKTRJ** is selected, then **MTB** will be inactive when the Bt (busy-trajectory) bit is 1, and active when Bt is zero.

When **BRKRLS** is followed by **OFF**, **MTB** will become inactive until the next **MTB** command is issued. Issuing **BRKENG** will not activate **MTB**.

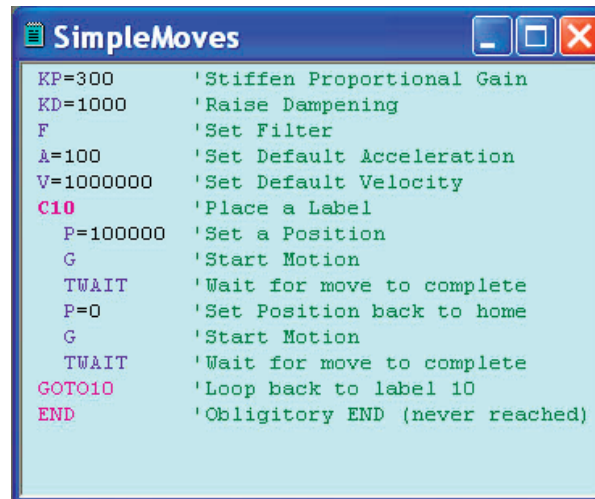
The **RMODE** command will report a "B" when **MTB** is active.

SLE, SLD Software Limits Enable and Disable (PLUS and ServoStep Firmware only)

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 in the event the motor strays beyond the desired region of operation.

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 its program is reset with the **Z** command, it will execute its program from top to bottom, with or without a host P.C. 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.



```

SimpleMoves
KP=300      'Stiffen Proportional Gain
KD=1000    'Raise Dampening
F          'Set Filter
A=100      'Set Default Acceleration
V=1000000  'Set Default Velocity
C10        'Place a Label
    P=100000 'Set a Position
    G        'Start Motion
    TWAIT    'Wait for move to complete
    P=0      'Set Position back to home
    G        'Start Motion
    TWAIT    'Wait for move to complete
    GOTO10   'Loop back to label 10
END         'Obligitory END (never reached)
  
```

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, 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 **RUN** is issued. 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 and jump over the **RUN?** command to the next line of code, continuing on.

```

PRINT("Boot-Up",#13)  'Message always prints
RUN?                  'Program stops here on power-up
PRINT(Run Issued",#13 'This runs if RUN received
END
  
```

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

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 setup determines what the programmed SmartMotor will do, and exactly how "smart" it will actually be.

PROGRAM FLOW

GOTO# Redirect program flow

C# Subroutine label, C0-C999

The most basic commands for redirecting program flow, without inherent conditions, are **GOTO#** in conjunction with **C#**. Labels are the letter **C** followed by a number (**#**) between 0 and 999 and are inserted in the program as place markers. 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**, the program flow will be redirected to label **C1** and the program will proceed from there.

```
C10           `Place label
  IF UAI==0    `Code
    GOSUB20    `Code
  ENDIF        `Code
  IF UBI==0    `Code
    GOSUB30    `Code
  ENDIF UAI==0 `Code
GOTO10        `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 and 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, which in turn, will make it easier to be read and changed.

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 and that's the only command that will be left on the SmartMotor until a new program is downloaded. To compile properly, every program needs and **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.

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

GOSUB# **Execute a subroutine**

RETURN **Return from subroutine**

Just like the **GOTO#** command, the **GOSUB#** command, 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 a **RETURN** command to return the program execution to the location of the **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 may be called any time from anywhere with a **GOSUB#**, 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 UAI==0        'Check Input A
    GOSUB20        'If Input A low, call Subroutine
  ENDIF            'End check Input A
GOTO10            'Will loop back to C10

C20                'Subroutine Label
  PRINT("Subroutine Activated",#13) 'Code
RETURN            'Return to line after GOSUB

```

F=32, F=64, RETURNF , RETURNI (PLUS and ServoStep only)

SmartMotors with **PLUS** firmware and **ServoStep** motors have automatic interrupt based **GOSUB** capabilities. After an **F=32** mode bit is set, the program will go immediately to the subroutine "**C1**" when a motor protection fault occurs. Subroutine **C1** should terminate with the **RETURNF** command rather than the standard **RETURN**. After the **RETURNF** is reached, program execution will take up precisely where it had been previously diverted. The following faults will trigger subroutine C1 in this mode:

- Be:** Position Error
- Bh:** Over Temperature or Over Current Error
- Bp:** Real time Positive Limit via software or hardware limits
- Bm:** Real time Negative Limit via software or hardware limits

It is important to note that without the **F=32** mode bit set (in **PLUS** and **ServoStep** only), and a **C1** routine present, any of the motor fault errors will result in program termination. The motor will stop under full MTB (Mode Torque Brake), where all coils of the motor are internally grounded. Once a motor protection fault as occurred, the **G** command will have no effect until the

With PLUS or ServoStep Firmware, any motor protection fault will result in program termination unless the F=32 bit is set and a C1 subroutine exists.

PROGRAM FLOW

fault has been cleared. The **ZS** command clears all faults at once.

Setting the **F=64** mode bit similarly causes program execution to jump to label **C2** when the I/O pin G goes from a high to a low. The **RETURNI** is used to terminate the **C2** subroutine.

The **WAIT** command is the only motor instruction that will be truncated automatically when interrupted. Wherever the **WAIT** command is in its timing cycle, it will be terminated upon return and so is at risk of being shortened in the event of an **F=64** interrupt. Since **F=64** uses the **G** port, it would be necessary that you turn off the **G** port's default action, which is to start motion. To deactivate the start function, issue a **UGI**. Note also that the **BRKG** command routes automated brake functions to the **G** port.

If both **F=32** and **F=64** need to be used at the same time, then be aware that these are bits in a broader configuration byte. It is best to use a shadow variable to store the byte and set them this way:

```
f=f|32      `Set F=32 bit in the shadow variable
              (C1 Routine)

f=f|64      `Set F=64 bit in the shadow variable
              (C2 Routine)

F=f         `Set the modes into action
```

When used together, the **C1** subroutine has higher priority than the **C2** subroutine. If the **C2** subroutine is executing and there is an error, the **C1** subroutine will execute and then return execution to the **C2** subroutine when finished. If this is not desired, the **C2** subroutine should clear the **F=32** mode bit in the beginning and reset the bit at the end of the **C2** code.

The **STACK** and **END** commands clear the tracking of subroutine nesting, even with interrupt subroutines.

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=UAI      `Variable 'a' set 0,1
a=a+UBI    `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 **A**. 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 farther reaching 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=UAI           'Variable 'a' set 0,1
a=a+UBI        '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 structures that can nest within one another.

The commands that can conditionally direct program flow to different areas use a constant **[#]** like 1 or 25, a variable like **a** or **a1[#]** or a function involving constants and/or variables **a+b** or **a/[#]**. Only one operator can be used in a function. The following is a list of the operators:

- +** Addition
- Subtraction
- *** Multiplication
- /** Division
- ==** Equals (use two =)
- !=** Not equal
- <** Less than
- >** Greater than
- <=** Less than or equal

PROGRAM FLOW

>= Greater than or equal

& Bit wise AND (see appendix A)

| Bit wise OR (see appendix A)

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
  UA=1       'Set output to 1
  UA=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 10x
```

The task or tasks within the **WHILE** loop will execute as long as the function 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, however, 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 previous program:


```

a=UAI          'Variable 'a' set 0,1
a=a+UBI        '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

```

The SWITCH statement makes use of the same memory space as variable "zzz". Do not use this variable or array space when using 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 during trajectory

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=exp Wait (exp) sample periods

There will probably be circumstances where the program execution needs to be paused for a specific period of time. Time, within the SmartMotor, is tracked in terms of servo sample periods. Unless otherwise programmed with the **PID#** command, the sample rate is about 4KHz. **WAIT=4000** would wait about one second. **WAIT=1000** would wait for about one quarter of a second. The following code would be the same as **WAIT=1000**, only it will allow code to execute during the wait if it is placed between the **WHILE** and the **LOOP**.

*For the exact sample period, use the **RSP** command*

PROGRAM FLOW

```
CLK=0           'Reset CLK to 0
WHILE CLK<1000  'CLK will grow
  IF UAI==0     'Monitor input A
    GOSUB911    'If input low
  ENDIF        'End the IF
LOOP           'Loop back
```

The above code example will check if port **A** ever goes low, while it is waiting for the **CLK** variable to count up to 1000.

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.

Here is an example program for PLUS or ServoStep firmware using the C1 Interrupt capability:

```
C1      'Interrupt routine C1 (enabled by F mode)
STACK   'Clear the nesting stack
RUN      'Begin the program, retaining variables
RETURNF  'Never reached, but necessary for comp.
```

Typical standard firmware use:

```
P=1234      'Set a position
GOSUB5      'Call subroutine 5

C5          'Subroutine 5
G          'Start Motion
TWAIT      'Wait for motion to stop
IF Bo      'Check to see if there was an error
  STACK    'Clear the nesting stack
  RUN      'Begin the program, retaining variables
ENDIF      'Never reached, but necessary for comp.
RETURN     'Never reached, but necessary for comp.
```

Variables are data holders that can be set and changed within the program or over one of the communication channels. All variables are 32-bit signed integers and are all lower case only. They are stored in volatile memory, meaning they are lost when power is removed and default to zero upon power-up. If they need to be saved, they can be stored in EEPROM, non volatile memory using the **VST** command.

There are three sets of variables containing 26 in each. The last 52 can also be accessed as byte, short or long array elements.

The first 26 variables are accessed with the lower case letters of the alphabet, **a, b, c, . . . x, y, z.**

a=# Set variable **a** to a numerical value

a=exp Set variable **a** to value of an expression

A variable can be set to an expression with only one operator and two operands. The operators can be any of the following:

- +** Addition
- Subtraction
- *** Multiplication
- /** Division
- &** Bit wise AND (see appendix A)
- |** Bit wise OR (see appendix A)

The following are legal:

a=b+c,	a=b+3	a=5+8
a=b-c	a=5-c	a=b-10
a=b*c	a=3*5	a=c*3
a=b/c	a=b/2	a=5/b
a=b&c	a=b&8	
a=b c	a=b 15	

ARRAYS

In addition to the first 26, there are 52 more long integer variables accessible with double and triple lower case letters: **aa, bb, cc, . . . xxx, yyy, zzz.** The memory space that holds these 52 variables is more flexible, however. This same variable space can be accessed with an array variable type. An array variable is one that has a numeric index component that allows the numeric selection of which variable a program is to access. This memory space is further made flexible by the fact that it can hold 51 thirty two bit integers,

VARIABLES

See **Appendix D**
for a table
describing
overlapping
memory allocation
for
**User Assigned
Array Variables.**

or 101 sixteen bit integers, or 201 eight bit integers (all signed). The array variables take the following form:

ab[i]=exp Set variable to a signed 8 bit value where index i = 0...200

aw[i]=exp Set variable to a signed 16 bit value where index i = 0...100

al[i]=exp Set variable to a signed 32 bit value where index i = 0...50

The index i may be a number, a variable **a** thorough **z**, or the sum or difference of any two variables **a** thorough **z** (variables only).

The same array space can be accessed with any combination of variable types. 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 space as the first single 32 bit number, and due to the way binary numbers work, would make the thirty two bit variable equal to 256. The order is most significant to least with ab[0] being the most.

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 would 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.

STORAGE OF VARIABLES

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, 0-7999

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.

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.

FIXED OR PRE-ASSIGNED READABLE VARIABLES

In addition to the general purpose variables there are variables that are gateways into the different functions of the SmartMotor itself.

@P	Current position
@PE	Current position error
@V	Current velocity
ADDR	Motor's self address
CHN0	RS-232 com error flags
CHN1	RS-485 com error flags
CLK	Read/Write sample rate counter (clock)
CTR	External encoder count variable
I	Last recorded index position
LEN	# of characters in RS-232 input buffer
LEN1	# of characters in RS-485 input buffer
TEMP	SmartMotor Temperature in Degrees Centigrade

VARIABLES

Reading SmartMotor Current can give an indication of the resistance the shaft is feeling, as can Position Error (@PE).

U	7-bit value of user input/output pins A-G (PLUS and ServoStep firmware only)
UAA - UGA	I/O Digital Input
UAI - UGI	I/O Analog Input
UIA	SmartMotor Current in Tens of Milliamps
UJA	SmartMotor Voltage in Tenths of Volts

VARIABLE SPACE RESTRICTIONS

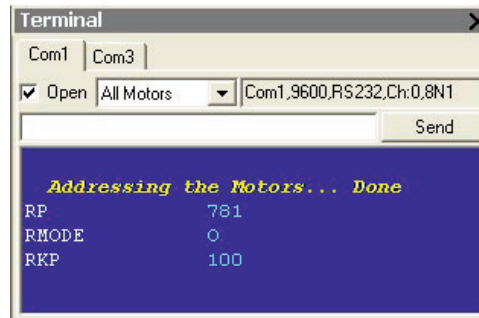
Due to limited Microprocessor resources within the SmartMotor, some functions use variable space otherwise accessible to the user. These are as follows:

- 1) SmartMotors with firmware 4.15 or lower, and 4.40 use variables xxx, yyy and zzz for the implementation of the **SWITCH** function.
- 2) DeviceNet and Profibus SmartMotors use variable zzz.
- 3) Contouring Mode uses variables aa - yyy, leaving zzz for **SWITCH**.

The SMI program uses variables to undertake certain functions as well. This can be useful to know as it may impact your program development.

- 1) Variable z is used in MotorView and Playground to read Digital Inputs for motors with firmware pre-dating PLUS and ServoStep.
- 2) Variables a, b, p, t, w and z are used with the Tuning Program (although they are saved and restored after the tuning)
- 3) Variables aaa-jjj, ab[0] and aw[0] are used when reading or writing information to the EEPROM by the SMI program (although these are saved and restored as well).
- 4) Variable yyy is used when calibrating a ServoStep.

The SmartMotor has a wealth of data that can be retrieved over the RS-232 or RS-485 ports simply by asking. Data and status reporting commands can be tested by issuing them in the SMI Terminal window. In the example to the right, 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 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 following are commands, that when sent to the SmartMotor, will return valuable data:



REPORT TO HOST COMMANDS

Ra...Rzzz	Report variables a ... zzz , 78 in all
Rab[i]*	Report 8 bit variable value Rab[i]
Raw[i]*	Report 16 bit variable value Raw[i]
Ral[i]*	Report 32 bit variable value Ral[i]
RA	Report buffered acceleration
RAIN{port}{ch}	Report 8 bit analog input port=A-H, ch= 1-4
RAMPS	Report assigned maximum current
RBa	Report over current status bit
RBb	Report parity error status bit
RBc	Report communications error bit
RBd	Report user math overflow status bit
RBe	Report position error status bit
RBf	Report communications framing error status bit
RBk	Report EEPROM read/write status bit
RBI	Report historical left limit status bit
RBi	Report index status bit
RBh	Report overheat status bit
RBm	Report negative limit status bit
RBo	Report motor off status bit
RBp	Report positive limit status bit
RB r	Report historical right limit status bit

See Appendix D for a table describing overlapping memory allocation for **User Assigned Array Data Variables.*

REPORTING COMMANDS

RBs	Report program scan status bit
RBt	Report trajectory status bit
RBu	Report user array index status bit
RBw	Report wrap around status bit
RBx	Report hardware index input level
RCHN	Report combined communications status bits
RCHN0	Report RS-232 communications status bits
RCHN1	Report RS-485 communications status bits
RCLK	Report clock value
RCTR	Report secondary counter
RCS	Report RS-232 communications check sum
RCS1	Report RS-485 communications check sum

*The **RCS** commands sum the ASCII values of all incoming serial bytes in an 8-bit, recycling register with a result that always falls on or between 0 and 255, and which resets to zero after being read. Using the SMI terminal, the following can be observed. Note that the SMI terminal uses a 'space' or ASCII 32 as a delimiter, and issue an RCS before testing this example to make sure you start with a value of zero:*

A=100 65 61 49 48 48 32 = 303

V=320000 85 61 51 50 48 48 48 48 32 = 472

G 71 32 = 103

RCS 82 67 83 32 = 264

The sum of all of these numbers is 1142. The formula to determine the recycling 8-bit value is:

$$1142 - \text{int}(1142/256) * 256 = 118$$

RD	Report buffered move distance value
RDIN{port}{ch}	Report 8 bit digital input byte, port=A-H, and ch=0-63
RE	Report buffered maximum position error
RI	Report last stored index position
RKA	Report buffered acceleration feed forward coefficient
RKD	Report buffered derivative coefficient
RKG	Report buffered gravity coefficient
RKI	Report buffered integral coefficient
RKL	Report buffered integral limit value
RKP	Report buffered proportional coefficient

REPORTING COMMANDS

RKS	Report buffered sampling interval
RKV	Report buffered velocity feed forward coefficient
RMODE	Report present positioning mode: P Absolute position move R Relative position move V Velocity move T Torque mode F Follow mode S Step and Direction mode C Cam Table mode W Drive mode X Follow mode with multiplier E Position error O Motor off H Contouring mode
RP	Report measured position
RPE	Report present position error
RS	Report status byte (8 system states) <i>The RS status byte consists of the lower 8 bits of RW, except that with RS, the limit bits are Real Time. RW is detailed in the following table.</i>
RSP	Report sample period and version number
RT	Report current requested torque
RU	Report all 7 I/O in one byte (PLUS & ServoStep)
RU{pin}	Report digital I/O states (PLUS & ServoStep)
RU{pin}A	Report analog I/O states (PLUS & ServoStep)
RV	Report velocity
RW	Report status word (16 system states) detailed in following table:

REPORTING COMMANDS

16 status bits can be read with a single RW request. Two bytes get reported with the status bits coded in them per this table.

RW Components:

Bit:	Value:	Meaning:	Clear:	Related Commands:
0	1	Trajectory In Progress		G, TWait
1	2	Historical Right (+) Limit	Zr, ZS	UCI, UCO, UC=, UCP
2	4	Historical Left (-) Limit	Zl, ZS	UDI, UDO, UD=, UDM
3	8	Index Report Available	Ri, =I	
4	16	Wraparound Occurred	Zw, ZS	O=, MF0, MS0
5	32	Excessive Position Error	Ze, ZS	E=
6	64	Excessive Temperature	ZS	THD, TH=
7	128	Motor is Off		OFF
8	256	Index Input Asserted		
9	512	Right (+) Limit Asserted		UCI, UCO, UC=, UCP
10	1024	Left (-) Limit Asserted		UDI, UDO, UD=, UDM
11	2048	User Math Overflow	Zd, ZS	
12	4096	User Array Index Error	Zu, ZS	ab[], aw[], al[]
13	8192	Syntax Error	Zs, ZS	
14	16384	Overcurrent Occurred	Za, ZS	
15	23768	Program Checksum Error	ZS	

Three very valuable pieces of data do not have direct report commands, these are Temperature, Voltage and Current.

To read Temperature, issue the following commands, for example:

```
z=TEMP      `Put TEMP into variable z
Rz          `report variable z
```

The number returned by the above example is in units of Degrees Centigrade.

To read Voltage, issue the following commands, for example:

```
z=UJA       `Put UJA into variable z
Rz          `report variable z
```

The number returned by the above example is in units of tenths of Volts. So, for example, if you read 259, that will mean 25.9 Volts.

To read Temperature, issue the following commands, for example:

```
z=UIA       `Put UIA into variable z
Rz          `report variable z
```

The number returned by the above example is in units of tens of milliamps. So, for example, if you read 145, that will mean 1.45 Amps.

ENCODER AND PULSE TRAIN FOLLOWING

Through the two pins, A and B of the I/O connector, quadrature or step and direction signals can be fed into the SmartMotor at high speeds and be followed by the motor itself. This feature brings about the following capabilities:

- 1 Mode Follow
- 2 Mode Step and Direction
- 3 Mode Follow with ratio
- 4 Mode Step and Direction with ratio
- 5 Mode Cam

In addition to the above embedded modes of operation, the internal counter can be set to either count encoder signals or step signals and be accessible to the internal program or a host through the **CTR** variable.

When the SmartMotor is in one of the above five modes it may also run internal programs and communicate with a host, all at the same time.

MF1, MF2 and MF4 Mode Follow

Mode Follow allows the SmartMotor™ to follow an external encoder. Three resolutions can be selected through hardware, and a virtually infinite number of resolutions can be set in firmware using the MFR command described ahead. Set the hardware for maximum resolution with the **MF4** command. The **MF2** The **MF1** commands set the hardware to lesser resolutions, but are obsolete with the advent of the newer MFR capability.

MF0, MS0

The **MF0** and **MS0** commands must not be issued during one of the other follow modes. They are used for an entirely different purpose. If it is not desired to directly follow an incoming encoder or step signal, but rather, just to track them and use the counter value within a program or from a host, then issuing **MF0** or **MS0** utilizes the maximum resolution available and makes the value available through the **CTR** variable. Issuing **MF0** or **MS0** will zero that variable and incoming encoder or step signals will increment or decrement the signed, 32-bit **CTR** variable value.

MFDIV=expression **Set Ratio divisor**

MFMUL=expression **Set Ratio multiplier**

where $-256.0000 < \text{Ratio} < 256.0000$

After the appropriate **MF#** command is issued, or the **MS** command has been issued, a floating point ratio can further be applied by the firmware. Since the SmartMotor is an integer machine, that floating point ratio is accomplished by dividing one number by another.

*The SM2315D
does not have
Quadrature
Encoder following
capability.*

ENCODER AND PULSE TRAIN FOLLOWING

MFR Calculate Mode Follow Ratio

MSR Calculate Mode Step Ratio

Once a numerator and denominator have been specified, and the appropriate hardware mode is selected, the motor can be put into ratio mode with the **MFR** or **MSR** commands (**MSR** for ratioing incoming step and direction signals). The following example sets up a 10.5:1 relationship:

```
MF4                      `Read in full quadrature decode
MF4MUL=2                `10.5:1=21:2
MF4DIV=21
D=0                      `be sure D is zero
MFR                      `Invoke calculation
G                        `Start
```

Once in a ratio mode the **V=#** and **D=#** commands will still work. They will invoke a phase shift of length **D** at a relative rate determined by **V**. For that reason, **D** must be zeroed out before issuing an **MFR** or **MSR** command or unexpected shifting could be taking place. In applications such as a Web Press, this ability to phase shift can be very useful.

MC Mode Cam

A cam is a basically round but irregular shape that rotates and causes a follower to move up and down in a profile determined by the shape of the cam's exterior.

Since the beginning of industrialization, cams have been used to create complex, reciprocating motion. Cams are most often carved out of steel and changing them, or having them invoke motion a great distance away are impractical. The SmartMotor provides an electronic alternative. Putting an encoder on the rotating part of a machine, sending the signals to a SmartMotor and programming the cam profile into the SmartMotor allows for the same complex, repeating motions to be accomplished without any of the typical mechanical limitations.

BASE=expression Base length

Part of defining a Cam relationship is specifying how many incoming encoder counts there are for one full cam rotation. Simply set **BASE** equal to this number.

SIZE=expression Number of Cam data entries

The upper variable array space holds the cam profile data. To instruct the SmartMotor as to how many data points have been specified, set **SIZE** equal to that number. The cam firmware looks at words (16 bit numbers). The maximum number of words that can be used is 100. The cam firmware will perform linear interpolation between those entries, as well as between the

ENCODER AND PULSE TRAIN FOLLOWING

last and the first as the cam progresses through the end of the table and back to the beginning. The cam table entries occupy the same space as variables **aa** through **yyy** which is the same space as the array variables. Invoking Cam Mode is done as follows:

```
BASE=2000      'Cam period
SIZE=25        'Data segments, this defines the data
               table size.
               'CTR data, note the period at the end
aw[0] 0 10 20 30 40 50 60 70 80 90 100 110 120 120
        110 100 90 80 70 60 50 40 30 20 10 0.
MF0           'Reset external encoder to zero
O=0           'Reset internal encoder position
MC            'Buffer CAM Mode
G             'Start following the external encoder
              using cam data
```

CI **Re-Initialize Cam Parameters at next zero crossover (For PLUS and ServoStep firmware only)**

The "Cam Initialize" command causes the buffered cam parameters to replace the current cam parameters, including the Dwell value, upon the next Zero Crossover. In cam mode the Dwell value is set with the **D=** command, normally used to set a relative move distance.

CX **Value of current Cam Index (For PLUS and ServoStep firmware only)**

The **CX** variable contains the real-time cam index while in cam mode. This enables the user to avoid the active area of the cam table when dynamically modifying the cam table data.

F=16 **Cause Cam to operate in Relative Position Mode (For PLUS and ServoStep firmware only)**

The **F** variable is used to store various different operational mode bits. The value 16 bit position is used to make cam mode operate in Relative Position Mode, rather than the default Absolute Mode. In Relative Position Mode, the cam table can end at a different position than it started for a progressive advance in position with every cycle.

Because of the binary nature of the **F** variable, care must be taken when setting a bit so that other bits in the **F** variable are not changed. The best way to keep track of **F** is to keep a shadow variable, say "f" for example. To set the 16 value bit, issue the following commands.

*Do not use variable
aa through **zzz**
while camming.*

*Don't use the
SWITCH state-
ment with a CAM
table of **SIZE=100**
because the top
two memory loca-
tions are shared.*

*Note that when
using MC with
Step pulses, the
direction can be
reversed from
inside the
SmartMotor by
setting port B to
an output and tog-
gling its value.*

ENCODER AND PULSE TRAIN FOLLOWING

```
f=f|16      'Modify the Shadow Variable  
F=f        'Set F with the Shadow Variable
```

To clear the 16 value bit, subtract 16 from 255 (255 - 16 = 239), then logically AND the result with the shadow variable:

```
f=f&239    'Modify the Shadow Variable  
F=f        'Set F with the Shadow Variable
```

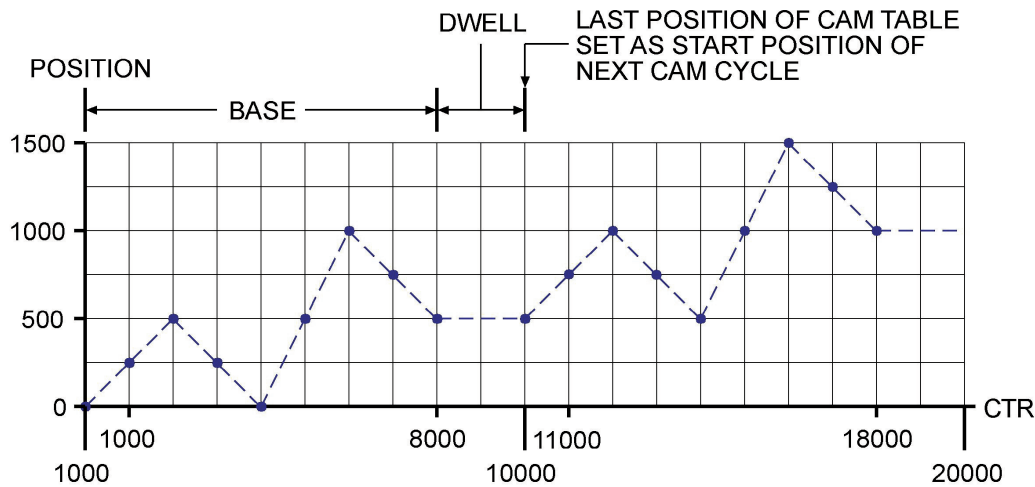
For more information about how binary numbers work, please refer to the appendix.

The following is an example of using Cam Mode in Relative Position Mode.

```
MF4  'Set external enc. in and zero ext. enc. count  
F=16 'Set to relative position mode in cam mode  
D=2000 'Set dwell in units of encoder counts  
BASE=8000 'Set number of ext. enc. = 1 cycle  
SIZE=8 'Set size - max array index  
aw[0]=0 'Define cam table  
aw[1]=250  
aw[2]=500  
aw[3]=250  
aw[4]=0  
aw[5]=5000  
aw[6]=1000  
aw[7]=750  
aw[8]=500  
O=0 'Set current shaft position to zero  
MC 'Set to mode cam  
G 'Start mode cam
```

The following diagram shows how in Relative Position Mode, cam position adopts the cycle ending point as the next cycle's starting point.

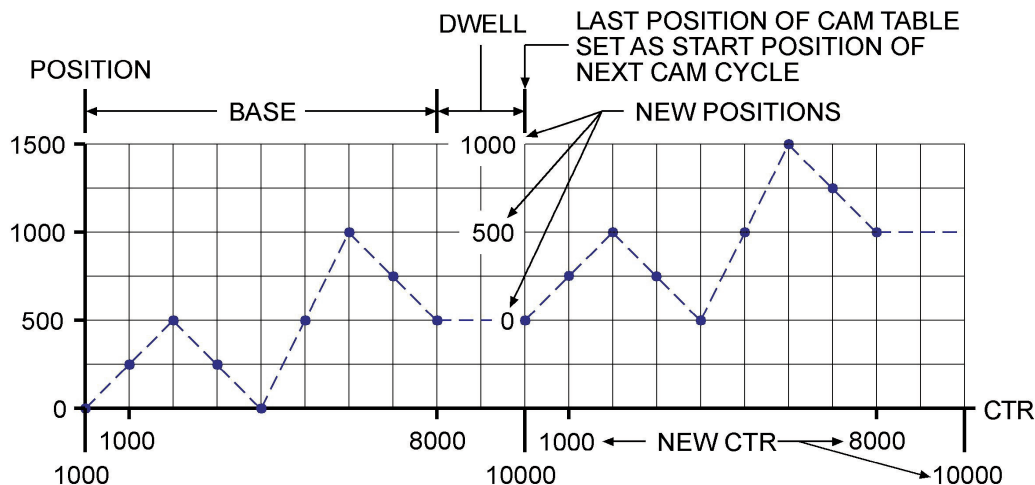
ENCODER AND PULSE TRAIN FOLLOWING



F=128 Set Cam to Mode Modulo (For PLUS and ServoStep firmware only)

Even with 32 bit position, high speed operation can cause a roll-over in the position register after several days. To avoid this problem, **F=128** will cause both the internal position register as well as the external encoder counter to reset to zero at the end of each combined cam and dwell cycle. This bit value 128 can be set and cleared the same way the previous example shows bit value 16 being set and cleared.

The following example shows the results of substituting **F=128** where **F=16** currently exists in the previous example:



PLUS and ServoStep Firmware allows the dynamic reprogramming of cam mode while it is functioning. Dynamically changeable cam parameters include Dwell (**D**), **BASE**, **SIZE**, multiplier **MC2**, **MC4** and **MC8**. Send the new values of **BASE** and/or **SIZE** to the motor followed by the MCn (**MC**, **MC2**, **MC4** or **MC8**) and **CI** command to begin using the new values at the next zero-point crossing.

ENCODER AND PULSE TRAIN FOLLOWING

F= must have values 16 or 128 true (on) for dwell to be operative.

The issuance of any of the commands **D=** dwell 0, **BASE**, **SIZE**, **MCn**, or changing **F=16** or 128, after the **CI** command, but before the next zero point crossing occurs, will invalidate the **CI** command, and no partial initialization will take place. A new **MCn** and **CI** command must be issued.

If **BASE** and/or **SIZE** is being changed on-the-fly, it is necessary to issue an **MCn** command following the new **BASE** and/or **SIZE** assignment (preceding the **CI** command, of course). This is the same as when mode cam is being started by a **G**. The **MCn** command causes the buffered cam interval length **BASE/SIZE** to be calculated.

Changed values **F=16** and **F=128** will not take effect in cam mode, even though the change has been made in **F=**, until the **CI** command is issued and there is a zero-point crossing. Changing **F=** values on-the-fly requires careful consideration of the effect.

The following example shows the dynamic changing of the cam function to vary the length of material cut-off:

```
F=144      'F=16+128, rel. (F=16), antiwrap (F=128)
BASE=2000  'Quick cutoff of rough end as belt begins
D=100      'Soon afterward, begin a cammed cutoff
MC2        'Double-wide first cutoff
G
BASE=10000 'Normal cutoff profile
D=107525   'Now allow normal length between cuts
MC2        'Recalculate base/size, set to mode cam.
CI         'Init. cam params at zero-point crossing
UAI        'Input pin A button makes longer
UBI        'Input pin B button makes shorter
WHILE 1
  IF UAI
    D=D+1    'Make longer
    MC2      'Recalculate base/size
    CI       'Update at next zero point crossing
  ENDIF
  IF UBI
    D=D-1    'Make shorter
    MC2      'Recalculate base/size
    CI       'Update at next zero point crossing
  ENDIF
  WAIT=100  'Max length change of 40 per second
LOOP
```


ENCODER AND PULSE TRAIN FOLLOWING

ENC0, ENC1 Encoder Select

The **ENC1** command causes the SmartMotor to servo off of an external encoder connected to inputs A and B. This can be useful if the external encoder has the potential of being more accurate. The **ENC0** command restores the default mode of servoing off of the internal encoder.

ENC1	`Servo off of external encoder
ENC0	`Servo from internal encoder (default)

While an external encoder can have its advantages, by going to a wired feedback solution the system sacrifices reliability by introducing a new failure mode.

This page has been intentionally left blank.

The following binary values can be tested by **IF** and **WHILE** control flow expressions, or assigned to any variable. They may all be reported using **RB{bit}** commands. Some may be reset using **Z{bit}** commands and some are reset when accessed. The first 8 states are reported in combination by the **RS** command. **RW** reports sixteen of these flags in combination. Be aware that the lower 8 bits of **RW** do not match **RS**; in **RS**, the limits are reported Real Time.

By writing programs to periodically test these bits, a SmartMotor application can be very "smart" about its own inner-workings and doings.

Bo	Motor off
Bh	Excessive temperature
Be	Excessive position error
Bw	Wraparound occurred
Bi	Index report available
Bm	Real Time negative limit, aka "Left" limit, Port D
Bp	Real Time positive limit, aka "Right" limit, Port C
Bt	Trajectory in progress
Ba	Over current state occurred
Bb	Parity error occurred
Bc	Communication overflow occurred
Bd	User math overflow occurred
Bf	Communications framing error occurred
Bk	Program check sum/EEPROM failure
Bl	Historical negative limit, aka "Left" limit, Port D
Br	Historical positive limit, aka "Right" limit, Port C
Bs	Syntax error occurred
Bu	User array index error occurred
Bx	Hardware index input level

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. The flags that need to be reset are listed. Their letter designator is preceded by the letter **Z** in the following list:

SYSTEM STATE FLAGS

G also resets several system state flags.

RESET SYSTEM STATE FLAGS

Za	Reset over current violation occurred
Zb	Reset parity error occurred
Zc	Reset com overflow error occurred
Zd	Reset user math overflow occurred
Zf	Reset communications framing error occurred
Zl	Reset historical left limit occurred
Zr	Reset historical right limit occurred
Zs	Reset syntax error occurred
Zu	Reset user array index error occurred
Zw	Reset wraparound occurred
ZS	Reset all Z{bit} state flags

An example of where one would use a System State Flag would be to replace the **TWAIT** command. The **TWAIT** command pauses program execution until motion is complete. Instead of using **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:

```
WHILE Bt    'While trajectory
  IF Bi      'Check index
    GOSUB500 'call subroutine
  ENDIF      'end checking
LOOP        'Loop back
```

INPUTS AND OUTPUTS

The standard SmartMotor brings out 5 volt power and ground, as well as seven I/O points. Each one has multiple functions. They are **UA**, **UB**, **UC**, **UD**, **UE**, **UF** and **UG** and have the following functions:

UA	Digital Input, TTL, 0 to 5 volts Digital Output, TTL, 0 to 5 volts Analog Input, 10 bit, 0 to 1023 External Encoder A Input* Step and Direction, Step Input*
UB	Digital Input, TTL, 0 to 5 volts Digital Output, TTL, 0 to 5 volts Analog Input, 10 bit, 0 to 1023 External Encoder B Input* Step and Direction, Direction Input*
UC	Digital Input, TTL, 0 to 5 volts Digital Output, TTL, 0 to 5 volts Analog Input, 10 bit, 0 to 1023 Positive Limit Input Alternate Brake Output (Plus Firmware)
UD	Digital Input, TTL, 0 to 5 volts Digital Output, TTL, 0 to 5 volts Analog Input, 10 bit, 0 to 1023 Negative Limit Input
UE	Digital Input, TTL, 0 to 5 volts Digital Output, TTL, 0 to 5 volts Analog Input, 10 bit, 0 to 1023 AniLink Data I/O** AniLink RS-485 Signal A ***
UF	Digital Input, TTL, 0 to 5 volts Digital Output, TTL, 0 to 5 volts Analog Input, 10 bit, 0 to 1023 AniLink Clock Output** AniLink RS-485 Signal B ***
UG	Digital Input, TTL, 0 to 5 volts Digital Output, TTL, 0 to 5 volts Analog Input, 10 bit, 0 to 1023 Start Motion (or GO) Input RS-485 Adapter Direction Control Alternate Brake Output (Plus Firmware)

**ServoStep breaks these signals out to different pins. External Encoder input is not available as standard on SMXXX5 SmartMotors*

***AniLink is not available on ServoStep*

****Secondary RS-485 is not available as standard on SMXXX5 SmartMotors or ServoStep*

INPUTS AND OUTPUTS

The following is a list of all of the commands used to relate to the SmartMotor's many I/O ports, grouped by port.

THE MAIN RS-232 PORT

ECHO	ECHO back all received characters
SADDR#	Set ADDR ess (0 to 120)
SILENT	Suppress print messages
TALK	Re-activate print message
SLEEP	Ignore all commands except WAKE
WAKE	Consider all following commands
BAUD19200	Set baud rate to 19200 bps
OCHN (RS2,0,N,9600,1,8,D)	OpenChnl - RS-232, Channel 0, No parity, 9,600 bps, 1 stop, 8 data, as Data
OCHN (RS4,0,N,38400,1,8,C)	OpenChnl - RS-485 (w/adapter), Channel 0, No parity, 38.4k bps, 1 stop, 8 data, as Control. This uses port G for direction control
IF LEN>0	Check to see if any (or how much) data is in the 16 byte input buffer, Data mode
c=GETCHR	Get byte from buffer into variable c for Data mode
PRINT ("Char Rcd:",c,#13)	Print text, data and ASCII code for carriage return

COUNTER FUNCTIONS OF PORTS A AND B

MF4	Set Mode Follow with full quadrature
MFR	Set Mode Follow with ratio for gearing
MS	Mode Step and Direction
MC	Mode Cam
MF0	Set follow mode to zero and increment counter only
MS0	Set step mode to zero and increment counter only
a=CTR	Set variable a to counter value
ENC0	Restores internal encoder as Servo Encoder
ENC1	Redirects Servo operation to External Encoder

Ports A through G
have internal 5k
Ohm pullups to 5V.

GENERAL I/O FUNCTIONS OF PORTS A AND B

UAI	Set port A to input (UBI for port B)
UAO	Set port A to output (UBO for port B)
UA=0	Set port A Low (UB=0 for port B, or UB=a to set to variable a)
UA=1	Set port A High (UB=1 for port B)
a=UAI	Set variable a to digital input (UBI for port B)
a=UAA or UBA	Set a to analog input, 0 to 1023 = 0 to 5V

THE LIMIT PORTS C AND D

UCI	Redefine Positive, or Right Limit as general input (UDI for Negative, or Left Limit)
UCO	Redefine Positive, or Right Limit as general output (UDO for Negative or Left Limit)
UCP	Return pin to Positive, or Right limit function (UDM for Negative, or Left Limit function)
LIMD	Enable Directional Limits
LIMH	Limits active High (Not in PLUS or ServoStep)
LIML	Limits active Low (Not in PLUS or ServoStep)
LIMN	Restore non-directional limits (Not in PLUS or ServoStep)
UC=0	Set Right Limit Low (UD=0 for Left, or UD=a to set to variable a)
UC=1	Set Right Limit High (UD=1 for Left Limit)
a=UCI or UDI	Set variable a to digital input
a=UCA or UDA	Set a to analog input, 0 to 1023 = 0 to 5V
BRKC	Alternate Brake Output (Plus Firmware)
BRKI	This command will restore the brake function to the internal signal and free the C pin for other uses
ZI, Zr	Reset Left & Right Limit Faults (Plus and ServoStep Firmwares require limit faults to be reset before motion is allowed)

*PLUS and ServoStep Firmware is designed to use normally closed limits connected to ground. With nothing connected, the motor will fault, even on power-up. To clear the fault, reset the fault condition with **ZS** (or **ZI** and **Zr**).*

*To disable the limits altogether, redefine the limits as general inputs with the **UCI** and **UDI** commands, then clear the fault.*

INPUTS AND OUTPUTS

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

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

Secondary RS-485 functionality is not available with the SM2315D or the ServoStep.

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.

COMMUNICATION FUNCTIONS OF PORTS E AND F

PORTS E AND F AS ANILINK (USING I²C PROTOCOL)

AOUTB,c	Send variable c out to Analog I/O board addressed as B
DOUTB0,c	Send variable c out to Digital I/O board addressed as B0
c=AINB2	Set variable c to input 2 from Analog I/O board addressed as B
c=DINB0	Set variable c to input from Digital I/O board addressed as B0
PRINTB("Temp:",c,#32)	Print to LCD on network - text, data and ASCII code

PORTS E AND F AS RS-485

OCHN(RS4,1,N,38400,1,8,D)	OpenChnl - RS-485, Channel 1, No parity, 38.4k bps, 1 stop, 8 data, as Data
IF LEN1>0	Check to see if data is in the 16 byte input buffer
c=GETCHR1	Get byte from buffer into variable c
PRINT1("Char Rcd:",c,#13)	Print text, data and ASCII code
ECHO1	ECHO back all received characters
SILENT1	Suppress print messages
SLEEP1	Ignore all commands except WAKE
WAKE1	Consider all following commands

PORTS E AND F AS GENERAL I/O

UEI	Set port E to input (UFI for port F)
UEO	Set port E to output (UFO for port F)
UE=0	Set port E Low (UF=0 or port F, or UF=c to set to variable c)
UE=1	Set port E High (UF=1 or port F)
c=UEI or UFI	Set variable c to digital input
c=UEA or UFA	Set c to analog input, 0 to 1023 = 0 to 5V

THE G PORT

UGI	Redefine as general input
UGO	Redefine as general output (Open collector, pulled to 5V)
UG	Return pin to default start function, when low motor starts motion
UG=0	Set G port Low (UG=a to set to variable a)
UG=1	Set G port High (Open collector, weakly pulled to 5V internally)
a=UGI	Set variable a to digital input
a=UGA	Set a to analog input, 0 to 1023 = 0 to 5V
BRKG	Alternate Brake Output (Plus Firmware)
BRKI	This command will restore the brake function to the internal signal and free the G pin for other uses
OCHN	If the OCHN command is used to support an external adapter to convert the main port to RS-485, then the G pin becomes dedicated to that function, to govern data direction control
F=64	Setting this bit in the F register will cause a high to low transition of the G pin to call subroutine C2, if it exists
RETURNI	This statement is to be placed at the end of a G-called C2 subroutine to return program execution to the main program where it was interrupted

The G port is not available on the RTC-4000.

*When **BRKG** is used, do not issue the following commands:*

```
RS4
OCHN(RS4,...
UGI
UG=<value>
<variable>=UG
```

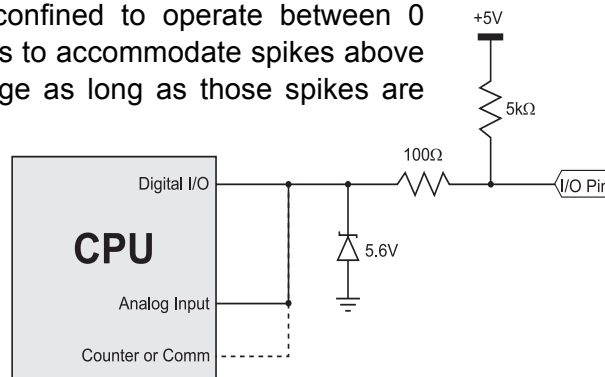
This page has been intentionally left blank.

The **UAA**, **UBA**, **UCA**, **UDA**, **UEA**, **UFA** and **UGA** variables reflect the analog voltages at the port pins regardless of how the pins are configured. The analog voltage of any pin can be read without effecting it's current mode of operation in any way. For example, a pin could be used as an output and then the analog input value could be read to see if it happened to be shorted, or RS-485* signal bias could be monitored at ports E and F.

The encoder and step counting capabilities of ports A and B are described in the section on External Encoder Modes. The serial data capabilities of ports E and F are described in the section on communications.

While all SmartMotor I/O is confined to operate between 0 and 5VDC, some circuitry exists to accommodate spikes above and below the operational range as long as those spikes are moderate and short lived.

Notice by the schematic that an I/O point can be configured as an output but still be readable as an analog input because the connections to the CPU are separate.



Knowing the SmartMotor's internal schematic can be useful when designing external interfaces.

All SmartMotor I/O points default to inputs when power is applied to the SmartMotor, until such time as the User Program makes a change. Because of the pull-up resistor, the voltage read at each port will be about 5VDC. When used as outputs to turn on external devices, it is highly recommended to design the system such that +5V is OFF and 0V is ON. This will prevent external equipment from being turned on immediately after power-up, before the User Program has a chance to take over.

SmartMotor I/O is logic 0 for voltages below 1.2V and a logic 1 for voltages above 3.0V. Logic states for voltages between these are unpredictable.

EXTERNAL RS-485 I/O

The DIN-RS-485 product adds 24 Volt I/O to applications requiring more I/O than what the SmartMotor has built-in. It can be connected to the SmartMotor's secondary RS-485 port where it exists, or to the ServoStep's primary RS-485 port. Communications are simple and the compact unit can mount on a DIN Rail inside of a standard controls cabinet.

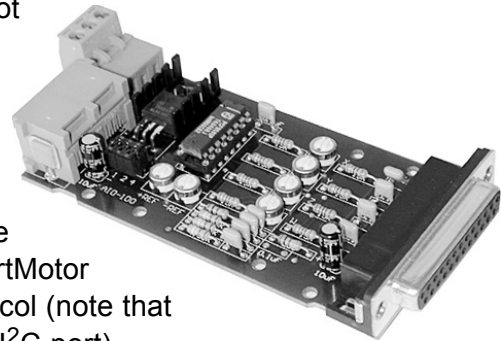


INPUTS AND OUTPUTS

The AIO-100 Card can add 4 analog inputs and 1 analog output to your I²C equipped SmartMotor.

ANILINK I/O MODULES

In the event the on-board I/O is not enough, additional I/O can be connected via the AniLink port. A variety of Analog and Digital I/O cards are available, as well as peripheral devices like LCD and LED displays, push-wheel input devices, pendants and more. These products communicate with the SmartMotor through the AniLink port using I²C protocol (note that ServoStep, however, does not have an I²C port).



OUTPUT ASSIGNMENTS

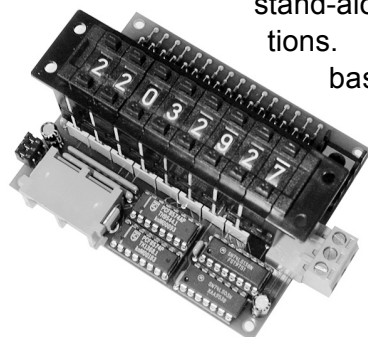
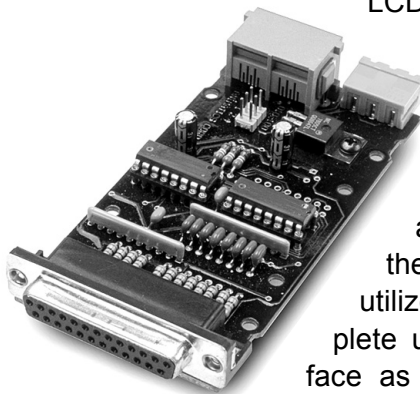
AOUT{address},exp	Output byte to analog address=A-H
DOUT{address}{ch},exp	Output byte to network, address=A-H, ch=0-63

INPUT ASSIGNMENTS

var=AIN{address}{input}	8 bit analog input from network, address=A-H, and input=1-4
var=DIN{address}{ch}	8 bit digital in from network, address=A-H, and ch=0-63

The DIO-100 Card can add 8 digital inputs or 8 digital output, with control lines to interface with parallel devices.

LCD Displays and Push-Wheel banks can complete an entire user interface.



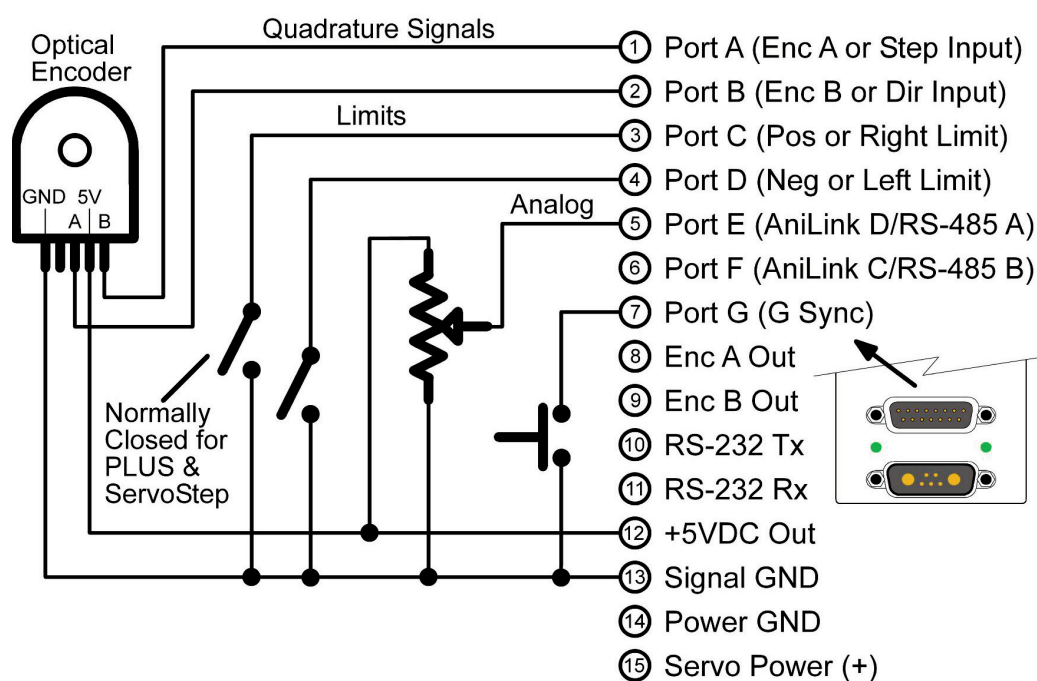
LCD and LED displays offer a means for the SmartMotor to print messages such as instructions and error alerts. PushWheel banks allow a user to enter numeric data without requiring a host computer.

Together, LCD displays and PushWheels allow the SmartMotor to utilize a complete user interface as the foundation of completely stand-alone applications. SmartMotor based machines that are completely independent of a host computer are extremely reliable and "boot-up" in a couple seconds rather than several minutes.

Some stand-alone applications can do without a PushWheel bank where a few buttons and switches connected to I/O are all that is needed.



I/O CONNECTION EXAMPLES



Encoders, Pots, Switches and Buttons are easy to connect directly to the SmartMotor's I/O pins.

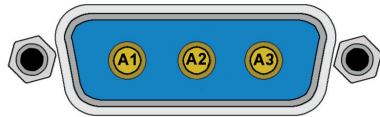
I/O VOLTAGE LEVELS

SmartMotor I/O is logic 0 for voltages below 1.2V and a logic 1 for voltages above 3.0V. Logic states for voltages between these are unpredictable.

INPUTS AND OUTPUTS

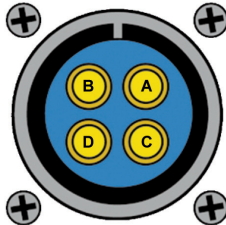
Motor Connector Pin Identifications

ServoStep (size 42) Power



A1 +20 to 75VDC
A3 Ground

4 Pin, 3 Phase Power



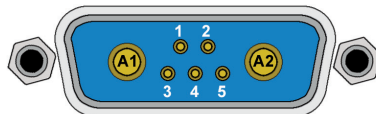
A Phase A
B Phase B
C Phase C
D Chassis

24 Pin Data I/O



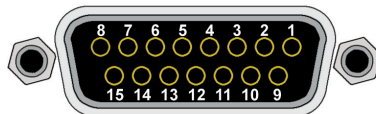
A Ground	N Ext Encoder A/Step/User I/O A
B Ground	P Ext Encoder B/Direction/User I/O B
C Ground	Q Reserved
D Ground	R User I/O G
E RS-232 Transmit	S Left Limit/User I/O D
F RS-232 Receive	T Right Limit/User I/O C
G Reserved	U AniLink Data/User I/O E
H Reserved	V AniLink Clock/User I/O F
J +5V Out, Fused	W Reserved
K Encoder A Out	X Reserved
L Encoder B Out	Y Reserved
M Encoder Index Out	Z Reserved

7 Pin Combo D-sub Power and I/O



A1 +20V to +48v DC	3 SM RS-232 Transmit
A2 Power Ground	4 SM RS-232 Receive
1 Sync or I/O G	5 RS-232 Ground
2 +5V Out	

15 Pin D-sub I/O



1 I/O A	9 Encoder B Out
2 I/O B	10 SM RS-232 Transmit
3 I/O C	11 SM RS-232 Receive
4 I/O D	12 +5V Out
5 I/O E	13 Ground
6 I/O F	14 Power Ground
7 I/O G	15 Power
8 Encoder A Out	

Legacy MOLEX Con:

Encoder I/O



7 Ground
6 +5V Out
5 Encoder Index Out
4 Encoder B Out
3 Encoder A Out
2 Ext Encoder A/Step/User I/O A
1 Ext Encoder B/Direction/User I/O B

AniLink I/O



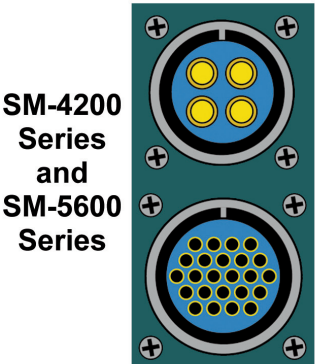
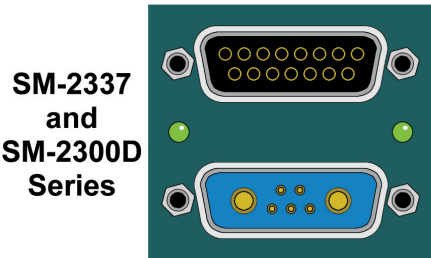
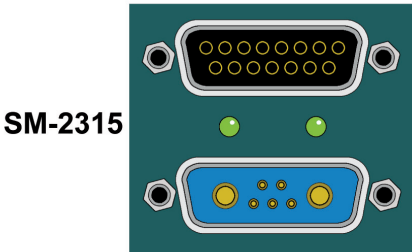
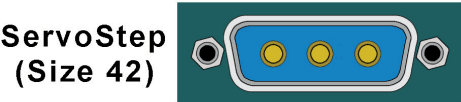
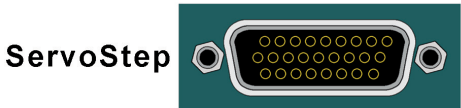
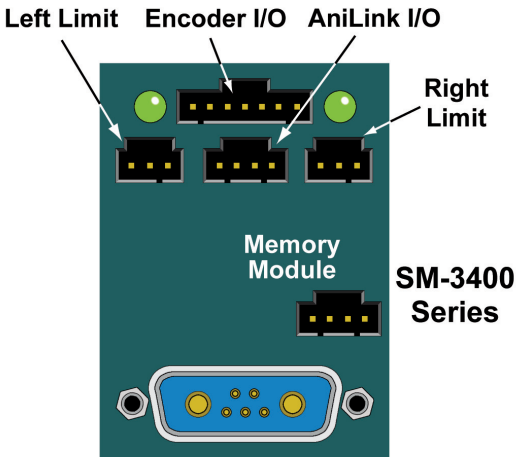
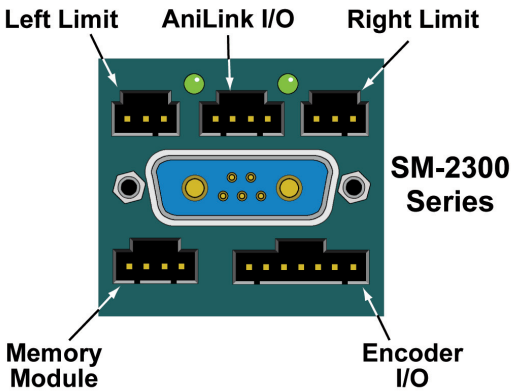
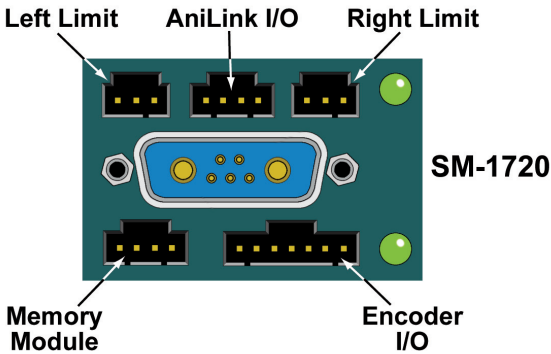
4 Clock/User I/O F
3 Data/User I/O E
2 Ground
1 +5V Out

Left/Right Limit I/O



3 Limit Input
2 Ground
1 +5V Out

INPUTS AND OUTPUTS



Motor Connector Locator

Please note the Memory Module location. In older SmartMotors using Molex Connectors, this module uses the same type of connector as the AniLink I/O. If a Memory Module is plugged into the AniLink I/O, it won't break, but it won't work either.

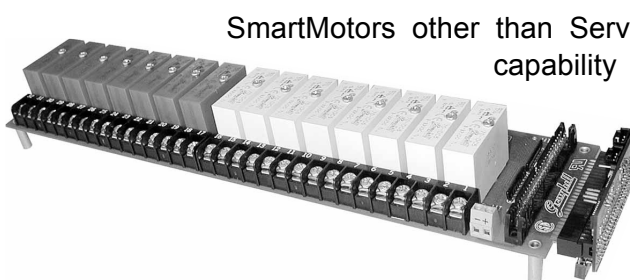
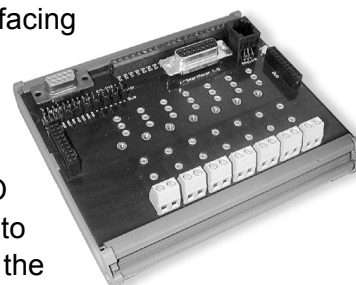
INPUTS AND OUTPUTS

The 5 Volt Logic of the SmartMotor can interface to 24 Volt devices through the use of standard interface modules.

INTERFACING STANDARD I/O MODULES

Animatics offers many convenient ways to connect standard I/O modules such as those identified on the facing page, other than simply wiring them to the SmartMotor's TTL I/O.

The DIN-IO7 is a DIN-Rail mountable platform on which can be mounted up to 7 standard I/O modules. This device conveniently connects to all SmartMotors and can dramatically simplify the wiring of applications requiring this capability. This interfacing means can take advantage of Analog I/O.



SmartMotors other than ServoStep are equipped with I²C capability in their AniLink ports. These motors can connect to platforms that hold as many as 16 standard I/O modules for the most demanding applications.

While there are a variety of options, the default mode for communicating with a SmartMotor is serial RS-232 for the main port, except for the ServoStep who's main port is Isolated RS-485. Most SmartMotors are equipped with a secondary serial port called the **AniLink** port. The **AniLink** port on a SmartMotor can be configured to communicate with either RS-485 or I²C. The I²C connects SmartMotor peripherals like LCD displays, I/O cards, etc., while the RS-485 will interface bar code readers, light curtains, and other "intelligent" peripherals including other SmartMotors if desired. SmartMotor models SMXXX5 do not have RS-485 capability in their **AniLink** ports. Series 4 ServoStep motors have neither secondary RS-485 nor **AniLink**.

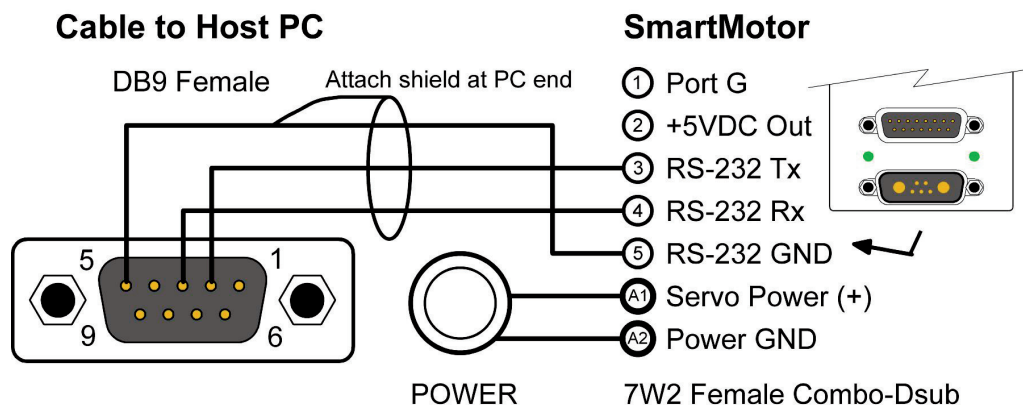
To maximize the flexibility of the SmartMotor, all serial communications ports are fully programmable with regard to bit-rate and protocol.

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 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 (w/adaptor or ServoStep)
Parity:	None	Odd or Even
Bit Rate:	9600	2400 to 38400
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. RS-232 SmartMotors connect as follows (look further for RS-485 SmartMotors and ServoSteps):



When using I²C, the SmartMotor is always the bus master. You cannot communicate between SmartMotors via I²C.



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

COMMUNICATIONS

Because of the buffers on both sides there is no need for any hand shaking 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 communication to regulate the download of a new program.

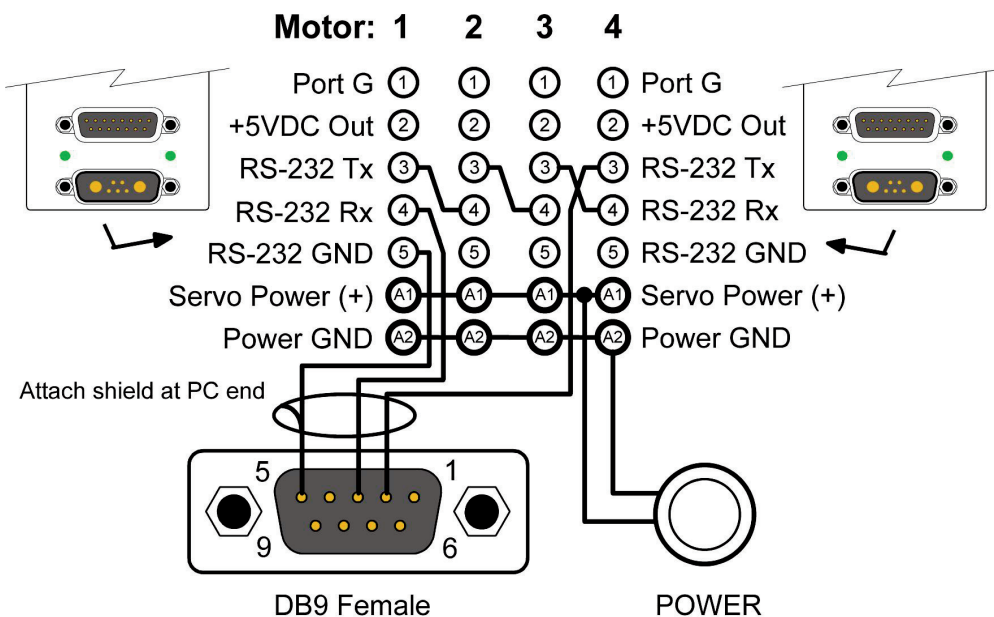
DAISY CHAINING RS-232

Multiple SmartMotors can be connected to a single RS-232 port as shown (For ServoStep, see Communicating over RS-485 further on)

This diagram could be expanded to as many as 120 motors. For independent motion, however, each motor must be programmed with a unique address. In 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.

You can create your own RS-232 Daisy Chain cable or purchase Add-A-Motor Cables from Animatics.

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.



SADDR#

Set motor to new address

The **SADDR#** command causes a SmartMotor to respond exclusively to commands addressed to it. 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 can add 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.

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 (not available in SM2315D and ServoStep).



ECHO, ECHO1

ECHO input

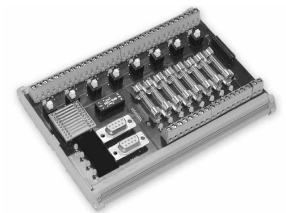
ECHO_OFF, ECHO_OFF1

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.

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

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



SmartMotors can be made to automatically ECHO received characters to the next SmartMotor in a Daisy Chain.

COMMUNICATIONS

Never use ECHO with ServoStep SmartMotors or you will crash the RS-485 bus.

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	

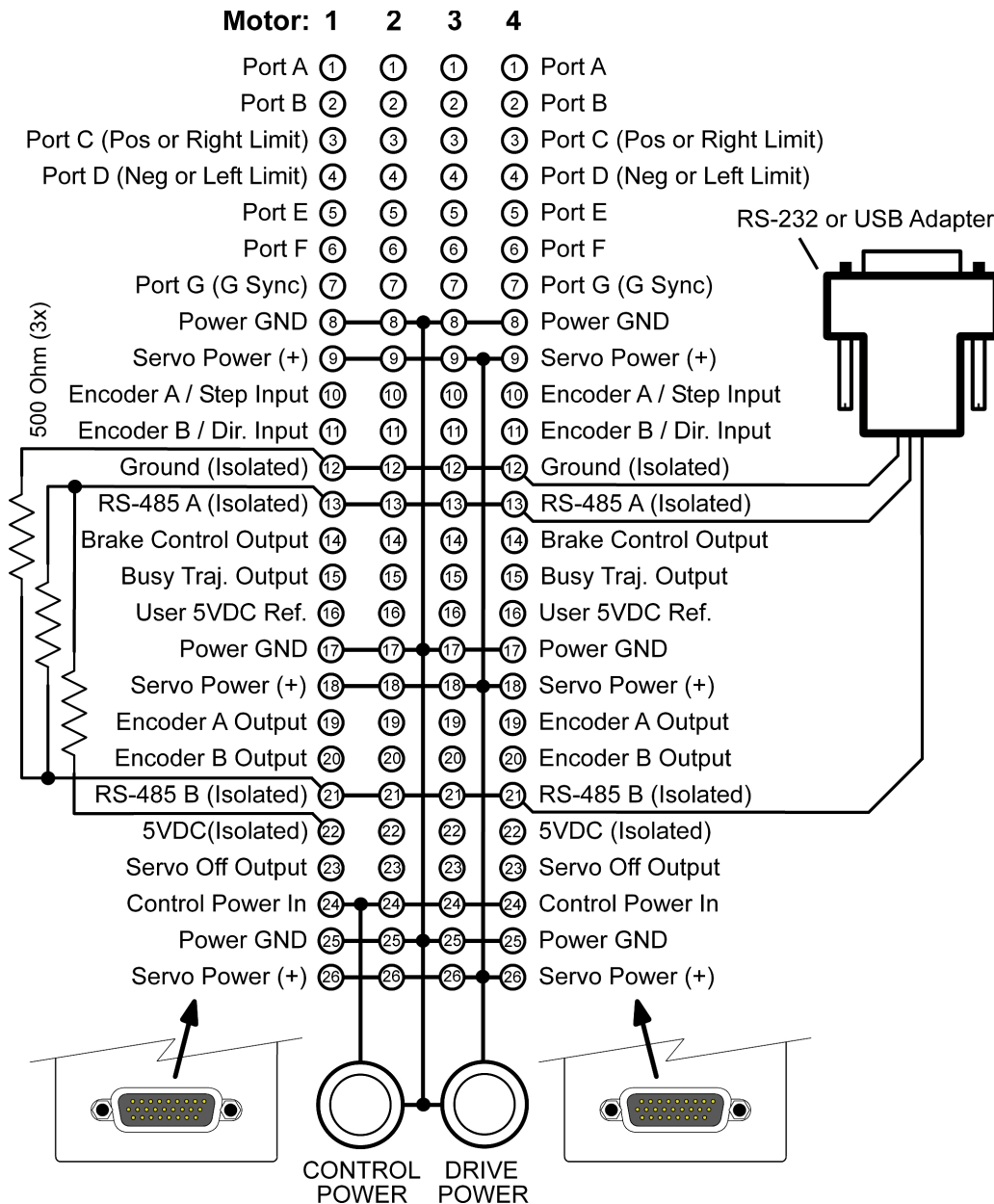
COMMUNICATING OVER RS-485

Multiple ServoStep 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 adaptor 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 A and B RS-485 signals in the ServoStep are isolated, making them immune to ground bounce. Proper cabling would include a shielded twisted pair for transmission.

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



*Plug any ServoStep into a standard PC port using the **RS-232485** or the **USB-232485** Adapter.*



The cable diagram to the left does not show the required shielding. To protect communications, shield the twisted-pair RS-485 signals, grounding the shield only at the Adapter. An additional shield around Drive Power will greatly reduce emissions.



Molded cable assemblies make wiring simple and reliable for ServoStep SmartMotors.

OCHN

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 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)

COMMUNICATIONS

The Main Port of the ServoStep is standard RS-485 and requires no adapter.

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# 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 2400 to 38400 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 SmartMotors use 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 while **PRINTA** prints to the AniLink port using I²C protocol in such a way as to send data to an LCD display or standard parallel input line printer (with a DIO-100 card on the AniLink bus).

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.

! Wait for character to be received

A single exclamation mark will cause program execution to stop until a character is received. This can be handy under certain circumstances like debugging a program in real time.

The following is a very useful routine for confirming RS-232 noise. Start the program from the SMI terminal and then sequentially activate and deactivate every other system in the machine while watching the terminal window. If the SmartMotor gets so much as a single byte of noise, the program will advance to the print statement:

```

WHILE 1           'Loop Forever
    !              'Hold until anything received
    PRINT("Noise Detected",#13)    'Print Message
LOOP             'Loop back to WHILE

```

a=CHN0, a=CHN1 Communications error flags

The **CHN0** and **CHN1** 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:

```

C911
IF CHN0           'If CHN0 != 0
    DOUT0,1        'Home LCD cursor
    IF CHN0&1
        PRINTA("BUFFER OVERFLOW")
    ENDIF
    IF CHN0&2
        PRINTA("FRAMING ERROR")
    ENDIF
    IF CHN0&4
        PRINTA("COMMAND SCAN ERROR")
    ENDIF
    IF CHN0&8
        PRINTA("PARITY ERROR")
    ENDIF

```

```
        CHN0=0          `Reset CHN0
    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 what address it is set to. When a motor is set to an address, the **ADDR** variable will reflect that address from 1 to 120.

GETTING DATA FROM A COM PORT

If a com 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 (First In First Out)** 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. Two things are very important when dealing with a data buffer for the protection of the data:

- 1) Never **GETCHR** if there is no **CHR** to **GET**.
- 2) Never let the buffer overflow.

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:

- 1) Other SmartMotors
- 2) Bar Code Readers
- 3) Light Curtains
- 4) Terminals
- 5) 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.

```

A=500                'Preset Accel.
V=1000000           'Preset Vel.
P=0                 'Zero out Pos.
O=0                 'Declare origin
G                   'Servo in place
OCHN(RS2,0,N,9600,1,8,D)
PRINT("RP",#13)
C0
  IF LEN              'Check for chars
    a=GETCHR          'Get char
    IF a==13          'If carriage return
      G                'Start motion
      P=0              'Reset buffered P to zero
      PRINT("RP",#13)  'Next
    ELSE
      P=P*10          'Shift buffered P
      a=a-48          'Adjust for ASCII
      P=P+a           'Build buffered P
    ENDIF
  ENDIF
GOTO0               '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.

This page has been intentionally left blank.

PID FILTER CONTROL

The SmartMotor™ includes a very high quality, high performance brushless D.C. servomotor. It has a rotor with extremely powerful rare earth magnets and a stator (the outside, stationary part) that is a densely wound multi-slotted electro-magnet.

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

The parameters that makes it all work are found in the **PID** (Proportional, Integral, Derivative) filter section. These are the three fundamental coefficients to a mathematical algorithm that intelligently recalculates and delivers the power needed by the motor about 4,000 times per second. The input to the **PID** filter is the instantaneous actual position minus the desired position, be it at rest, or part of an ongoing trajectory. This difference is called the error.

The **Proportional** parameter of the filter 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.

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 cancel 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. If the bumper was jumped up and down on, however, 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 filter 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 shafts vibration has the same effect as putting a shock absorber in the system, and the algorithm never goes bad.

Even with the two parameters 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 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 over react.

Each of these parameters have their own scaling factor to tailor the overall performance of the filter to the specific load conditions of any one particular

The PID filter is OFF during Torque Mode.

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.

THE PID FILTER

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

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

application. The scaling factors are as follows:

KP	Proportional
KI	Integral
KD	Derivative
KL	Integral Limit

TUNING THE PID FILTER

The task of tuning the filter 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.

KP=exp	Set KP , proportional coefficient
KI=exp	Set KI , time-error coefficient
KD=exp	Set KD , damping coefficient
KL=exp	Set KL , time-error term limit
F	Update PID filter

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 to begin with (**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 physically to 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 limit is reached, find the appropriate derivative compensation. Move **KD** up and down until the position is found that gives the quickest stability. If **KD** is way 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. 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.

E=expression Set maximum position error

The difference between where the motor shaft is and where it is supposed to be is appropriately called the “error”. The magnitude and sign of the error is delivered to the motor in the form of torque, after it is put through the **PID** filter. The higher the error, the more out of control the motor is. Therefore, it is often useful to put a limit on the allowable error, after which time the motor will be turned off. That is what the **E** command is for. It defaults to 1000 encoder counts, but can be set from 1 to 32,000.

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.

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 filter to balance the system. **KG** is the term. **KG** can range from -8388608 to 8388607. To tune **KG**, simply make changes to **KG** until the load equally favors upward and downward motion.

KV=expression Set KVff, 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.

***KV** and **KA** have no effect in **MS** or **MF** follow modes.*

THE PID FILTER

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

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

KA=expression

Set KAff, 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** filter. It's effects can be programmed out with the **KA** term. 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.

KS=expression

Set KS, dampening sample rate

Reduce the sampling rate of the derivative term, **KD**, with the **KS** term. This can sometimes add stability to very high inertial loads. Useful values of **KS** range from 1 (the default) to 20. Results will vary from application to application.

The **PID** rate of the SmartMotor can be slowed down.

PID1 Set normal **PID** update rate

PID2 Divide normal **PID** update rate by 2

PID4 Divide normal **PID** update rate by 4

PID8 Divide normal **PID** update rate by 8

The trajectory and **PID** filter calculations occur within the SmartMotor™ 4069 times per second. That is faster than is necessary for very good control, especially with the larger motors. A reduction in the **PID** rate can result in an increase in the SmartMotor™ application program execution rate. The **PID2** command will divide the **PID** rate by two, and the others even more. The most dramatic effect on program execution rate occurs with **PID4**. **PID8** does little more and is encroaching upon poor control. If the **PID** rate is lowered, keep in mind that this is the "sample" rate that is the basis for **Velocity** values, **Acceleration** values, **PID** coefficients and **WAIT** times. If the rate is cut in half, expect to do the following to keep all else the same:

Halve **WAIT** times

Double **Velocity**

Increase **Acceleration** by a factor of 4

KGON

Change Drive Characteristic for Vertical Application
(no longer supported, see F= mode register)

KGOFF

Restore Drive Characteristic to Default
(no longer supported, see F= mode register)

CURRENT LIMIT CONTROL

AMPS=expression Set current limit, 0 to 1023

In some applications, if the motor misapplied 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 put out. Use the **AMPS** command with a number, variable or expression within the range of 0 to 1023. The units are tenths of a percent of full scale peak current, and varies in actual torque with the size of the SmartMotor.

Current is limited by limiting the maximum PWM duty cycle. For this reason, it will reduce the maximum speed of the motor as well. The **AMPS** command has no effect in Torque Mode.

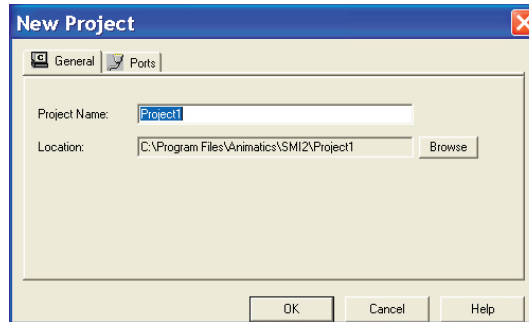
This page has been intentionally left blank.

SMI SOFTWARE

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

SMI PROJECTS

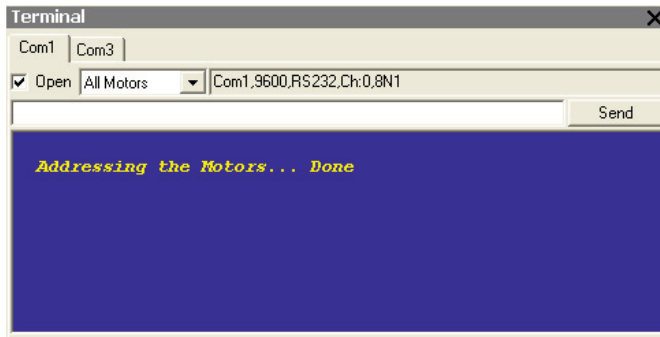
In applications with more than one SmartMotor and possibly more than one program or communications port, it is helpful to organize all of the elements as a **PROJECT**, rather than deal with individual files. Projects can be created from the FILE menu. When starting a new project, you have the option 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.



*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.*

TERMINAL WINDOW

The **Terminal** Window acts as a Real Time portal between you and the SmartMotor. By typing commands in the **Terminal**, 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 and 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 pro-

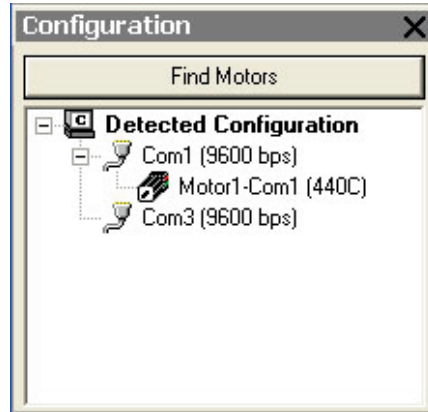
gram will automatically send the appropriate codes to the network to route the data to the intended motors. Commands can be entered in the white text window or the blue screen. If data is flooding back from the motor, then the white text window will be more convenient.

PRINT Statements containing data can be sprinkled in programs to send data up to the Terminal Window as an aid to debugging. Data with associated report commands like Position with the "RP" command can be more 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. Try putting in a **WAIT=50** command. The Terminal Window has a scroll feature that allows the user to review history.

The **Configuration Window** is essential to keeping multiple-SmartMotor systems organized.

CONFIGURATION WINDOW


The **Configuration Window** both shows the current configuration and allows access to specific ports and motors to alter properties. Press "Find Motors", or the Address Button



to detect and analyze your system. 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

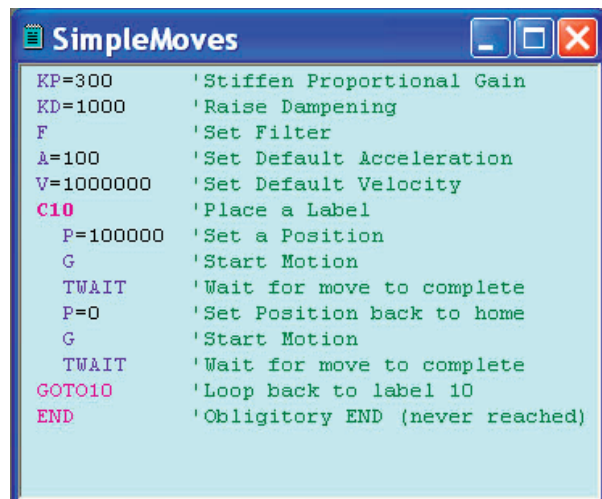
SmartMotor programs are written in the SMI **Program Editor** before being scanned for errors and downloaded to the motor. To get the Program Editor to appear, simply go to the **FILE** menu and select **NEW** or simply press the  button on the toolbar. As you write your program, the editor will highlight 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.

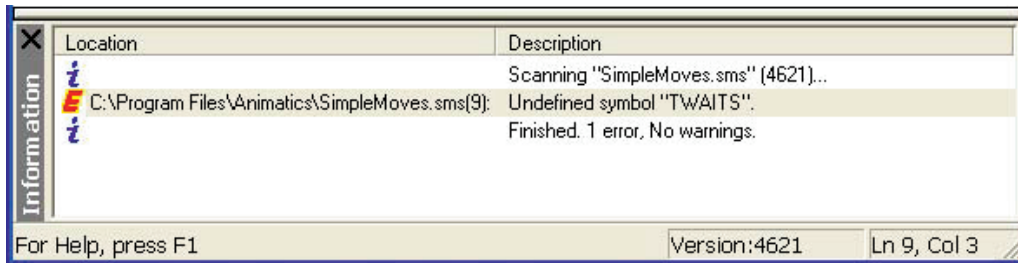
Every program requires an **END**, even if the program is designed to run indefinitely and the **END** is never reached.

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.

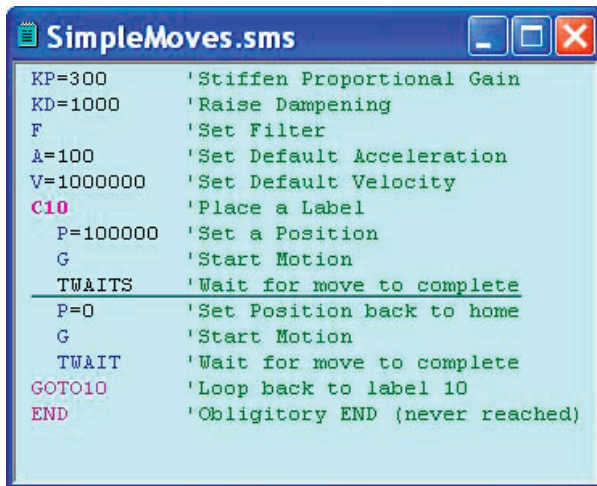
Once a program is complete, you can simply scan it for errors by pressing the  button on the toolbar or scan and download it at one time by pressing the  button. If errors are found, the download will be aborted and the problems will be identified in the **Information Window** located at the bottom of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



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 **E**.



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 P.C. and the SmartMotor, press the button and the **Serial Data Analyzer Window** will appear.



*Program Errors can be located instantly by double-clicking on the error listed in the **Information Window**.*

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

SMI ADVANCED FUNCTIONS

MotorView provides a window into the inter workings of a SmartMotor, in Real-Time.

The 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 to the data channel to be monitored.

MOTOR VIEW

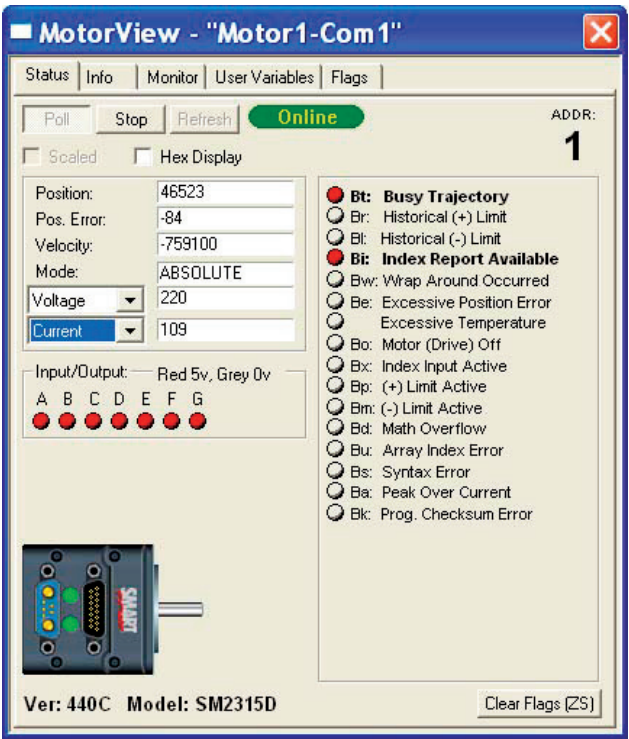
The SMI **Motor View Window** enables the user to view multiple parameters related to the motor, in real time. It is most conveniently accessible by double clicking the motor of interest in the configuration window.

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

A program can be running in the motor while the **MotorView Window** is polling so long as the program itself does not print text to the serial channel being used for the polling.

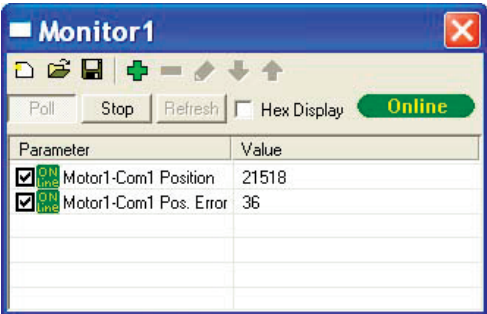
In addition to the standard items displayed, there are two fields that allow the user to select from a list of additional parameters to display. In the example here, Voltage and Current are polled. This information can be useful when setting up a system for the first time, or debugging a system in the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

Newer SmartMotors have built-in provisions to allow 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 **MotorView Window**. Tabs across the top offer a wealth of additional information.



THE SMI 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 custom monitor. You can find the **Monitor Window** by going to the Tools menu and selecting "Monitor View".



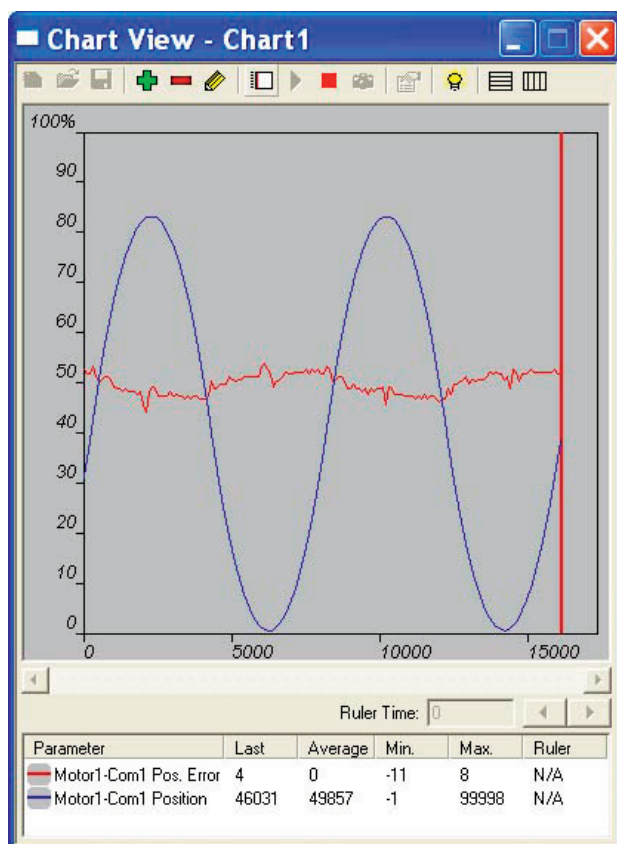
Polling items can be added by pressing the **+** button. The "Add new Monitor Item" window will appear and offer special fields for every portion of the monitoring function.

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.

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.

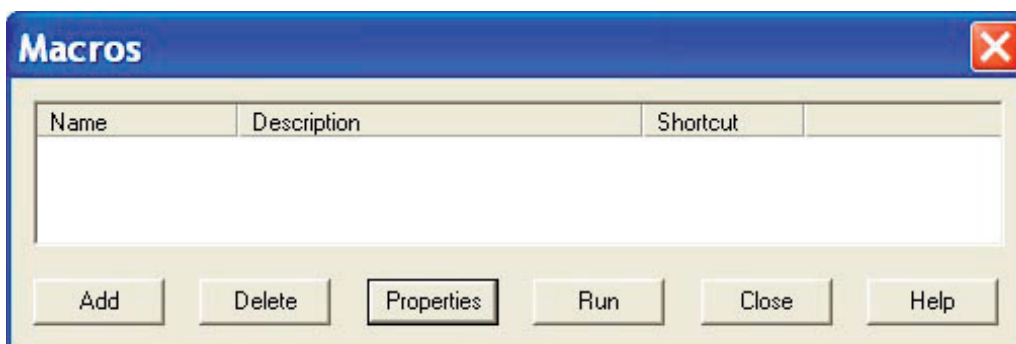
While Chart View does not have an intrinsic printing function for a paper copy, Window's standard "Print Screen" key can capture the graph to be pasted into

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.

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.

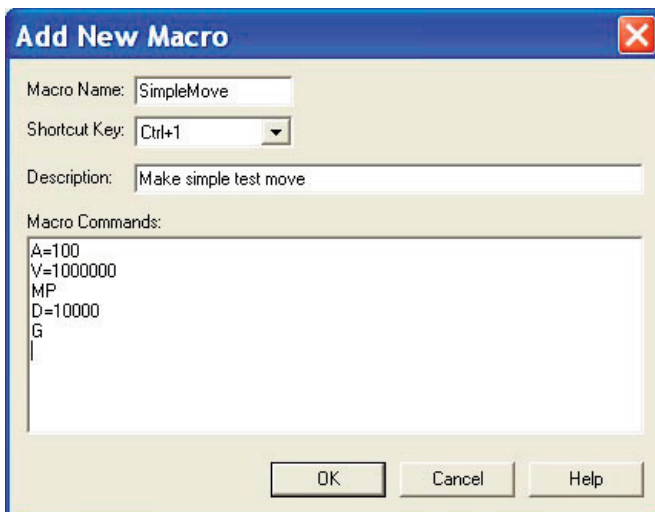
MACROS

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.



SMI ADVANCED FUNCTIONS

To add a macro, start by pressing the ADD button in the Macro Window.



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

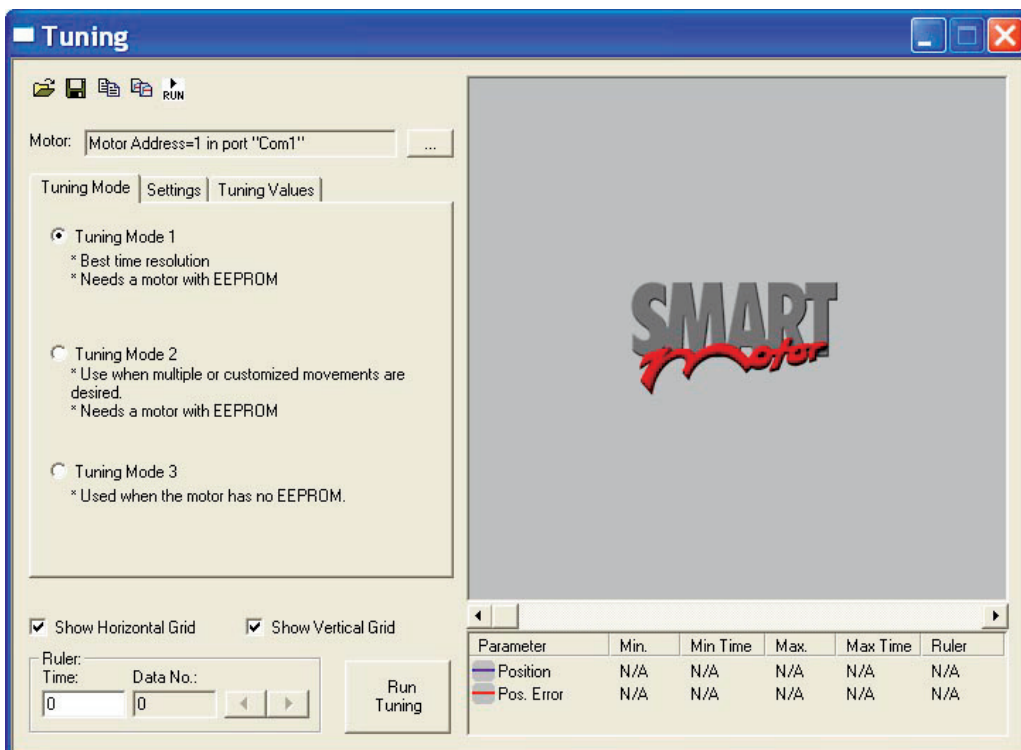
When this is complete, press the OK button. You will again be presented with the Macro window. Click once on the macro you have written and press the "RUN" button in the Macro window to test it.

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 more simple than tuning traditional servos, but it can be even easier using the SMI Tuner to see the actual results of different tuning parameters.

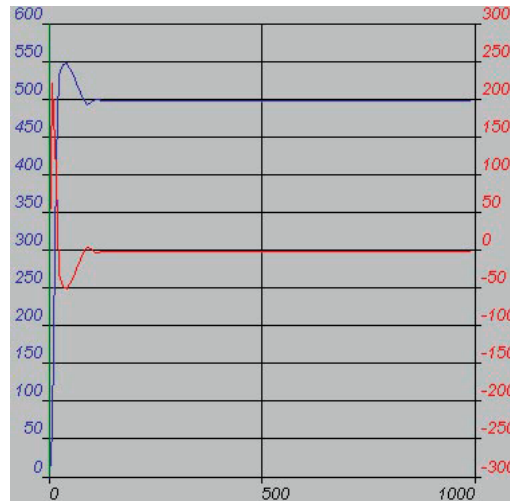
For information on how to tune a SmartMotor, refer to the preceding section "The PID Filter". Each SmartMotor has very soft default tuning. Increasing the stiffness of that tuning can increase the accuracy of the machine the SmartMotor is controlling. To bring



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 SmartMotors 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.

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 Tuning Window. If the SmartMotor was connected, 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 Smart-Motor's actual position over time. Notice that it overshoot its target position before settling in.



This is the soft default tuning for an SM2315D at 24V. Exercising the procedure outlined in the preceding 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 and so significant overshoot will not exist. Never the less, it is useful to look at the "worst case scenario" of a Step Response.

Tuning Mode	Settings	Tuning Values																											
		<table border="1"> <thead> <tr> <th></th> <th>Motor</th> <th>New</th> </tr> </thead> <tbody> <tr> <td>KP (Proportional coefficient):</td> <td>42</td> <td>250</td> </tr> <tr> <td>KI (Integral coefficient):</td> <td>28</td> <td>28</td> </tr> <tr> <td>KD (Differential coefficient):</td> <td>550</td> <td>1500</td> </tr> <tr> <td>KL (Integral limit):</td> <td>20</td> <td>20</td> </tr> <tr> <td>KS (Differential sample rate):</td> <td>1</td> <td>1</td> </tr> <tr> <td>KV (Velocity feed forward):</td> <td>0</td> <td>0</td> </tr> <tr> <td>KA (Acceleration feed forward):</td> <td>0</td> <td>0</td> </tr> <tr> <td>KG (Gravitational coefficient):</td> <td>0</td> <td>0</td> </tr> </tbody> </table>		Motor	New	KP (Proportional coefficient):	42	250	KI (Integral coefficient):	28	28	KD (Differential coefficient):	550	1500	KL (Integral limit):	20	20	KS (Differential sample rate):	1	1	KV (Velocity feed forward):	0	0	KA (Acceleration feed forward):	0	0	KG (Gravitational coefficient):	0	0
	Motor	New																											
KP (Proportional coefficient):	42	250																											
KI (Integral coefficient):	28	28																											
KD (Differential coefficient):	550	1500																											
KL (Integral limit):	20	20																											
KS (Differential sample rate):	1	1																											
KV (Velocity feed forward):	0	0																											
KA (Acceleration feed forward):	0	0																											
KG (Gravitational coefficient):	0	0																											
<input type="button" value="Copy Values"/>		<input type="button" value="Apply new values"/>																											
<input type="button" value="Load Saved Values"/>		<input type="button" value="Save Values"/>																											

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 existing 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, we change KP to 250 and KD to 1500, then clicked the "Apply new values" button.

Now, these new values are in the SmartMotor and we can execute the test of another Step Response by

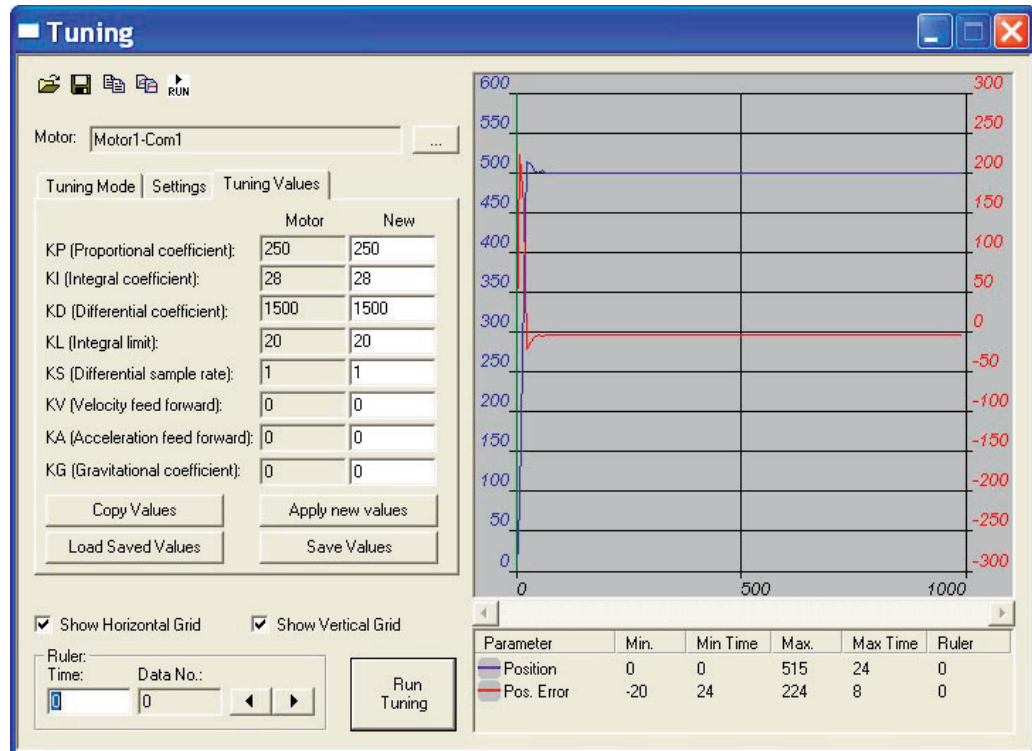
pressing the "Run tuner" button at the bottom of the Tuning Window. The motor will jolt again and the results of the Step Response will overwrite the previous graph.

Normally, this process involves repeated trials, again, exercising the procedure outlined in the previous section on "The PID Filter".

In this example, we stiffened up the tuning by raising KP and increased the KD (or dampening) to keep the motor stable.

SMI ADVANCED FUNCTIONS

The results are a significant reduction in Overshoot as seen in the graph.



Once you are happy with the results, the best parameters 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=250
KI=28
KD=1500
KL=20
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 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 go more to the issues of preferences.

SMI HELP

The most complete and up-to-date information available for SMI functions is available within the programs 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.

This page has been intentionally left blank.

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 screen 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		

This page has been intentionally left blank.

The SmartMotor'stm 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 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

APPENDIX B: BINARY DATA

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:

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
1 0 0 0 0 0 0 0	128
1 0 0 0 0 0 0 1	129
1 1 1 1 1 1 1 1	255

Consider the following two bytes of information:

Byte	Value
0 0 1 1 1 1 0 0	60
0 0 0 1 1 1 1 0	30

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 byte. A single value can be uploaded from a SmartMotor and have coded into it, in binary, eight or sixteen independent bits of information.

The following is the status byte and its coded information:

Name	Description	Bit	Value
Bo	Motor OFF	7	128
Bh	Excessive temp.	6	64
Be	Excessive pos. err.	5	32
Bw	Wraparound	4	16
Bi	Index reportable	3	8
Bm	Real time neg. lim.	2	4
Bp	Real time pos. lim.	1	2
Bt	Trajectory going	0	1

There are two useful mathematical operators that work on binary data, the “&” (and) and the “|” (or). The “&” compares two bytes, words or longs and looks for what they have in common. The resulting data has ones only where there were ones in both the first byte and the second. The “|” looks for a one in the same location of either the first data field or the second. Both functions are illustrated in the following example:

A	B	A&B	A B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

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

Example 1:

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

Example 2:

```
IF S&6        'Look at both lim
  GOSUB100    'Execute subroutine
ENDIF
```

Both examples will execute subroutine 100 if both limit inputs are high. By “anding” the status byte (S) by six, the second routine filters out all of the other status information. If either limit is high, then the result will be non-zero and subroutine 100 will execute. Example two uses much less code than example one and will run much 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 UAI        'Look for input A
  GOSUB200    'Execute subroutine
ENDIF
IF UBI        'look for input B
  GOSUB200    'Execute subroutine
ENDIF
```

Example 4:

```
IF UAI|UBI    'Look at both A,B
  GOSUB200    'Execute subroutine
ENDIF
```

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

This page has been intentionally left blank.

!	(exclamation point)
(space)	Single space between user variables
@P	Current position
@PE	Current position error
@V	Current velocity
a . . . z	User variables
aa . . . zzz	More user variables
al[index]	Array variable 32 bit
aw[index]	Array variable 16 bit
ab[index]	Array variable 8 bit
A=exp	Set acceleration
ADDR	Motor's self address variable
AIN{port}{channel}	Assign input byte from module
AMPS=expression	Set PWM drive signal limit
AOUT{port}{expression}	Output analog byte to module
Ba	Over current status bit
Bb	Parity error status bit
Bc	Communication overflow status bit
Bd	Math overflow status bit
Be	Excessive position error status bit
Bf	Communications framing error status bit
Bh	Excessive temperature status bit
Bi	Index captured status bit
Bk	EEPROM data integrity status bit
Bl	Historical left limit status bit
Bm	Real time left limit status bit
Bo	Motor off status bit
Bp	Real time right limit status bit
Br	Historical right limit status bit

APPENDIX C: COMMANDS

Bs	Syntax error status bit
Bt	Trajectory in progress status bit
Bu	Array index error status bit
Bv	EEPROM locked state (obsolete)
Bw	Encoder wrap around status bit
Bx	Real time index input status bit
BASE	Cam encoder count cycle length
BAUD	Host communications control
BRKENG	Brake engage
BRKRLS	Brake release
BRKSRV	Brake without servo
BRKTRJ	Brake without trajectory
BRKC	Route Brake Sig. to I/O C (PLUS & ServoStep)
BRKG	Route Brake Sig. to I/O G (PLUS & ServoStep)
BRKI	Restore Brake to Internal (PLUS & ServoStep)
BREAK	Program execution flow control
C#	Program subroutine label
CCHN{type}{channel}	Close communications channel
CHN0	RS-232 communications error flags
CHN1	RS-485 communications error flags
CLK	Hardware clock variable
CI	Cam Mode Re-initialize (PLUS & ServoStep)
CTR	Second encoder/step and direction counter
CX	Current Cam Index value (PLUS & ServoStep)
D=exp	Set relative distance
DEFAULT	Switch-case structure element
DIN{port}{channel}	Input byte from module
DOU{port}{channel}{expression}	Output byte to module
E=expression	Set allowable position error
ECHO	Echo input data back out main channel

ECHO_OFF	Stop echo main channel
ECHO1	Echo input data back out second channel
ECHO1_OFF	Stop echo second channel
ELSE	If structure element
ENC0	Select internal encoder for servo
ENC1	Select external encoder for servo
END	End program
ENDIF	End IF statement
EPTR=expression	Set data EEPROM pointer
ES400*	Slow data EEPROM read/write speed
ES1000*	Increase data EEPROM read/write speed
F	Load filter
F=expression	Special functions control (See Appendix F)
G	Start motion (GO)
GETCHR	Get character from main comm channel
GETCHR1	Get character from second comm channel
GOSUB#	Call a subroutine
GOTO#	Branch program execution to a label
I (capital i)	Hardware index position variable
IF expression	Conditional test
KA=expression	PID acceleration feed-forward
KD=expression	PID derivative compensation
KG=expression	PID gravity compensation
KGOFF**	PID gravity mode off
KGON**	PID gravity mode on
KI=expression	PID integral compensation
KL=expression	PID integral limit
KP=expression	PID proportional compensation
KS=expression	PID derivative term sample rate
KV=expression	PID velocity feed forward

**These commands removed in PLUS & ServoStep Firmware. They are obsolete in V4.15 and above firmware.*

***These commands removed in PLUS & ServoStep Firmware.*

APPENDIX C: COMMANDS

**These commands
removed in PLUS
& ServoStep
Firmware.*

***This command
removed in PLUS
& ServoStep
Firmware.*

LEN	Main comm chnl buffer fill level, data mode
LEN1	Second comm chnl buffer fill level, data mode
LIMD	Enable directional constraints on limit inputs
LIMH*	Limit active high
LIML*	Limit active low
LIMN*	Restore non-directional limits
LOAD	Initiate program download to motor
LOOP	While structure element
MC	Enable cam mode
MC2	Enable cam mode with position scaled x2
MC4	Enable cam mode with position scaled x4
MC8	Enable cam mode with position scaled x8
MD	Enable contouring mode
MD50**	Enable drive mode
MF0	Set mode follow for variable only
MF1	Configure follow hardware for x1 scaling
MF2	Configure follow hardware for x2 scaling
MF4	Configure follow hardware for x4 scaling
MFDIV	Mode follow with ratio divisor
MFMUL	Mode follow with ratio multiplier
MFR	Initiate mode follow ratio calculation
MP	Enable position mode
MS	Enable step and direction input mode
MS0	Configure step and direction for variable only
MSR	Initiate mode step ratio calculation
MT	Enable torque mode
MTB	Mode Torque Brake (PLUS & ServoStep)
MV	Enable velocity mode
O=expression	Set origin
OCHN	Open main communications channel

OFF	Stop servoing the motor
P=expression	Set position
PID1	Restore PID sample rate to default
PID2	Divide PID sample rate by two
PID4	Divide PID sample rate by four
PID8	Divide PID sample rate by eight
PRINT{expression}	Print data to main comm channel
PRINT1{expression}	Print data to second comm channel
PRINT{port}{expression}	Print data to AniLink peripheral
Q	Report status in contouring mode
Ra . . . Rz	Report variables
Raa . . . Rzz	Report variables
Raaa . . . Rzzz	Report variables
Rab[index]	Report byte array variables (8-bit)
Ral[index]	Report long array variables (32-bit)
Raw[index]	Report word array variables (16-bit)
RA	Report acceleration
RAIN{expression}{input}	Report value from analog AniLink card
RAMPS	Report assigned max. drive PWM limit
RBa	Report over current status
RBb	Report parity error status
RBc	Report communications error status
RBd	Report user math overflow status
RBe	Report position error status
RBf	Report communications framing error status
RBh	Report overheat status
RBi	Report index status
RBk	Report EEPROM read/write status
RBl	Report historical left limit status
RBm	Report negative limit status

APPENDIX C: COMMANDS

RBo	Report motor off status
RBp	Report positive limit status
RBr	Report historical right limit status
RBs	Report program scan status
RBt	Report trajectory status
RBu	Report user array index status
RBw	Report wrap around status
RBx	Report hardware indexinput level
RCHN	Report combined communications status
RCHN0	Report RS-232 communications status
RCHN1	Report RS-485 communications status
RCS	Report RS-232 communications check sum
RCS1	Report RS-485 communications check sum
RCTR	Report secondary counter
RD	Return buffered move distance value
RDIN{port}{channel}	Report value from digital AniLink card
RE	Report buffered maximum position error
RETURN	Return from subroutine
RETURNF	Ret from C1 Interrupt Sub (PLUS & ServoStep)
RETURNI	Ret from C2 Interrupt Sub (PLUS & ServoStep)
RI	Report last stored index position
RKA	Report buffered acceleration feed forward coef.
RKD	Report buffered derivative coefficient
RKG	Report buffered gravity coefficient
RKI	Report buffered integral coefficient
RKL	Report buffered integral limit
RKP	Report buffered proportional coefficient
RKS	Report buffered sampling interval
RKV	Report buffered velocity feed forward coefficient
RMODE	Report current mode of operation

RP	Report present position
RPE	Report present position error
RPW	Report position and status
RS	Report status byte
RT	Report current requested torque
RU	Report all I/O (PLUS & ServoStep)
RU{pin}	Report digital I/O value (PLUS & ServoStep)
RU{pin}A	Report analog I/O value (PLUS & ServoStep)
RUN	Execute stored program
RUN?	Override automatic program execution
RV	Report velocity
RW	Report status word
S (as command)	Stop move in progress abruptly
SADDR#	Set motor to new address
SILENT	Suppress PRINT messages main channel
SILENT1	Suppress PRINT messages second channel
SIZE=expression	Number of data entries in cam table
SLD	Disable Software Limits (PLUS & ServoStep)
SLE	Enable Software Limits (PLUS & ServoStep)
SLEEP	Initiate sleep mode main channel
SLEEP1	Initiate sleep mode second channel
SLN=<exp>	Set Neg Software Limit (PLUS & ServoStep)
SLP=<exp>	Set Pos Software Limit (PLUS & ServoStep)
STACK	Reset nesting stack tracking
SWITCH expression	Program execution control
T=expression	Assign torque value in torque mode
TALK	Enable PRINT messages on main channel
TALK1	Enable PRINT messages on main channel
TEMP	Temperature variable
TH	Sets high temperature set point

APPENDIX C: COMMANDS

THD	Sets temperature fault delay
TWAIT	Pause program during a move
U	7 bit val of combined I/O (PLUS & ServoStep)
UA=expression	Set I/O A output
UAA	I/O A analog input value (0 to 1024)
UAI (as command)	Set I/O A to input
UAI (as input value)	I/O A input value variable
UAO (as command)	Set I/O A to output
UB=expression	Set I/O B output
UBA	I/O B analog input value (0 to 1024)
UBI (as command)	Set I/O B to input
UBI (as input value)	I/O B input value variable
UBO (as command)	Set I/O B to output
UC=expression	Set I/O C output
UCA	I/O C analog input value (0 to 1024)
UCI (as command)	Set I/O C to input
UCI (as input value)	I/O C input value variable
UCO (as command)	Set I/O C to output
UCP (as command)	Set I/O C to be a right limit input
UD=expression	Set I/O D output
UDA	I/O D analog input value (0 to 1024)
UDI (as command)	Set I/O D to input
UDI (as input value)	I/O D input value variable
UDM (as command)	Set I/O D to be a left limit input
UDO (as command)	Set I/O D to output
UE=expression	Set I/O E output
UEA	I/O E analog input value (0 to 1024)
UEI (as command)	Set I/O E to input
UEI (as input value)	I/O E input value variable
UEO (as command)	Set I/O E to output

UF=expression	Set I/O F output
UFA	I/O F analog input value (0 to 1024)
UFI (as command)	Set I/O F to input
UFI (as input value)	I/O F input value variable
UFO (as command)	Set I/O F to output
UG=expression	Set I/O G output
UGA	I/O G analog input value (0 to 1024)
UGA (as command)	Set I/O G to G synchronous function
UGI (as command)	Set I/O G to input
UGI (as input value)	I/O G input value variable
UGO (as command)	Set I/O G to output
UIA	Read Current (Amps = UIA/100)
UJA	Read Voltage (Volts = UJA/10)
UP	Upload user EEPROM program contents
UPLOAD	Upload user EEPROM readable program
V=expression	Set maximum permitted velocity
VLD	Sequentially load variables from data EEPROM
VST	Sequentially store variables to data EEPROM
WAIT=expression	Suspends program for number of PID samples
WAKE	Terminate sleep mode main channel
WAKE1	Terminate sleep mode second channel
WHILE expression	Conditional program flow command
X	Slow motor motion to stop
Z	Total system reset
Za	Reset current limit violation latch bit
Zb	Reset serial data parity violation latch bit
Zc	Reset communications buffer overflow latch bit
Zd	Reset math overflow violation latch bit
Ze	Reset zero pos error flag (PLUS & ServoStep)
Zf	Reset serial comm framing error latch bit

APPENDIX C: COMMANDS

Zh	Reset zero overheat flag (PLUS & ServoStep)
Zl	Reset historical left limit latch bit
Zr	Reset historical right limit latch bit
Zs	Reset command scan error latch bit
Zu	Reset user array index access latch bit
Zw	Reset encoder wrap around event latch bit
ZS	Reset system latches to power-up state

APPENDIX D: DATA VARIABLES MEMORY MAP

aw[0] is the most significant word of al[0], and ab[0] is the most significant byte of aw[0] and al[0] (aka "aa").

aa	al[0]	aw[0]	ab[0]
			ab[1]
bb	al[1]	aw[1]	ab[2]
			ab[3]
cc	al[2]	aw[2]	ab[4]
			ab[5]
dd	al[3]	aw[3]	ab[6]
			ab[7]
ee	al[4]	aw[4]	ab[8]
			ab[9]
ff	al[5]	aw[5]	ab[10]
			ab[11]
gg	al[6]	aw[6]	ab[12]
			ab[13]
hh	al[7]	aw[7]	ab[14]
			ab[15]
ii	al[8]	aw[8]	ab[16]
			ab[17]
jj	al[9]	aw[9]	ab[18]
			ab[19]
kk	al[10]	aw[10]	ab[20]
			ab[21]
ll	al[11]	aw[11]	ab[22]
			ab[23]
mm	al[12]	aw[12]	ab[24]
			ab[25]
			ab[26]
			ab[27]
			ab[28]
			ab[29]
			ab[30]
			ab[31]
			ab[32]
			ab[33]
			ab[34]
			ab[35]
			ab[36]
			ab[37]
			ab[38]
			ab[39]
			ab[40]
			ab[41]
			ab[42]
			ab[43]
			ab[44]
			ab[45]
			ab[46]
			ab[47]
			ab[48]
			ab[49]
			ab[50]
			ab[51]

nn	al[13]	aw[26]	ab[52]
			ab[53]
oo	al[14]	aw[27]	ab[54]
			ab[55]
pp	al[15]	aw[28]	ab[56]
			ab[57]
qq	al[16]	aw[29]	ab[58]
			ab[59]
rr	al[17]	aw[30]	ab[60]
			ab[61]
ss	al[18]	aw[31]	ab[62]
			ab[63]
tt	al[19]	aw[32]	ab[64]
			ab[65]
uu	al[20]	aw[33]	ab[66]
			ab[67]
vv	al[21]	aw[34]	ab[68]
			ab[69]
ww	al[22]	aw[35]	ab[70]
			ab[71]
xx	al[23]	aw[36]	ab[72]
			ab[73]
yy	al[24]	aw[37]	ab[74]
			ab[75]
zz	al[25]	aw[38]	ab[76]
			ab[77]
		aw[39]	ab[78]
			ab[79]
		aw[40]	ab[80]
			ab[81]
		aw[41]	ab[82]
			ab[83]
		aw[42]	ab[84]
			ab[85]
		aw[43]	ab[86]
			ab[87]
		aw[44]	ab[88]
			ab[89]
		aw[45]	ab[90]
			ab[91]
		aw[46]	ab[92]
			ab[93]
		aw[47]	ab[94]
			ab[95]
		aw[48]	ab[96]
			ab[97]
		aw[49]	ab[98]
			ab[99]
		aw[50]	ab[100]
			ab[101]
		aw[51]	ab[102]
			ab[103]

APPENDIX D: DATA VARIABLES MEMORY MAP

aaa	al[26]	aw[52]	ab[104]
			ab[105]
		aw[53]	ab[106]
			ab[107]
bbb	al[27]	aw[54]	ab[108]
			ab[109]
		aw[55]	ab[110]
			ab[111]
ccc	al[28]	aw[56]	ab[112]
			ab[113]
		aw[57]	ab[114]
			ab[115]
ddd	al[29]	aw[58]	ab[116]
			ab[117]
		aw[59]	ab[118]
			ab[119]
eee	al[30]	aw[60]	ab[120]
			ab[121]
		aw[61]	ab[122]
			ab[123]
fff	al[31]	aw[62]	ab[124]
			ab[125]
		aw[63]	ab[126]
			ab[127]
ggg	al[32]	aw[64]	ab[128]
			ab[129]
		aw[65]	ab[130]
			ab[131]
hhh	al[33]	aw[66]	ab[132]
			ab[133]
		aw[67]	ab[134]
			ab[135]
iii	al[34]	aw[68]	ab[136]
			ab[137]
		aw[69]	ab[138]
			ab[139]
jjj	al[35]	aw[70]	ab[140]
			ab[141]
		aw[71]	ab[142]
			ab[143]
kkk	al[36]	aw[72]	ab[144]
			ab[145]
		aw[73]	ab[146]
			ab[147]
lll	al[37]	aw[74]	ab[148]
			ab[149]
		aw[75]	ab[150]
			ab[151]
mmm	al[38]	aw[76]	ab[152]
			ab[153]
		aw[77]	ab[154]
			ab[155]

nnn	al[39]	aw[78]	ab[156]
			ab[157]
		aw[79]	ab[158]
			ab[159]
ooo	al[40]	aw[80]	ab[160]
			ab[161]
		aw[81]	ab[162]
			ab[163]
ppp	al[41]	aw[82]	ab[164]
			ab[165]
		aw[83]	ab[166]
			ab[167]
qqq	al[42]	aw[84]	ab[168]
			ab[169]
		aw[85]	ab[170]
			ab[171]
rrr	al[43]	aw[86]	ab[172]
			ab[173]
		aw[87]	ab[174]
			ab[175]
sss	al[44]	aw[88]	ab[176]
			ab[177]
		aw[89]	ab[178]
			ab[179]
ttt	al[45]	aw[90]	ab[180]
			ab[181]
		aw[91]	ab[182]
			ab[183]
uuu	al[46]	aw[92]	ab[184]
			ab[185]
		aw[93]	ab[186]
			ab[187]
vvv	al[47]	aw[94]	ab[188]
			ab[189]
		aw[95]	ab[190]
			ab[191]
www	al[48]	aw[96]	ab[192]
			ab[193]
		aw[97]	ab[194]
			ab[195]
xxx	al[49]	aw[98]	ab[196]
			ab[197]
		aw[99]	ab[198]
			ab[199]
yyy	al[50]	aw[100]	ab[200]
			ab[201]*
		aw[101]*	ab[202]*
			ab[203]*

* For versions 4.16 and above

Note that **zzz** is used by the **SWITCH** statement and not available to the user program when **SWITCH** is used.

APPENDIX E: EXAMPLE PROGRAMS

Remember that if you are programming a ServoStep or a SmartMotor with PLUS firmware, you will need to connect the limit switches OR enter the following at or near the top of the program:

UCI
UDI
ZS

MOVING BACK AND FORTH

About the most simple program that can be written is 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.

KP=200	'Increase stiffness from default
KD=1000	'Increase dampening from default
F	'Activate new tuning parameters
A=100	'Set maximum acceleration
V=1000000	'Set maximum velocity
MP	'Set Position Mode
C10	'Place a label
P=100000	'Set position
G	'Start motion
TWAIT	'Wait for move to complete
P=0	'Set position
G	'Start motion
TWAIT	'Wait for move to complete
GOTO10	'Loop back to label 10
END	'Obligatory END (never reached)

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.

KP=200	'Increase stiffness from default
KD=1000	'Increase dampening from default
F	'Activate new tuning parameters
A=100	'Set maximum acceleration
V=1000000	'Set maximum velocity
MP	'Set Position Mode
C10	'Place a label
P=100000	'Set position
G	'Start motion
GOSUB100	'Call wait subroutine
P=0	'Set position
G	'Start motion
GOSUB100	'Call wait subroutine
GOTO10	'Loop back to label 10
END	'Obligatory END (never reached)

APPENDIX E: EXAMPLE PROGRAMS

```
C100          `Subroutine 100
  □□□□□ □t    `Loop while trajectory in progress
  p=@PE        `Record position error into variable
  p=p*p        `Make absolute value
  IF p>10000    `Test for excessive load |@PE|>100
    PRINT("Excessive Load",#13)  `Print warning
  ENDIF        `End test
  L□□P        `Loop back to While during motion
  RET□RN       `Loop back to label 10
```

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.

```
KP=200        `Increase stiffness from default
KD=1200       `Increase dampening from default
F             `Activate new tuning parameters
AMPS=100      `Lower current limit to 10%
V=-10000      `Set maximum velocity
A=100         `Set maximum acceleration
MV           `Set Velocity Mode
G            `Start Motion
WHILE @PE>-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 Positoin Mode
V=20000       `Set higher maximum velocity
P=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

SmartMotors have encodes 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 mark.

```
KP=200        `Increase stiffness from default
KD=1000       `Increase dampening from default
```

APPENDIX E: EXAMPLE PROGRAMS

Set "D" equal to 20+ the encoder resolution of the particular SmartMotor. Typically, 17 and 23 sizes have 2000 counts per revolution, whereas 34 and larger have 4000. ServoStep typically has 8000.

```
F           'Activate new tuning parameters
A=100       'Set maximum acceleration
V=1000000   'Set maximum velocity
MP          'Set to Mode Position
D=20        'Move off in case on Index
G           'Start Motion
TWAIT      'Wait for motion to complete
i=I         'Clear Index flag by read
D=-2020     'Set 1+ rev, specific to motor
G           'Start Motion
WHILE Bi==0 'Wait for Index Flag to be true
LOOP        'Loop back to Wait
X           'Decelerate to stop
TWAIT      'Wait for motion to complete
P=I         '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.

```
KP=200      'Increase stiffness from default
KD=1000     'Increase dampening from default
F           'Activate new tuning parameters
A=100       'Set maximum acceleration
MV          'Set to Mode Velocity
d=10        'Analog Dead band, 1024 = Full Scale
o=512       'Offset to allow negative swings
m=40        'Multiplier for speed
w=10        'Time delay between reads
b=o         'Set default a value
C10         'Label to create infinite loop
  a=UCA-o   'Take analog reading of C and offset
  x=a-b     'Set x to determine change in input
  IF x>d    'Check if change beyond deadband
    V=b*m   'Multiplier for appropriate speed
    G       'Initiate new velocity
  ELSEIF x<-d 'Check if change beyond deadband
    V=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)
```

APPENDIX E: EXAMPLE PROGRAMS

LONG TERM VARIABLE STORAGE

SmartMotors are 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.

```
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.

```
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 Bi      `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.

```
KP=200      `Increase stiffness from default
```


APPENDIX E: EXAMPLE PROGRAMS

```
KD=1000      'Increase dampening from default
F            'Activate new tuning parameters
UAI          'Set I/O A to input (default)
A=100        'Set maximum acceleration
V=100000     'Set maximum velocity
P=1000000    'Set final position
            MP 'Set Position Mode
            G  'Start motion
            BtWHILE 'Loop while motion continues
        IF UAI==0 'If input is low
            IF V==100000 'Check V so change happens once
                V=150000 'Set new velocity
                G         'Initiate new velocity
            ENDIF
        ENDIF
    LOOP      'Loop back to WHILE
END          'Obligatory 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 250000.

```
KP=200      'Increase stiffness from default
KD=1000     'Increase dampening from default
F           'Activate new tuning parameters
UB=1        'Preset future output value
UBO         'Set I/O B to output, high
A=100       'Set maximum acceleration
V=1000000   'Set maximum velocity
P=1000000   'Set final position
MP          'Set Position Mode
G           'Start motion
WHILE @P<250000 'Loop while motion continues
    LOOP      'If input is low
        UB=0  'Check V so change happens once
        WAIT 400 'Set new velocity
        UB=1  'Initiate new velocity
    END       'Obligatory END
```

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

APPENDIX E: EXAMPLE PROGRAMS

volts.

```
KP=200          'Increase stiffness from default
KD=1000         'Increase dampening from default
F              'Activate new tuning parameters
A=100          'Set maximum acceleration
V=100000       'Set maximum velocity
P=1000000      'Set final position
MP             'Set Position Mode
G              'Start motion
WHILE Bt        'Loop while motion continues
  IF UJA<185    'If voltage is below 18.5 Volts
    OFF        'Turn motor off
  ENDIF
LOOP           'Loop back to WHILE
END            'Obligatory END
```

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

MEASURING COMMAND EXECUTION TIME

This routine will measure the time of a basic loop, then measure the time of a basic loop with an additional command (to be chosen for measure). It will finally subtract out the main loop time and report the actual execution time of the command inserted in the second loop. This program can be run during motion or servoing to show how the execution times vary. The PID loop and Trajectory generator have priority so when they are heavily loaded, program execution time is what gives. There is some margin of error. Even two identical loops can operate at slightly different rates due to how different segments of memory and their boundaries are treated. Note also that the **PID2** and **PID4** commands dramatically increase program execution speed.

```
t=0            'Calibration loop, to get basic loop time
c=CLK          'Store the running clock value
WHILE t<1000   'Loop 1,000 times
  t=t+1
LOOP
c=CLK-c        'c=execution time in sample periods per
               ' 1000 operations
c=c*1000       'Go from sample periods per 1k ops to
               ' sample periods per 1M ops.
d=c/4069       'Divide by sample time to get seconds
               ' per 1M operations.
t=0            'Now run the actual measuring loop
c=CLK
WHILE t<1000
  t=t+1
  'PUT COMMAND TO BE TIME-MEASURED HERE
LOOP
```

APPENDIX E: EXAMPLE PROGRAMS

```
c=CLK-c
c=c*1000
c=c/4069
c=c-d      'Subtract off the main loop time
Rc         'Report the actual time, in MicroSeconds
           ' the command takes to execute
END        'End program
```

CUSTOM PARSER WITH CHECKSUM

This is an example parser. It reopens the main communications port for data input only. The program takes over command interpretation. It is configured to take an ASCII command that can optionally be followed by up to four ASCII numbers separated by commas and finally a checksum value identified by a preceding "~". The program keeps a running checksum which is used for two purposes, for command differentiation and also to verify proper communications. The Checksum is multiplied by 2 before each additional character value is added. This provides a more reliably unique value. This program example leaves single letter variables available to the application, except for "p" and "q". Only single letter variables can be used as pointers (between brackets []). This parses very slowly in "Series 4" SmartMotors and is not recommended for time critical applications.

```
SADDR1     'Always a good idea to declare an address
ECHO_OFF   'Assure character Echo Mode is off
PID4       'Slow PID to 1kHz to up program speed
KP=100     'Set PID Proportional Gain, diff for PID4
KD=500     'Set PID Derivative Gain, diff for PID4
F          'Update Filter
A=1600     'Set default acc (extra high - PID4)
V=4000000  'Set default vel (4x because of PID4)
MP         'Ensure Mode-Position
mm=1       'Initialize Minus Flag to one
ss=0       'Initialize Main Checksum
tt=0       'Initialize Command-Only Checksum
nn=0       'Initialize Input Number to zero
yy=0       'Initialize Incoming Checksum Flag
zz=0       'Initialize Incoming Checksum to zero
al[0]=0    'Initialize Input Num array-uses aa space
al[1]=0    'Initialize Input Num array-uses bb space
al[2]=0    'Initialize Input Num array-uses cc space
al[3]=0    'Initialize Input Num array-uses dd space
p=0        'Initialize Input Number pointer
q=104      'Initialize Incoming Cmd Record Pointer
ab[q]=0    'Init first byte of Incoming Cmd Record
           ' 104 is the memory location of "aaa".
           ' 104 to 203 are available byte slots
```

APPENDIX E: EXAMPLE PROGRAMS

```
        ' from aaa to yyy.
OCHN(RS2,0,N,9600,1,8,D) 'Open comm port as Data
C10      'Place a label to create main loop
IF LEN   'Check if something in input buffer.
    vv=GETCHR 'Input character from buffer
    IF yy==0  'Advance Checksum
        ss=ss*2 'Shift checksum for more security
        ss=ss+vv 'Add input byte to checksum
        ab[q]=vv 'Record Incoming Command
        q=q+1   'Increment Record Incoming Cmd Ptr
        ab[q]=0 'Clear next position
    ENDIF
    IF vv<33   'Change to vv==13 for non SMI
        ' terminal testing.
        IF yy  'If Incoming Checksum Flag set
            zz=nn 'Store Incoming Checksum in zz
        ELSEIF nn
            al[p]=nn*mm 'Store Input Number into array
            nn=0       'Zero Input Number
        ENDIF
    SWITCH tt 'Identify specific incoming command
    CASE 1173 'MOVE
        GOSUB100
        BREAK
    CASE 1214 'STAT
        GOSUB200
        BREAK
    CASE 500 'END
        GOSUB300
        BREAK
    CASE 1239 'ZERO
        GOSUB400
        BREAK
    DEFAULT 'Anything Else
        GOSUB500
        BREAK
    ENDS
    ss=0      'Reset Main Checksum
    tt=0      'Reset Command-Only Checksum
    al[0]=0   'Zero Input Num array-uses aa space
    al[1]=0   'Zero Input Num array-uses bb space
    al[2]=0   'Zero Input Num array-uses cc space
    al[3]=0   'Zero Input Num array-uses dd space
    p=0       'Zero Input Number Pointer
```

Remove the first comment marks when you are ready to test the checksum feature.

The PRINT statements are useful in debugging, but would most likely be removed in the final application.

APPENDIX E: EXAMPLE PROGRAMS

```

nn=0      'Be sure Input Number is zero
mm=1      'Be sure mm sign flag is set to +
yy=0      'Init Incoming Checksum Flag to 0
zz=0      'Init Incoming Checksum to 0
q=104     'Init Incoming Cmd Record Pointer
ab[q]=0   'Init byte 1 Incomming Cmd Record
ELSEIF vv==44 'If there is a comma
  al[p]=nn*mm 'Store Input Number into array
  nn=0      'Zero Input Number
  p=p+1     'Increment Input Number Pointer
  mm=1      'Restore mm sign flag to positive
ELSEIF vv==126 'Tilde peceding Checksum value
  al[p]=nn*mm 'Store Input Number into array
              ' (if there is one)
  nn=0      'Zero Input Number
  p=p+1     'Increment Input Number Pointer
  mm=1      'Restore mm sign flag to positive
  yy=1      'Set Incoming Checksum flag
ELSE
  IF vv>65 'Look for ASCII text
    IF vv<123 'Build up Command here
      tt=ss 'Update Cmd Checksum for Parser
    ENDIF
  ELSEIF vv<58 'Look for ASCII numeric digits
    IF vv>44 'Build up number here
      IF vv==45 'Remember preceding - sign
        mm=-1 'Set flag to change number to -
      ELSE 'Build number
        nn=nn*10 'Shift previous number by 10
        uu=vv-48 'Cnvrt ASCII to val from 0-9
        nn=nn+uu 'Add to prev num to build it
      ENDIF
    ENDIF
  ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
GOTO10 'Loop back to main label forever

C100 'MOVE - can be followed by a number to
     ' move to and optionally followed by a
     ' new Velocity and Acceleration,
     ' seperated by commas.

PRINT("MOVING TO: ",al[0],#13)
PRINT("P",al[0]," V",al[1]," A",al[2],#13)

```

APPENDIX E: EXAMPLE PROGRAMS

```
PRINT("CCS: ",tt,#13)      'Print Command Checksum
                             ' (for development)
PRINT("TCS: ",ss,#13)      'Print Total Checksum
                             ' (for development)
PRINT("ICS: ",zz,#13)      'Print Incomming Checksum
                             ' (for development)
' IF ss==zz                'Engage this code to activate
                             ' Checksum Security
P=a1[0]                    'Set Position to first input number
IF a1[1]>0                  'If Vel is positive and non-zero
    V=a1[1]                'Set Velocity
ENDIF
IF a1[2]>0                  'If Acc is positive and non-zero
    A=a1[2]                'Set Acceleration
ENDIF
G
' ELSE                    'Engage this code to activate
                             ' Checksum Security
' PRINT("CHECKSUM ERROR, TCS:",ss," ICS:",zz,#13)
' q=104                   'Point to first char of Command
' PRINT("CMD: ")           'Prep to print errant command
' WHILE ab[q]              'Loop until entire cmd printed
'     PRINT(ab[q])
'     q=q+1
' LOOP
' PRINT(#13)
' ENDIF                   'Engage this code to activate
                             ' Checksum Security
RETURN

C200      'STAT
PRINT("STAT: ")
RS
PRINT(#13)
PRINT("CCS: ",tt,#13)      'Print Command Checksum
PRINT("TCS: ",ss,#13)      'Print Total Checksum
PRINT("ICS: ",zz,#13)      'Print Incomming Checksum
RETURN

C300      'END
PRINT("ENDING...",#13)
PRINT("CCS: ",tt,#13)      'Print Command Checksum
PRINT("TCS: ",ss,#13)      'Print Total Checksum
PRINT("ICS: ",zz,#13)      'Print Incomming Checksum
```

APPENDIX E: EXAMPLE PROGRAMS

```
OFF                                'Turn the servo off
OCHN(RS2,0,N,9600,1,8,C) 'Restore cmd interp.
END
RETURN

C400      'ZERO
PRINT("ZEROING",#13)
PRINT("CCS: ",tt,#13) 'Print Command Checksum
PRINT("TCS: ",ss,#13) 'Print Total Checksum
PRINT("ICS: ",zz,#13) 'Print Incoming Checksum
O=a1[0]          'Reset origin, if a1[0]
                  ' was not set, then
                  ' default zero

RETURN

C500      'Unrecognized or NEW commands
PRINT("ERROR",#13)
PRINT("CCS: ",tt,#13) 'Parser value calculator
                  ' for new commands.
                  ' This will show the
                  ' SWITCH value for any
                  ' new command

PRINT("TCS: ",ss,#13) 'Print Total Checksum
PRINT("ICS: ",zz,#13) 'Print Incoming Checksum

RETURN

END 'All programs must have an END, even if it
    ' is never reached
```

Using a command interpreter such as this is much slower than simply working with the built-in interpreter, nevertheless, it is sometimes necessary, particularly in retrofitting legacy systems. The SmartMotor can keep track of the actual checksum of incoming commands and data, and of text printed out, but it cannot know the checksum of data printed out because the Binary to ASCII conversion happens within the PRINT statement and the SmartMotor does not support a PRINT to an internal array. The SmartMotor standard interpreter has the "RCS" command to support checksum based communications security.

This page has been intentionally left blank.

APPENDIX F: F= COMMANDS

For purposes of efficiency, the SmartMotor utilizes a byte of memory (8 bits) for the setting of certain modes. The byte is accessed through the **F=** command. Because of its Binary nature, bits must be set and cleared with an understanding of how Binary works. If you want help understanding binary data, please refer to Appendix A, or for just the facts relevant to **F=**, refer to the end of this section.

Not all **F=** bit controlled modes relate to each other, but still, here is the total list for reference (in one place):

- F=1** Causes the motor to decelerate to a stop upon the trigger of a soft or hard limit, whereas the default action is to simply turn the amplifier off and allow the motor to free wheel (PLUS & ServoStep only).
- F=2** Reverses the shaft direction. All shaft motion commands and modes will cause motion in the opposite direction they would have otherwise (PLUS & ServoStep only).
- F=4** Redirects user program prints and report commands to channel 1 rather than the main communications port, channel 0.
- F=8** Causes the PID integral term to clear at the end of each trajectory. This eliminates what is referred to as "wind up" and can be useful in some applications.
- F=16** Causes CAM mode to use Relative Positioning (PLUS & ServoStep only).
- F=32** Causes faults to call subroutine C1 (PLUS & ServoStep only).
- F=64** Makes low transitions of port G trigger subroutine C2 (PLUS & ServoStep only).
- F=128** Resets the value of "**@P**", "**RP**" and external counter "**CTR**" to zero at modulo "**BASE**" plus dwell "**D**" in Relative Cam Mode (PLUS & ServoStep only).

The best way to load **F=** is with a shadow variable, like "**f**" for example. To set bit value 8, you could issue the command "**F=8**", but what you would also be doing is clearing all of the other bits of **F=**. A better way is to issue these commands:

```
f=f|8      'Set F=8 bit in the shadow variable
F=f       'Set the F= byte
```

The above example sets bit value 8 without disturbing the others.

Clearing bit 8 is a bit more convoluted. The bit value must first be subtracted from the maximum byte value of 255. For example $255 - 8 = 247$. By "anding" the shadow variable with 247, all bits will remain except for bit value 8. Here is how that would be done:

```
f=f&247    'Set F=32 bit in the shadow variable
```

*See Appendix B
to get a better
understanding of
Binary Data*

APPENDIX F: F= COMMANDS

F=f

'Set the modes into action

The above example "ands" everything except the bit value 8 with ones. If they were ones to begin with, they will remain ones, and if they were zeros to begin with, they will remain zeros. Furthermore, bit value 8 will be anded with zero and become zero, turning off the 8 position feature.