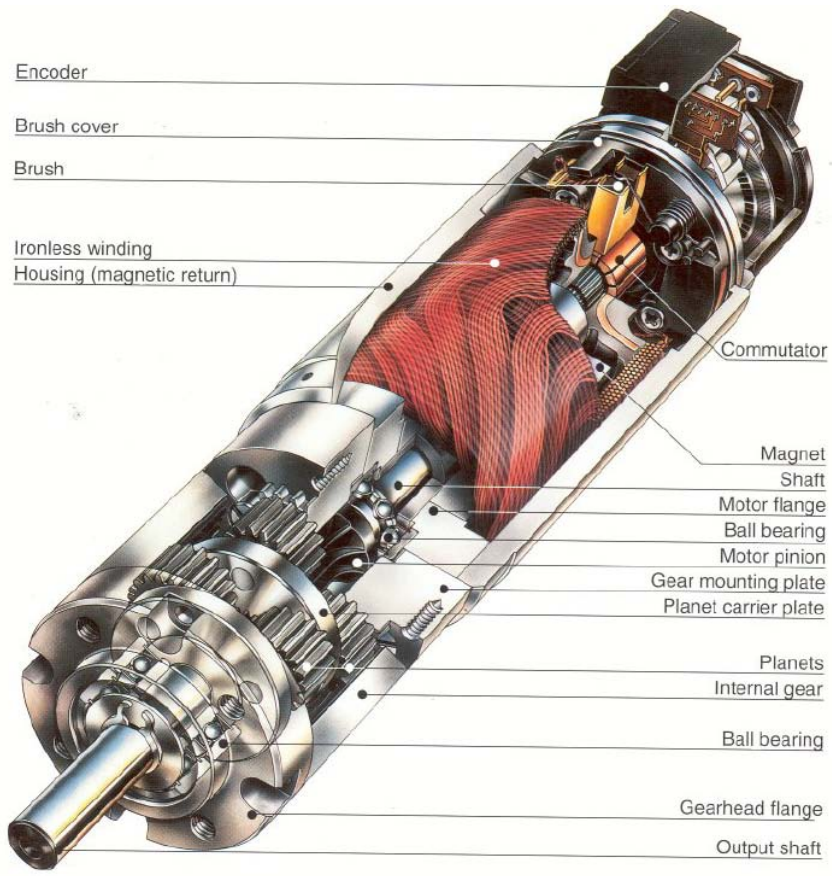
ENGR3390: ACTUATION AND CONTROL



TEAM BRAVO

J. GORASIA – 11/1/09

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

We developed a control system for four different motors in this lab, a digital DC motor, a stepper motor, an analog DC motor and a RC servo motor. Since we used a single board RIO as our controller, we had access to two real time operating systems, on the PowerPC and a FPGA. This gave us the ability to have accurate high speed control of the motors, and make precise closed loop control schemes. While the motors needed to be interfaced to in different ways, they can be abstracted out to be identical. Doing so makes it trivial to control all motors, and make them do things like sway in unison, or play music.

SUMMARY OF READINGS

Actuation and control is very important to mobile robotics. If robots are going to have any actuators, these need to be controlled. In this lab, we are granted specialized hardware to better accomplish this, the Single Board RIO.

The single board RIO consists of a PowerPC chip, a FPGA and many interchangeable modules. The advantage of using the Single Board RIO is that it is a real time operating system. That means that we can better predict and control the execution of operations on the computer. This is important in a physical system, as we want to control the physical systems precisely in time.

While both the PowerPC and the FPGA operate in real time, the FPGA operates much faster than the PowerPC at certain operations. The FPGA is software reconfigurable hardware which means we can shape hardware with software into highly optimized structures for certain operations. While this means that the FPGA is limited in the operations it can perform (it can't do floating point math) it is very good at what it does. Therefore, by using the PowerPC and FPGA together, it is possible to design a very computationally efficient method for controlling motors.

SOFTWARE SYSTEMS OVERVIEW

The code can be broadly divided in two: code running on the FPGA, and code running on the PowerPC. Vis running on the FPGA will be referred to as chips, while Vis running on the PowerPC will be referred to as sub-Vis.

MAIN CODE

The main code is a combination of all the code for the four different motors. It was developed by independent teams then pieced together to make the main code. Therefore, the main code can be easily grouped in bands.

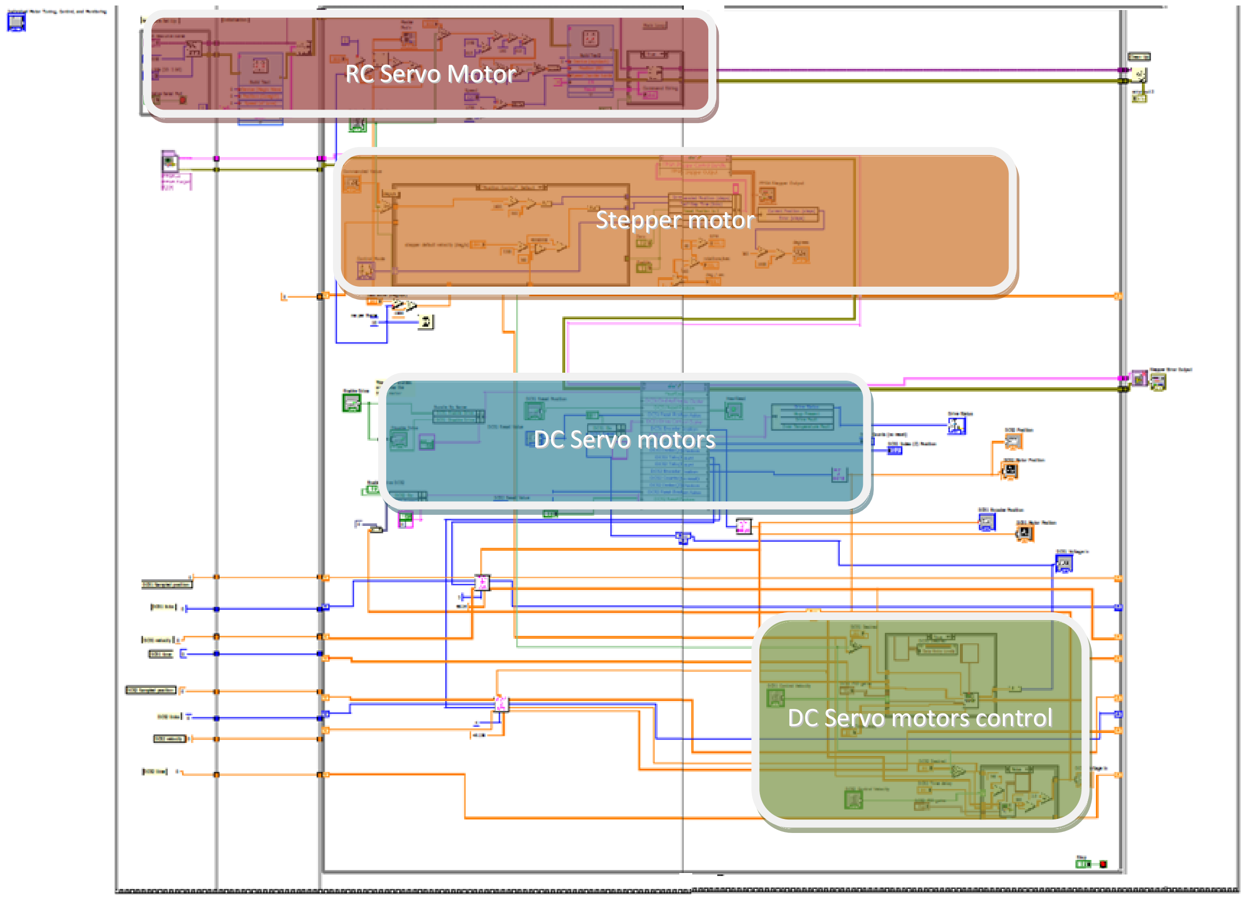
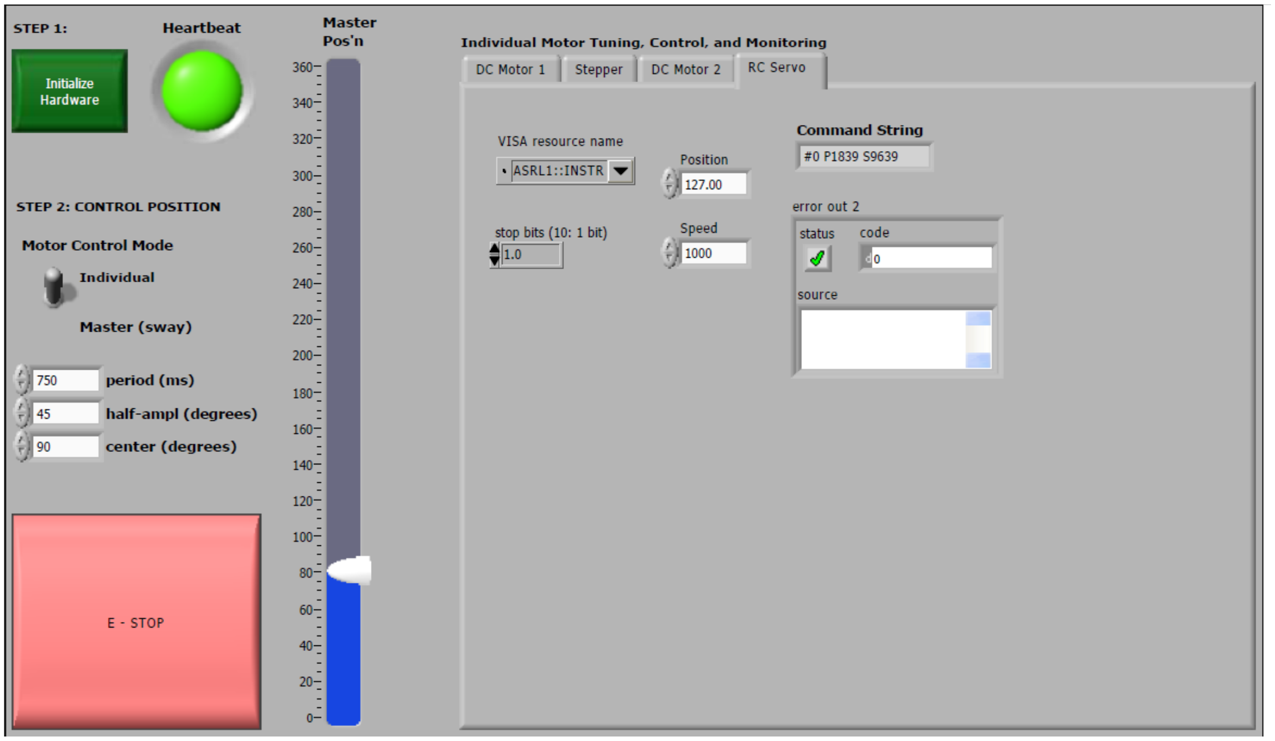


Figure 1: Main motor block diagram. This is a huge VI which is divided into bands for separate motors.

The main VI gives the possibility of controlling the all four motors together or individually. In addition, a sway command was implemented to make the motors oscillate with a configurable period, amplitude and offset.



VISA resource name: ASRL1::INSTR
Position: 127.00
Command String: #0 P1839 S9639
stop bits (10: 1 bit): 1.0
Speed: 1000
code: 0
period (ms): 750
half-ampl (degrees): 45
center (degrees): 90

How the individual motors are controlled is described in the next few sections.

RC SERVO MOTOR

The RC servo motor is controlled through a Lynxmotion SSC-32 servo controller. The controller takes in a serial command input which it interprets to control up to 32 servo motors simultaneously.

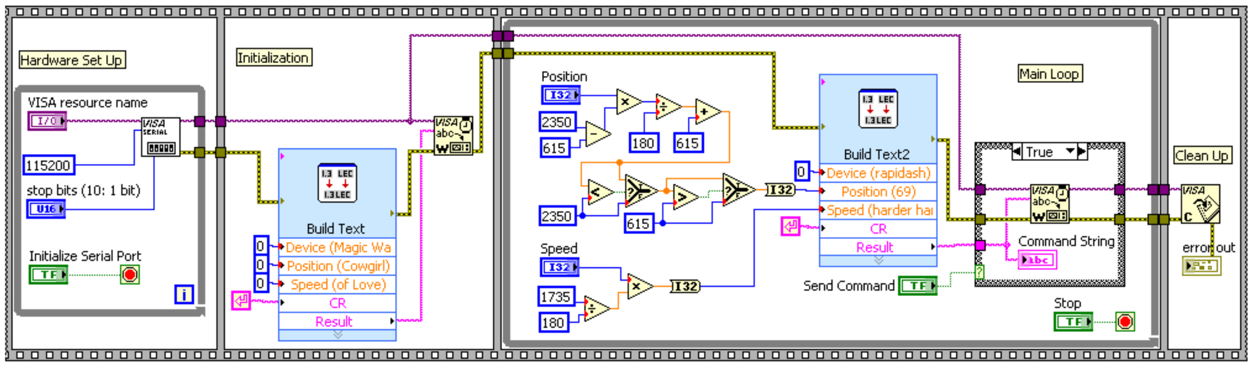


Figure 2: The block diagram for the RC servo motor. This packages the string to be sent via serial to the SSC 32 motor controller. Inputs from the front panel are scaled from degrees to the appropriate values for the motor controller to use.

The command string for the controller follows the following format:

# <ch> P <pw> S <spd> ... # <ch> P <pw> S <spd> T <time> <cr>

Example: "#0 P1600 S750 <cr>"

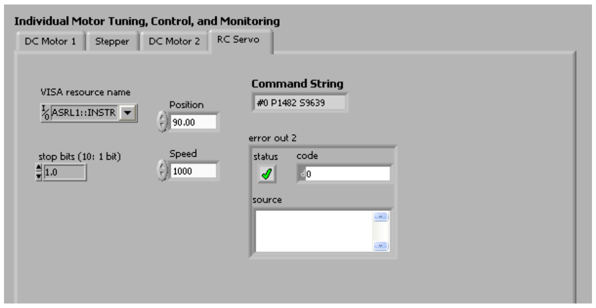
From the user manual:

- "<ch>   Channel number in decimal, 0-31

- <pw> Pulse width in microseconds, 500-2500
- <spd> Movement speed in uS per second for one channel (Optional)
- <time> Time in mS for the wntire move, affects all channels, 65535 max (Optional)
- <cr> Carriage return character, ASCII 13 (Required to initiate action)
- <esc> Cancel the current action, ASCII 27"

For us, the channel number used was 0 since there was only one servo motor. The pulse **<pw>** was a value from 615 to 2350. By scaling the pulse width to the corresponding position of the motor in degrees, we found that 0 degrees corresponds to a pulse width of 615 and 180 degrees to a pulse width of 2350. The rest of the positions varied linearly within this range. The speed <spd> was determined in the same fashion, giving a speed of 180 degrees per second with a value of 1735.

A note from the user manual[1] was that the first positioning command should be "# <ch> P <pw>". This allows the controller to learn where the servo is positioned on power up. If this step is ignored, speed and time commands will be ignored by the controller. Thus we created a button to send this default string.



VISA resource name: ASRL1::INSTR

stop bits (10: 1 bit): 1.0

Position: 90.00

Speed: 1000

Command String: #0 P1482 S9639

error out 2

status: ✓

code: 0

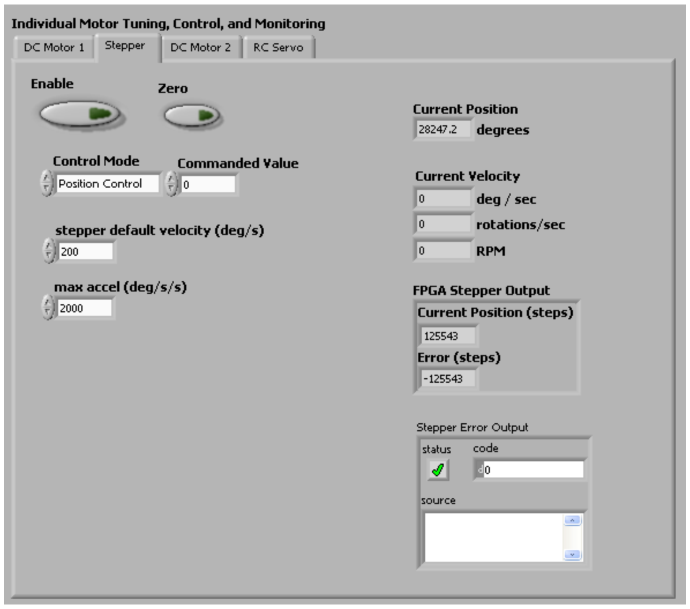
source:

Figure 3: Front panel for the RC servo motor. The initialize serial port command initializes the serial port, while the position and speed are controlled in degrees.

STEPPER MOTOR

The stepper is unique in this set of motors as it does not have position feedback. Therefore, it must keep track of its position by keeping count of the number of steps it has moved and make sure it does not skip steps.

[1] http://www.lynxmotion.com/images/html/build136.htm#comform



Control Mode: Position Control
Commanded Value: 0
stepper default velocity (deg/s): 200
max accel (deg/s/s): 2000
Current Position: 28247.2 degrees
Current Velocity: 0 deg / sec; 0 rotations/sec; 0 RPM
FPGA Stepper Output
Current Position (steps): 125543
Error (steps): -125543
Stepper Error Output
status: ✓
code: 0
source

Figure 4: Front panel to control the Stepper motor

The FPGA code is shown in Figure 5. The FPGA compare the stepper's current position with the intended position and then steps in the appropriate direction that would close that difference. Since there is no encoder, the FPGA keeps track of the number of steps that it has commanded in a tally which is later used to determine the position of the motor. In addition, the motor also gets the time between steps, allowing it to do velocity control.

Complication in the code arises because a few gotchas that needs to be accounted for. Firstly, the FPGA should only update its tally of number of steps when it actually sends a step command. Next, there is a feature to zero the stepper tally, to allow the position of the motor to be set. Finally, direction control is achieved by simple position control, comparing the desired position to the current position. There is also a control mode input, which gives either velocity control or position control. Velocity control is achieved by comparing the position to a known position and determining how past the stepper should have moved in the timestep.

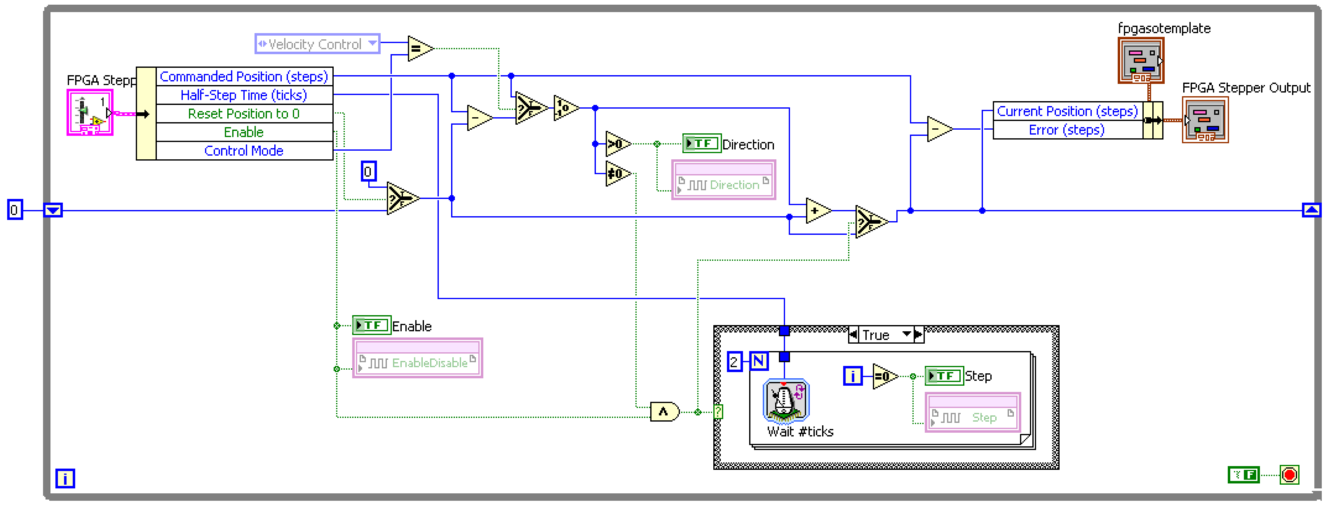
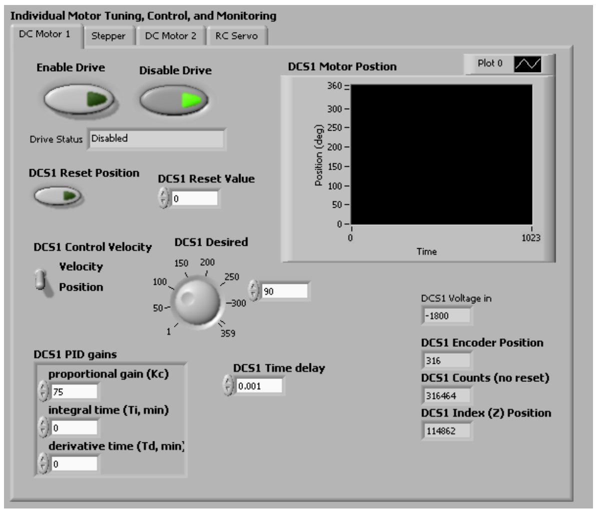


Figure 5: Chip for controlling the stepper motor.

DC SERVO MOTOR 1 (DIGITAL CONTROL)

DC Servo motor 1 (DCS1) is controlled through a NI 9505 motion control module. The module gives full H-bridge control of the DC motor. It also takes in the encoder inputs, allowing us to implement closed loop control of the motor.

To control the motor, we pipe a PWM signal to the motion control module, and this is translated to an analog voltage to command the motor.



Drive Status Disabled

DCS1 Reset Value 0

DCS1 Desired 90

proportional gain (Kc) 75

integral time (Ti, min) 0

derivative time (Td, min) 0

DCS1 Time delay 0.001

DCS1 Voltage in -1800

DCS1 Encoder Position 316

DCS1 Counts (no reset) 316464

DCS1 Index (Z) Position 114862

Figure 6: Front Panel for DCS1

The control on the FPGA for DCS1 consists of 4 chips:

- Signal Enable / Disable
- PWM pulse generator
- Encoder reading

Since DCS1 is controlled by a motion control module, it needs to be enabled before it can be used, as shown in Figure 7.

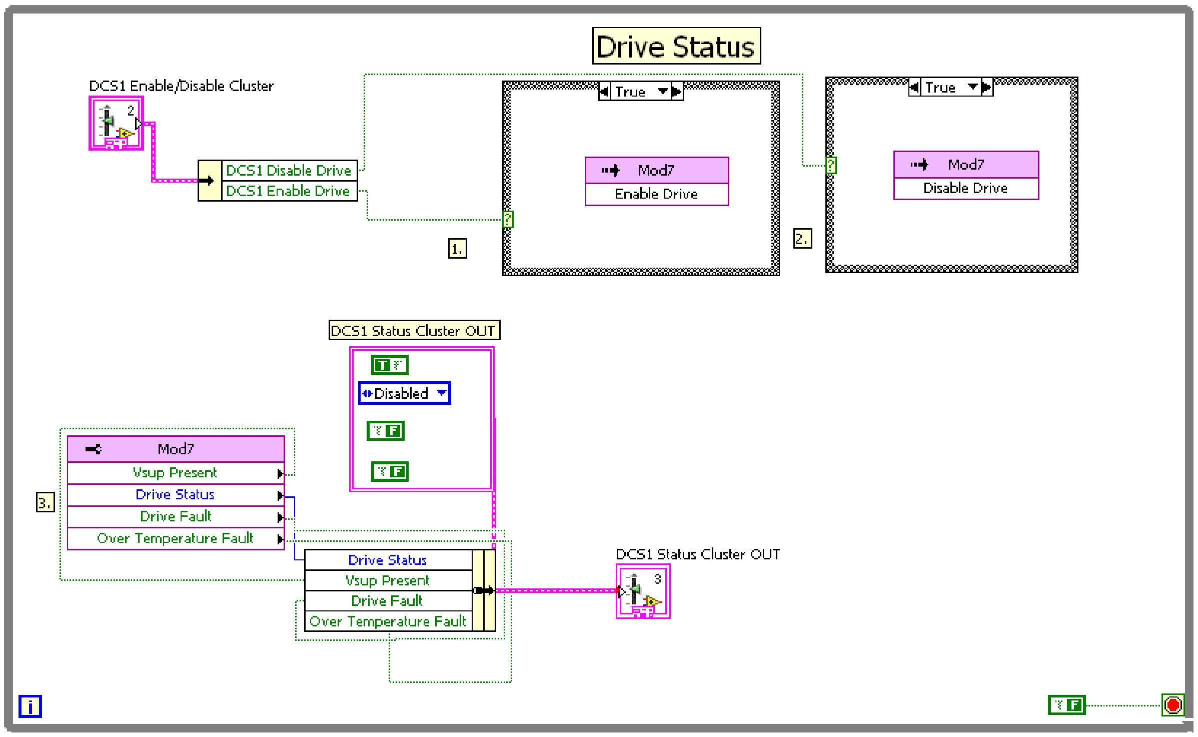


Figure 7: DCS1 Enable and disable chip. DCS1 is controlled by a NI 9505 motion control module, which needs to be enabled before being used. Using the motion control module gives a few indicators to troubleshoot motor performance like drive faults, or over temperature faults.

Next, a PWM signal needs to be generated to control the speed of the motor. Looking at the datasheet of the motion control module, it can take a 20kHz PWM signal, and modulate the speed of the motor based on the duty of the signal. To make the control easier, the PWM signal input, DCS1 speed, is allowed to range from -2000 to 2000. This allows us to also insert the direction for the motor to travel to into the DCS1 input, as that is a separate control to the motion control module. The PWM signal works by incrementing two counters. One keeps track of the signal length while another keeps track of duty length. By incrementing the two counters appropriately, it is easy to create a PWM signal.

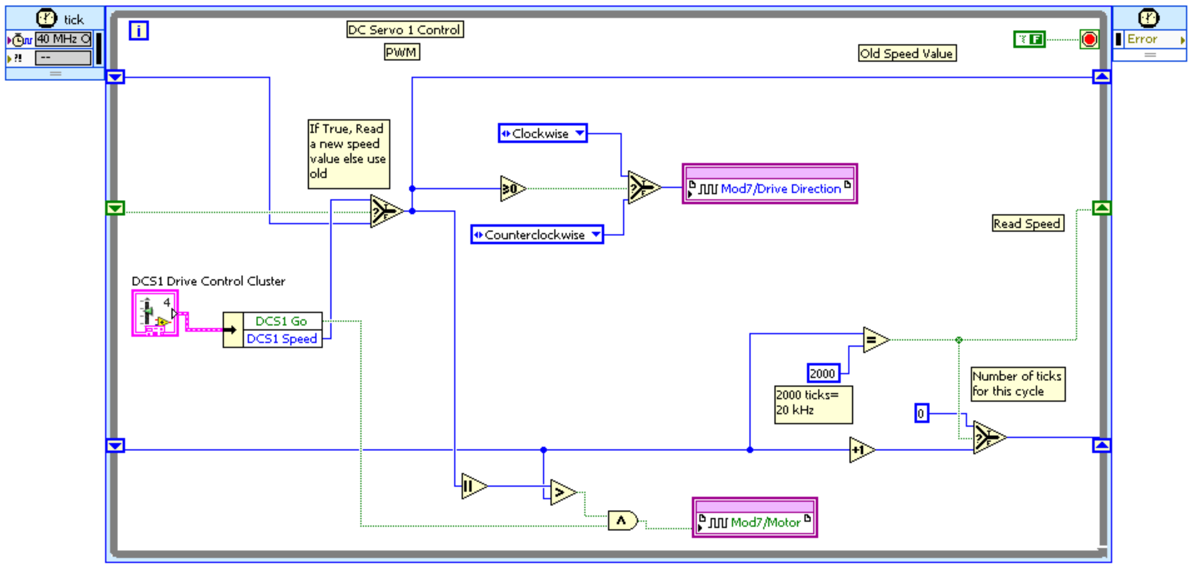


Figure 8: DCS1 Control chip. This chip generates the PWM signal that drives DCS1, and controls the motor direction.

Finally, the encoder position is determined using clever use of XOR logic gates and shift registers. The encoders consist of 3 separate rings: two of which are for relative positions, while the third is for absolute positioning. We can think of the encoders producing square wave signals. The number of counts of the square wave can be used to determine the relative position of the motor. On the other hand, the phase offset of Encoder A and Encoder B can be used to determine if the motor is spinning clockwise or counter clockwise.

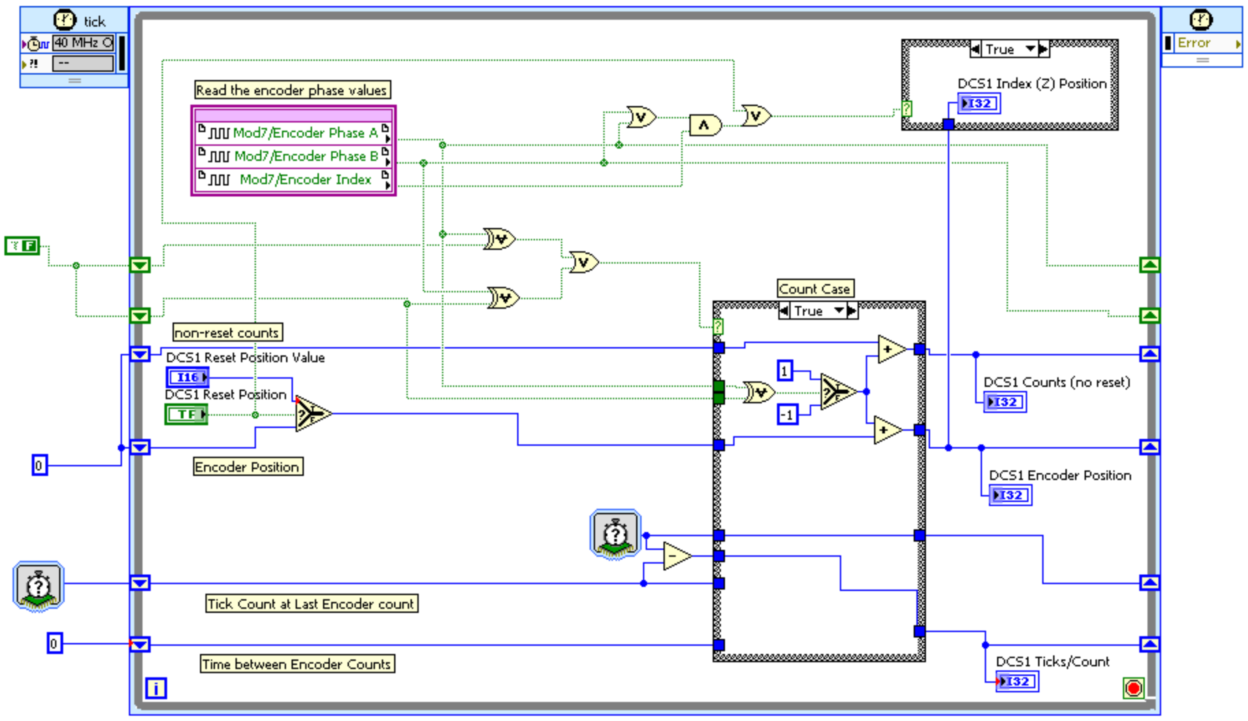


Figure 9: The encoder reading chip uses XOR gates and shift registers to convert the quadrature encoder signals into a motor direction and rotation count. The chip also keeps absolute position of the encoders, taking that responsibility off the PowerPC. In addition, the chip gives a time between encoder counts, which is useful to determine the velocity of the motor.

The diagram in Figure 10 helps explain how the chip in Figure 9 works.

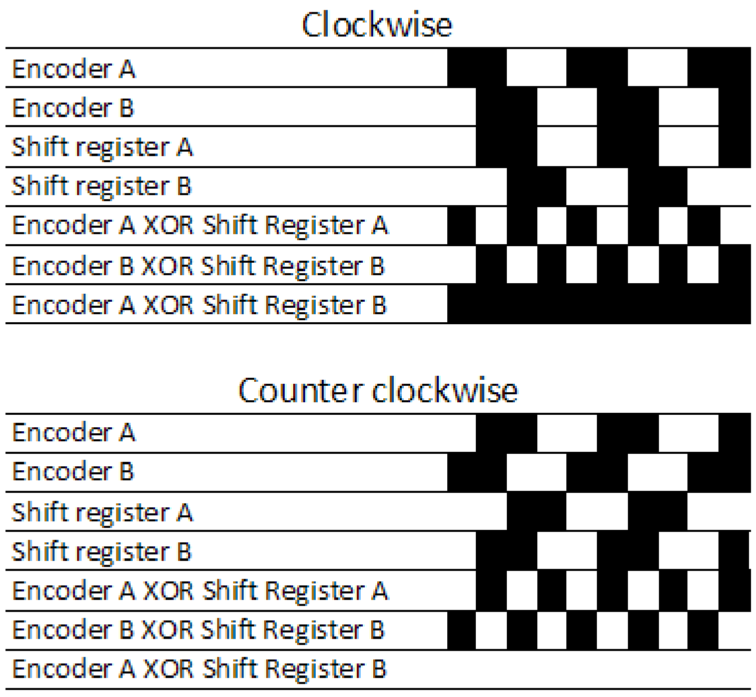
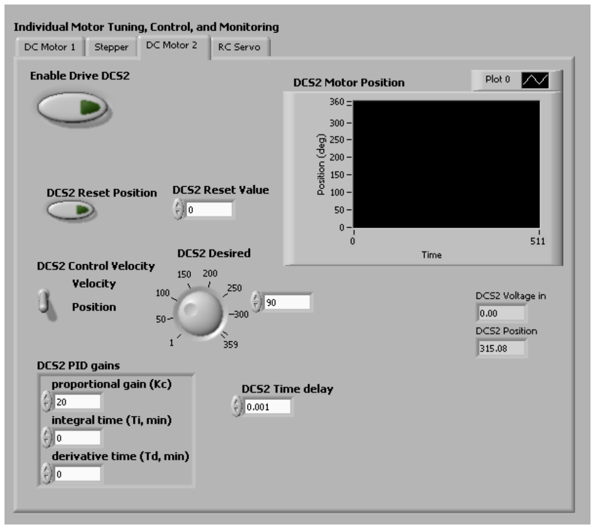

Clockwise
Encoder A
Encoder B
Shift register A
Shift register B
Encoder A XOR Shift Register A
Encoder B XOR Shift Register B
Encoder A XOR Shift Register B
Counter clockwise
Encoder A
Encoder B
Shift register A
Shift register B
Encoder A XOR Shift Register A
Encoder B XOR Shift Register B
Encoder A XOR Shift Register B

Figure 10: The shift registers are used to create a delay to the encoder signals. If the XOR of Encoder A or B and its shift register is true, the count increments. Finally, if the XOR of Encoder A and the shift register B is true, we know the motor is rotating clockwise and vice versa.

DC SERVO MOTOR 2 (ANALOG CONTROL)

DC Servo Motor 2 (DCS2) is very similar to DC Servo Motor 1. The main difference is that DC servo 2 is controlled by an analog voltage which is passed through a Sabertooth motor controller to control the motor. The encoder position is fed back to the PowerPC via digital IOs.



DCS2 Reset Value: 0
DCS2 Desired: 90
DCS2 Voltage in: 0.00
DCS2 Position: 315.08
proportional gain (Kc): 20
integral time (Ti, min): 0
derivative time (Td, min): 0
DCS2 Time delay: 0.001

Figure 11: Front panel for DC Servo 2.

Since the wiring of DCS2 is simple, only a single chip is required to enable the motor and to control its speed. The motor will not move if sent a signal of 2.5V, while a signal above that will make it rotate clockwise, and a signal below that will make it rotate counter clockwise. This chip is shown in Figure 12.

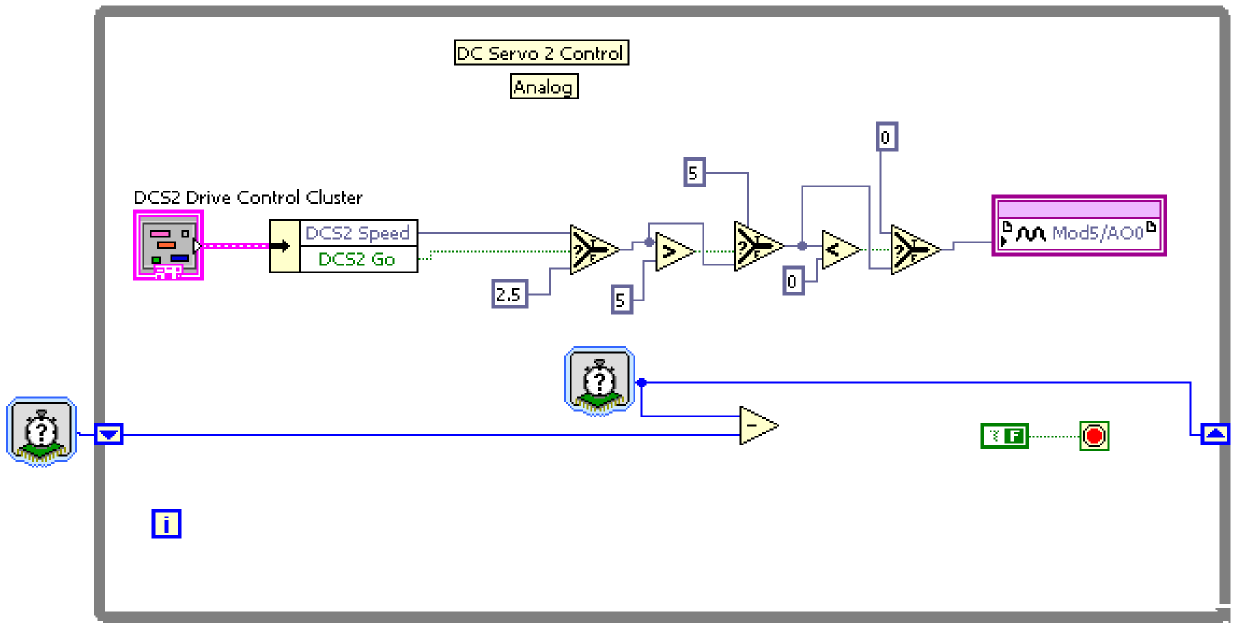


Figure 12: Enable and speed control chip for DCS2. It is a simple chip that keeps the output voltage between 0 and 5 volts.

The encoder used for DCS2 was identical to DCS1, therefore the chip in Figure 9 was reused.

SUPPORTING VIS FOR DC SERVOMOTORS

On the PowerPC, we receive position in ticks from the DC motors, and we send either PWM duty signals or an analog voltage value to control the velocity of the motors.

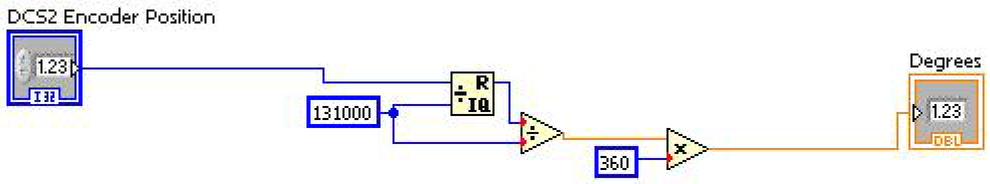


Figure 13: Ticks to degrees sub-VI. This takes the number of ticks moved coming from the encoder and scales it to degrees.

To convert the tick values coming from the encoders to rotation values in degrees of the motors, we employ a modulus operator and linear scaling. Since the encoder is a quadrature encoder with 500 counts per revolution, and is connected to a 65.5:1 gear reduction, there will be 131000 counts per revolution of the output shaft. Taking the remainder of the cumulative number of ticks divided by 131000 will constrain the number of ticks between 0 and 131000, a position in ticks. Finally, scaling this number will give the position of the output shaft in degrees.

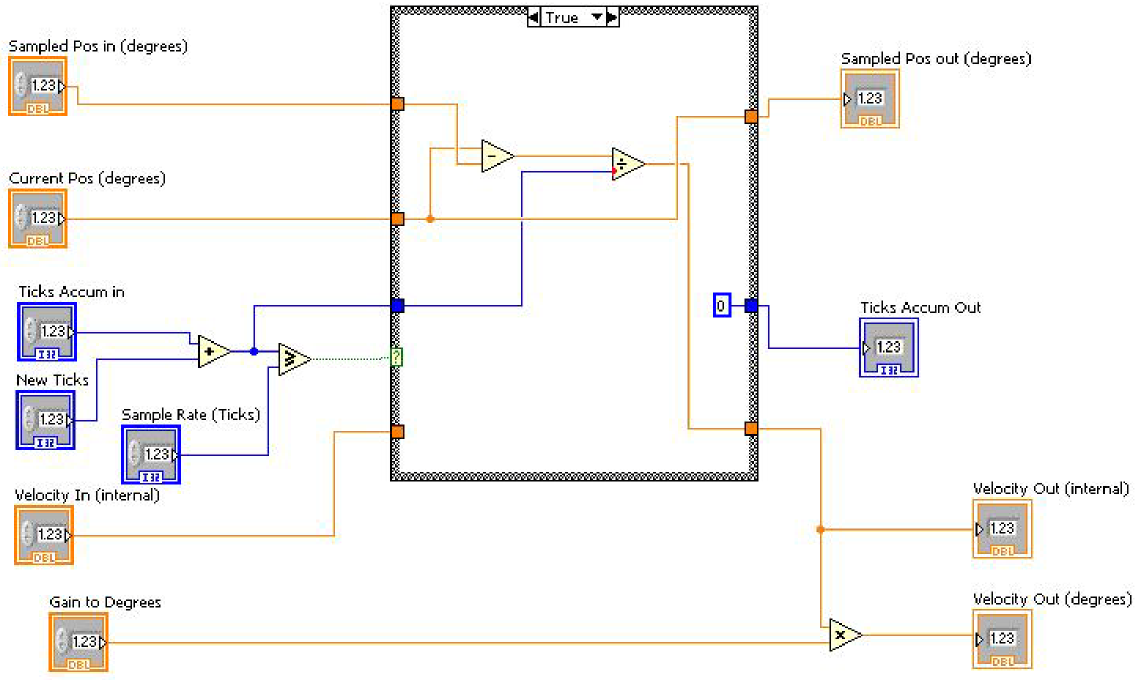


Figure 14: Position to velocity converter based on the change in position of the motor and the ticks per count value.

Next, we have a sub-VI which uses the change in position of the motor and the sample tick rate to give the velocity of the motor. Interestingly, it does not use an accumulator for the tick rate to give time. Instead, it simply reads the ticks per count value from the encoder, then resets the shift register that could be used to keep track of it. The false case for this VI simply passes values through.

TESTING

To create a control system, we needed to determine if the motor's responded linearly to their input signals. Therefore, we tested DCS1 by sending it PWM signals of different duties, and determining the velocities of the motor. Using a stopwatch and eyeballing the rotation of the motors for several revolutions, we managed to get the graph in Figure 15. From the graph we can tell that the motor responds linearly to changes in duty, which makes controlling it easier.

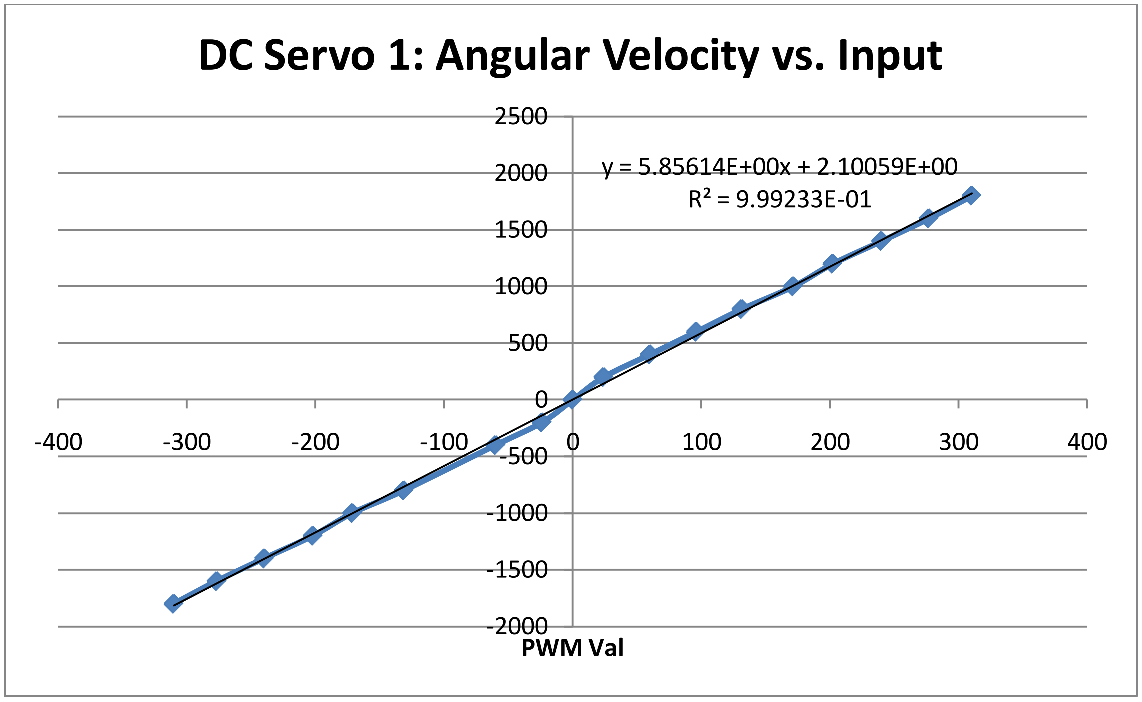


Figure 15: Velocity of DCS1 as a function of the PWM duty. This chart compares the measured angular velocity in degrees per second to the inputted PWM duty cycle expressed in our internal control units consisting of vales between -1800 and 1800. The response is very linear, with a correlation coefficient of over 99%. However, there is a small dead zone around 0. Though we did not try to quantify the extent, it is slightly visible in this graph.

The other motors were tested in similar ways, by eyeballing the speeds and making sure they matched up. For DCS1 and the stepper motor, we used the physical dials to make sure that the motor was moving to the positions we thought it was supposed to.

Through testing we learned that we could make the stepper motor run at about 700 RPM. This made it possible for us to produce audible notes, which we then utilized to produce music.

CONCLUSIONS

Using the FPGA allowed us to have real time control over the motors which is important when dealing with a physical system like motors. We were lucky that all motors had very linear responses, making it trivial to develop a control schematic for them.

Based on our experiences in this lab, the steps to control a motor can be summarized as:

- Interface with the motor
- Make the motor move
- Determine the motor position
- Test the response of the motor to different inputs.
- Develop a control scheme.

This plan worked well for this lab, and will be the method I use for future robotics projects.

REVISED LAB WRITEUP

The lab handout is confusing, which is a function of having strange Vis in it that seems disparate. More work needs to be done to make the VIs flow better.

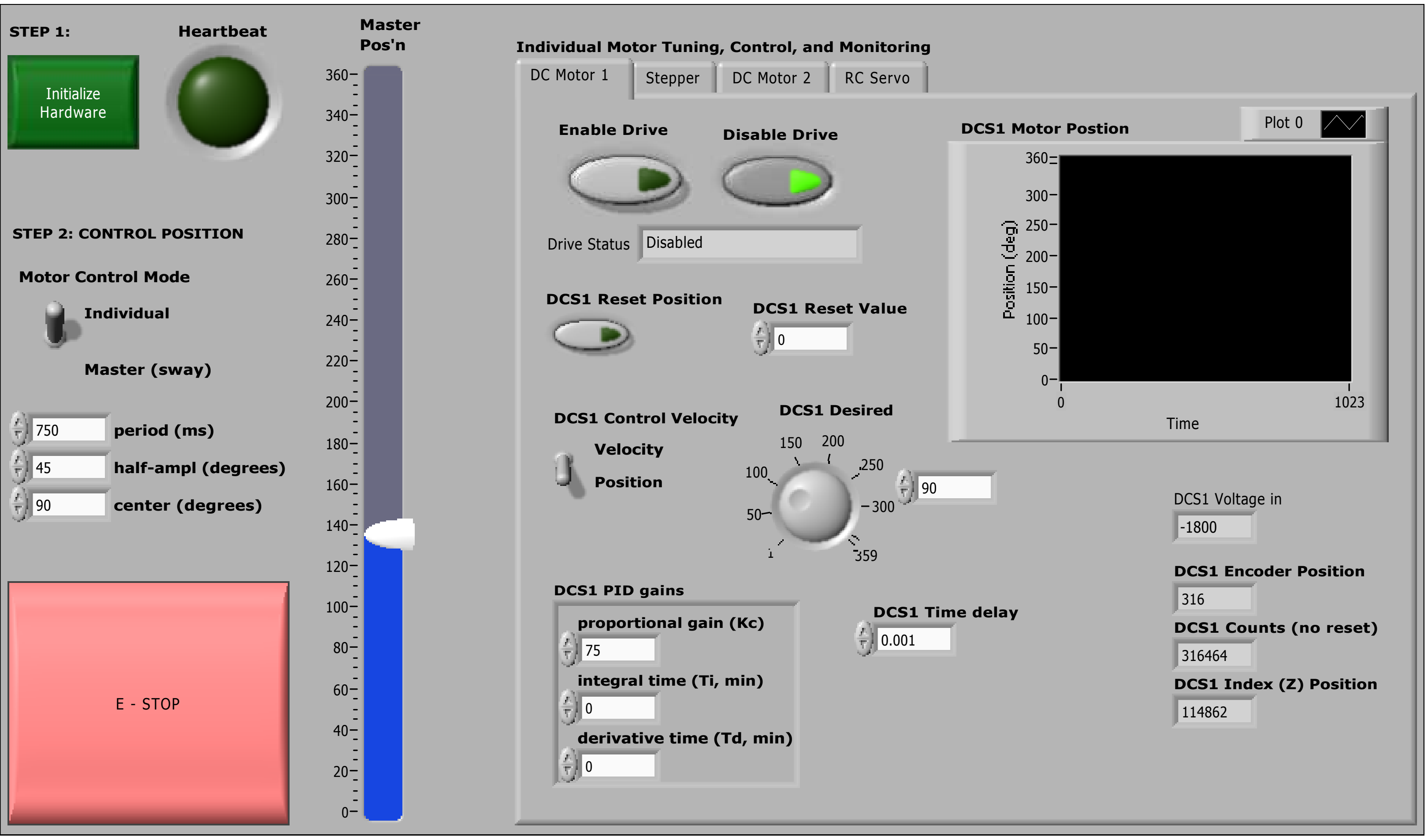
In addition, the wiring schematic for DCS2 was completely wrong. The correct wiring is:

- Encoder Phase A: Module 1-DIO4
- Encoder Phase B: Module 1-DIO5
- Encoder Index: Module 1-DIO6
- AO0: Module 5-AO0

APPENDIX

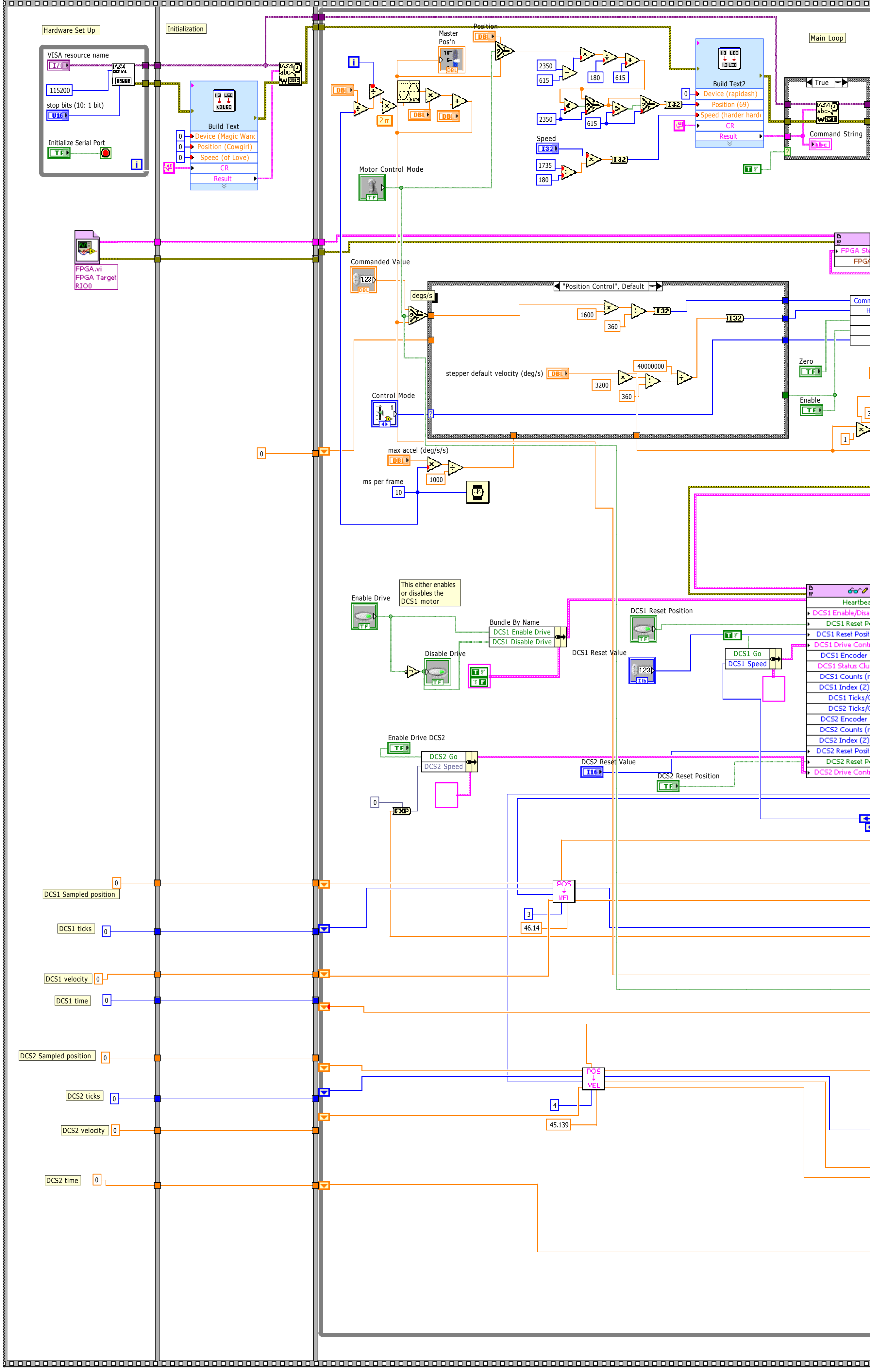
Code is attached at the end of this document.

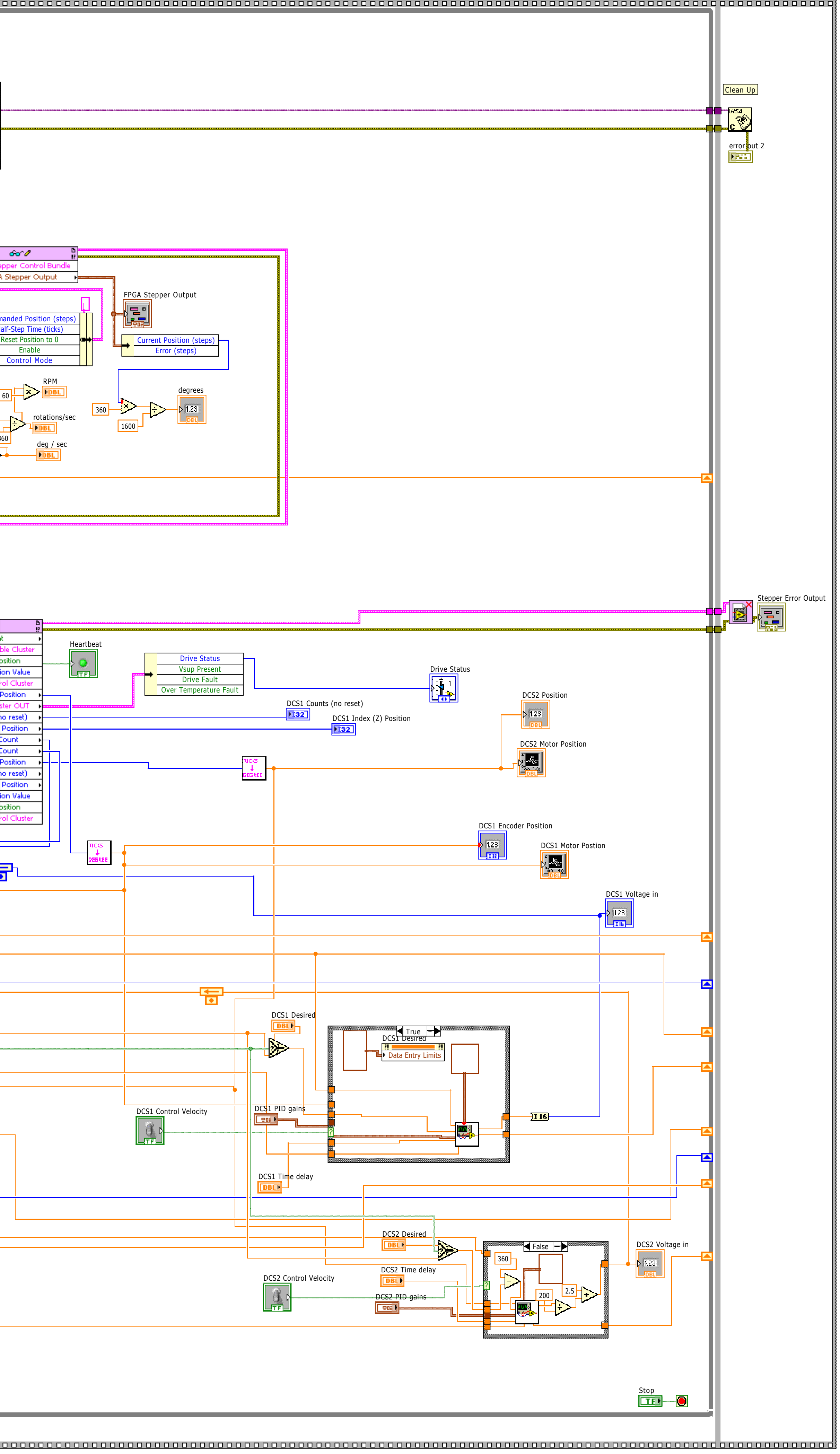
PowerPC.vi

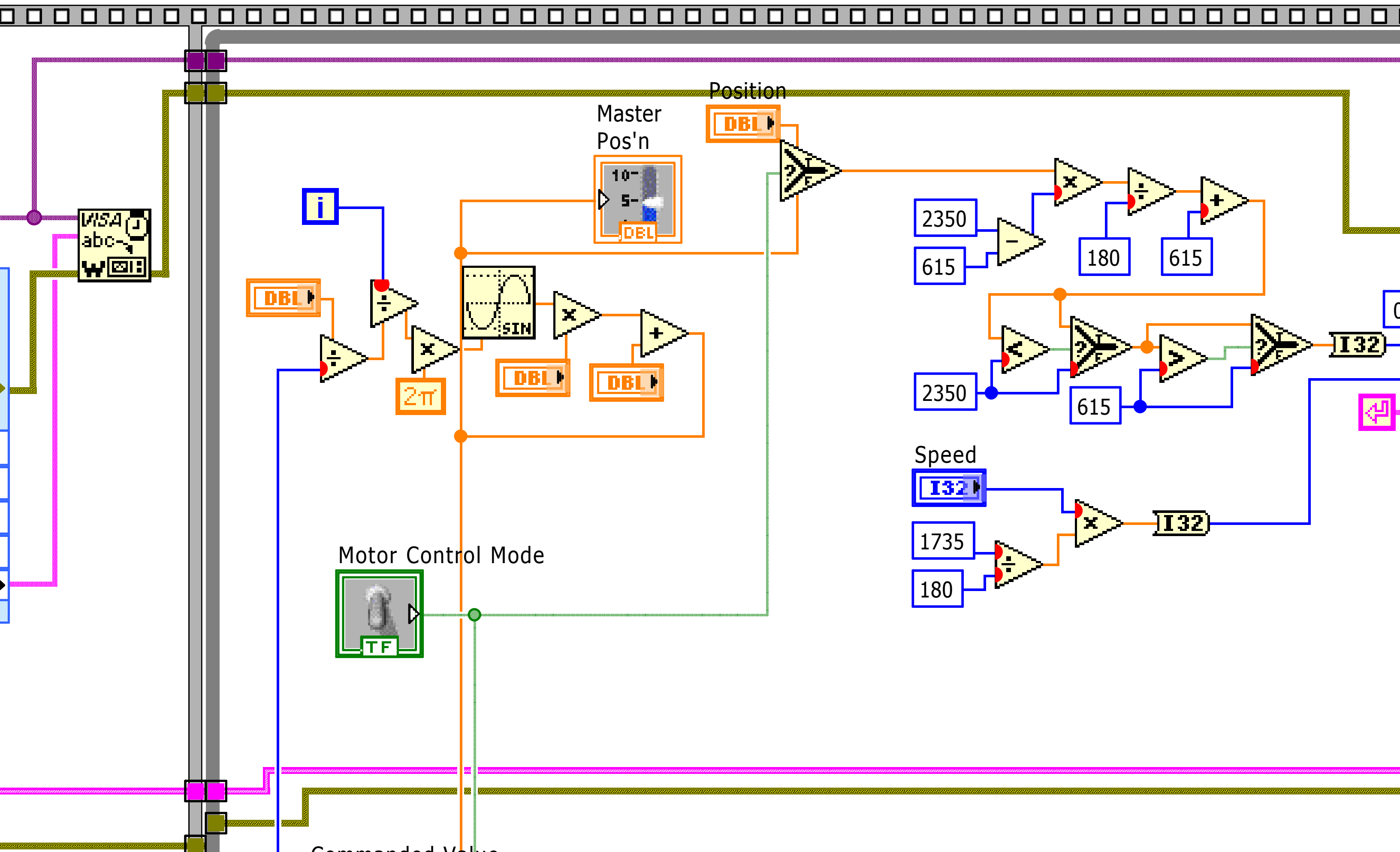


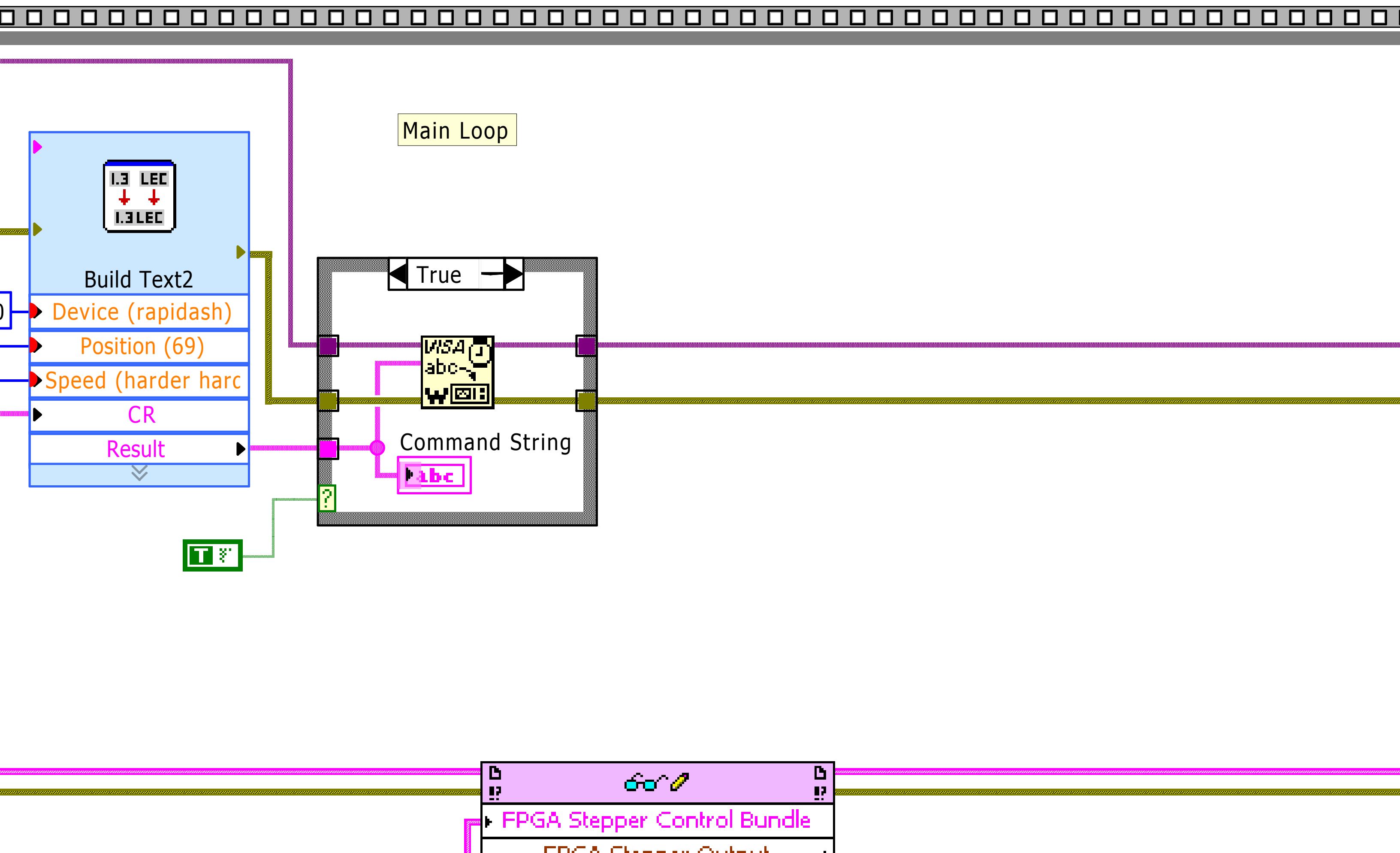
period (ms): 750
half-ampl (degrees): 45
center (degrees): 90
Drive Status: Disabled
DCS1 Reset Value: 0
DCS1 Desired: 90
proportional gain (Kc): 75
integral time (Ti, min): 0
derivative time (Td, min): 0
DCS1 Time delay: 0.001
DCS1 Voltage in: -1800
DCS1 Encoder Position: 316
DCS1 Counts (no reset): 316464
DCS1 Index (Z) Position: 114862

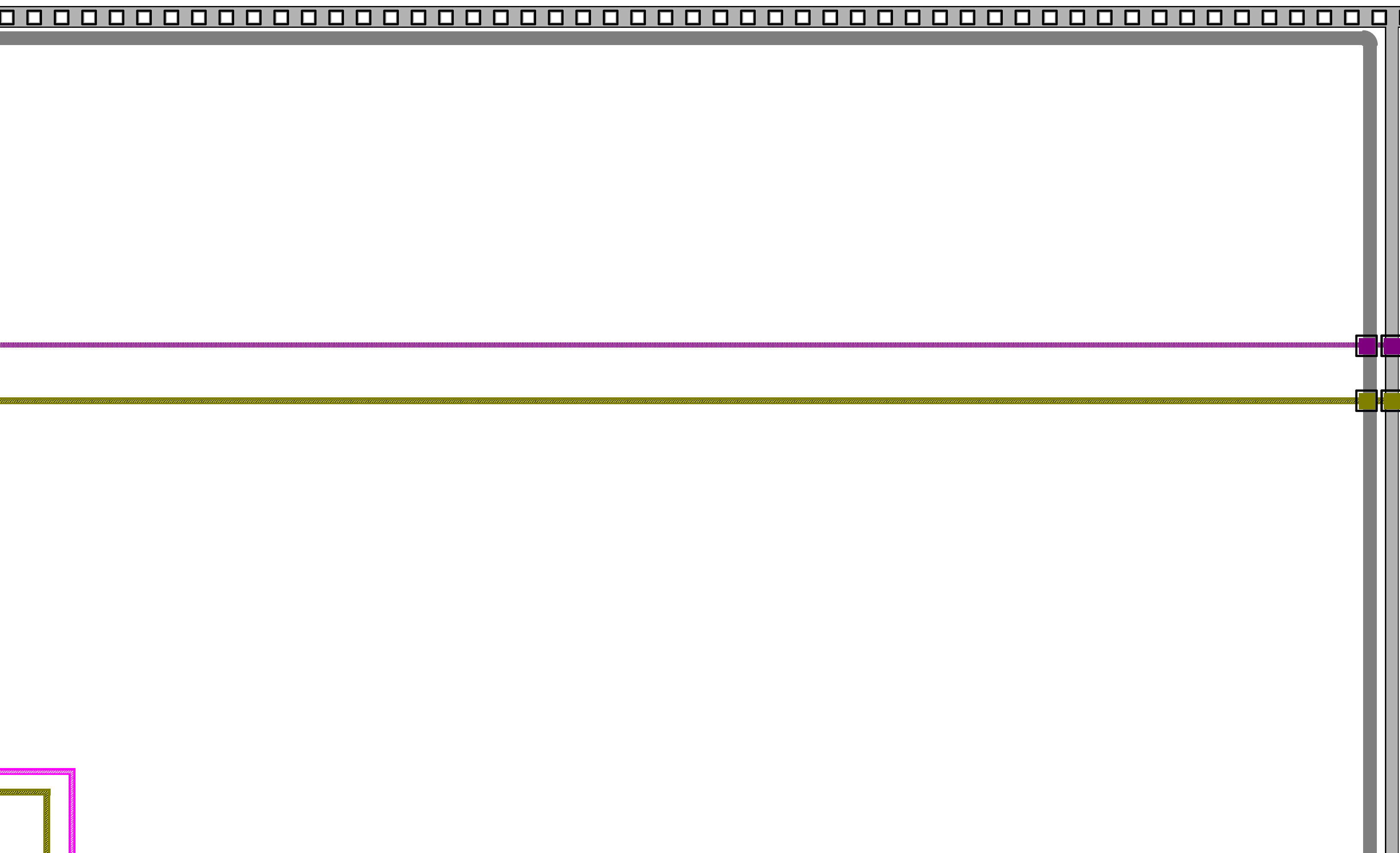
Individual Motor Tuning, Control, and Monitoring

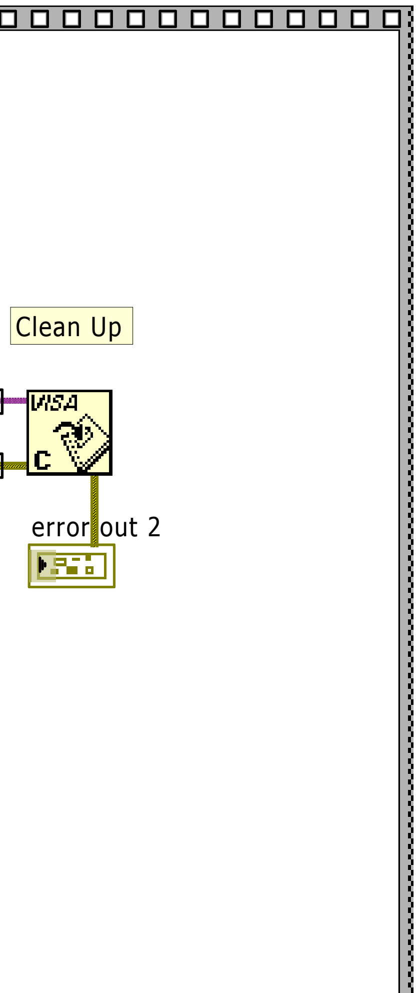


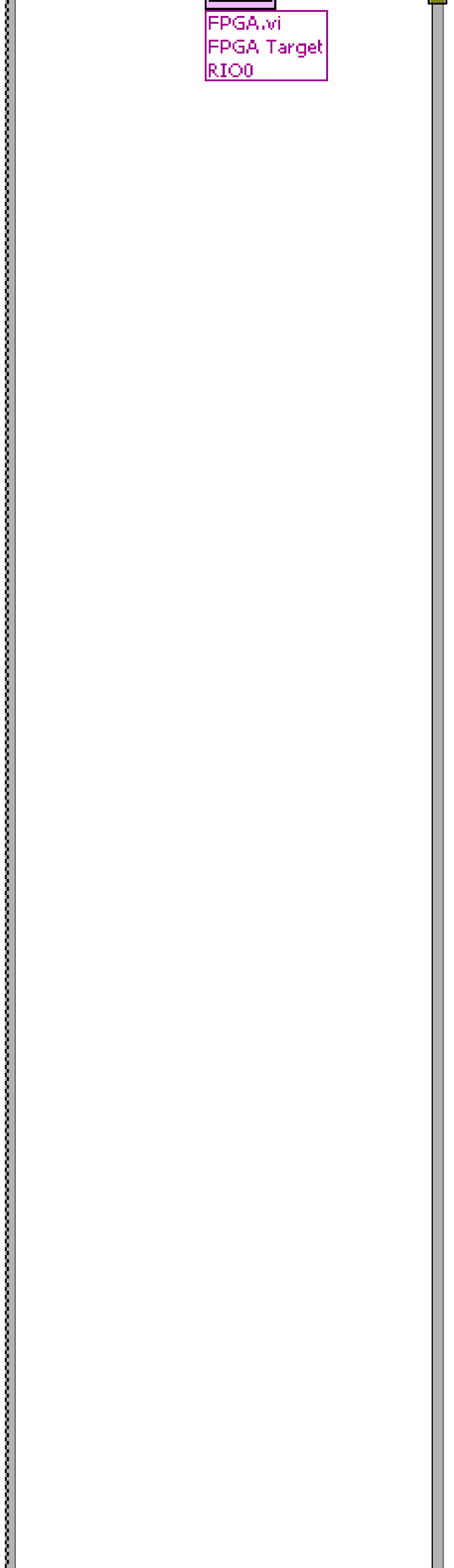

Clean Up
error out 2
epper Control Bundle
A Stepper Output
FPGA Stepper Output
anded Position (steps)
alf-Step Time (ticks)
Reset Position to 0
Enable
Control Mode
Current Position (steps)
Error (steps)
60
RPM
rotations/sec
360
deg / sec
360
1600
degrees
Stepper Error Output
ble Cluster
osition
ion Value
rol Cluster
Position
ster OUT
no reset)
Position
Count
Count
Position
no reset)
Position
ion Value
osition
rol Cluster
Heartbeat
Drive Status
Vsup Present
Drive Fault
Over Temperature Fault
Drive Status
DCS1 Counts (no reset)
DCS1 Index (Z) Position
DCS2 Position
DCS2 Motor Position
DCS1 Encoder Position
DCS1 Motor Postion
DCS1 Voltage in
DCS1 Desired
True
DCS1 Desired
Data Entry Limits
I16
DCS1 PID gains
DCS1 Control Velocity
DCS1 Time delay
DCS2 Desired
False
360
200
2.5
DCS2 Time delay
DCS2 Control Velocity
DCS2 PID gains
DCS2 Voltage in
Stop

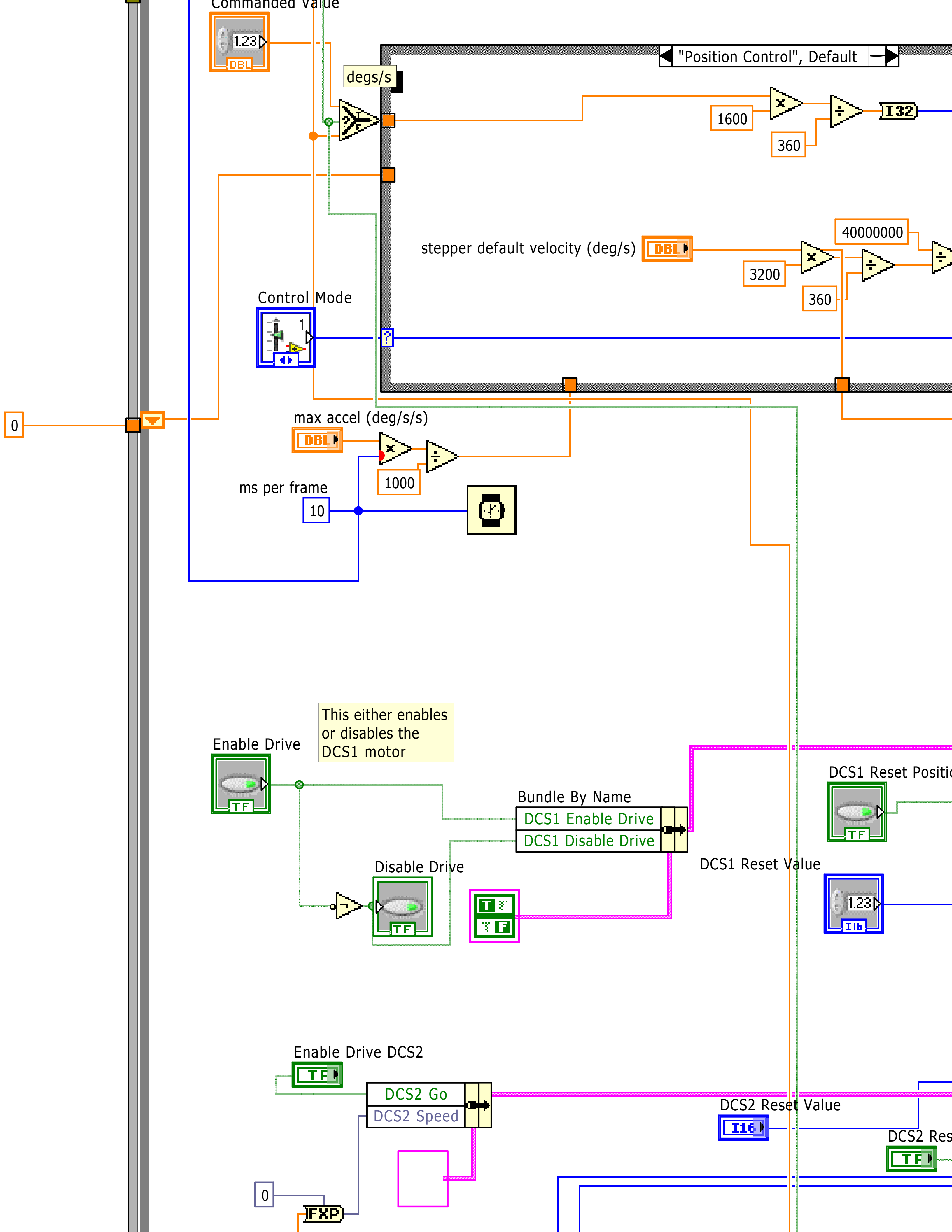

Position
Master
Pos'n
DBL
10
5
i
DBL
2π
SIN
DBL
DBL
2350
615
180
615
2350
615
I32
Speed
I32
1735
180
I32
Motor Control Mode
TF
Commanded Value
VISA
abc


Main Loop
Build Text2
Device (rapidash)
Position (69)
Speed (harder harc
CR
Result
True
VISA
abc
Command String
abc
FPGA Stepper Control Bundle
FPGA Stepper Output

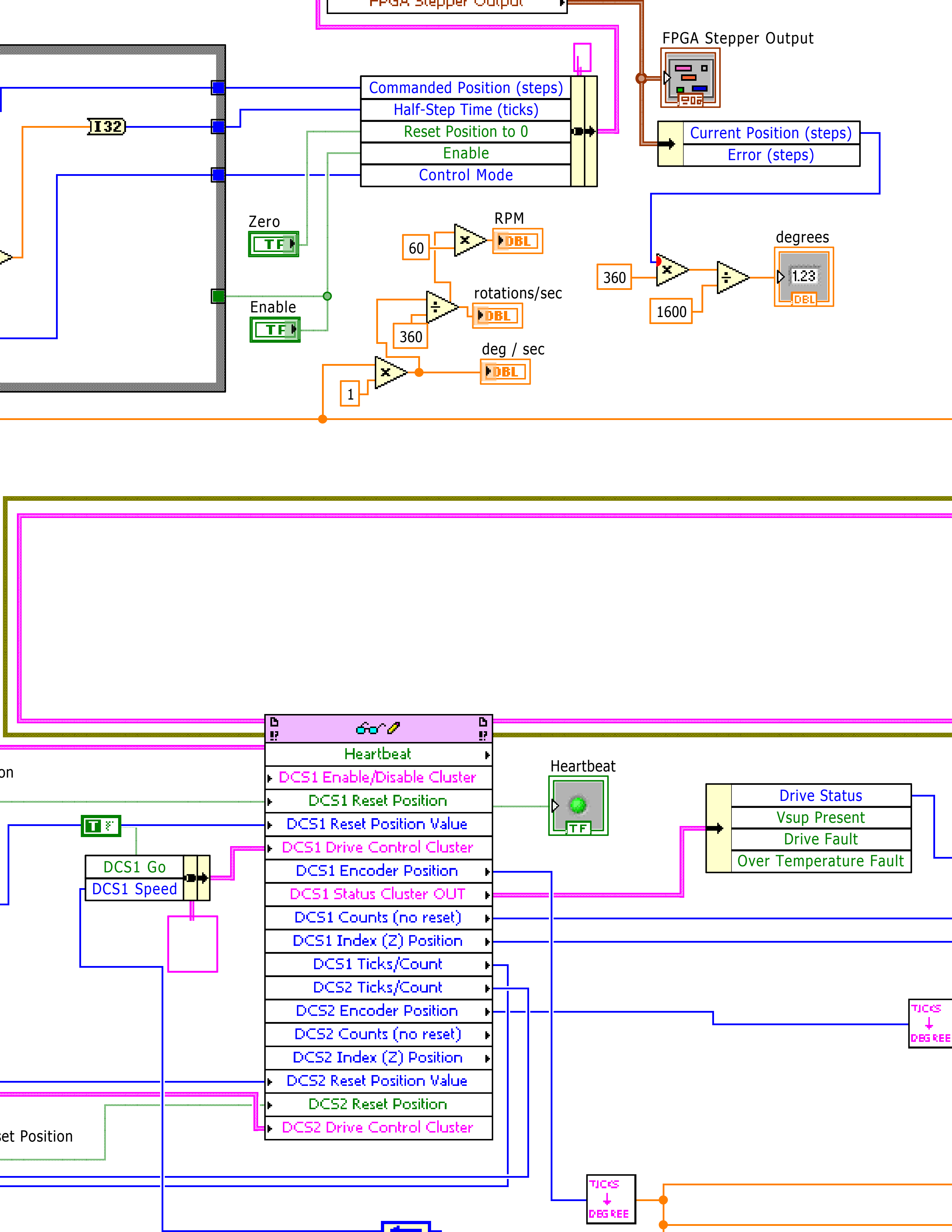


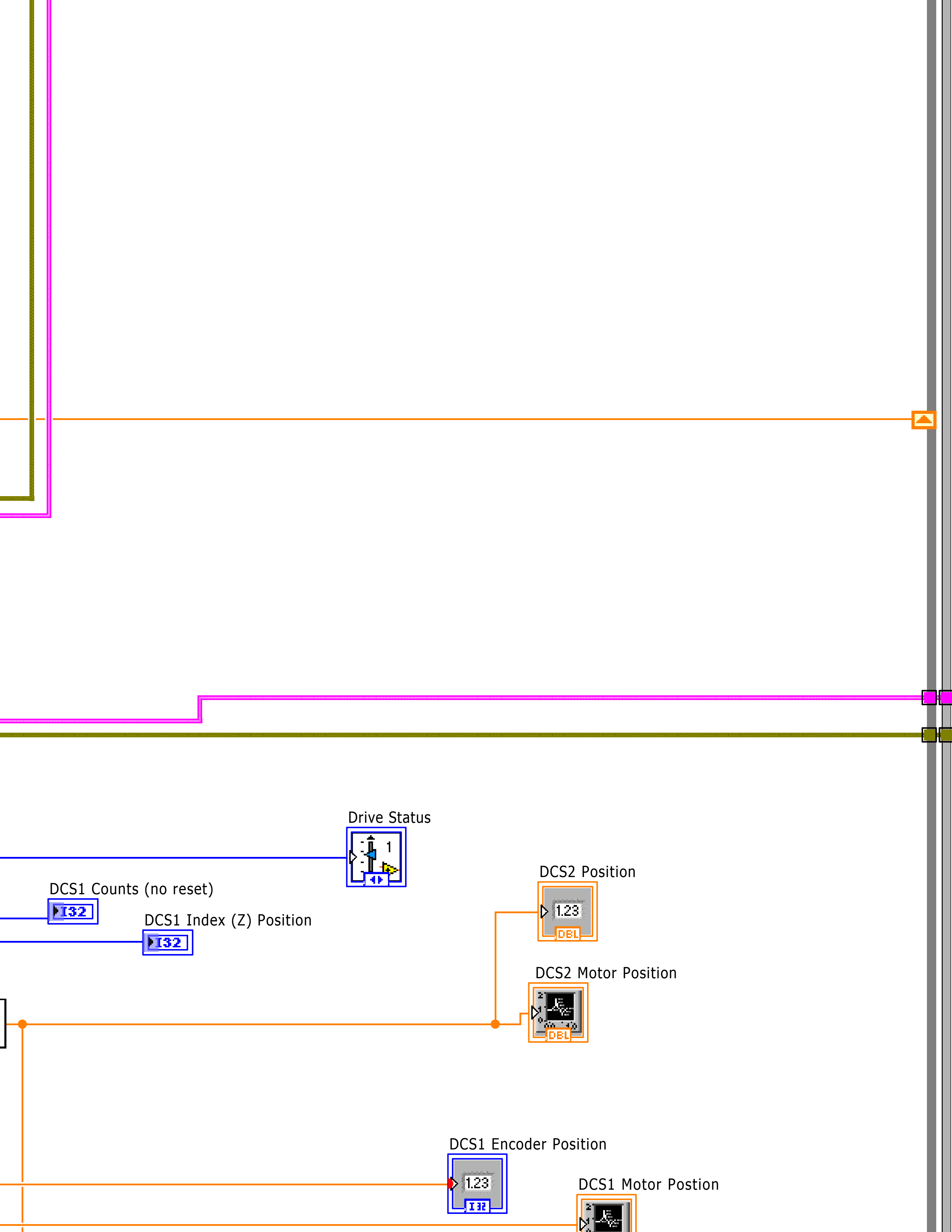

Clean Up
VISA
C
error out 2

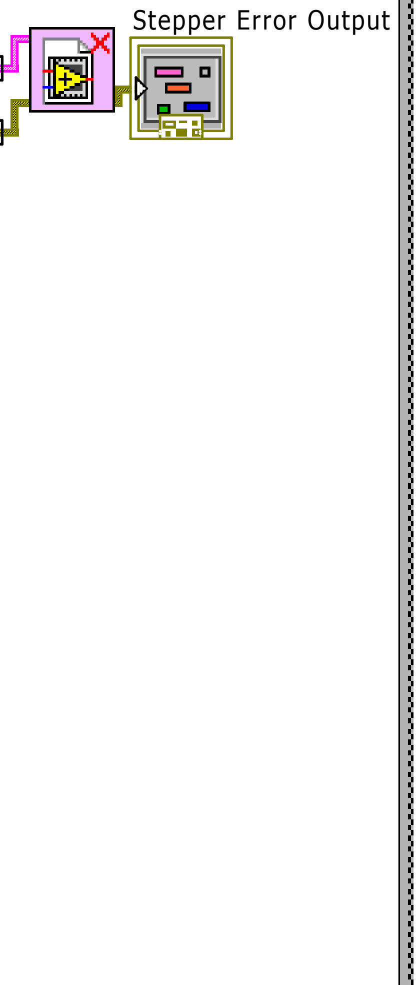

FPGA.vi
FPGA Target
RIO0

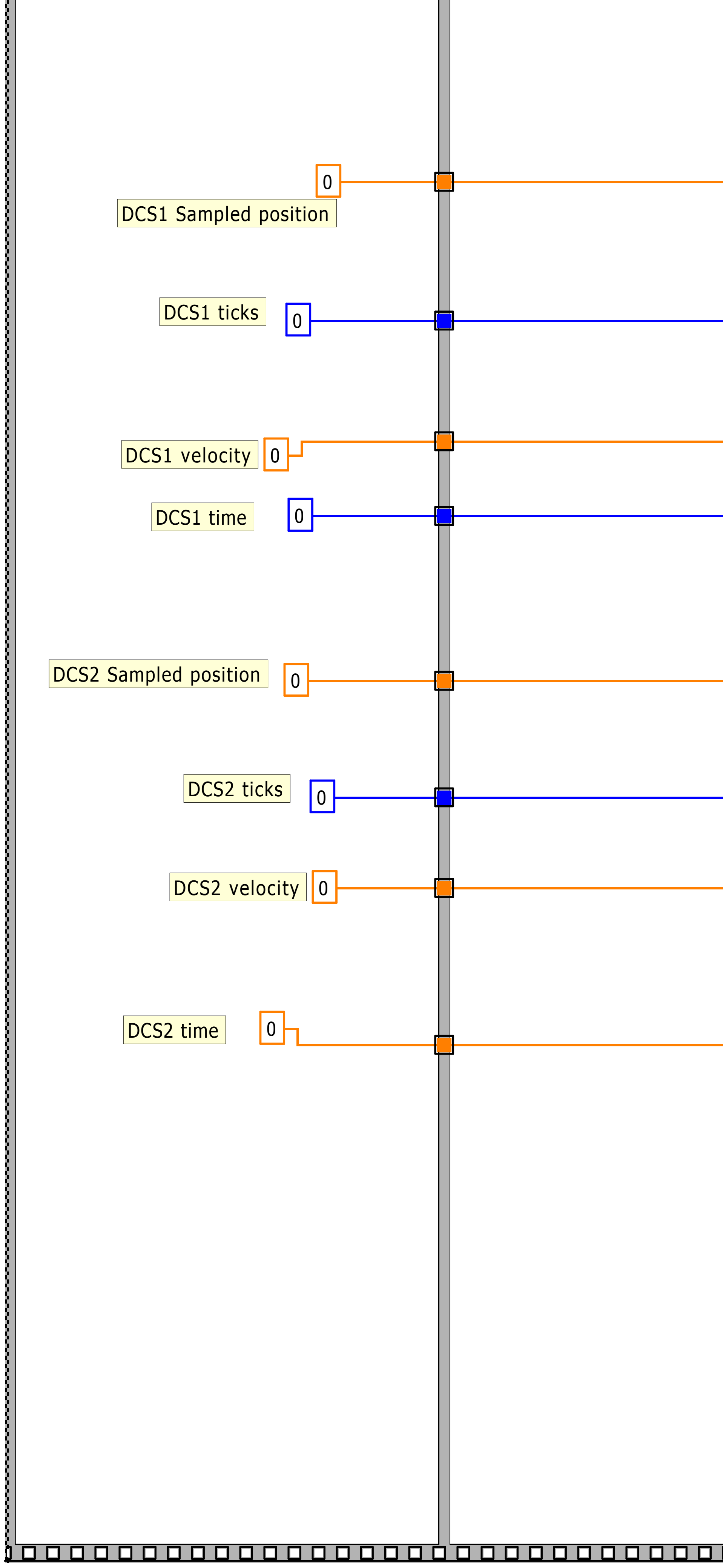


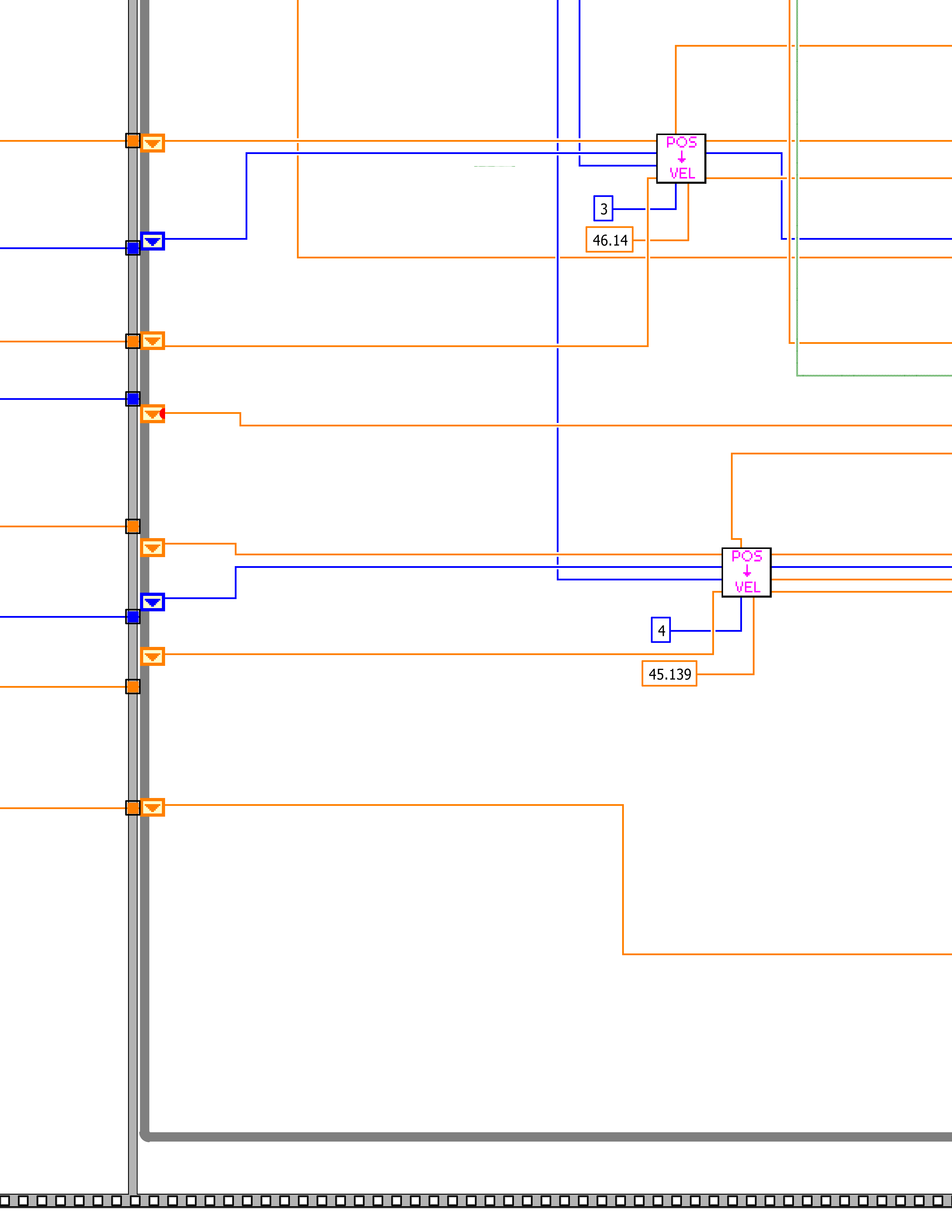
This either enables or disables the DCS1 motor

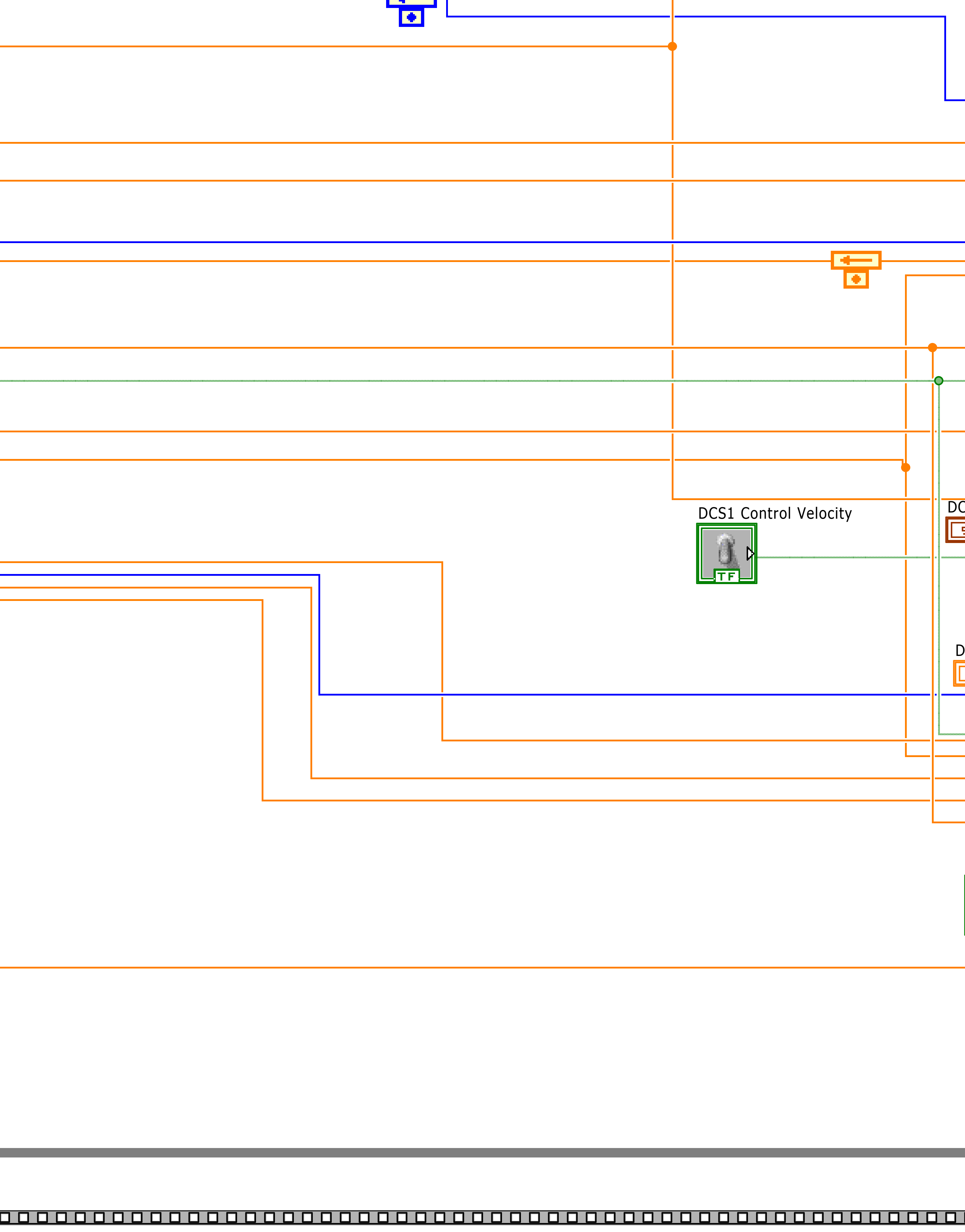

FPGA Stepper Output
FPGA Stepper Output
Commanded Position (steps)
Half-Step Time (ticks)
Reset Position to 0
Enable
Control Mode
I32
Current Position (steps)
Error (steps)
Zero
Enable
60
RPM
rotations/sec
360
deg / sec
1
360
1600
degrees
1.23
Heartbeat
DCS1 Enable/Disable Cluster
DCS1 Reset Position
DCS1 Reset Position Value
DCS1 Drive Control Cluster
DCS1 Encoder Position
DCS1 Status Cluster OUT
DCS1 Counts (no reset)
DCS1 Index (Z) Position
DCS1 Ticks/Count
DCS2 Ticks/Count
DCS2 Encoder Position
DCS2 Counts (no reset)
DCS2 Index (Z) Position
DCS2 Reset Position Value
DCS2 Reset Position
DCS2 Drive Control Cluster
Heartbeat
Drive Status
Vsup Present
Drive Fault
Over Temperature Fault
DCS1 Go
DCS1 Speed
TICKS DEGREE
TICKS DEGREE
on
set Position

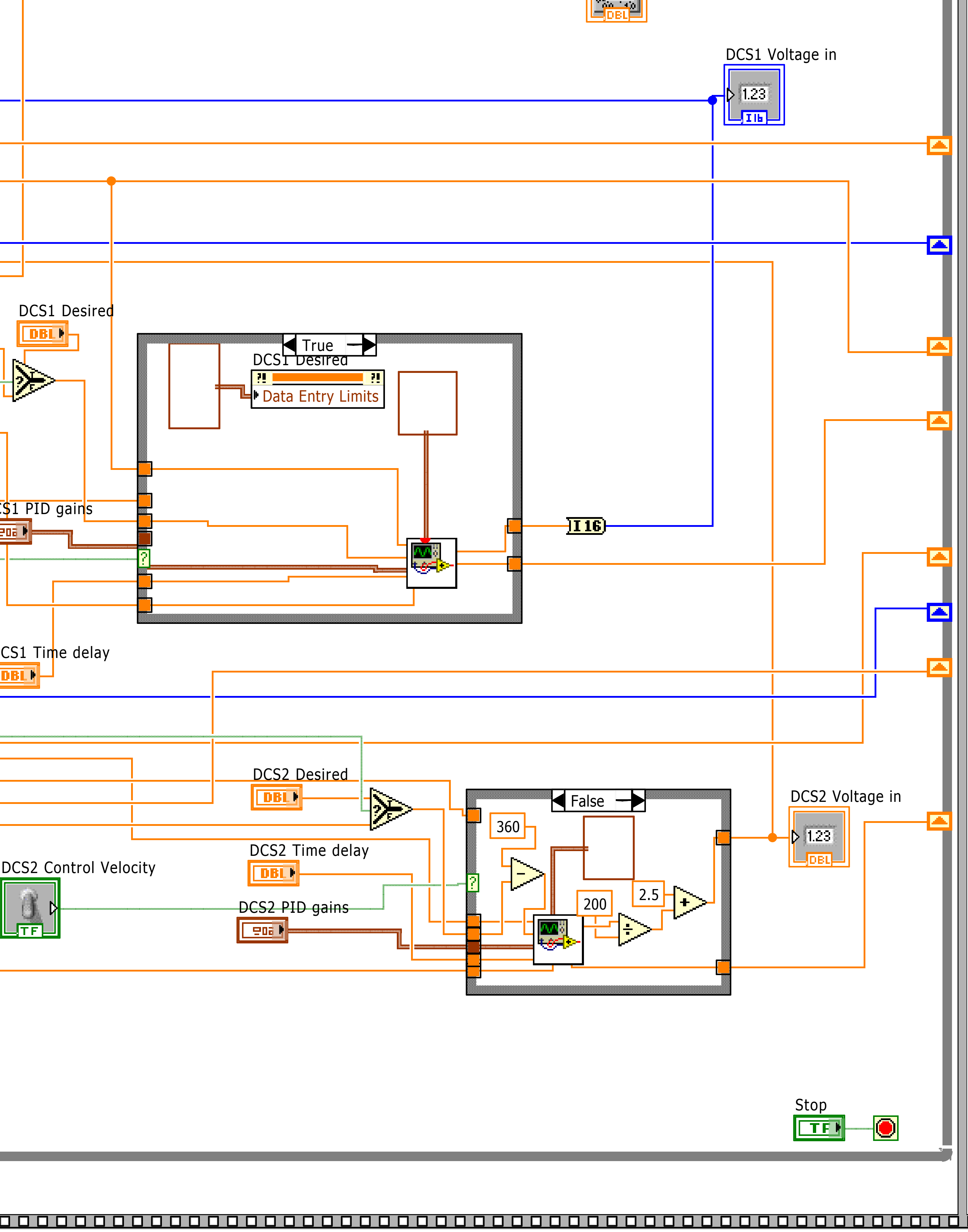

Drive Status
1
DCS1 Counts (no reset)
I32
DCS1 Index (Z) Position
I32
DCS2 Position
1.23
DBL
DCS2 Motor Position
DBL
DCS1 Encoder Position
1.23
I32
DCS1 Motor Postion


Stepper Error Output


0
DCS1 Sampled position
DCS1 ticks
0
DCS1 velocity
0
DCS1 time
0
DCS2 Sampled position
0
DCS2 ticks
0
DCS2 velocity
0
DCS2 time
0

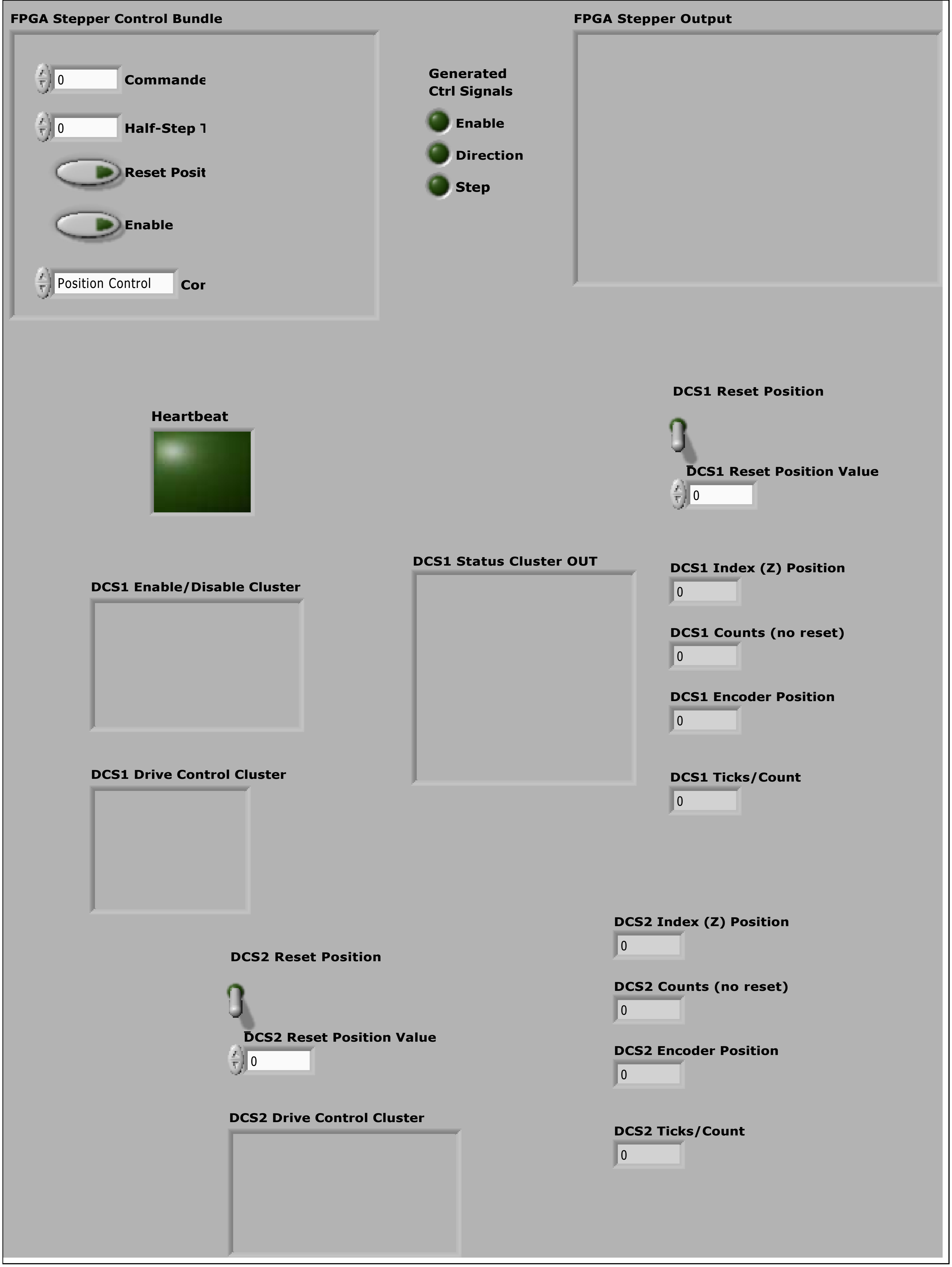

POS ↓ VEL
3
46.14
POS ↓ VEL
4
45.139


DCS1 Control Velocity
TF
DC
D



DCS1 Voltage in
1.23
I16
DCS1 Desired
DBL
True
DCS1 Desired
Data Entry Limits
CS1 PID gains
I 16
CS1 Time delay
DBL
DCS2 Desired
DBL
False
360
2.5
200
DCS2 Voltage in
1.23
DBL
DCS2 Time delay
DBL
DCS2 Control Velocity
TF
DCS2 PID gains
Stop
TF

FPGA.vi



FPGA Stepper Control Bundle

Commande: 0

Half-Step T: 0

Reset Posit

Enable

Cor: Position Control

Generated Ctrl Signals

● Enable

● Direction

● Step

FPGA Stepper Output

Heartbeat

DCS1 Reset Position

DCS1 Reset Position Value: 0

DCS1 Enable/Disable Cluster

DCS1 Status Cluster OUT

DCS1 Index (Z) Position: 0

DCS1 Counts (no reset): 0

DCS1 Encoder Position: 0

DCS1 Drive Control Cluster

DCS1 Ticks/Count: 0

DCS2 Index (Z) Position: 0

DCS2 Reset Position

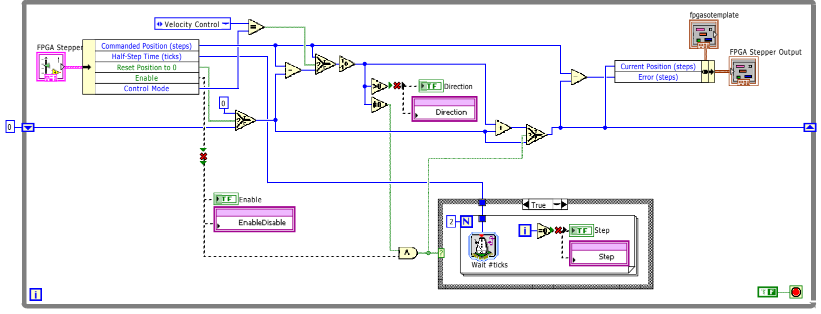
DCS2 Counts (no reset): 0

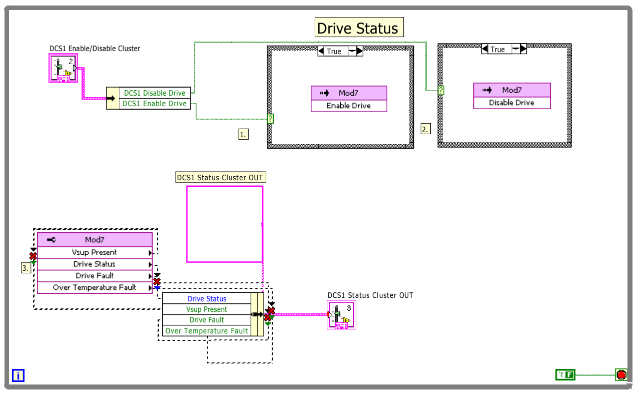
DCS2 Reset Position Value: 0

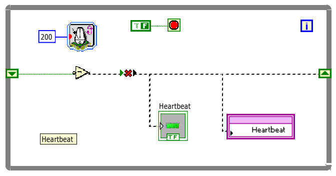
DCS2 Encoder Position: 0

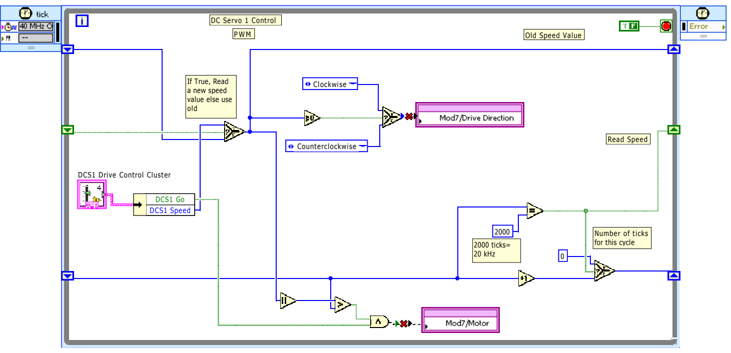
DCS2 Drive Control Cluster

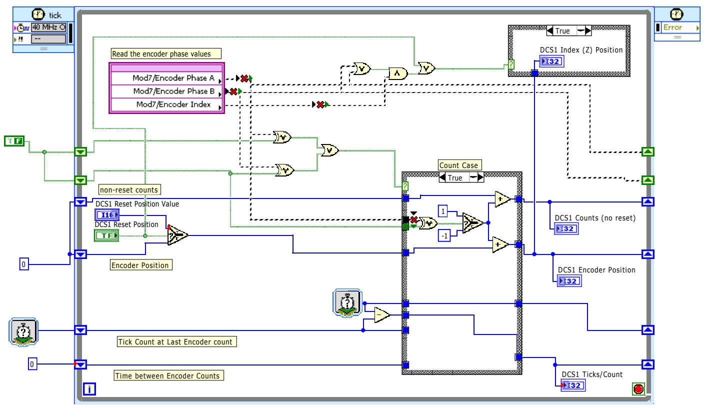
DCS2 Ticks/Count: 0

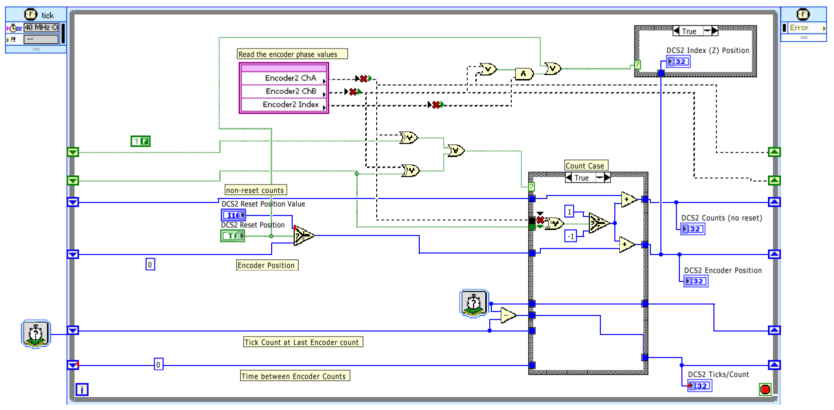

Velocity Control
FPGA Stepper
Commanded Position (steps)
Half-Step Time (ticks)
Reset Position to 0
Enable
Control Mode
Direction
Enable
EnableDisable
fpgasotemplate
FPGA Stepper Output
Current Position (steps)
Error (steps)
True
Wait #ticks
Step
0
2
N

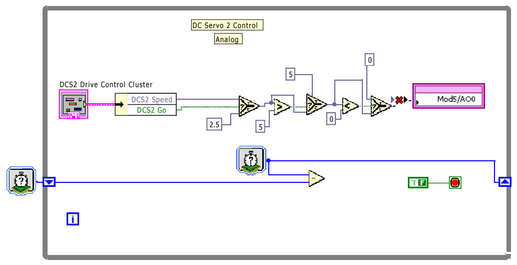

Drive Status
DCS1 Enable/Disable Cluster
DCS1 Disable Drive
DCS1 Enable Drive
True
Mod7
Enable Drive
Disable Drive
1.
2.
3.
DCS1 Status Cluster OUT
Vsup Present
Drive Status
Drive Fault
Over Temperature Fault

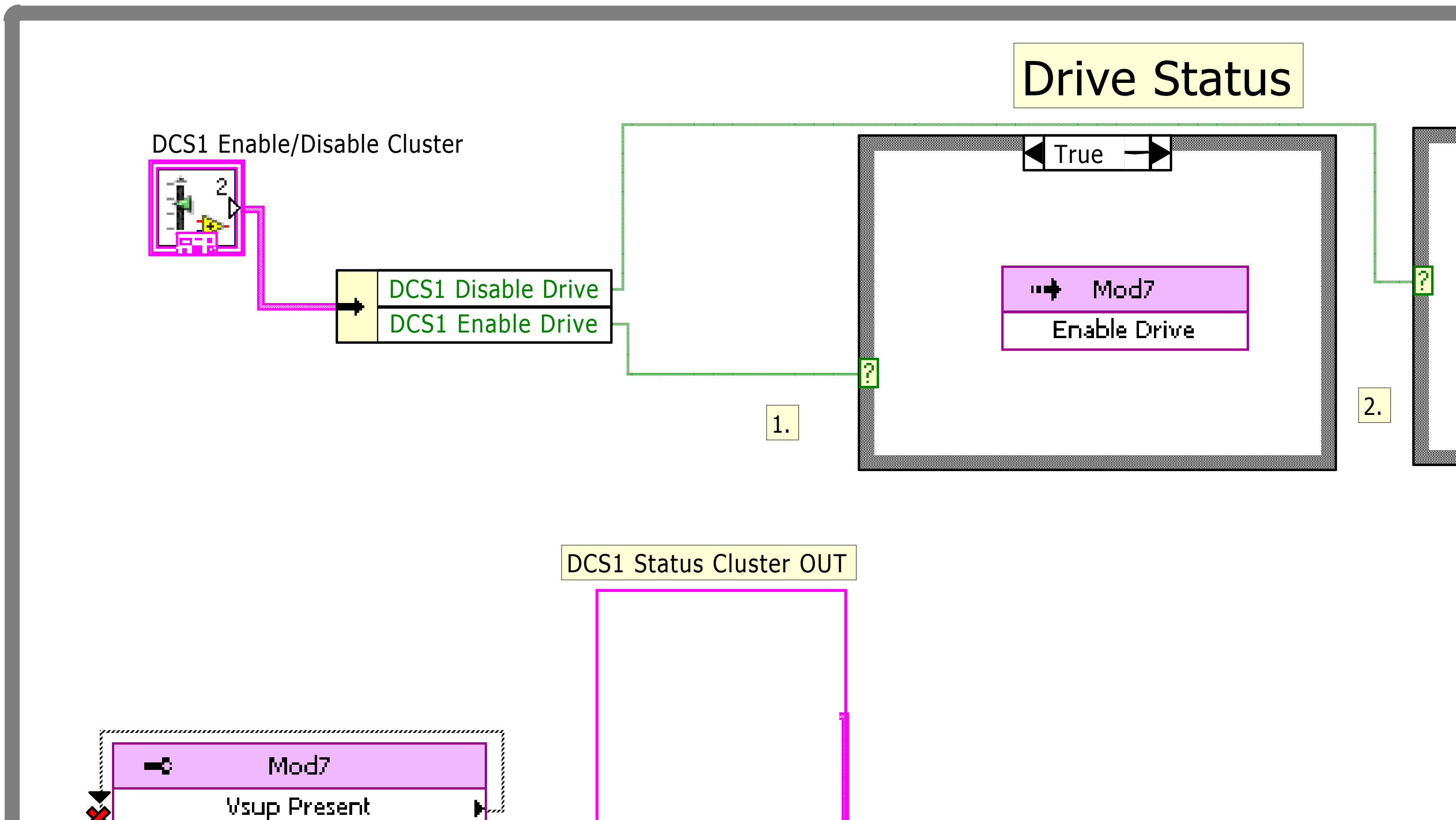

200
Heartbeat

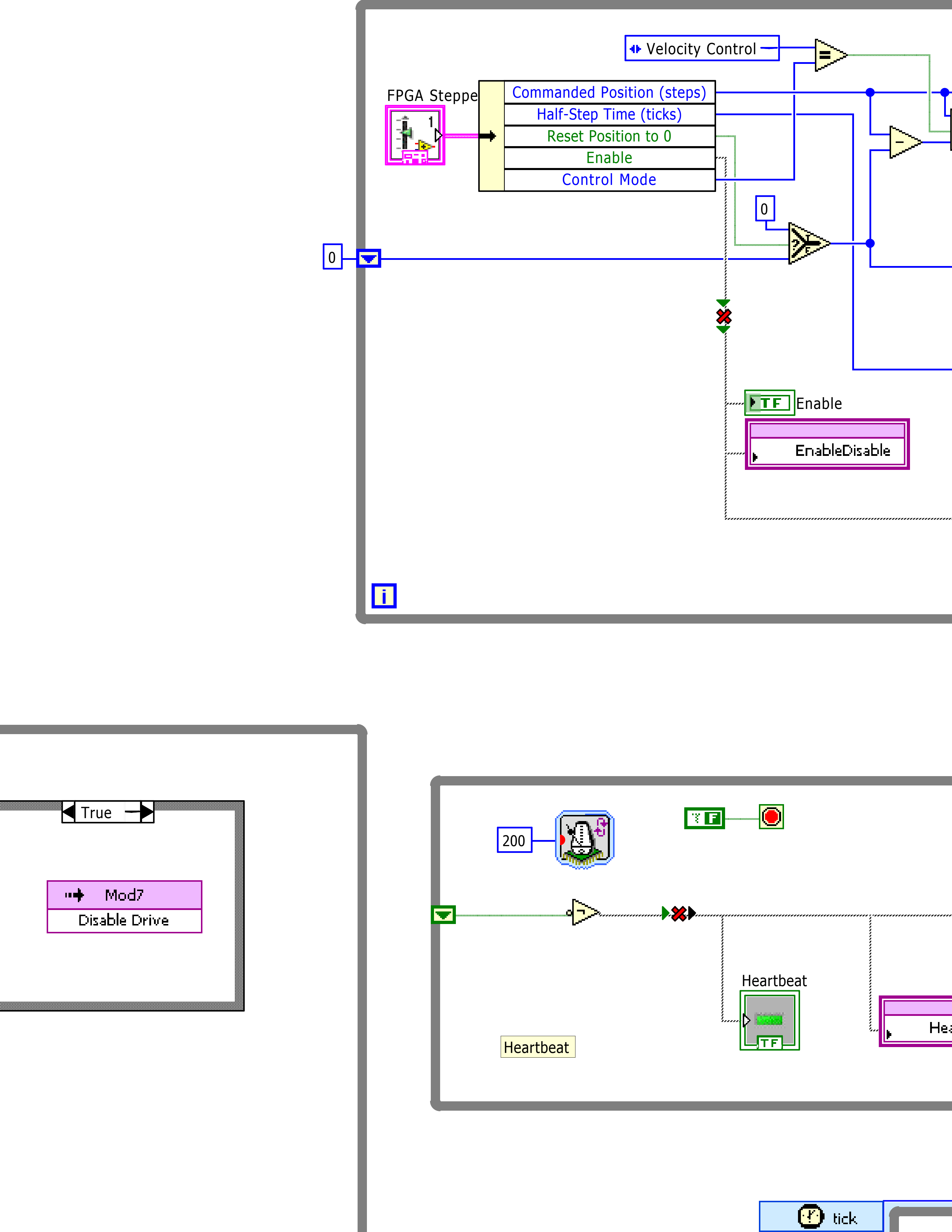

tick
40 MHz
Error
DC Servo 1 Control
PWM
Old Speed Value
If True, Read a new speed value else use old
Clockwise
Counterclockwise
Mod7/Drive Direction
Read Speed
DCS1 Drive Control Cluster
DCS1 Go
DCS1 Speed
2000
2000 ticks= 20 kHz
Number of ticks for this cycle
0
Mod7/Motor

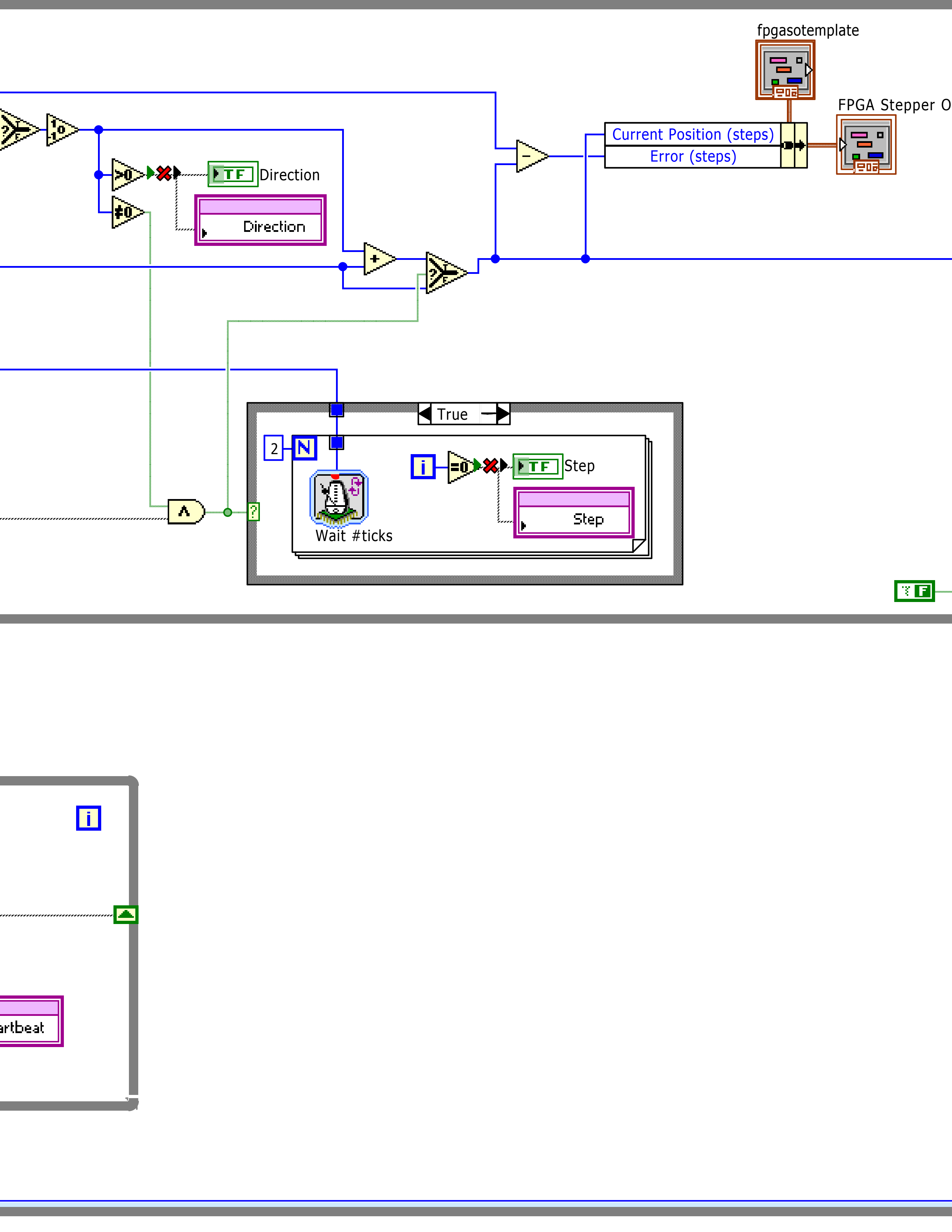

tick
40 MHz
Error
True
Read the encoder phase values
Mod7/Encoder Phase A
Mod7/Encoder Phase B
Mod7/Encoder Index
DCS1 Index (Z) Position
Count Case
non-reset counts
DCS1 Reset Position Value
DCS1 Reset Position
Encoder Position
DCS1 Counts (no reset)
DCS1 Encoder Position
Tick Count at Last Encoder count
Time between Encoder Counts
DCS1 Ticks/Count
0
1
-1

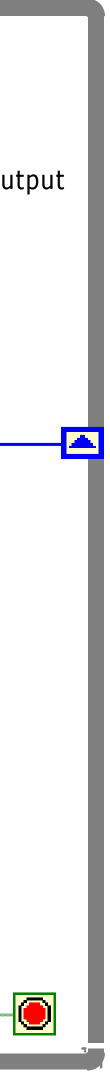

tick
40 MHz
Error
True
Read the encoder phase values
Encoder2 ChA
Encoder2 ChB
Encoder2 Index
DCS2 Index (Z) Position
Count Case
non-reset counts
DCS2 Reset Position Value
DCS2 Reset Position
Encoder Position
DCS2 Counts (no reset)
DCS2 Encoder Position
Tick Count at Last Encoder count
Time between Encoder Counts
DCS2 Ticks/Count
0
1
-1

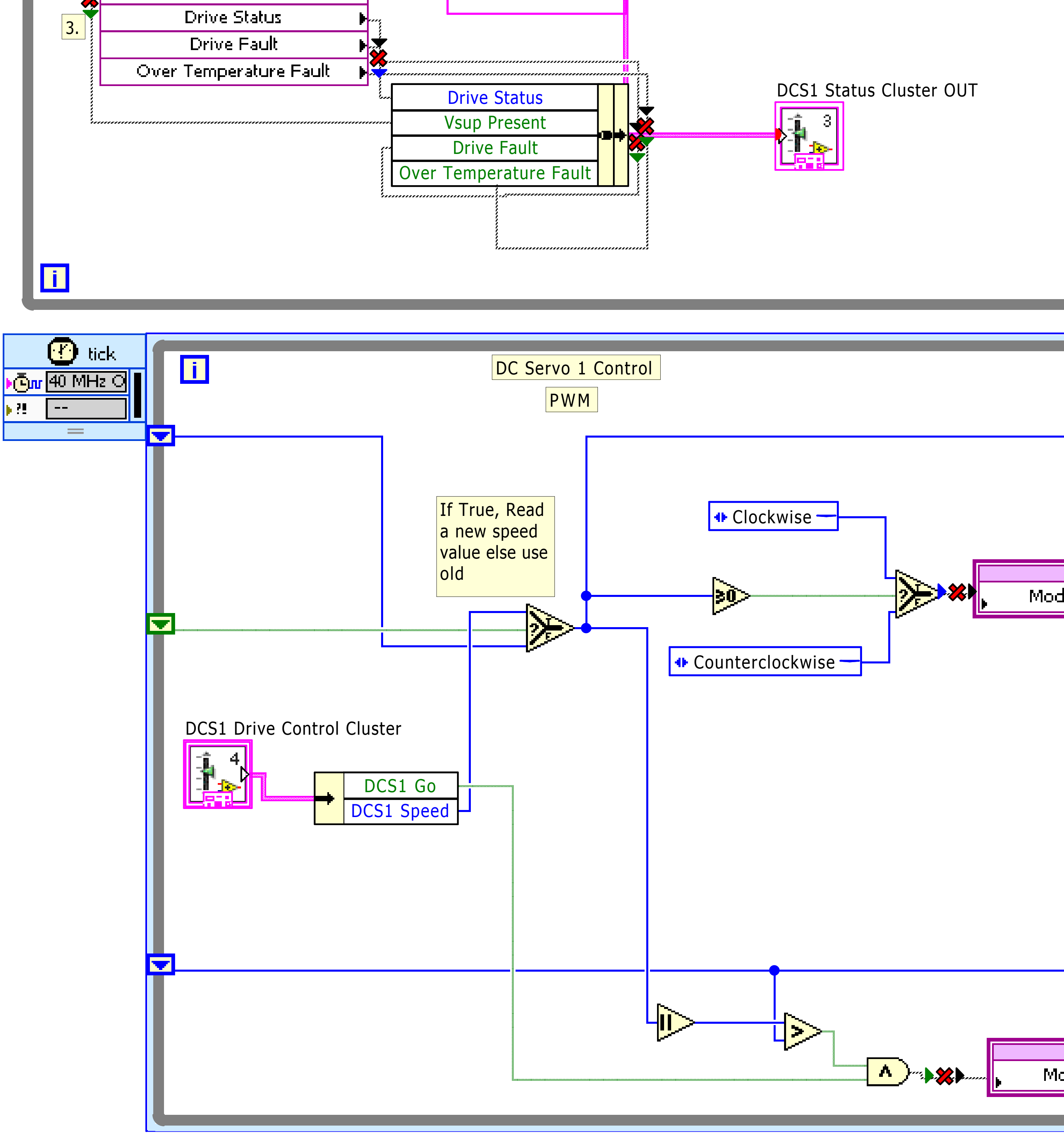

DC Servo 2 Control
Analog
DCS2 Drive Control Cluster
DCS2 Speed
DCS2 Go
5
0
2.5
Mod5/AO0

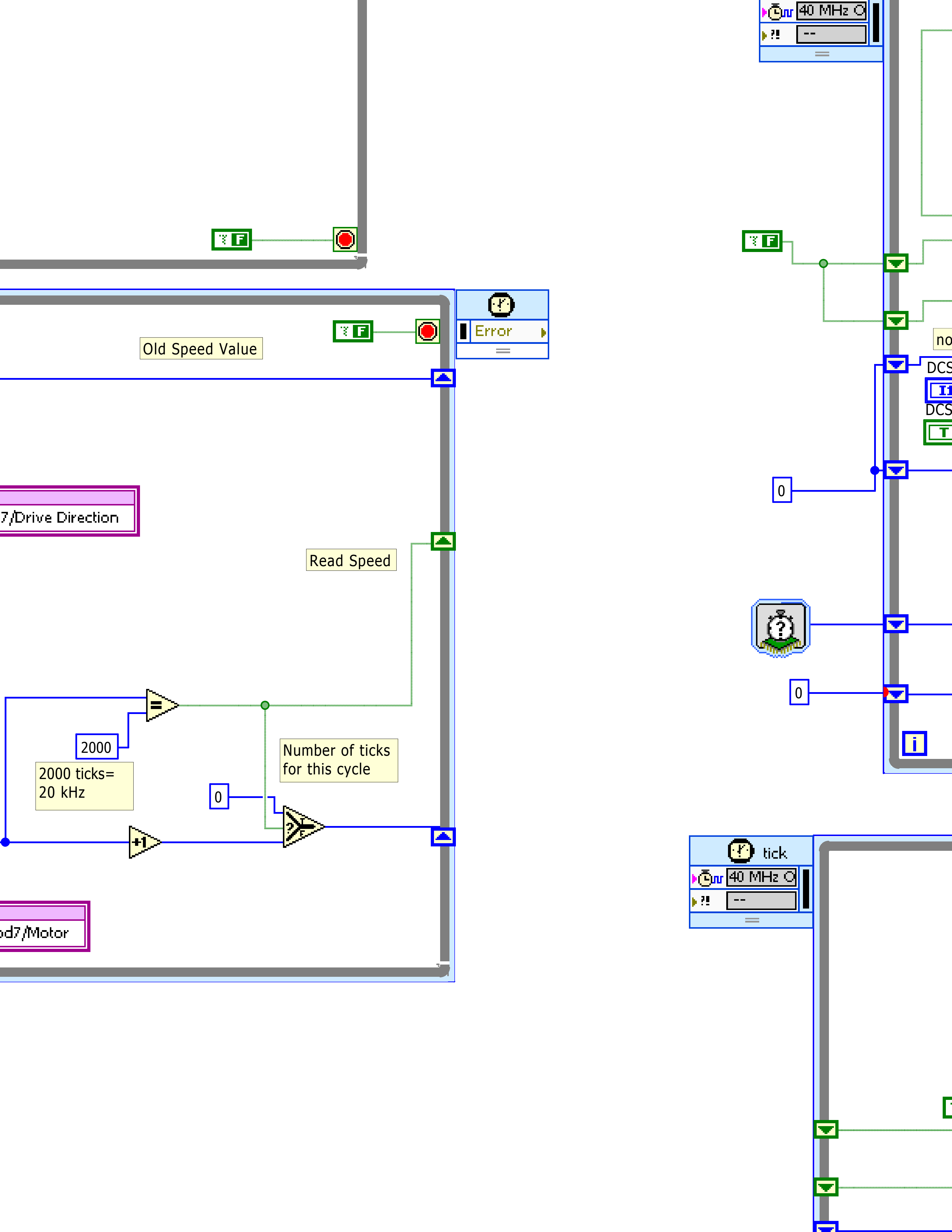

Drive Status
DCS1 Enable/Disable Cluster
2
DCS1 Disable Drive
DCS1 Enable Drive
True
Mod7
Enable Drive
1.
2.
DCS1 Status Cluster OUT
Mod7
Vsup Present

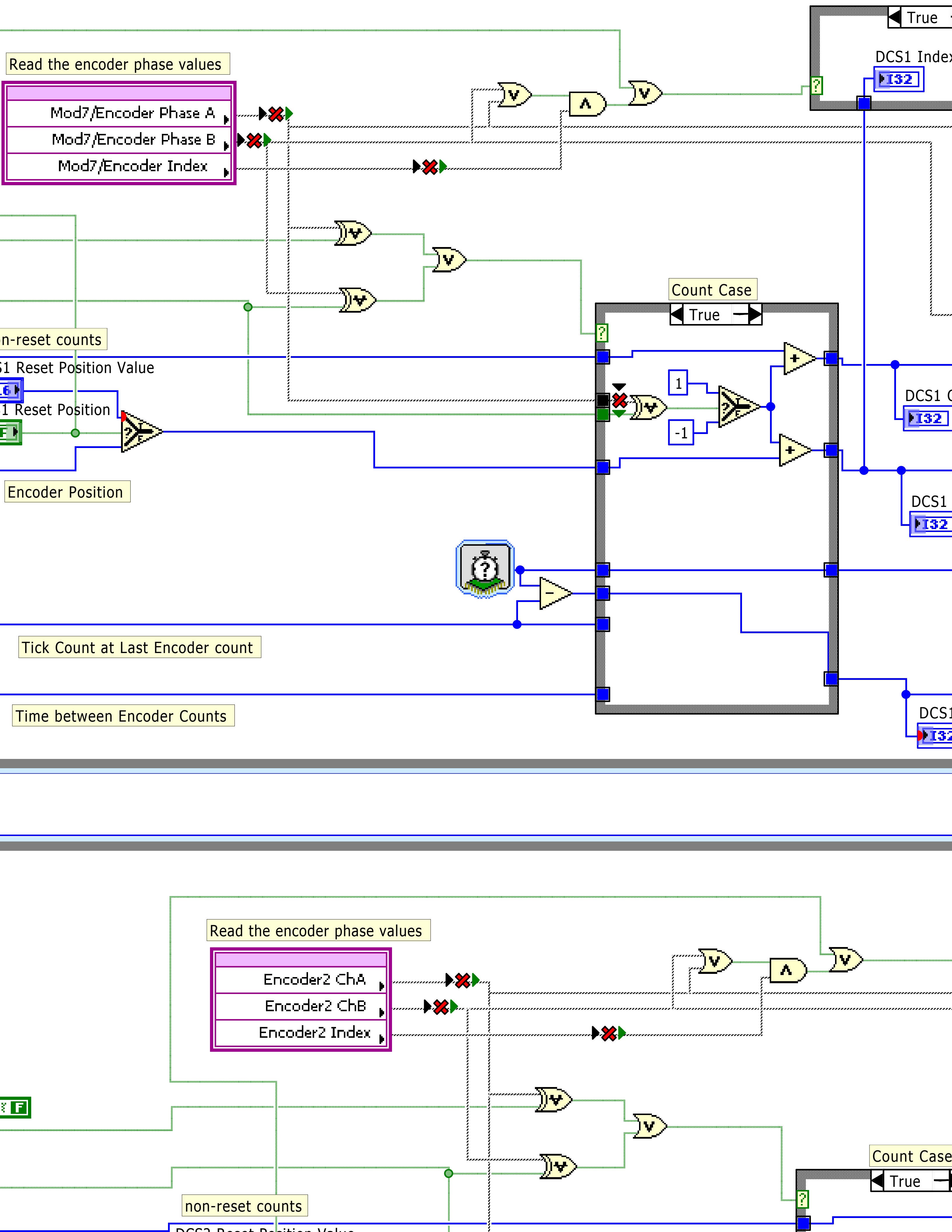

Velocity Control
FPGA Steppe
Commanded Position (steps)
Half-Step Time (ticks)
Reset Position to 0
Enable
Control Mode
0
0
Enable
EnableDisable
True
Mod7
Disable Drive
200
Heartbeat
Heartbeat
Hea
tick

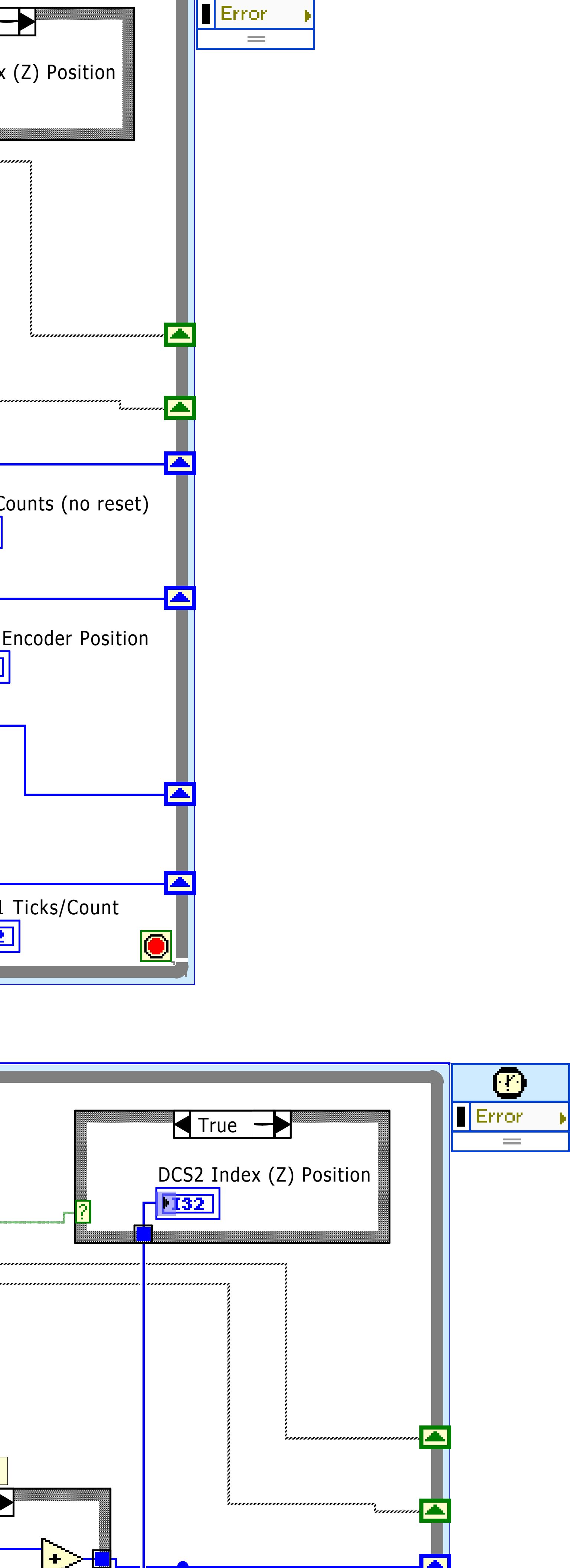

fpgasotemplate
FPGA Stepper O
Current Position (steps)
Error (steps)
Direction
Direction
True
2
Wait #ticks
Step
Step
artbeat

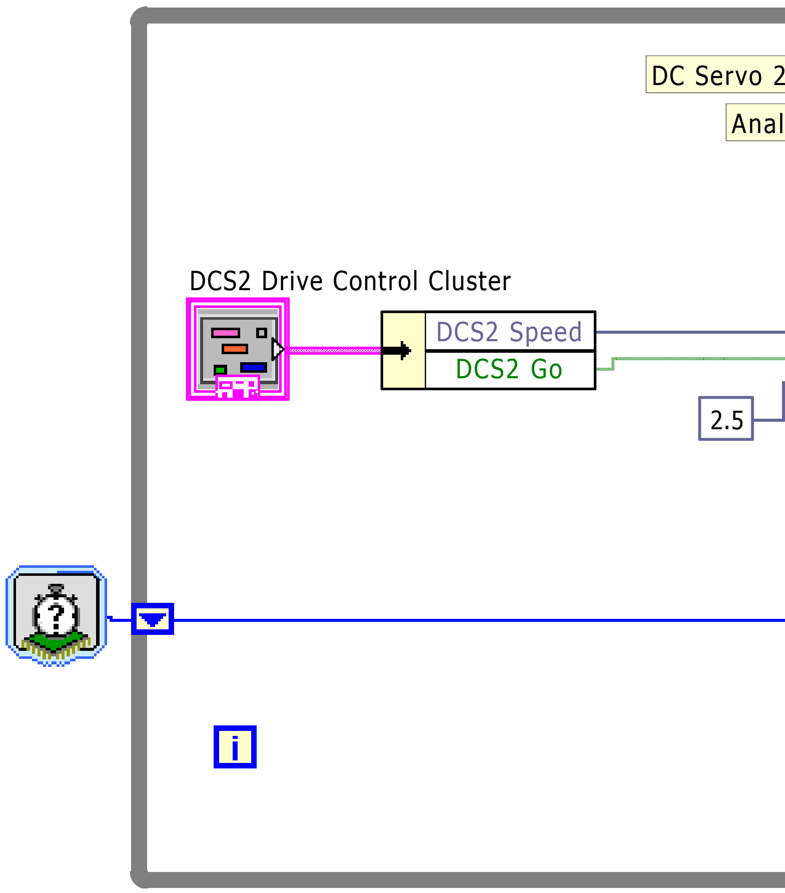

utput

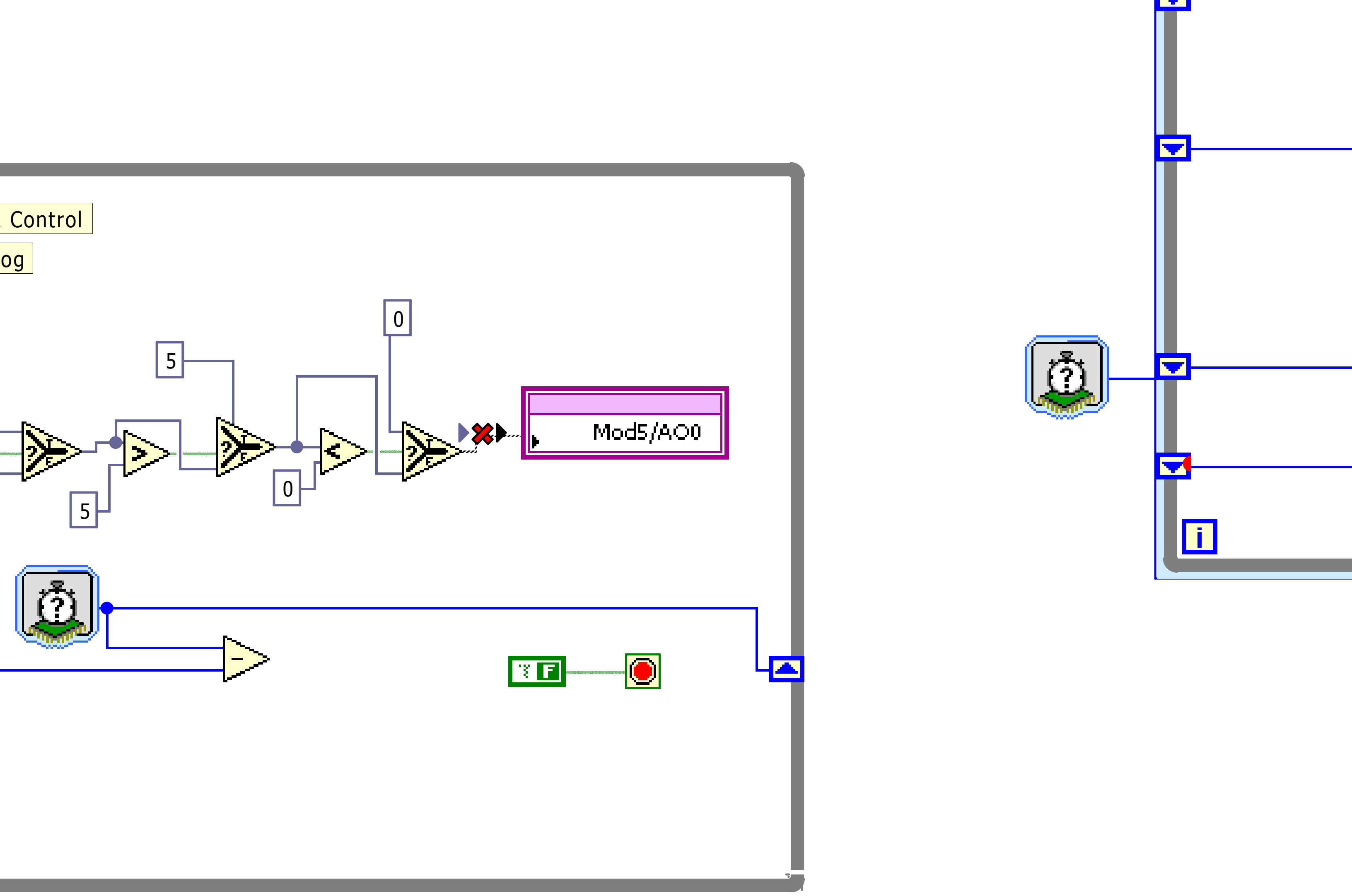

3.
Drive Status
Drive Fault
Over Temperature Fault
Drive Status
Vsup Present
Drive Fault
Over Temperature Fault
DCS1 Status Cluster OUT
3
i
tick
40 MHz
--
i
DC Servo 1 Control
PWM
If True, Read a new speed value else use old
Clockwise
Counterclockwise
Mod
DCS1 Drive Control Cluster
4
DCS1 Go
DCS1 Speed
Mo

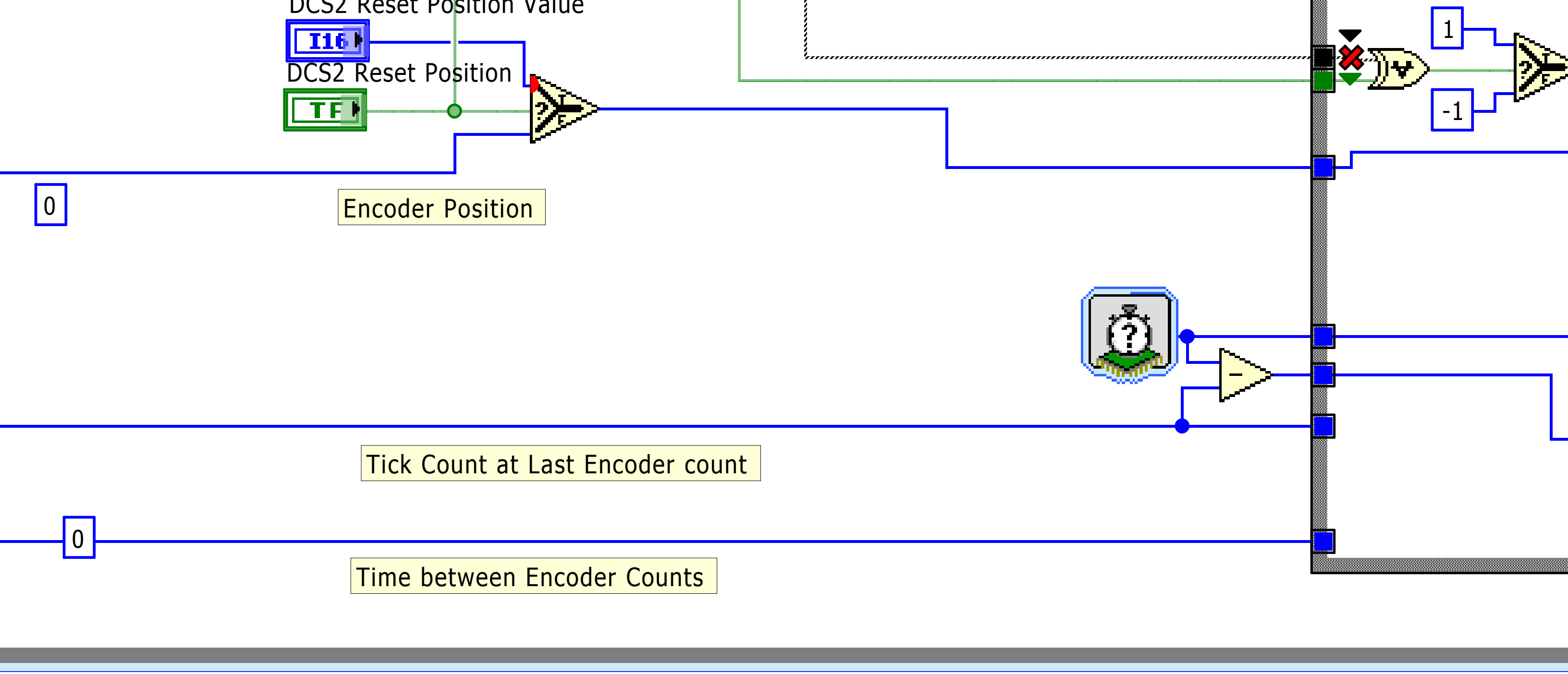

40 MHz O
--
Error
Old Speed Value
no
DCS
DCS
7/Drive Direction
Read Speed
0
0
2000
2000 ticks=
20 kHz
0
Number of ticks
for this cycle
+1
d7/Motor
tick
40 MHz O
--

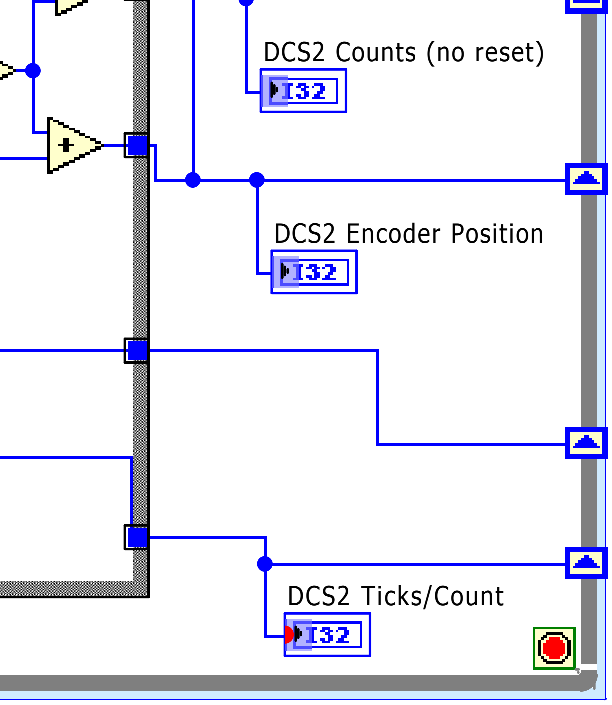

Read the encoder phase values
Mod7/Encoder Phase A
Mod7/Encoder Phase B
Mod7/Encoder Index
True
DCS1 Index
I32
n-reset counts
1 Reset Position Value
1 Reset Position
Count Case
True
1
-1
DCS1
I32
DCS1
I32
Encoder Position
Tick Count at Last Encoder count
Time between Encoder Counts
DCS1
I32
Read the encoder phase values
Encoder2 ChA
Encoder2 ChB
Encoder2 Index
F
Count Case
True
non-reset counts
DCS2 Reset Position Value


Error
x (Z) Position
Counts (no reset)
Encoder Position
1 Ticks/Count
Error
True
DCS2 Index (Z) Position
I32
?


DC Servo 2
Anal
DCS2 Drive Control Cluster
DCS2 Speed
DCS2 Go
2.5
i


Control
og
5
5
0
0
Mod5/AO0
T F
i


DCS2 Reset Position Value
I16
DCS2 Reset Position
TF
0
Encoder Position
1
-1
Tick Count at Last Encoder count
0
Time between Encoder Counts


DCS2 Counts (no reset)
I32
DCS2 Encoder Position
I32
DCS2 Ticks/Count
I32