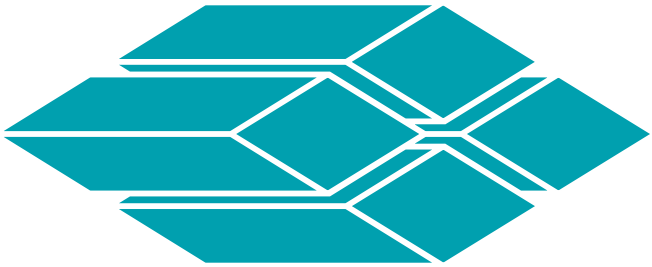
Animatics Institute Training Manual 2008



ANIMATICS®

Animatics Institute Training, October 2008

Contents

Contents

What You Will Need to Get Your SmartMotor™ Running

1. A SmartMotor™
2. A computer running MS Windows 95/98, 2000, NT, XP or VISTA.
3. A DC power supply for those SmartMotors that requre DC voltage.
4. A data cable to connect the SmartMotor to the computer's serial p o r t
5. or serial adapter.
6. Host level software to communicate with the SmartMotor

Optional SmartMotor™ cable (CBLSM1-10)

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 particualr motor, more Torque and Speed is available with higher voltage.

Optional PS24V8A or PS48V6A power supply

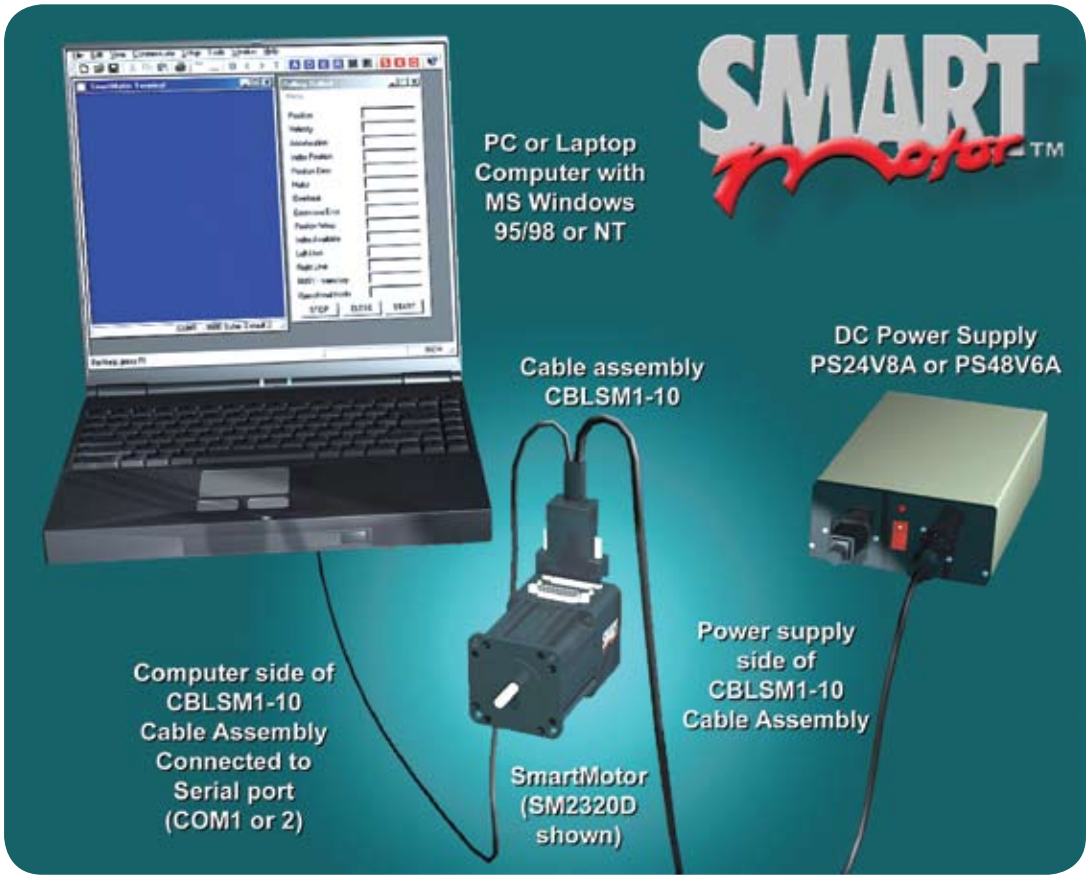
When relying on Torque/Speed curves, pay close attention to the voltage on which they are based. Also, special care must be taken when near the upper voltage limit or in vertical applications that can back-drive the SmartMotor. Gravity influenced applications can turn the SmartMotor into a generator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

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

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

Example of Connecting a SM2320D SmartMotor™ usinga 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 Backgound

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 digial 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

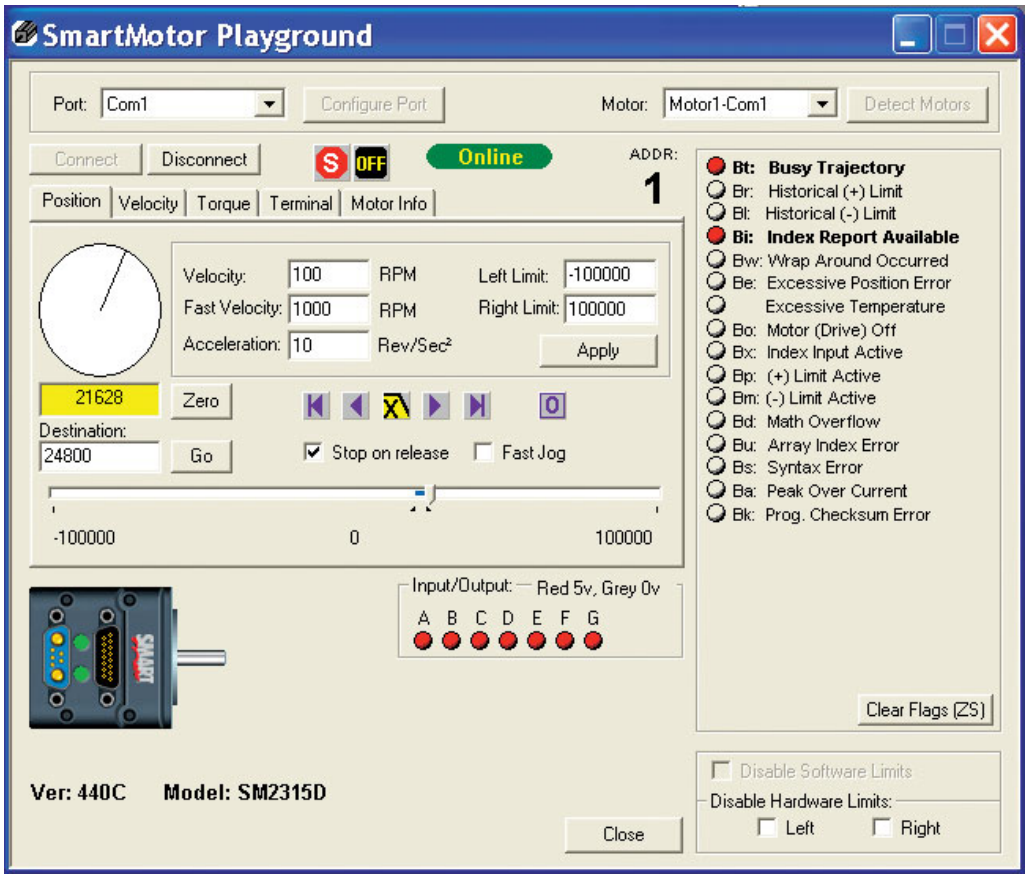
The SmartMotor Playground allows the user to immediately begin making motion without having to know anything about the programming.

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 corrected, or another computer substituted.

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, 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.

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

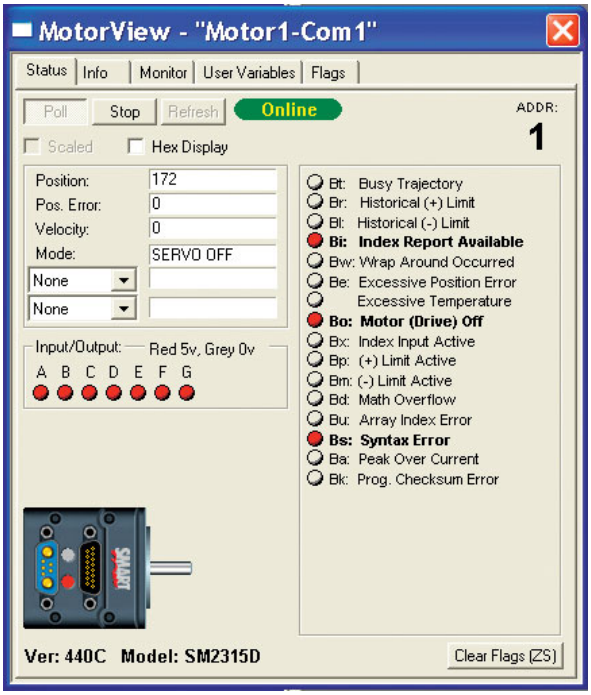
WARNING: The SmartMotor Playground changes both Software and Hardware Limit settings in the background which may cause unexpected results later. It is best to fully reset the motor upon exiting the Playground!

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.

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.

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 redefine 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
```

Acceleration, Velocity and Position fully describe a trapezoidal motion profile

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

Insure motors are properly mounted when under load

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.

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.

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 redefi nition 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:

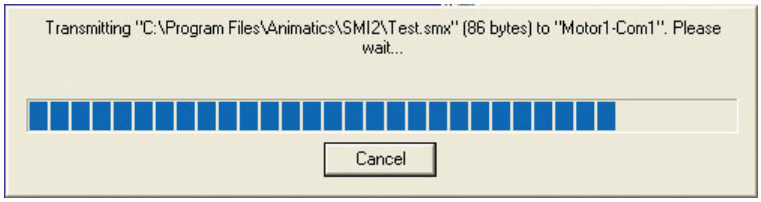
```
A=100          'Set buffered acceleration
V=1000000      'Set buffered velocity
P=300000       'Set buffered relative move
G              'Start Motion
TWAIT          'Wait for move to complete
P=0            'Set buffered move back to home
G              'Start Motion
END            'End program
```

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 fi le. If errors are found, make the necessary corrections and try again.

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.

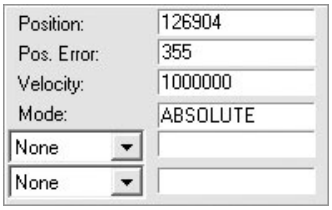
Since the program ends before the return move is finished, you can try running the program during a return move and learn a bit about how programs and motion work within the SmartMotor.

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.



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.

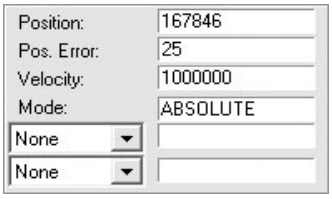
Looking at the Position Error and feeling the motor shaft will show that the motion, so far, is a bit sloppy. That is because the motor's PID Filter is tuned by default to be stable in almost any start-up environment. Try issuing the following commands in the Terminal and run the program again:

```
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":

Refer to the section on the PID fi lter for more information on Tuning.

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



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 surpised at the vast array of different paramters 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 amplifi er has several calculations worth of delay in the analog signaling, making them difficult to tune. By virtue of it's all-inclusive design, the SmartMotor requires no analog circuitry or associated noise immunity circuitry, and so the amplifi er portion conveys all of the responsiveness the controller can deliver.

SmartMotors Hardware and Control at a Glance

All Items here are important points of reference:
(please see appropriate sections in the manual or help files for more detail)

> Each SmartMotor is an Integrated Motion controller, Drive Amp and Motor.
> As a result, care should be taken with regards to hook-up, communications, and control.

Supplying Power to SmartMotors:

NEMA 17, 23, and 34 frame sizes:

All SmartMotors in frame sizes from NEMA 17 to NEMA 34 run off of 24-48VDC. Under no circumstances should they be allowed to run off of any higher voltages. Lower voltages could cause a brownout shutdown of the CPU or what would appear as a down power reset under sudden load changes. If power is reversed on any NEMA 17-34 frame size SmartMotors, immediate damage will occur and the SmartMotor will no longer operate.

> Note: During hard fast decelerations, a SmartMotor can pull up supply voltages to the point of damage if a shunt resistor pack is not used.

CPU, I/O, and Communications Power restrictions:

CPU Power:
All SmartMotors have an internal 5VDC Power supply to run the internal CPU. This supply can be easily damaged if an external voltage source of a higher potential is applied. Do not exceed 5VDC on and I/O pin or 5VDC pin on any SmartMotor.

I/O Restrictions and limitations:
Each on-board I/O pin has a minimum amount of protection consisting of a 100-Ohm Current limit resistor and a 5.6VDC Zener diode. Each I/O pin also has a 5Kohm pull-up resistor.

When assigned as outputs, they act as a push-pull amplifier that drives hard to either the positive or negative 5VDC rail. This means they are not open collector I/O pins. Each I/O Pin can sink up to 12mA and source up to 4mA. Exceeding this could result in damage to the I/O port.

Communications:
Each Motor has a 2 wire RS-232 port. This port meets IEEE standards with full +/- 12VDC potential on the transmit line. Proper serial ground signal referencing and shielding techniques should be used.

Under no circumstances should the shield of a cable be used for the RS-232 ground reference. This could result in noise or corrupt data as well as ground loops that could damage the serial port chip set.

Each SmartMotor boots up default to the ECHO_OFF state. This means that nothing received is transmitted or echoed back out. This is important to remember in serial "daisy-chain" set-ups. They also boot-up defaulted to base address zero meaning they will listen and respond to any incoming valid SmartMotor commands.

Programs, Variables and Modes of operation summary:

Programs:
All SmartMotors will run any valid program stored in memory upon boot-up by default. The only way to prevent this is to add the RUN? command at the top of the program or send any communications to the SmartMotor within the first 500msec.

Any PRINT statement containing long text strings should be reviewed carefully. Any text strings that appear as valid commands could cause other motors in the "Addressed" state to respond. using GOTO and multiple GOSUB or GOTO calls from multiple sources could cause unexpected results. This can cause the program stack pointer to become corrupt. At that point the CPU will not know where to go in the program. (Multiple sources means from the RS-232 port, RS-485 port and/or Program.)

System and User Variables:
Each SmartMotor runs and operates from volatile RAM. This means that upon loss of power, all variables lose their present value and will be zero at the next power-up. All tuning parameters will return to factory defaults. Motor addressing will re-set to zero. Clock, Counter and Position registers also reset to zero.

The only way to insure desired values upon power-up is to store them in hard code in a downloaded program. All SmartMotor variables are pre-defined as shown in the manual and help files. No user defined variables are available. All variables are global within a single SmartMotor. This means that any update or change to a variable will be seen in any subsequently called subroutines. If it is desired to have local variables to a given subroutine, they should be assigned the needed value within that subroutine as needed.

Modes of Operation:
Mode Set (MS), Mode Follow (MF1, MF2, MF4), and Torque Mode (MT) are the only modes that do not require a "G" command to initiate. This means the mode will be immediately initiated upon receiving the associated Mode commands. This could result in immediate shaft movement. Be careful when initiating these Modes of operation.

Mode Follow Ratio (MFR) and CAM Mode (MC, MC2, MC4, MC8) both require a "G" to become fully effective, however, they also require Mode Follow commands in their set-up. This means that during the set-up process, motor shaft movement could result.

All other modes require a "G" to initiate or to change velocities and/or accelerations.

Port G on all motors defaults to "G –sync" which means if it is grounded, it will be no different that having received a "G" command. If Port G is grounded prior to power-up, the processor will see this as a valid "G" command upon boot-up. Depending on code, this could cause unexpected motor shaft movement.

The "D" register is the relative commanded move register. Any non-zero values of D could result in unexpected moves upon any valid "G" command be it from the "G" command or the grounding of port G.

SmartMotors default to position mode on boot-up. They only require a non-zero velocity and accel a long with a non-zero D value or P value other than present position to initiate a move.

Motor response to Loss of power and Faults:

Loss of Power:
All SmartMotors will coast or "free-wheel" on loss of power. This is because there is no control power to prevent it from occurring. You may notice what appears as a dynamic braking characteristic upon loss of power. This occurs to some extent due to internal capacitance in the drive amp circuit or external power supply circuit resistance or capacitance. To insure fast fail-safe stopping for safety reasons, some mechanical brake type of mechanisms should be used.

Hardware Protection Faults:

In all firmware revisions prior to V4.76, Motor hardware protection faults (Over Current and over Temp) result in "free-wheel" of motor shaft. This is to prevent further damage to the motor or drive amp. Versions 4.76 and later implement dynamic braking on error.

Software protection faults:
Limit switch inputs and Position error limits are both "software" protection faults. This means they are not firmware unchangeable. The effects of Limit switches and Position error can be changed via valid software commands or set-up parameters. Position error is predicated by a value set by the user and can drastically effect motor response under varying load conditions and tuning. Limit switches can be set up to cause the motor to servo in place instead of free wheel. Refer to specific firmware addendums for various limit switch options and capabilities.

Motor Response with respect to Motor Tuning:
Care should be taken with any closed loop servo with regards to motor tuning. Improper tuning may cause undesired effects ranging from excessive noise and vibration to mechanical damage to a machine or overheating of the motor or drive amp. Improper tuning can also lead to repeated or undesired amounts of over current or position errors. Proper tuning can make or break a successful application

Motor Torque Limits: {AMPS command and T (Torque) command}:
Motor T (torque) command is only for use in Mode Torque (MT). It has no effect on motor operation outside of Mode Torque.

The AMPS command has effect over all other modes of operation. It limits absolute maximum power available from the drive amp to the motor windings as a function of percent duty cycle of PWM (Pulse Width Modulation). The AMPS command should be used when it is desired to limit motor torque to a sensitive or torque input limited load. IT may also be used to reduce the chance of reaching peak over current errors on high acceleration applications.

Error Handling, Motor Status Bits and Internal conditions:
SmartMotors have a 16 Bit status word that contain interrupt registers triggered by selected events. These events include Position Errors reached, Over Current reached, Limit switch conditions, Syntax errors and so on. In addition, in the newer motors, Bus Voltage, Drive Current, and Motor Temperature are also available. By proper use of these status bits very simple and very flexible error handling can be achieved. Motors can be made to respond under varying load conditions in different ways and recover from any given software or hardware fault in a controlled manner.

Please review this in detail in the manual and help files.

SmartMotor Communications at a glance

Notes on Boot-Up sequence of SmartMotors with regards to communications:

SmartMotors default to 9600 Baud, no parity 8 data bits and 1 stop bit. (9600,N,8,1). All SmartMotors boot up in ECHO_OFF mode with global address zero. This means they will respond to globally addressed commands, i.e. commands proceeded by dec128 or hex80.

With in the first 500msec's or so of power up, if they have not received any serial communications, they will begin executing code previously downloaded to them from the top down.

If the Code begins with RUN? , execution will stop at that line until a "RUN" is received via RS-232 serial port.

Multiple motors on a communications line

If more than one motor is to be placed on a communications line, they must be set up properly to avoid communications errors.

RS-232

If RS-232 is used from a host PC or other RS-232 compatible device, all motors must be in ECHO mode.
While in ECHO mode, all data reaching a motor's received pin will be "echoed" back out it's transmit port.
Since RS-232 serial lines must be daisy-chained together, the motors must be in ECHO mode to work properly.

RS-485

If RS-485 is used, all motors must be in ECHO-OFF mode. RS-485 is a parallel communications network. If any motor was to echo out commands received, it would cause all motors or any other devices on the network to get hit with the same data.

> **Note:** SmartMotors use 2 wire RS-485 standards. This means line biasing determines whether or not the motor is in transmit mode or receive mode at the hardware level. To insure motors do not hang up in the transmit mode, there must be a minimum of a 200mVolt differential between RS-485 A and B channels.

This is easily achieved by placing a pull down resistor of approximately 500 Ohms from the B channel to ground somewhere on the RS-485 network.. All SmartMotors have a 5Kohm pull-up resistor on both A and B channels already. The 500 Ohm resistor will provide the enough biasing needed to make the hardware default to the receive state. If there are long distances between motors, it may be necessary to provide a resistance across channels A and B. A 200 Ohm resistor wired from A to B at the remote end of the RS-485 line should provide ample voltage drop for needed biasing.

If the above electrical rules are not applied, communications cannot be guaranteed to work.

> **Note:** Resistor values above are approximate. The actual values needed may vary depending on communications line impedance due to things such as cable length.

Pre-Addressed Motors on Boot-Up

If it is desired for the motor to have a non-zero address on boot-up, the motor must have a program downloaded to it with the set address command at or near the top of the program.

Example:

SADDR1 will set the motor's address to 1.

Or:

ADDR=1 will set the motor's address to 1.

Note: "ADDR=" syntax not available in v4.40 series firmware."

where ADDR= is matching blue from above.

The motor address command uses integer numbers 1-100, however to specifically address a particular SmartMotor, you must precede the desired command with decimal or hex equivalent addresses. If a motor's program begins with SADDR1, then a command specifically meant for that motor must be preceded by dec129 or hex81for example.

Example 1: `PRINT(#131,"A=1200",#13)`

The above code sends out dec131 immediately followed by A=1000 immediately followed by a carriage return.

Example 2: `PRINT(#131,"A=1200 ")`

would give the same result because a space was included after A=1200.

Global addressing:

As mentioned at the beginning, all motors without an address respond to any command preceded by dec128 or hex80. This is also true for any motor with an address that is not in "sleep" mode.

If a motor is in Sleep mode, it will only start listening again if the WAKE command preceded by its address is sent to that motor.

Example: If hex85SLEEP is transmitted, motor 5 will go into "sleep mode."

If hex80x=123 is transmitted, every motor on the serial line will have

variable x updated to 123 except motor 5.

If hex85WAKE is transmitted, motor 5 will "wake up" and will respond once again to commands.

Note: WAKE and SLEEP do not affect the ECHO state.

Addressed or De-Addressed state:

It is important to understand **addressed** or **de-addressed** states of SmartMotors. These states determine whether or not a SmartMotor will respond to commands.

Lets assume for example that we have 5 motors on a communications network.. All of them have programs downloaded with addresses 1-5 respectively via the SADDR command.

On Boot-up all are ready to listen. They will respond to either a globally addressed command or a specific motor will respond to a specifically addressed command.

Once a specific address is sent out on the line, that motor will be in the "**addressed**" state and All other motors will be in the "**de-addressed**" state.

What this means is that from that point on, any command sent out to the motors without an address proceeding it, will be acted upon only by the motor in the addressed state. All other motors will basically ignore anything received.

By sending out a command preceded by the global address (hex80 or dec128), all motors will be placed into the "addressed state and will remain in that state until another specific address is transmitted. Under the above case of 5 motors addressed 1-5 respectively, if a hex89 for example or any other address outside of hex80-hex85 is sent, all motors would become "de-addressed". No motor would respond to any command until an address within the range of motors on the line is received.

This does come in handy if other Non-SmartMotor devices are on the same network.

Example:

Suppose you have the same 5 SmartMotors on line with a barcode reader or temperature controller. If you wanted to send set-up parameters to these devices but wanted to insure the SmartMotors would ignore the set-up parameters (since they are more than likely not even valid SmartMotor syntax), you could "de-address" all SmartMotors by sending out an address outside the bounds of any motor on the line.

Note: Most RS-485 devices operate this way.

Any hex value>79 or decimal ASCII value > 127 will be seen as a control or address character by SmartMotors. Care should be taken when mixing SmartMotors with other devices on an RS-485 bus.

Addressing SmartMotors from a Host PC or other Serial Device

Note: The following only applies to an RS-232 serial daisy chain where the motors do not have programs downloaded with addresses in them. It will not work on an RS-485 network. Motors must be pre-addresses in downloaded programs for an RS-485 networks to work at all.

It is important to realize the boot-up state of SmartMotors from the first section to understand this sequence. Please review it if you have not already done so.

Since SmartMotors without addresses default to address zero (hex80 or dec128), a sequence of commands must be issued in proper order to achieve addressing of the SmartMotors.

> **SmartMotors without programs downloaded into them will not retain addresses from this procedure upon loss and return of power!**

The following is an example sequence of addressing 3 SmartMotors **from the SMI software terminal screen!**

Assumptions are as follow:

1. Host PC is set up for 9600 Baud,N,8,1 since this is the power-up default for SmartMotors.
2. Three SmartMotors are wired in serial daisy chain with Tx of Host PC wired to Motor-1 Rx, Motor-1 Tx wired to Motor-2 Rx, Motor-2 Tx wired to Motor-3 Rx, Motor-3 Rx wired to Host PC Rx. (Tx is RS-232 transmit, Rx is RS-232 Receive)

0ECHO_OFF	Places all motors in echo off
0SADDR1	Set first motor to address 1
1ECHO	Set it to echo mode so the next motor will be able to receive commands
1SLEEP	Set it to sleep mode so it will not act upon following commands
0SADDR2	Set next motor to address 2 and repeat sequence
2ECHO	
2SLEEP	
0SADDR3	
3ECHO	
3SLEEP	
1WAKE	Set all motors to wake status.
2WAKE	
3WAKE	

Note: SMI Software **automatically replaces** leading numbers in commands with a decimal offset of 128.
In other words, 0ECHO_OFF resulted in "(dec128)ECHO_OFF" being transmitted.

This is the equivalent from any other software source:

```
(dec128)ECHO_OFF
(dec128)SADDR1
(dec129)ECHO
(dec129)SLEEP
(dec128)SADDR2
(dec130)ECHO
(dec130)SLEEP
(dec128)SADDR3
(dec131)ECHO
(dec131)SLEEP
(dec129)WAKE
(dec130)WAKE
(dec131)WAKE
```

Note: (hex80-83) is the same as (dec128-131), All lines must be terminated with either a carriage return (dec13) or space character.

SmartMotor Program Flow at a Glance

Introduction to the SmartMotor Language

SmartMotors use a simple form of code similar to BASIC.

Various commands include means of creating continuous loops, jumping to different locations on given conditions, and doing general math functions.

- All commands begin with Capitol letters.
- All variables are pre-assigned and are lower case only.
- Each program must contain at least one occurrence of the **END** statement.
- Each subroutine call must have a label with a **RETURN** statement somewhere below it.
- All programs will automatically execute upon power-up if nothing is received on the serial port within the first ½ second or so.

If you add the statement: **RUN?** to a program, it will not execute any code past the RUN? until the SmartMotor receives a **RUN** command via the serial port.

This means you can add **RUN**? to the very top of a program to prevent any code from executing on power-up.

You can however, at any time, tell the motor to run subroutines via serial port even if the stored program is not running.

Like BASIC, you can print using the **PRINT** command to print to the screen.

Sample Program:

```
RUN?
PRINT("Hello World",#13)
END
```

This is a simple program that will not run on power up. You can run it by typing in the **RUN** command at the terminal screen.
It will then simply print Hello World to the terminal screen and then end.

To put comments in the program you precede your line of text with an apostrophe:

```
RUN?                        ' the program stops at this line until a "RUN" is received.
PRINT("Hello World",#13)    ' #13 is a carriage return
END                         ' the end statement marks the end of the program.
```

If you wanted to have the above example run on power-up, just delete **RUN?** from the program.

General Expressions

There are means of assigning variables different values as well as checking values and comparing them. These are the "operators" for doing that:

Assigning values

= (Equal sign) Assignment operator

Note: No spaces are used on either side of the equals sing.

```
a=5          'assign to a the value of 5
c=@P         'assign to c the present motor position
V=500000     'set velocity to 500000
```

Math operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division

Note: No spaces are used on either side of any math operators.

```
a=b+c        ' set a to b+c
P=a*1000     ' set commanded position to a multiplied by 100
```

Comparison Operators:

==	Equals (use two equal signs in a row)
=	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
&	Bit wise AND (bit wise means binary number comparison)
|	Bit wise OR (these operators are good for handling I/O)

Note: No spaces are used on either side of any comparison operators. Examples of some of these will be shown on the following pages.

Flow Control Commands

IF, ENDIF

If you want a portion of code to execute only once based on a certain condition then use the "IF" statement.

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

```
IF a==1      ' If the variable a is equal to 1
  b=20       ' Set the variable b to twenty
ENDIF        ' End the IF code block and continue on to the next line of code
```

In the above example,. b will be set to 20 only if a is equal to 1

ELSE, ELSEIF

The 'ELSE' and 'ELSEIF' commands can be used to add flexibility to the 'IF' statement. What if you wanted to execute different code for each possible state of variable 'a'?

You could write the program as follows:

```
IF a==1      ' If the variable "a" is equal to 1
 b=20        ' Set the variable "b" to twenty
ELSEIF a==2  ' Else if the variable "a" is equal to two
 c=30        'Set "c" to thirty
ELSE         'Else If a is not equal to one or two
 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 comparison check statement in the structure before the 'ENDIF'.

You can also have 'IF' structures inside 'IF' structures. That is called "nesting" and there is no practical limit to the number of "IF" structures you can nest within one another.

SWITCH, CASE, DEFAULT, BREAK, ENDS

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

The following code will accomplish the same thing as the previous ELSE, ELSEIF example.

```
SWITCH a      ' look at the variable "a"

 CASE 1       ' in the case that it equals 1
   b=20       ' Set "b" to twenty
 BREAK

 CASE 2       ' in the case that it equals 2
   c=30       ' Set "c" to thirty
 BREAK

 DEFAULT      ' If a is not equal to 1, or 2
   d=1        ' Set "d" to one
 BREAK

ENDS          'End SWITCH
```

Like a rotary switch directs electricity, the 'SWITCH' structure directs the flow of the program. The 'BREAK' statement then makes the processor jump to the code following the associated 'ENDS' command.

The **ENDS** command means "end switch" and marks the end of the SWITCH routine.

The **DEFAULT** command covers any condition not found by the **CASE** commands.

The default command is optional.

WHILE, LOOP

The most basic looping function is a 'WHILE'. The word 'WHILE' is followed by an expression that determines whether the code between the 'WHILE' and it's associated 'LOOP' will execute or be passed over.

- While the expression is true, the code will execute.
- An expression is "true" when it's value is non-zero.
- If the expression results in a zero value then it is false.

Example 1

```
WHILE 1==1   ' 1 is always true
      UA=1   ' Set Port A output to one (5 volts)
      UB=0   ' Set Port B output to zero (zero volts)
             ' This code serves no practical purpose. It is for example only.
LOOP         ' this code will loop forever
```

Example 2

```
a=1          ' Initialize variable a
WHILE a!=0   ' while a does not equal zero
      a=0    ' Set a to zero
             ' This code serves no practical purpose. It is for example only.
LOOP         ' This never loops back
```

Example 3

```
a=0          ' Initialize variable a
WHILE a<10   ' a starts less
      a=a+1  ' add one to a each time you go through the loop
LOOP         ' loop back to the WHILE until a is 10 or greater
```

The task or tasks within the "WHILE" loop will execute as long as the function remains true.

The 'BREAK' command in conjunction with the IF structure can be used to break out of a 'WHILE' loop. It is very handy for error checking and Input triggers.

```
a=0
WHILE a<10
      a=a+1        ' add one to "a" each time you go through the loop
      IF b==1      ' if b is equal to 1
            BREAK  ' Break out of while loop immediately
      ENDIF
LOOP               ' loop back to the WHILE until a is 10 or greater
```

NOTE: "b" could have been changed via serial port update.

Subroutines, Flow routing, and Timing

GOTO# go to a labeled line in the program

Note: Line Labels are always the capitol letter C followed by a positive integer between 0 and 999 with no spaces in between.

Example:

```
WHILE 1             ' short for WHILE 1==1
      a=@P          ' Assign present motor position to "a"
      IF a>3175
            GOTO1   ' Go to a line labeled C1
      ENDIF
LOOP                ' loop back to the WHILE

C1                  ' C1 is a label, it could be any number 0-999 after the C
PRINT("Motor position exceeded 3175 encoder counts",#13)
```

GOSUB# go to a subroutine label in a program

Subroutines are a collection of code that will run when ever called via a **GOSUB** call.

```
WHILE 1                  ' short for WHILE 1==1
  IF UCI==0              ' If Port C input equals zero
    GOSUB5               ' run subroutine C1
    WHILE UCI==0 LOOP    ' prevent multiple calls to C5
  ENDIF
  IF UCI==1              ' If Port C input equals 1
    GOSUB6               ' run subroutine C2
    WHILE UC1==1 LOOP    ' prevent multiple calls to C6
  ENDIF
LOOP                     ' loop forever

C5
  PRINT("Port C is at zero volts",#13)
RETURN

C6
  PRINT("Port C is at five volts",#13)
RETURN
```

STACK Resets the GOSUB return stack

In the **CPU**, the **STACK** is like a register that holds a stack of numbers telling the **CPU** where to go next in the program.

In the event you direct program flow out of a subroutine without executing the associated **RETURN** command, you could corrupt the stack.

Example: (of really bad programming)

```
C100
            GOSUB105    ' run subroutine C105
            GOTO100
C105
            GOTO110     ' jump out of subroutine 105 and to line 110
RETURN
C110
GOTO100
```

In the above example a **GOTO** was used to get out of a subroutine.

The **CPU** would be expecting to see a **RETURN** and then return to a label but the GOTO defeated it. Subsequent calls to the same subroutine by the GOTO100 line at the bottom would eventually cause what is termed as a "stack overflow" because each time the subroutine is called, the stack register remembers the line where the subroutine was started and adds it to the stack. Eventually, the stack will get too full and the CPU will get a stack overflow error.

Example: (of not much better programming, but shows the STACK command usage)

```
C100
      STACK                 ' reset the stack
      GOSUB105              ' run subroutine C105
      GOTO100
      C105
            GOTO110     ' jump out of subroutine 105 and to line 110
      RETURN
      C110
      GOTO100C100
      STACK                 ' reset the stack
      GOSUB105              ' run subroutine C105
      GOTO100
      C105
            GOTO110     ' jump out of subroutine 105 and to line 110
      RETURN
      C110
      GOTO100
```

By Issuing the command Stack, the program clears the stack to zero and will sequentially run down in the code from that point on following directive of what ever comes up.

Note: Although sometimes necessary, Programs should be written in such a way as to never need a STACK command.

WAIT pauses program execution for a given amount of time

```
WAIT=2000          ' Wait 2000 sample periods
WAIT=a+2000        ' Wait a+2000 sample periods
```

A sample period is about 1/4000 of a second depending on the model motor, so the above code would cause about ½ second pause or more.

The following code would be the same as "WAIT=40000", only it will allow you to execute code during the wait if you place it between the 'WHILE' and the 'LOOP'.

```
CLK=0                          ' sets system clock to 0
WHILE CLK<40000                ' while less than about 10 second have passed
            IF UAI==0          ' Monitor input port A to see if it goes low
                  GOSUB100     ' If it does, jump to subroutine 100
            ENDIF
LOOP                           'Loop back
```

The above code example will check if input 'A' for about 10 seconds to see if it ever goes low. After that, it continues on to any code that follows the WHILE LOOP.

TWAIT Trajectory WAIT

If the motor is busy moving to a commanded position, TWAIT will stop program flow until it has finished the move.

In other words, any time the motor is actively trying to reach a commanded position in any mode other than Torque Mode, it is actively pursuing a trajectory.

Example:

```
MP           ' Set Mode to position mode
P=1000       ' Set commanded position to 1000
A=1000       ' Set Acceleration to 1000
V=100000     ' Set velocity to 100000
G            ' Go (Start moving)
TWAIT        ' Wait here until the move is complete
```

SmartMotor Modes of Operation and Motion Control Commands

SmartMotors can be operated in many different modes. You can switch to and from almost any mode freely at any time. Consult the command reference for more details on any given command.

he following is an overview of each mode of operation:

MP **Mode Position** (Position Mode)

Absolute Mode

Position mode is the default power-up mode of operation for the SmartMotor.
In Position mode, the SmartMotor operates on Absolute position commands in encoder counts

Minimum Requirements for a move to occur in position mode:

• Initiate the Mode via the	MP command (if not already in Position mode)	
• Non-zero value of Velocity	V=###	set velocity equal to ###
• Non-zero value of Acceleration	A=###	set acceleration equal to ###
• Absolute commanded position	P=###	set commanded position to ###
• Go command to initiate the move	G	Start move immediately

Note: Commanded position must be different than present position to cause a move.

If Acceleration or Velocity are at zero, the Motor will not move.

Example 1: Basic Absolute Move

```
MP          ' set motor to position mode (may be required of currently in another mode)
V=100000    ' set velocity to 100000
A=1000      ' set acceleration to 1000
P=20000     ' set commended absolute position to 20000
G           ' Go (Start moving)
```

In the above example, the motor will start moving upon seeing the G command and will stop at an absolute position of 20000 encoder counts.

Example 2: Two Moves with a delay in between

```
O=0         ' set current position to zero
MP          ' set motor to position mode (may be required of currently in another mode)
V=100000    ' set velocity to 100000
A=1000           ' set acceleration to 1000
P=20000     ' set commended absolute position to 20000
G           ' Go (Start moving)
TWAIT            ' wait here until the motor has reached 20000
WAIT=4000   ' wait about 1 second
P=-500           ' Set commanded position of -500
G          'start moving to new commanded position
```

Note: motor move will be made at previously commanded speed and acceleration.

Example 3: Change in commanded speed and acceleration on the fly

```
O=0          ' set current position to zero
MP           ' set motor to position mode (may be required of currently in another mode)
V=100000     ' set velocity to 100000
A=1000       ' set acceleration to 1000
P=1000000    ' set commended absolute position to 1000000
G            ' Go (Start moving)
WAIT=8000    ' wait about 2 seconds
V=800000     ' set new velocity of 800000
A=500        ' set new acceleration of 500
G            ' initiate change in speed and acceleration
```

Example 4: Change in point of origin between moves

```
O=0          ' set current position to zero
MP           ' set motor to position mode (may be required of currently in another mode)
V=100000     ' set velocity to 100000
A=1000            ' set acceleration to 1000
P=2000            ' set commended absolute position to 2000
G            ' Go (Start moving)
TWAIT        ' wait until move is complete
O=0          ' set current position to zero|

WAIT=8000    ' wait about 2 seconds
P=2000            ' set commended absolute position to 2000
G            ' Go (Start moving)
TWAIT        ' wait until move is complete
```

Note: motor has moved a total of 4000 counts, but it's current position is only 2000 because it's position was reset to zero in between moves. The Origin command "O" can be set to any number.

Relative Mode

In Relative Mode the SmartMotor moves relative to where it is at any time by the use of the D (Distance) command. Minimum requirements are the same as in absolute mode.

Example:

```
MP           ' set motor to position mode (may be required of currently in another mode)
V=100000     ' set velocity to 100000
A=1000       ' set acceleration to 1000
D=2000       ' set commended relative position move to 2000
G            ' Go (Start moving 2000 counts)
TWAIT        ' wait until move is complete
G            ' Go (move 2000 counts again)
TWAIT        ' wait until move is complete
G            ' Go (One more time)
```

The motor will have moved 3 2000 count moves or a total of 6000 counts upon completion.

MV Mode Velocity (Velocity Position Mode)

Velocity mode allows the SmartMotor to run at a constant commanded speed. SmartMotors close the speed loop on position, not encoder counts per unit time. As a result, moving to and from Position mode to velocity mode is very simple.

Minimum Requirements for a move to occur in velocity mode move:

- Initiate the Mode via the MV command (if not already in velocity mode)
- Non-zero value of Velocity V=### set velocity equal to ###
- Non-zero value of Acceleration A=### set acceleration equal to ###
- Go command to initiate the move G Start move immediately

Example 1: Basic Constant Velocity Move

```
MV          ' set motor to Velocity mode
V=100000    ' set velocity to 100000
A=1000           ' set acceleration to 1000
G           ' Go (Start moving)
```

In the above example, the motor will start moving upon seeing the G command,. It will accelerate up to a velocity of 100000 at a rate or 1000 samples /sec/sec. It will then stay at that speed until told to do otherwise.

Example 2: Change in commanded speed and acceleration on the fly

```
O=0         ' set current position to zero
MV          ' set motor to Velocity mode
V=100000    ' set velocity to 100000
A=1000      ' set acceleration to 1000
G           ' Go (Start moving)
WAIT=8000   ' wait about 2 seconds

V=800000    ' set new velocity of 800000
A=500       ' set new acceleration of 500
G           ' initiate change in speed and acceleration
```

In this example, the motor's move parameters will be changed about 2 seconds after the initial commanded move was made.

MT **Mode Torque** (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 in a 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.

Positive numbers apply a clockwise torque. Negative numbers apply a counter-clockwise torque.

The Default value for T is zero. Keep in mind, speed is proportional to counter torque or load on the shaft when in torque mode. The larger the load, the slower the motor will turn for a given torque value.

Minimum Requirements for a Torque Mote to operate:

Initiate the mode with the MT command

Note: Torque mode is an immediate response mode. No "G" is required for the motor to go into torque mode. Upon receiving MT, the motor will immediately go into torque mode at the present value of T.

Example 1: Basic Constant Torque Move

```
T=200        ' set torque to 200
MT           ' set motor to Torque Mode
```

Note: Again as stated above, The motor will immediately start moving upon receiving the MT command. The example shows T being set prior to MT so as to have a known commanded torque ahead of issuing MT.

Example 2: Changing from Velocity Mode to Torque Mode dynamically

```
MV           ' set motor to Velocity mode
V=100000     ' set velocity to 100000
A=1000       ' set acceleration to 1000
G            ' Go (Start moving)
WAIT=8000    ' wait about 2 seconds
T=200        ' set torque to 200
MT           ' set motor to Torque Mode
WAIT=8000    ' wait about 2 seconds
OFF          ' turn the motor off
```

In the above example, about 2 seconds after going in to velocity mode, the motor is switched to torque mode. Then 2 seconds later, the motor is turned off.

Mode Follow (Electronic Gearing)

MF1, MF2, MF4, MF0, RCTR

Mode-Follow allows an external encoder to be used as a command reference to position. It is an extension of Position-mode to some degree in that it operated on a continuously updating commanded position in absolute position mode.

Minimum Requirements for a move to occur in Mode Follow:

- Phase A and B of external encoder must be wired into ports A and B.
- The External encode must be powered up and be 5VTTL compatible.
- **MF1, MF2,** or **MF4** must be issued

Note: Single ended or differential encoders may be used wapping phases swaps direction that motor will turn for a given external encoder direction of rotation.

As in Torque mode, MF1, MF2, and MF4 are immediate in response. No G is required to make the motor go into Mode Follow.

A G can, however re-initiate mode-follow after an error such as over current or position error.

Definition:

MF1	read external encoders at 1:1 of base encoder counts.
MF2	read external encoders at 2:1 of base encoder counts.
MF4	read external encoders at 4:1 of base encoder counts.
MF0	Zero the external encoder count read register.
RCTR	Report external encoder-count register value.

Note: SmartMotors default to reading external encoders at full quadrature (MF4). If an external encode is wired up and you read the external counter register (**RCTR** to read counter), it will display counts dynamically without issuing MF4. The motor will not be in Mode follow, but the counter will be active.

Example:

Say you have a free-standing encoder with 500 lines or pulses per revolution on each phase A or B. This means that for each revolution of the encoder shaft, the respective phase will pulse high and back low 500 times:

If **MF1 i**s issued, the SmartMotor will move 500 internal encoder counts for each single revolution of the 5000 line external encoder.

If **MF2** is issued, the SmartMotor will move 1000 internal encoder counts for each single revolution of the 5000 line external encoder.

If **MF4** is issued, the SmartMotor will move 4000 internal encoder counts for each single revolution of the 5000 line external encoder.

MF4 makes use of what is termed as "full quadrature" of a standard quadrature output encoder. As a result, MF4 alows maximum resolution tracking of the external encoder.

All examples that follow will use MF4 for simplicity.

Example 1: Make a SmartMotor follow an external encoder 1:1 at full quadrature:

```
MF4          ' That's it. Nothing else to do.
```

MFR, MFMUL, MFDIV (Mode Follow with Ratio)

Mode follow with ratio allows the user to define the number of internal counts the motor will move for any given external encode count change.

Minimum Requirements for a move to occur in Mode Follow with ratio:

- Phase A and B of external encoder must be wired into ports A and B.
- The External encode must be powered up and be 5VTTL compatible.
- MF1, MF2, or MF4 must be issued if encoder ports have been re-assigned from default
- MFMUL must be pre-defined (Mode follow multiplier)
- MFDIV must be pre-defined (Mode follow divisor)
- MFR must be issued (Mode Follow Ratio)
- G must be issued for the ratio to take effect

Example 1: Simple Mode follow ratio

Suppose you have a belt conveyor with a 500 base line encoder on it and you want to match speed with a SmartMotor. It turns out that for every 73 (post quadrature) encoder counts of the conveyor, you want the SmartMotor to go 11 counts.

You must first set a basic mode follow resolution.
Then you must set a specific ratio for your application.
Then you need to initiate it with a G (Go)

```
MF4          ' set motor to full quadrature mode follow (4 times base at 1:1)
MFMUL=11     ' set Mode Follow multiplier to 11
MFDIV=73     ' set Mode Follow Divisor to 73
MFR          ' set motor to Ratio Mode
G            ' Go (Initiate the above ratio values)
```

Note: The motor will actually start moving at the point the MF4 command is seen. The actual ratio does not take effect until the G is received.

If Ports A and B have not been used for anything else, MF4 may be omitted. If MF1, or MF2 is desired, they must be issued prior to MFR.

Example 2: On the Fly change to ratio

```
MF4          ' set motor to full quadrature mode follow (4 times base at 1:1)
MFMUL=11     ' set Mode Follow multiplier to 11
MFDIV=73     ' set Mode Follow Divisor to 73
MFR          ' set motor to Ratio Mode
G            ' Go (Initiate the above ratio values)
WAIT=40000   ' wait about 10 seconds
MFMUL= 22    ' change multiplier
MFR
G            ' initiate new ratio after 10 second of running at the older one
```

This is the same as the previous example but with a change added once the motor is already in Mode follow. The values of MFMUL and MFDIV can be derived from serial data, internal math expressions or Analog values.

Phase Offset Adjust in Mode Follow

It may be necessary to actually move the motor forward or backwards in real time over top of being in mode follow. For instance, you may have 2 conveyors matched in speed, but one is slightly behind the other positionally.

You can set a speed and a differential move to allow the one in Mode Follow to "catch up" with the other. this is done with the D (Differential Move) command and the V (Velocity) command.

Example:

Lets say this code was used to set up the ratio and start tracking a conveyor:

```
MF4          ' set motor to full quadrature mode follow (4 times base at 1:1)
MFMUL=11     ' set Mode Follow multiplier to 11
MFDIV=73     ' set Mode Follow Divisor to 73
MFR          ' set motor to Ratio Mode
G            ' Go (Initiate the above ratio values)
```

It is realized that after the motor is up to speed. it is about 4000 encoder counts behind whee it although be although it is going the right speed.

The following code will adjust its position or phase on the fly:

```
V=100000     ' set velocity to 100000
D=4000            ' set acceleration to 1000
G            ' Go (Start moving)
```

In this example, the SmartMotor will begin moving forward at a differential velocity of 100000 and stop 4000 counts further up. Differential velocity means the virtual speed difference between the master (External) encoder and the motor's internal encoder.

Mode Step (Receive Pulse and Direction Inputs)

MS, MS0, RCTR Standard Mode Step

Mode Step allows an external pulse train to be used as a command reference to position. It is an extension of Position mode to some degree in that it operated on a continuously updating commanded position in absolute position mode. It makes use of Port A for Step input and Port B for Direction Input.

Minimum Requirements for a move to occur in Mode Follow:

- MS must be issued
- Port A must have a step pulse applied

Port B is used to determine which direction the motor will turn. If it is at +5VDC the motor wil turn clockwise. If it is at Zero volts, the motor will turn counter-clockwise.

MS0 can be used to zero out the external counter register.
RCTR as in mode follow, can be used to report the external counter register.

Example:

MS 'initiate Mode step.

That's it.
The SmartMotor will now move 1 encoder count for very pulse received on port A.
To change it' direction, Port B can be changed from +5VDCot zero or vice versa.

MSR, MFMUL, MFDIV Step Mode with Ratio

Mode Step with ratio allows the user to define the number of internal counts the motor will move for any given external encode count change.

Minimum Requirements for a move to occur in Mode Follow with ratio:

- MS must be issued
- MFMUL must be pre-defined (Mode follow multiplier)
- MFDIV must be pre-defined (Mode follow divisor)
- MSR must be issued (Mode Step Ratio)
- G must be issued for the ratio to take effect

Example:

Suppose you want the motor to move 7 counts for every 39 pulses from an external source:

```
MS          ' set motor to mode step
MFMUL=7     ' set Mode Follow multiplier to 7
MFDIV=39    ' set Mode Follow Divisor to 39
MSR         ' set motor to Ratio Mode
G           ' Go (Initiate the above ratio values)
```

The Motor will now follow the pulse train at a ratio of 7:39 of internal to external counts.

Mode CAM (Electronic Camming)

Electronic Camming provides a means to produce cyclical motion such as found in conventional; mechanical cams. In a mechanical cam, rotation of a major axis provides a pre-defined motion off of a CAM follower or secondary axis.

This is the case with CAM mode in a SmartMotor as well. The main axis or master axis is an external encoder signal from another source. The SmartMotor then follows point table values to move to pre-defined positions while following the external encoder. Up to 100 positions can be defined. The positions are stored in byte sized (16 bit) array variables aw[0] through aw[99].

CAM Mode is similar to Mode Follow. In fact, the same minimum requirements must be met for Mode Follow as well as some additional requirements for defining how the CAM table will operate.

Minimum Requirements for a CAM Mode:

- Phase A and B of external encoder must be wired into ports A and B.
- The External encode must be powered up and be 5VTTL compatible.
- MF1, MF2, or MF4 must be issued
- BASE must be defined (Master Encoder cycle)
- SIZE must be defined (Number of points in CAM table)
- A Pre-defined CAM table must contain valid position data
- MC, MC2, MC4, or MC8 command must be issued
- G must be issued for the ratio to take effect

Note: SmartMotors use a 32-bit position register.

CAM array values are 16 bit. MC1, MC2, MC4, and MC8 are means of stretching 16 bit data as needed to fill the need for full 32 bit distances.

Definitions:

MC uses array values directly as positional data

MC2 multiplies array values by 2 to get position data

MC4 multiplies array values by 4

MC8 multiplies array values by 8

BASE total number of master encoder counts to get through the entire CAM table. This is the number of encoder counts the external or master encoder travels for the slave to compete it's cycle.

SIZE number of points in the CAM table

The motor takes **BASE** and divides it by **SIZE** to get point to point linear gearing data as in MODE Follow ratio. It then moves seamlessly from point to point in the CAM table as the master encoder signal changes.

Example: CAM Mode Press machine

Suppose you have a servo operated vertical press that must come down and stamp a product every 22 inches as it travels under the press on a conveyor. The press travel is 4500 counts from raised position to full down stroke. The conveyor is equipped with an encoder that puts out 80300 encoder counts every 22 inches.

You could then define a CAM table that has the following array data in it:

aw[0] 0 1000 2000 3000 4000 4500 4000 3000 2000 1000 0.

This is the same as aw[0]=0, aw[1]=1000, aw[2]=2000……through aw[11]=0
This example uses 11 points in the table, so **SIZE=11**
The conveyor distance defines the BASE, **BASE=80300.**

Now, from this data we can set up our CAM mode to free rum and stamp the product continuously.

```
                ' Define CAM table points into array variables
aw[0] 0 1000 2000 3000 4000 4500 4000 3000 2000 1000 0.
MF4             ' set motor to full quadrature mode follow
BASE=80300      ' number of encoder counts of master per cam cycle.
SIZE=11         ' number of points to look for in array variables aw[0-99]
MC              ' set motor to CAM Mode
G               ' Go (Initiate the above ratio values)
```

Let's see what will happen:

BASE/SIZE=80300/11=7300

Suppose the master encoder and the SmartMotor are both at zero:

as the master encoder starts to move from 0 to 7300 counts, the SmartMotor will linearly move from 0 to 1000 counts (1000 is the second point in the CAM table)

The SmartMotor will reach 1000 at the same time the master encoder reaches 7300.

As the Master encoder moves to 7300+7300 (14600), the SmartMotor will linearly move from 1000 to 2000. This same linear interpolation occurs throughout the entire CAM cycle. The SmartMotor will move to 4500 counts and back to zero for every 80300 counts of the master encoder cycle.

Note: CAM Mode table data must always start with zero. The last data pint does not have to be zero, but can generate an instantaneous position error that may exceed error setting. So be careful to create gradual CAM points so as to not cause position errors.

Motion Commands summary (Not inclusive in or necessarily covered in above sections)

A
Purpose: sets acceleration (also deceleration)
Syntax: A=##
Units: Encoder samples/sec/sec
Range: 32 bit signed value
Example: **A=700**
Default value: 0
(Typical values are from 100 to 3000. 3000 is very fast.

D
Purpose: sets relative move distance
Syntax: D=##
Units: Encoder counts
Range: 32 bit signed value
Example: **D=4321**
Default value: 0

E
Purpose: sets maximum position error allowed (exceeding causes a fault
Syntax: E=##
Units: Encoder counts
Range: 32 bit signed value
Example: **E=300**
Default value: 1000

G
Purpose: Initiates moves as well as various modes of operation
Example: **G**

I
Purpose: Holds Encoder Index Position
Syntax: **RI** (Reports location of Index pulse)
Units: Encoder counts
Range: 32 bit signed value

Note: RI clears "Index-Report-Available" (Bi) Status bit as well.

O
Purpose: sets motor encoder counter to desired value
Syntax: O=##
Units: Encoder counts
Range: 32 bit signed value
Example: **O=250**

OFF
Purpose: Disables Drive Amp
Syntax: **OFF**

P
Purpose: sets Absolute commanded position
Syntax: P=##
Units: Encoder counts

Range: 32 bit signed value
Example: P=20000

S
Purpose: Stops Motor at maximum capable speed
Syntax: S

T

Purpose	sets commanded torque for Torque Mode
Syntax	T=##
Units	None
Range	10 bit signed value (+/- 1023
Example	**T=512** (Sets motor to 50% full torque)
Default value	0

V

Purpose	sets maximum commanded velocity
Syntax	V=##
Units	Encoder samples/sec
Range	32 bit signed value
Example	**V=180000**
Default value	0

(Typical values are from 100 to 2000000)

X

Purpose	Stops Motor at rate of acceleration equal to A
Syntax	**X**

@P

Purpose	Holds Current real-time motor position
Syntax	R@P (Reports present position)
Units	Encoder counts
Range	32 bit signed value

Example:

```
IF @P>4000
        PRINT("Motor Position is greater than 4000",#13)
ENDIF
```

@V

Purpose	Holds Current Motor Velocity
Syntax	R@V (Reports present speed)
Units	Encoder samples/sec
Range	32 bit signed value

Example

```
IF @V<100000
        PRINT("Speed fell below 1000000",#13)
ENDIF
```

SmartMotor I/O Control at a Glance

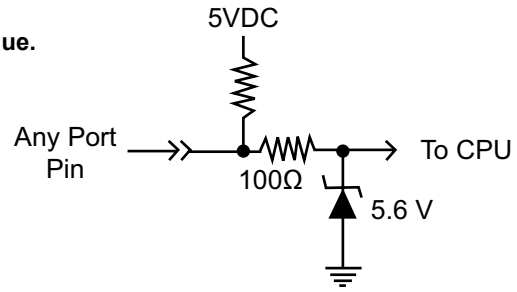
There are seven I/O channels available on a Smart Motor.

They can be assigned as either Inputs, Outputs, or 10 Bit analog inputs individually.

Note: At any time, regardless of usage, the ports may be read as an analog value.

I/O Port Hardware

Each I/O point has a 100 Ohm series current limit resistor with a 5Kohm pull-up
Each has a 5.6VDC Zener diode as well.
Each can sink 15mA and source 4mA



NOTE: They are not open collector P or N type outputs! They are Totem Pole CMOS driven outputs. When assigned as outputs they drive hard to either 5VDC or Ground when set to a 1 or 0 respectively

It is possible to use an Open Collector NPN type 24VDC prox with the SmartMotors.

The Use of PNP sourcing type 24VDC prox switches will damage the SmartMotor!

Dry contacts, if used, should be wired from the port pin to ground. If you wire them from the port pin to 5VDC, the pull up resistor will cause you to always see a logic 1 on the port.

Software Control of I/O Ports

This is an overview of SmartMotor Code to control the on-board I/O:
In each case, X denotes any port A through G and should be replaced with the port letter to be used.

Reading a Port as an Input:

UXI : defines port "X" as an input port; X is any port A-F

Example: UAI would define port "A" as an input port.

Assigning Port status to a variable:

x=UXI : assign the variable "x" the value at port "X".
: x will equal "1" if the port is at 5VDC.
: x will equal "0" if the port is at 0VDC.
: x can be any variable a-z, aa-zz, aaa-zzz..

Example: y=UAI would assign the logic state of port A to the variable y.

Reading a Port as an Analog value:

x=UXA : assigns the variable "x" a 10 Bit analog value from 0 to 1023 where 0 VDC=0 and 5VDC=1023

This command reads the selected port via a 10BIT A/D converter.

Example: z=UBA would assign the 10-Bit analog value of port B to the variable z.

Note: This is useful for self-diagnostics. By placing a known voltage dropping resistor on an external switch or other device, you can use the port as a general input, and yet do an analog read to check if the switch or sensor wire may have become disconnected from the port.

Assigning a Port as an Output:

UXO : defines port "X" as output port

Example: UEO would assign port B as an output.

UX=1 : drives voltage on port "X" to 5 VDC

Example: UE=1 would set port E to 5VDC.

UX=0 : drives voltage on port "X" to 0 VDC

Example: UE=0 would set port E to 0VDC.

Note: You can pre-define the state of a port prior to assigning it as an output. Then you can toggle it between input and output to change from a driven state to a floating state. This is useful when tying I/O together between different SmartMotors.

Default States and special uses of I/O ports

Ports A and B Defaults and Specifics:

Ports A and B as quadrature encoder inputs:

Default state: Ports A and B default to phase A and B Encoder Inputs. If no Port Control commands have been issued to Ports A or B, at any time, you can use the CTR command to get counter values from these ports:

Example: RCTR would report counter status from ports A and B

x=CTR would assign the counter value to the variable x

MF0 would re-set the counter to zero. (Mode Follow Zero)

Using MF1, MF2, MF4 or MFR will allow you to make use of ports A & B as quadrature encoder input ports and place the Motors into Electronic gearing or Mode Follow.

See Help files or manual for more.

Ports A and B as Step and Direction inputs:

The MSO command re-sets the counter to zero and sets up Ports A and B as Step and Direction inputs respectively. Once doing so, the same rules apply as listed above for the RCTR and CTR commands.

Setting up the motor as a stepper via MS, MS0, or MSR command will make port A be a step input and port B a direction input as well.

See Help files or manual for more.

Note: MS0 command is a good way of using port A as a high speed counter It zeros the Counter without changing the motor's mode of operation. From that point on, RCTR can report counts from prox switches, laser scanners etc.. as a high speed counter.

Ports C and D Defaults and Specifics:

Ports C and D default to right and left limit inputs respectively. If they are assigned as input or output ports, they must be redefined as limit switch inputs if you want them to be used as such. To redefine them as limit inputs you do the following:

Example:
```
UCP   'redefines port "C" as right limit input  (P for plus rotation)
UDM   'redefines port "D" as left  limit input  (M for minus rotation)
```

Note: BRKC Command has been added in later V4.15 and >=Versions 4.76 firmware. This allows Port C to control and External brake via brake control commands. See also: BRKSRV, BRKTRJ, BRKENG, BRKRLS and the V4.76 Firmware document for more details.

Default States and special uses of I/O ports (Continued)

Ports E and F Defaults and Specifics:

Ports E and F can be used as the Anilink port or an RS-485 port. Any command used for communication to Anilink devices will automatically set the ports as needed for Address and Data information. Using the OCHN and OCHR commands with regards to RS-485 usage will automatically set the ports to half duplex RS-485 operation.

See Help files or manual for more.

Ports G Defaults and Specifics:

By Default, when port G sees a transition from 5 to zero volts, it means the same as typing a "G" and pressing enter or seeing a G in a program. This is why it is called the G Sync pin. This allows you to synchronize "goes" on multiple motors at the same time. The G Port can be redefined as general I/O like the other ports.

UG : command returns Port "G" to default sync port so if you assigned it as an input or output, To redefine it as "G sync" simply invoke the UG command again.

Port G also has the option of being used as a handshake line for RS-232 to RS-485 Adapters. This is covered in the Help Files and manual under communications.

Note: BRKG Command has been added in later V4.15 and >=Versions 4.76 firmware. This allows Port G to control and External brake via brake control commands.

See also: BRKSRV, BRKTRJ, BRKENG, BRKRLS and the V4.76 Firmware document for more details.

I/O Programming examples

The following are examples of triggering events off of Port I/O state changes.

Level Triggered Subroutine call

This code causes subroutine 100 to be called when port A goes high

```
IF UAI==1
      GOSUB100
ENDIF
```

Positive-Edge-Triggered Subroutine Call

This code causes subroutine 100 to be called when port A goes high only after first going low (negative pulse triggered subroutine call)

```
IF UAI==0
      WHILE UAI==0
      LOOP
      GOSUB100
ENDIF
```

Default States and special uses of I/O ports (Continued)

Negative-Edge Triggered Subroutine Call

This code causes subroutine 100 to be called when port A goes low only after first going high (positive pulse triggered subroutine call)

```
IF UAI==1
      WHILE UAI==1
      LOOP
      GOSUB100
ENDIF
```

Level and State Change Print-Out example

```
WHILE 1==1                'while forever
               WHILE UBI==1 LOOP     'while port B is high, do nothing
               PRINT("Port B just went Low",#13)
               PRINT("Do nothing while it is Low",#13)
WHILE UBI==0 LOOP         'while port B is low, do nothing
               PRINT("Port B just went High",#13)
               PRINT("Do nothing while it is High",#13)
LOOP
```

The F command value allows Enabling or Disabling of special firmware functions of the SmartMotor Processor and Drive Stage.

Syntax: F=value

The Value is a Binary Bit weighted value with each bit controlling a specific feature.

Bit Value Functions are as follows:

1 **Decelerate to stop on limit switch input (as opposed to just turning off)**

2 * **Invert Commutation (Changes Shaft rotation)**

4 **Any Report commands transmit to Com 1 only. (Use with Extreme Caution)**

8 **Clear PID integral term at trajectory-end to avoid possible slow settling**

16 * **Mode Cam positions are relative for each re-entry into CAM table (from either direction)**

32 * **GOSUB1 is issued under motor fault condition**
C1 can not be called again prior to receiving a RETURNF

64 * **GOSUB2 is issued on user input G transition from high to low**
C2 can not be called again prior to receiving a RETURNI

128 * **Internal Slave Counter = base + dwell modulo while in CAM Mode**

256 * **Set T.O.B. to be active for entire move profile.**

512 * **Suppress T.O.B. until Slew Velocity has been reached**

1024 * **Enables Port G to Index trigger latch function (only in SM2316D/DT >=4.93 firmware)**

*** Note: Only Applies to >=v4.77 only……..**

Warning: C1 has priority over C2. C1 can be activated when in C2.

The F value can be changed on the fly while in an Interrupt subroutine to change its effect. An example would be turning off the G interrupt once in C2 to prevent any subsequent calls.

F Command is Binary Bit flag additive:

Example:

```
F=21 would break down to F=(16+4+1).
```

Motor would run CAM Mode relative, redirect print statements to port 1, and decelerate on limits.

Example using F=32 for Interrupt driven Fault routine

```
F=32            'Enable C1 Fault routine


MV              'Set to Velocity Mode
V=10000         'Set Speed
A=100           'Set Acceleration
G               'Start moving in Velocity Mode

END


C1                          ' Fault Routine (Gets called on any of the following faults)
        IF Be               ' Checking for error status bits
                PRINT("Position Error",#13)
        ENDIF
        IF Bh
                PRINT("Over Temp Error",#13)
        ENDIF
        IF Bi
                PRINT("Over Current Error",#13)
        ENDIF
        IF Bl
                PRINT("Left/Positive Travel Limit Error",#13)
        ENDIF
        IF Br
                PRINT("Right/Negative Travel Limit Error",#13)
        ENDIF
        WHILE 1             'Wait for Motor Reset
                IF r==1     'If host sends r=1 via serial port
                        ZS  'Reset the motor
                ENDIF
                IF UAI==0   'If Input A gets rounded
                        ZS  'Reset the motor
                ENDIF
        LOOP
RETURNF                     'Return form Fault routine
```

Example using F=64 for Port G, C2 interrupt subroutine call

```
F=64            'Enable Port G interrupt routine
END
C2              ' Port G interrupt Routine
        PRINT("Port G was grounded",#13)
RETURNI         ' Return  from Input Trigger
```

Example using F=64 for C2 subroutine call and F=1024 Index Re-direct for position capture

```
F=64            'Enable Port G interrupt routine
END
C2              ' Port G interrupt Routine
        PRINT("Port G was grounded",#13)
RETURNI         ' Return  from Input Trigger
```

1. Serial Buffer command: ! YES….., "!"…… is a command……

This command halts subsequent code execution until ANYTHING is received into the serial port. It can be used as an effective means to detect electrical noise on the communications cables.

Example:

```
WHILE 1      'while forever
      !      'wait here for any incoming serial data
      PRINT("noise",#13)
LOOP
```

2. Break Control Commands: (means to control internal break option)

BRKSRV (Default State) This command causes the break to mechanically engage on any protective fault.

This includes: Position Error, Over Temp, and Travel Limits.

BRKTRJ (Optional State) This command will cause automatically Disengage and re-engage of break upon any commanded shaft motion. It DIRECTLY follows the Bt (Busy Trajectory) Status Bit. If Bt is 1, then the break gets power and is mechanically disengaged. If Bt is a zero the Break loses power and is re-engaged Proper delays have been included in firmware to allow for mechanical response time.

NOTE: When BRKTRJ is issued, the motor drive stage will turn off (or be off) any time the Bt bit clears. The LED that would normally follow the Bo bit would be Red any time the Bt bit is zero, and yet the Bo bit will still be zero as well. This is crucial to know when writing error detection code.

BRKI (Default State) Directs Break Control signal to optional internal Break
BRKC Redirect Break control signals to Port C pin.
BRKG Redirect Break Control signals to Port G pin

> **Note 1.**: in both cases, the I/O pin is at ground level when the break should be powered up and mechanically disengaged.
>
> **Note2**. : These commands are useful for automatic "busy" signal to PLC's.

3. MF0 and MS0.

Both commands can be issued without causing motion or changed mode of operation.

MF0: Sets up Port A and B to read external encoder signal at 4X interpolation.

MS0: Sets up Port A for Pulse (or Step) input and Port B for Direction Input

In both cases, CTR (External Counter Register) will be set to Zero
To read CTRE, issue RTCR at any time.

MS0 is good to using Port a as a high speed counter. It can read signals at up to 2MegaHerts rate.

> **Note:** If it is desired to initiate Mode Follow or Mode Step with MFR (Mode Follow Raito) or MSR (Mode Step Ratio), the use of MF0 or MS0 is the only means to enter without first causing a mode change prior to "G" command.

4. UG (Default state control of Port G Input pin).

By Default, at any time Port G pin is grounded, this is the same as issuing the "G" command. On rare occasions, this default state can cause accidental motion or changes in Modes of Operation. This can occur when a long cable is connected to the Port G pin but not terminated to 5VDC or ground.

If the pin is grounded on power-up, the motor will immediately turn on the drive stage and hold position.
To turn off this "G" function, issue UGI or UGO to assign the I/O pin as needed.
At any time, UG can be entered to enable "G" command functionality.

5. PID1, PID2, PID4, PID8 commands.

Each of these commands control the rate at which the CPU updated the PID Positional Control Loop.

PID1 is the default state. This means every servo-sample (every ~250 microseconds) the processor compares calculated trajectory position with actual real-time position, calculated the error and sends updated PWM commanded to the drive stage to provide proper force at the shaft.

PID2, PID4, and **PID8** will lower the update rate to every 2, 4, or 8 servo sample periods respectively allowing for more Code and Communications execution time.

> **NOTE: PID** commands DIRCTLY change the effects of V (Velocity), A (Acceleration), and PID "K" values.
> This **PID** commands are best utilized when faster I/O control is needed (WHILE NOT MOVING). Otherwise, proper V, A , and KP, KI, KD changes must also be made when changing PID settings.

6. KG parameter. (Gravitational PID term)

This term DOES NOT require the now obsolete KGON and KGOFF commands. The KG command applies a net offset to the operating PID filter to compensate for vertical load or continuous offset load in one direction only.
It can be used to shift position error to zero as well.

> NOTE: KG parameter is a 32 bit parameter and may not show ANY effect until values well above +/-1 million are used. As with other PID values, the "F" command must be issued for it to take effect.

7. ENC0 (Default) and ENC1 (Optional) commands

These commands tell the processor which encoder to use when calculating PID and PWM trajectory control. ENC0 is default and utilizes the internal encoder. ENC1 causes the processor to look at Port a and B for external encoder input.

NOTE: External Encoder Resolution directly effects PID effectiveness.

> **Example:** Suppose internal encoder resolution fo 2000 counts/shaft revolution. Then you add an external encoder that has an effective resolution of 4000 counts/shaft revolution for the same distance traveled. Then you have effectively doubled ALL PID parameters. In other words, when ENC1 is issued, this will also have the effect of Setting KP to KP*2 and KI to KI*2 etc……

8. D command.

This command is normally only thought of as a relatively position move command. Under Normal operation in MP (Position Mode) a non-zero value of D with a subsequent "g" command results in a relative distance move from where ever the motor shaft is to "D" encoder counts from there.

HOEWEVER: The "D" command is also used for Phase Offset moves while in Mode Follow or Mode Step. As a result, if you were to use MFR (Mode Follow Ratio) or MSR (Mode Step Ratio), Both require a "G" command to initiate the ratio. If "D" is not zero (for example 2000), then immediately upon issuing the "g" command after either MFR or MSR, the motor shaft will move forward 2000 counts.

The "D" command is also used for "Dwell" in CAM mode in the "PLS" firmware. Consult the PLS firmware addendum on the web for more information on this specifically. To be safe, "D" should be set to Zero at all times when not specifically needed.

9. Bs Status Bit, (Syntax Error Bit) also known as the Bull$H1T command……

At any time there exists more than one motor on a Serial bus (RS-232 or RS-485), the Bs status bit may be set to 1. The reason for this is quite simple. It is set ot one any time a motor in the "addressed" state (meaning it is accepting and processing all incoming commands) receives a character string that I s not a valid command string.

> **Example:** You send the command RP to motor 1. It responds with 1234 Well, Motor 2 or any other motor downstream on the serial chain will get "1234" and will know it is just numbers and not a valid command. Therefore the Bs bit will be set to 1. This is very common any time global address commands are sent to motors.

10. OFF command and Bo (Motor Off Status bit).

The OFF command simply sets the drive stage to an OFF condition.

Note: In all firmware, the **Bo** bit will immediately be set to "1". In ALL PLS firmware motors (4.62 (servo step) and >=4.76 (SmartMotor series), the motor will be placedin MTB (Mode Torque Break.) The result is this: In non-PLS firmware, the motor will freewheel and the shaft will turn freely.

On all PLS firmware motors, the motor windings will be shorted out. So the OFF command can result in quick stops.
The **Bo** bit ONLY gives the state of the drive, NOT what the processor thinks caused the "OFF" condition.

> **Example:** On Power-up, the **Bo** bit will be "1" prior to any subsequent motion commands. The LED on the motor will be Red. This does NOT mean the motor is in a faulted state. It simply means the motor is OFF.

11. RUN?

This command does not prevent the motor from running code upon power-up. It prevents the motor from running code beyond the RUN command on power-up.

> **Example:**

```
PRINT("Hello World",#13)
RUN?
PRINT("Hello Again",#13)
END
```

Upon Power-up the motor WILL print "Hello World", but WILL NOT print "Hello Again", until "RUN" is issued via serial port.

12. SILENT, SILENT1

These command prevent PRINT commands from printing to their respective ports. They do not work when sent form the serial port. They only work when in the downloaded program.

13. VLD and VST,

These commands are used for EPROM read and writes. They cannot break the pre=assigned variable sets.

> **Example:**

```
EPTR=100
VLD(a,26)    'loads values starting from 100 to all variables a though z.
EPTR=100
VLD(a,27)    'Invalid because it breaches the "z" variable.
EPTR=100
VLD(ww,10)   'Invalid because it breaches the "zz" variable.
```

The same riles above apply to VST command as well.

14. RETURNF, RETURNI (PLS firmware only)

RETURNF is the proper form of **RETURN** used at the end of C1 subroutine user code when C1 is called on interrupt when a protective fault occurs.

(Note: F=32 bit flag must be set to enable C1 to be called).

RETURNI is the proper form of RETURN used at the end of C2 subroutine user code when C2 is called on interrupt due to Port G being grounded.

(Note: F=64 bit flag must be set to enable C2 to be called)

If it is desired to call C1 or C2 manually (or via code) by use of GOSUB1 or GOSUB2, then a regular "**RETURN"** command must exist somewhere below C1 or C2 to avoid a memory mapping error or undesired program flow.

15. MTB (Mode Torque Break)

This command immediately causes the motor winding to short out. No "G" command is required.
Care should be take when it is used. The stopping force of the motor using **MTB** is >60% better than in any other mode of operation or any deceleration. The motor uses no power from the external power supply to stop the shaft. It only uses dynamic Back-EMF to stop shaft rotation. This does not mean that a "Z" axis can be used without a fail safe mechanical break..

16. TH and THD commands and the Bh Status Bit

The **TH** command sets the temperature trip point to a value LOWER than default.
The normal setting is 70 for a trip point of 70Deg.C (or 85Deg.C on optional motors).
The **TH** command ONLY further limits operating conditions of the motors.

The **Bh** (Over temp) status bit will be set to 1 if the **TH** command setting is exceeded.
The motor must drop 5 Degrees below the **TH** set point before continued operation is allowed. This setting is in firmware and cannot be altered.
Example: If **TH** is set to 55, then the motor must cool to 50Deg C before the CPU will allow the motor to operate again.

The **Bh** Bit is ALSO the RMS over-current bit. The reason for this is that both RMS Over-current and Over Temp are effectively heat-generated failure conditions.
The **Bh** bit will be set any time the RMS over current setting (factory set per motor drive) is exceeded for a a time set by **THD**. The Default value for **THD** is 12000. This is approximately 3 seconds.
The time can be increased or decreased. To increase it can cause sustained RMS over currents long enough to possibly damage the motor.
It is not recommended to increase **THD**.

17. AMPS command. (Defaults to 1000)

This command sets a limit to the maximum % PWM available to the drive Stage.
Valid values are from 0 to 1023 representing 0 to 100% PWM.
NOTE: The AMPS command DOES NOT LIMIT PEAK POWER, ONLY RMS POWER.
The scope of the full physics behind this is beyond the intension of this document.

18. STACK

The Stack Command clears ll return-memory pointers from anyh WHILE LOOP or GOSUB call that had been previously stored. Only a maximum of 7 pointers can be stored. This means the motor program code cannot be "nested" more than 7 levels deep or have more than 7 recursive calls.
Example:

```
WHILE 1
      WHILE 1
            WHILE 1
                  WHILE 1 etc…  7 levels deep.
```

The STACK command should be used with care. It is recommended that it only be used if followed by a GOTO command back to some higher place in code if not the very beginning.

19. X and S commands

When the **X** or **S** commands are issued, the motors will decelerate to a stop from their present trajectory calculated speed. This calculated speed has a high granularity fo measurement. It is on the order of 2^16 or ~65000 in Velocity units. As a result, it may be that issuing an **X** or **S** command while moving very slowly, could cause a jump in speed and on rare occasions a reversal in direction. The firmware has been adjusted as of late 2004 to minimize the effect, but cadre should be taken if very slow speeds or high fluctuations in position error are expected.

Recall, the **X** command decelerates at the value determined by A" (Acceleration. The S command deceleration rate is at a high value of around A=8000 or so. So it is possible that the **X** command can decelerate faster than the S command.

High values of "A" in conjunction with the **X** command can result in the Ba (peak over current) status bit being set. If sustained long enough, the motor may get a position error.

20. Ba (Peak Over Current) Status Bit.

This bit indicated a change to active operational drive stage control. When the **Ba** Bit is reached, this indicated the drive stage has reached saturation. It DOES NOT indicate that the motor has faulted or that it will fault.
When the Drive stage reaches saturation as detected by the internal current-sense resistor, the processor sets PWM to 30% ntil the current sense resistor shows the current demand drop off.
Motion will continue through this and the motor may complete the more without any noticeable effect.

What may be seen is a slight Yellow glow the the LED or possibly completely Red for a moment.

If the motor load is high enough such that the Peak over current condition persists, then the motor will begin to fall behind at a faster rate. If it falls behind far enough to exceed "E" (maximum allowed following error), then the motor will in fact fault on Position Error (Be). But the Ba bit would not have been the physical reason for the fault.

If a given application results in multiple instances of the Ba bit being set, this indicated that the motor may be undersized in the Peak range.

If the motor has no attributable heat build-up, then it indicates ONLY pe4ak over loading, not continuous RMS overloading.

Keep in mind, RMS overloading causes heat build-up.

Peak overloading causes instantaneous Ba Bit and 3-% PWM limitation and shows signs of peak over loading.

21. LOAD and RCKS command.

The **LOAD** command causes the motor to go into EEPROM Program Write Mode. Once the command is issued, all subsequent data is written directly into Program EPROM. This will continue until 2 (two) HexFF bytes are sent to tell the CPU the download process has ended.
Normally, this is done from a PC or the SMI software or possibly a PLC. The data that is sent should be a valid .sms compiled program for the SMI software.
Note: The **LOAD** command should NEVER be issued manually at any time in any terminal screen.

After a download is complete, the SMI software automatically issues the RCKS command. This command reports the Checksum for the downloaded program to insure the download was successful.

Of the motor processor calculates a matching checksum to the checksum in the downloaded program header file, then the checksum will be followed with a "P" for passed. Otherwise it will be followed with an "F" for failed.

Any time RCKS returns a number followed by "F", then the motor program uis corrupt and should be cleared form EEPROM by downloading again.

Any .smx compiled program should not be opened in Word or Notepad. Doing so will corrupt the files because Microsoft will insert line ends and line feeds to all lines in the smx file resulting in syntax errors throughout the entire program. If it is desired to store an smx file in a PLC, then the file must be downloaded to the PLC as is.

Selecting Power Supplies: Switching, Linear, and Unregulated Power Supplies:

Switching Regulator Power Supplies are the most common, most compact and now becoming the most economical power supplies. A "Switcher" or Switch Mode Power Supply typically takes incoming high voltage AC power, Rectifies it to High Voltage DC power and then through a Voltage-Mode PWM control, will deliver a much lower DC voltage to loads. Some are referred to as "PFC" switcher meaning they are "Power Factor Corrected". Without going into detail, this means basically a few things:

1. They can take in a very wide range of AC voltage, typically 100 to 240VAC,
2. They will correct for impedance shifts from "real" to "reactive" power giving a more unity power factor as is seen from the AC side.
3. As a result of the above, they will also reflect load changes back to the AC side.

Most switchers will go into an OFF state on over voltage. Few of them have a buck-regulator that prevents over voltage. The ones that do are very costly and large. As a result of this it is highly recommended to use a Shunt when using a switching power supply to aid in suppression of bus over voltage. Switching Power supplies should be sized to provide maximum expected current for the entire motor system under the worst load considerations. This is because Switchers have no "reserve" like Linear Power Supplies do.

When they reach maximum current, they shut down or reset. If a Switcher is rated for 10 Amps, for example then if the load exceeds 10 Amps, it will shot down. However, the voltage does not drop down at all from no load to full load. As a result, a well sized switcher works well when an application requires very high speeds (no voltage losses) or extremely fast accel/decal times. As long as the peak current of the motor does not exceed that of the continuous rating of the switcher, then a switcher will work well with servo motors.

Linear Regulated Power Supplies are becoming less common. Unlike switchers, they must be used at a proper fixed input voltage. They do have a reserve current capacity but as load increases, their voltage drops while current goes up. They still maintain a fairly good control over voltage regulation though. As a result, they have little fluctuation in current or voltage. They are good for consistent continuous duty applications that have only small changes in load. Constant conveyor feed, mixers, pumps and similar applications work well when run by regulated linear supplies.

Unregulated Power Supplies are typically brute-force open frame transformers with a full wave bridge rectifier and one large Electrolytic Capacitor. Similar to Linear Regulated supplies, they are designed to run at fixed input AC voltages, however, they are often times supplied with multiple input taps for a selection of input voltages.

There is no regulation in them. As load goes up, current goes up and voltage goes down. As a result, they can output extremely high current surges even if at a substantially reduced voltage. When sizing an Unregulated Supply, it is good to know the output voltage at maximum load. Be sure that this voltage is high enough for the required speed of the servo motor. If so, then an unregulated supply is the best choice for demanding applications with high load swings and heavy peak loads. It is not uncommon to find reasonably priced unregulated supplies that can surge >80 Amps short duration. This makes them idea for heavy loads accelerating quickly.

Mechanical Brakes:

If at any time, a load can be easily back driven or is in a vertical orientation, a electromechanical fail-safe brake is highly recommended. Under no situation should a PLC or external controller be used to control a fail safe brake on a servo. The response time will be diminished to the point of defeating protection. Instead, use the SmartMotor interrupt control features stated here:

```
BRKG command in conjunction with the BRKTRJ or BRKSRV commands.
```

Here is the detail on that:

```
BRKC         'send automatic brake control signal to Port C.

BRKG         'send automatic brake control signal to Port G
BRKSRV       'Engage brake on any motor shaft fault
(Position Error, Limit Switch  Error, Continuous Over Current/Over Temp)
BRKTRJ       'Engage brake when not moving
(Follows the Bt "busy trajectory" status bit)
```

In making use of selected commands from above, the brake will get a signal to engage (be de-energized) within 250 to 500 microseconds of it's trip condition. Using the PLC will cause a delay of anywhere from 4 to 10 milliseconds due to scan time, process time and brake release time. By then, current in the control could have already well exceeded limits.

Position Error Limits:

Let's suppose you have a maximum allowable position error limit of 1000 encoder counts: The motor can hit a hard stop and go up to 999 encoder counts into position error before a trip condition is met. The time is takes to get to that position error my be slow or fast depending on the speed you are moving. Set "E" to lowest value possible to allow continued machine operation without spurious position error faults occurring. This will provide the most protection both for the equipment and for personal safety.

By lowering the limit to a close margin above normal operating conditions, the motor will fault out quickly upon hitting any unexpected object (or person) A properly set following error limit can literally save lives.

The best way to set the limit is to run the application under normal operating conditions. Then poll PE (Position Error) live while running. Note the peak value of position error, typically while accelerating quickly. Then set E to ~5 to 10% above that value. This will give a good margin for variations in load and friction.

Amplifier Tuning

Lets suppose you have "tight" tuning. KP>300 or so, KD>2000 or so, This is just an example of slightly tight tuning, but not too high. The higher the numbers, the faster motor current will rise under a given increase in position error. Collectively with the above mentioned facts about "E" maximum allowed position error, current may rise much faster. It is best to ratio acceptable tuning values with good Position Error values so as to maintain the lowest running position error with the lowest value of "E" possible,

The ironic thing here is that usually to get Position error down implies increasing tuning. this is true, but for example: KV (Velocity Feed Forward) and KA (acceleration feed forward) are better means to achieve this. They lower position error while moving without increasing motor current because they shift the motor position command forward in the trajectory for the entire move, vice during the dynamics of changes in moves. As a result, you get lower peak currents in the motor. This being the case allows for E to be set lower thereby making the machine safer and the hardware last longer. As a rule of thumb for present day SmartMotors:

KP is Proportional Gain and defaults to 42 Typical values are from 20 to 500. KP is a gin that command PWM to the drive stage directly proportional to position error. As Position error increases, KP increase PWM duty cycle proportionally. The higher the value, the faster the PWM increases for an increase in position error. KP can be considered as Velocity Gain.

KI is Integral gain, It integrates over Position error and is the integral gain for KP. IT is therefore somewhat like Position gain. It reacts slowly and aids in maintaining position, not speed. When there is high friction in a system and proportional gain is not set high enough to get a motor into final position, the KI gain will build up PWM over time until the position error is reduced, Then KI will drift back down until the motor settles into position. As a result, KI can substantially reduce heat in a high friction application that has any considerable amount of dwell time my reducing dwell time position error to a minimum. Once position error is reduced to near zero, the drive stage needs very little current. Less current means less heat. If you find a motor to be hot when sitting still but cooler when running, then KI gain needs to be increased. **KI** is typically ½ to 1-1/2 times KP.

KD is Differential gain, the differential of **KP**. Since **KP** is like velocity gain and the differential of velocity is acceleration, then **KD** could be thought of as acceleration gain. **KD** is a high frequency and fast acting gain. It handles the quick changes in velocities and deals with surges in demand. Since much higher PWM is needed to accelerate quickly, **KD** typically needs to be much higher than **KP**. Many have heard the rule of thumb that Moment of Inertia Mismatch should be <=10:1. This is because most servos can accelerate ~10 times faster than gravity and moment of inertia is based on gravitational acceleration. As a result of this KD is typically set around 10 times the value of KP. KP defaults to 42 and KD defaults to 550. In tight tuning KP may be 250, and KD would be nest at around 2500 as a result.

Power supply Voltage Levels

The higher the voltage, the fastest the motor can move and the faster it can accelerate. This is a good thing. But in conjunction with that, the higher the voltage, the closer to a peak voltage for over-voltage break down of the controller. Also, the higher the voltage, the faster a rate of change of current can occur. It is a risk with any application to get faster response by moving towards a higher voltage. Typically speaking, it is the dynamics of sudden changes that increases risk by a square function whereas the continuous load risk is only a direct ratio increase. This is because rate-of-change in current is proportional to acceleration which is the square of velocity, i.e. V^2.

For safety sake, a 42VDC supply for a 48VDC system gives good margin with little speed losses. In the same manner it must be note: To move from 24 to 48VDC gives you the ability to go twice as fast. However, the ability to accelerate goes up by a squared increase or 4 times faster.

To repeat this: To double voltage doubles speed and gives 4X acceleration capability. But with this comes the above mentioned increase in current as well. So be sure the power supplies and motors are sized properly.

Firmware Options:

Once the motor is purchased, the firmware is already there and cannot be changed. For the sake of added safety, here are some notes on the newest options: 4.78xx firmware: This newest of the "plus" version firmware options has the ability to suppress back-EMF voltages any time the calculated trajectory has been exceeded by actual motor motion. In other words, the processor is looking at where it should be vice where it actually is. Any time the motor exceeds dynamic position per calculated trajectory, the drive amplifier shunts power to maintain dynamic position control. As a result, excessive currents are suppressed at a rate or response of ~250seconds. (within PID update rate). The result is faster and higher stopping power and les overshoots in speed. For more on this, consult Trajectory Overshoot Braking documentations

Back EMF and Hard Stop Crashes

Back EMF is the voltage generated when a rotor is moving within the stator of any motor. It is literally the motor acting a s a generator. Automatic Shunts should be employed in any system where Back-EMF is likely, such as vertical loads or back-driven loads. There is a common rule that Back EMF or voltage generated is proportional to Velocity. This is true in a constant velocity condition only. Back EMF is actually proportional to the rate of change of magnetic flux (magnetic field strength) inside the stator windings of the motor. The faster the rate of change, the higher the voltage rises. In other words, RPM of the motor shaft does not have to be that high to have very high voltages created.

Here is an example: Take any relay coil or solenoid valve coil in a 24VDC system. When it is energized, the magnetic field pulls in the contactor or pilot valve. The magnetic flux reaches saturation and a DC electromagnet is formed. When the power is removed from the coil, the magnetic flux rapidly collapses because there is no forward voltage to maintain it. Since the circuit is now electrically open, there is nothing to prevent the magnetic flux from collapsing rapidly at a hyperbolic rate. The result is something called "inductive-kick". This kick or spike in voltage for a 24VDC coil can reach very high voltages and currents on the order of 100 times that of the original applied voltage, i.e. 2400VDC!.This is why it is very common to place reverse polarity diodes across relay coils and solenoid valve coils. It protects the system from high voltage spikes.

The same thing occurs when a motor hits a hard stop. Suddenly, the rate of change of magnetic flux in the stator windings skyrockets upward because the rotor stopped moving. This sudden change causes an excessive voltage and current spike in the controller and can damage components.

Now: what can we do about it?
Practically speaking, not much. This is similar to a car crashing in to a brick wall. If the passengers are belted in, they may survive, but the car will sustain unavoidable damage due to the rapid change in speed. (Infinite deceleration to zero speed). No amount of "practical" mechanical design for a typical car will save it from damage when it hits the brick wall.

Practical design, means, yes, you could make that car into a large bulky tank that would not get hurt, but then the car would be very heavy, with little space for passengers and be very slow and bad on fuel consumption. This is not practical. The same applies to motor drive design. We could design the drive stage to be able to take the hit of a fast hard stop. But the drive stage would be very large. The controller would have a lot more components in it and the practicality of it would be diminished. The motor would grow in size for the same torque output to 3 times larger. This is just not practical.

Hard Stop Crashes:

The best recommendation for preventing damage to the motor/controller in the case of hitting a hard stop is to place a limit switch near the hard stop that trips the motor off line just prior to hitting the stop. The best way to prevent it beyond that is to prevent the cause of hitting the hard stop in the first place. If this is due to slow brake response, then use automatic control as mentioned above in the brake section of this document.

If this is due to jogging the motor in Velocity mode and not letting of the jog switch in time, then jog in position mode instead and use the "X" or "S" command to stop the motor when the jog switch is released. In any case, much care should be taken to be sure the motor is not intentionally or unintentionally allowed to hit a hard stop while under normal speeds and load conditions. Even a protective shunt will not always be good enough to suppress a large inductive kick that results form a hard stop crash.

Loss of Power at motor connector while under load:

At first glance, this may not seem like a big deal. In reality, it is. As discussed above, Back EMF is potentially damaging to the motors. If the connection to the motor is lost while under load, there is a chance that the back-EMF spike could reach damaging levels. If E-stop safety in a particular applications requires removal of dive power, then be sure a protective shunt is between the E-stop contact and the motor connector.

Various Loops, Trigger Events and Subroutines

Wait on Input to Go Low

```
WHILE UAI==1 LOOP        ' Watch Port A until it goes low
```

Wait on Input to Go High

```
WHILE UAI==0 LOOP        ' Watch Port A until It goes high
```

Check Input While Moving and perform some function

```
WHILE Bt                 ' While Busy Trajectory Bit is On
      IF UAI==1          ' If Port A goes high
            GOSUB10      ' Run Subroutine 10
      ENDIF
LOOP                     ' Continue Checking
```

Check For Errors after a move

```
TWAIT                    ' Hold While Busy Trajectory Bit is On
IF Bo==1                 ' If Motor Off Bit comes is on
GOSUB1                   ' Run Subroutine 1      (see below)
ENDIF                    ' Continue Checking
```

Example Error Handler Routine (see above)

```
C1     ' Start of Subroutine 1
IF Be                    ' Checking for error status bits
      PRINT("Position Error",#13)
ENDIF
IF Bh
      PRINT("Over Temp Error",#13)
ENDIF
IF Bi
      PRINT("Over Current Error",#13)
ENDIF
RETURN                   ' Return to line just below GOSUB1
```

Note: Bo is the Motor-Off status bit. It represents the state of the drive amplifier but NOT a fault condition. It just states whether the drive amplifier is on or not.

The Drive Amplifier WILL turn off under any of the following conditions: Position Error (Be, position error status bit)

Travel Limit Error (Br, Bl, Bp, Bm limit switch status bits)

Over Temperature/Continuous Over Current (Bh status Bit)

In the "Plus" version firmware (4.76 or later), The motor will automatically suspend all program execution upon reaching any of the above stated faults. (In other words the "END" command is issued). There is a n option to automatically call C1 on interrupt vice ending program execution by issuing F=32. The C1 subroutine must end with RETURNF (return-fault) for this to work properly.

Various Loops, Trigger Events and Subroutines (continued)

Pause Move on Input-High and Continue upon going low

```
x=100000                ' Assign desired position to some variable
P=x                     ' Assign that variable to Commanded Position
G                       ' Start move
```

Scan input while moving

```
WHILE Bt                ' While Busy Trajectory Bit is On
      IF UAI==0         ' If Port A goes low
                        ' jump to "pause" subroutine
            GOSUB20     ' Run Subroutine 20
      ENDIF
LOOP
```

Pause Subroutine (see above for calling code to this)

```
WHILE Bt                ' While Busy Trajectory Bit is On
C20   ' Start of Pause Subroutine
      X                 ' Stop Motor with X command (Decel to a stop)
      WHILE UAI==0 LOOP ' While Input is low, do nothing
      P=x               ' Re-set position to desired position
      G                 ' Continue on with move
RETURN
```

Pulse output on at a given position

```
x=20000                 ' Set Position to trigger on during a move
P=50000                 ' Set commanded position to move to
G                       ' Start Move
WHILE Bt                ' While Moving
      IF @P>x           ' If present position exceeds "x"
            UB=1        ' Set Port B to 5VDC
            WAIT=400    ' wait about 1/10th of a second
            UB=0        ' Set Port B to Zero Volts
            BREAK       ' Break Out of Loop
      ENDIF
LOOP
TWAIT                   ' wait for move to complete
```

Pulse output on at a given position

```
x=20000                 ' Set Position to trigger on during a move
P=50000                 ' Set commanded position to move to
G                       ' Start Move
WHILE Bt                ' While Moving
      IF @P>x           ' If present position exceeds "x"
            UB=1        ' Set Port B to 5VDC
            WAIT=400    ' wait about 1/10th of a second
            UB=0        ' Set Port B to Zero Volts
            BREAK       ' Break Out of Loop
      ENDIF
LOOP
TWAIT                   ' wait for move to complete
```

Various Loops, Trigger Events and Subroutines (continued)

Turn Output on If Real Time Position Error exceed user amount

```
O=0
P=50000                  ' Set commanded position to move to
e=50                     ' Set Desired Trigger Point
E=1000                        ' Set Max allowable Position Error to 1000
UBO                      ' Set Port B as An Output
UB=1                     ' Set it to 5VDC
G                        ' Start Move
WHILE Bt                 ' While Moving
      IF @PE>e           ' If Real Time Position Error Exceed e
            UB=0         ' Set Port B to Zero
      ENDIF
LOOP
```

Stop Motion if Motor Current Exceeds Specified amount

```
O=0
V=1500000                ' Set commanded speed
A=500                    ' Set commanded acceleration
MV                       ' Start Velocity Move
x=50                     ' Set Desired Trigger Point
G                        ' Start Move
WAIT=500                 ' Wait to get past accel Knee
WHILE Bt                 ' While Moving
      IF UIA>x           ' If Motor Current Exceeds 500 milliamps
            OFF          ' Turn off the motor
      ENDIF
LOOP
```

Stop Motion if Motor Current Exceeds Specified amount

```
O=0
V=1500000                ' Set commanded speed
A=500                    ' Set commanded acceleration
MV                       ' Start Velocity Move
x=50                     ' Set Desired Trigger Point
G                        ' Start Move
WAIT=500                 ' Wait to get past accel Knee
WHILE Bt                 ' While Moving
      IF UIA>x           ' If Motor Current Exceeds 500 milliamps
            OFF          ' Turn off the motor
      ENDIF
LOOP
```

Home to Hard Stop

```
'=========================================================
'Example of Home to a Hard Stop
'=========================================================

F=32                ' Enable Fault routine
h=-500              ' This is some home offset from hard stop
                    ' to define home as zero
r=1                 ' Make r a -1 to reverse home direction

v=500000            'some faster home speed off of hard stop
'=========================================================
GOSUB11               ' call some home routine
'=========================================================
END
'=========================================================
C1                  ' Fault Routine
 IF q==123          ' If HOMING
   ZS               ' Clear fault and return to HOMING
RETURNF

C11
 q=123
 PRINT("Homing motor",#13)
 MV
 V=-100000*r        ' HOMING VELOCITY
 A=1000             ' HOMING ACCEL
 AMPS=200           ' Limit motor current while homing
 E=30               ' SETTING ERROR FOR HARD STOP LIMIT
                    ' This may need to be changed
 G                  ' start a velocity mode move
 TWAIT              ' Motor will drop out of TWAIT
                    ' when it errors against hard stop.

 PRINT("Hit hard stop",#13)
 PRINT("Switching to torque mode",#13)
 T=-100*r           ' Set Torque Value
 MT                 ' Mode torque to hold against hard-stop
 WAIT=1000          ' insure gap is closed
 PRINT("Setting position register",#13)
 O=h*r              ' Setting present position to HOME OFFSET value
 X
 AMPS=1000          ' Raise motor current for normal operation
 E=1000             ' setting normal running error limit
 V=v
 MP
 P=0
 PRINT("Moving to Zero",#13)
 G
 TWAIT              ' do nothing until trajectory is complete
 PRINT("Motor is at Home",#13)
 q=0
RETURN
```

Home To Index (3 Examples)

```
END

C30    ' Torque Mode method (Slingshot mode)
' In this method, up to 2 shaft rev's could occur
' It is good for direct drive where single load turn equals single shaft turn
 i=I         ' This clears out the Index register
 T=300       ' Set torque level
 MT          ' start moving
 WHILE i==I LOOP
 T=-1        ' stop fast (dynamic break)
 MP          ' shift to position mode
 P=I+2000    ' go to next index to prevent backup
 G           ' start moving
 TWAIT       ' wait until move is complete
 WAIT=200    ' settling time
 O=0         ' set position to zero
RETURN

C31    ' Relative mode method (move quick, go to last seen)
' In this method, motor could back up almost a full rev
 i=I         ' This clears out the Index register
 D=2020      ' (4040 for a 34 frame or larger)
 V=100000
 A=500
 G           ' move relative just over 1 shaft turn
 TWAIT
 P=I         ' set commanded position to last index seen
 G
 TWAIT
 WAIT=200
 O=0
RETURN

C32    'Velocity mode method (slowly find next index)
'In this method, motor could back up almost a full rev
 i=I
 V=100000
 A=500
 MV
 G
 WHILE i==I LOOP
 X
 P=I
 G
 TWAIT
 WAIT=200
 O=0
RETURN
```

Cycle Time Calculator Subroutine

```
C30         ' This subroutine give time to 4 decimal places from clock ticks
            ' To run it, assign "u" the value in clock ticks to test.
      t=u/4069                'Getting Seconds
      PRINT("Cycle Time=")
      PRINT(t,".")            'Printing Full Seconds
      v=0
      WHILE v<4
            t=t*4069          'Multiply out last whole value
            t=u-t             'Getting remainder
            u=t*10            'multiplying by 10
            t=u/4069          'to get next tenths
            PRINT(t)          'printing NEXT 10ths
            v=v+1
      LOOP
      PRINT(" seconds",#13)
RETURN
```

Long Term Memory Example Storing Error Bits

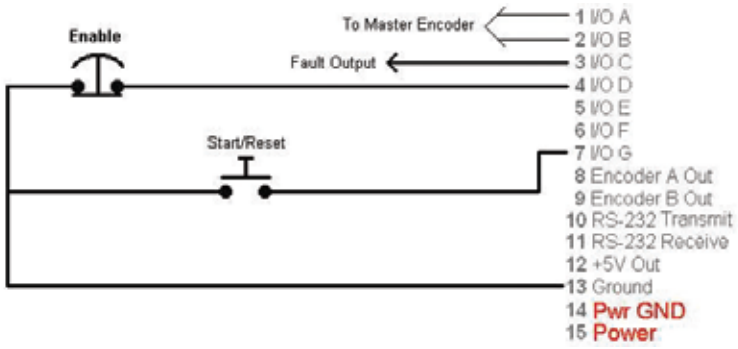
```
'=========================================================
C1
      aa=aa+Be    'assigning Status Bits to 4 consecutive variables
      bb=bb+Bh
      cc=cc+Ba
      dd=dd+Bo
      EPTR=100    'Setting EEPROM Memory Pointer
      VST(aa,4)   'Storing 4 consecutive 32-bit variables into EEPROM
RETURNF
'=========================================================

'=========================================================
C3    'Get latest Status Bit totals
      EPTR=100
      VLD (aa,4)  'Loading 4 consecutive 32-bit variables from EEPROM data
      PRINT("Error Bit Totals",#13)
      PRINT("Be:",aa," (Position Error)",#13)
      PRINT("Bh:",bb," (Over Temperature)",#13)
      PRINT("Ba:",cc," (Peak Over Current)",#13)
      PRINT("Bo:",dd," (Motor OFF)",#13)
RETURN
'=========================================================
```

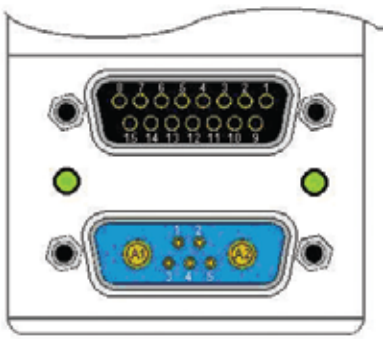
Analog Controlled Variable Speed with Dead-Band and Offset

```
'The following code takes an analog reading,
'places a "no-change" dead band on it and uses it to command
'a velocity to a given trajectory.
'This code does not include accel values.
'The WHILE loop assumes a previous
'move or velocity mode has been issued.

d=10         'this is the dead band in A/D counts of the analog signal
o=512        'this is an offset which allows negative swings in value
m=40         'this is a multiplier used as a span adjust to the speed
w=10         'this is the delay time between reads

WHILE 1==1
       a=UCA-o           'take analog reading of PORT C
       x=a-b             'This is a way to check for changes in the
                         'Pot value and give a dead band

      'The following code waits until the analog value
      'changes by +/- the dead band prior to sending
      'a new SPEED to the motor.
       IF x>d
             V=b*m      'multiplier for Pot input to position
            G
      ELSEIF x<-d
             V=b*m
             G
      ENDIF

      b=a                'update d for prevention of hunting in place
      WAIT=w

LOOP

END
```

Slave Conveyor Application

```
(Written for >= V4.76 Firmware only)
' Contains Feed Hold, Reset, Fault Out
'=============================================================================
'Setting up I/O                          (Note: Port D is being used as a Limit switch Enable)
UGI     ' Set Port G as Input for "Error Restore"
UCO     ' Port C set as output for "Fault Out"
BRKC    ' Make Port C be brake control to be fault output
        ' Port D will be used as a motor enable
'=============================================================================
'Set Interrupt Control
F=97    ' F=1 causes controlled deceleration on fault
        ' F=32 C1 gets called on motor fault
        ' F=64 C2 gets called when Port G gets grounded
'=============================================================================
'Setting variables
m=168           ' used for MFMUL
d=100           ' used for MFDIV
E=20000 ' Set high error count in case Master Encoder is moving when reset
'=============================================================================
'Tuning
KP=50
KI=50           ' usually 1/2 of KP
KD=1200 ' usually 5 to 10 times KP
KV=1000 ' Velocity Feed Forward, 1000 is a good value
F               ' Update/Set PID Filter
'=============================================================================
END
'=============================================================================
C1      ' This subroutine is called on interrupt any time the motor errors out.
  IF q==1               'Note: q can be used to print out or not
        IF Bh   PRINT("Excessive temperature",#13)    ENDIF
        IF Bl   PRINT("Enable Lost",#13)                      ENDIF
        IF Be   PRINT("Excessive position error",#13) ENDIF
  ENDIF
        IF Bw   GOSUB2 ENDIF            ' Checking for Math Wrap Status
        WHILE UDI==1    LOOP            ' DO nothing until enable returns
RETURNF
'=============================================================================
C2      ' This subroutine is called any time the "Go" switch is made
        O=0                     ' Set Position to zero
        MF4                     ' Set to count external encoder at full quadrature
        MFMUL=m         ' Multiply incoming counts by "m"
        MFDIV=d         ' Divide incoming counts by "d"
        MFR                     ' Calculate Ratio
        IF UDI==0               ' If enable returned
                ZS              ' Reset Motor Errors for next move
        ENDIF
        G                       ' Start following at above parameters
        ' Note: If ZS was not issued, G will not take effect
RETURNI ' Return from interrupt called C2
RETURN  ' Return from GOSUB called C2
'=============================================================================
```





16-Position Pre-Select, BCD-Triggered

```
'=========================================================
' 16 Position Preset triggered via 4 bit command and Go input
'=========================================================
'     This program was written around the CBL43-52 cable
'     It is set up to use Ports A, B, C, and D as BCD inputs
'     and Port G as a Go input
'     While Port E is a busy output and Port F is Fault output.
'=========================================================
' Setting up I/O
UAI               ' Ports A through D will be used
UBI               ' As 4 bit BCD Inputs
UCI
UDI
UE=1              ' Port e will be busy output
UEO               ' When set to zero, it is busy
UF=1              ' Port F will be fault output
UFO               ' when set to 1, there is a fault
UGI               ' Go Input
'=========================================================
' Setting up tuning
KP=100
KI=50
KD=1200
F
'=========================================================
GOSUB10           ' call some home routine
UF=0              ' clear fault output bit
UE=0              ' clear busy output bit
'=========================================================
GOSUB35           'get data
WHILE 1
      IF UGI==0         'If Go pressed
            GOSUB31
      ENDIF
LOOP
'=========================================================
END
'=========================================================
C1
                  'place fault code here
RETURN
'=========================================================
C4    'check binary switch, assign it to "d"
' Note, For Version >=4.76, d=U&15 will do the same as all the following code.
b=UBI*2
a=UAI+b
c=UCI*4
d=UDI*8
d=c+d
d=d+a             'd now contains the 4 bit value of the inputs
RETURN
```

16-Posiotion Pre-Select, BCD-Triggered (Continued)

```
C10     'PLACE HOME ROUTINE HERE
        O=0
RETURN
'=========================================================
C31      ' BCD to parameter and move subroutine
        GOSUB4                  ' check BCD Input
        a=d                     ' Accel array index
        v=d+8                   ' Velocity Array Index
        p=d+24          ' Position Array Index
        w=d+80          ' Wait Time Array Index (Or some other move time variable)
        A=aw[a]
        V=al[v]
        P=al[p]
        UE=1                    ' Set busy Output bit
        G
        WHILE Bt                ' While Moving
                IF UGI==1
                        X       'STOP IF GO RELEASED
                ENDIF
        LOOP
        IF UGI==0
        WAIT=w                  ' Use if needing settling time
        ENDIF
        UE=0                    ' Clear Busy Output Bit
        IF Bo                   ' If Fault occurred
                UF=1
                GOSUB1
        ENDIF
        WHILE UGI==0 LOOP  ' WAIT FOR GO TO RELEASE
RETURN
'=========================================================
C35     ' DEFAULT DATA, Change as needed
'ACCEL          VELOCITY                POSITION                WAIT TIME AT END OF MOVE
aw[0]=2500      al[8]=2000000   al[24]=s         aw[80]=200
aw[1]=2500      al[9]=2000000   al[25]=0                 aw[81]=1000
aw[2]=1500      al[10]=2000000  al[26]=s/2               aw[82]=4000
aw[3]=200       al[11]=100000   al[27]=s         aw[83]=1000
aw[4]=3500      al[12]=2000000  al[28]=s/8               aw[84]=500
aw[5]=3500      al[13]=1000000  al[29]=0                 aw[85]=1000
aw[6]=500       al[14]=1000000  al[30]=s/4               aw[86]=100
aw[7]=200       al[15]=1000000  al[31]=0                 aw[87]=1000
aw[8]=500       al[16]=1000000  al[32]=s/10              aw[88]=1000
aw[9]=500       al[17]=1000000  al[33]=s/9               aw[89]=1000
aw[10]=500      al[18]=1000000  al[34]=s/8               aw[90]=1000
aw[11]=500      al[19]=1000000  al[35]=s/7               aw[91]=1000
aw[12]=500      al[20]=1000000  al[36]=s/6               aw[92]=1000
aw[13]=500      al[21]=1000000  al[37]=s/5               aw[93]=1000
aw[14]=500      al[22]=1000000  al[38]=s/4               aw[94]=1000
aw[15]=500      al[23]=1000000  al[39]=s/3               aw[95]=1000
RETURN
```

16-Subroutine Pre-Select, BCD-Triggered

```
'=========================================================
' 16 Subroutine Preset triggered via 4 bit command and Go input
'=========================================================
'     This program was written around the CBL43-52 cable
'     It is set up to use Ports A, B, C, and D as BCD inputs
'     and Port G as a Go input
'     While Port E is a busy output and Port F is Fault output.
'=========================================================
' Setting up I/O
UAI               ' Ports A through D will be used
UBI               ' As 4 bit BCD Inputs
UCI
UDI
UE=1              ' Port e will be busy output
UEO               ' When set to zero, it is busy
UF=1              ' Port F will be fault output
UFO               ' when set to 1, there is a fault
UGI               ' Go Input
'=========================================================
WHILE 1
      IF UGI==0         'If Go pressed
            GOSUB31
      ENDIF
LOOP
'=========================================================
END
'=========================================================
C1
                  'place fault code here
RETURN
'=========================================================
C4    'check binary switch, assign it to "d"
' Note, For Version >=4.76,
' d=U&15 will do the same as all the following code.
b=UBI*2
a=UAI+b
c=UCI*4
d=UDI*8
d=c+d
d=d+a             'd now contains the 4 bit value of the inputs
RETURN
'=========================================================
```

16-Posiotion Pre-Select, BCD-Triggered (Continued)

```
C31                ' BCD to parameter and move subroutine
GOSUB4       ' check BCD Input
   SWITCH d
      CASE 0       GOSUB100    BREAK
      CASE 1       GOSUB101    BREAK
      CASE 2       GOSUB102    BREAK
      CASE 3       GOSUB103    BREAK
      CASE 4       GOSUB104    BREAK
      CASE 5       GOSUB105    BREAK
      CASE 6       GOSUB106    BREAK
      CASE 7       GOSUB107    BREAK
      CASE 8       GOSUB108    BREAK
      CASE 9       GOSUB109    BREAK
      CASE 10      GOSUB110    BREAK
      CASE 11      GOSUB111    BREAK
      CASE 12      GOSUB112    BREAK
      CASE 13      GOSUB113    BREAK
      CASE 14      GOSUB114    BREAK
      CASE 15      GOSUB115    BREAK
   ENDS
RETURN
'=========================================================
C100   'Place Needed Code here for Selected Subroutine
RETURN
C101   'Place Needed Code here for Selected Subroutine
RETURN
C102   'Place Needed Code here for Selected Subroutine
RETURN
C103   'Place Needed Code here for Selected Subroutine
RETURN
C104   'Place Needed Code here for Selected Subroutine
RETURN
C105   'Place Needed Code here for Selected Subroutine
RETURN
C106   'Place Needed Code here for Selected Subroutine
RETURN
C107   'Place Needed Code here for Selected Subroutine
RETURN
C108   'Place Needed Code here for Selected Subroutine
RETURN
C109   'Place Needed Code here for Selected Subroutine
RETURN
C110   'Place Needed Code here for Selected Subroutine
RETURN
C111   'Place Needed Code here for Selected Subroutine
RETURN
C112   'Place Needed Code here for Selected Subroutine
RETURN
C113   'Place Needed Code here for Selected Subroutine
RETURN
C114   'Place Needed Code here for Selected Subroutine
RETURN
C115   'Place Needed Code here for Selected Subroutine
RETURN
```

Record and Playback Example

```
'This program demonstrates recording to non-volatile memory
' for a teach or record and play back example.
' It Records 5 positions, and them plays them back
'Port D is used to both record playback.

OFF                ' TURN MOTOR OFF
 MP                ' Set to Position Mode
 O=0               ' RESET ORIGIN
 UDI               ' Use Port D as Event Trigger
 EPTR=0            ' Reset EEPROM Pointer
 e=0               ' Use the Variable "e" as a counter
 WHILE e<5
      IF UDI==0          ' IF Port D INPUT IS GROUNDED
            a=@P         ' RECORD POSITION IN VARIABLE a
            VST(a,1)     ' STORE 1 VARIABLE IN THE INTERNAL EEPROM
                         ' the Pointer will index by 4 because  "a" is ' a 4 byte number
            e=e+1
      ENDIF
LOOP

C0    'ROUTINE 0 will go to positions stored in long term ram
 WAIT=8000         ' WAIT 2 SECONDS
 P=0               ' GO BACK TO ORIGIN
 V=100000          ' SET VELOCITY
 A=100             ' SET ACCELERATION
 G                 ' GO
 TWAIT             ' WAIT UNTIL MOVE FINISHED
 WAIT=4000         ' WAIT 1 SECOND

 EPTR=0            ' RESET ELECTRONIC POINTER

 c=0               ' INITIALIZE VARIABLE
 WHILE c<5         ' WHILE LOOP
      VLD(b,1)     ' LOAD VALUE STORED IN THE INTERNAL  EEPROM
                   ' AT EPTR=0 INTO VARIABLE b
       P=b         ' SET POSITION
      A=100        ' ACCELERATION
      V=100000     ' VELOCITY
      G            ' GO
      TWAIT        ' WAIT UNTIL MOVE IS FINISHED
      WAIT=4000    ' WAIT FOR 1 SECOND
      c=c+1        ' INCREMENT VARIABLE
 LOOP              ' LOOP

 P=0               ' POSITION TO ZERO
 G                 ' GO
 TWAIT             ' WAIT UNTIL MOVE IS FINISHED

WHILE UDI==1        LOOP       ' WHILE INPUT IS NOT GROUNDED

 GOTO0             ' AND REPEAT playback cycle
END                ' END
```

Expanded I/O Using the DINIO-485

```
'=============================================================================
'This is test and Example Code for DINIO-RS485 I/O Cards
'=============================================================================
GOSUB99       'open RS-485 channel
PRINT("Opening RS-485 Port",#13)
'=============================================================================
GOSUB5 ' Address I/O cards
GOSUB1 ' turn on outputs one at a time
GOSUB2 ' turn off outputs one at a time
GOSUB0 ' reset all I/O blocks to boot-up condition
GOSUB8 ' Set Report commands for full reports
'=============================================================================
WHILE 1
      GOSUB20       'Copy Inputs to Outputs
LOOP
END
'=============================================================================
C99    'Opening RS-485 Channel to give and receive commands
PRINT("Opening RS-485 Port",#13)
      OCHN(RS4,1,N,9600,1,8,C)
RETURN
'=============================================================================
C0     ' This subroutine resets I/O blocks to default boot-up status
      ' Note: I/O blocks will be in the de-addressed state after doing this.
PRINT("Resetting Expanded I/O Card",#13)
      PRINT1("ZA ")
RETURN
'=============================================================================
C1     ' This subroutine turns on output bits 0 though 15
      ' The OS command (Output Set) turns on or "sets" output "n".
PRINT("Turning on Expanded Outputs one at a time",#13)
      x=0
      y=15
      WHILE x<=y
            WAIT=1500
            PRINT1("OS",x,#13)
            x=x+1
      LOOP
      x=0
RETURN
'=============================================================================
C2     ' This subroutine turns off output bits 0 though 15
      ' The OR command (Output Reset) turns off or "Resets" output "n".
PRINT("Turning off Expanded Outputs one at a time",#13)
      x=0
      y=15
      WHILE x<=y
            WAIT=1500
            PRINT1("OR",x,#13)
            x=x+1
      LOOP
      x=0
RETURN
'=============================================================================
```

Expanded I/O Using the DINIO-485 (Continued)

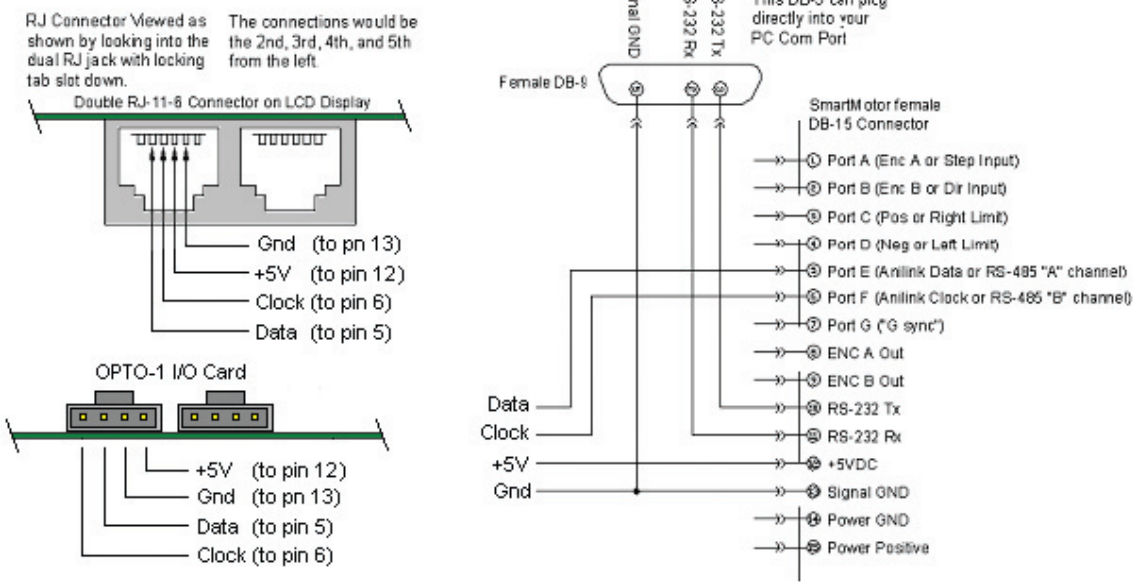
```
'=============================================================================
C5      ' This Subroutine "addresses" the I/O cards.
        PRINT("Addressing Expanded I/O Card",#13)
        ' All I/O cards boot-up to the de-addressed state.
        ' All I/O cards are fixed to ASCII dec. address 244
        PRINT1(#244," ")
RETURN
'=============================================================================
C8      ' This Subroutine Makes Card Report all Bits and Bytes
PRINT("Setting up Expanded I/O card to report all bits and bytes",#13)
        PRINT1(#244,"IOF=127 ")
        ' Syntax:     IOF=number (binary option additive 0-127)

RETURN
'=============================================================================
C9      ' This subroutine sends out a global address.
PRINT("Sending out global address",#13)
        PRINT1(#128," ")
RETURN
'=============================================================================
C10     ' This subroutine is for troubleshooting reports
        ' It is used in conjunction with an Advanced Monitor Status Watch file.
PRINT("Scanning ab[i] for 5 seconds",#13)
        CLK=0
        WHILE CLK<50000
                PRINT1(#244,"Rab[",i,"] ")
                WAIT=50
        LOOP
RETURN
'=============================================================================
C11     ' This Subroutine changes baud rate to 38400 on both
        ' the SmartMotor running this program and all I/O cards attached.
PRINT("Setting RS-485 port to 38400 baud",#13)
        PRINT1(#244,"BAUD38400 ")
        OCHN(RS4,1,N,38400,1,8,C)
RETURN
'=============================================================================
C12     ' This Subroutine changes baud rate to 9600 on both
        ' the SmartMotor running this program and all I/O cards attached.
PRINT("Setting RS-485 port to 9600 baud",#13)
        PRINT1(#244,"BAUD9600 ")
        OCHN(RS4,1,N,9600,1,8,C)
RETURN
'=============================================================================
C20     ' This subroutine copies input status to outputs
PRINT("Outputs mimic inputs while x = 0",#13)
        WHILE x==1
                PRINT1("ab[1]=",ab[0]," ")
                WAIT=20
        LOOP
RETURN
'=============================================================================
```

Expanded I/O Using the Anilink Opto-1 Board

```
'=========================================================================
'Anilink I/O Example
'=========================================================================
'      Port          Default                          Alternate
'      E             AniLink Data               RS-485 A(Half Duplex)
'      F             AniLink Clock                    RS-485 B(Half Duplex)
'=========================================================================

' This code shows how to communicate with Anilink Devices.
' Unlike RS-485 Mode, Ports E and F do not require any code
' to set them up as Anilink.
' Upon any Anilink calls, the ports automatically revert to Anilink I/O.

END

'=========================================================================

C1
' The following code allows toggling of expanded I/O
' via the OPTO1 or DIO100 Anilink I/O boards.
' These are 16 channel boards that are addresses as
' 2 blocks of 8 each.
' In this code, the OPTO1 board was used .
' It was plugged into an industry standard PB-16 board.
' The board was jumpered to base address A
' In this code, I used port A as inputs and port B as outputs.
' However, either address could have been ins or outs.

'=========================================================================

i=DINA0             ' assigns i the entire input port

' The following will yield a 1 if on or 0 if off
' Into the variable i.

i=DINA0&1           ' read channel 1
i=DINA0&2           ' read channel 2
i=DINA0&4           ' read channel 3
i=DINA0&8           ' read channel 4
i=DINA0&16          ' read channel 5
i=DINA0&32          ' read channel 6
i=DINA0&64          ' read channel 7
i=DINA0&128         ' read channel 8

oo=255              ' oo will track output status
' 255 is 8 ones , ones re off and zeros are on.

DOUTB0,oo           ' turns all outputs off (channel 8-16)
```

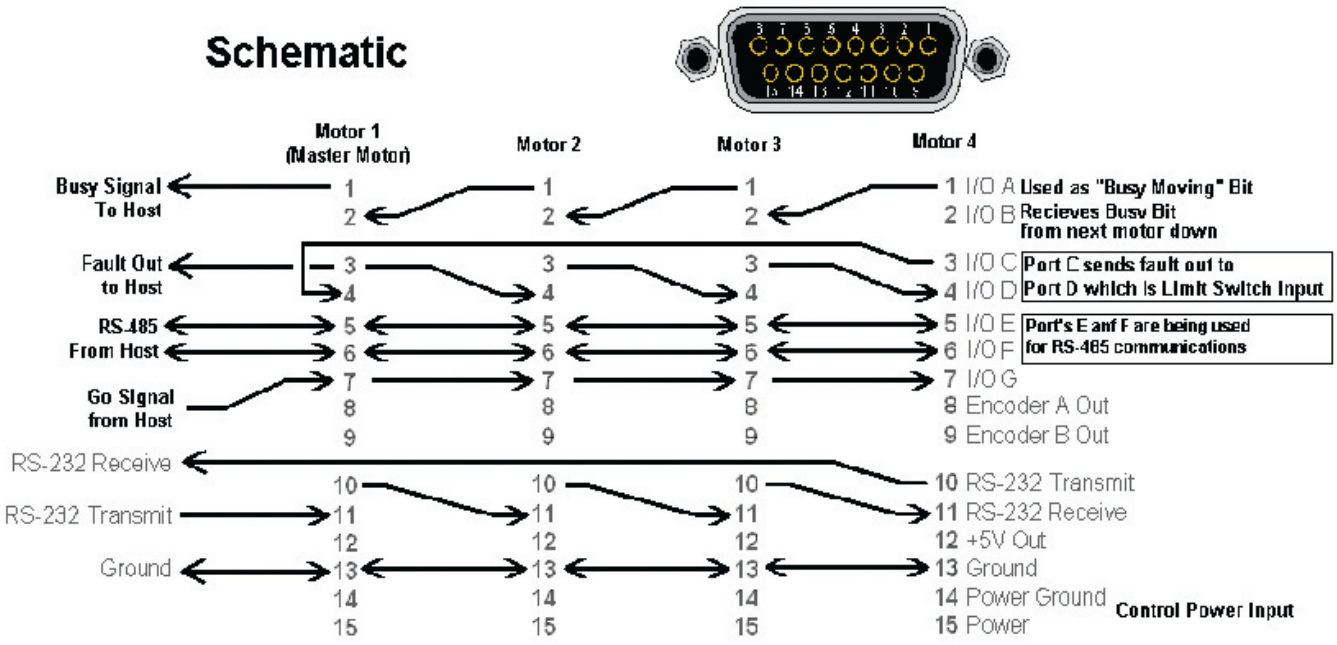
Expanded I/O Anilink (Continued)

```
' For Opto racks, a 1 is off and a 0 is on for output bits.
DOUTB0,1|oo       ' turns on output channel 8
DOUTB0,2|oo       ' turns on output channel 9
DOUTB0,4|oo       ' turns on output channel 10
DOUTB0,8|oo       ' turns on output channel 11
DOUTB0,16|oo      ' turns on output channel 12
DOUTB0,32|oo      ' turns on output channel 13
DOUTB0,64|oo      ' turns on output channel 14
DOUTB0,128|oo     ' turns on output channel 15
DOUTB0,256|oo     ' turns on output channel 16



DOUTB0,254&oo           ' turns off output channel 8
DOUTB0,253&oo           ' turns off output channel 9
DOUTB0,251&oo           ' turns off output channel 10
DOUTB0,247&oo           ' turns off output channel 11
DOUTB0,239&oo           ' turns off output channel 12
DOUTB0,223&oo           ' turns off output channel 13
DOUTB0,191&oo           ' turns off output channel 14
DOUTB0,127&oo           ' turns off output channel 15
DOUTB0,127&oo           ' turns off output channel 15

RETURN
```



Hardware Error Handling Setup-Code: (See next page for Interrupt Subroutines)

```
'=========================================================================
'Error handling Example and RS-485 I/O code example using 4 motors
' with motor 1 as master
'****** This was written for V4.76 or later Firmware *****
' Note, The Motors handle I/O handshaking for errors
' All other code is assumed to come from a Host PLC or PC via comm port
' for this example.
' Motor 1 can be set up as mater though.
'=========================================================================

'=========================================================================
OCHN(RS4,1,N,9600,1,8,C)   ' Open RS-485 comm channel (Ports E and F)
WAIT=20
'=========================================================================
' Setting up I/O:
UCO    ' Set Port C as an Output
UC=0   ' set to zero volts
WAIT=100
ZS     ' Clear errors
' Now all motors will have zero volts on Port D and no limit faults.
'=========================================================================
' Set up 4.76 motors to call C1 on any shaft protection fault
' and to call C2 on Port G grounded
F=96
' Note F-32 sets C1 interrupt
' F-64 sets C2 interrupt.
' F parameters are bit additive.
'=========================================================================
END                                          ' mark END of program..
'=========================================================================
```



Hardware Error Handling Setup-Code (Continued)

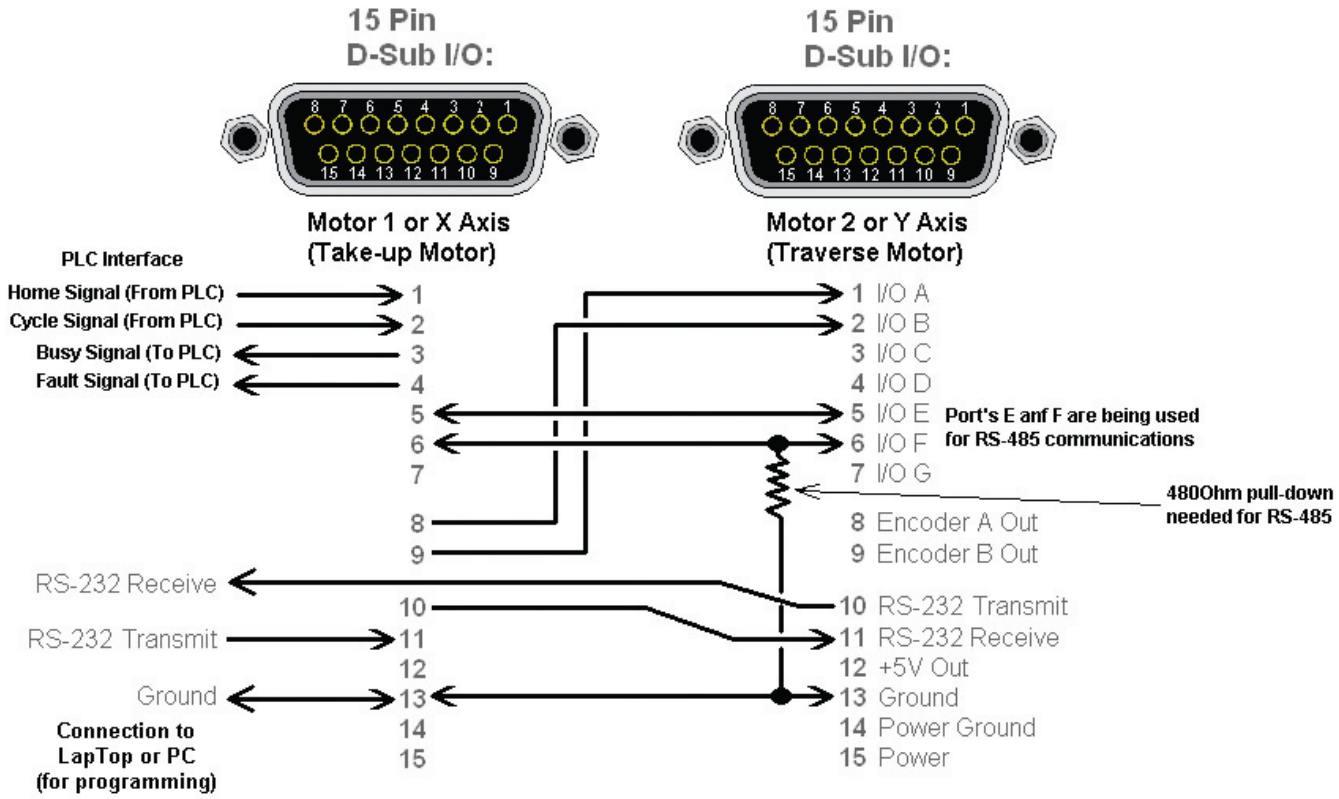
```
C1      ' Fault interrupt.
' All motors  jump to this subroutine on interrupt on
' any shaft protection  fault.

UDI     'Disable Port D as negative limit input

' IF  one motor faults, it will cause a chain reaction
' of all motors faulting on loss-of-limit
' Remember that Port C is an output tied to Port-D limit-switch
' input of next motor down for each motor.

       ' The Limits are active high asserted, i.e.
        ' when they go to 5VDC,the motor will fault.
       ' The 5Kohm resistor internally causes fail-safe
       ' protection on loss of connection.

       UC=1          'Echo error to next motor
       WAIT=40       ' wait FOR 40/4069 sec before reset output C
       UC=0          ' Set Port C to zero volts to check connection
       WAIT=300

       IF Be         'If Position Error

                     ' place code here for position errors in this motor

       ELSEIF Bh     'If Over Temp Error

                     ' place code here for thermal errors in this motor

       ELSEIF UDI==1       ' Wait =300 time delay enables this to work.

                     ' place code here for errors passed from other motors

       ENDIF

       IF ADDR==1    ' If Master Motor

              UC=1   ' Hold Fault Out until Host clears the fault

              ' place additional code here if master needed to handle faults.
       ENDIF

       UDM           ' Reset Port D as Limit switch input

RETURNF       ' RETURNF is for >=V4.76 motors calling C1 on Fault Interrupt

'================================================================================
C2            ' Called on Interrupt because Port G went Low
       F=32           ' Disable Port G interrupt while moving
       UA=0           ' Set busy bit to host
       WHILE Bt LOOP' Maintain Port A low until move is complete
       IF UBI==0      ' Check for other motor busy bit from Port B
              UA=0    ' and echo it out to next motor
              WHILE UBI==0 LOOP
       ENDIF
       UA=1
RETURNI       ' RETURNI is for >=V4.76 motors calling C2 on Input interrupt
```

Traverse and Take-Up Winder Application

```
'-------------------------------------------------------------------------------------------
' This is an example traverse and take-up program.
' NOTE: THIS WAS WRITTEN FOR VERSION 4.76 MOTORS
' 4.76 MOTORS DEFAULT TO MODE TORQUE BRAKE ON ANY MOTOR PROTECTION FAULT.
' SOFTWARE LIMITS (ONLY IN VERSION 4.76) ARE BEING USED AS TRAVERSE REVERSAL POINTS.
' WHEN THE SOFTWARE LIMITS ARE REACHED, THE TRAVERSE MOTOR WILL
' DYNAMICALLY BRAKE TO A STOP.
' IT WILL THEN SIT THERE FOR "d" COUNTS OF TAKE-UP MOTOR BEFORE CONTINUING
' ON IN THE OTHER DIRECTION.
' THE TRAVERSE MOTOR WILL CONTINUE TRAVERSING back and forth UNTIL THE
' TAKE-UP MOTOR IS DONE.
' THE TAKE-UP MOTOR IS MASTER. IT SPINS ONE TURN BEFORE TELLING
' THE TRAVERSE TO START.
' AT THE END OF THE CYCLE, BOTH MOTORS ARE LEFT IN SERVO LOCK.
' PORT B OF MOTOR 1 (TAKE-UP MOTOR) IS THE CYCLE START INPUT
' PORT A OF MOTOR 1 IS THE HOME SIGNAL.
' TWO OUTPUTS ARE USED ONE FOR BUSY (MOTOR 1 PORT C), THE OTHER
' FOR FAULT (MOTOR 1 PORT D).
' PORT G IS AVAILABLE ON BOTH MOTORS AS AN INTERRUPT CALL TO C2
' PORTS C AND D ARE OPEN FOR USE ON MOTOR 2 (TAKE-UP MOTOR)
' THEY COULD BE USED AS HARD TRAVEL LIMITS, BUT KEEP IN MIND THAT SOFT LIMITS
' ARE BEING USED
' AND THE TAKE-UP MOTOR HOMES TO A HARD STOP, SO NO LIMIT SWITCHES ARE EVER NEEDED

' This program is written in such a way that the same program gets downloaded to both motors.
' The schematic below show how to wire the motors together for this program.
' It shows the use of RS-485 communications for purposes of error handling and Master/
  slave operation.
' The program can be modified to use only RS-232 if the I/O are needed.
'-------------------------------------------------------------------------------------------
' Motor 1 (X Axis) is the Take-Up Axis
'
' Motor 2 (Y Axis) is the Traverse Axis and will follow Motor 1 (Take-up axis
```

Traverse and Take-Up Winder Application (Continued)

```
'--------------------------------------------------------------------------------
'Schematic)
'
```




```
'--------------------------------------------------------------------------------


'--------------------------------------------------------------------------------
'    Setting Addresses, Motors use RS-232 comms to determine which one is motor 1.
q=0
WAIT=1000
PRINT(#128,"q=1 ",#13)          'each motor is print out of the serial port
WAIT=4000
IF q==0                         'only motor 1 can print to motor 2
      SADDR1
ELSEIF q==1                     ' motor 2 can not print to motor 1
      SADDR2
ENDIF
ECHO                            ' set to echo mode for RS-232 communications
'--------------------------------------------------------------------------------
```

Traverse and Take-Up Winder Application (Continued)

```
'-------------------------------------------------------------------------------------------
F=104                          ' This sets up error code interrupts for both motors since
they are 4.76 motors
' Note: On version 4.76 firmware
'     F=32 causes interrupt call to subroutine 1 on any motor fault
'     F=64 causes interrupt call to subroutine 2 on port G getting grounded
'     F=1 reverses shaft rotation
'     F=8 will zero KI term at end of move

' This program does not use C2 for anything though.
'-------------------------------------------------------------------------------------------

'-------------------------------------------------------------------------------------------
'    Setting up RS-485 Communications for both motors

OCHN(RS4,1,N,19200,1,8,C)                        ' open the RS-485 port
WAIT=4000                                  ' Wait a bit
'-------------------------------------------------------------------------------------------



'-------------------------------------------------------------------------------------------
'    Setting up Tuning for both motors

KP=100
KI=50
KD=1200
KV=1000
KA=1000
F
E=1000
AMPS=950
'-------------------------------------------------------------------------------------------
IF ADDR==1         ' Setting up I/O for the Take-Up or X axis

' Setting up Inputs
      UAI          ' Port A being used as Home input
      UBI          ' Port B being used to start the cycle
' Setting up Outputs
      UCO          ' Port C being used as Busy or "Moving" output
      UC=0         ' Port C will be GND when motors are not busy and at 5VDC when they are
      UDO          ' Port D being used as Fault Output
      UD=0         ' Port D will be Grounded when there is no fault and 5VDC if either
motor faults

' Note:  Setting a Port to zero makes it zero volts. Setting it to 1 makes it 5VDC

      ZS                ' This clears any boot-up motor errors if you have Version 4.76 or
higher
'-------------------------------------------------------------------------------------------
```

Traverse and Take-Up Winder Application (Continued)

```
'-------------------------------------------------------------------------------

ELSEIF ADDR==2     ' Setting up I/O for the Traverse or Y axis

      UCI                    ' Motor 2 needs limits disabled if it is a 4.76 motor
      UDI
      ZS                     ' This clears any boot-up motor errors if you have Version
4.76 or higher

ENDIF

'-------------------------------------------------------------------------------
'-------------------------------------------------------------------------------
'    Setting up Variables for both motors

'----------------------      Speeds and
accelerations-----------------------------------------------------
'-----------------------------------      Take-Up
Motor-------------------------------------------------------------
s=100000                           ' Velocity for TAKE-UP MOTOR
a=200                              '' Acceleration for TAKE-UP MOTOR
e=2500                                  '' Position Error for TAKE-UP MOTOR
'-----------------------------------      Traverse
Motor-------------------------------------------------------------
ss=100000                          ' Velocity for TRAVERSE MOTOR
aa=200                             '' Acceleration for TRAVERSE MOTOR
ee=2500                            '' Position Error for TRAVERSE MOTOR
'-------------------------------------------------------------------------------
'-------------------------------------------------------------------------------
'----------------------      Operating Variables (distances and counts)
'-----------------------------------      Take-Up
Motor-------------------------------------------------------------
k=2000                                  ' Encoder counts per turn of take-up spool
n=100                              ' Number of turns desired on spool
'-----------------------------------      Traverse
Motor-------------------------------------------------------------
mm=5                               '' gear ratio multiplier
dd=10                                          '' gear ratio divisor
SLN=100                            '' Low side soft limit for traverse motor
SLP=7000                           '' High side soft limit for traverse motor
d=225                              ' dwell for traverse motor (amount of master encoder
counts traverse motor dwells before reversing)
'-------------------------------------------------------------------------------
'-------------------------------------------------------------------------------
IF ADDR!=1 GOTO999 ENDIF                ' If this is not motor 1, END
WAIT=4000
'-------------------------------------------------------------------------------
'-------------------------------------------------------------------------------
GOSUB100                                ' HOME MOTORS

GOSUB10                                 ' set base speed and accel for each motor
'-------------------------------------------------------------------------------
```

Traverse and Take-Up Winder Application (Continued)

```
'--------------------------------------------------------------------------------------
' Main Program Loop
WHILE 1
      IF UAI==0               ' PLC/Reset-switch told me to home
            UC=1              ' Set busy bit to PLC/Light Stack
            GOSUB100
            WHILE UAI==0 LOOP ' Latch up catch
            UC=0              ' Clear  busy bit to PLC
      ENDIF

      WAIT=100

      IF UBI==0               ' PLC/Start-switch told me to run the cycle
            UC=1              ' Set busy bit to PLC/Light Stack
            GOSUB50
            WHILE UBI==0 LOOP ' Latch up catch
            UC=0              ' Clear  busy bit to PLC
      ENDIF

LOOP
'--------------------------------------------------------------------------------------
STACK
C999
END
'--------------------------------------------------------------------------------------
C1    ' Error handler deals with any motor protection fault during run-time

      IF ADDR==2
            IF q==4                 ' If traverse motor errors on soft limits while
traversing

                  j=CTR+d           ' Get end-of-dwell position
                  ZS                ' Clear soft limit error
                  MFMUL=-MFMUL      ' negate Mode Follow Ratio
                  MFR               ' Re-instate MFR
                  WHILE CTR<j  LOOP ' Wait until dwell is complete
                  G                 ' Start traversing the other way

            ELSEIF q==5       ' If motor errors against hard stop while in home routine

                  PRINT("Traverse hit hard stop while homing",#13)
                  ZS    'clear the error
            ELSE  ' UNEXPECTED ERROR OCCURED
                  STACK
                  END
            ENDIF

      ELSEIF ADDR==1
            PRINT("Take-Up motor error",#13)
      ENDIF                     ' Turn off motor 1

RETURNF
'--------------------------------------------------------------------------------------
```

Traverse and Take-Up Winder Application (Continued)

```
'-------------------------------------------------------------------------------------------
' Write code in here for any interrupt call needs when port G is grounded.
C2
       IF ADDR==1
              PRINT("Take-Up Motor G pin grounded",#13)
       ELSEIF ADDR==2
              PRINT("Traverse Motor G pin grounded",#13)
       ENDIF
RETURNI
'-------------------------------------------------------------------------------------------
'-------------------------------------------------------------------------------------------
C3     ' This subroutine Commands  Y axis move to y and verifies it via RS-485 returns
       f=1               ' trap variable which will be re-set by Y axis motor via RS-232
       PRINT1(#130,"MP P=",y," GOSUB7 ")
       ' Sub 7 in Motor-2  updates  f  to zero when finished moving
       WHILE f==1 LOOP   ' Error check routine
            f=0          ' reset trap variable
RETURN
'-------------------------------------------------------------------------------------------
'-------------------------------------------------------------------------------------------
' The following subroutine is used by motor 2 only and is called from motor 1 as needed:
C7                            ' Motor 2 move subroutine
       G                      ' Motor 1 called C7 and set the position to go to
       TWAIT
       PRINT1(#129,"f=0 ")    ' tell motor 1, motor 2 is finished
RETURN
'-------------------------------------------------------------------------------------------
C10    ' This code sets up move parameters for both motors

' NOTE: If it is not called or an equivalent code is not executed
'      the motors will note move (speed defaults to zero)

       PRINT1(#130,"MP",#13)          ' Set Y axis to position mode
       WAIT=20
       PRINT1(#130,"V=",ss,#13)       ' Set Y velocity
       WAIT=20
       PRINT1(#130,"A=",aa,#13)       ' Set Y accel
       WAIT=20
       PRINT1(#129,#13)               ' Re-address motor 1(X axis)
                                      ' This is done for convenience sake

       V=s                            ' Set X axis Velocity
       A=a                            ' Set X axis accel
       MP                             ' Set to position mode
RETURN
'-------------------------------------------------------------------------------------------
```

Traverse and Take-Up Winder Application (Continued)

```
'-------------------------------------------------------------------------------------
C50
      PRINT(#140,#13)    'DE-ADRESSING MOTOR 2
y=bb                     ' Set start point for Traverse motor
      P=@P G TWAIT       ' Hold position on Take-up motor
WAIT=200
O=0
GOSUB3             '      Send traverse motor to start point
PRINT("Traverse Motor in Start Positon",#13)
            V=s
      A=a                ' Set Take-up motor commanded position to number of wraps to wind

      P=k*n              ' Set Take-up motor commanded position to number of wraps to wind


PRINT("Starting Take-Up Motor",#13)
      G                  ' Start winding

      WHILE @P<k LOOP                ' DWELL AT START
      PRINT1(#130,"GOSUB51  ")       ' Run Traverse subroutine in Traverse motor
      WAIT=20
      PRINT1(#130,"SLE  ")           ' ENABLE SOFTWARE LIMITS IN MOTOR 2

      TWAIT

            PRINT1(#130,"q=0  ")
      WAIT=20
      PRINT1(#130,"SLD  ")           ' DISABLE SOFTWARE LIMITS IN MOTOR 2
      WAIT=20
      PRINT("Finished wrapping ",n," turns.",#13)
RETURN
'-------------------------------------------------------------------------------------
'-------------------------------------------------------------------------------------
C51   ' THIS SUBROUTINE SETS UP GEAR RATIO FOR TRAVERSE
      q=5
      MF4
      MFMUL=mm
      MFDIV=dd
MFR G
      WHILE q==5 LOOP          ' MOTOR 1 WILL UPDATE q AT END OF TAKE-UP OPERATION
      MP P=@P G TWAIT          ' HOLD POSITION
RETURN
'-------------------------------------------------------------------------------------
```

Traverse and Take-Up Winder Application (Continued)

```
'-------------------------------------------------------------------------------------------
C100
      PRINT(#140,#13)          'DE-ADRESSING MOTOR 2
      PRINT("HOME",#13)
      P=@P  G                  'hold in place
      TWAIT
      O=0   'set to zero
      PRINT("TAKE UP AT HOME",#13)  'NOTE: the word UP is a valid command.
                               'Motor 2 was de-addressed to prevent it from
                               'uploading while homing.
                               'else you would have a bunch of zeros pop up.
      ' I could have changed UP to Up or up and it would have worked.

      PRINT1(#130,"GOSUB101 ")' SEND TRAVERSE MOTOR TO HOME
      f=1
      WHILE f==1 LOOP
      PRINT("TRAVERSE AT HOME",#13)
      PRINT(#128,#13)          'sending universal address (Addressing motor 2)
RETURN
'-------------------------------------------------------------------------------------------
'-------------------------------------------------------------------------------------------
C101  'HOME ROUTINE FOR TRAVERSE AXIS
      q=5                ' allows C1 to know motor is in home routine
      MV                 ' moving towards the hard stop in velocity mode
      V=-100000          ' set home speed
      A=100              ' set home accel
      AMPS=85            ' Limiting motor torque to protect traverse slide from damage
      E=50               ' Setting a low error count so it errors as soon as it stops
      G                  ' start moving towards the stop
      TWAIT              ' wait until trajectory bit clears
      'Note:  you will error out against the hard stop which clears the trajectory bit.
      ' Version 4.76 will jump to C1 on error automatically
      T=-60              ' go into torque mode to hold against hard stop (in case of bounce)
      MT
      WAIT=1000          ' let motor settle against hard stop
      O=-100             ' set position to some offset
      OFF
      q=0
      E=ee               ' set error limit
      AMPS=1020          ' set allowable motor power
      MP
      P=0                ' go to home position
      G
      TWAIT
      PRINT1(#129,"f=0 ")      ' tell motor 1, motor 2 is finished
RETURN
```

The following pages cover various schematics to help interface electrically to SmartMotors™

SmartMotor™ Connections:

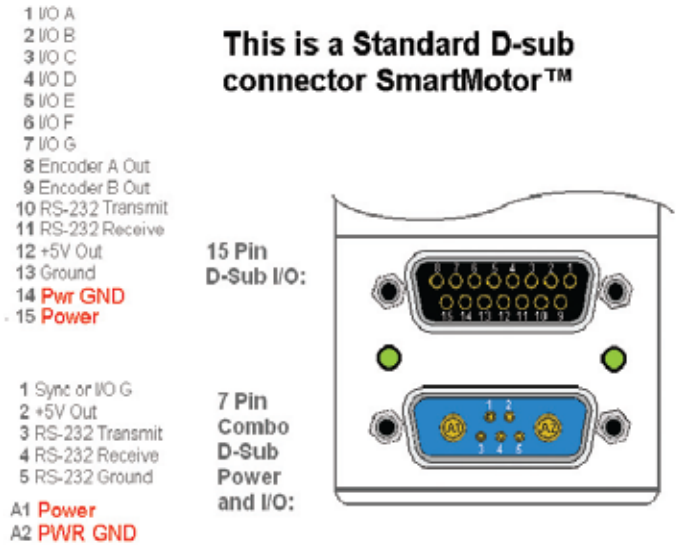
Note: All Connections on the 7W2 combo connectors are available on the DB-15 connector as well. They are a direct internal

Connection on all standard D-Sub connector motors.

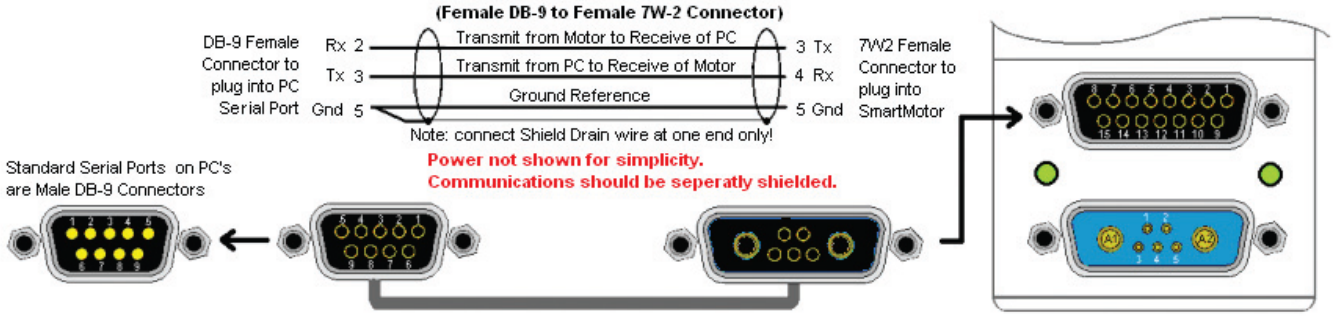
If 1 or less I/O Points and RS-232 Port Connection is all that is needed, then the 7W2 connector is the only connection needed to operate the motor.

Any additional I/O features are found on the DB-15 connector.

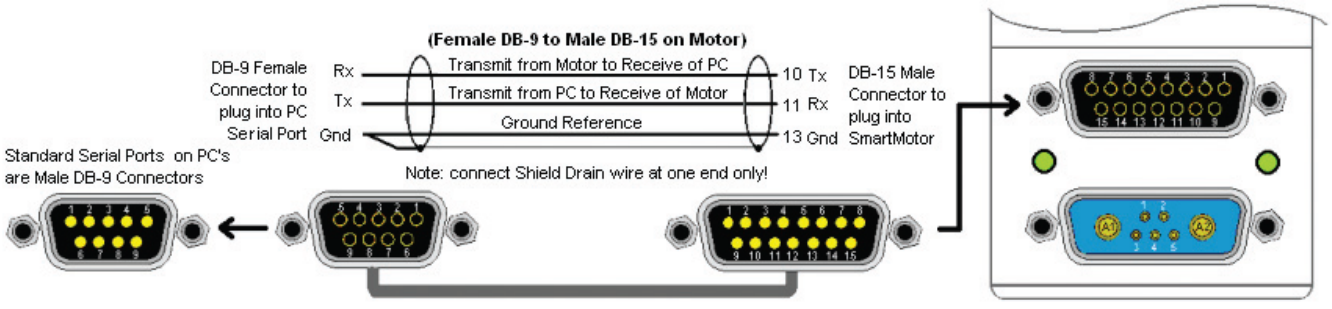
Please refer to other documentation on programmable operation of I/O.



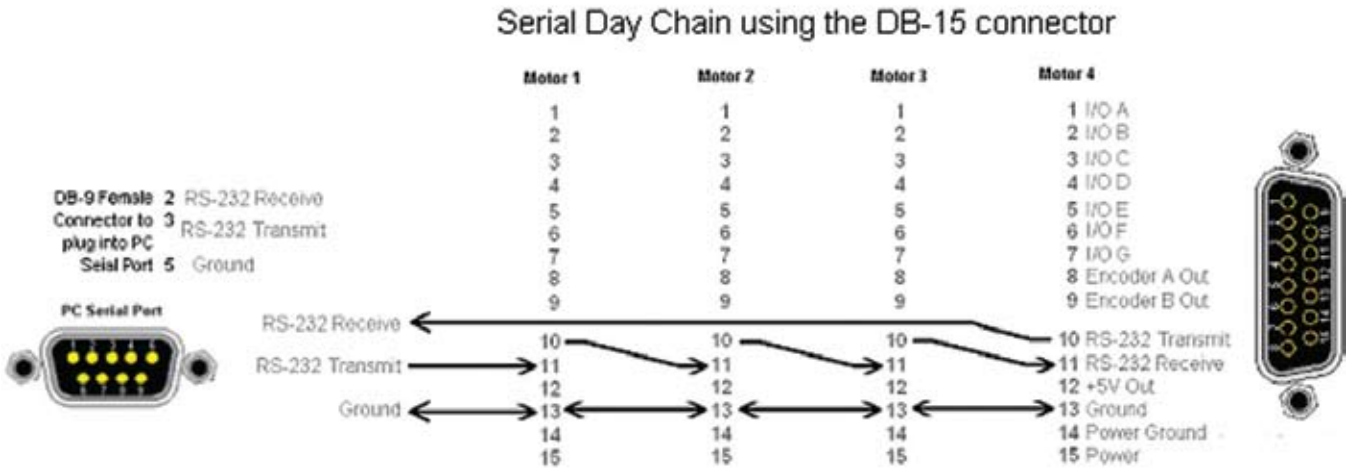
RS-232 Programming cable schematic to communicate with one motor via the main 7W2 Connector:



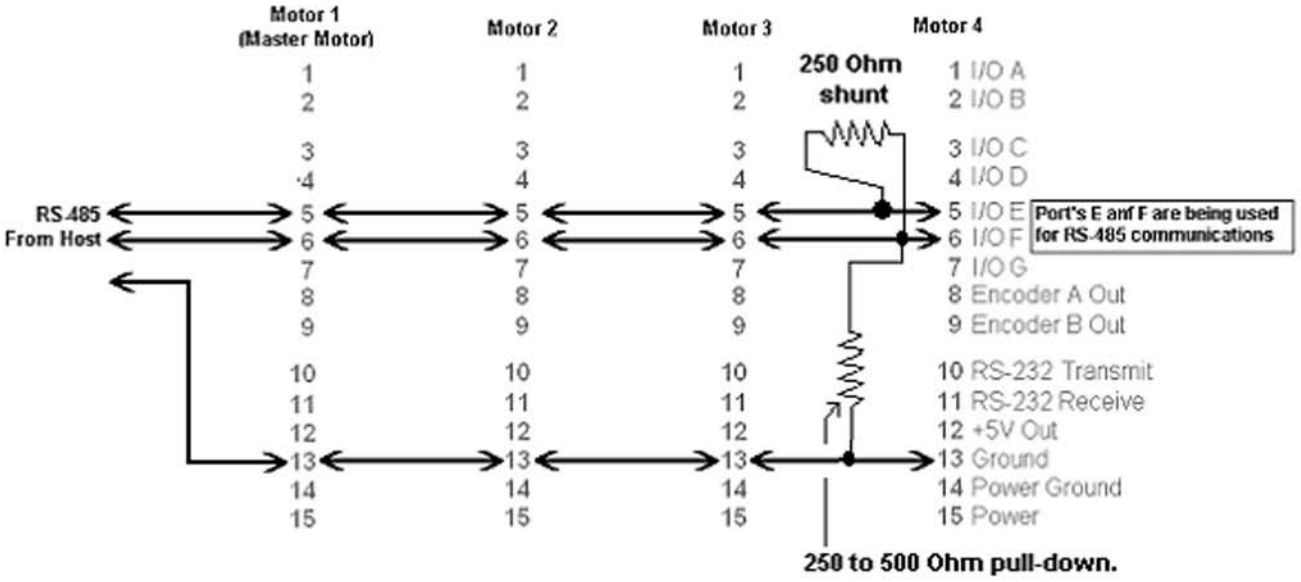
RS-232 Programming cable schematic to communicate with one motor via the DB-15 Connector:



RS-232 Serial Daisy-Chain cable to communicate to multiple motors via the DB-15 Connector



RS-485 Parallel Daisy-Chain to communicate to multiple motors via the DB-15 Connector



Note: All RS-485 networks require >= 400mVolt differential bias to work properly. The SmartMotor employs 5Kohm pull-ups on all I/O pins, as a result, the pull down resistor shown is needed for proper operation. The shunt resistor may be required of the distance to the last motor is significantly long.

Port E is referred to as the "A" or "Positive" side of the RS-485 bus while Port F is referred to as the "B" or "Negative" side of the bus.

Ideal cable would be dual twisted pair with shield. The shield should be tied to ground at one point only. The ideal point would be the host, or if no host, the first motor. The shield should not be used as a ground reference or be tied to any more than one ground point. This would cause noise to be induced into the bus.

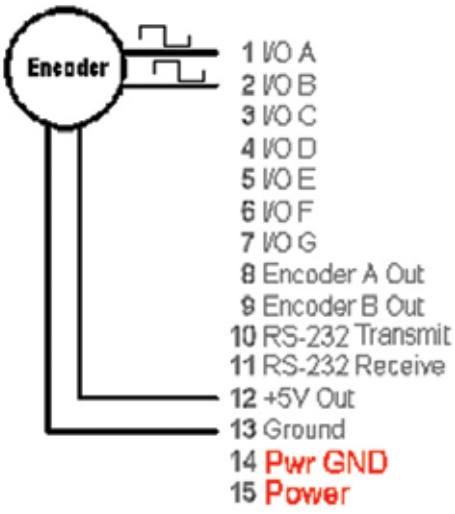
If RS485ISO adapters are used, they make use of the Main RS-232 port. As a result, the bus needs to be powered from a separate 5VDC source. If a motor is used as that source, opto-isolation would be defeated. The best means to power it would be from the host.

Connecting an external encoder for External closed-loop operation or for electronic gearing:

Note: The schematic shown is for an Encoder that can be powered from the internal 5VDC supply of the motor. The motor can only supply a maximum of 150mAmps.

Ensure the chosen encoder does not draw too much current.

If the external encoder has differential outputs, such as A(+), A(-) and B(+), B(-), then just wire the plus connections to Ports A and B inputs respectively. Maximum input frequency is 2MHerts.



Example Code to initiate Encoder Following:

```
MF4             ' Interpolate incoming pulses in full
          ' quadrature
MFMUL=4   ' Multiply incoming counts by 4
MFDIV=7   ' Divide incoming counts by 7
MFR             'Calculate Mode-Follow-Ratio
G         ' Begin following at that ratio
```

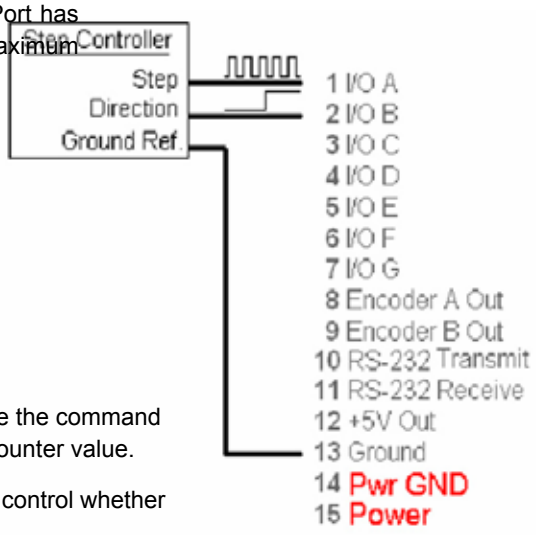
Side Note:
Ports A and B can also be used as a high speed input counter. Issue the command "MF0", and the counter will be set to zero. The command "RCTR" will report counter value. The value will be total full quadrature counts received since MF0 was issued. This method can be used to trigger events in one motor based off of positions from another motor.

Example:

```
MF0       ' Set counter to zero
WHILE CTR<20000 LOOP ' Loop until count exceeds 20000
V=100000
A=100
MV
G  ' Start moving in velocity mode
```

Connection to a PLC or stepper card output for running in Step Mode:

Note: Schematic shown is for sinking-output stepper controllers. Each I/O Port has a 5Kohm Pull-up resistor that the step controller would need to pull down. Maximum step input frequency is 2MHerts.



Example Code to initiate Step/Direction Following:

```
MS        ' Set motor to Mode-Step
MFMUL=4   ' Multiply incoming counts by 4
MFDIV=7   ' Divide incoming counts by 7
MSR             ' Calculate Mode-Step-Ratio
G         ' Begin following at that ratio
```

Side Note:
Port A can also be used as a high speed input counter for parts counting Issue the command "MS0", and the counter will be set to zero The command "RCTR" will report counter value.

The value will be total step pulses received since MS0 was issued. Port B will control whether it counts up or down from zero.

Connecting 2 motors for Electronic Gearing:

Note:

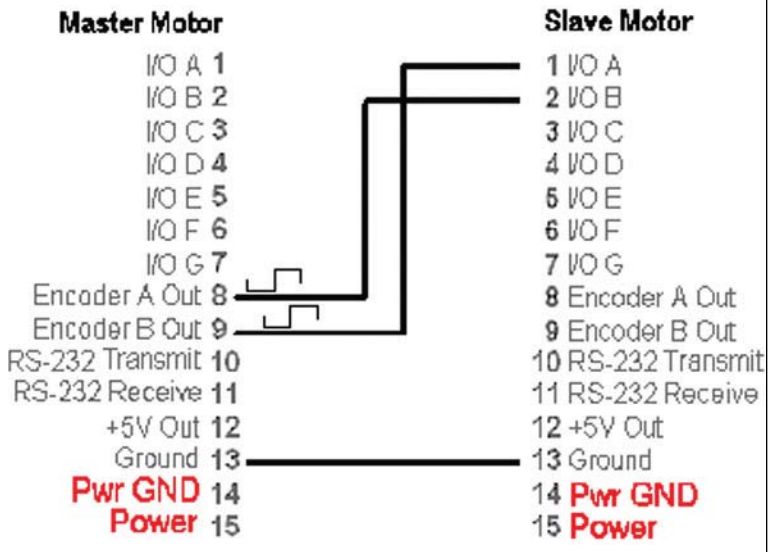
Pin's 8 and 9 are the inverted outputs of the motor's internal encoder. This is why A-out is connected to B in and vice versa.

Otherwise the slave motor would spin the opposite direction.

Software code still allows for reversal if hardware change is not desired.

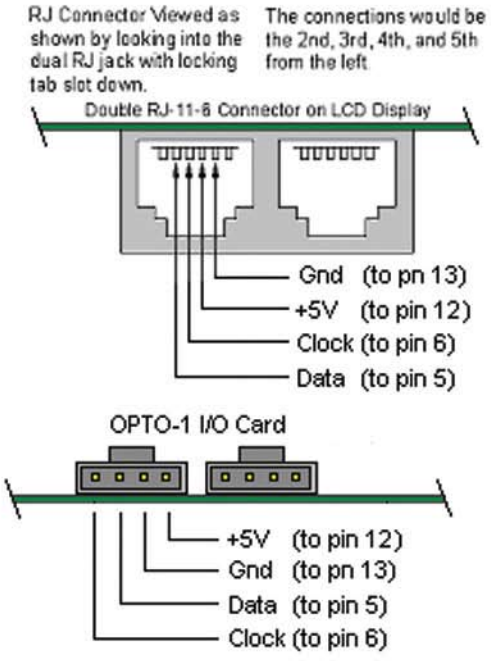
Ground reference is needed for proper operation.

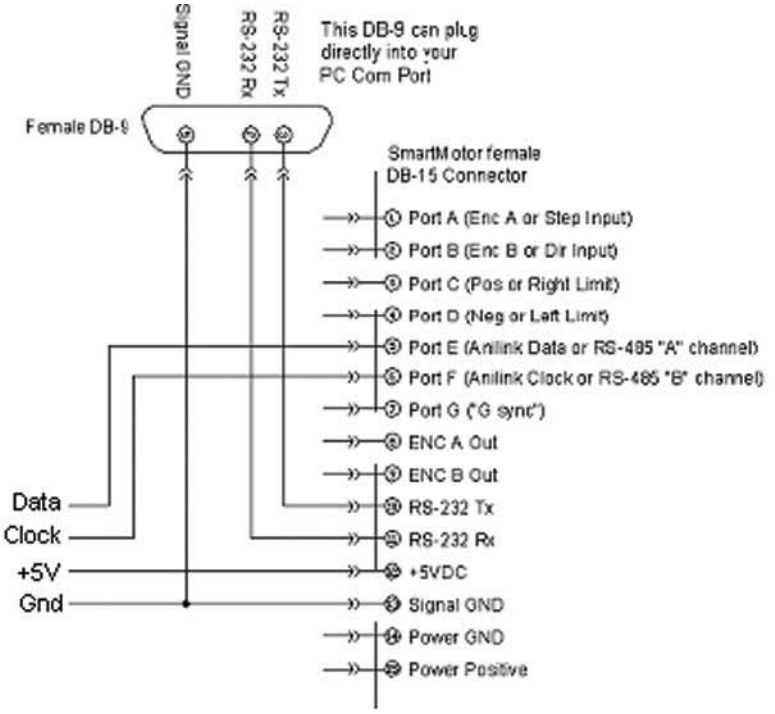
See previous page for example program code.



Connection to Anilink Devices

(Both LCD RJ Connection and OPTO-1 Molex connection shown)





Note: Maximum distance for Anilink devices is 4 feet.

RS-232 communications is shown for clarity.

RS-485 communications is not available when Anilink Devices are used.

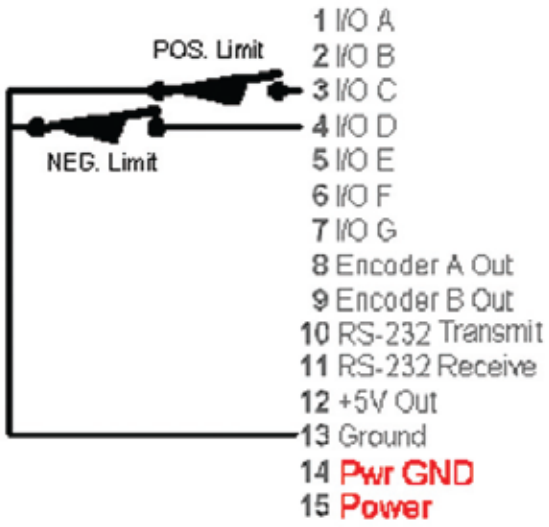
Typical Limit Switch Inputs:

As Shown, they are Active-High Asserted. This means when the limit switches open or the connection breaks, the motor will stop.

This is because each input on the motor has a 5Kohm pull-up to 5VDC.

In Versions of firmware **prior to** 4.76, this requires the LIMH command to make them active-high. Version 4.76 and later default to active high.

Only 5VDC sensors or dry-contact switches can be used. If solid state sensors are used, they should be NPN or "Sinking" type outputs to pull down the 5Kohm pull-ups that are in the motor.

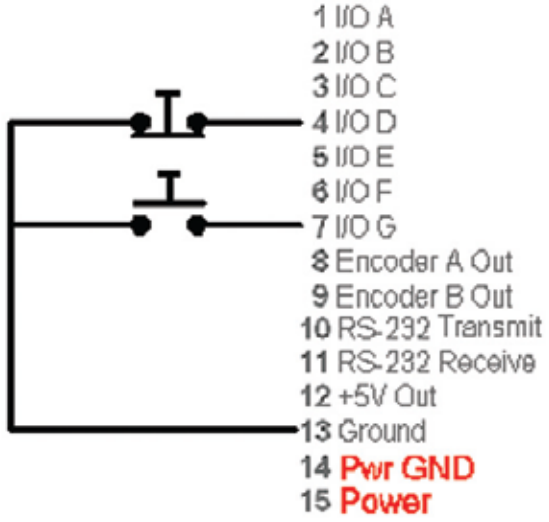


Simple Start/Stop Switch Input

Simple Start/Stop set-up using the "G-sync" function of Port G for start/go and Limit switch input to Stop.

Note: By default, When Port G is grounded, the processor interprets it as a "G" command being issued.

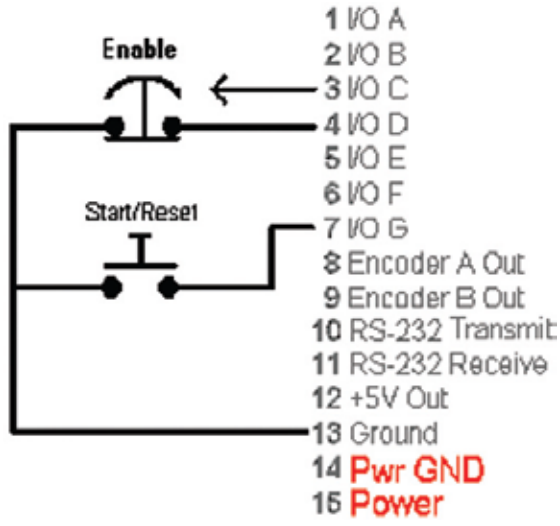
Port D limit input was used for this example. Either Port C or port D could have been used to stop motion as long as the respective limit input is enabled and active-high.



Start-E Stop Input

Similar to above, with Limit used as an E-Stop Enable.

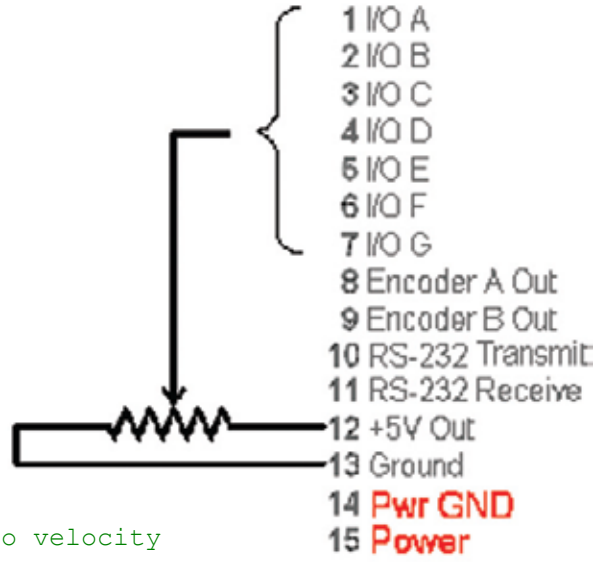
Note: By default, When Port G is grounded, the processor interprets it as a "G" command being issued.



Analog Input to a SmartMotor:

Any of the 7 I/O Ports can be read as a 10 Bit analog input. The voltage range must be from zero to 5VDC (0-1023 value returned).

Example: a=UEA would assign the analog value of Port E to the variable "a". If a standard Potentiometer or linear adjustable resistor is used, it should be 1Kohm or less in value to give the best response. This is because the motor's internal 5K Pull-ups on each I/O port pin must be "pulled down" via the external analog input. It is best to use shielded cable to keep noise levels to a minimum.



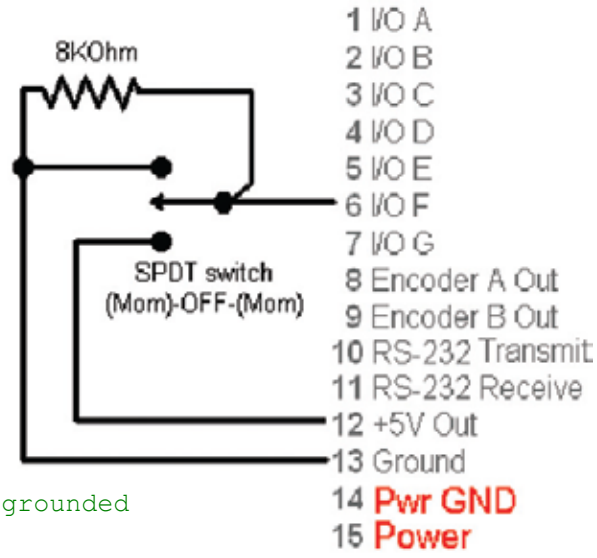
Example code:

```
MV              'Set to Velocity Mode
A=100           'Set acceleration
WHILE 1         'While forever
   V=UAA*1000   'Assign Port-A analog value to velocity
   G            'Make new velocity take effect
LOOP
```

Note: I/O Ports can be read as analog inputs even while being used as Digital Inputs or Outputs as seen in this next example:

Obtaining 2 functions out of One Input:

In this example, a spring-to-center toggle switch and an 8KOhm pull-down resistor is connected to Port F. Combined with the 5K-internal Pull-up, the port will normally read a logic high when read as a digital input. When read as an analog input, it will read about 600 (0-1023 for 0-5VDC). If the switch is swung to ground, it will read a digital zero. When swung to 5VDC, it will read ~1023 on the analog scale. This means a single Input pin could be used as a Jog Up/Down switch. An added benefit is that if the connector comes off the motor, you will know it because the input will always read high and it's analog value will be ~1023.



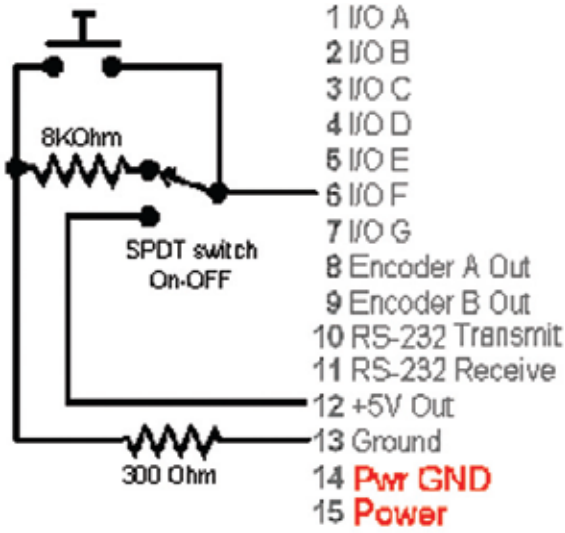
Example Code:

```
WHILE 1
   IF UFI==0          '     If Port F is hard grounded
         PRINT("Pushbutton pressed",#13)
         WHILE UFI==0 LOOP
   ELSEIF UFA<600     '     If Port F is biased between 5 and 0 volts
               PRINT("Switch in upper position",#13)
         WHILE UFA<600 LOOP
   ELSE               '      If Port F is hard pulled to 5VDC
         PRINT("Switch in lower position",#13)
         WHILE UFA>700 LOOP
   ENDIF
LOOP
```

Note: The above toggle switch could have been 2 separate momentary pushbuttons as well., But if someone were to press both at once, the 5VDC supply would be shorted out. To avoid this, an extra resistor could be employed on the ground line. (See next Example)

Push-Button and Toggle Switch into single input:

This is a twist to the last example. This time a simple on-of switch is being used with the same 8K Pull-up. Normally the Port will read about 600 as an Analog value. If the Momentary pushbutton is pressed, it will read zero and logic level zero. If the switch is opened, it will read If the Toggle switch is spring return to the shown position, then it is possible to detect if the connection came loose.



The 300 Ohm resistor is to prevent a 5VDC supply short in case the pushbutton and toggle were pressed at the same time. (See Note on previous example)

Example Code:

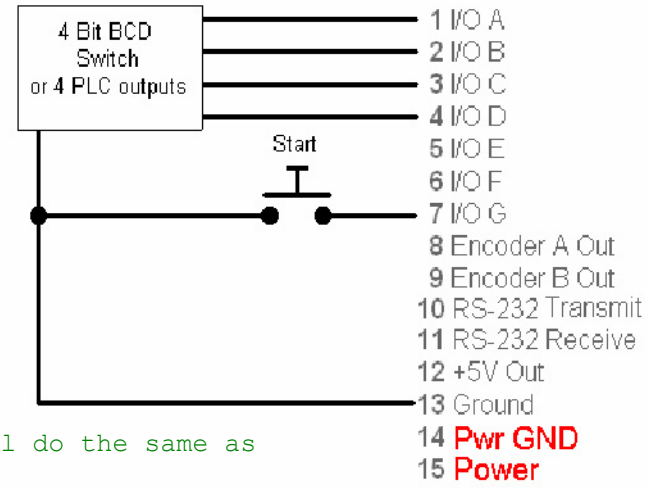
```
WHILE 1
IF UFI==0       'if Port F is hard grounded
   PRINT("Pushbutton pressed",#13)
   WHILE UFI==0 LOOP
ELSEIF UFA<600 ' If Port F is biased between 5 and 0 volts
   PRINT("Switch in upper position",#13)
   WHILE UFA<600 LOOP
ELSE           ' If Port F is hard pulled to 5VDC
   PRINT("Switch in lower position",#13)
   WHILE UFA>700 LOOP
ENDIF
LOOP
```

Binary (4 Bit BCD) input control:

By using 4 inputs and a binary switch or 4 PLC outputs, up to 16 functions can be achieved.

This could be 16 subroutine calls, 16 pre-set speeds or positions or any of the above combinations. Via programming capabilities, It could be a sequencing operation even controlled from another SmartMotor.

A common use for the other 2 I/O pins is to use them as Busy and Fault outputs back to a PLC.

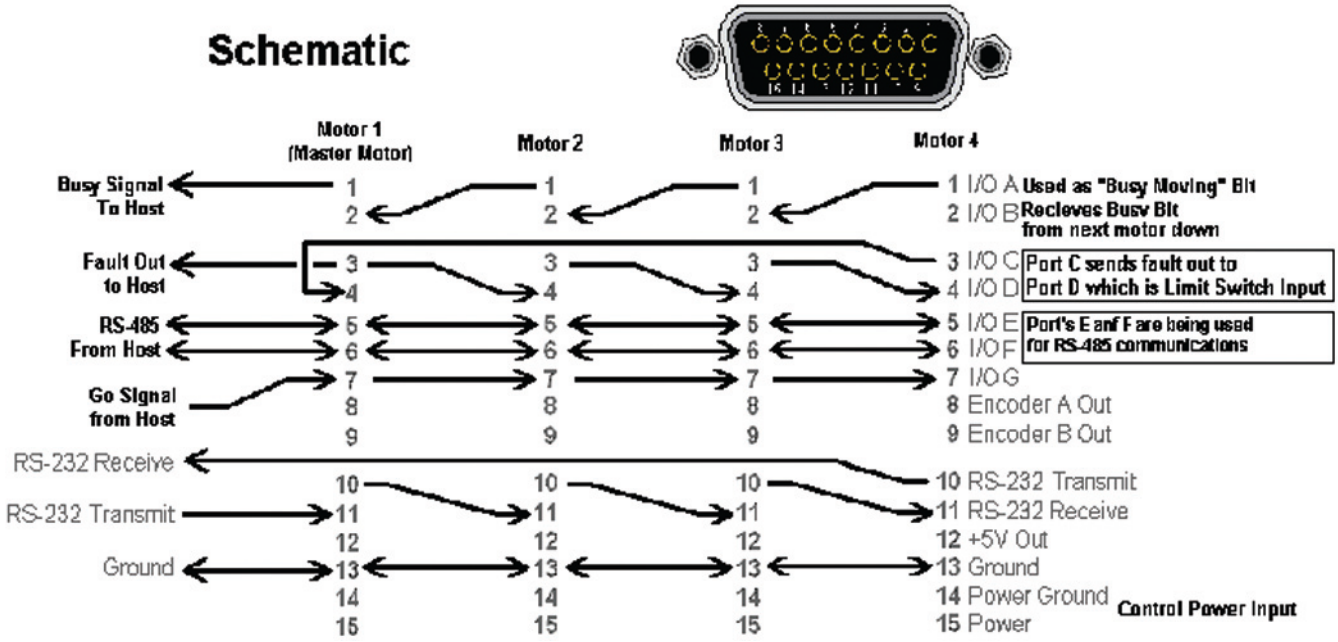


Example Code:

```
C4 'check binary switch, assign it to "d"
   ' Note, For Versions >=4.76, d=U&511 will do the same as
   ' all the following code.
b=UBI*2
a=UAI+b
c=UCI*4
d=UDI*8
d=c+d
   'd now contains the 4 bit value of the inputs
RETURN
```

Cascade I/O Fault Control

The following is a more complex arrangement of I/O handling that includes hard wired Fault protection, RS-232 and RS-485 communications.



Notice how I/O is cascaded through from one motor to the next.
By using Port G's function as a G-sync line, we can trigger moves on all motors at one time.

If a PLC or Host controller does the triggering, it will need conformation back that the motors are busy.

Port A and B are being used for this. Each motor sends a busy signal out of port A and into Port B of the next motor up the chain. When all motors are completed, the first motor will signaCascade I/OI the PLC.

Example:

```
'Slave motor code for sending Busy signal up the chain:
WHILE 1           ' standing by to detect a move
      IF Bt       ' If port G was grounded, the Bt bit will go from 0 to 1
                  ' indicating Busy Trajectory
            UAI=1             ' Set Port A to 1
            TWAIT             ' Wait for move to complete
            WHILE UBI LOOP    ' Wait for Port B to go to 0, as fed from next
                  ' motor down.  (last motor should omit this code)
            UAI=0 ' Set Port A to zero to trigger next motor up
      ENDIF
LOOP
```

By utilizing the BRKC function of V4.15 and later firmware, Port C becomes a fault output. By feeding this into Port D with Port D being defaulted as a limit switch, if any one motor faults, it will trigger a chain reaction and stop all motors immediately.

The RS-232 chain can be used for PLC, HMI, or PC connection while the RS-485 communications bus can be used between each motor to allow isolated control from the host.

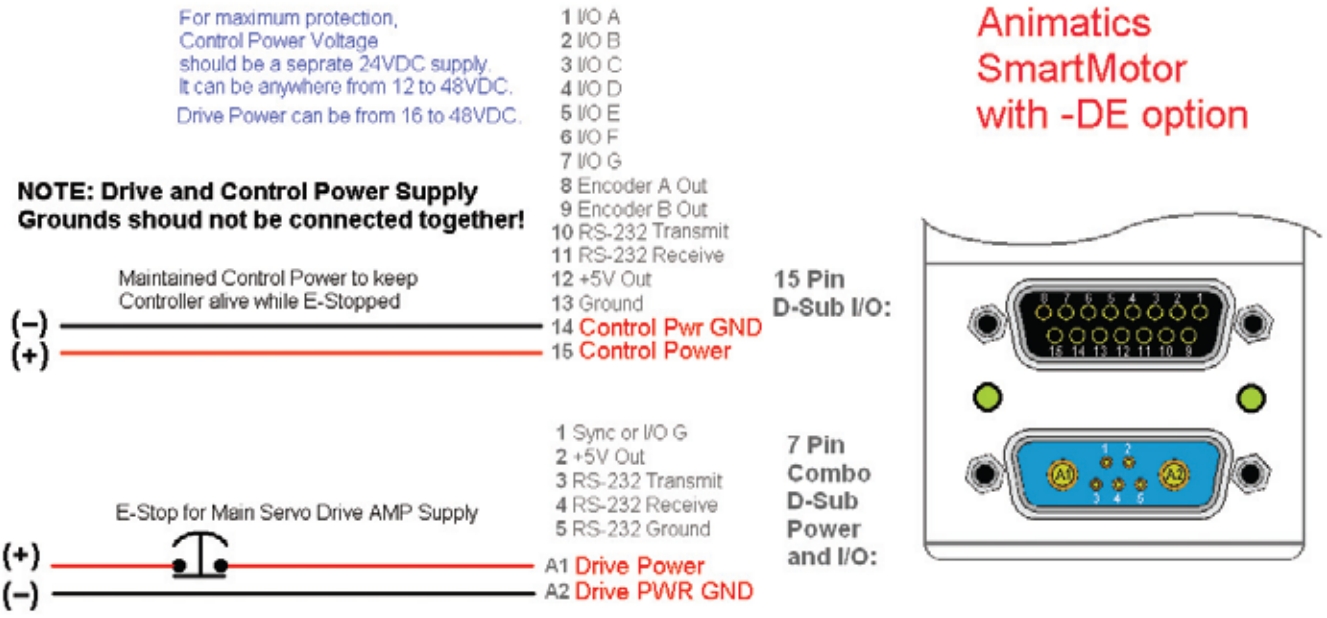
DE (Drive Enable) Option

All SM23XX and SM34XX motors come with a "-DE" option.
This option separates Pin 15 of the DB-15 connector from Pin A1 of the 7W2 connector allowing separate power supplies to run the controller and Drive amplifier sections of the motor.

The connection between A2 (Power Ground) and pin 14 are maintained though. This means that if separate power supplies are used, they cannot have their grounds tied together outside of the motor. To do so would cause a serious ground loop with drive currents being placed on the controller ground.

The reason for the –DE option is 3-fold.:

1. It allows the controller to b "kept alive" under an E-Stop condition so re-homing is not necessary.
2. The controller is protected from current surges caused by the drive amplifier (or other motors on the same supply)
3. Better protection against Back-EMF voltage spikes. (they will not reach the controller).
4. The drive Amplifier can take much higher spikes than the controller.



Modes of Operation:

MP Set Controller for Position Mode, pending a G

MV Set Controller for Velocity Mode, pending a G

MT Immediately set controller to Torque Mode

MFx Immediately set Controller to Mode-Follow (Electronic Gearing to follow External Encoder) where x = 1,2,4

MS Immediately Set Controller to Step-Mode (Step and Direction Input)

MC Initialize Mode Cam awaiting a G

MTB Mode Torque Brake (Dynamically brake)

Note: MTB applies to PLS firmware only.

Position Commands:

A Value of absolute acceleration

A=*expression* Set Acceleration for Position and Velocity Modes (unsigned 16-bit value)

V Value of buffered requested velocity

V=*expression* Set required Velocity for Position and Velocit Modes (Signed 32-bit value)

D Value of buffered relative position, phase offset, and [Dwell (F=16, F=128)]

D=*expression* Set Relative Distance for Relative Position Mode, (signed 32-bit value)
Set Phase Offset Distance In Electronic Gearing,
Set Dwell in Cam Mode (See F-Function Comands for more)

P Value of buffered target position

P=*expression* Set buffered target Position for Absolute Position Mode (signed 32-bit value)

G Start buffered motion profile or trajectory;
Initiate Mode Follow Ratio in Electronic Gearing
Initiate Phase Offset Move in Electronic Gearing
Initiate all buffered move profile values such as Velocity, Acceleration, etc.

TWAIT Halt program command execution until trajectory completed

X Decelerate to a stop using present buffered acceleration value

S Decelerate to a stop using firmware fixed high rate of deceleration

I Index Pulse Position of Internal Encoder at last point of capture

O=*expression* Reset Origin in Position Register (to a signed 32-bit value)

E Value of Maximum Allowable Following Error in Encoder Counts

E=*expression* Set Maximum allowable Position Error (unsigned 0-300000 max)

AMPS Value of the power limit

AMPS=*expression* Set PWM Power limit, 0 to 1023 represents 0-100% allowable PWM

OFF Turn off Drive Stage of SmartMotor™ servo

T Value of Commanded Torque (Open-Loop Commanded PWM to Drive Stage)

T=*expression* Set torque magnitude and direction, (signed values of -1023 to 1023)

External Encoder Motion Commands:

MF0 Reset secondary encoder counter to zero

MS0 Reset secondary encoder to zero

MFDIV Value of Mode Follow Ratio Divisor

MFDIV=*expression*
Set Ratio divisor value (16-bit signed value)

MFMUL Value of Mode Follow Ratio Multiplier

MFMUL=*expression*
Set Ratio Multiplier value (16-bit signed value)

MSR Calculate New Buffered Step Mode Ratio values from MFMUL and MFDIV, pending a G

MFR Calculate New Buffered Follow Mode Ratio values from MFMUL and MFDIV, pending a G

MCx Initialize Cam Mode awaiting a G, where x =2, 4, or 8 times result

CI Mode Cam Table Index Value, (present Cam table pointer)

BASE=*expression*
Cam Mode periodic encoder base where SIZE < BASE <= 32767

SIZE=*expression*
Number of Array Points in Cam Table for Cam Mode operation where 2 <= SIZE <= 100

CTR External Encoder Position Register Value

CTR=0 Set External Encoder Register to Zero

ENC0 Close Position Loop on Internal Encoder (Default State)

ENC1 Close Position Loop on External Encoder (Optional State)

Program Flow Structures:

Nesting program flow structure is permitted (6 levels deep)

IF *expression ...*	Beginning of "IF" code block
ELSEIF *expression*	Next "IF" test case, extended only if "IF" above is false
ELSE	Remaining "IF" test case
ENDIF	End of IF, ELSEIF, and ELSE code block
SWITCH *expression ...*	ENDS SWITCH code block (resultant value of expression stored in the variable zzz)
CASE *value*	Individual SWITCH test case
BREAK	Jump to exit of WHILE or SWITCH
DEFAULT	If all SWITCH test cases false
ENDS	End of SWITCH code block
WHILE *expression*	WHILE code block
LOOP	End of WHILE code block
RUN	Executed the stored EEPROM program, from the beginning
!	Suspend program execution until ANY Incoming Communications is received
RUN?	Stop program executing at point of command until RUN command is received
BREAK	Jump to exit of WHILE or SWITCH
GOSUBnnn	Execute subroutine at statement label nnn, and then return to next statement
GOTOnnn	Jump to program statement label nnn
C#	Program Location Label for GOT and GOSUB calls, C0 to C999
RETURN	Return subroutine to program address on the stack (just below GOSUB call)
WAIT=*expression*	Suspend program execution for given number of PID cycles, ~4069cyles = 1sec
Z	Perform Software CPU Reset of SmartMotor™
END	Stop Program Code Execution

User Program EEPROM Read/Write Commands:

LOAD	Receive and Store into EEPROM a compiled SmartMotor™ program file
UPLOAD	Upload User Program to host terminal
UP	Upload Compiled User Program and Header file to host terminal
RCKS	Report Compiled User Program EEPROM checksum

Variable/Data Storage EEPROM Read/ Write Commands:

EPTR=*expression*	Set user EEPROM memory pointer where n is 0 to 32255
VLD *(variable, number)*	Load contiguous user variables from user EEPROM number is the number of variables to be loaded
VST *(variable, number)*	Store contiguous user variables into user EEPROM, number is the number of variables to be stored

Variables/System-Variables:

@P	Value of measured position
@PE	Value of measured position error
@V	Value of measured velocity
a thru **z**	32-bit Signed Integer value variables
aa thru **zz**	32-bit Signed Integer value variables, (shares memory location with array variables)
aaa thru **zzz**	32-bit Signed Integer value variables, (shares memory location with array variables)
ab[0] thru **ab[200]**	8-bit Signed Integer Array Variables, (shares memory location with aa-zz, and aaa-zzz)
aw[0] thru **aw[100]**	16-bit Signed Integer Array Variables, (shares memory location with aa-zz, and aaa-zzz)
al[0] thru **al[50]**	32-bit Signed Integer Array Variables, (shares memory location with aa- zz, and aaa-zzz)

System State Flags:

The follow 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 and are ideal for Fault Detection and control when operating via Serial Communications.

RW Report Status Word (See Individual Status Bits Below)

Bt =1 if trajectory in progress,	Bit: 0, value: 1
Br =1 if Positive Travel Limit Exceeded	Bit: 1, value: 2
Bl =1 if negative limit crash occurred	Bit: 2, value: 4
Bi =1 if new index report available	Bit: 3, value: 8
Bw =1 if Wrap Around occurred	Bit: 4, value: 16
Be =1 if position error occurred	Bit: 5, value: 32
Bh =1 if Exceeded Thermal Limit	Bit: 6, value: 64
Bo =1 if Drive Stage is OFF	Bit: 7, value: 128
Bx =1 if Drive Stage is OFF	Bit: 8, value: 256
Bp =1 if on Positive Travel Limit,	Bit: 9, value: 512
Bm =1 if on Negative Travel Limit ,	Bit:10, value:1024
Bd =1 if math overflow occurred,	Bit: 11 value:2048
Bu =1 if user array index error occurred,	Bit: 12, value:4096
Bs =1 if syntax error occurred,	Bit: 13, value:8192
Ba =1 if over current occurred,	Bit: 14, value:16384
Bk =1 if EEPROM I/O error occurred,	Bit :15, value:32768

Other Status Bit Flags:

Bb =1 if comm parity error occurred

Bc =1 if comm buffer overflow occurred

Bf =1 if comm framing error occurred

By =1 if step direction change overrun occurred (V4.40 only)

Reset System State Flag:

Za	Reset (Ba) over-amps flag bit
Zb	Reset (Bb) comm parity flag bit
Zc	Reset (Bc) comm overflow flag bit
Zd	Reset (Bd) math overflow flag bit
Zf	Reset (Bf) comm framing flag bit
Zl	Reset (Bl) negative limit crash flag bit
Zr	Reset (Br) positive limit crash flag bit
Zs	Reset (Bs) syntax error flag bit
Zu	Reset (Bu) array index error flag bit
Zw	Reset (Bw) position wrap flag bit
Zy	Reset (By) step dir bit (V4.40 only)
ZS	Reset all reset-able system flags

AniLink™ I/O Commands:

AIN{port}{input} value of 8-bit analog input

AOUT{port},{exp.8} output byte to analog port

DIN{port}{channel } AniLink digital input byte

DOUT{port}{channel},{exp.8} output digital byte value to AniLink

{port} is A, B, C, D, E, F, G, or H

{input} is 1, 2, 3, or 4

{channel} is 0 thru 63

{exp.8} i is 8 bit value: 0 thru 255

Report to Host Commands:

R{user variable} report user variable to host
User variable is a thru z, aa thru zz, aaa thru zzz, ab[0] thru ab[200], aw[0] thru aw[100], or al[0]

R{X} report to host various commands (where {x} can be position commands, variables, system state flags, communicationcommands, etc.)

Motor Over Travel Limit Commands:

UCP	Assign pin C to positive limit switch input, (default state)
	Note: Disable with either or UCO or UCI
UDM	Assign pin D as negative limit switch input, (default state)
	Note: Disable with either or UDO or UDI
	The following apply to non-PLS firmware only:
LIMD	Makes Limits Directional.
	A new occurrence of either limit still halts the motor. A move begun on a limit is only allowed to move in the opposite direction of the limit.
LIMN	Makes Limits Non-directional. This is the defaultfor <=V4.40c.
LIMH	Set Limits to active High. Motor will fault when limit goes high.
LIML	Set Limits active-Low, Motor will fault when limit goes low This is the default for <=V4.40c.
	The following apply to PLS firmware only:
SLD	Disable software limits
	(always disable prior to changing values below)
SLP=*expression*	Assign value in encoder counts to Programmable Positive Software Travel Limit
SLN=*expression*	Assign value in encoder counts to Programmable Negative Software Travel Limit
SLE	Enable software limits

Motor I/O Commands:

UG Assign pin G to synchronous "GO" (default State)

U{pin}O Assign pin to be an output

U{pin}=*expression* Set pin output latch to 0 or 1
where 0 is zero volts, and 1 is 5VDC

U{pin}I Assign pin to be a general input

var=U{pin}I Assign digital value of pin to variable (returns a 0 or 1)

v**ar=U{pins}A** Assign 10-bit analog value of a pin to a variable

In all above cases:

{pin} is **A, B, C, D, E, F,** or **G**

exp. is **0** or **1**

var is any variable **a** thru **z**,
aa thru **zz**,
aaa thru **zzz**,
ab[0] thru **ab[200]**,
aw[0] thru **aw[100]**, or
al[0] thru **al[100]**

Examples: UAI, UBO, c=UDI, UE=0, f=UGA

Brake Commands:

BRKENG Engage the brake (requires hardware brake)

BRKRLS Release the brake (requires hardware brake)

BRKSRV Engage break whenever servo off
(requires hardware brake)

BRKTRJ Engage break when trajectory is not running
(requires hardware brake)

BRKC* Re-direct brake control from internal brake pin to Port C
(V4.15b or higher firmware only)
UCO must be issued prior to this command
Automatic Functionality follows BRKTRJ or BRKSRV commands as listed above

BRKG* Re-direct brake control from internal brake pin to Port G
(V4.15b or higher firmware only)
UGO must be issued prior to this command
Automatic Functionality follows BRKTRJ or BRKSRV commands as listed above

BRKI* Redirect brake control to internal brake control pin
(Default state)
(V4.15b or higher firmware only)

*Note: Not available with 440c firmware (i.e. SM2315D and SM2315DT)

PID Filter Commands:

PIDx Set PID update rate where x=1, 2, 4, or 8
(default is PID1)

KA Value of buffered acceleration feed forward gain coefficient

KA=*expression* Set buffered acceleration feed forward gain coefficient

KD Value of buffered derivative gain coefficient

KD=*expression* Set buffered PID derivative gain coefficient

KG Value of buffered PID constant coefficient

KG=*expression* Set buffered PID constant coefficient

KI Value of buffered integral gain coefficient

KI=*expression* Set buffered PID integral gain coefficient

KL Value of buffered PID integral term contribution limit

KL=*expression* Set buffered PID integral limit

KP Value of buffered PID proportional gain coefficient

KP=*expression* Set buffered PID proportional gain coefficient

KS Value of buffered KS differential sample rate coefficient

KS=*expression* Set buffered PID differential sample rate

KV Value of buffered velocity feed forward gain coefficient

KV=*expression* Set buffered PID velocity feed forward gain

F Apply buffered filter coefficients to PID calculation

Communication Commands:

ADDR=exp set motor address between 0 and 99

BAUDX Set baud rate to (x=2400, 4800, 9600, 19200, 38400 bps)

SADDR*address*
Set SmartMotor™ address, were address = 0 to 115

ECHO Set Channel 0 (Main RS-232 Port) to Echo all received data to the transmit line

ECHO_OFF Turn off Echo function above, Default state is ECHO_OFF

SILENT Prohibit outgoing messages onto Channel 0, (RS-232) originating from within user program

SILENT1 Prohibit outgoing messages onto Channel 1, (RS-485) originating from within user program

SLEEP Prohibit SmartMotor executing received Channel 0 commands except WAKE

SLEEP1 Prohibit SmartMotor executing received Channel 1 commands except WAKE1

TALK Permit outgoing messages originating from within user program to Channel 0 (RS-232)

TALK1 Permit outgoing messages originating from within user program to Channel 1 (RS-485)

WAKE Permit any Received Commands on Channel 0 (RS-232) to be executed

WAKE1 Permit any Received Commands on Channel 1 (RS-232) to be executed

OCHN (type,comm,parity,bit rate,stop bits,data bits, specification)

Open a communications channel where:

type is RS2 or RS4

comm is either primary channel 0 or secondary channel 1

baudrate 2400, 4800, 9600, 19200, or 38400 (bps)

data bits is 8

stop bits is 1

specification is C (for command) or D (for data)

PRINT( ) Print to Com Ch. 0 (RS-232 main channel)

PRINT1( ) Print to Com Ch 1 (RS-485)

PRINT{port}( ) Print to AniLink™ port choice of A thru H

Note: See Animatics User's Guide for more information on PRINT commands

GETCHR Capture next character from Com Ch.0 input buffer

GETCHR1 Capture next character from Com Ch.1 input buffer

LEN Number of characters presently in Com Ch.0 buffer

LEN1 Number of characters presently in Com Ch.1 buffer

Note: See Animatics User's Guide for more information on PRINT commands

---

Miscellaneous Commands:

CLK Value of SmartMotor™ clock

CLK=*expression* Set/Reset value of SmartMotor™ clock

TEMP Value of Slave processor unit temperature in degrees C.
(It must be assigned to a variable to be reported.)

UIA Value of motor current in 100ths of Amps
(It must be assigned to a variable to be reported)

UJA Value of motor DC bus Voltage in 10ths of Volts.
(It must be assigned to a variable to be reported)

- **Downloading and Uploading Programs to SmartMotors™**
- **I/O Handling**
- **Power Supplies and BackEMF Subjects:**
- **Serial Communications**
- **Tips and Tricks to Better Code and Motor Performance:**

Downloading and Uploading Programs to SmartMotors

Q
A

How do I download Programs to SmartMotors without using SMI?

By using the "LOAD" command you can download from any dcontroller/HMI/PLC or PC based program capable of storing an ASCI text file. For any given motor that is actively addressed, (i.e. you are talking to it and it responds) If you issue the LOAD command to the motor, it immediately goes into a memory-write mode while checking all incoming data. Every ASCII character that is received after the LOAD command is issued goes directly onto the Program EPROM. To terminate the LOAD command, the last characters to send are 2(two) hexFF characters. The hexFF characters tell the motor that it is the end of the file and to drop back into regular command mode.

Details on the downloadable file: When you compile an SMS file with the SMI software, it creates an SMX file extension with the same name in the same directory. This is the file you need to download to the motor. So basically here is what you should do: Do an initial download of your program to the motor from SMI on some other machine. Issue the "RCKS" command. This is the "Report Checksum" command.

It will respond with a string in the form of:

RCKS 000000 0000EB P
where the 000000 0000EB will be different than shown and represent a unique 2-byte checksum to any given program. The P at the end will be either a P (passed) or F (failed).

Keep this number in your own program/PLC that will do the downloading.

1. Store the SMX file for downloading.
2. Store the string received from the RCKS command above as well.
3. Establish serial communications with the motor.
4. Issue RCKS command
5. If it does not match the stored checksum number:
   - Open the smx file.
   - Issue the LOAD command
   - Start sending down all characters in the smx file from beginning to end.
   - When the last character is read from the file and sent
6. to the motor then send 2(two) hexFF characters to the motor. Issue RCKS command again. If it comas back with the stored string (with the "P" at the end) then the download was successful.
7. Issue "RUN" to see if it works as expected.

Reasons for unsuccessful download:

a. Noise on serial port
b. loss of connection during download.
c. failure to send the two hexFF's before power-down.
d. The SMX file as SMI compiled it was altered in some way.

Note: If you were to open an SMX file in Note Pad to look at it and then save it, Notepad will automatically add CR(13) carriage return characters at the end of each line it sees. The resultant file will not work. Each carriage return would have to be stripped back out prior to download. So do not alter the smx file in any way from how SMI generated it.

Downloading and Uploading Programs to SmartMotors

Q **I have multiple motors on a network. How can I download to all or any of the motors I choose?**

A The SMI software allows you to globally download the same program to all motors or to download to a specific motor based on it's address. Using the older SMI software (SMI1), click on the "D" icon at the top. the pop-up window will ask you to set the default motor address.

Type in the address of the motor you wish to download to and that will be the motor that gets the program when you click on the "T" icon at the top. the "T" icon both compiles and transmits the program.
Note: It also saves the program, so if you are testing changes, I would recommend you save the file under another name first. To globally download the same program to multiple motors, go to the Communications setup menu and select transmit setup. then select to download to all motors.

In SMI2 (The new version Windows based software, if you right click on the program you are editing, you will see a pop-up menu with an option to compile and transmit the program. When you click on it, SMI will prompt you for what motor to download to. Choose accordingly and the jog will be done for you.

Q **While downloading to one motor on a chain, I lost communications. When I reestablished communications, SMI found one less motor on the chain than was actually there. What happened?**

A Let us suppose you have 4 motors on a serial daisy chain. You begin download a program and for what ever reason, a glitch occurs with the serial line. Here is what can happen: SMI issues "SLEEP to all other motors so they will not respond to any commands. It then issues "LOAD" to the motor to be downloaded to. That motor is now in ECHO mode, meaning it will echo all incoming characters to downstream motors (or the Host) and yet is also in a memory-write mode. That motor will stay in this mode until it receives a null terminator telling it the download is complete. If for some reason communications from the host is interrupted, the SMI software will throw an error to the effect of "No response received". You then will attempt to regain communications. At this point, the motor that was receiving the program is still in it's memory-write mode. It has yet to see any null terminating characters but yet still echoes through any incoming data.

As a result, the host will not see the motor. It will address all other motors and the chain will have appeared to lose a motor. Newer versions of SMI1 and SMI2 have been updated to check for this, but if you are writing your own download routine from VB, C++, a PLC, etc..., then you need to adjust accordingly for this possible condition. to avoid this, when attempting to address a serial daisy chain, issue 2(two) hexFF characters when setting the ECHO commands so as to end the memory-write state of any motor accidentally left in that condition from a prior incomplete download.

The LOAD command was issued to the motor receiving the program. The motor went into

Q **Why do I get a pop-up that says: EPROM Locked or Missing when I try to download a program?**

A This only occurs under the following conditions:

1. The EPROM is locked or missing in older Molex styled motors with external EPROMS.
2. You are attempting a download to a "Plus" version firmware motor that is in a faulted condition. While faulted, the newer plus version firmware will not allow a program to run. When SMI attempts a download, it first sends a blank test program that it trys to run. If the program does not run, then SMI assumes the EPROM is either locked or missing. SMI2 does not pop up this message because it knows how to test for it ahead of time.

Q **Why will SMI2 not let me download to a motor that is moving or running a program?**

A Due to safety concerns, the new SMI2 software forces you to turn off motor holding current or stop a running program prior to download to prevent possible unexpected motion. Example: Let's suppose you have a program that places the motor in Velocity mode or Torque Mode. Then you try to download without first tuning off the motor. Travel, it will crash into the end stop while downloading. So, for safety sake we want to ensure the motor is in the OFF condition.

Q **When I start a download, the motor stops doing what it was. Why?**

A When SMI starts a download of a new program, it issues the END command to stop prior code form running. This is to prevent processor memory pointer errors while the EPROM header portion is being re-written. It is also for safety.

Downloading and Uploading Programs to SmartMotors™ (continued)

Q **Is there a way to have just one program in all motors instead of downloading separate programs to each motor?**

A Yes. This is more of a programming methodology. Write a single program that tests for motor addresses. And on this test, it tells each motor what to do. Here is an example: This one program would be downloaded to all motors. the program.

```
ADDR=1                  'Change address for each motor as needed prior to download
a=ADDR                  'Set the variable "a" to the motor address.
SWITCH a
          CASE 1        ' In the case of motor 1
          END
          BREAK
          CASE 2         ' In the case of motor 2
          GOSUB20       ' run subroutine 20
          END
          BREAK
          CASE 3        ' In the case of motor 3
          GOSUB30       ' run subroutine 30
          END
          BREAK
ENDS
END
C10
                        ' Place Motor 1 code here
RETURN
C20
                        ' Place Motor 2 code here
RETURN
C30
                        ' Place Motor 3 code here
RETURN
```

Q **Uploading Overview:**

A There are two types of Uploads.

1. Issue the command upload This will cause just the "code" portion of the program to be uploaded. You will notice all comments and empty spaces have been removed from the prior download When you "receive program from motor" in SMI, it issues the UPLOAD command.
2. "UP" When "UP" is issued, the program in it's entirety will be uploaded including all header file portions of the code. You will see the top of the program includes a lot of extra numbers. Many of them are memory pointer "Holders" to tell the processor where to go and how much memory to allocate. The checksum is also stored in the header portion of the file. If you issue RCKS and notice the checksum that is returned, you find a matching string near the front of the header file. This is the stored checksum in the downloaded program.

Q **Is there a way to prevent someone from uploading a program?**

A Yes. You can protect your program from being seen or copied by adding the following command to the top of it LOCKP. The LOCKP command is a means of locking the program. It does not prevent a user from downloading a new program, but it does prevent them from seeing the program you have downloaded.

Downloading and Uploading Programs to SmartMotors™ (continued)

Q

How many times can I download a program to a motor?

A

The chip manufacturer gives a limit of 100000 writes to the EPROM. Tests have been done to in excess of 4 million writes. It is not recommended to continuously re-write a program to a motor. It is far better to have the same program reside in the motor, but instead, change parameters and run-time variables as needed.

Q

Can one SmartMotor download a program to another SmartMotor?

A

No. It is beyond the scope of design to do this. Downloading is better handled where files can be easily stored and retrieved for such use. SmartMotor programs can not really store other program header file information in them.

I/O Handeling

Q

How many I/O are there on each motor and how can they be used?

A

There are 7 universal I/O on each SmartMotor and the RTC3000. They are each labeled as Ports A through G. Note: Port G Digital I/O port is not available on the RTC4000.

Each of the 7 I/O (ports A-G) are 5VTTL and can be assigned as either inputs or outputs. Each I/O port also has a parallel 10 Bit analog input tied to it. For 0-5VDC it will return 0-1023. See the SmartMotor Training Overview Document on this website and read through the section labeled "I/O and Control at a Glance"

Q

Since the I/O is non-isolated 5VTTL, are ther any options for 24VC I/O?

A

Yes. There are a few options: Animatics provides cables with built in 5V to 24V isolated logic circuits right in the connector hood. This allows the user to have a choice of either 4 Inputs and 3 Outputs or 5 Inputs and 2 Outputs at 24VDC. They can be set as sinking or sourcing. The partial Part numbers are CBLIO43 or CBLIO52 and can be purchased in lengths from 3 to 10 meters.

The DINIO7 is a Dinrail mount breakout board that also provides a means of isolation using Industry Standard Opto-22, Gordos, or Grahill I/O modules such as ODC, IDC, OAC and IAC series. The DINIO7 also allows interconnection to other motors and their I/O via a built-in back plane.

Q

Is there any Expanded I/O option?

A

Yes. Each motor has the ability to control expanded I/O via the Anilink protocol or RS-485. There are several Digital and Analog expansion option that allow up to a maximum of 64 channels of expanded I/O. In each case, Ports E and F are used to communicate with the Anilink Products.

Also available is the DINRS-485 I/O card. Each card is a din-rail mount 16 channel card with 8 24VDC sourcing inputs and 8 24VDC sourcing outputs. All I/O are optically isolated. Up to 200 DINRS485 cards can be on a single RS-485 bus controlled by SmartMotors. The outputs are short circuit protected.

Q

Are the I/O pins sourcing or sinking? (PNP or NPN)?

A

A strait answer is neither. They are actually CMOS compatible totem-pole outputs with the ability to be read as inputs. What this means is, that when any given I/O pin (Port A through G) is set as an output (via UAO through UGO commands), and then is set to logic 1 or 0 (via UA-1 or UA-0 for example), the CMOS totem pole MOSFETs either hard drive the output pin to 5VDC or 0VDC. As a result, they are not open-collector outputs.

They BOTH source AND sink. However: when set as inputs via UAI through UGI commands, is nothing is connected to the I/O pin at the connector, the input will appear as a logic level 1 (5VDC). This is because ALL I/O pins have an internal 5KOhm pull-up resistor tied to them.

Q

What type of electrical potection/isolation does each I/O pin have?

A

Each I/O pin has a 100 Ohm series current-limit resistor tied to a 5.6VDC over-voltage limiting zener diode. The user ties into the 100OHm resistor directly. The connection between the 100Ohm resistor and zener diode ties in directly to the CPU pin. This is why the motors are limited to 5VTTL I/O logic levels only. Animatics does however, provide 24VDC I/O adapters and adapter cables for converting the 5VDC I/O to optically isolated 24VDC logic for connection to PLC's and other equipment. Please visit the Animatics website and search under Cables and accessories for the CBLIO-43 and CBLIO-52 cables: http://www.animatics.com/catalog_pdf/51.pdf

I/O Handling (Continued)

Q
A

When set as an output, how much can the ins source or sink?

They source ~4 to 5mAmps max when set to a logic level 1.

They can sink ~12mAmps when set to a logic level zero.

Sample code:

```
UAO    'set port A as an Output
UA=1   'set to 5VDC
UA=0   'set to zero VDC
```

Note, If you want to be sure the Output is set to the proper level immediately, then set it's value prior to setting it as an output:

Example:

```
UA=1      'preset to 5VDC
UAO       'et Port a as Output
```

Q
A

Can I set the output state prior to assigning an I/O pin as an output?

Yes. Example:

```
UA=0   'set port a output state to logic zero (0VDC)
       'at this point, port A is still an input port.
UAO    'set Port A as an output Port.
```

'It will change to being an output and immediately be at ZeroVDC. In fact, doing it by this method allows for slightly faster response and a safer means of control.

Q
A

How do I poll inputs fast?

If the input can be polled while the motor is not required to be moving, there is a simply little known trick you can do to greatly speed up I/O polling. Issue PID8 just prior to the polling code and then issue PID1 just after completion. This will cause the CPU to dedicate more time to program code and less time to PID update rate.

Example:

```
PID8               ' slow down PID
WHILE UAI LOOP     'while port A is not grounded loop
                   'do what ever here...
PID1               'return PID to normal
```

Power Supplies and BackEMF Subjects

Q
A

What is better? Linear or Switcher Supplies?

It depends on the application. Typically speaking linear supplies are better suited for inductive loads. Motors are inductive loads. Linear supplies can handle high current surges typically caused by starting and stopping of servo motors.

However, linear supplies have what is known as voltage-droop. This is characteristic of voltage dropping down with an increase in load. Typically, unregulated Torroidal transformer supplies will drop 4 to 7% and E-Core types (the big square transformers) are >10%. Switchers have no voltage droop until they reach maximum load. Then they just drop completely to zero volts. However, since they maintain a tight control over voltage up to the trip point, they can typically aid greatly in reaching maximum speed and acceleration of a given servo. However, the Switching supply must be sized for the maximum expected peak current draw of the motor system. A linear supply only needs to be sized for continuous load. Linear supplies have a large capacitance to supply much higher current surges when needed. So this is more of an application specific question.

Power Supplies and BackEMF Subjects (Continued)

Q A How do I protect against BackEMF?

Short Answer: Don't back-drive the motor.
Problem is, back driving the motor isn't the only means to produce Back EMF. Read the FAQ on "What is BACKEMF and where does it come from".

The best means to protect against Back EMF are to use a shunt such as the active 48VDC 100Watt shunt supplied by Animatics. It drops a 100 watt load onto the bus any time bus voltage exceeds 49.5VDC. It removes the load when Bus voltage goes back down below 46 VDC.

It will work with Switching or Linear supplies as long as no-load voltage does not stay above 48VDC. Otherwise the shunt will be on all the time. Another method of protection is to use a mechanical break controlled by the Break commands in the motor. The motor can respond to a fault and send a signal to the break within 250useconds to help hold the shaft from back driving. None of these ideas helps against hitting a hard stop. Please read the FAQ on Back EMF for more.

What is Back EMF and where does it come from?

Back EMF:

Back EMF is the voltage generated when a rotor is moving within the stator of any motor. It is literally the motor acting as a generator. There is a common rule that Back EMF or voltage generated is proportional to Velocity. This is true in a constant velocity condition only. Back EMF is actually proportional to the rate of change of magnetic flux (magnetic field strength) inside the stator windings of the motor. The faster the rate of change, the higher the voltage rises. In other words, RPM of the motor shaft does not have to be that high to have very high voltages created.

Here is an example:
Take any relay coil or solenoid valve coil in a 24VDC system. Then it is energized, the magnetic field pulls in the contactor or pilot valve. The magnetic flux reaches saturation and a DC electromagnet is formed. When the power is removed from the coil, the magnetic flux rapidly collapses because there is no forward voltage to maintain it. Since the circuit is not electrically open, there is nothing to prevent the magnetic flux from collapsing rapidly at a hyperbolic rate.

The result is something called "inductive-kick". This kick or spike in voltage for a 24VDC coil can reach very high voltages and currents on the order of 100 times that of the original applied voltage, i.e. 2400VDC!. This is why it is very common to place reverse polarity diodes across relay coils and solenoid valve coils. It protects the system from high voltage spikes. The same thing occurs when a motor hits a hard stop. Suddenly, the rate of change of magnetic flux in the stator windings skyrockets upward because the rotor stopped moving. This sudden change causes an excessive voltage and current spike in the controller and can damage components.

Now: wheat can we do about it: Practically speaking, not much. This is similar to a car crashing in to a brick wall. If the passengers are belted in, they may survive, but the car will sustain unavoidable damage due to the rapid change in speed. (Infinite deceleration to zero speed). No amount of "practical" mechanical design for a typical car will save it from damage when it hits the brick wall. Practical design, means, yes, you could make that car into a large bulky tank that would not get hurt, but then the car would be very heavy, with little space for passengers and be very slow and bad on fuel consumption. This is not practical. The same applies to motor drive design. We could design the drive stage to be able to take the hit of a fast hard stop. But the drive stage would be very large. The controller would have a lot more components in it and the practicality of it would be diminished. The motor would grow in size for the same torque output to 3 times larger. This is just not practical.

Q A What concerns are there with maximizing voltage on the supply?

Power supply Voltage Levels:
The higher the voltage, the fastest the motor can move and the faster it can accelerate. This is a good thing. But in conjunction with that, the higher the voltage, the closer to a peak voltage for over-voltage break down of the controller. Also, the higher the voltage, the faster a rate of change of current can occur. It is a risk with any application to get faster response by moving towards a higher voltage.

Typically speaking, it is the dynamics of sudden changes that increases risk by a "x^2" factor whereas the continuous load risk is only a direct ratio increase. This is because rate-of-change in current is proportional to acceleration which is the square of velocity, i.e. x^2. For safety sake, a 42VDC supply for a 48VDC system gives good margin with little speed losses.

Power Supplies and BackEMF Subjects (Continued)

Q **I was told the motor failed due to over voltage, but I never back drove it or ran it fast.....**
HARD STOP CRASHES:

A See the FAQ on BACKEMF and what it is for more:

Hard Stop Crashes: The best recommendation for preventing damage to the motor/controller in the case of hitting a hard stop is to place a limit switch near the hard stop that trips the motor off line just prior to hitting the stop. The best way to prevent it beyond that is to prevent the cause of hitting the hard stop in the first place.

If this is due to jogging the motor in Velocity mode and not letting of the jog switch in time, then jog in position mode instead and use the "X" or "S" command to stop the motor when the jog switch is released. In any case, much care should be taken to be sure the motor is not intentionally or unintentionally allowed to hit a hard stop while under normal speeds and load conditions.

Q **How do I size Power Supplies?**

A The quick answer is "more is better".

First be sure you have the correct motor for the job. Once that has been down, take the nominal power rating for that motor and you should size a LINEAR supply to provide about 10% more to allow for longer sustained current loads. Any LINEAR supply will typically provide more than enough peak current. This is where sizing gets tricky with Switch-Mode Power supplies. "SWITCHERS" typically come with some rated voltage and current.

For Example: 48VDC at 6Amps.

Well, that is it..... They can provide 48VDC nailed to the wall all the way up to 6Amps, but any more current and the power supply will drop out to zero VDC and typically reset. So any time you wish to use Switchers, you need to take the peak expected load of the motor and size the switcher's continuous rating for that.

As a rule of thumb:
Any 23 Frame SmartMotor™ "can" pull as much as 12 Amps instantaneous. Most 23 frame SmartMotors will not pull more than about 9Amps instantaneous. A 10 amp switcher can supply any 23 frame motor for MOST applications.

A 34 frame motor, forget it, you need 20Amps continuous rating to be sure you will not get a tripped power supply. 34's can pull as much as 40Amps or more for a few milliseconds. so as originally stated: "Bigger is better" especially when it comes to switchers.

Serial Communications

What is meant be ECHO and ECHO_OFF?

There are two states for the main RS-232 communications

serial port:

`ECHO`
And
`ECHO_OFF`

Upon Power-up, the motors default to the ECHO_OFF state. This means that for any serial communications coming in the receive pin of the RS-232 port, they will not "echo" them back out to the transmit port. If you have more than one motor on a serial daisy chain, the only means to get messages or commands to downstream motors is to have them in the "ECHO state.

When the SMI software is told to "address the motor chain", it automatically turns on the ECHO state for each motor it find. From that point forward, any serial data that comes into the receive pin will be "echoed" out to the transmit pin. This will allow downstream motors on the daisy chain to receive serial data as needed. Otherwise, communications could not be established through the entire chain.

Serial Communications (Continued)

Q: What does it mean when the motors are in the "addressed" state or "de-addressed" state?

A: It is important to understand addressed or de-addressed states of SmartMotors™. These states determine whether or not a SmartMotor will respond to commands.

Lets assume for example that we have 5 motors on a communications network.. All of them have programs downloaded with addresses 1-5 respectively via the SADDR command.

On Boot-up all are ready to listen. They will respond to either a globally addressed command or a specific motor will respond to a specifically addressed command. Once a specific address is sent out on the line, that motor will be in the "addressed" state and All other motors will be in the "de-addressed" state. What this means is that from that point on, any command sent

out to the motors without an address proceeding it, will be acted upon only by the motor in the addressed state. All other motors will basically ignore anything received.

By sending out a command preceded by the global address (hex80 or dec128), all motors will be placed into the "addressed state and will remain in that state until another specific address is transmitted. Under the above case of 5 motors addressed 1-5 respectively, if a hex89 for example or any other address outside of hex80-hex85 is sent, all motors would become "de-addressed". No motor would respond to any command until an address within the range of motors on the line is received.

What is the default boot-up state of the communications Ports?

SmartMotors default to 9600 Baud, no parity 8 data bits and 1 stop bit. (9600,N,8,1). All SmartMotors boot up in ECHO_OFF mode with global address zero. This means they will respond to globally addressed commands, i.e. commands proceeded by dec128 or hex80.

They also boot-up in the "Addressed" state. this means that even if no address bit has been seen by the motor since power-up and a valid command is received, the motor will take action on it.

Note: With in the first 500msec's or so of power up, if a SmartMotor has not received any serial communications, it will begin executing code previously downloaded to them from the top down. If it does receive any serial communications within that time period, the processor will not run or execute any downloaded code. This allows a user to re-gain control of a motor with a bad program in it.

Q: I have a motor that will not communicate no matter what I try. What should I do?

A: It is possible to unintentionally write and download a program that will lock up the CPU or prevent serial communications. If you power it up and there is one or more LED lit up, then try the following: Isolate the motor by itself such that you have a single motor power and communications cable between the motor and the PC. Connect the motor to the PC serial Port, but DO NOT power up the motor. In other words, have the power supply either disconnected or turned off. Then Start up the SMI software and click on the TOOLS drop down menu.

Under there, click on "Communications Lock-up Recovery." A pop-up window will tell you to do what is described above. Then click next. The Lock-up recovery utility will begin transmitting multiple "E" characters to the motor. It will tell you to power-up the motor at this point.

Then it will wait about 1 second and attempt to establish communications. If it does establish communications, it will tell you and then prompt to either clear the program or upload the program. At this point, it is advisable to clear the EEPORM so as to down-power reset the motor and reestablish communications normally. This way you will know if it was just a program issue or a hardware issue.

If you apply power and there are no LED's lit up on the motor, then there is a major problem with it electrically. It will have to be sent back for repair.

Q: I downloaded programs to each motor on the serial chain, but after a downpower-reset, I could not communicate with each motor. Why?

A: If you have multiple motors on a single serial daisy chain, they need to be in the "ECHO" state to be able to communicate with each one. See the FAQ on ECHO and ECHO_OFF for more information.

If you reset the motors after downloading the programs, make sure they were not re-addressed incorrectly, i.e more than one motor with the same address.

Serial Communications (Continued)

Q **I have 4 motors on a serial chain, but can only find 3 of them when I try to detect or address them. Why?**

A There is a case where it is possible to "look over" one motor in a chain. Lets suppose you are downloading a large program to one particular motor. If you get a noise glitch or lose power to 1 of the other motors while doing so (but not the one you were downloading to), the motor being downloaded to will be stuck in EPROM write mode. It will be expecting two hexFF terminators to know when the end of the program load has occurred. While in the download EPROM write state, it will ECHO every incoming character to the transmit port. Upon re-establishing communications with the SmartMotors™, that motor will remain in the download state and not answer any commands. As a result, it will be looked over as if it were not even there. The only way out of this condition is to fully down-power all motors and then re-establish communications.

Q **Occasionally I lose communications and don't know why. what could be causing this?**
This is a vague question. To better handle it, more detail would be needed. Here are some things to consider:

A

1. Shielding and Grounding must be done properly to insure good signal integrity.
2. Long character strings should be avoided. the receive buffer is only 16 bytes long.
3. Never use the shield as the ground reference connection for RS-232 or RS-485.
4. Make sure there are no non-terminating strings being transmitted. If a string is not followed by a carriage return or space character, the motor will hang indefinitely waiting on the terminating character. It is done this way to allow priority to the serial ports over any downloaded program execution.

Q **My laptop does not have a serial port. What USB to Serial Adapters do you recommend?**

A There are several available but few good ones. Do not get one that says "Serial to PDA" adapter. They don't work.

- Tried and true good one from Keyspan: http://www.keyspan.com
- Available at CompUSA: http://www.compusa.com
- Saeligue Corp has the FTDI chipsetl: http://www.saelig.com
- Low cost unit that has worked well: http://store.yahoo.com/gomadic-new/onepiusbtors.html
  It is identical to one available at Fry's Electronics out west.

In any and all cases, it is highly recommended to do the following to insure good sustained communications: If running Windows: Open up the Device Manager. Search fro the USB to RS-232 adapter under Ports Serial to PDA then open up the properties for it and look for "Advanced settings if available. Reduce the buffer size to the smallest value available and reduce time-out's to minimum as well. This will prevent Windows from going to slow at getting data from the virtual receive buffers used with USB devices.

Q **How far can I transmit on RS-232 or RS-485?**
The Real Story:

A There are a lot of people that give a wide range of answers.

Here is the real deal:
"RS" in the RS-232 and RS-485 specification means "Recommended Standard". Not every company or chip manufacturer actually meets the "RS"....

The IEEE spec. says RS-232 single ended signal is SUPPOSE to be +/-12VDC or a 24VDC swing from logic zero to logic 1. The spec. for RS-485 is +/-5VDC Differential. By voltage levels, RS-232 will logically be able to transmit much further. By noise immunity, a differential signal should be able to reach it's destination "cleaner". But the real side is this. A clean whisper can't be heard a mile away. But a load guttural voice can. So in reality, RS-232 can transmit further, but RS-485 transmits cleaner.

Also note, the higher the voltage level, the higher the induced noise must be to overcome the signal. So RS-232 isn't so bad after all. As far as actual distances go: There are applications out there running 250 feet on RS-232. RS-485 just can't drive the cables well enough to go beyond 100 feet without loading problems. Also, RS-485 is a parallel bus. The more motors you add, the shorter the over-all distance due to bus loading. RS-232 is serial. Therefor, one motor transmits directly to only one receive buffer. This means compounded bus loading does not occur. This is another reason RS-232 can actually transmit further.

Serial Communications (Continued)

I can't get my RS-485 bus to work at all. What should I look for?

RS-485:

If RS-485 is used, all motors must be in ECHO-OFF mode. RS-485 is a parallel communications network. If any motor was to echo out commands received, it would cause all motors or any other devices on the network to get hit with the same data.

Note: SmartMotors™ use 2 wire RS-485 standards. This means line biasing determines whether or not the motor is in transmit mode or receive mode at the hardware level. To insure motors do not hang up in the transmit mode, there must be a minimum of a 200mVolt differential between RS-485 A and B channels.

This is easily achieved by placing a pull down resistor of approximately 500 Ohms from the B channel to ground somewhere on the RS-485 network.. All SmartMotors have a 5Kohm pull-up resistor on both A and B channels already. The 500 Ohm resistor will provide enough biasing needed to make the hardware default to the receive state. If there are long distances between motors, it may be necessary to provide a shunt resistance across channels A and B. A 200 Ohm resistor wired from A to B at the remote end of the RS-485 line should provide ample voltage drop for needed biasing.

If the above electrical rules are not applied, communications cannot be guaranteed to work. Note: Resistor values above are approximate. The actual values needed may vary depending on communications line impedance due to things such as cable length and the number

How can I address motors on an RS-232 serial chain without using the SmatMotor Interface software?

Addressing SmartMotors from a Host PC or other Serial Device:

Note: The following only applies to an RS-232 serial daisy chain where the motors do not have programs downloaded with addresses in them. It will not work on an RS-485 network. Motors must be pre-addresses in downloaded programs for an RS-485 networks to work at all. Since SmartMotors without addresses default to address zero (hex80 or dec128), a sequence of commands must be issued in proper order to achieve addressing of the SmartMotors. SmartMotors without programs downloaded into them will not retain addresses from this procedure upon loss and return of power!

The following is an example sequence of addressing 3 SmartMotors from the SMI software terminal screen.

Assumptions are as follow:

1. Host PC is set up for 9600 Baud,N,8,1 since this is the power-up default for SmartMotors.
2. Three SmartMotors are wired in serial daisy chain with Tx of Host PC wired to Motor-1 Rx, Motor-1 Tx wired to Motor-2 Rx, Motor-2 Tx wired to Motor-3 Rx, Motor-3 Rx wired to Host PC Rx. (Tx is RS-232 transmit, Rx is RS-232 Receive)

0ECHO_OFF	Places all motors in echo off
0SADDR1	Set first motor to address 1
1ECHO	Set it to echo mode so the next motor will be able to receive commands
1SLEEP	Set it to sleep mode so it will not act upon following commands
0SADDR2	Set next motor to address 2 and repeat sequence
2ECHO	
2SLEEP	
0SADDR3	
3ECHO	
3SLEEP	
1WAKE	Set all motors to wake status
2WAKE	
3WAKE	

Note: SMI Software automatically replaces leading numbers in commands with a decimal offset of 128. In other words, 0ECHO_OFF resulted in "(dec128)ECHO_OFF" being transmitted.

Serial Communications (Continued)

Q
A

I can't get my RS-485 bus to work at all. What should I look for? (Continued)

This is the equivalent from any other software source:

(dec128)ECHO_OFF
(dec128)SADDR1
(dec129)ECHO
(dec129)SLEEP
(dec128)SADDR2
(dec130)ECHO
(dec130)SLEEP
(dec128)SADDR3
(dec131)ECHO
(dec131)SLEEP
(dec129)WAKE
(dec130)WAKE
(dec131)WAKE

Note: (hex80-83) is the same as (dec180-313), All lines must be terminated with either a carriage return (dec13) or space character.

Tips and Tricks to Better Code and Motor Performance:

Q
A

Running code on power-up:

On power-up, all SmartMotors™ and ServoSteps start running their program. They run code from the top down in order as the code is written. You can issue the command "RUN?" to prevent this. The code will stop on the line "RUN?" until the motor specifically receives "RUN" via serial communications. This can be via the primary or secondary port.

Note: the RUN? Command can be placed ANYWHERE in code and have the same effect. It is basically a "pause here until RUN is received" command. But when "RUN" is received, the code WILL begin from the top down again and then jump beyond the "RUN?".

Serial Port Noise: how can you detect it?

A Little known command: "!", that is the exclamation point symbol by itself: !

If it is placed somewhere in code, the code will stop there until ANYTHING is received via serial communications on ANY port.

Example:

```
WHILE 1==1   'while forever
!            'wait here until anything comes in via
comms.
PRINT("noise",#13)
LOOP
```

Now so much as a noise glitch hits the serial receive buffer, the motor will print noise out the port. Keep in mind, anything deliberately sent will cause it to print "noise" as well.

Tips and Tricks to Better Code and Motor Performance: (Continued)

How to create a high speed counter:

Port A and B can be configured as either quadrature encoder input or step and direction input. When used as Step and Direction, you can use port A as a high speed counter. Simply issue "MS0" (stands for Mode-Step-Zero). This will clear out the external encoder counter register (CTR) and set it to zero.

Then for each incoming pulse on port A, CTR will increment (or decrement) it's value depending on the logic level of port B.

You can also set Port B as an output and control the direction of count. This method can be sued to reliably count at rates up to 2megahertz.

Capture Events and make decisions based on existing conditions:

Use Port G interrupt available in the "PLS" and "PS2" version firmware (available in all products except SM2315 and RTC series)

In the latest firmware, you can assign Port G to call C2 subroutine on interrupt. In doing so, any time Port G is grounded, it will call C2 within 250 to 500 use conds. Then the CPU will process code sequentially from C2 down until it reaches "RETURNI" which stands for "Return-Input".

The C2 code will be called and executed regardless of what the motor was doing at the time. It will not effect any motion (unless the code in C2 is written to do so).

Example:

```
C2
x=@P         'Capture position
y=CLK        'Capture clock
z=@PE        'Capture Position Error
i=U&7        'capture Ports A, B and C as a 3-bit masked value.
PRINT( Port G was grounded! ",#13)
SWITCH i     'make decision based on input status.
      CASE 0 'do what ever
BREAK
      CASE 1 'do what ever
BREAK
      CASE 2 'do what ever
BREAK
      CASE 3 'do what ever
BREAK
      CASE 4 'do what ever
BREAK
      CASE 5 'do what ever
BREAK
      CASE 6 'do what ever
BREAK
      CASE 7 'do what ever
BREAK
ENDS
RETURN
```

In so writing the code in this method, you can have a motor not actually running any code at all until Port G is grounded.