

GONZAGA UNIVERSITY

School of Engineering and Applied Science

Final Report For Winch Embedded Controller System

Electrical and Computer Engineering 4

Engineering Design Team:

Troy Cosentino,
Megan Nickolaus,
Kelsey Zaches

Faculty Project Advisor:

Claudio Talarico

Liaison Engineer:

George Moore

Center for Engineering Design & Entrepreneurship

Project EE&CPEN 4

April 25, 2014

Gonzaga University

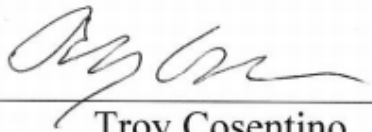
School of Engineering and Applied Science

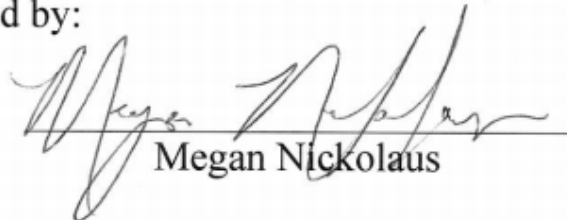
FINAL PROJECT REPORT

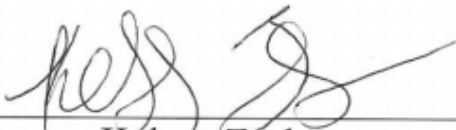
For

Winch Embedded Controller System

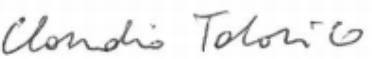
Prepared by:


Troy Cosentino


Megan Nickolaus


Kelsey Zaches

Reviewed by:


Advisor, Claudio Talarico


Project Liaison, George Moore

Center for Engineering Design and Entrepreneurship

Project EE 4

April 25, 2014

Contents

Introduction:	4
Work Performed:	6
Embedded System	6
Control Panel and Circuit	12
Simulations.....	14
Further Expansions Anticipated:.....	16
Acknowledgements and Thanks:	18
Appendices:.....	19
Appendix A: Winch Information	19
Appendix B: Matlab Code	19
Physics Simulation.....	19
Master Controller Simulation	23
Appendix C: Embedded Code and State Machine Drawings	24
State Machine Drawings	25
State Machine Code	34
Control Law Code	43
F4 IO pins.....	47
Appendix D: Control Panel and Parts.....	47
References:	51

Introduction:

The goal of this project was to develop an embedded controller system for a battery-electric winch that will be used by soaring clubs in the United States to launch sailplanes. Soaring is a sport in which a non-powered or low powered plane soars through the use of airstreams much like a bird. Currently in the US, sailplanes are most commonly launched by the aerotow method in which the glider is pulled by another plane to cruising altitude, then released. A less popular method in the US, though it is fairly common in European soaring clubs, is to use a combustion engine winch to launch the sailplane. Switching to a battery-electric winch would significantly lower the cost in comparison to aerotow launching, opening the sport to newcomers. In addition, the embedded controller system will allow for tension control during the launching process giving an increased performance and safety than its internal combustion engine counterpart.

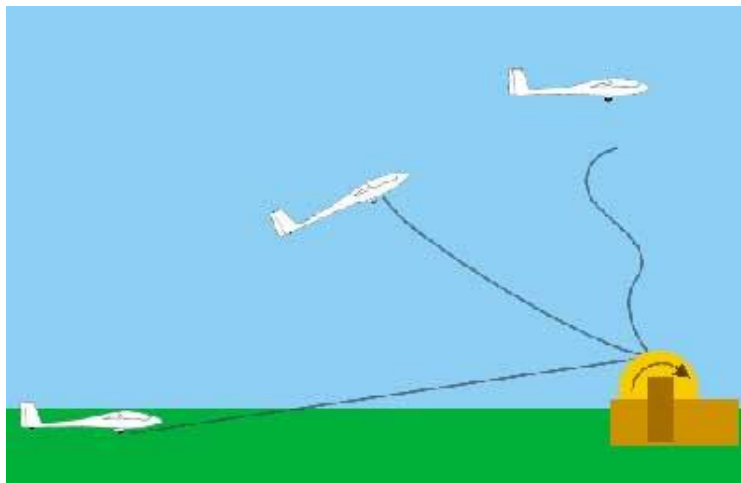


Figure 1: Winch launching a sailplane

In approaching this project, it was broken it into three main roles: the designing the embedded system called the Master Controller, simulating the physics and host controller, and creating the state machine drawings and designing the control panel. Over the course of the

project a prototype of the embedded controller, a simulation for both the sailplane physics and Host Controller, and a control panel prototype were developed. The completion of these four components allowed the team to make significant progress toward simulating a launch.

Figure 2 shows the control panel with the Host Controller (developed by a Computer Science project) computer on it, and the connections via CANbus to various subsystems. These subsystems are simulated in the physics simulation. The Master Controller is inside the control panel and takes the inputs from the control panel to process the state machine, then passes the messages to the Host Controller and subsystems. The control panel will be located inside the cab of a winch and operated by a person trained in its use.

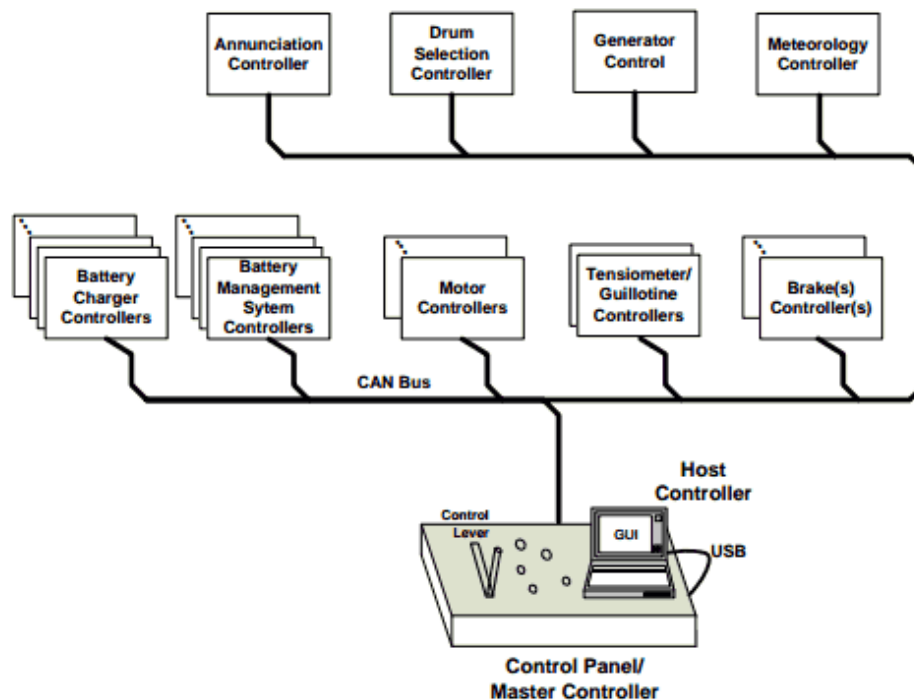


Figure 2: Distributed Real-Time Embedded Controller System

This project was intended to create a base level foundation for the embedded system to be used in future winches. The team focused on the success path of a winch launch and more

development is needed in non-success paths and lower priority functions. Work was also done to insure compatibility with the Host Controller being developed by the Computer Science 3 team. Further development is expected on the controller system to tailor them more specifically to individual clubs and allow the systems to take advantage of advancements in technology.

The code and plans involved in the system is open source and is hoped to be used by other soaring clubs in order to open the sport to a new generation of people.

Work Performed:

This project consisted of three major parts. The primary piece was the embedded controller system itself. Another main part was creating MATLAB simulations in order to test the embedded system. The third part was designing and building the control panel and circuit to feed the inputs to the embedded system, and circuit to drive LEDs. All of these parts came together in the final testing done with the master controller.

Embedded System

The embedded system design consisted of a multitude of smaller tasks. These included the creation of detailed state machines, coding the state machine and control law, and defining message structures for communication with the Host Controller and physics simulation.

The STM-32 F4 Discovery microcontroller was chosen for development. The current design uses two boards, but more could be added as further features are added to the system. One board functions as the master controller; it progresses the state machine, contains the control law, and passes messages to and from the host controller. A second board links the glider simulation to the CANbus, as well as simulates other inputs needed by the state machine. In future

iterations, this board will be connected to various environmental inputs. Currently, both boards connect to a PC - one to the host controller simulation, and the other to the physics simulation computer - via a serial to USB conversion chip that allows communication between the F4 and the PCs.

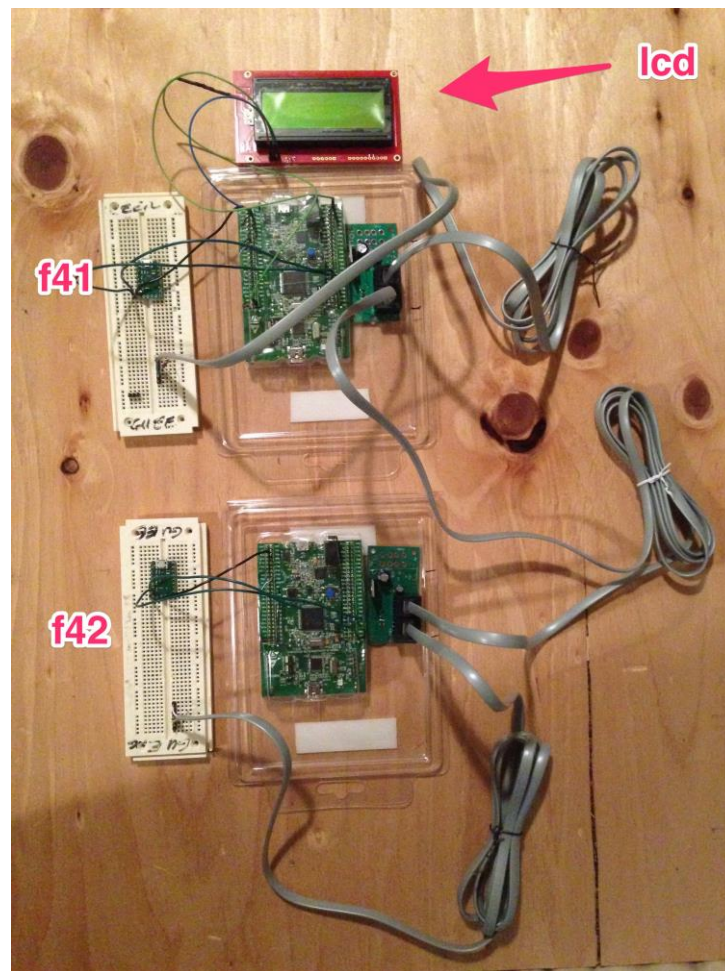


Figure 3: STM-32 F4 Discovery boards and LCD

A high level state machine was provided at the outset of the project, from this and from a white paper provided by the liaison, a detailed state machine was created detailing each of the ten states and showing sub-states. From these state machine drawings, the state machine was coded.



Page 8

abnormal ways to exit from the Arm state, indicated to the system by a button pressed on the control panel, but a normal launch will get the launch parameters from the Host Controller then progress to the Profile state when the control lever is fully advanced. The Profile state corresponds with ground roll and early rotation in the launch; the state machine looks for any abnormal or emergency procedures as indicated in various ways on the control panel or otherwise advances to the Ramp state when the speed of the cable has fallen by a certain percentage (generally around 5%) of the peak value. At this point the Ramp state sets a timer which indicates when to advance to the Constant state. The Constant state occurs during the climb phase with constant tension. It exits to Recovery when the maximum cable speed has been reached. In Recovery, the cable is being lowered to the ground attached to a parachute to control speed. This exits back to the prep state when the cable speed is zero and the prep/retrieve button is pushed. The Retrieve state allows for calibration to occur. There are various safety check that occur in the state machine causing movement into the Stop state or the Abort State. Details of these procedures can be found in the state machine diagrams in the appendix, and further development is planned by the project liaison to ensure safety before the system is implemented for used by any soaring clubs.

In order for the system to function, the Master Controller must be able to communicate clearly with the various sub-states of the system. It was decided to pass the messages through the system using CANbus. CANbus determines message priority based on the ID of the message being passed. A higher priority message is indicated by a lower ID. This allows the system to recognize when a message is related to safety during the launch. High priority is given to messages going to the brakes and contactors so in case of an emergency the winch and cable can be stopped as quickly as possible. The team has defined an ID structure in accordance with this

protocol, and assigned IDs to all currently known message types. The first four of eleven bits will be used to designate different levels of priority. The different priorities are shown in Figure 5 below.

First 4 bits	Meaning
0010	Fixed High
0011	<u>High</u> /Low
0100	High/ <u>Low</u>
0101	Fixed Low
0110	Fixed Low
0111	Fixed Low
1000	Fixed Low
1001	Fixed Low
1010	Fixed Low

Figure 5: Message Priorities

The remaining seven bits are used as the actual message identifiers within each priority level. Conventions have been defined to allow the same message type with different destinations. This is to allow easy additions in future versions of the system. For example, there may be multiple motors running the winch that need to receive the torque/speed command. The first five of seven bits are the ID of the message, leaving the final two to designate which of the four possible motors the message is for.

The CodeSourcery tool chain was used to compile, flash, and debug the program. The team has an ubuntu computer setup with the tool chain, and is able to run the whole process. Additional time was spent learning how to use the debugger with the microcontroller which helped with testing and debugging.

An application called hub-server was developed by the liaison's colleagues, Don Hasselwood and Charles Wells, which assisted in the development. The hub-server allows the simulation and Master Controller to connect wirelessly when they are both connected to the same local network rather than connecting them directly. This also allows the Computer Science team to listen to messages passing between the simulation and master controller. The simulation connects to an open socket on a computer connected to the master controller, and the hub-server relays all messages it receives from the master controller to connections on its socket.

Another important part of the master controller is the control law. Upon completion of the entire project, anticipated by the liaison to happen in the next year, the master controller will send messages to the engine controller indicating desired drum speed or cable tension. Our system simulates this relationship, the physics simulation receives these messages and uses them to advance the simulation. The control law uses defined parameters, called launch parameters, generated specifically for each glider/pilot combination, as well as the current tension or cable speed of the system to determine what torque to require from the motor.

During different stages of the launch different behavior is desired, and therefore the control law uses the current state of the state machine to determine which calculation to use. For instance, during an actual launch most of the process is tension commanded, however in the Retrieve state the cable is speed controlled so the cable never moves faster than a person on the ground. The actual calculations were defined by the liaison during the project. These include quarter sine taper downs to allow for smooth flights, and relations between torque and tension. In the future, more detailed equations will be implemented accounting for stretching in the cable and other complicated factors that were put aside in order to complete a base level control law to

build from. In many states, the control law can be scaled through the use of the control lever on the control panel which allows the winch to react to unanticipated factors.

Control Panel and Circuit

A control panel circuit was designed to provide the inputs to the embedded system, and to display the outputs as well. The inputs were taken from the buttons and switches on the control panel by a parallel in serial out shift register (74HC165) and sent to the F4 in order to indicate advancement in the state machine. The embedded system then sends outputs by a serial in parallel out shift register (74HCT4094) which activates LED drivers according to the state the launch is in.

The team mapped out the I/O pins on the F4 Discovery board used to implement the project. The spreadsheet can be found in the appendix. Many of the pins on the F4 are used by functions built into the board, so care was being taken to avoid using pins required by built in functions. In order to reduce the number of pins required, it was decided to use SPI and shift registers to drive LEDs and read from switches as discussed previously. The main control lever on the control panel produces an analog signal that is passed through an analog to digital converter before being used in the master controller.

The control panel itself went through multiple designs taking into consideration ergonomic factors as well as aesthetic, and a final design was submitted to George for fabrication. He found that the design did not leave enough room between buttons, and adjusted the layout in order to allow it to be built.

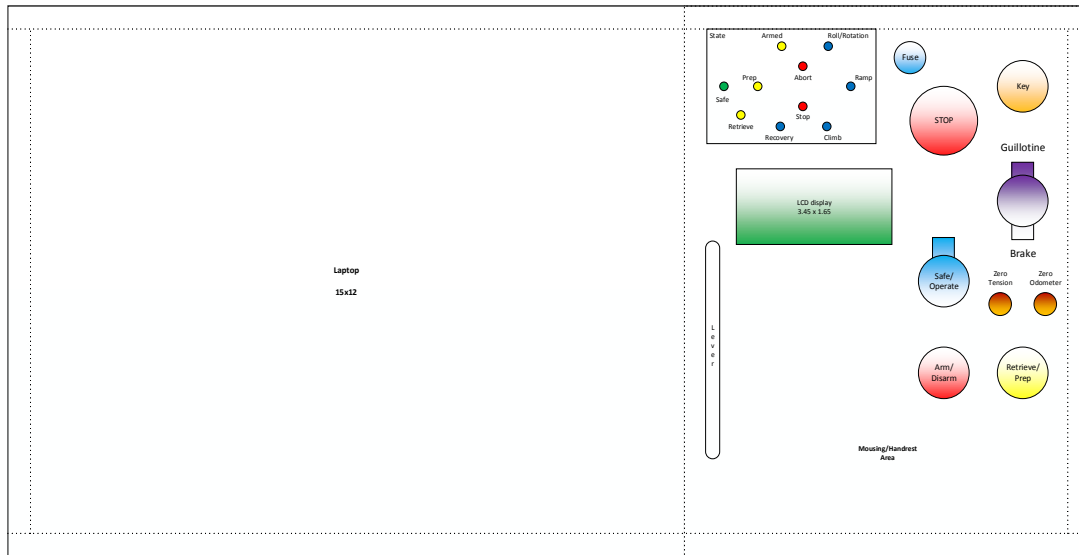


Figure 6: Final Control Panel Design

The team built the control panel circuit which allowed a launch to be simulated using the control panel. The control panel was used to debug the embedded system code and to advance the simulation.



Figure 7: Built Control Panel

Simulations

The initial method chosen to create the simulation was a combination of Matlab and Simulink. This method gave the desired results in the physics simulation but without additional software Simulink was unable to run a real-time simulation as desired. To have the simulation run in real-time, it was decided to use two Matlab instances instead. One instance of Matlab simulates the functionality of the master controller, and the other consists of the physics simulation; which was originally planned to be in Simulink. The two instances of Matlab connect over a java socket.

The master controller acts as the server in reference to the socket connection. The master controller opens the socket connection. Once connected, the program enters a for-loop in which the master controller sends the initial time and tension values as an object over the socket. After the physics simulation sends a message back, the master controller calculates a new tension based on the received information. The initial values are update to progress the simulation. This continues until the maximum time set in the master controller is reached.

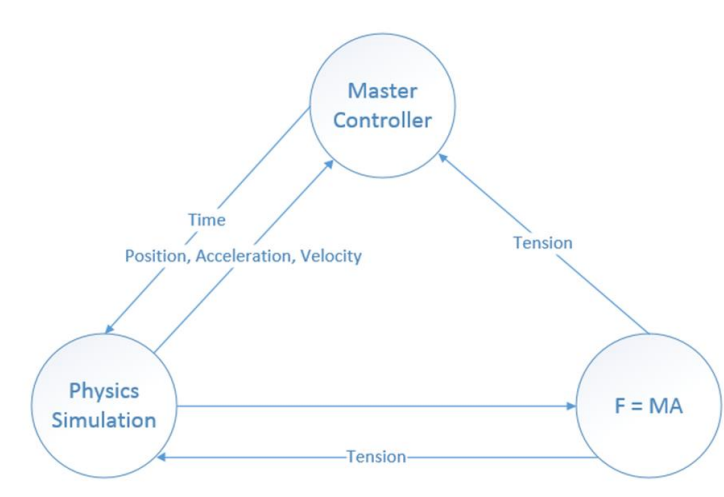


Figure 8: Connection for testing Matlab-Matlab functionality

When the physics simulation receives the initial values of the time and tension from the master controller it responds with the initial values of the state position, velocity, and acceleration of the plane. These values are written as an object over the socket. The simulation progresses and uses the Runge-Kutta method to update the values of the position, velocity, and acceleration for a one-sixty-fourth second interval. Once these values are updated the new state is determined. The state reflects the point of launch in the simulation which affects the value of the calculated tension. This process repeats until the defined time is reached. The figure below shows the output of the simulation.

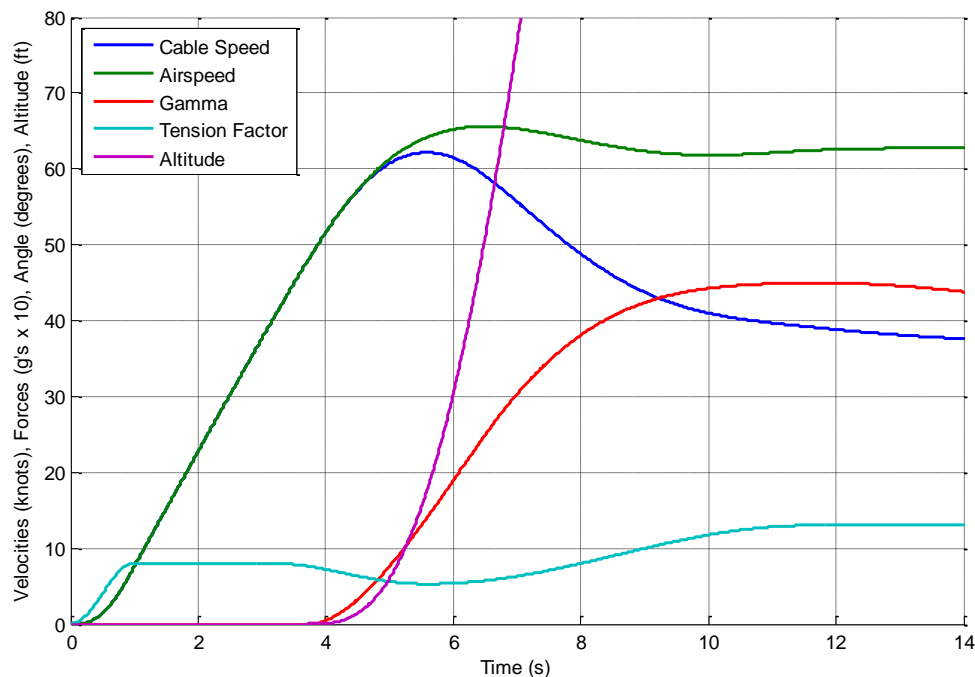


Figure 9: Output of the simulation shows multiple aspects of launch

Work is still being done on the simulation. The skeleton of the system is the same, but instead of using the Master Controller simulation developed in Matlab, it is being connected to the Master Controller of the embedded system. The difference between the simulation described above and the simulation program still in development is the message formatting. In the current

simulation the process passes objects over the socket. In the program in development, the messages are sent and received as strings in CAN message formatting.

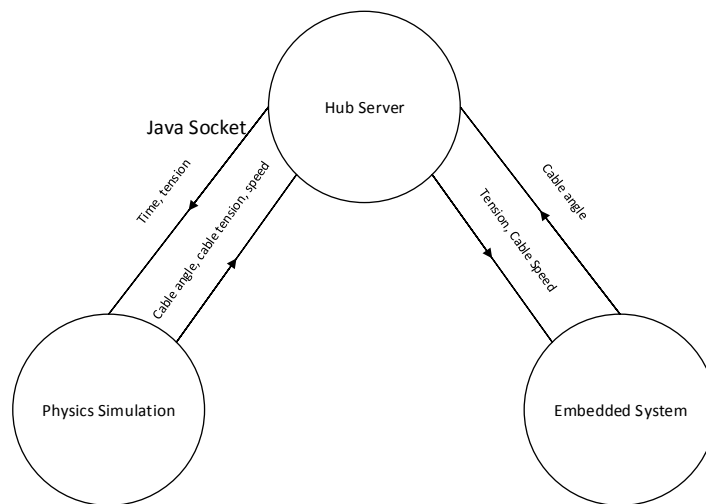


Figure 10: Connection for Matlab-F4 testing

The CAN messages received by Matlab are strings varying from sixteen to eighteen characters. The types of messages that are sent by the master controller are identified by the third through eighth characters. The unique ID of the message specifies the start of the launch, a time message, a torque message, and the end of the launch. From testing, the embedded system and physics simulation are able to connect over the java socket. Matlab also has the ability to receive and identify the CAN messages from the embedded system. The physics simulation is still in need of development to properly analyze the message sent as well as send a CAN formatted message of its own. This is anticipated to be completed in the next week allowing the system to be fully tested for presentation to the Design Advisory Board.

Further Expansions Anticipated:

The project was very ambitious, and over the course of working on it, the scope of the project was focus more and more. Decisions were made on high and low priority tasks, and most

low priority tasks were not accomplished. The team made sure to be compatible with expansions desired by the liaison. Below are some, but not all, of the anticipated expansions to happen before the system will be used in a real world winch launch.

One of the anticipated deliverables at the beginning of the project was to connect the embedded controller system with the user interface being developed by the related Computer Science team. This became a lower priority, though provisions were made on both sides to facilitate this combination in the near future. Work was done with the liaison and between the teams to ensure common messages and to create documentation. Testing for this is planned to occur before the end of the semester.

Another possible future addition is to connect a GPS to the system. This would allow for the best time messages possible as well as providing field location and elevation. Provisions were made for this possibility to happen in the future. Other subsystems may also be connected to feed other environmental data to the Host controller such as temperature, wind speed, and any other information that may be useful.

The original design was for a multi-drum system to be implemented. In order to achieve the most possible with the time available it was decided to do this prototype for a single drum winch, though there are some indications in the state machine as to what would need to change to implement multi-drum.

The final design of the embedded system will include a calibration mode for the winch. This will allow the operator to zero the tension and odometer ensuing safe launches. This was deemed non-essential by the team and left for future development, though it was indicated in the state machine drawing.

Another non-essential task was defining what “non-normal” brake and guillotine function means. This will have to be defined before a physical launch is performed.

Acknowledgements and Thanks:

The team would like to thank Donald Hasslewood and Charles Wells, colleagues of George Moore, who helped the team get started with the F4 Discovery boards and tool chain being used on the project. They also provided and walked through some pre-existing code including a driver for CANbus, and were instrumental in the success of the project.

We would also like to thank George and Professor Talarico for helping us navigate the project. They made themselves available for many long meetings. George is incredibly passionate about soaring and it was inspiring to work with him on a project that means so much to him.

Appendices:

Appendix A: Winch Information

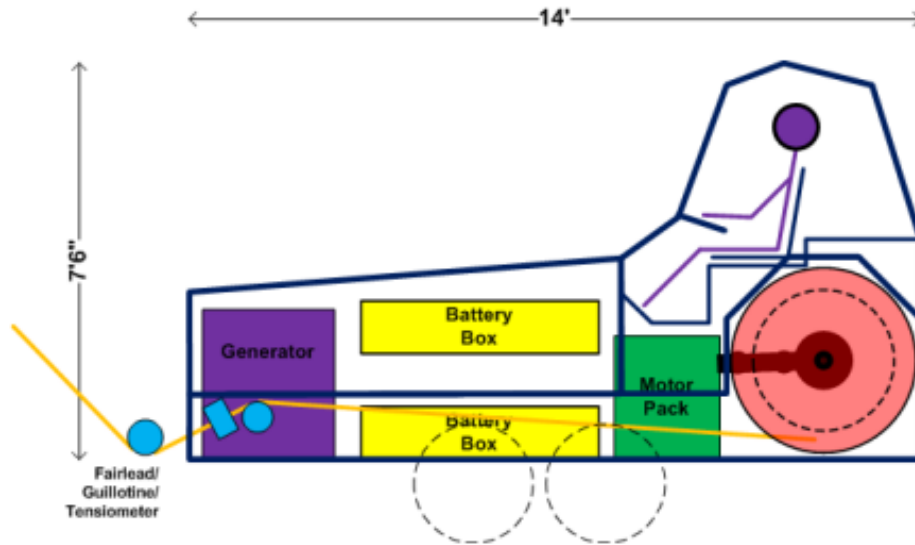


Figure 11: Full winch. The control panel is at the person's hands

Appendix B: Matlab Code

Physics Simulation

```
%This is the client
%This script will evaluate the position, velocity, and acceleration of the
%plane based on the tension and the state from the values that come from
%M_C.
function main
global u state fG ydotdot xdotdot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%conversion factors
m2f = 3.281; % Conversion factor for meters to feet
m2k = 1.9438; % Conversion factor for m/s to knots
G = 9.81; % Accel due to gravity

%Parameters
Vs = 36 / m2k; % 1G stall speed
V1 = 1.3333 * Vs; % V1 Speed
Xi = 5999 / m2f; % Run Length
LoD = 20; % L/D
Fg = 1.0; % Ground run tension factor
Fc = 1.3; % Climb tension factor
Vtr = 40/m2k; % Velocity to start tapering tension
```

```

Vmx = 81/m2k;           % Velocity where tension reaches 0
Ttu = 1;                % Initial taper up period
tpr_g = 2;              % Ground roll taper; 1 for sine, 2 for raised
cosine
tpr_r = 1;              % Rotation taper; 1 for sine, 2 for raised cosine
tpr2c = 2;              % Taper to Climb, 1 for sine; 2 for raised cosine
Trmp = 6;               % Tension factor ramp up period
Kd = 0.08;              % Derivative gain
lambda = G / V1^2;      % Lambda lift factor
delta = lambda/LoD;     % Delta drag factor
sc = (pi/2)/(Vmx - Vtr);
FcG = Fc * G;
FgG = Fg * G;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                                %Initialization Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
t1 = 1/64;
tinc = 1/64;
xdotinit = 0;
ydotinit = 0;
yinit = 0;
state = 0;
xyint = [Xi xdotinit yinit ydotinit];
data = [0 0 0 Xi xdotinit yinit ydotinit 0];
v_int = [0 0 0 Xi xdotinit yinit ydotinit 0];
Tmax = 60;
num_tim_intrv = Tmax/tinc + 1 ;           %Maximum number of time steps
data = zeros(num_tim_intrv-20, 8) ;      %Preallocate data variable for maximum
number of rows
data(1,:) = v_int;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                                %Define Socket Connection
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
import java.net.*
import java.io.*
import java.lang.*
input_socket = java.net.Socket('127.0.0.1', 32123);
in =ObjectInputStream(input_socket.getInputStream());
outstream = ObjectOutputStream(input_socket.getOutputStream());
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                                %Physics Simulation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i = 1:3*(num_tim_intrv)
    %read from M_C
    msg = in.readObject();
    time = msg(1);
    fG = msg(2);
    max = msg(3);
    %Write the current speed, velocity, and acceleration to the master
    %controller
    u = [xyint state t1]
    outstream.writeObject(u)
    outstream.flush()
    %Use the Runge Kutta method to calculate y'',x'',x',y'
    [tintr, xy] = ode45(@Physics_Ode, [t1 time], xyint);
    v = [ydotdot, xdotdot, fG, xyint, t1];

```

```

%Determine the State
state = State_Sequencer(v,FcG,Ttu);
%Build the data matrix that will be given to the master controller
data(i,:) = v;
%Reset the values of xyint and t1
xyint = xy(end, :);
t1 =time      ;
if time ==max
    break
end
end
% Write this to the M_C
ostream.writeObject(data)
ostream.flush()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                        %Analyze Results
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Gather information from the simulation
ydotdot = data(2:end-1, 1);
xdotdot = data(2:end-1, 2);
fG = data(2:end-1, 3);
x = data(2:end-1, 4);
y = data(2:end-1, 6);
xdot = data(2:end-1, 5);
ydot = data(2:end-1, 7);
t = data(2:end-1, 8);
%Remodel/get various calculations that correspond to the physics
v = sqrt(xdot.^2 + ydot.^2);
L = 10 * lambda * v.^2/G;
alpha = atan2(ydot, -xdot);
alphadeg = alpha * 180/pi;
alphadot = -(xdot .* ydotdot - ydot .* xdotdot) ./...
    (xdot.^2 + ydot.^2);
%vls = max(sqrt(L/10) * (Vs * m2k), Vs * m2k);
rsq = x.^2 + y.^2;
r = sqrt(rsq);
rdot = -(x .* xdot + y .* ydot) ./ r;
% Rescale for presentation
x = m2f * x;
y = m2f * y;
xdot = -m2k * xdot;
ydot = m2k * ydot;
v = m2k * v;
rdot = m2k * rdot;
f = 10 * fG/G;
alphadot = alphadot * (180/pi);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                        %Plot the figures
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
figure(1)
plot( x, y, 'linewidth', 2)
zoom on
figure(2)
plot( t, rdot,t, v, t, alphadeg, t, L, t, f, 'linewidth', 2)
legend('Cable Speed', 'Airspeed', 'Alpha', 'Lift Factor', 'Tension Factor',
'location', 'east')

```

```

    tmx =xlim;
    tmx = tmx(2);
    ylm = ylim;
    ylim([0 ylm(2)])
    zoom on
figure(3) % Plot Early Trajectory
hold on
plot(t, rdot, t, v, t, alphadeg, t, L, t, f, 'linewidth', 2)
tmx =xlim;
tmx = tmx(2);
ylm = ylim;
plot(t, y, 'b--', 'linewidth', 1.5); % Plot altitude
plot(t, alphadot, 'm--', 'linewidth', 1.5)
legend('Cable Speed', 'Airspeed', 'Alpha', 'Lift Factor', ...
    'Tension Factor', 'Altitude', 'dAlpha/dt', 'location', 'northeast')
xlim([0 14])
ylim([-1, 100.25])
hold off
figure(1)
figure(3)
figure(2)
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                                %Physics Calculutions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function output= Physics_Ode(tnxt,xy)
global u ydotdot xdotdot
%conversion factors
m2f = 3.281; % Conversion factor for meters to feet
m2k = 1.9438; % Conersion factor for m/s to knots
G = 9.81; % Accel due to gravity
Vs = 36 / m2k; % 1G stall speed
V1 = 1.3333 * Vs; % V1 Speed
Xi = 5999 / m2f; % Run Length
LoD = 20; % L/D
Fg = 1.0; % Ground run tension factor
Fc = 1.3; % Climb tension factor
Vtr = 40/m2k; % Velocity to start tapering tension
Vmx = 81/m2k; % Velocity where tension reaches 0
Ttu = 1; % Initial taper up period
tpr_g = 2; % Ground roll taper; 1 for sine, 2 for raised
cosine
tpr_r = 1; % Rotation taper; 1 for sine, 2 for raised cosine
tpr2c = 2; % Taper to Climb, 1 for sine; 2 for raised cosine
Trmp = 6; % Tension factor ramp up period
Kd = 0.08; % Derivative gain
lambda = G / V1^2; % Lambda lift factor
delta = lambda/LoD; % Delta drag factor
sc = (pi/2)/(Vmx - Vtr);
FcG = Fc * G;
FgG = Fg * G;
%define differential equations
x = xy(1);
xdot = xy(2);
y = xy(3);
ydot = xy(4);

```

```

%Do the calculations for ydotdot and xdotdot
FMA = FMA_Parachute(u, lambda, delta, LoD, G, FcG, FgG, Ttu, tpr_g, tpr_r,
tpr2c, Vtr, Trmp, sc, Kd);
ydotdot = FMA(1);
xdotdot = FMA(2);
term = FMA(3);
output = [xdot; xdotdot; ydot; ydotdot];
return
end

```

Master Controller Simulation

```

%%This file will act as the master controller. It will calculate the
%%tension as well as deliver the time in order for physics simulation to
%%continue the calculations
%%This file sends time and tension
clc clear all
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                                %Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%conversion factors
m2f = 3.281;      %   Converstion factor for meters to feet
m2k = 1.9438 ;    %   Conersion factor for m/s to knots
G = 9.81         ; %   Accel due to gravity
%Parameters
Vs = 36 / m2k;    %   1G stall speed
V1 = 1.3333 * Vs; %   V1 Speed
Xi = 5999 / m2f;  %   Run Length
LoD = 20;         %   L/D
Fg = 1.0;         %   Ground run tension factor
Fc = 1.3;         %   Climb tension factor
Vtr = 40/m2k;     %   Velocity to start tapering tension
Vmx = 81/m2k;     %   Velocity where tension reaches 0
Ttu = 1;          %   Initial taper up period
tpr_g = 2;        %   Ground roll taper; 1 for sine, 2 for raised
cosine
tpr_r = 1;        %   Rotation taper; 1 for sine, 2 for raised cosine
tpr2c = 2;        %   Taper to Climb, 1 for sine; 2 for raised cosine
Trmp = 6;         %   Tension factor ramp up period
Kd = 0.08;        %   Derivative gain
lambda = G / V1^2; %   Lambda lift factor
delta = lambda/LoD; %   Delta drag factor
sc = (pi/2)/(Vmx - Vtr);
FcG = Fc * G;
FgG = Fg * G;
tmax = 60;
fG = 0; %initialize tension
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                                %Define Socket Connection
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
import com.mathworks.jmi.*
import java.net.*
import java.io.*
import java.lang.*
providerSocket = java.net.ServerSocket(32123) ;
System.out.println('Waiting for the connection')

```

```

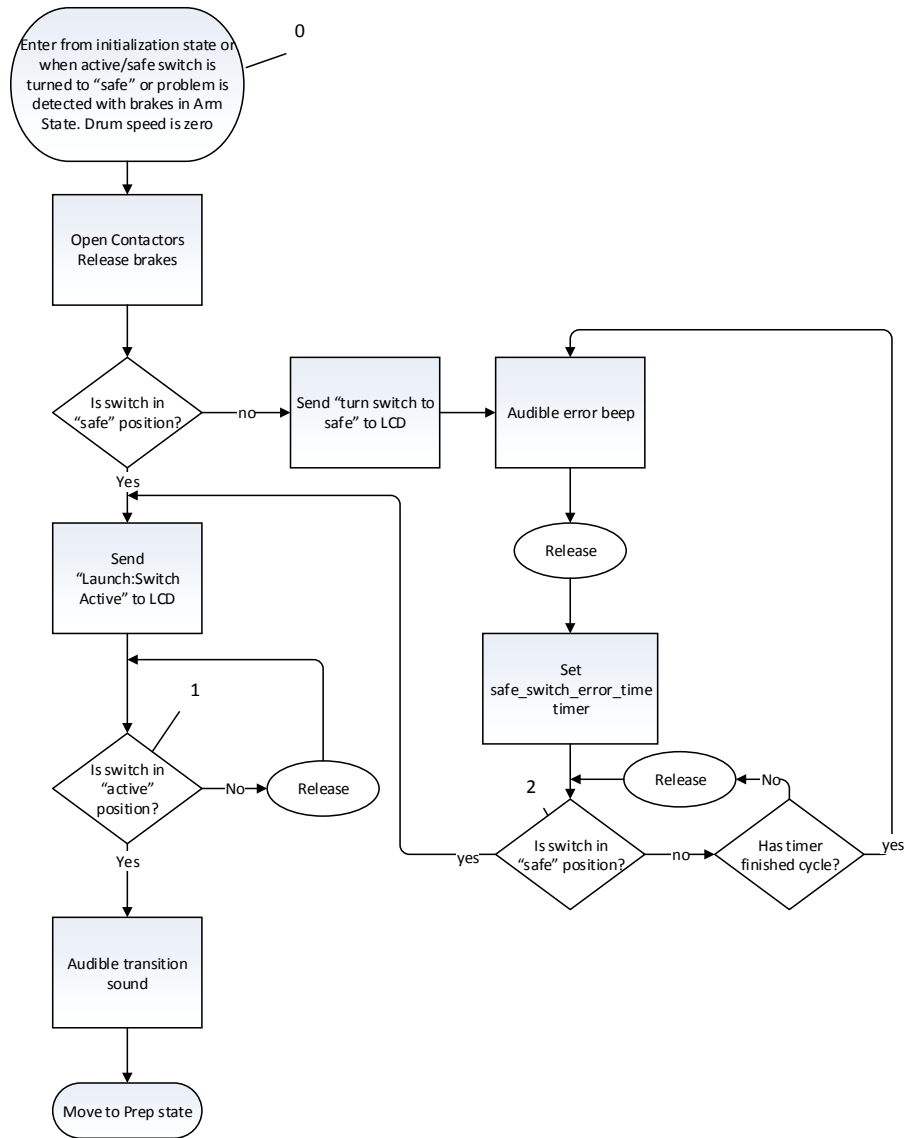
connection = providerSocket.accept;
%Sends information to Physics
output_stream = ObjectOutputStream(connection.getOutputStream);
%Read from Physics
in = ObjectInputStream(connection.getInputStream());
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for time = 1:1/64:tmax
i = [time,fG,tmax] %send over time, tension, and end time
pause(1/128)
output_stream.writeObject(i);
output_stream.flush;
% recieve x,y,xdot,ydot, state, and time
u = in.readObject();
fG = Tension_Calc(u, LoD, FcG, FgG, Ttu, tpr_g, tpr_r, tpr2c, Vtr, Trmp, sc);
end
    data = in.readObject()

```

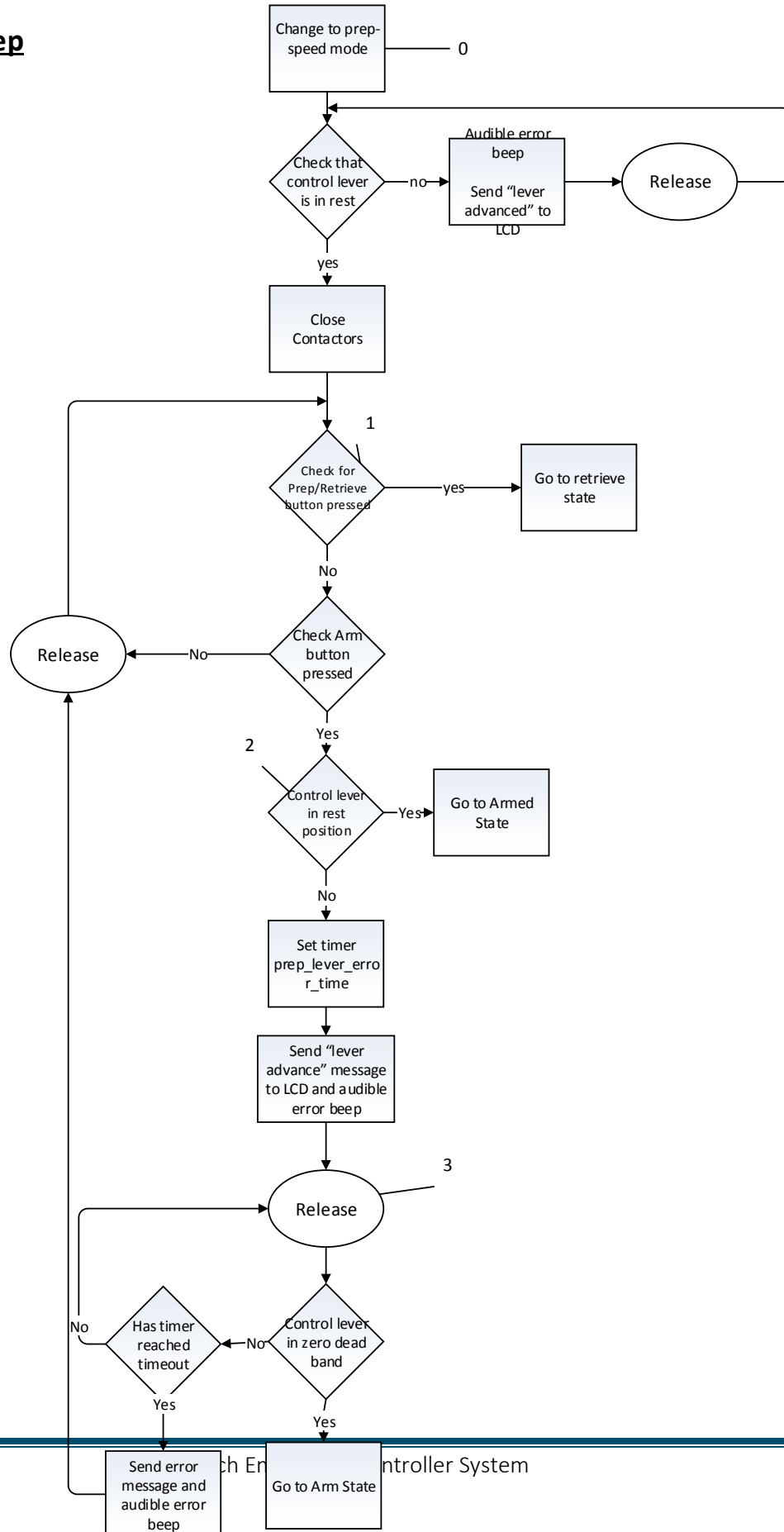
Appendix C: Embedded Code and State Machine Drawings

State Machine Drawings

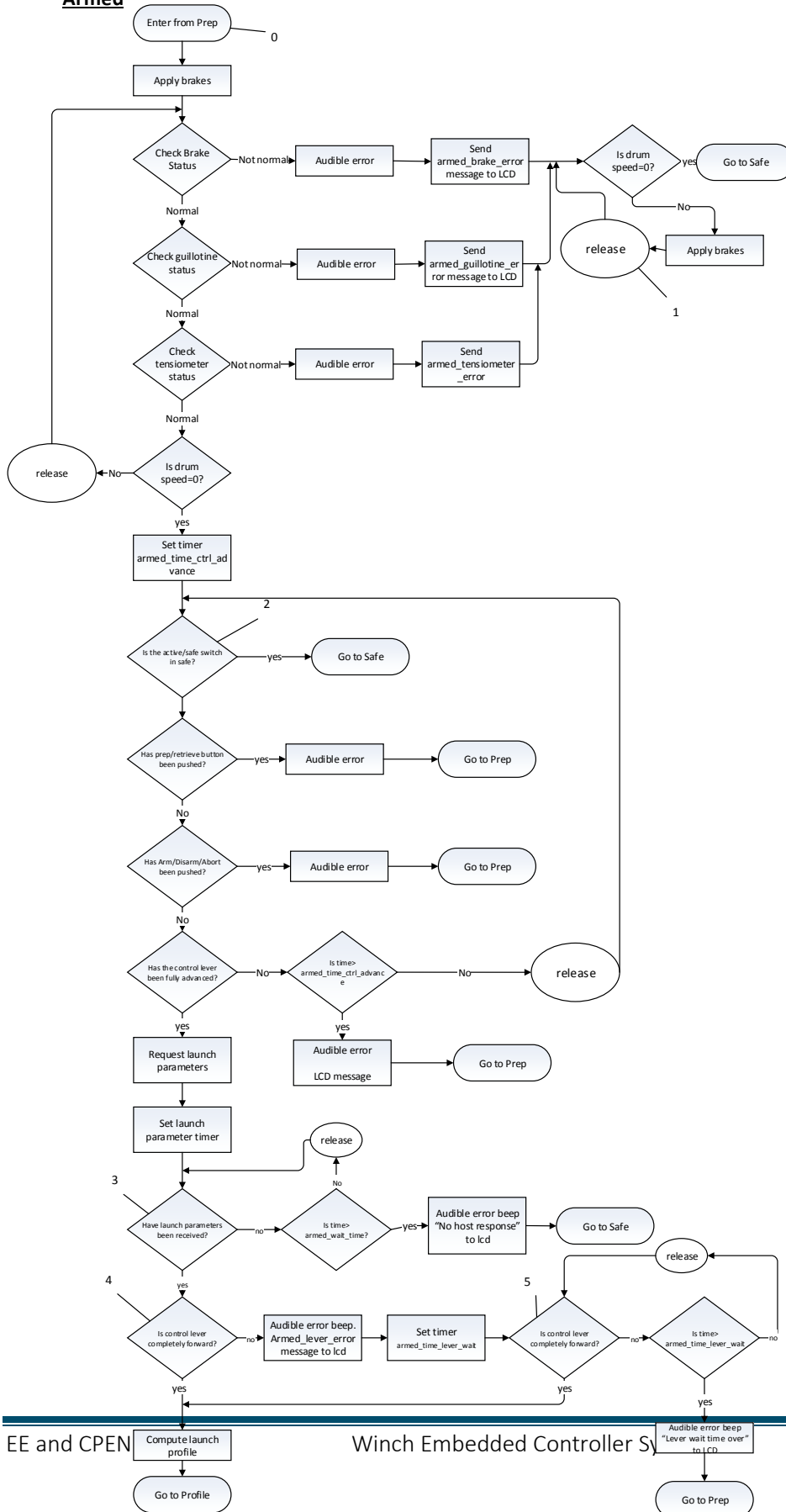
Safe



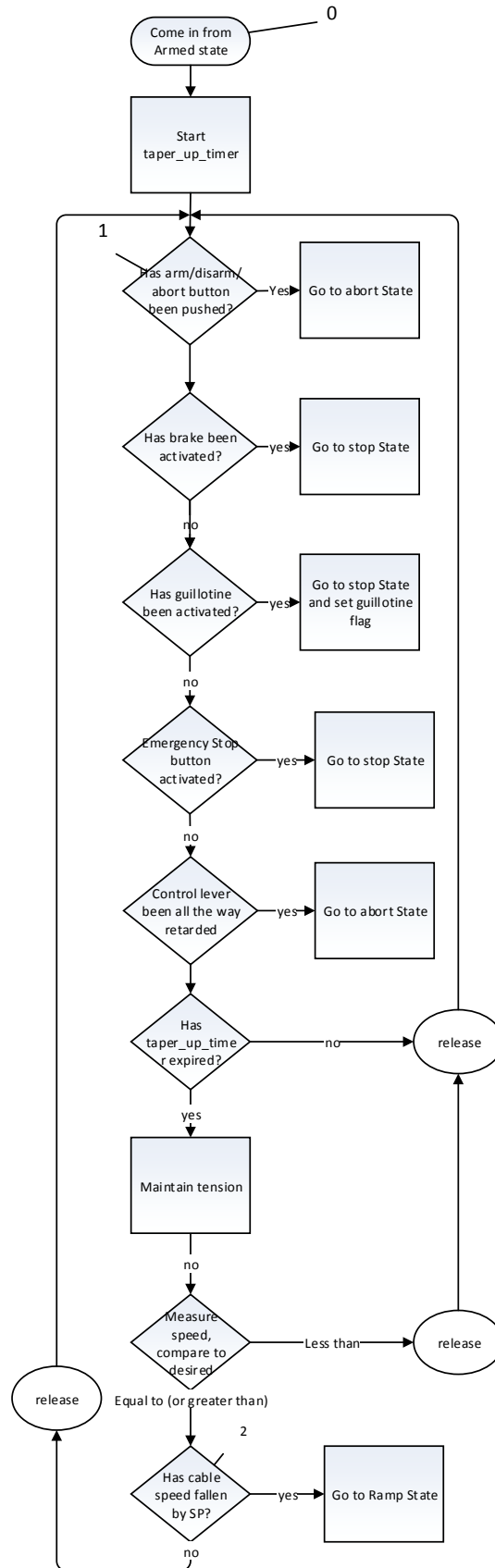
Prep



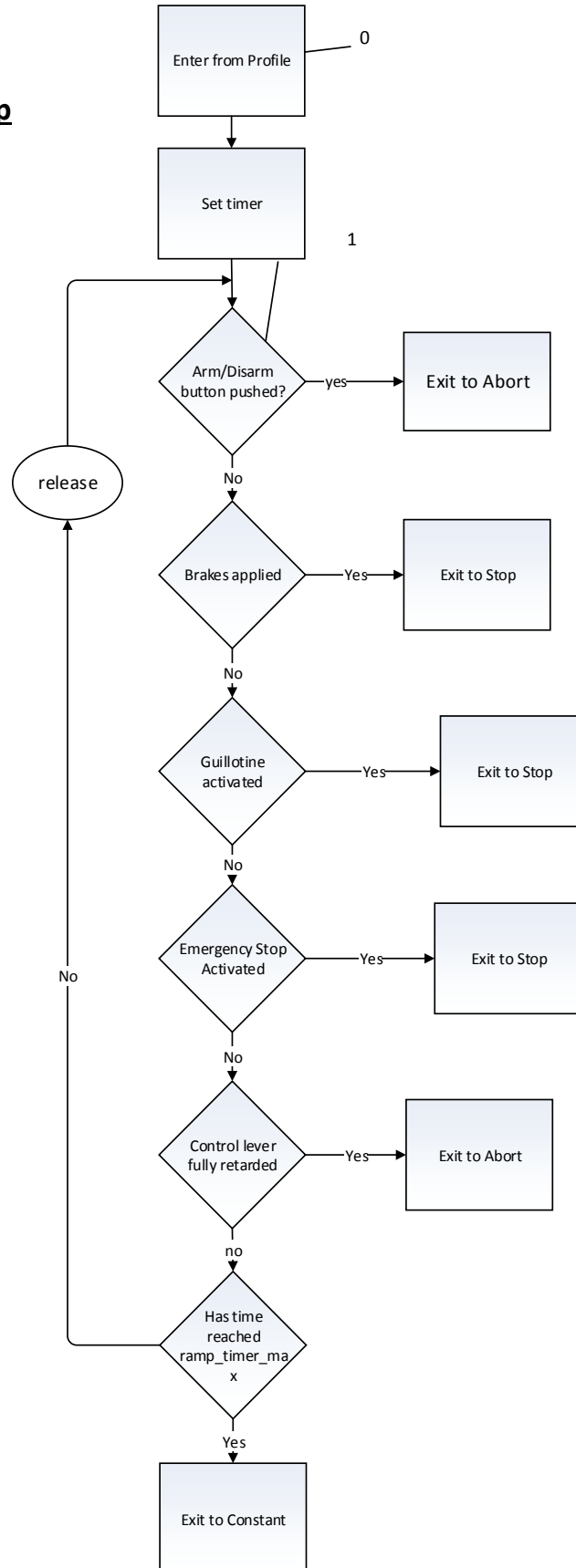
Armed



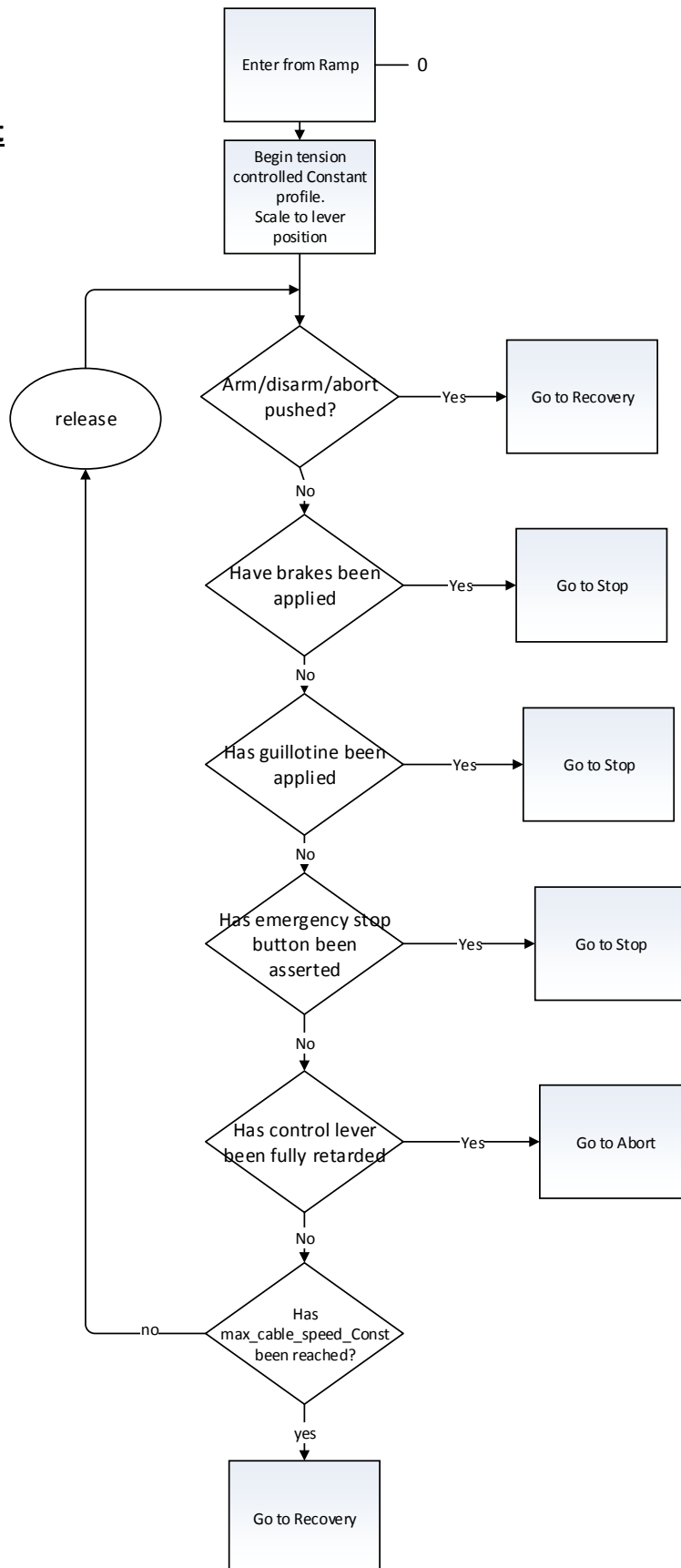
Profile



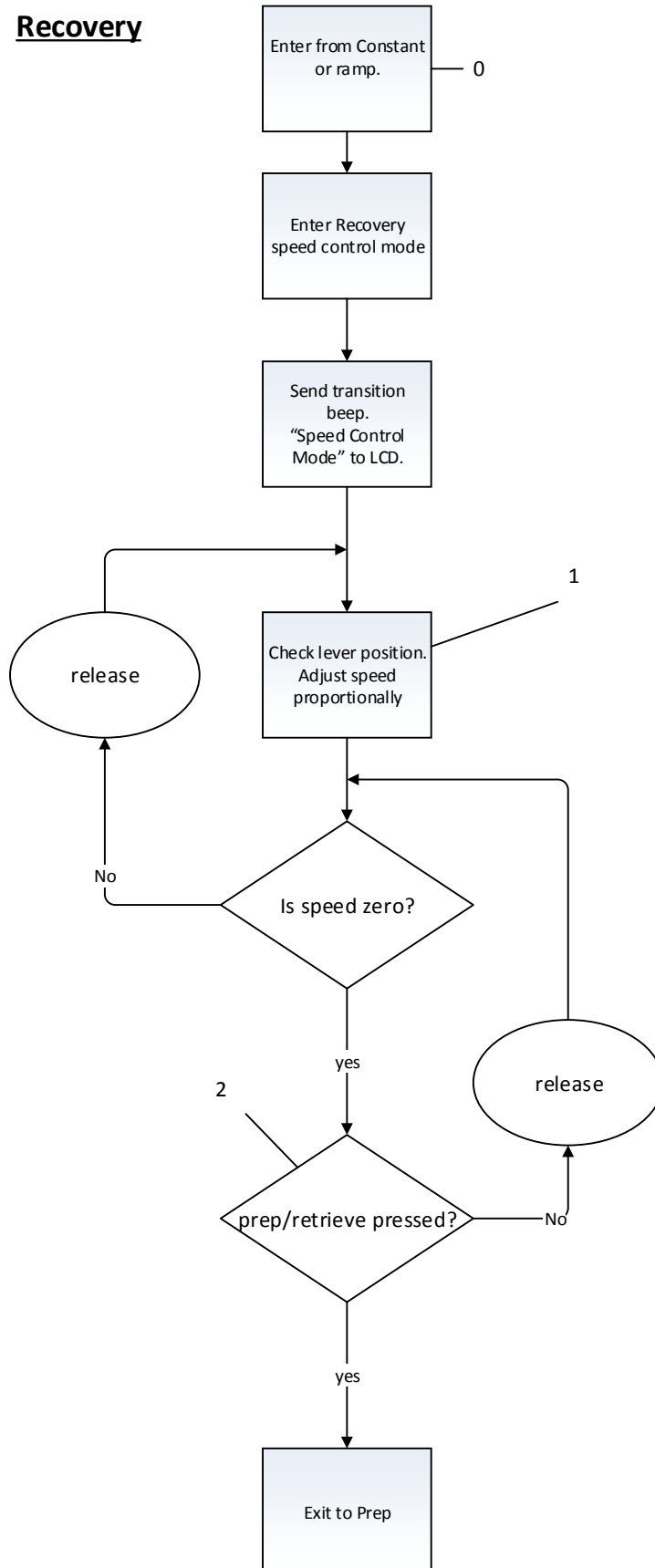
Ramp



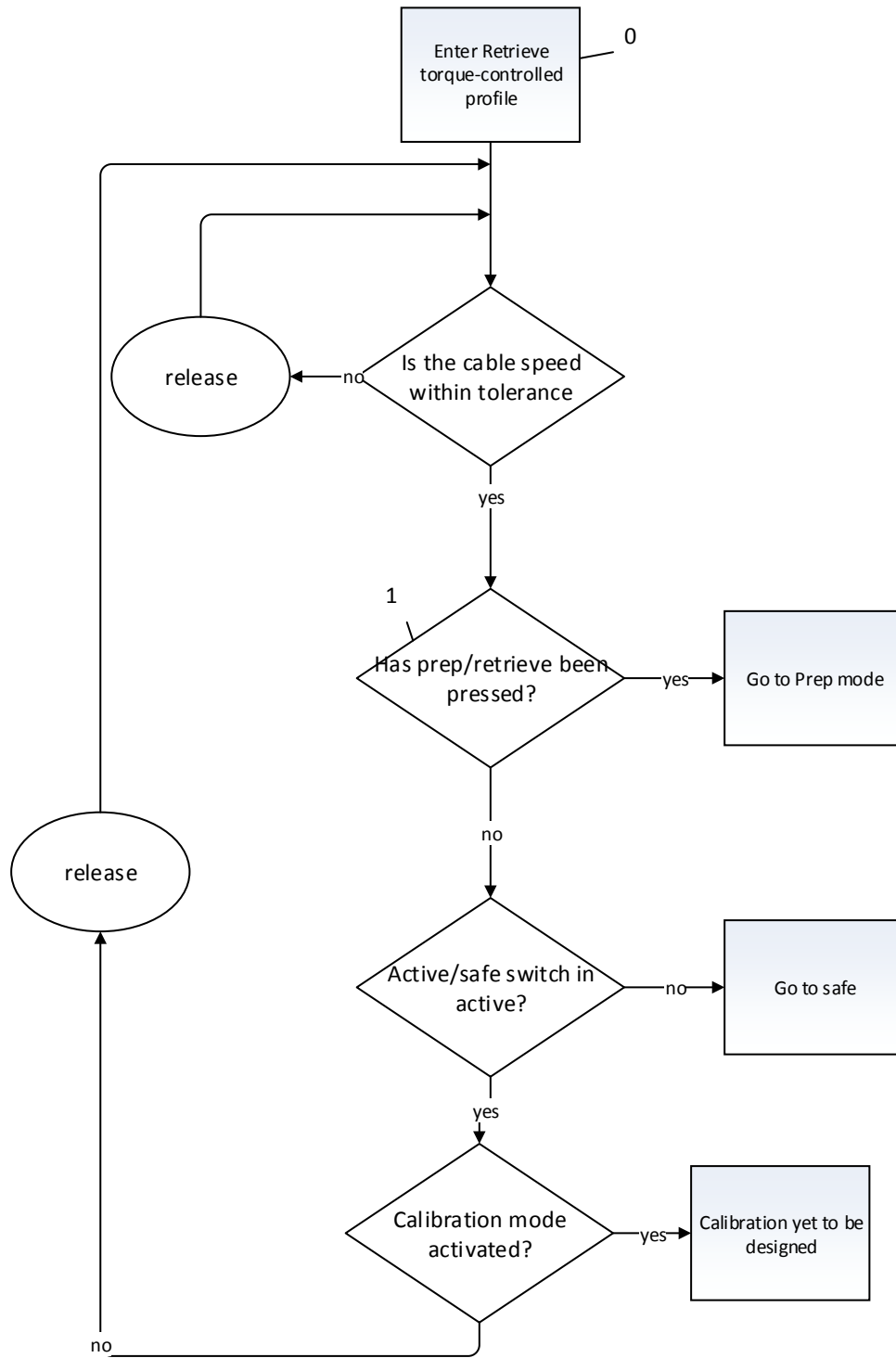
Constant



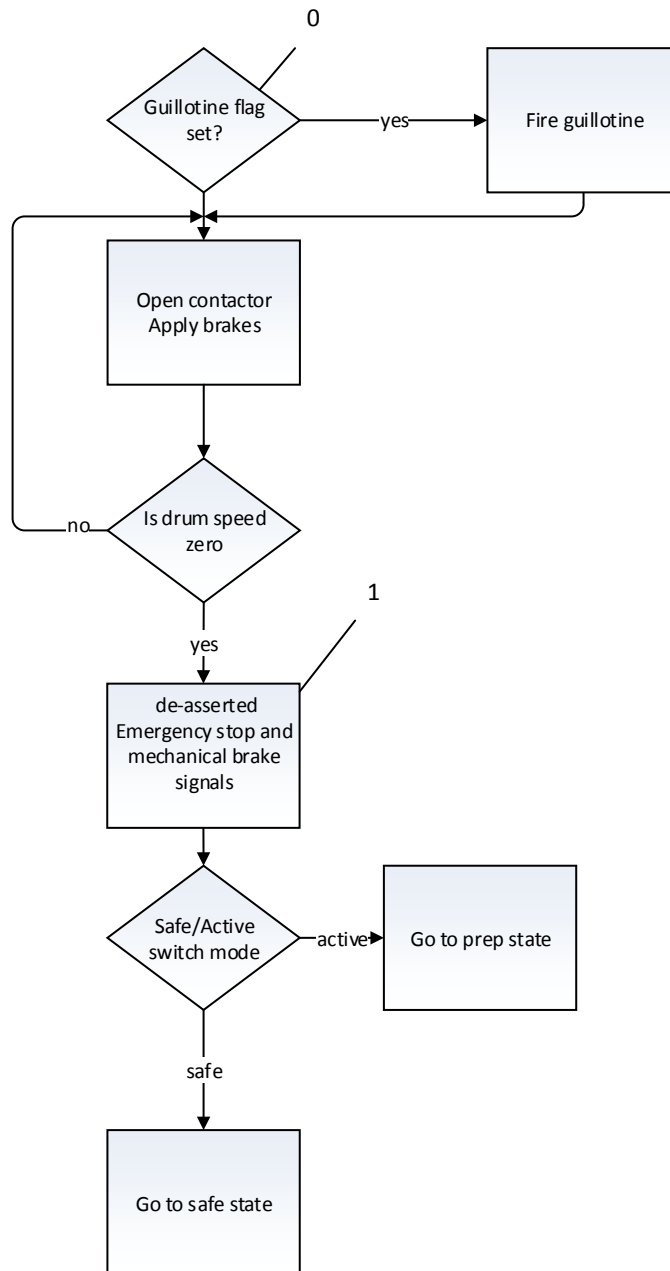
Recovery



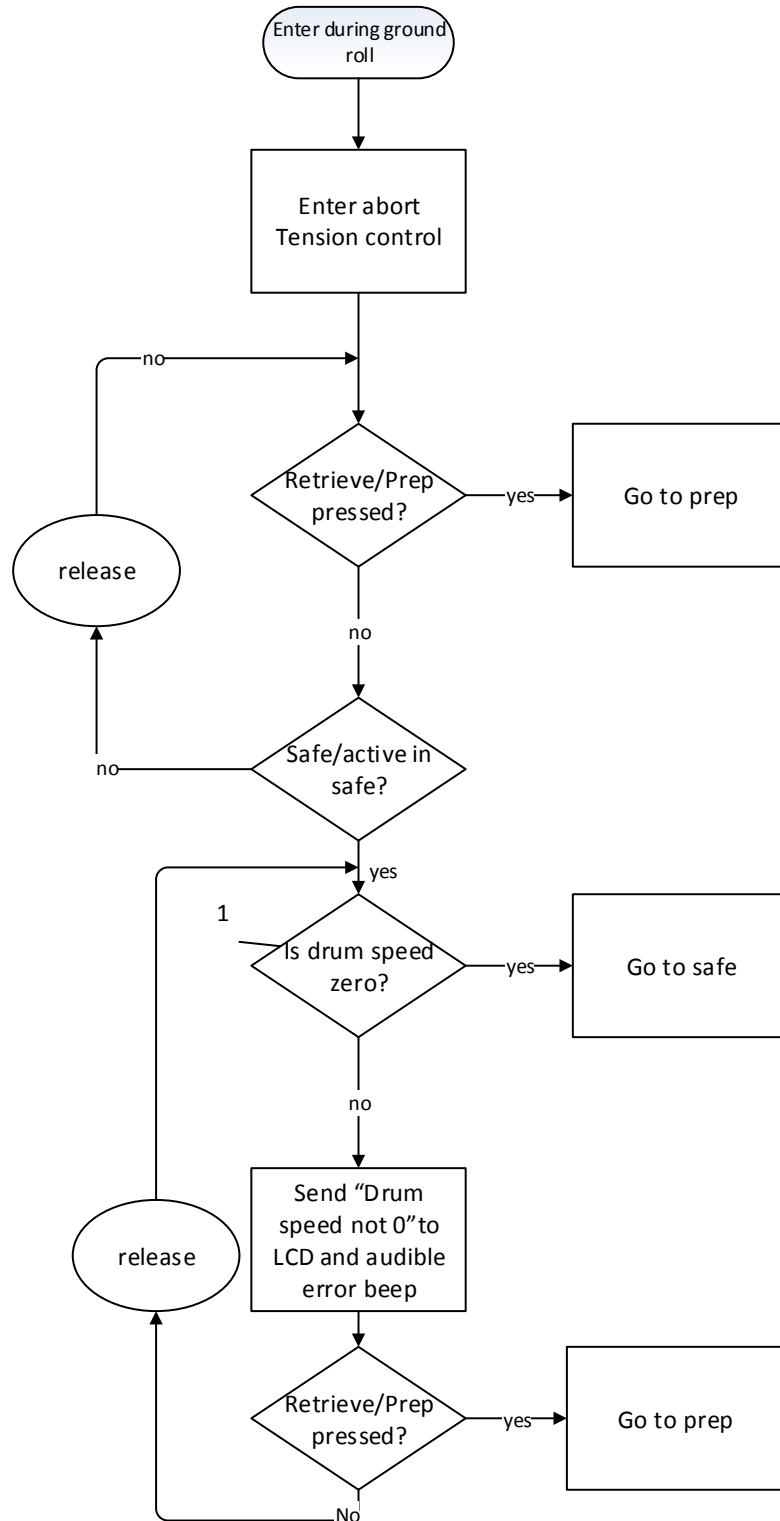
Retrieve



Stop



Abort



State Machine Code

```
// constants / system parameters
float ZERO_CABLE_SPEED_TOLERANCE = 0.1;
float ZERO_DRUM_SPEED_TOLERANCE = 0.1;
```

```

float PERCENT_PEAK_CABLE_SPEED = 0.95;
int PREP_ERROR_TIME = 186000000 * 3;
int SAFE_ERROR_TIME = 186000000 * 3;
int ARMED_CONTROL_TIME = 186000000 * 3;
int LAUNCH_PARAMETER_TIME = 186000000;
int LEVER_WAIT_TIME = 186000000 * 3;
int PROFILE_RAMP_TIME = 186000000; // 1 second
int RAMP_TIME = 186000000 * 6; // 6 seconds

// variables needed
int prepSubState = 0;
int rampSubState = 0;
int safeSubState = 0;
int stopSubState = 0;
int armedSubState = 0;
int abortSubState = 0;
int retrieveSubState = 0;
int profileSubState = 0;
int recoverySubState = 0;
int constantSubState = 0;

int brakeStatus = 0;
int guillotineStatus = 0;
int tensiometerStatus = 0;

int calibrationMode = 0;
float peakCableSpeed = 0;
float startingRampTension = 0;

int prepErrorTimer = 0;
int safeErrorTimer = 0;
int armedControlTimer = 0;
int launchParameterTimer = 0;
int leverWaitTimer = 0;
int profileRampTimer = 0;
int rampTimer = 0;

int commandedTorque = 0;
float leverPosition = 0.0; // percentage forward that the lever is

// Launch Parameters
float Fg = 1.0; // ground run tension factor
float Fc = 1.3; // climb tension factor
float G = 9.81; // Gravity?
float gliderMass = 600; // kg
float gliderWeight = gliderMass * G;
int k = 2; // tension/torque conversion
float ki = 1 / k;
int k1 = 0; // based on time of ramp up
float k2 = 0.074; // sc in matlab
float k3 = (3.14159 / (2 * RAMP_TIME));
float k4 = 3.14159 / (2 * 1); // constant for quick stop on retrieve
int MAX_CABLE_SPEED = 35;
float PROFILE_TRIGGER_CABLE_SPEED = 20.578; // 40 knots

```

```
float retrieveTension = 50; // N
```

```
// PREP STATE
```

```
void prepState () {  
    switch (prepSubState) {  
        case 0:  
            if (leverMicroSwitchBack == 1) {  
                closeContactor();  
                prepSubState = 1;  
            } else {  
                errorBeep();  
                lcdLine3 = "lever advanced";  
            }  
            break;  
        case 1:  
            if (prepButton == 1) {  
                nextState = 'retreive';  
                prepSubState = 0;  
            } else if (armButton == 1) {  
                prepSubState = 2;  
            }  
            break;  
        case 2:  
            if (leverMicroSwitchBack == 1) {  
                nextState = 'armedState';  
                prepSubState = 0;  
            } else {  
                startPrepErrorTimer();  
                errorBeep();  
                lcdLine3 = "lever advanced";  
                prepSubState = 3;  
            }  
            break;  
        case 3:  
            if (leverMicroSwitchBack == 1) {  
                nextState = 'armedState';  
                prepSubState = 0;  
            } else if (((int)*(volatile unsigned int *)0xE0001004 -  
prepErrorTimer)) > PREP_ERROR_TIME) {  
                errorBeep();  
                lcdLine3 = "Prep lever timeout";  
                prepSubState = 1;  
            }  
            break;  
    }  
}
```

```
// retrieve state
```

```
void retrieveState () {  
    switch (retrieveSubState) {  
        case 0:  
            if (abs(cableSpeed) < ZERO_CABLE_SPEED_TOLERANCE) {
```

```

        retrieveSubState = 1;
    }
    break;
case 1:
    if (prepButton == 1) {
        nextState = "prep";
        retrieveSubState = 0;
    } else if (safeSwitch == 0) {
        nextState = "safe";
        retrieveSubState = 0;
    } else if (calibrationMode == 1) {
        nextState = "calibration";
        retrieveSubState = 0;
    }
    break;
}
}

// safe state
void safeState () {
    switch (safeSubState) {
        case 0:
            openContactor();
            releaseBrakes();
            if (safeSwitch == 1) {
                lcdLine3 = "Launch:Switch Active";
                safeSubState = 1;
            } else {
                lcdLine3 = "Turn switch to safe";
                errorBeep();
                startSafeErrorTimer();
                safeSubState = 2;
            }
            break;
        case 1:
            if (safeSwitch == 0) {
                transitionBeep();
                nextState = "prep";
                safeSubState = 0;
            }
            break;
        case 2:
            if (safeSwitch == 1) {
                safeSubState = 1;
            } else if (((int)*(volatile unsigned int *)0xE0001004 -
safeErrorTimer)) > SAFE_ERROR_TIME) {
                errorBeep();
                startSafeErrorTimer();
            }
            break;
    }
}

// abort state

```

```

void abortState () {
    switch (abortSubState) {
        case 0:
            // enter abort tension control
            if (prepButton == 1) {
                nextState = "prep";
                abortSubState = 0;
            } else {
                if (safeSwitch == 1) {
                    abortSubState = 1;
                }
            }
            break;
        case 1:
            if (abs(drumSpeed) < ZERO_DRUM_SPEED_TOLERANCE) {
                nextState = "safe";
                abortSubState = 0;
            } else {
                errorBeep();
                lcdLine3 = "Drum speed not 0";
                if (prepButton == 1) {
                    nextState = "prep";
                    abortSubState = 0;
                }
            }
            break;
    }
}

// constant state
void constantState () {
    switch (constantSubState) {
        case 0:
            if (armButton == 1) {
                nextState = "abort";
                profileSubState = 0;
            }
            if (brakeButton == 1) {
                nextState = "stop";
                profileSubState = 0;
            }
            if (guillotineButton == 1) {
                nextState = "stop";
                profileSubState = 0;
            }
            if (emergencyStopButton == 1) {
                nextState = "stop";
                profileSubState = 0;
            }
            if (leverMicroSwitchBack == 1) {
                nextState = "abort";
                profileSubState = 0;
            }
    }
}

```

```

        if (((int)(*(volatile unsigned int *)0xE0001004 - cableSpeed)) >=
MAX_CABLE_SPEED) {
            nextState = "recovery";
            constantSubState = 0;
        }
        break;
    }
}

// recovery state
void recoveryState () {
    switch (recoverySubState) {
        case 0:
            transitionBeep();
            lcdLine3 = "unknown message";
            recoverySubState = 1;
            break;
        case 1:
            if (abs(cableSpeed) < ZERO_CABLE_SPEED_TOLERANCE) { // waiting
for speed to be 0
                recoverySubState = 2;
            }
            break;
        case 2:
            if (prepButton == 1) {
                nextState = "prep";
                recoverySubState = 0;
            }
            break;
    }
}

// ramp state
void rampState () {
    switch (rampSubState) {
        case 0:
            startingRampTension = tension;
            startRampTimer();
            rampSubState = 1;
            break;
        case 1:
            if (armButton == 1) {
                nextState = "abort";
                profileSubState = 0;
            }
            if (brakeButton == 1) {
                nextState = "stop";
                profileSubState = 0;
            }
            if (guillotineButton == 1) {
                nextState = "stop";
                profileSubState = 0;
            }
            if (emergencyStopButton == 1) {

```

```

        nextState = "stop";
        profileSubState = 0;
    }
    if (leverMicroSwitchBack == 1) {
        nextState = "abort";
        profileSubState = 0;
    }
    if (((int)(*(volatile unsigned int *)0xE0001004 - rampTimer)) >
RAMP_TIME) {
        nextState = 'constant';
        rampSubState = 0;
    }
    break;
}
}

// stop state
void stopState () {
    switch (stopSubState) {
        case 0:
            if (guillotineButton == 1) {
                fireGuillotine();
            }
            openContactor();
            applyBrakes();
            if (abs(drumSpeed) < ZERO_DRUM_SPEED_TOLERANCE) {
                stopSubState = 1;
            }
            break;
        case 1:
            releaseBrakes();
            if (safeSwitch == 0) {
                nextState = "prep";
                stopSubState = 0;
            } else {
                nextState = "safe";
                stopSubState = 0;
            }
            break;
    }
}

// profile state
void profileState () {
    switch (profileSubState) {
        case 0:
            startProfileRampTimer();
            profileSubState = 1;
            break;
        case 1:
            if (armButton == 1) {
                nextState = "abort";
                profileSubState = 0;
            }
    }
}

```



```

        if (brakeButton == 1) {
            nextState = "stop";
            profileSubState = 0;
        }
        if (guillotineButton == 1) {
            nextState = "stop";
            profileSubState = 0;
        }
        if (emergencyStopButton == 1) {
            nextState = "stop";
            profileSubState = 0;
        }
        if (leverMicroSwitchBack == 1) {
            nextState = "abort";
            profileSubState = 0;
        }
        if (((int)(*(volatile unsigned int *)0xE0001004 -
profileRampTimer)) > PROFILE_RAMP_TIME) {
            profileSubState = 2;
        }
        break;
    case 2:
        // find the max cable speed
        if (cableSpeed > peakCableSpeed) {
            peakCableSpeed = cableSpeed;
        }
        if (cableSpeed < peakCableSpeed * PERCENT_PEAK_CABLE_SPEED) {
            nextState = 'ramp';
            profileSubState = 0;
        }
        break;
    }
}

// armed state
void armedState () {
    switch (armedSubState) {
        case 0: // check to make sure everything is working ok
            applyBrakes();
            if (brakeStatus == 1) { // not normal
                errorBeep();
                lcdLine3 = "Brake Error!";
                armedSubState = 1;
            } else if (guillotineStatus == 1) { // not normal
                errorBeep();
                lcdLine3 = "Guillotine Error!";
                armedSubState = 1;
            } else if (tensiometerStatus == 1) { // not normal
                errorBeep();
                lcdLine3 = "Tensiometer Error!";
                armedSubState = 1;
            } else if (abs(drumSpeed) < ZERO_DRUM_SPEED_TOLERANCE) {
                startArmedControlTimer();
                armedSubState = 2;
            }
    }
}

```

```

    }
    break;
case 1:
    if (abs(drumSpeed) < ZERO_DRUM_SPEED_TOLERANCE) {
        nextState = "safe";
        armedSubState = 0;
    } else {
        applyBrakes();
    }
    break;
case 2:
    if (safeSwitch == 1) { // in safe position
        nextState = "safe";
        armedSubState = 0;
    } else if (prepButton == 1) {
        errorBeep();
        nextState = "prep";
        armedSubState = 0;
    } else if (armButton == 1) {
        errorBeep();
        nextState = "prep";
        armedSubState = 0;
    } else if (leverMicroSwitchFront == 1) {
        requestLaunchParameters();
        startLaunchParameterTimer();
        armedSubState = 3;
    } else if (((int)*(volatile unsigned int *)0xE0001004 -
armedControlTimer)) > ARMED_CONTROL_TIME) {
        errorBeep();
        lcdLine3 = "Armed time error";
        nextState = "prep";
        armedSubState = 0;
    }
    break;
case 3:
    if (launchParametersComplete()) {
        armedSubState = 4;
    } else if (((int)*(volatile unsigned int *)0xE0001004 -
launchParameterTimer)) > LAUNCH_PARAMETER_TIME) {
        errorBeep();
        lcdLine3 = "No host response";
        nextState = "safe";
        armedSubState = 0;
    }
    break;
case 4:
    if (leverMicroSwitchFront == 1) {
        computeLaunchProfile();
        nextState = "profile";
        armedSubState = 0;
    } else {
        errorBeep();
        lcdLine3 = "Advance Ctrl Lever";
        startLeverWaitTimer();
    }

```

```

        armedSubState = 5;
    }
    break;
case 5:
    if (leverMicroSwitchFront == 1) {
        computeLaunchProfile();
        nextState = "profile";
        armedSubState = 0;
    } else if (((int)(*(volatile unsigned int *)0xE0001004 -
leverWaitTimer)) > LEVER_WAIT_TIME) {
        errorBeep();
        lcdLine3 = "Lever wait time over";
        nextState = "prep";
        armedSubState = 0;
    }
    break;
}
}
}

```

Control Law Code

```

void controllaw () {
    if (currentState == 'retrieve') {
        if (cableSpeed <= 0) {
            commandedTorque = retrieveTension * ki;
        } else {
            commandedTorque = ki * retrieveTension * cos(k4 * cableSpeed);
        }

    } else if (currentState == 'profile') {
        if (profileSubState == 1) { // ramping
            commandedTorque = leverPosition * ( (Fg * gliderWeight) / (2 * k)
) * (1 - cos(2 * k1 * profileRampTimer));
        } else if (profileSubState == 2) {
            if (cableSpeed < PROFILE_TRIGGER_CABLE_SPEED) {
                commandedTorque = leverPosition * Fg * gliderWeight * ki;
            } else {
                commandedTorque = leverPosition * Fg * gliderWeight * ki *
cos(k2 * (cableSpeed - PROFILE_TRIGGER_CABLE_SPEED));
            }
        }

    } else if (currentState = 'ramp') {
        commandedTorque = startingRampTension + (((Fc * gliderWeight * G) -
startingRampTension) * sin(k3 * rampTimer));
        commandedTorque = commandedTorque * leverPosition * ki; // scale by
lever position

    } else if (currentState = 'constant') {
        commandedTorque = leverPosition * Fc * gliderWeight * G * ki;

    } else if (currentState = 'recovery') {
        commandedTorque = leverPosition * Fg * gliderWeight * ki * cos(k2 *
(cableSpeed - PROFILE_TRIGGER_CABLE_SPEED));
    }
}

```

```

        } else if (currentState = 'stop') {
            commandedTorque = -10000 * ki;
        }
    }

// Other functions

void errorBeep () { // double beep
}

void transitionBeep () { // single beep
}

void closeContactor () {
    struct CANRCVBUF can;
    int tmp;

    can.id      = 0x23000000;
    can.dlc     = 0x00000001;
    can.cd.us[0] = 0xFF; // close it

    tmp = CAN_gateway_send(&can);
    canbuf_add(&can);
}

void openContactor () {
    struct CANRCVBUF can;
    int tmp;

    can.id      = 0x23000000;
    can.dlc     = 0x00000001;
    can.cd.us[0] = 0x00; // open it

    tmp = CAN_gateway_send(&can);
    canbuf_add(&can);
}

void releaseBrakes () {
    struct CANRCVBUF can;
    int tmp;

    can.id      = 0x21000000;
    can.dlc     = 0x00000001;
    can.cd.us[0] = 0x00; // release

    tmp = CAN_gateway_send(&can);
    canbuf_add(&can);
}

void applyBrakes () {
    struct CANRCVBUF can;
    int tmp;

    can.id      = 0x21000000;

```

```

        can.dlc      = 0x00000001;
        can.cd.us[0] = 0xFF; // apply

        tmp = CAN_gateway_send(&can);
        canbuf_add(&can);
    }

void fireGuillotine () {
    struct CANRCVBUF can;
    int tmp;

    can.id      = 0x22000000;
    can.dlc     = 0x00000001;
    can.cd.us[0] = 0xFF; // fire

    tmp = CAN_gateway_send(&can);
    canbuf_add(&can);
}

void requestLaunchParameters () {
    struct CANRCVBUF can;
    int tmp;

    can.id      = 0x27000000;
    can.dlc     = 0x00000001;
    can.cd.us[0] = 0x00; // nothing

    tmp = CAN_gateway_send(&can);
    canbuf_add(&can);
}

bool launchParametersComplete () {
    // Placeholder, eventually this will check to see if all launch parameters
    // have been recieved from the host controller.

    // For now, just return true.
    return true;
}

void computeLaunchProfile () {
    // assign launch parameters from messages
    // nothing in our system
}

// timer functions

void startLeverWaitTimer () {
    leverWaitTimer = ((int)(*(volatile unsigned int *)0xE0001004));
}

void startLaunchParameterTimer () {
    launchParameterTimer = ((int)(*(volatile unsigned int *)0xE0001004));
}

```

```
void startArmedControlTimer () {  
    armedControlTimer = ((int)(*(volatile unsigned int *)0xE0001004));  
}  
  
void startSafeErrorTimer () {  
    safeErrorTimer = ((int)(*(volatile unsigned int *)0xE0001004));  
}  
  
void startPrepErrorTimer () {  
    prepErrorTimer = ((int)(*(volatile unsigned int *)0xE0001004));  
}  
  
void startProfileRampTimer () {  
    profileRampTimer = ((int)(*(volatile unsigned int *)0xE0001004));  
}  
  
void startRampTimer () {  
    rampTimer = ((int)(*(volatile unsigned int *)0xE0001004));  
}
```

F4 IO pins

Use	Ports	Our Use Function	Board Function 1	P1	P2
Input Switches and Output LEDs (GP and SPI)	PB12	SPI2_NSS	Free I/O	36	
	PB13	SPI2_SCK/	Free I/O	37	
	PB14	SPI2_MISO/	Free I/O	38	
	PB15	SPI2_MOSI	Free I/O	39	
ADCs (3 allocated)	PC1	ADC123_IN11	Free I/O	7	
	PC2	ADC123_IN12	Free I/O	10	
	PC4	ADC123_IN14	Free I/O	20	
CANbus	PD0	CAN1_RX	Free I/O		36
	PD1	CAN1_TX	Free I/O		33
UART (Bootloader)	PA9	USART1_TX	VBUS_FS		44
	PA10	USART1_RX	OTG_FS_ID		41
UART (GPS)	PA2	USART2_TX	Free I/O	14	
	PA3	USART2_RX	Free I/O	13	
UART (LCD)	PD8	USART3_TX	Free I/O	40	
	PD9	USART3_RX	Free I/O	41	
GPS 1PPS	PA1	TIM5_CH2	Free I/O	35	
Beeper	PB8	TIM10_CH1	Free I/O		43
PWM (Contactor)	PE5	TIM9_CH1	Free I/O		14
PWM (Panel Dim)	PE6	TIM9_CH2	Free I/O		11
Audio (I2S and SPI)	PB6		Audio_SCL		
	PB9		Audio_DSA		
	PC7		I2S3_SCK		
	PC10		IS23_SCK		
	PC12		IS23_SD		
	PA4		I2S3_WA		
	PD4		Audio_RST		
	PC3		PDM_OUT		
Accelerometers	PB10		CLK_IN		
	PA7		SPI1_MOSI		
	PA5		SPI1_SCK		
	PE0		MEMS_INT1		
	PE1		MEMS_INT2		
	PE3		CS_I2C/SPI		
USB_OTG	PA6		SPI1_MOSI		
	PC0		OTG_FS_PowerSwitchOn		
	PA9		VBUS_FS		
	PA10		OTG_FS_ID		
	PA11		OTG_FS_DM		
	PA12		OTG_FS_DP		
LED (Green)	PD5		OTG_FS_PverCurrent		
	PD12		GPIO		
	PD13		GPIO		
	PD14		GPIO		
LED (Orange)	PD15		GPIO		
LED (Red)					
LED (Blue)					
Reset PB	NRST		NRST		
User PB	PA0		GPIO		

Appendix D: Control Panel and Parts



Figure 12: Built Control Panel

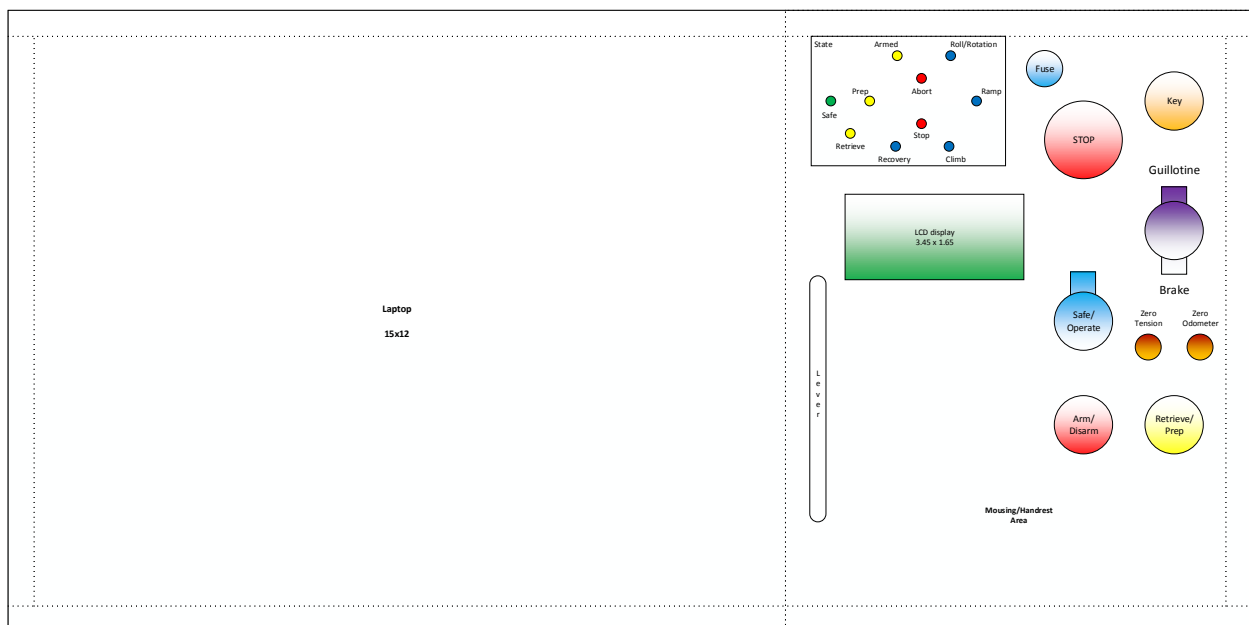


Figure 13: Final Control Panel Design

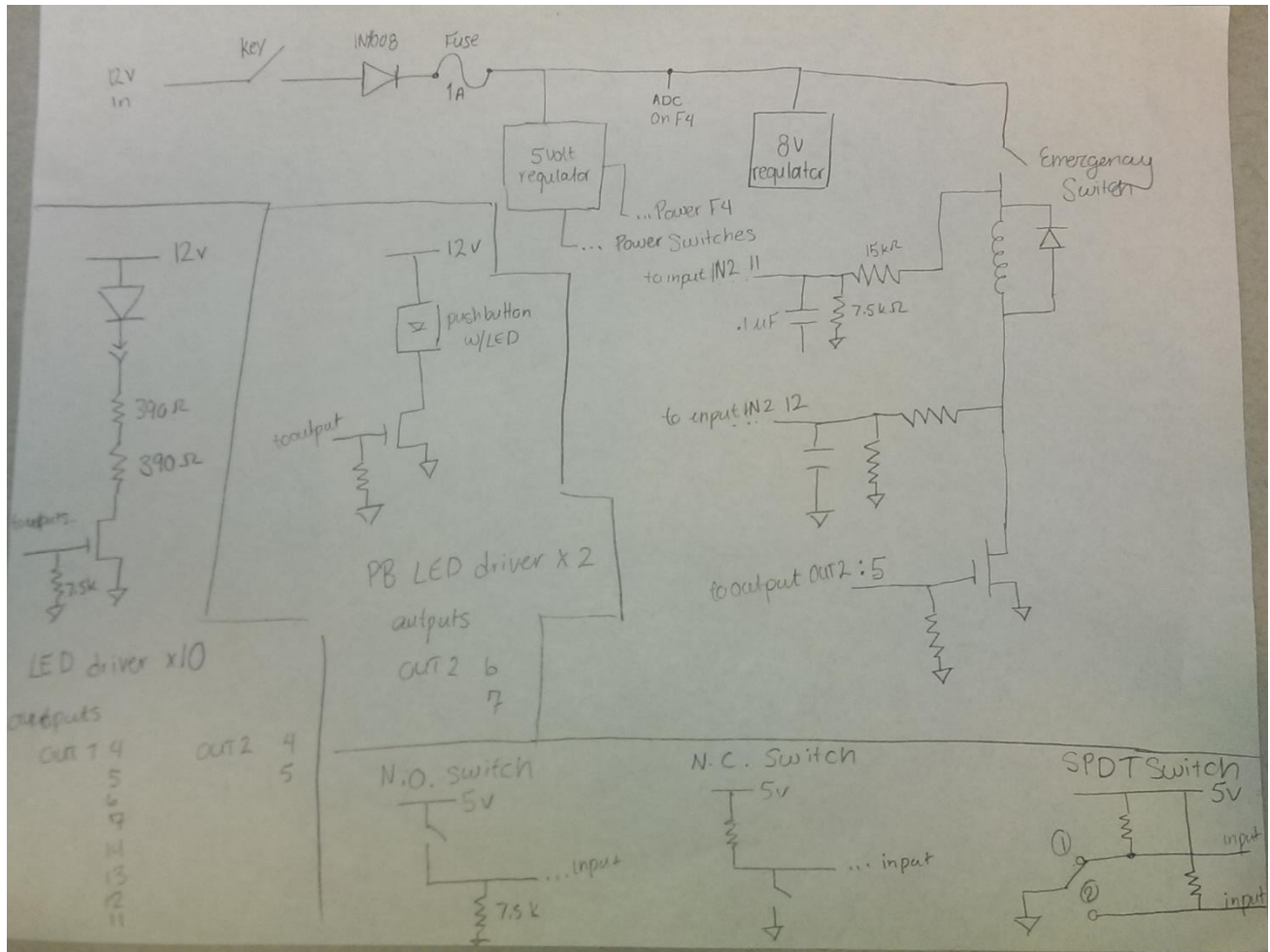


Figure 14: Connection to 12V and other multi-use components

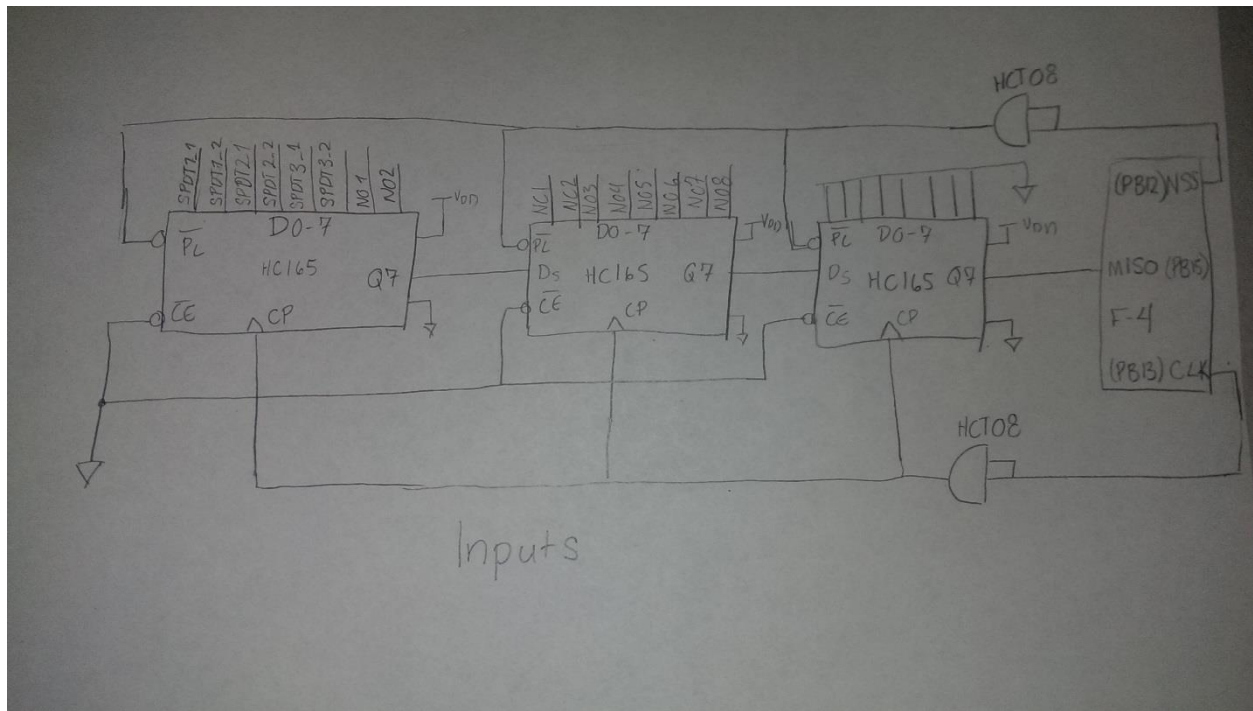


Figure 15: Connections to HC165 Shift Register- Input to F4

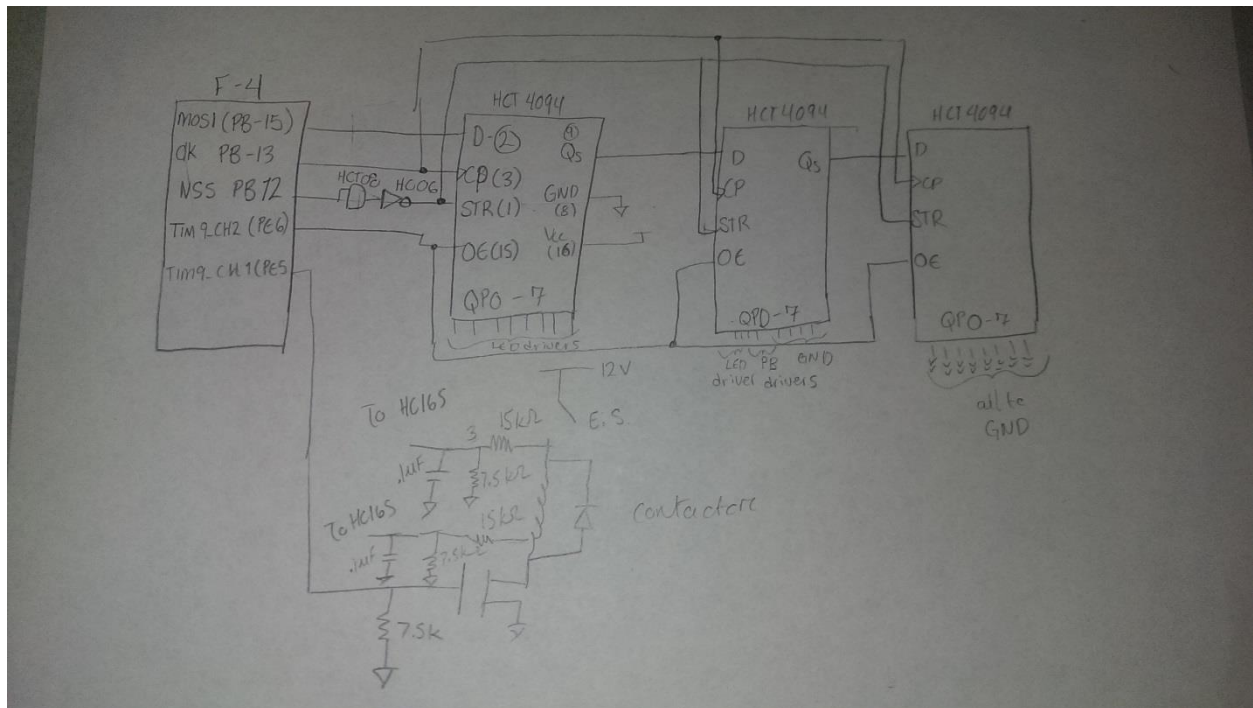


Figure 16: Connection to HCT4094 Shift Register- Outputs from F4

References:

Moore, George. *Battery-Electric Winch Automation High-level Design*. Spokane.
6 Nov 2013. PDF.

STMicroelectronics. *UM1472 User Manual*. January 2012.