BEGINNER'S GUIDE TO BASIC POSITION CONTROL

Prepared by: **James P. Lynch**
Project Manager
Control Techniques Drives
359 Lang Boulevard
Grand Island, New York 14072
Phone: 1-716-774-1193
E-mail: JamesPLynch@compuserve.com

BASIC POSITION CONTROL

SIMPLE POSITION CONTROL

The UD70 has a built-in position controller that can position a hoist or conveyor belt within 1/4096 of a motor revolution (if using a 1024 ppr encoder) as shown in Figure 1 below:

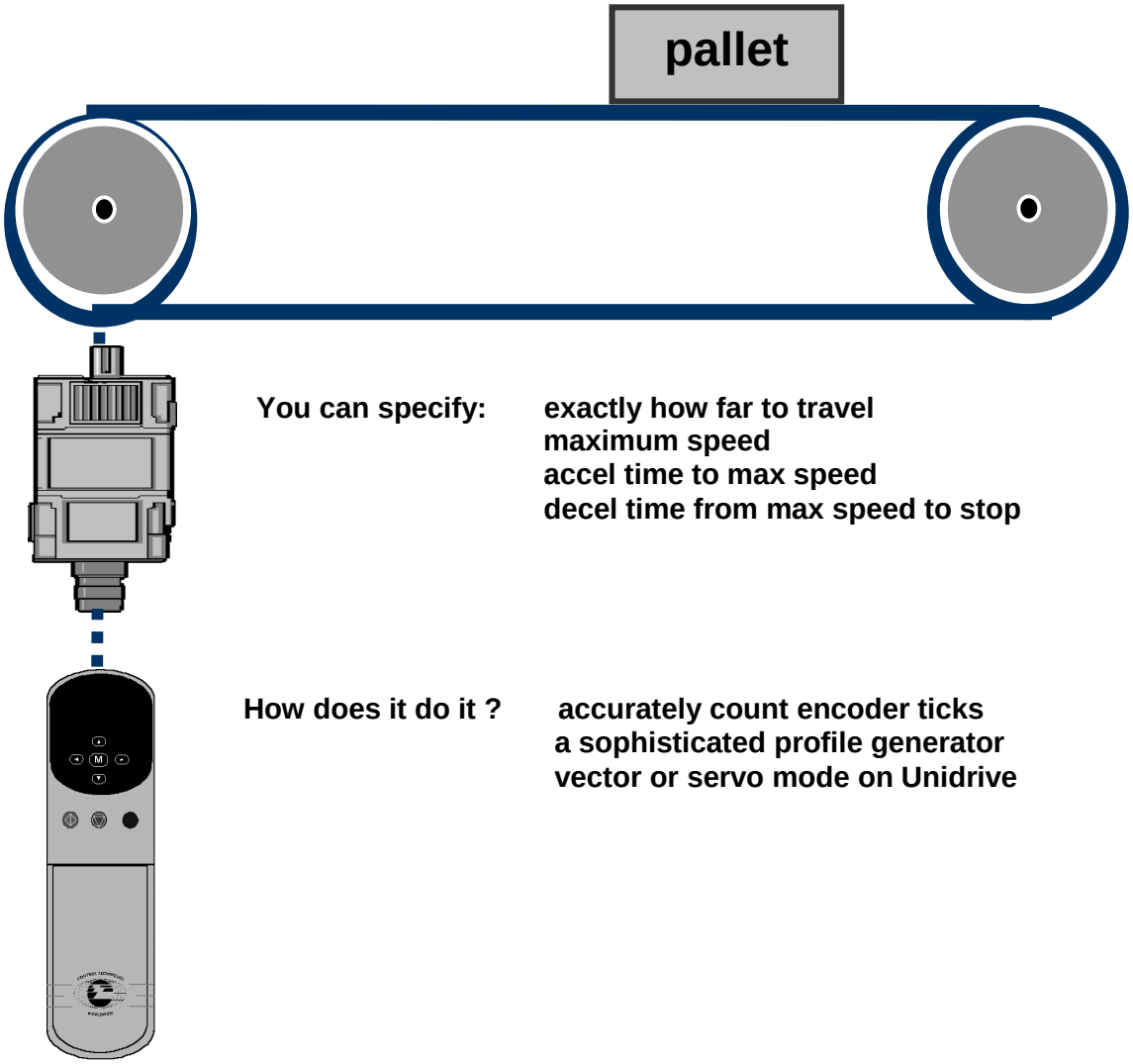


Figure 1. Simple Positioning Application

This feature is permanently imbedded in the UD70's system software and utilizes about half of the UD70 **Q-registers** for setup and control. This article provides a very simple explanation of how position control works and provides a simple but professionally-produced **SYPT** application demonstrating these principles.

PROFILE GENERATOR

The key to proper operation of the position controller is the Profile Generator, which generates a position profile that rotates the motor at the correct speed and accel/decel rates to move from the current position to the destination position, as shown below in Figure 2.

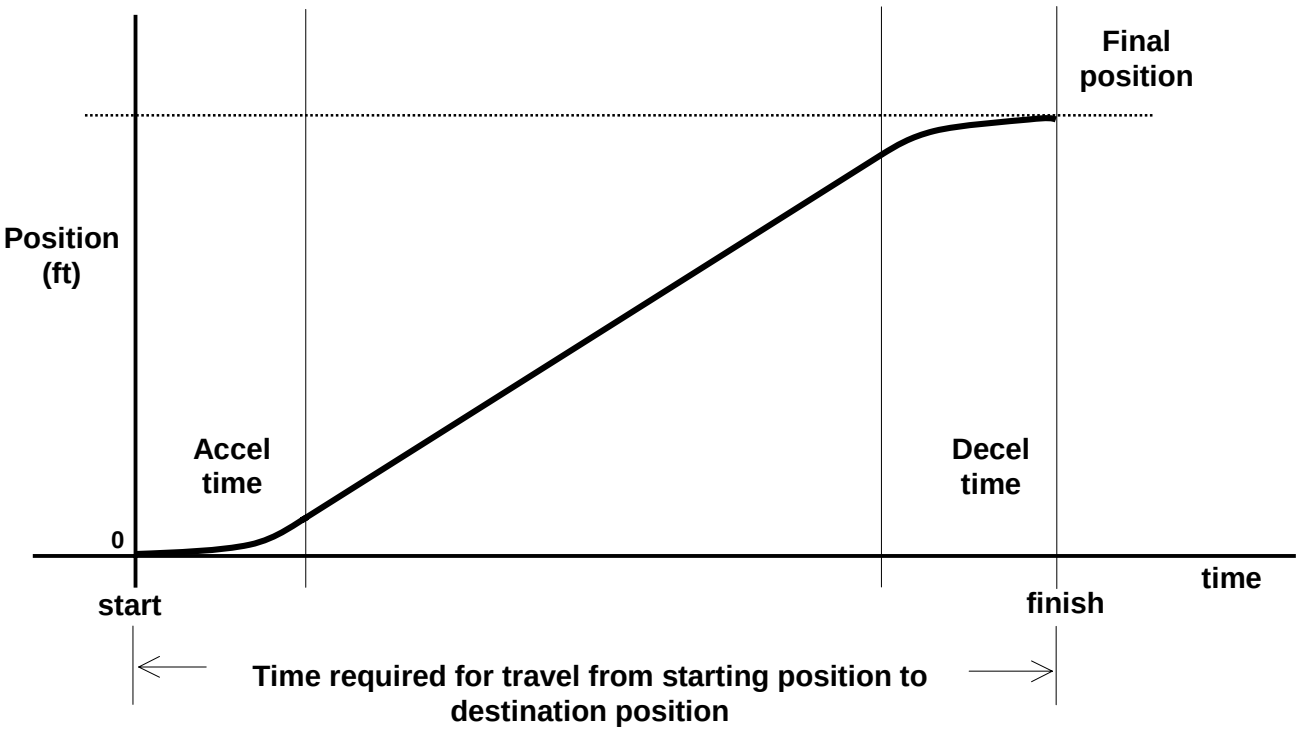


Figure 2. Generated Position Profile

The user may specify the maximum speed and the accel and decel times. This means that the generated position profile will accelerate the motor to the maximum speed in the specified accel time and near the end of the profile decelerate the motor to zero speed in the specified decel time, ending at the target position right on the dot as shown in Figure 3.

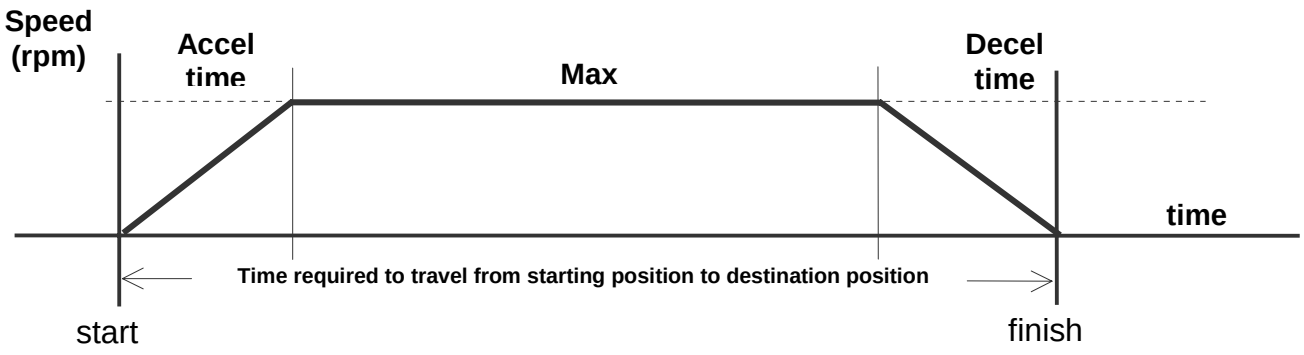


Figure 3. Speed Profile

POSITION CONTROL SYSTEM

A very detailed block diagram of the position controller is available, but it is so complex that it's hard to visualize the basic operation. The greatly simplified block diagram shown in Figure 4 below will help make the system's operation understandable.

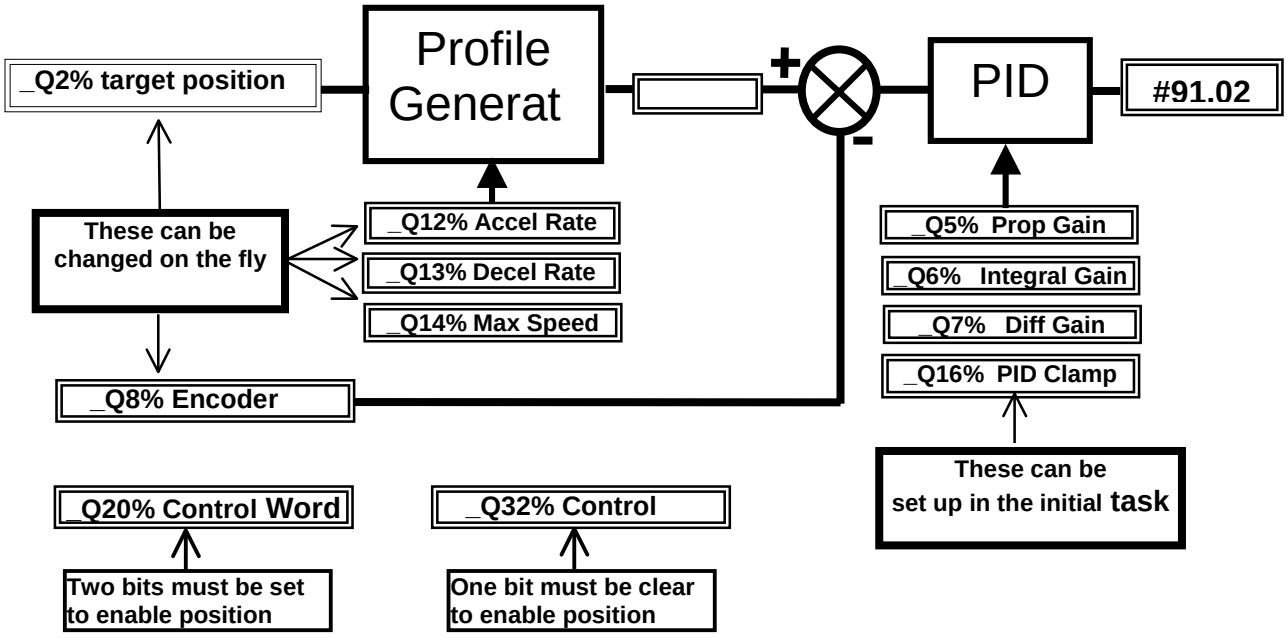


Figure 4. Position Controller – Block Diagram

The target position is entered into _**Q2%** in "encoder lines". **An encoder line is defined as 1/4096 of a motor revolution if a 1024 PPR encoder is being used.**

When a new target position is entered in _**Q2%**, the profile generator generates a dynamic "position versus time" profile in _**Q4%** advancing from the current position to the destination position that was entered into _**Q2%.** The encoder count in _**Q8%** will fall behind the dynamic position in _**Q4%** and this will generate an error signal. The error signal is processed by the PID algorithm and eventually is used to command the drive's reference input in **#91.02**. This causes the motor to turn and thus reduce the error between the actual position _**Q8% (the encoder position)** and the dynamic position from the profile generator _**Q4%.** This simple control system will attempt to follow the dynamic position command in _**Q4%** until the destination position in _**Q2%** is reached. At that point, motion stops and the position move is completed.

While forty eight _**Qxx%** registers are used in the complete position controller, only a small handful as shown above in Figure 4 are required to do simple position moves. The rest of this article describes this subset of _**Qxx%** setup registers and the mathematics required to properly specify them.

BASIC MATHEMATICS OF POSITION CONTROL

To properly specify the correct values for the _**Qxx%** parameters, some simple mathematics is required. Nobody would expess a real-world design in “encoder lines” so it behooves us to develop some simple equations that would convert feet into encoder lines, etc.

Assume we’re designing a conveyor belt with a **2.5** foot drum. Obviously the linear motion of the belt is related to the diameter of the drum, the gear ratio and the revolutions of the motor. A conceptual diagram of this system is shown in Figure 5 below.

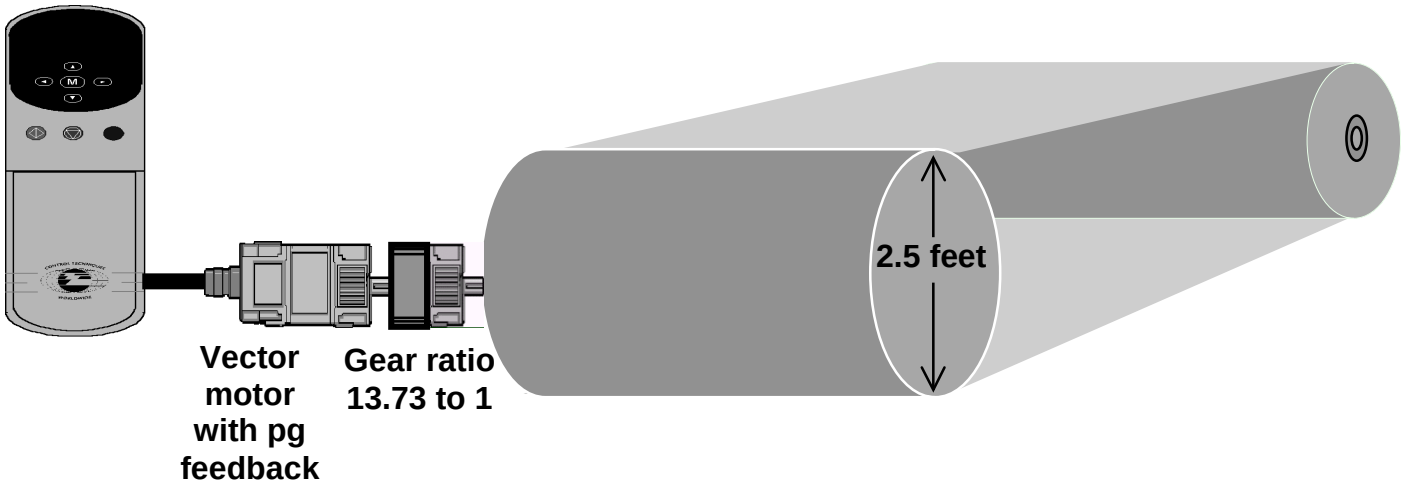


Figure 5. Simple Conveyor Belt System

First we need a scale factor relating linear feet of belt motion to motor revolutions.

$$\mathbf{SCALE} = \frac{\mathbf{GEAR\_RATIO}}{\mathbf{3.14159 * DIAMETER}}$$

For our example system above, the scale factor would be:

$$\mathbf{SCALE} = \frac{\mathbf{13.73}}{\mathbf{3.14159 * 2.5}} = \mathbf{1.748159}$$

Example: How many motor revolutions for 5 feet of travel?

motor_revolutions = scale * distance_feet = 1.748159 * 5.0 = 8.740795 revs

With a scale factor determined, we can now create an equation to calculate the target position **_Q2%** in encoder lines. The scale factor converts feet into revolutions and it was mentioned earlier that there are 4096 "encoder lines" in a single revolution. Thus the target position may be calculated as:

_Q2% = target_position = target_feet * scale * 4096

As an example, to travel 40 feet from the current position:

_Q2% = 40 * 1.748159 * 4096 = 286418 encoder lines encoder lines

Now we need to determine the maximum speed **_Q14%** in encoder lines per second. The same scale factor can be used to convert feet/sec into revolutions/sec. Observing again that there 4096 "encoder lines" in a single revolution, a very similar equation will convert feet/sec into encoder lines/sec.

Q14% = max speed = max speed fps * scale * 4096

The maximum speed should never exceed the baseplate speed of the motor or the drive's maximum speed setting **#5.08**. If you do exceed the baseplate speed, this introduces "clipping" into the control system. You will eventually reach the target position, but not with the accel and decel profiles you specified. To prevent this, always submit the value calculated for **_Q14%** to the following test:

_Q14_MAX% = (#5.08 / 60) * 4096 = #5.08 * 68.26666

IF (Q14% > _Q14_MAX%) THEN _Q14% = Q14_MAX%

As an example, If we wanted our maximum hoist speed to be 1.5 feet/sec:

_Q14% = 1.5 * 1.748159 * 4096 = 10740 encoder lines/sec

The acceleration rate that the position controller accelerates the drive from zero speed to the maximum speed is entered into **_Q12%** in encoder lines per sec^2. Likewise, the deceleration rate that the position controller decelerates the drive from maximum speed to zero is entered into **_Q13%** in encoder lines per sec^2. Of course, the user would rather specify an accel and decel time in seconds.

The equation for accel rate in lines/sec^2 is:

$$\text{accel\_rate (lines/sec}^2\text{)} = \frac{\text{max\_speed\_rpm} * 4096}{\text{accel\_time} * 60}$$

$$\text{accel\_rate (lines/sec}^2\text{)} = \frac{\text{max\_speed\_rpm} * 68.266666}{\text{accel\_time}}$$

The maximum speed in RPM is:

$$\text{max\_speed\_rpm} = \text{max\_speed\_fps} * \text{scale} * 60$$

Finally, the proper specification of the accel rate is:

$$\text{\_Q12\%} = \frac{\text{max\_speed\_fps} * \text{scale} * 60 * 4096}{\text{accel\_time} * 60}$$

$$\text{\_Q12\%} = \text{accel\_rate} = \frac{\text{max\_speed\_fps} * \text{scale} * 4096}{\text{accel\_time}}$$

The decel rate calculation is quite similar:

$$\text{\_Q13\%} = \text{decel\_rate} = \frac{\text{max\_speed\_fps} * \text{scale} * 4096}{\text{decel\_time}}$$

As an example, suppose we wanted an accel rate of 5 seconds and a decel rate of 8 seconds.

$$\text{\_Q12\%} = \frac{1.5 * 1.748159 * 4096}{5.0} = 2148 \text{ encoder lines/sec}^2$$

$$\text{\_Q13\%} = \frac{1.5 * 1.748159 * 4096}{8.0} = 1342 \text{ encoder lines/sec}^2$$

The PID output clamp should be set to 10% of the maximum speed **_Q14%**

Q16% = PID_OUTPUT_CLAMP = _Q14% * .1

As an example, using the maximum speed calculated earlier:

_Q16% = _Q14% * 0.10 = 10740 * .10 = 1074

The PID gains (proportional, integral and differential) are entered in “times 1000” motif. For example, a gain of 5 would be entered as 5000.

PID gains are typically tweaked by the user to adjust dynamic performance of the control system, but the following values are recommended as a good start.

Q5% = proportional_gain = 20000

Q6% = integral_gain = 0

Q7% = differential_gain = 1000

The Position Controller writes its speed command to **#91.02**, the “fast update” virtual parameter as shown in Figure 6 below. This is essentially the same as writing to Preset Speed #1 with improved resolution and update rate.

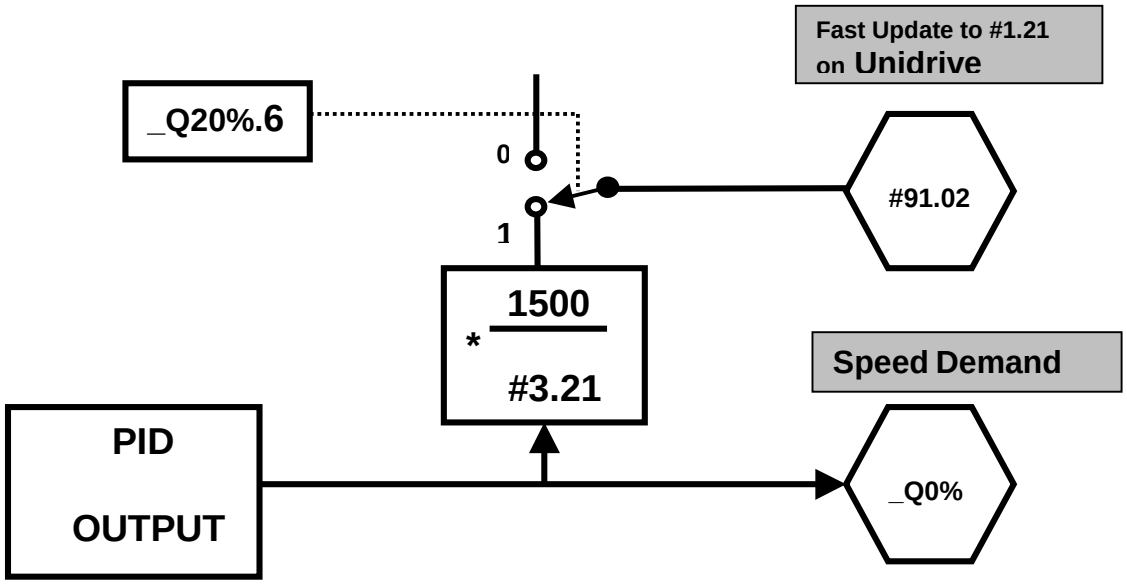
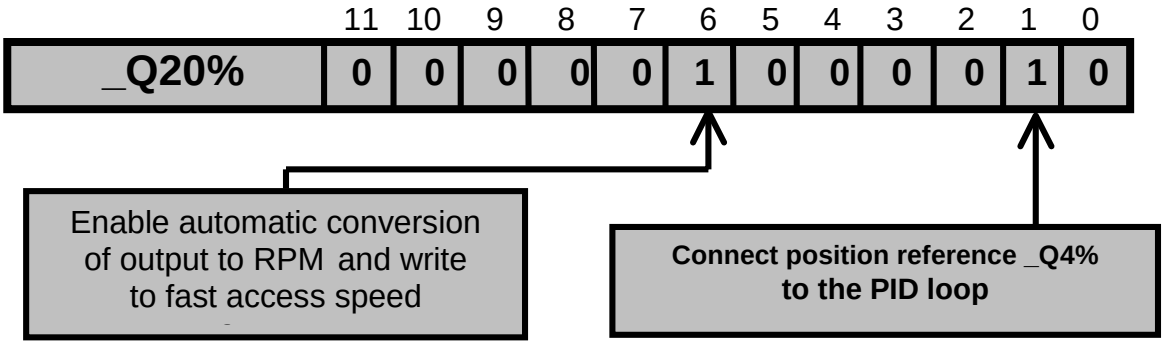


Figure 6. Disposition of Final Reference

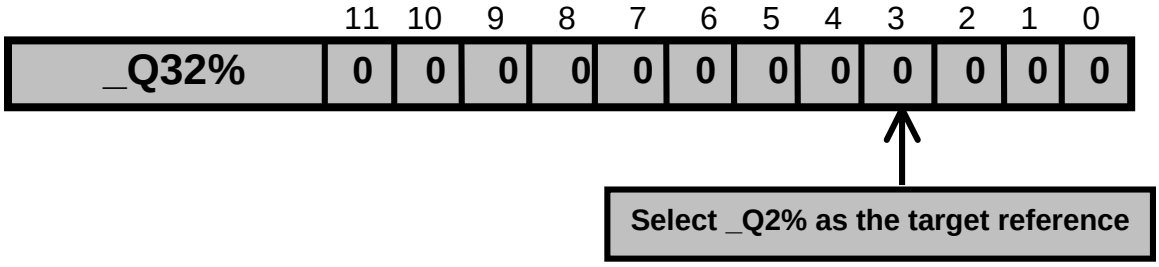
To set up the Unidrive to utilize the “Fast Update” virtual parameter as the reference, the following Unidrive parameters should be set up:

#91.01 = 3	**Fast Key Enable – Select #91.02 as**
#91.05 = #5.08	**Use nameplate max speed as scale**
#1.14 = 3	**Select Preset Speeds as reference**
#1.15 = 1	**Select Preset Speed 1**

The Position Controller **CONTROL WORD _Q20%** needs two bits set to enable simple position control, as shown directly below:



Bit 3 in the **CONTROL PARAMETER _Q32%** needs to be cleared to enable simple position control also.



The two control words should be set up as follows:

_Q20%	**=**	**Control Word**	**= 0x42**
_Q32%	**=**	**Control Parameter**	**= 0x00**

This completes the setup of the Q-registers required for simple position control. **All other Q-registers may be cleared to zero.**

PROFILE_IN_PROGRESS_BIT

Bit 10 in **_Q31%** indicates that a position profile is in progress, as shown in Figure 7 below. The the clearing of the bit when the profile completes can be used as a indicator to **STOP THE DRIVE**.

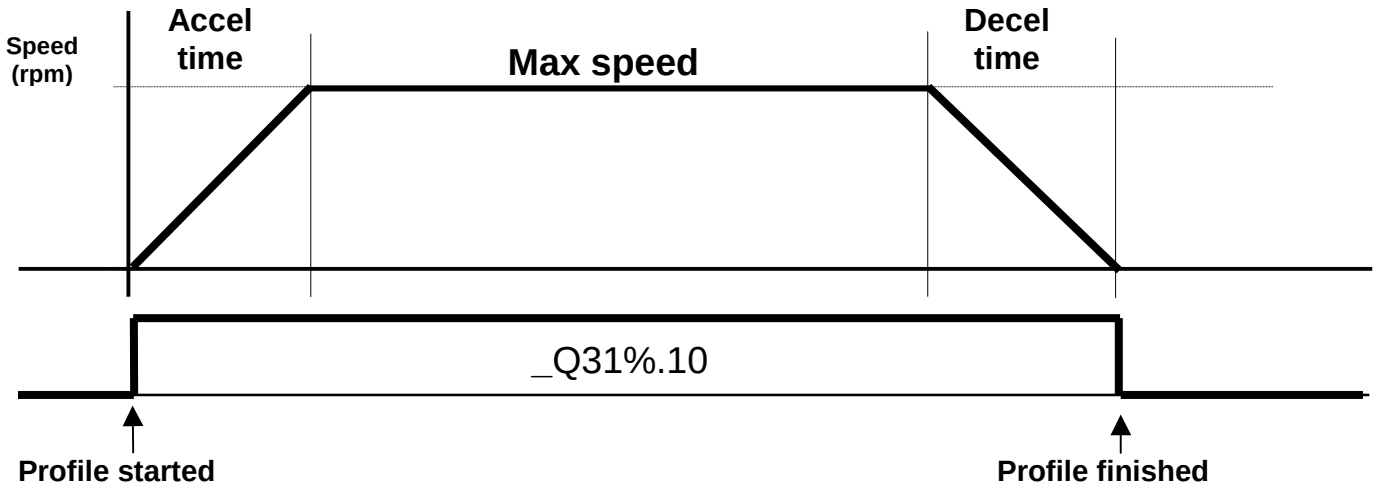
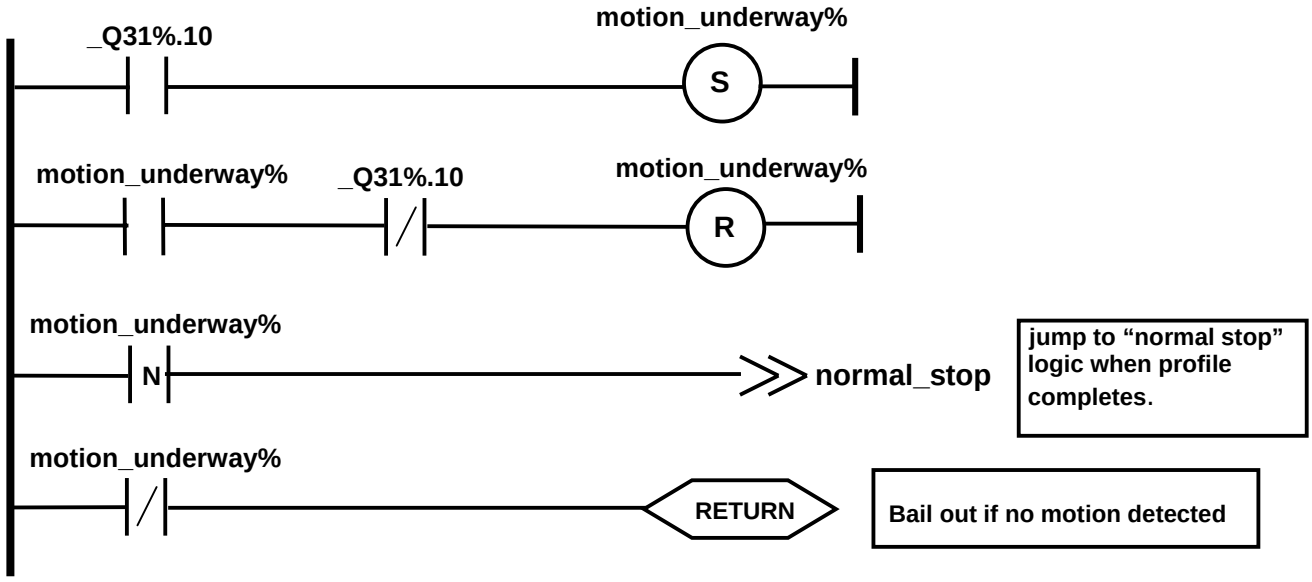


Figure 7. Profile-in-Progress Bit

The following ladder logic in the background task uses the "profile_in_progress" bit to set a **motion_underway%** semaphore and reliably detect normal end of motion.



In the ladder diagram above, the top rung sets the **motion_underway%** semaphore when the profile_in_progress bit first sets. The second rung clears the **motion_underway%** bit when the profile completes normally. Note that the check for profile complete is only made after motion start has been detected.

A negative edge-triggered contact in the third rung detects the exact moment when the position profile completes normally and jumps to the part of the ladder diagram that handles "normal stop" procedures.

SETTING UP THE UNIDRIVE

For the position controller to work, the Unidrive must be set into “Closed-loop Vector” or “Servo” mode. For this document, we shall use “Closed-loop Vector” mode.

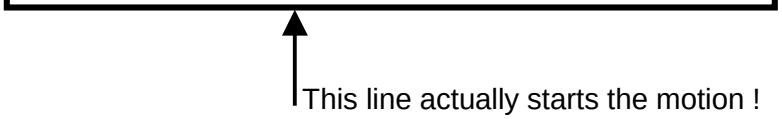
The Unidrive parameters required to setup “Closed-Loop Vector” mode are:

UNIDRIVE SETUPS FOR POSITION CONTROL	
allow mode change (1253=50Hz, 1254=60Hz)	**00.00 = 1254**
“closed loop vector” mode	**00.48 = Cl.VEC (reset)**
Set motor poles to 4	**00.42 = 4**
Set power factor to 0.85	**00.43 = 0.85**
rated motor voltage to 460 volts	**00.44 = 460**
Set rated motor speed	**00.45 = 1725**
rated motor current to 1.1 amps	**00.46 = 1.10**
Set rated motor frequency to 60 hz	**00.47 = 60**
Maximum Speed	**01.06 = 1750**
Minimum Speed	**01.07 = 0**
bipolar mode	**01.10 = 1**
preset speed reference	**01.14 = 3**
preset speed # 1	**01.15 = 1**
Disable Speed Ramps	**02.02 = 0**
Drive Ramps set to FAST	**02.04 = 1**
“At Zero Speed” threshold to 1 rpm	**03.05 = 1**
Overspeed Trip Threshold	**03.08 = 2000**
Speed Loop Proportional Gain	**03.10 = 1000**
Speed Loop Integral Gain	**03.11 = 100**
Speed Loop Differential Gain	**03.12 = 0**
Disable Hard Speed Reference	**03.20 = 0**
1024 PPR Encoder	**03.21 = 1024**
Set encoder as feedback device	**03.22 = 0**
Set encoder voltage to 5 volts	**03.23 = 0**
Verify that the encoder is working properly Rotate motor shaft clockwise (as viewed from shaft end)	**#3.26 should be positive #3.27 should count up**
Speed Feedback Filter	**03.30 = 1**
Symmetrical Current Limit	**04.07 = 175.0**
Current Demand Filter Time Constant	**04.12 = 3**
Switching Frequency (12 MHz)	**05.18 = 4**
No Stopping Ramps	**06.01 = 2**
PLC Sequencing Mode (PLC mode)	**06.04 = 3**
Hold Torque at Zero Speed	**06.08 = 0**
Drive Enable	**06.15 = 1**
Sequencing Bit 0 (drive start)	**06.30 = 0**
Defeat the Th thermistor	**07.15 = volts**
Reprogram the digital inputs to null	**08.13 = 00.00**
	08.16 = 00.00
	08.19 = 00.00
	08.21 = 00.00
	08.23 = 00.00
	08.25 = 00.00
Negative Polarity Input Logic	**08.27 = 0**
Enable the Position Controller	**17.12 = 2**
Save these setups	**00.00 = 1000 (reset)**
Restart the UD70	**00.00 = 1070 (reset)**

STARTING THE POSITION CONTROLLER

Starting a position move is quite simple; you start the drive and then jam in a new target position (in encoder lines) into **_Q2%.**

```
#6.15  =  1                    // enable the drive

#6.30  =  1                    // start the drive

_Q2%  =  286416      // load new target position
```



STOPPING THE POSITION CONTROLLER

The position controller normally runs to completion, stopping motion “on the dot” at the specified destination position. However, it does not stop the drive. To do this, you have to be able to detect that the position profile has finished, as the following code suggests. An equivalent ladder diagram for this procedure was given earlier.

```
Background {

If  Q31%.10  =  1  then  motion_in_progress% = 1        // detect profile start

If  motion_in_progress%  =  1  then              // only check for stop after start detected

        If  _Q31%.10  =  0  then                 // did profile complete ?

                _Q2%  =  _Q8%                    // yes,  clamp position controller
                _Q4%  =  _Q8%                    //       clamp position controller
                #6.30  =  0                      //       stop drive
                motion_in_progress% = 0          //       clear semaphore
        endif
endif

}
```

The two lines commented as “clamp position controller” guarantee there will be no error signal.

PANIC STOP THE POSITION CONTROLLER

Sometimes you need to stop the position controller before it completes its profile. This may be an emergency situation, such as someone hitting the Emergency Stop button or a limit switch turning on. The simplist way to stop during a profile is to to set the maximum speed to zero. This will force a controlled ramp to zero speed, as the following code illustrates.

```
Background {

// Panic stop the controller
_Q14% = 0                    // Set Max speed to zero (forces a ramp-to-zero)

do while #10.03 = 0 loop     // wait here until speed hits zero

_Q2% = _Q8%                  // clamp the position controller
_Q4% = _Q8%                  // clamp the position controller
#6.30 = 0                    // stop the drive

}
```

The ramp-to-zero occurs with the previously established accel and decel times. Since you might want to panic stop as fast as possible, you can change the accel and decel time before executing the above code. For example, if we want to panic stop in **0.25** seconds:

$$\text{Max\_speed\_rpm} = \frac{\text{\_Q14\% * 60}}{\text{4096}}$$

$$\text{\_Q12\%} = \text{ACCEL\_RATE} = \frac{\text{Max\_speed\_rpm * 68.266666}}{\text{.25}}$$

$$\text{\_Q13\%} = \text{DECEL\_RATE} = \frac{\text{Max\_speed\_rpm * 68.266666}}{\text{.25}}$$

HELPFUL USER DEFINED FUNCTIONS

One of the nicest features of the Control Techniques **SYPT** Toolkit is the ability for the user to craft custom function blocks. A set of custom functions supporting position control would be a good start towards developing any position controller application.

1. INITIALIZE THE POSITION CONTROLLER

(status%) = _pos_init(En%, Ctrl%, MaxSp%, PILim%, Pgain%, Igain%, FF%) {

```
//  Initialize the UD70 Position Controller
//  -------------------------------------------------------
//
//  INPUTS:
//
//  En% -         Enable the initialisation.
//
//  Ctrl% -       Control Word, each bit is configured as follows:
//                      bit0 = Task, Position loop synchronisation.    0 = encoder task (#17.12=1)
//                                                                     1 = speed task   (#17.12=2)
//                      bit1 = Reset bit. if set   Q32%=Q20%=Q1%=Q2%=Q3%=Q8%=Q4%=Q9% = 0
//                      bit2 = Feedback Selection.      0 = main encoder       (_Q20%.7=0)
//                                                      1 = auxillary encoder  (_Q20%.7=1)
//                      bit3 = Reserved
//                      bit4-6 = Reference selection:-
//                                              Position  Speed  Digital-Lock  CAM  None
//                              Ref. No.        1         2      3            4    0
//                              _Q20%.1 =       1         1      1            1    0
//                              _Q32%.2 =       0         0      1            0    0
//                              _Q32%.3 =       0         1      0            0    0
//                              _Q32%.4 =       0         0      0            1    0
//                      Note:   The master and slave arrays must be declared along with a
//                              succesful CAMINIT before the CAM can be selected.
//
//  MaxSp% -    Maxspeed in RPM this will set parameters:- #01.06, #91.05.
//
//  PILim% -    PI terms max output clamp entered as percentage of Max Speed.
//                      (Q16%= (MaxSp%*(#3.21or #16.04)*PILIM%*4)/(60*100))
//                      To ensure there is no rollover the max range is limited to +ve 0 - 300%.
//
//  Pgain% -    Proportion Gain _Q5%.     (x1000)
//
//  Igain% -    Integral Gain _Q6%.       (x1000)
//
//  FF%-        Feedforward Gain _Q7%.  (x1000)  note: also referred to as "differential gain"
//
//
//  OUTPUTS:
//
//  status% -   Function Block executed correctly without any errors.
//
//
//   Programmer:  Kevin Manton, Technical Manager SSPD  Created May 18, 1999
//                James P. Lynch  ( some modest twiddling of Kevin's original version )
```

```
// initialisation code
// ===================
Temp% = 0
Tempa%= 0

FBbody
// main body code
// ==============

IF En%=1 then

        // disable drive while setting up
        #6.15 = 0

        Tempa%=Ctrl%

        // position loop task selection
        // ======================
        //      0 = Encoder Task
        //      1 = Speed Task
        IF #17.12 <> (Tempa%.0 + 1) THEN
                #17.12 = (Tempa%.0 + 1)         // Enable Position to update with selected task.
                #00.00 = 1070                   // UD70 Reset
                DO WHILE Temp% < 50
                        #10.38 = 100            // Reset
                        Temp%  = Temp% + 1
                LOOP
        ENDIF

        // position loop reset
        // ===============
        IF Tempa%.1=1 THEN
                // zero all _Qxx% registers that the position controller might use
                Q_pointer% = 7100
                do
                        #Q_pointer% = 0
                        Q_pointer% = Q_pointer% + 1
                loop while  Q_pointer% < 7141
        ENDIF

        // feedback selection
        // ===============
        // 0 = Standard Motor Feedback
        // 1 = Second Encoder Feedback
        _Q20%.7 = Tempa%.2

        // reference selector
        // ===============
        // 0 = No Reference selection. (_Q9% only)
        // 1 = Position Reference.
        // 2 = Speed Reference.
        // 3 = CAM Profiler reference.
        Temp% = (Tempa% & 0x70)/16          // Mask reference selector bits
                                            // and shift right x4
```

```
IF Temp% = 1 THEN              // Enable Position reference
        _Q20%.1 = 1
        _Q32%.2 = 0
        _Q32%.3 = 0
        _Q32%.4 = 0
ELSEIF Temp% = 2 THEN          // Enable Speed reference
        _Q20%.1 = 1
        _Q32%.2 = 0
        _Q32%.3 = 1
        _Q32%.4 = 0
ELSEIF Temp% = 3 THEN          // Enable Digital Lock
        _Q20%.1 = 1
        _Q32%.2 = 1
        _Q32%.3 = 0
        _Q32%.4 = 0
ELSEIF Temp% = 4 THEN
        _Q20%.1 = 1            // Enable CAM reference
        _Q32%.2 = 0
        _Q32%.3 = 0
        _Q32%.4 = 1
ELSE
        _Q20%.1 = 0            // Disable reference
        _Q32%.2 = 0
        _Q32%.3 = 0
        _Q32%.4 = 0
ENDIF

// Max Speed
// =========
// Set in RPM
#91.05 = MaxSp%                // Default MAX Speed Resolution.(RPM)
#01.06 = MaxSp%                // Default MAX Speed.(RPM)

// PI Clamp
// ========
// Set in %.
#03.21 = 1024                                  // 1024 PPR Encoder
PILim%=MIN((MAX(PILim%,0)),300)                // Clamp range Positive 0-300%.

// counts per rev.(required as output is in rpm)
IF _Q20%.7 = 1 THEN                            // Determine Feedback Pulses Per Revolution.
        Temp%=(INT(#16.04)*4)                  // Second Encoder
ELSEIF _Q20%.7 = 0 THEN
        Temp%=(#03.21*4)                       // Main Motor Encoder
ELSEIF _Q20%.7 =0 AND #16.01 = 4 THEN          // SIN/COS
                                               // Note this only applicable for sysfile V2.7.4
        IF _Q19% = 0 THEN Temp% = 16384        // 14 bit
        IF _Q19% = 1 THEN Temp% = 32768        // 15 bit
        IF _Q19% = 2 THEN Temp% = 65536        // 16 bit
        IF _Q19% = 3 THEN Temp% = 131072       // 17 bit
        IF _Q19% = 4 THEN Temp% = 262144       // 18bit max
ENDIF
```

```
        Tempa% = (MaxSp%*Temp%) / (60)          // Max Speed in Counts
        _Q16%  = (Tempa%*PILim%)/(100)          // Set PI Output Clamp.

        // PID gains
        // ========
        _Q5% = Pgain%
        _Q6% = Igain%
        _Q7% = FF%


        // other parameter settings
        // ====================
        #91.03 = 3                              // Enable fast speed update.
        #01.10 = 1                              // Enable Bipolar speed ref.
        #01.14 = 3                              // Select Preset Speed reference.
        #01.15 = 1                              // Select Preset Speed #1.
        #02.02 = 0                              // Disable Drive Ramps.
        #02.04 = 1                              // Drive Ramps set to Fast Ramp.
        #03.05 = 1                              // "At Zero Speed" threshold
        #03.08 = 2000                           // Overspeed Trip Threshold
        #03.10 = 1000                           // Speed Loop Proportional Gain
        #03.11 = 100                            // Speed Loop Integral Gain
        #03.12 = 0                              // Speed Loop Differential Gain
        #03.20 = 0                              // Disable Hard Speed Reference
        #03.30 = 1                              // Speed Feedback Filter
        #04.07 = 175                            // Symmetrical Current Limit
        #04.12 = 3                              // Current Demand Filter Time Constant
        #05.18 = 4                              // Switching Frequency  ( 12 MHz )
        #06.01 = 2                              // No Stopping Ramps
        #06.04 = 3                              // PLC Sequencing Mode  ( PLC mode )
        #06.08 = 0                              // Hold Torque at Zero Speed
        #06.30 = 0                              // stop the drive
        #08.13 = 0                              // disconnect F2
        #08.16 = 0                              // disconnect F3
        #08.19 = 0                              // disconnect F4
        #08.21 = 0                              // disconnect F5
        #08.23 = 0                              // disconnect F6
        _Q20%.6= 1                              // Enable output of PID
        #06.15 = 1                              // Drive Enable

        status% = 1                             // Status code OK.

ELSE

        status% = 0                             // If NOT enabled Reset status code.

ENDIF

} // (status%)  =  _pos_init( En%, Ctrl%, MaxSp%, PILim%, Pgain%, Igain%, FF% )
```

As an example, assume we'd like to initialise the built-in position controller to operate with basic position regulation synchronised to the **SPEED** Task, using the main encoder, zeroing out the _**Qxx%** registers before setting them up. Assume also that we'd like a maximum speed of 1725 RPM, a PID clamp at 10% of maximum speed and appropriate PID gains.

The **En%** input is a simple enable, setting it to 1 will execute the block. Setting **En%** to 0 will cause the block to fall through with no changes.

The difficult setting is the multi-purpose **Ctrl%** input (it's a bit field!).

bit0 = task synchronisation. **1 = sync to speed task**
bit1 = reset bit. **1 = clear the _Qxx% registers before starting**
bit2 = Feedback Selection. **0 = select main encoder**
bit3 = unused **0 = unused**
bit4-6 = Reference selection **001 = simple position control where _Q02% is the reference**

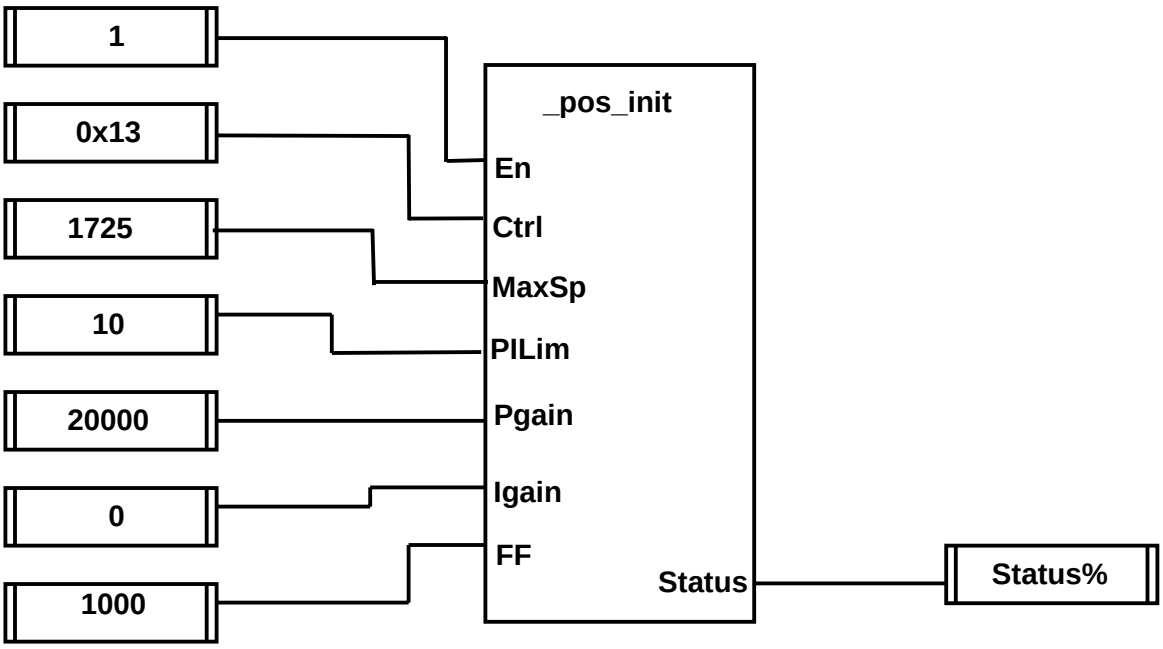
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	1	0	0	1	1
unused	Reference Selection			unused	encoder select	reset bit	sync to task

Ctrl% = 0x13

The maximum speed input **MaxSp%** will be set to 1725, a typical maximum speed for 60 hz 3-phase motors.

The **PILim%** input specifies the clamp for the PID block, here we set it to 10% of the maximum speed.

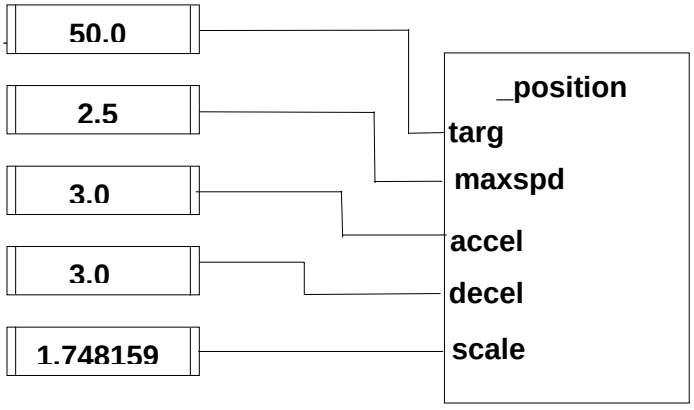
The PID gains are established by the **Pgain%**, **Igain%** and **FF%** inputs. Good starting values for these variables are 20000 for **Pgain%**, 0 for **Igain%** and 1000 for **FF%**. These gains are in "times 1000" motif.



2. MOVE TO A POSITION

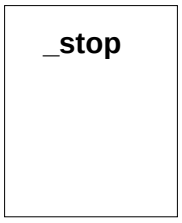
```
_position(targ, maxspd, accel, decel, scale) {

//   Set Up a Normal Position Move
//   -----------------------------------------------
//
//   Sets up a normal position move using the UD70's built-in position controller.
//
//   Inputs:  targ        =  target position in feet
//            maxsp       =  maximum speed in feet/sec
//            accel       =  accel time in seconds
//            decel       =  decel time in seconds
//            scale       =  converts feet into motor revolutions
//
//  Outputs:    none
//
//  Programmer:  James P. Lynch  March 23, 2000

// initialisation code
encoder_lines = 4096.
rated_speed = #1.06

FBbody
// main body code
_Q32%.3 = 0
_Q20% = 0x42                                          //  enable position reference
maxspeedrpm  = maxsp * scale * 60.                    //  convert max speed to rpm
if  maxspeedrpm > rated_speed  then  maxspeedrpm = rated_speed - 5.0   //  clamp max speed
if  accel <= 0.0  then accel = .5                     //  prevent "divide by zero"
_Q12%= INT ( (maxspeedrpm * 68.2666) / accel )        //  accel rate
if  decel <= 0.0  then decel = .5                     //  prevent "divide by zero"
_Q13%= INT ( (maxspeedrpm * 68.2666) / decel )        //  decel rate
_Q14%= INT (maxspeedrpm * encoder_lines) / 60.        //  max speed
#6.15 = 1                                             //  enable drive
#6.30 = 1                                             //  start the drive
_Q2%  = INT( targ * scale * encoder_lines )           //  load target position – starts the move
} // _position(targ, maxspd, accel, decel, scale)
```

Example: Move to 50.0 feet with 2.5 feet/sec max speed, accel/decel time of 3 seconds, and a scale factor of 1.748159



3. NORMAL STOP THE POSITION CONTROLLER

```
_stop( ) {

//   Stop drive and clamp position controller
//   ----------------------------------------------------------------
//
//         This function is usually called after normal completion of a position move.
//
//         It clamps the position controller ( sets target and post-ramp reference equal to the encoder )
//         This ensures that there's no error signal and the position controller is completely stopped.
//
//          Finally, it stops the drive.
//
//   Inputs:    none
//
//  Outputs:    none
//
//  Note:  this function sets target (_Q2%)  to feedback (_Q8%),
//           slave position (_Q4%) to feedback (_Q8%) and stops the drive
//
//  Programmer:  James P. Lynch  2/3/2000

// initialisation code

FBbody
// main body code

_Q2% = _Q8%                          // clamp the position controller
_Q4% = _Q8%                          // clamp the position controller
#6.30 = 0                            // stop the drive

} // _stop( )
```

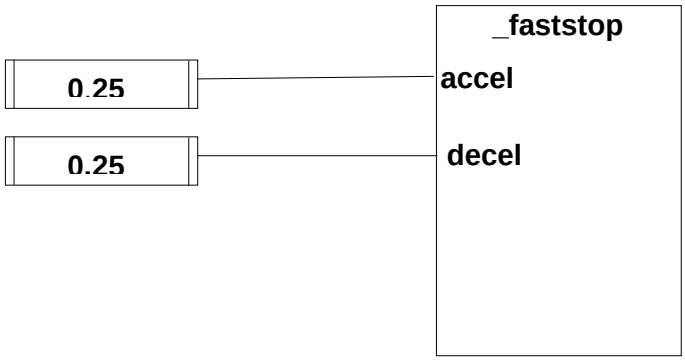


Note: This function block has no inputs and no outputs.

4. PANIC STOP THE POSITION CONTROLLER

```
_faststop( accel,  decel  ) {

//   fast stop the position controller
//   ---------------------------------------------------
//
//   Stops the position controller - usually before the position move is finished.
//
//   Sets max speed ( _Q14% ) to zero, this causes a controlled ramp to zero speed.
//   When zero speed detected ( #10.03 = 1 ), clamps the position controller and stops the drive.
//
//   Inputs:   accel  =  user-defined accel rate in sec
//             decel  =  user-defined decel rate in sec
//
//  Outputs:   none
//
//  Programmer:  James P. Lynch  2/3/2000

// initialisation code
encoder_lines = 4096.
max_speed = #5.08

FBbody
// main body code

max_speed = FLOAT( _Q14%  * 60 ) / encoder_lines   // calculate the current max_speed
if  accel <= 0.0  then accel = .5                  // prevent "divide by zero"
_Q12%= INT ( (max_speed * 68.2666) / accel )       // accel rate
if  decel <= 0.0  then decel = .5                  // prevent "divide by zero"
_Q13%= INT ( (max_speed * 68.2666) / decel )       // decel rate
_Q14%= 0                                           // zero max speed, this forces a ramp to zero
do while  #10.03 = 0  loop                         // now wait for the drive's speed to hit zero
_stop( )                                           // clamp the position loop and stop the drive

}  // _faststop( )
```

Assume we want to panic stop with a decel time of .25 seconds.

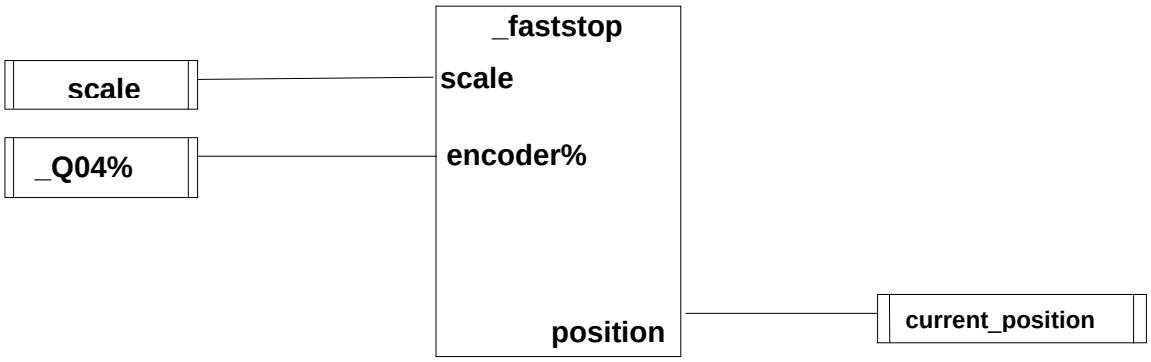


5. CONVERT ENCODER_LINES INTO FEET

```
( pos )  =  _getpos( scale, encoder_lines )  {

//   returns position in feet
//   ------------------------------------
//
//   returns the position in feet given a encoder value in enc_lines.
//
//   Inputs:      scale          =  converts feet into motor revolutions
//                encoder%       =  position in raw encoder lines
//
//  Outputs:    position  (feet)
//
//  Programmer:  James P. Lynch  2/8/2000

// initialisation code
encoder_lines = 4096.

FBbody
// main body code
pos = FLOAT( encoder% ) / ( scale * encoder_lines )

}  // ( pos )  =  _getpos( scale, encoder_lines )
```

position = _getpos(scale, _Q4%)

For example, to convert the current dynamic position **_Q4%** into feet:

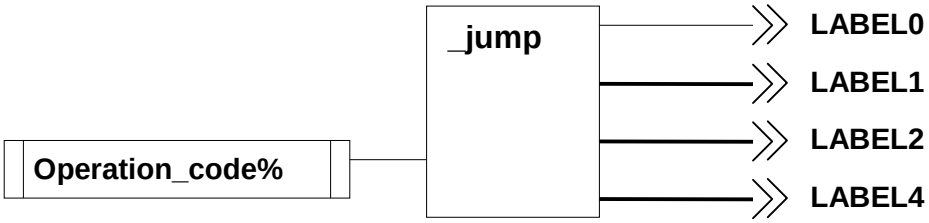


6. JUMP TO ONE OF FOUR LABELS

The current **SYPT** function block library has no “jump” block which would allow a jump to any of four labels based on a selection value. This is analogous to a memory decoder “chip select” circuit where only one chip select line is set and all others are zero. The following custom function block simulates this circuit.

```
_jump( ) {

//  _jump( ) -   set four discrete outputs according to input code
//               typically used to implement a case statement in LD/FBD diagrams
//               each output would be connected to a "jump to label" symbol
//
//  Inputs:   select%   =  bit select code
//
//  Outputs:      label0%    =  set when select% = 0
//                label1%    =  set when select% = 1
//                label2%    =  set when select% = 2
//                label3%    =  set when select% = 3
//                all off    =  when select% > 3
//

FBbody
// main body code
// start with all outputs cleared
label0% = 0
label1% = 0
label2% = 0
label3% = 0

// set proper output based on input code
if  ( select% = 0 ) then
          label0% = 1
elseif ( select% = 1 )  then
          label1% = 1
elseif  ( select% = 2 ) then
          label2% = 1
elseif  ( select% = 3 ) then
          label3% = 1
endif

}  // _jump( )
```

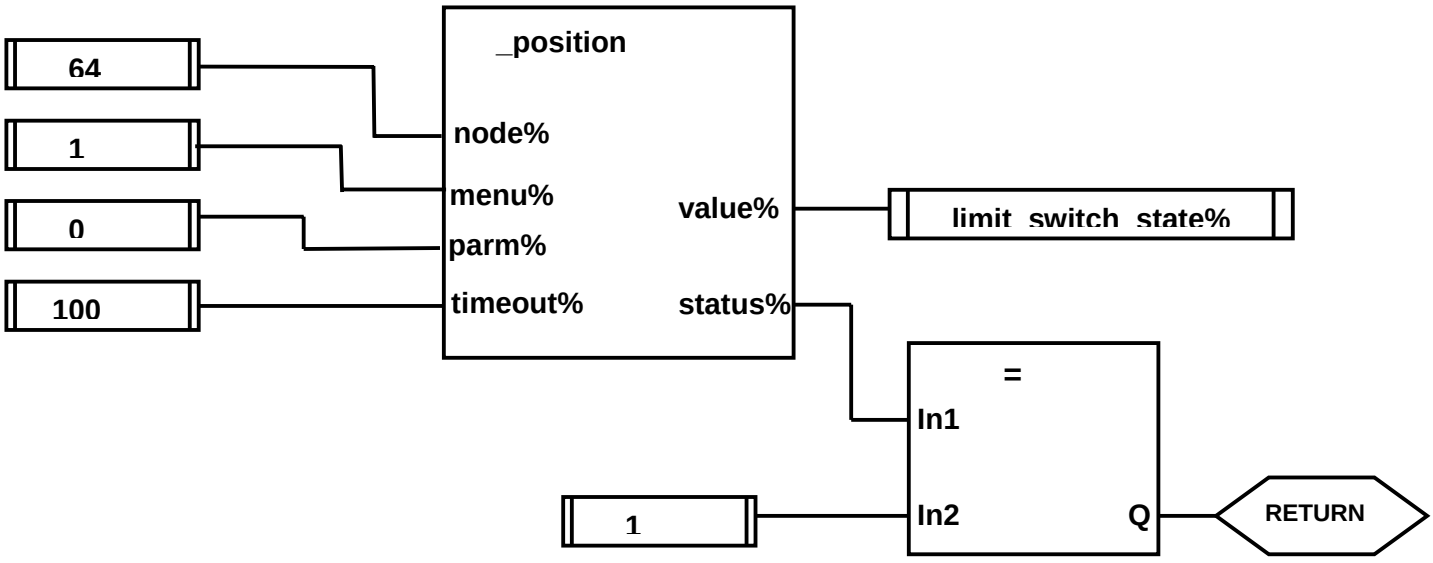
As an example, the integer variable “**operation_code%”** causes a jump to one of four labels.



7. READ A PARAMETER VIA CTNET

```
( value%, status% ) = _readCTNet( node%, menu%, parameter%, timeout% ) {

//  _readCTNet  -  read a parameter from another node with timeout
//
//  Inputs:    node%        =  destination CTNet node ID
//             menu%        =  destination menu #
//             parameter%   =  destination parameter #
//             timeout%     =  # of milliseconds to wait for a response
//
//  Outputs:  value%   =  returned parameter value (32-bit integer)
//              status%  =  success code  ( 1 = legal value returned,  4 = timeout )
//
// initialisation code

FBbody
// main body  code

// request a CTNet read operation
status% = RDNET(node%, menu%, parm%, timeout%)

// read the returned value (may be meaningless if the operation timed out)
value% = NETREPLY(0)

// if we timed out, force the returned value to zero
if  status% <> 1 then
        value% = 0
endif

}  // ( value%, status% )  =  _readCTNet( node%, menu%, parameter%, timeout% )
```

For example, to read Node 64, parameter #1.00 with a 100 msec timeout using the above function, the following diagram would suffice. Note that the output value (limit_switch_state%) will clear to zero if the CTNet read operation failed.



HOIST Application

Assume that you are the contractor for a high-rise building site. You need to set up a cargo lift to move building materials from ground level to any floor. A hoist drum mounted on the roof can be used to lift the cargo pallet and the Unidrive's built-in position controller makes it easy to command the hoist to move the cargo pallet from floor to floor.

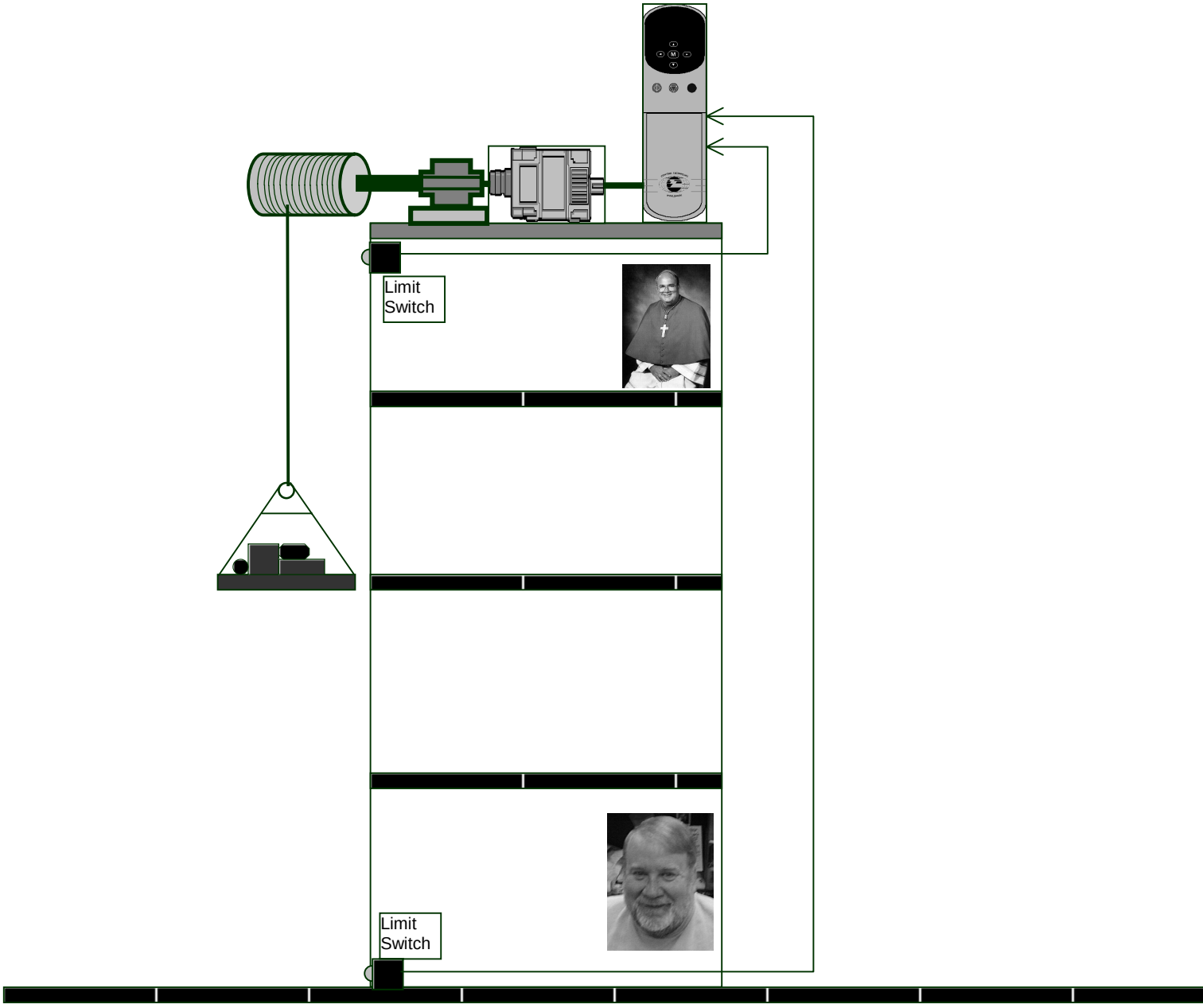


Figure 8. Construction Site Cargo Hoist

The project will require two nodes, a Unidrive operating in closed-loop vector mode **(node 12)** and a Beckhoff CTNet I/O unit **(node 64**) that has at least two discrete inputs. Therefore, the SYPT configuration screen will look like Figure 9 below:

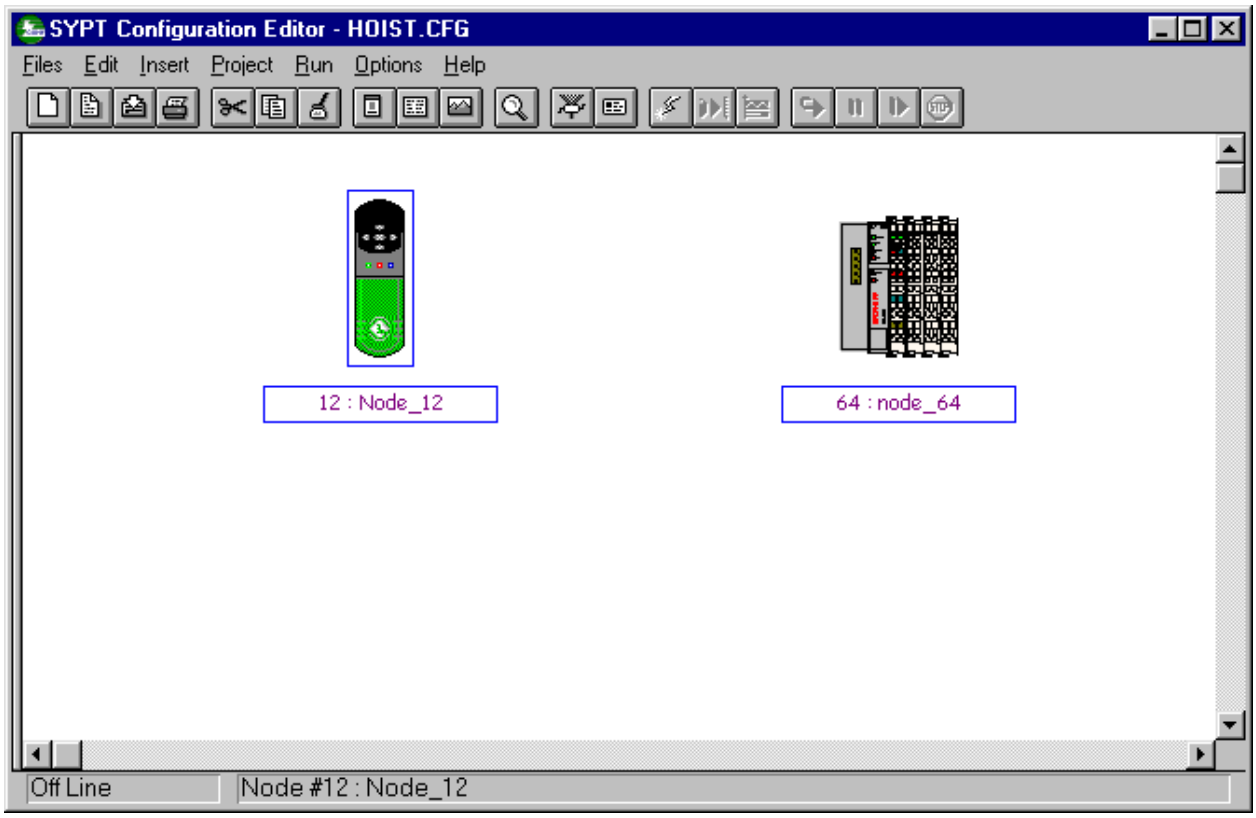


Figure 9. SYPT Configuration Screen

The Hoist application operates in three distinct modes: **normal position move**, **home down** and **home up**. An op-code parameter selects the desired operational mode.

In **normal position move** mode, the user can specify the absolute position that the hoist is to move to. There is no direction to specify, if the cargo pallet is at the roof or on the ground, it will automatically move in the correct direction to deliver the load. The user can specify the maximum speed that the pallet moves and the accel and decel time (which is the time it takes to go from zero speed to the maximum speed or vice versa).

In home up or home down mode, the hoist is moving up or down in order to properly locate the limit switches. When the limit switch is detected, the hoist stops rapidly and a “soft” limit is established that is 5 revolutions above or below the limit switches.

After the homing operation, anytime the hoist encounters the “soft” limits, it will stop without ever contacting the hard limit switches again. The hard limit switches are connected to two Beckhoff digital input bits. In practice, a simple photodetector would suffice for this purpose.

The Hoist application will be controlled by several Menu 20 registers. The reason these registers were used to control the application is that they are modifiable from the Unidrive’s keypad. A singular disadvantage is that these registers are limited to ±32000 which restricts the hoist’s operating range to 320 feet (assuming we enter everything in “times 100” motif).

CONTROL REGISTERS

To make the application easy to control from the Unidrive's keypad and display, eight menu 20 parameters are allocated to specify and initiate hoist operations:

#20.20 = execute bit (initiates all moves, will reset instantly to zero)

#20.21 = operation code 0 = no operation (resets to zero after operation is complete)
1 = normal move
2 = home down
3 = home up

#20.22 = target position (feet * 100) valid for normal moves only

#20.23 = maximum speed (feet/sec * 100) valid for normal moves only

#20.24 = accel time (sec * 100) valid for normal moves only

#20.25 = decel time (sec * 100) valid for normal moves only

#20.26 = diameter (feet * 100)

#20.27 = gear ratio (* 100)

The application also includes four "display" parameters that can be used by an HMI device to display the current speed and position. These "display" parameters are updated constantly in the background task.

DISPLAY PARAMETERS

#20.30 = current position (feet * 100)

#20.31 = current speed (feet/sec * 100)

#20.32 = low soft limit (feet * 100)

#20.33 = high soft limit (feet * 100)

Details of the three operational modes are as follows:

HOME_DOWN

- Command the position controller to go to some very large negative distance, say –9999. feet.
- While it's moving, monitor via CTNet the limit switches.
- When the low limit switch closes, immediately stop the position controller.
- Zero the encoder count _**Q08%**.
- Add 5 revolutions (in encoder lines) to the encoder count to establish the low soft limit.

HOME_UP

- Command the position controller to go to some very large positive distance, say +9999. feet.
- While it's moving, monitor via CTNet the limit switches.
- When the high limit switch closes, immediately stop the position controller.
- subtract 5 revolutions (in encoder lines) from the current encoder value _**Q08%** to establish the high soft limit.

NORMAL_POSITION_MOVE

- Command the position controller to go to the target position specified by the user. The user also specifies the maximum speed and the accel/decel times.
- While it's moving, monitor the "soft" limits.
- When either of the "soft" limits are exceeded, immediately stop the position controller.
- Things must be arranged so that the position controller can be commanded to go the other way in order to get off the "soft" limit (it may have decelerated a bit past the "soft" limits while it was stopping).
- Most professionally installed hoists have an electric brake system to prevent run-away. We can't simulate that in this example, but most designers release the electric brake when the motor is rotating and drawing some current. For all other cases, the spring-loaded brake will clamp the drum after completion of a move or after a power-failure.

Professional Solution

The following **SYPT** program demonstrates the use of the Control Techniques UD75 built-in Position Controller in a vertical hoist application. It can be used to raise and lower a load between two limit switches.

To develop this application with **SYPT**, a project called HOIST is created. A CTNet Beckhoff I/O unit configured with a 4-input discrete module reads the two limit switches.

The **SYPT** configuration screen directly below (Figure 10) shows two nodes; node 12 contains the application and node 64 contains the two Beckhoff I/O bits used as the limit switches. It's easy to simulate the limit switches with toggle switches, but in a normal application an optical proximity switch or a specialized limit switch would be employed.

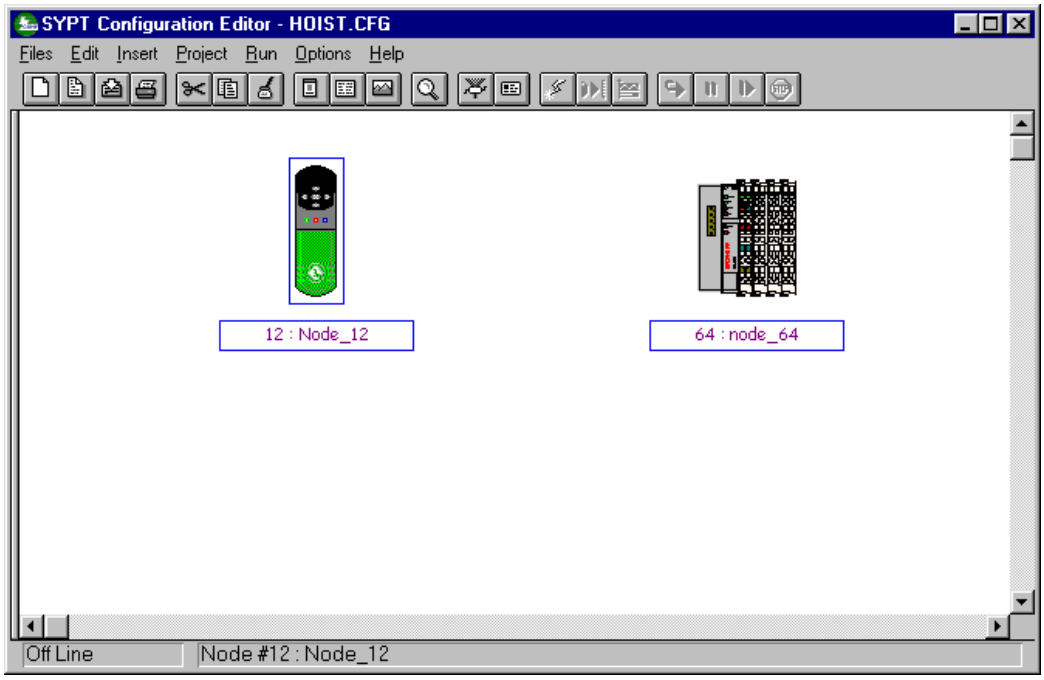


Figure 10. HOIST APPLICATION CONFIGURATION SCREEN

While the Initial Task sets up most of the drive's parameters for proper vector-mode operation, you are still responsible for getting the Unidrive into vector mode before downloading and starting the **SYPT** application.

```
#00.00  =  1254          // allow mode change and set to USA defaults

#00.48  =  CI.VECt       // select closed loop vector mode

RESET                    // hit the red RESET button
```

Every SYPT application should have a Notes section concisely explaining the application's purpose and any inputs and outputs used, as shown in Figure 11 below:

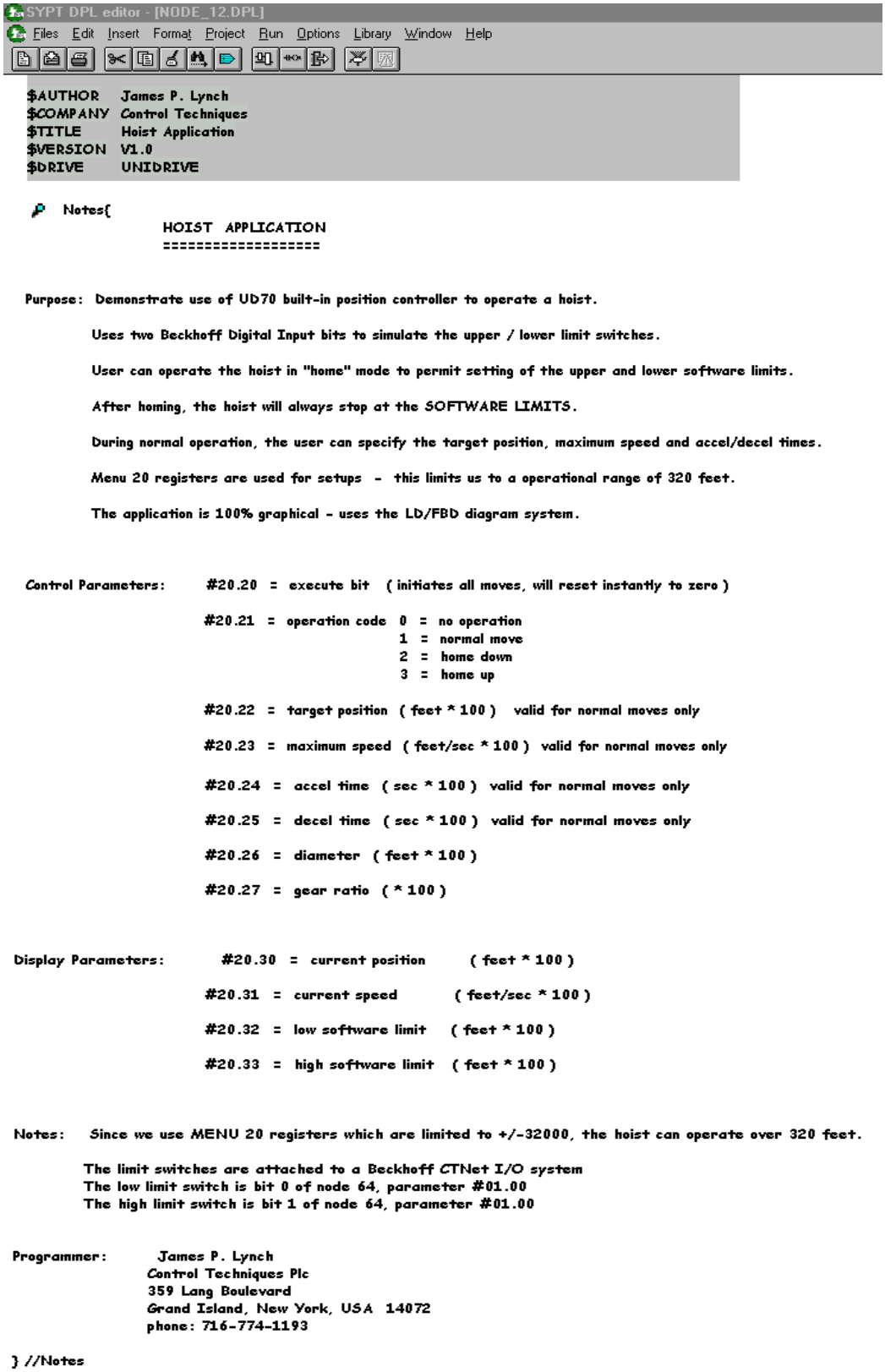


Figure 11. Hoist Application Notes Task

Next comes the specification of the “user-defined function blocks” as delineated earlier in this document. Note the extensive annotation in the specification of these custom function blocks.

```
(status%) = _pos_init (En%, Ctrl%, MaxSp%, PILim%, Pgain%, Igain%, FF%){

//  Initialize the UD70 Position Controller
//  -------------------------------
//
// INPUTS:
//
// En% -     Enable the initialisation.
// Ctrl% -   Control Word, each bit is configured as follows:
//           bit0 = Task, Position loop synchronisation. 0 = encoder task (#17.12=1)
//                                                       1 = speed task   (#17.12=2)
//           bit1 = Reset bit. if set   Q32%=Q20%=Q1%=Q2%=Q3%=Q8%=Q4%=Q9% = 0
//           bit2 = Feedback Selection. 0 = main encoder      (_Q20%.7=0)
//                                      1 = auxillary encoder  (_Q20%.7=1)
//           bit3 = Reserved
//           bit4-6 = Reference selection:-
//                                Position  Speed  Digital-Lock  CAM  None
//                    Ref. No.       1       2        3         4     0
//                    _Q20%.1 =      1       1        1         1     0
//                    _Q32%.2 =      0       0        1         0     0
//                    _Q32%.3 =      0       1        0         0     0
//                    _Q32%.4 =      0       0        0         1     0
//           Note:    The master and slave arrays must be declared along with a
//                    succesful CAMINIT before the CAM can be selected.
//
// MaxSp% - Maxspeed in RPM this will set parameters:- #01.06, #91.05.
// PILim% -  PI terms max output clamp entered as percentage of Max Speed.
//           (Q16%= (MaxSp%*(#3.21or #16.04)*PILIM%*4)/(60*100))
//           To ensure there is no rollover the max range is limited to +ve 0 - 300%.
// Pgain% -  Proportion Gain _Q5%.     (x1000)
// Igain% -  Integral Gain _Q6%.       (x1000)
// FF%-      Feedforward Gain _Q7%.  (x1000)  note: also referred to as "differential gain"
```

```
//
// OUTPUTS:
// status% - Function Block executed correctly without any errors.
//
//
//  Programmer:  Kevin Manton, Technical Manager SSPD  Created May 18, 1999
//
//  New Areas to be added
//  1. Enable should be edge triggered.
//  2. Sin/Cos cpr Detection Required, only detects standard encoder interfaces. DONE 29/07/99 KTM
//
//  Revisions:   none


// initialisation code
// ===================
Temp% = 0
Tempa%= 0

FBbody
// main body code
// ==============

IF En%=1 then

    // disable drive while setting up
    #6.15 = 0

    Tempa%=Ctrl%

     // position loop task selection
     // ======================
     //       0 = Encoder Task
     //       1 = Speed Task
     IF #17.12 <> (Tempa%.0 + 1) THEN
          #17.12 = (Tempa%.0 + 1)              // Enable Position to update with selected task.
          #00.00 = 1070                        // UD70 Reset
          DO WHILE Temp% < 50
               #10.38 = 100                    // Reset
               Temp%  = Temp% + 1
          LOOP
     ENDIF

     // position loop reset
     // ===============
     IF Tempa%.1=1 THEN
          // zero all _Qxx% registers that the position controller might use
          Q_pointer% = 7100
          do
               #Q_pointer% = 0
               Q_pointer% = Q_pointer% + 1
          loop while  Q_pointer% < 7141
     ENDIF

     // feedback selection
     // ===============
     // 0 = Standard Motor Feedback
     // 1 = Second Encoder Feedback
     _Q20%.7 = Tempa%.2
```

```
// reference selector
// ================
// 0 = No Reference selection. (_Q9% only)
// 1 = Position Reference.
// 2 = Speed Reference.
// 3 = CAM Profiler reference.
Temp% = (Tempa% & 0x70)/16              // Mask reference selector bits
                                        // and shift right x4

IF Temp% = 1 THEN                       // Enable Position reference
     _Q20%.1 = 1
     _Q32%.2 = 0
     _Q32%.3 = 0
     _Q32%.4 = 0
ELSEIF Temp% = 2 THEN                   // Enable Speed reference
     _Q20%.1 = 1
     _Q32%.2 = 0
     _Q32%.3 = 1
     _Q32%.4 = 0
ELSEIF Temp% = 3 THEN                   // Enable Digital Lock
     _Q20%.1 = 1
     _Q32%.2 = 1
     _Q32%.3 = 0
     _Q32%.4 = 0
ELSEIF Temp% = 4 THEN
     _Q20%.1 = 1                        // Enable CAM reference
     _Q32%.2 = 0
     _Q32%.3 = 0
     _Q32%.4 = 1
ELSE
    _Q20%.1 = 0                     // Disable reference
    _Q32%.2 = 0
    _Q32%.3 = 0
    _Q32%.4 = 0
ENDIF

// Max Speed
// =========
// Set in RPM
#91.05 = MaxSp%                     // Default MAX Speed Resolution.(RPM)
#01.06 = MaxSp%                     // Default MAX Speed.(RPM)

// PI Clamp
// ========
// Set in %.
#03.21 = 1024                           // 1024 PPR Encoder
PILim%=MIN((MAX(PILim%,0)),300)         // Clamp range Positive 0-300%.

// counts per rev.(required as output is in rpm)
    IF _Q20%.7 = 1 THEN                 // Determine Feedback Pulses Per Revolution.
    Temp%=(INT(#16.04)*4)               // Second Encoder
ELSEIF _Q20%.7 = 0 THEN
    Temp%=(#03.21*4)                    // Main Motor Encoder
ELSEIF _Q20%.7 =0 AND #16.01 = 4 THEN   // SIN/COS
                                        // Note this only applicable for sysfile V2.7.4
    IF _Q19% = 0 THEN Temp% = 16384     // 14 bit
    IF _Q19% = 1 THEN Temp% = 32768     // 15 bit
    IF _Q19% = 2 THEN Temp% = 65536     // 16 bit
    IF _Q19% = 3 THEN Temp% = 131072    // 17 bit
    IF _Q19% = 4 THEN Temp% = 262144    // 18bit max
ENDIF
```

```
    Tempa% = (MaxSp%*Temp%) / (60)              // Max Speed in Counts
    _Q16%  = (Tempa%*PILim%)/(100)              // Set PI Output Clamp.

    // PID gains
    // ========
    _Q5% = Pgain%
    _Q6% = Igain%
    _Q7% = FF%

    // other parameter settings
    // ====================
    #91.03 = 3                          // Enable fast speed update.
    #01.10 = 1                          // Enable Bipolar speed ref.
    #01.14 = 3                          // Select Preset Speed reference.
    #01.15 = 1                          // Select Preset Speed #1.
    #02.02 = 0                          // Disable Drive Ramps.
    #02.04 = 1                          // Drive Ramps set to Fast Ramp.
    #03.05 = 1                          // "At Zero Speed" threshold
    #03.08 = 2000                       // Overspeed Trip Threshold
    #03.10 = 1000                       // Speed Loop Proportional Gain
    #03.11 = 100                        // Speed Loop Integral Gain
    #03.12 = 0                          // Speed Loop Differential Gain
    #03.20 = 0                          // Disable Hard Speed Reference
    #03.30 = 1                          // Speed Feedback Filter
    #04.07 = 175                        // Symmetrical Current Limit
    #04.12 = 3                          // Current Demand Filter Time Constant
    #05.18 = 4                          // Switching Frequency  ( 12 MHz )
    #06.01 = 2                          // No Stopping Ramps
    #06.04 = 3                          // PLC Sequencing Mode  ( PLC mode )
    #06.08 = 0                          // Hold Torque at Zero Speed
    #06.30 = 0                          // stop the drive
    #08.13 = 0                          // disconnect F2
    #08.16 = 0                            // disconnect F3
    #08.19 = 0                            // disconnect F4
    #08.21 = 0                            // disconnect F5
    #08.23 = 0                            // disconnect F6
    _Q20%.6= 1                            // Enable output of PID
    #06.15 = 1                            // Drive Enable

    status% = 1                           // Status code OK.

ELSE

    status% = 0                           // If NOT enabled Reset status code.

ENDIF

} //(status%) = _pos_init (En%, Ctrl%, MaxSp%, PILim%, Pgain%, Igain%, FF%)
```

```
_position (targ, maxsp, accel, decel, scale){

//    Set Up a Normal Position Move
//    ----------------------------
//
//    Sets up a normal position move using the UD70's built-in position controller.
//
//    Inputs:    targ         =  target position in feet
//               maxsp        =  maximum speed in feet/sec
//               accel        =  accel time in seconds
//               decel        =  decel time in seconds
//               scale        =  converts feet into motor revolutions
//
//  Outputs:    none
//
//  Programmer:  James P. Lynch  March 23, 2000

// initialisation code
encoder_lines = 4096.0
rated_speed = #1.06

FBbody
// main body code
_Q32%.3 = 0                                                          //  _Q2% = target position
_Q20% = 0x42                                                         //  enable position reference
maxspeedrpm  = maxsp * scale * 60.0                                  //  convert max speed to rpm
if  maxspeedrpm > rated_speed  then  maxspeedrpm = rated_speed       //  clamp max speed
if  accel <= 0.0  then accel = 0.5                                   //  prevent "divide by zero"
_Q12%= INT ( (maxspeedrpm * 68.2666) / accel )                       //  accel rate
if  decel <= 0.0  then decel = 0.5                                   //  prevent "divide by zero"
_Q13%= INT ( (maxspeedrpm * 68.2666) / decel )                       //  decel rate
_Q14%= INT ((maxspeedrpm * encoder_lines) / 60.0)                    //  max speed

#6.15 = 1                                                            //  enable drive
#6.30 = 1                                                            //  start the drive
_Q2%  = INT( targ * scale * encoder_lines )                          //  enter  target position

} //_position (targ, maxsp, accel, decel, scale)
```

```
_stop (){

//    Stop drive and clamp position controller
//    ----------------------------------------------------------------
//
//    This function is usually called after normal completion of a position move.
//
//    It clamps the position controller ( sets target and post-ramp reference equal to the encoder )
//    This ensures that there's no error signal and the position controller is completely stopped.
//
//    Finally, it stops the drive.
//
//    Inputs:    none
//
//   Outputs:    none
//
//  Note:  this function sets target (_Q2%)  to feedback (_Q8%),
//           slave position (_Q4%) to feedback (_Q8%) and stops the drive
//
//  Programmer:  James P. Lynch  2/3/2000

// initialisation code

FBbody
// main body code

_Q2% = _Q8%                 // clamp the position controller
_Q4% = _Q8%                 // clamp the position controller
#6.30 = 0                   // stop the drive

} //_stop ()
```

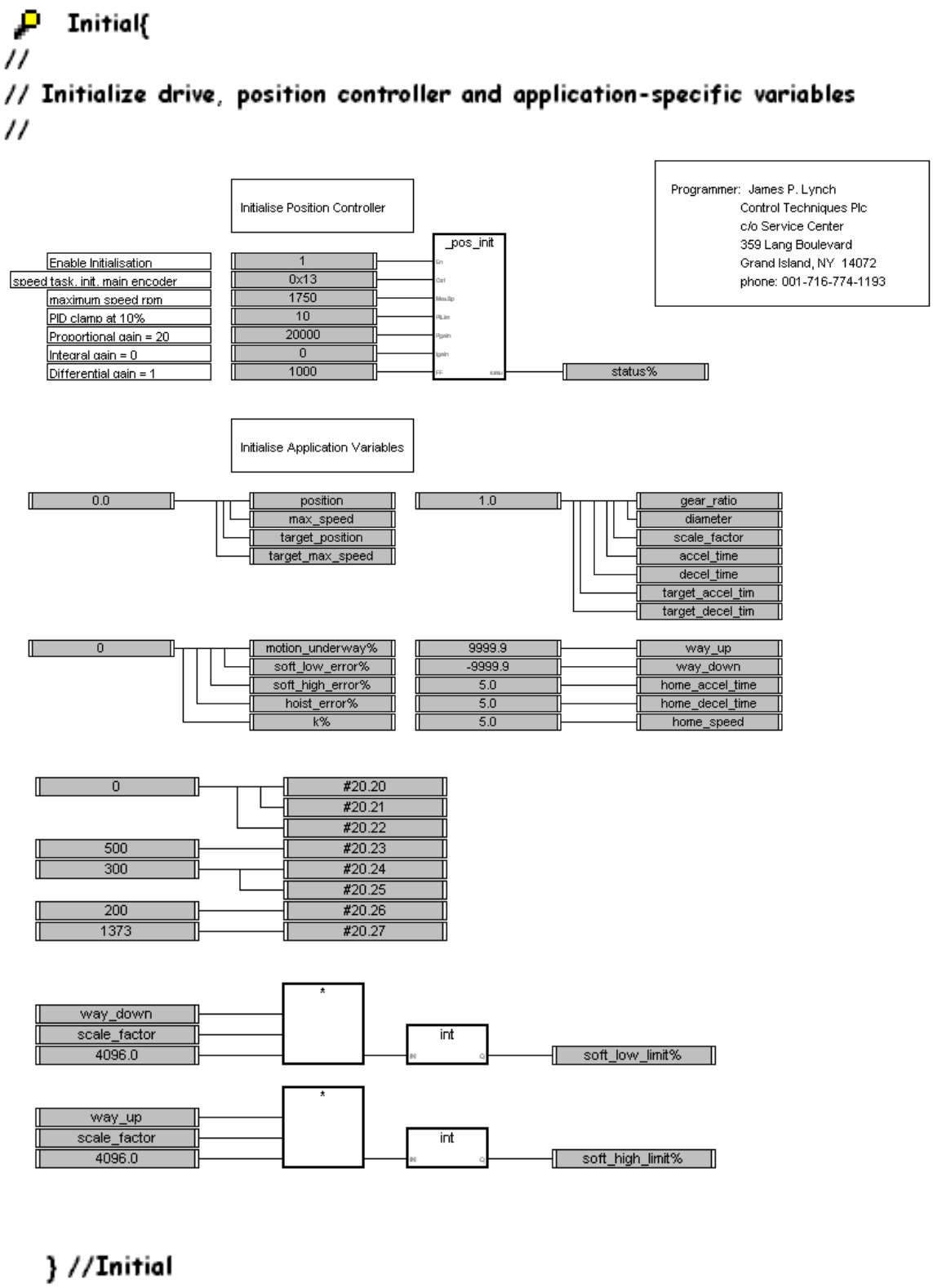
```
_faststop (accel, decel){

//    fast stop the position controller
//    ------------------------------------------------
//
//    Stops the position controller - usually before the position move is finished.
//
//    Sets max speed ( _Q14% ) to zero, this causes a controlled ramp to zero speed.
//    When zero speed detected ( #10.03 = 1 ), clamps the position controller and stops the drive.
//
//    Inputs:    accel  =  user-defined accel rate in sec
//                  decel  =  user-defined decel rate in sec
//
//   Outputs:    none
//
//  Programmer:  James P. Lynch  2/3/2000

// initialisation code
encoder_lines = 4096.0

FBbody
// main body code

max_speed = FLOAT( _Q14%  * 60 ) / encoder_lines          // calculate the current max_speed
if  accel <= 0.0  then accel = .5                         // prevent "divide by zero"
_Q12%= INT ( (max_speed * 68.2666) / accel )              // accel rate
if  decel <= 0.0  then decel = .5                         // prevent "divide by zero"
_Q13%= INT ( (max_speed * 68.2666) / decel )              // decel rate
_Q14%= 0                                                  // zero max speed, this forces a ramp to zero
do while  #10.03 = 0  loop                                // now wait for the drive's speed to hit zero
_stop( )                                                  // clamp the position loop and stop the drive

} //_faststop (accel, decel)
```

```
(pos) = _getpos (scale, encoder%){

//    returns position in feet
//    ------------------------------------
//
//    returns the position in feet given a encoder value in enc_lines.
//
//    Inputs:    scale          =  converts feet into motor revolutions
//                  encoder%    =  position in raw encoder lines
//
//  Outputs:    position  (feet)
//
//  Programmer:  James P. Lynch  2/8/2000

// initialisation code
encoder_lines = 4096.0

FBbody
// main body code
pos = FLOAT( encoder% ) / ( scale * encoder_lines )

} //(pos) = _getpos (scale, encoder%)
```

```
(label0%, label1%, label2%, label3%) = _jump (select%){
//  _jump( )  -   set four discrete outputs according to input code
//                typically used to implement a case statement in LD/FBD diagrams
//                each output would be connected to a "jump to label" symbol
//
//  Inputs:    select%   =  bit select code
//
//  Outputs:  label0%    =  set when select% = 0
//            label1%    =  set when select% = 1
//            label2%    =  set when select% = 2
//            label3%    =  set when select% = 3
//            all off    =  when select% > 3
//
//  Programmer:  James P. Lynch  2/8/2000

FBbody
// main body code
// start with all outputs cleared
label0% = 0
label1% = 0
label2% = 0
label3% = 0

// set proper output based on input code
if  ( select% = 0 ) then
     label0% = 1
elseif ( select% = 1 )  then
     label1% = 1
elseif  ( select% = 2 ) then
     label2% = 1
elseif  ( select% = 3 ) then
     label3% = 1
endif
} //(label0%, label1%, label2%, label3%) = _jump (select%)
```

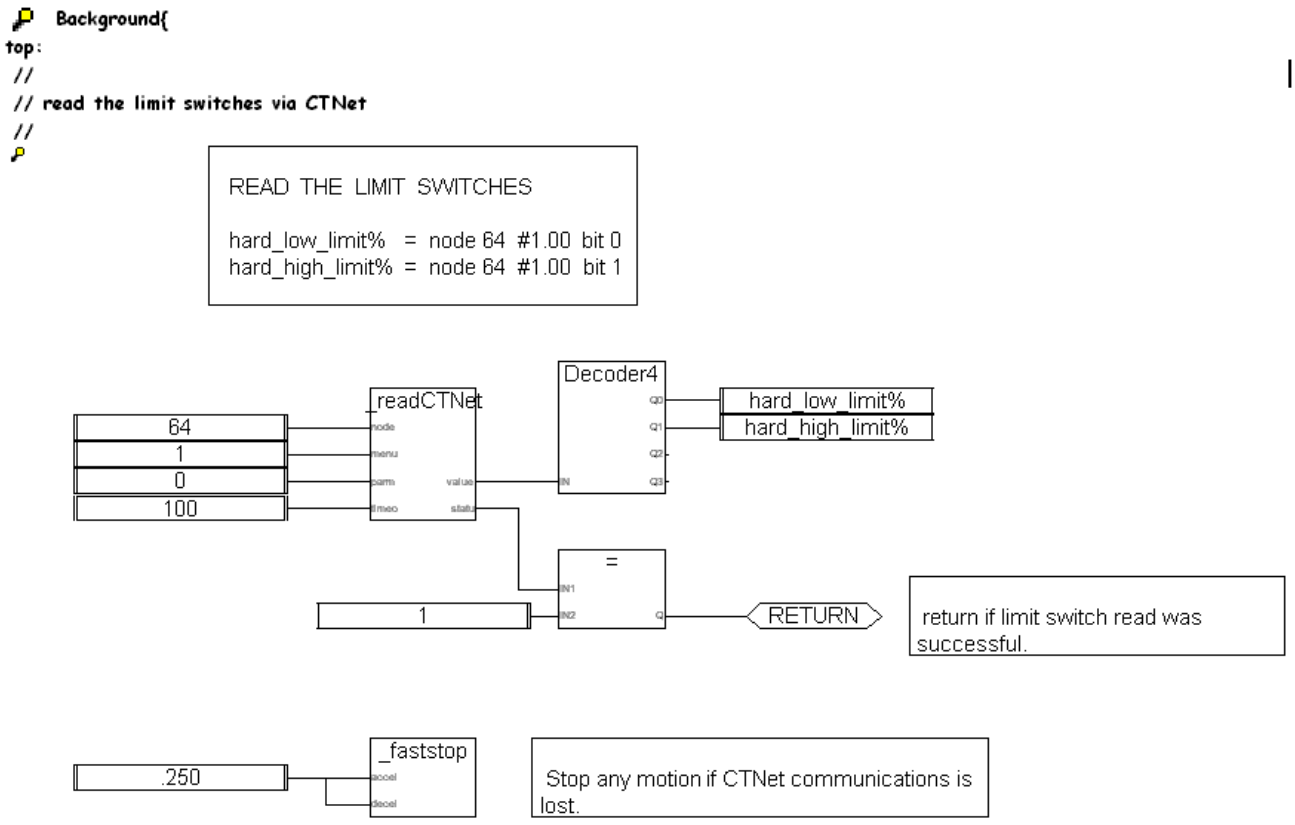
```
(value%, status%) = _readCTNet (node%, menu%, parm%, timeout%){
//  _readCTNet  -  read a parameter from another node with timeout
//
//  Inputs:    node%         =  destination CTNet node ID
//               menu%         =  destination menu #
//               parameter%  =  destination parameter #
//               timeout%      =  # of milliseconds to wait for a response
//
//  Outputs:  value%   =  returned parameter value (32-bit integer)
//                status%  =  success code  ( 1 = legal value returned,  4 = timeout )
//
//
//  Programmer:  James P. Lynch  2/8/2000

// initialisation code

FBbody
// main body  code
// request a CTNet read operation
status% = RDNET(node%, menu%, parm%, timeout%)
// read the returned value (may be meaningless if the operation timed out)
value% = NETREPLY(0)
// if we timed out, force the returned value to zero
if  status% <> 1 then
     value% = 0
endif
} //(value%, status%) = _readCTNet (node%, menu%, parm%, timeout%)
```

As a purely academic exercise, the rest of the **SYPT** application is entirely graphical, specifically using the **LD/FBD** graphic editor. The **INITIAL** task looks as follows:



The rest of the application is a single Background Task. The first thing to do is to read the current state of the Limit Switches. While the Beckhoff unit can be set up to move this data via cyclic data exchanges, it's instructional to see how a "one-shot" CTNet query could be accomplished graphically.

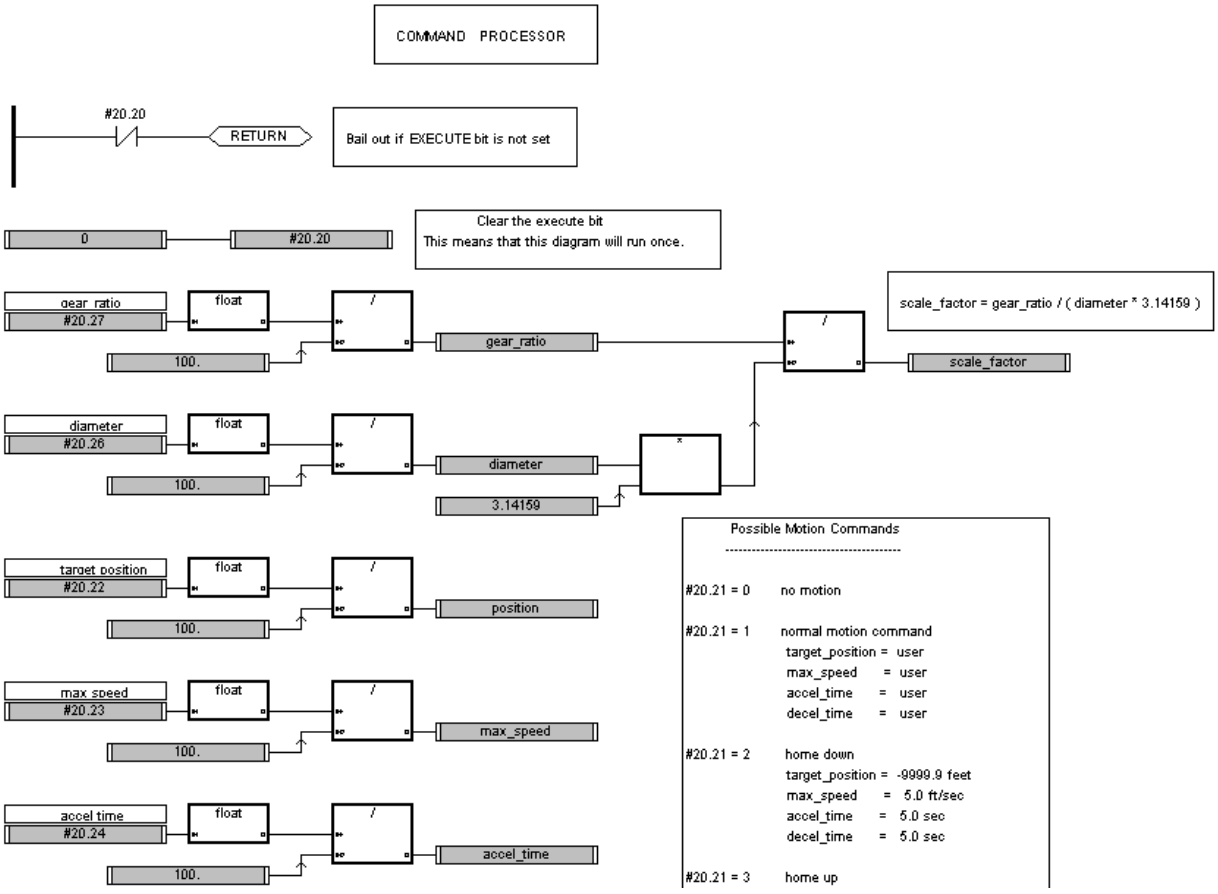
Note that if CTNet communication is ever interrupted, the Hoist will do a panic stop.



Users should be aware that the application will "hang" in the **_readCTNet()** function block for 8 to 12 milliseconds while the CTNet communication progresses. If the network connection completely failed, the application will hang for 100 msec (the timeout) and then panic stop.

The sole purpose of this diagram is to read the limit switch bits every background scan.

The next part of the background task is the **Command Processor** diagram. Its purpose to determine if the “execute bit” (#20.20) has been set. If so, the execute bit is promptly cleared and this guarantees that the command analysis and execution is run once and only once.



The one rung of logic at the top of the diagram detects if the “execute” bit was set. If not, the rest of the diagram is skipped.

If the “execute bit” was set, then the setup parameters in menu 20 are read and converted from “times 100” integer values into floating point. The scale factor is also calculated.

Note the liberal use of intermediate registers in the diagram. There are two good reasons for this. First, it makes it easier to debug since these intermediate registers animate with the current values while running. Second, the **SYPT** DPL compiler is prone to failing due to an “expression too complex” error on large diagrams and the use of intermediate registers helps prevent the creation of overly complex expressions.

When the “execute bit” is set (#20.20 = 1), it is assumed that the user has previously specified the gear ratio, diameter, target position, maximum speed and accel/decel rates. It’s also assumed that the user has specified the type of motion desired (normal motion, home down, home up or none) before setting the “execute bit”.

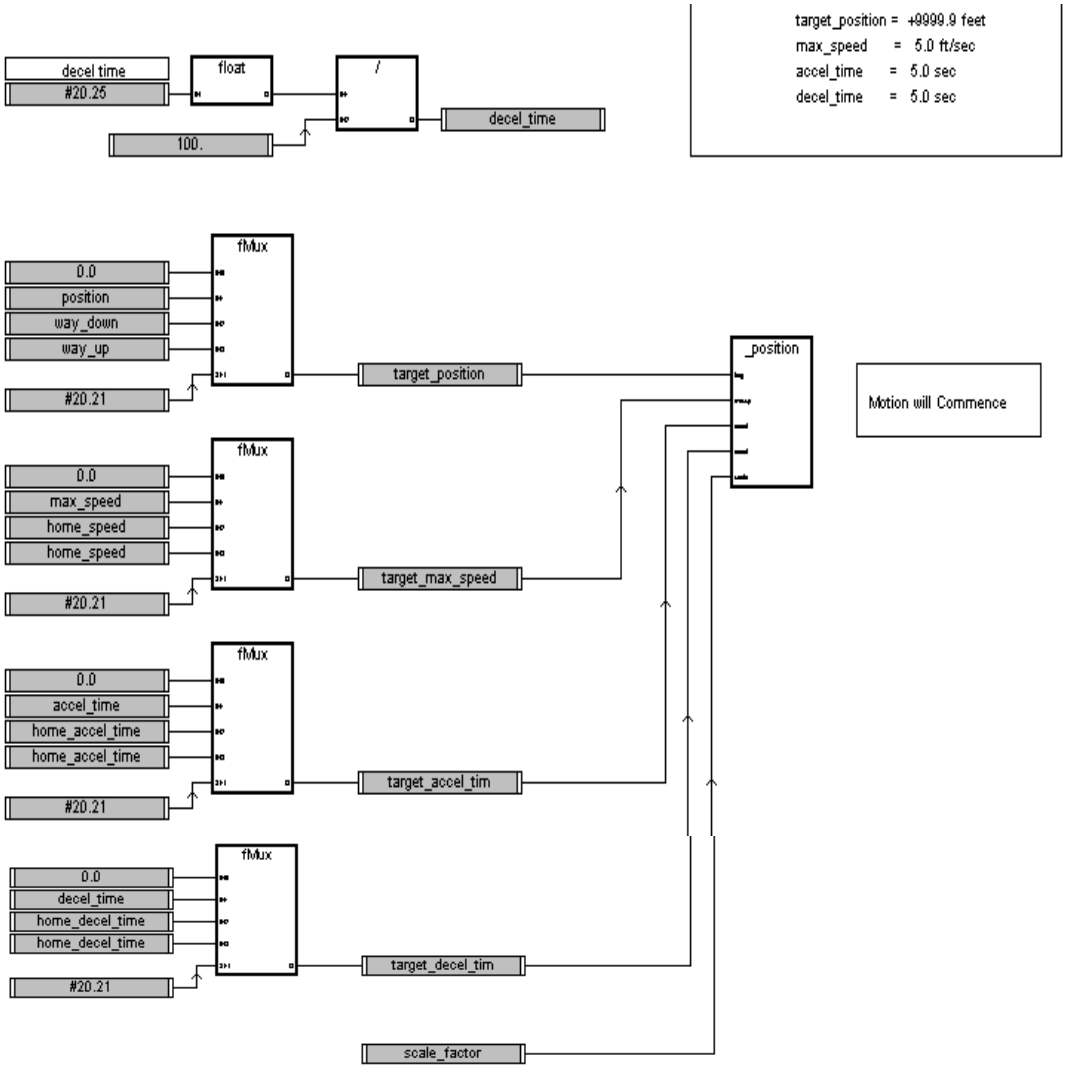
The second part of the Command Processor diagram actually causes motion to start. In a normal motion request, the **_position()** custom block will get the target position, max speed and accel/decel rates as specified by the user in the menu 20 registers.

In a “home down” request, the target position will be –9999.0 feet and the maximum speed is forced to 5 feet/sec (home_speed) and the accel and decel rates are forced to 5 seconds (home_accel_time and home_decel_time).

In a “home up” request, the target position will be +9999.0 feet and the maximum speed is forced to 5 feet/sec (home_speed) and the accel and decel rates are forced to 5 seconds (home_accel_time and home_decel_time).

Some designers elect to use “speed control” when homing up or down, but this application uses position control for the homing operation with a target position set to a very large value. It keeps things simple!

The second part of the Command Processor calls the **_position()** custom function block and motion will commence.



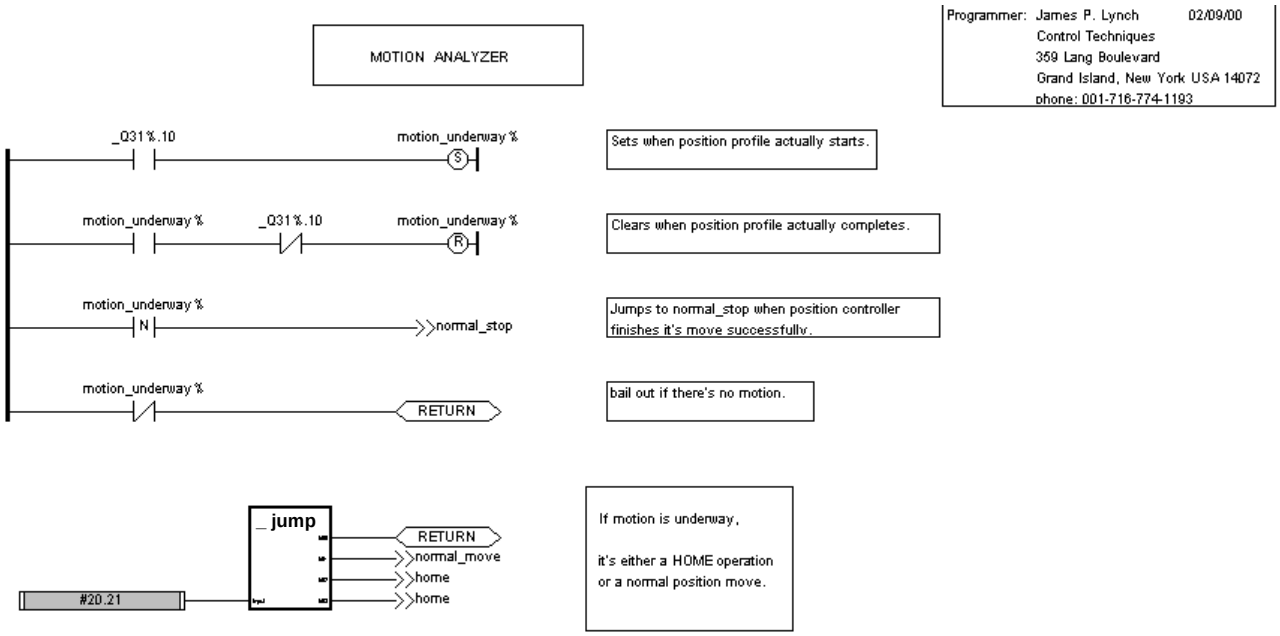
Once motion has started, the background task must monitor it.

First, the background task must detect normal end of motion (position profile completed) so that the drive can be turned off.

Second, the background task must detect hitting the software limits and if so, panic stop the position controller. When this happens, it must still be possible to command the position controller to go in the reverse direction, thus getting off the limit switches.

Finally, if the commanded motion is “home-up” or “home-down”, the background task must implement the specifics of those procedures.

The diagram to monitor and act on events such as limit switch detection, etc. is called the **“motion analyzer”.**



The four rungs of ladder logic in the motion analyzer diagram above attempt to determine when position controller motion starts and stops. The variable **motion_underway%** is associated with set and reset contacts. The **motion_underway%** semaphore is set when the “profile-in-progress” bit sets and is cleared when the “profile-in-progress” bit clears.

The falling edge of **motion_underway%** is used to jump to the “normal_stop” part of the diagram that will simply stop the drive after normal completion of the move. If there’s no motion at all (**motion_underway% = 0**), then the rest of the diagram is skipped.

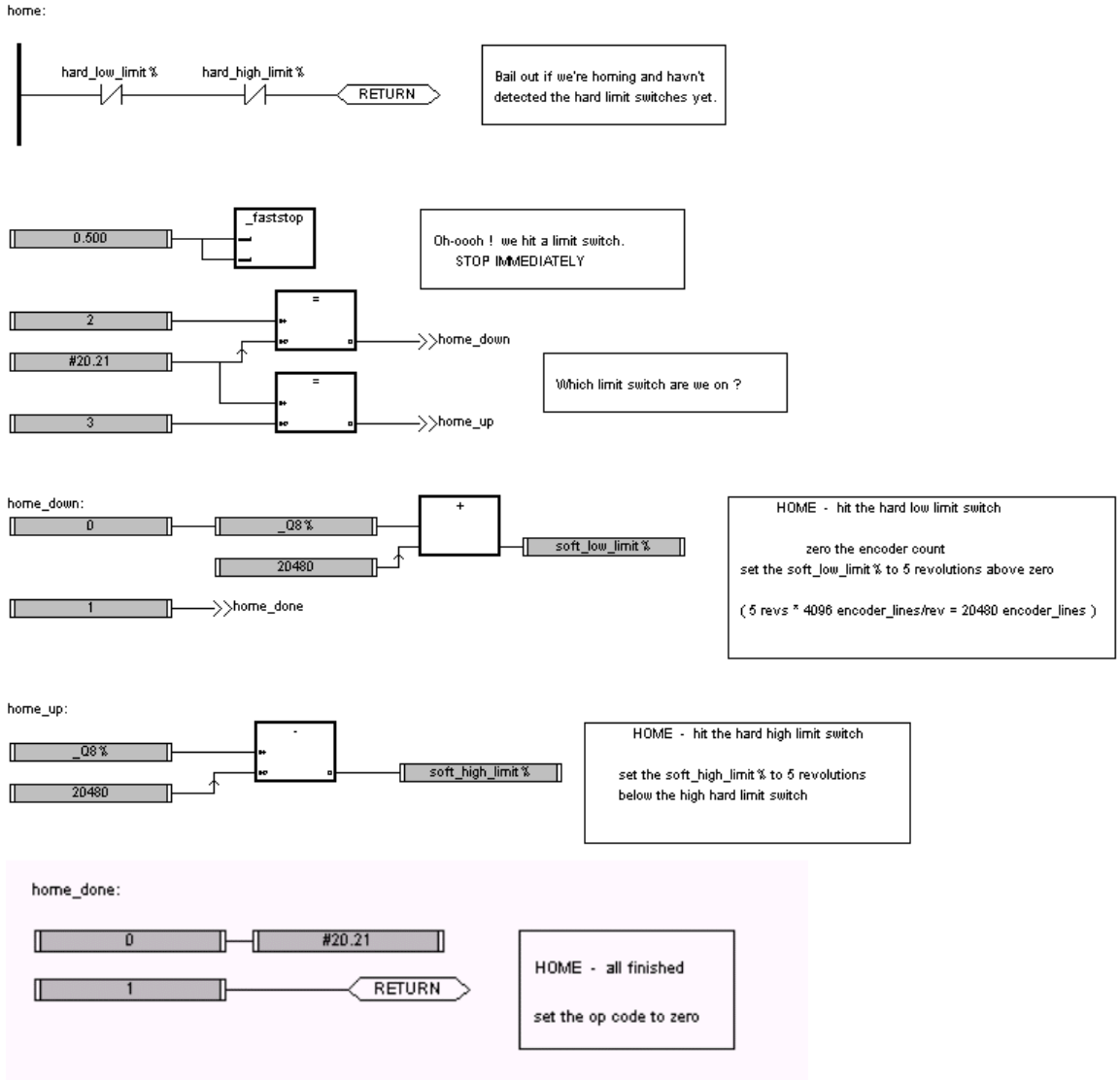
If there is motion and it is underway (**motion_underway% = 1**), then the **_jump()** function block is used to determine what to do next.

If the user has specified a normal move (#20.21 = 1), then the jump block takes you to the section of the diagram labeled “**normal_move**”. Here the program will check if the software limits have been reached.

If the user specified a “home-down” or a “home-up” operation, then the jump block takes you to the section of the diagram labeled “**home**”.

In the homing operation (either home-down or home-up) the system checks for either of the hard limit switches closing (these switches are read from the Beckhoff I/O unit repetitively). If neither limit switch has been encountered, then the rest of the diagram is skipped.

If we do hit one of the limit switches, then the hoist is stopped immediately by calling the **_faststop()** function block with a 0.500 second decel time.



Once we have fast stopped, then the appropriate soft limit must be set up. Parameter **#20.21** is checked to determine if we're homing down or homing up. If we're homing down, then the encoder count **_Q8%** is set to zero and the **soft_low_limit%** is set to five revolutions above zero.

If we're homing up, then the **soft_high_limit%** is set to five revolutions below the encoder count in **_Q8%.** Remember that one revolution is 4096 encoder lines so five revolutions are 20480 encoder lines. When the home operation is done, the operation code **#20.21** is cleared to zero to signify that the operation has completed.

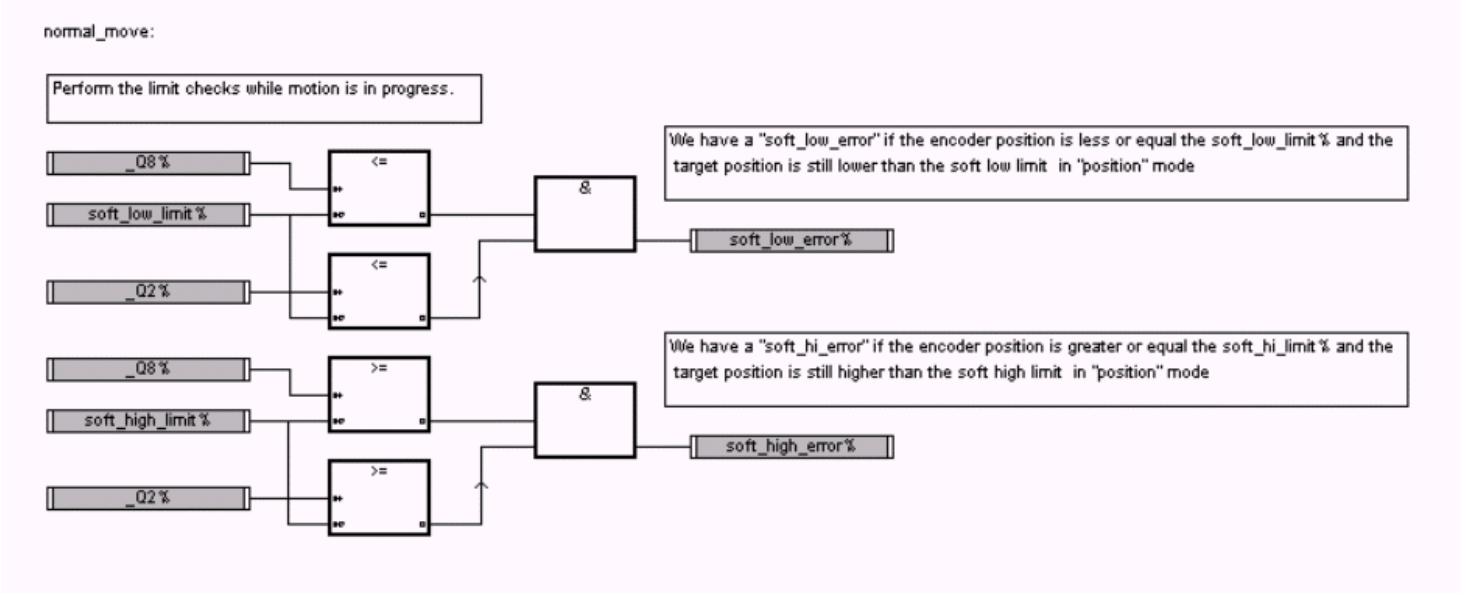
During normal operation the hoist is never permitted to exceed the soft limits.

During normal operation, the motion analyzer checks that the soft limits have not been exceeded. The reader might be inclined to assume that when the soft limit has been detected, the hoist must always fast stop. If this were the case, you would never be able to get the hoist away from the soft limit since it would always stop immediately anytime you re-commanded it to move.

The best approach is to create variables "**soft_low_error%**" and "**soft_high_error%"** and use these to determine if the hoist must be fast stopped.

For example, if the hoist is moving down and is at or below the **soft_low_limit%**, that is a bona-fide "**soft_low_error%**" and the hoist must be stopped. If, however, the hoist is moving up and is still at or below the **soft_low_limit%,** this is not an "**soft_low_error%**" and the hoist must be allowed to continue moving so it can get off the **soft_low_limit%.**

Likewise, if the hoist is moving up and is at or above the **soft_high_limit%**, that is a bona-fide "**soft_high_error%"** and the hoist must be stopped. If, however, the hoist is moving down and is still at or above the **soft_high_limit%,** this is not a "**soft_high_error%"** and the hoist must be allowed to continue moving so it can get off the **soft_high_limit%.**

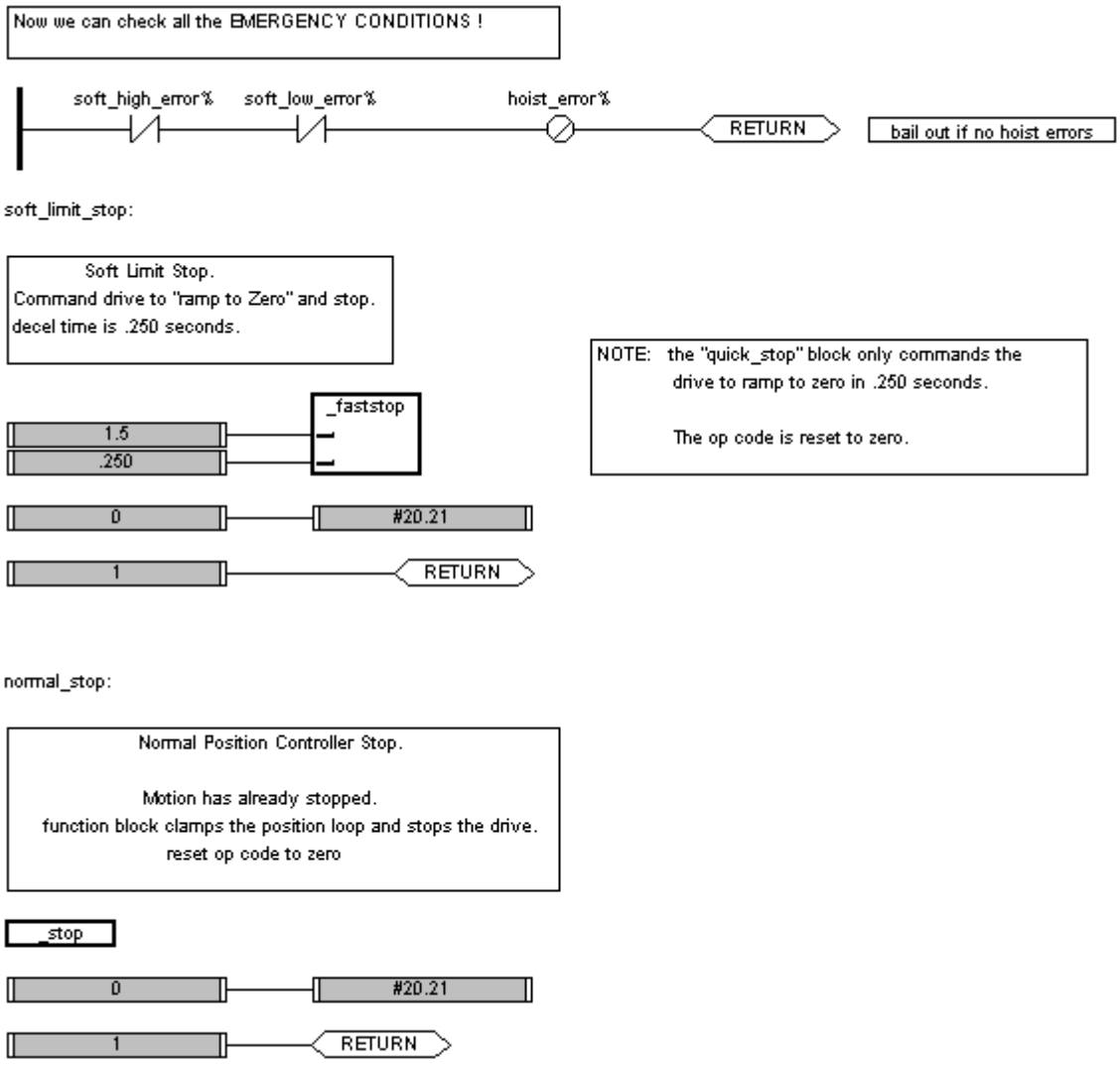


Once the **soft_high_error%** and **soft_low_error%** semaphores have been determined, the motion analyzer can check these to see if the hoist must be stopped. To review, the semaphores are:

error condition	reason
soft_low_error%	below the soft_low_limit% and the hoist is attempting to go down.
soft_high_error%	above the soft_high_limit% and the hoist is attempting to go up.

If either of these conditions is true, the hoist must be stopped immediately! Otherwise, the rest of the diagram may be skipped.

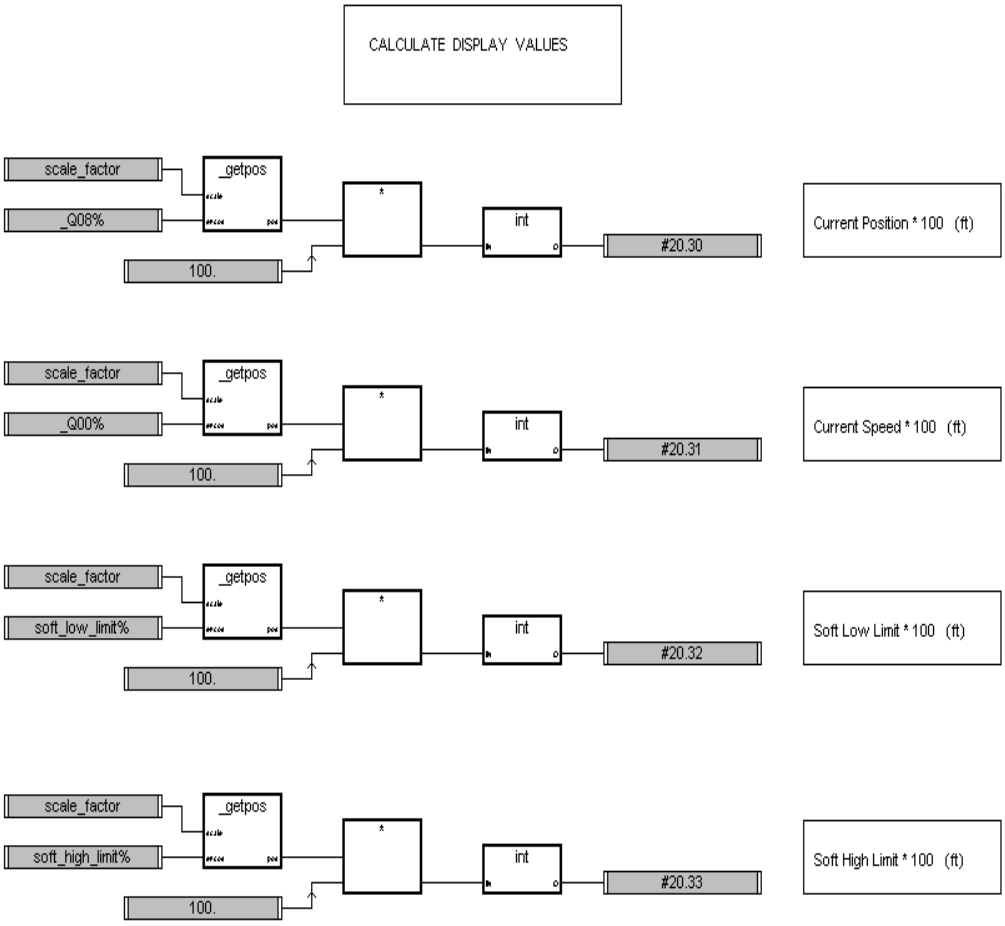
In the diagram below, the error semaphores are checked by a simple ladder rung. A fast stop is executed if either semaphore is set. The diagram also includes the “normal stop” operation that is accessed at the top of the motion analyzer diagram.



Note that we clear the “execute bit” **#20.20** when the motion command has been accepted and we clear the “operation code” **#20.21** when the motion has finished. This gives the user a visual indication that the operation has finished and a new command can be given.

The last part of the background task is the “display values” section that provides feedbacks in some menu 20 registers. The current position, current speed and the soft limits are displayed in “feet * 100” motif.

```
//
//  calculate the "display" values
//
```




```
goto top: // main background loop
} //Background
```

RUNNING THE APPLICATION

To run this position control application, you must have a Unidrive attached to a vector motor outfitted with a 1024 ppr encoder. The Unidrive must, of course, have a UD75 coprocessor with a CTNet cable attached to the **SYPT** workstation and a Beckhoff CTNet I/O module. The Beckhoff I/O system should have a KL1104 4-input discrete module with the limit switches attached to bit 0 (low limit) and bit 1 (high limit). The UD75 should be set to node 12 while the Beckhoff I/O subsystem should be set to node 64.

The Unidrive must be set into vector mode and the position controller must be enabled.

```
#00.00  =  1254          // allow mode change and set to USA defaults
#00.48  =  CI.VECt       // select closed loop vector mode
RESET                    // hit the red RESET button
#17.12  =   2            // enable position controller synchronized to speed task
#17.00  =  1000          // save all parameters
RESET                    // hit the red RESET button
#17.00  =  1070          // restart the UD75
RESET                    // hit the red RESET button
```

The **SYPT** project must be built and downloaded and the application started.

To properly operate the hoist, the user must first perform the “home-down” and “home-up” operations. These are required to locate the limit switches and establish the soft limits. The low limit switch is also established as “ground zero” and all normal position moves are referenced from that point.

1. HOME DOWN

Home Down refers to searching for the low limit switch. The following parameters will setup and execute the home-down operation.

```
#20.20  =  0          //  execute bit not used yet
#20.21  =  2          //  command = home-down
#20.22  =  0          //  target position (not used in home-down)
#20.23  =  0          //  max speed     (not used in home-down)
#20.24  =  0          //  accel time    (not used in home-down)
#20.25  =  0          //  decel time    (not used in home-down)
#20.26  =  250        //  diameter = 2.5 feet  (x100)
#20.27  =  1373       //  gear_ratio = 13.73   (x100)

#20.20  =  1          // execute bit will clear immediately and hoist will start moving down

wait a bit then close the low limit switch

drive stops quickly and the low soft limit is established and the low limit switch position
becomes “ground-zero”.
```

2. HOME UP

Home Up refers to searching for the high limit switch. The following parameters will setup and execute the home-up operation.

```
#20.20  =  0          //  execute bit not used yet
#20.21  =  3          //  command = home-up
#20.22  =  0          //  target position (not used in home-down)
#20.23  =  0          //  max speed      (not used in home-down)
#20.24  =  0          //  accel time     (not used in home-down)
#20.25  =  0          //  decel time     (not used in home-down)
#20.26  =  250        //  diameter = 2.5 feet  (x100)
#20.27  =  1373       //  gear_ratio = 13.73   (x100)

#20.20  =  1          // execute bit will clear immediately and hoist will start moving up

wait for the hoist to move up approximately 50 feet and then close the high limit switch

drive stops quickly and the high soft limit is established
```

3. NORMAL POSITION MOVE

Normal Position Move refers to moving in a controlled fashion to a new position between the soft limits. The following parameters will setup and execute a normal position move to **35.0** feet with a max speed of **1.5** feet/sec and an accel/decel time of **8** seconds.

```
#20.20  =  0          //  execute bit not used yet
#20.21  =  1          //  command = normal position move
#20.22  =  3500       //  target position (x100)
#20.23  =  150        //  max speed      (x100)
#20.24  =  800        //  accel time     (x100)
#20.25  =  800        //  decel time     (x100)
#20.26  =  250        //  diameter = 2.5 feet  (x100)
#20.27  =  1373       //  gear_ratio = 13.73   (x100)

#20.20  =  1          // execute bit will clear immediately and hoist will start moving


drive will move to the new target position
```

Please note that if you request a new target position that exceeds either of the two limits, the hoist will move to and then fast stop at the soft limit. Further attempts to command a new target position outside the limit will produce no motion. However, if the new target position is within the limits, the hoist will move off the limit and go to the new position.

That completes the design and coding of the hoist application. As the reader can see, it's a simple application consisting of an initial task and a background task.

To keep the application simple, there was no consideration given to Emergency Stopping or Braking Systems. In a more complex application, these issues must be addressed.

The entire **SYPT** project is available on the CTSUPPORT web site for the user to download and experiment with.