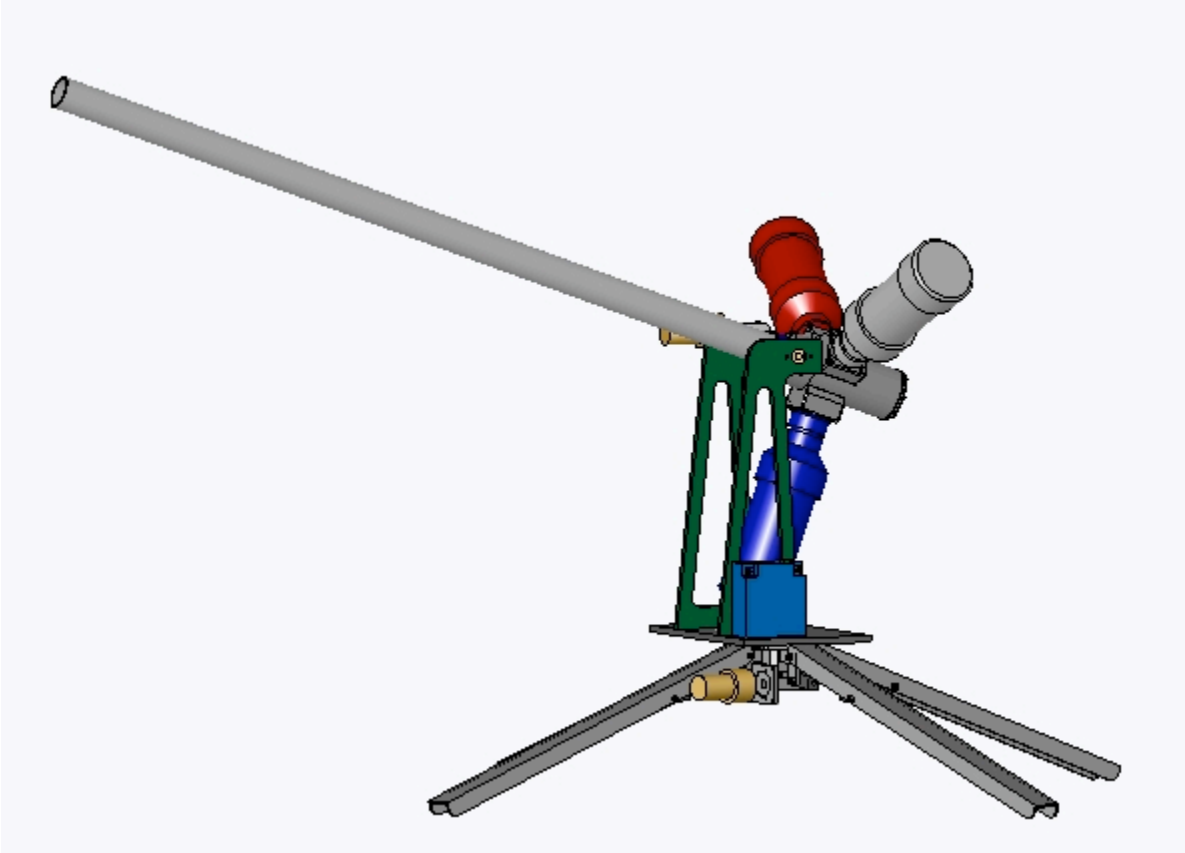
Targeting Control for Projectile Launcher



Kyle Brookes

Ted Fernando

Michael Kasmin

Stephen Ludwig

ECSE-4790: Microprocessor Systems

5 December 2002

Table of Contents

List of Figures

List of Tables

Abstract

Motion control systems are widely prevalent in many of today's engineered systems, from mailing machines to avionics to home automation. Most systems utilize independent microprocessor-based controllers to meet a variety of motion applications ranging form motor speed control to end-effecter position manipulation. Microcontrollers, such as the MC68HC812A4, while not specifically suited for motion control applications, have many features possessed by their widely used counterparts. A wide variety of control schemes can be implemented, such as discrete on/off and analog control. Analog and digital sensors provide feedback to the controller to facilitate the loop closing of feedback control. One of the most popular forms of control is PWM control of AC and DC motors. This control scheme combines the relative simplicity of on/off control with the accuracy of analog control.

This project focuses on the application of PWM feedback control applied to a two-axis 3D pointer system. The heart of the motion control is a PWM controller utilizing a PID control algorithm to feed two full H bridge circuits. These, in turn, will drive two brushed DC motors. The two-axis arrangement follows an azimuth/elevation scheme where the horizontal axis is stacked on the vertical axis to achieve a full half sphere of movement. The control system consists of a master controller and a slave controller. The master controller will control the entire system; more specifically, the actual movement of the motors. The slave controller will monitor the motor encoders to provide motor control. When the master processor needs motor data, it will communicate with the slave.

Introduction

Many of today's motion control systems rely heavily on the use of a wide array of microcontrollers. For our project, we attempted to tackle the task of motion control of a projectile launcher primarily using the 68HC908 microcontroller. In the end, our implementation included many subsystems which are heavily interdependent. There are five main components which comprise the entire system: a system control module to control the overall system and the firing of the launcher; a motor driver module to accurately control the movement of the various motors; an encoder module to provide feedback to the system so it can accurately position itself; a communication module which provides a protocol in which to encode all the necessary firing data; and a motor control module which runs a PID control scheme to control the two motors. The purpose of the software-based System Control Module is to handle and control the various subsystems. It ensures that each module operates correctly both independently and also as part of the whole system. Much of the System Control Module was based in LabVIEW, which provides much of the backend calculations and GUI interface for the system. The Communication Interface defines a protocol that is used throughout the system. The Motor Control Module is responsible for the physical manipulation of the gun. The hardware H-bridge circuit of the Motor Driver Module provides the power to drive the motors that move the launcher along the theta and y-axis. It can be seen how the Motor Driver Module is heavily dependent on the Encoder Module for proper operation. The Encoder Module provides the feedback for the Motor Driver Module. The Encoder Module essentially interprets actual motor control feedback and adjusts the system accordingly. The final components of our system are the various interface boards. These interface boards provide for physical connections between the various modules and systems. The interface boards allow for simple and tidy connections between our various subcomponents. By using standard interface boards, we were also able to easily debug and replace each component independently.

Materials & Methods

As mentioned above in the Introduction, the entire system is made up of many different modules. Each of these modules will be discussed separately.

Motor Driver Module

The Motor Driver Module is centered on two National Instruments LMD18200 H-Bridge Integrated Circuits. The LMD18200 is a 3A H-Bridge designed for motion control applications. The device accommodates peak output currents of 6A and continuous currents of 3A for each motor. Each chip also has its own individual high current motor power connection. The supply power is typically around 12VDC; however, it can be increased up to a maximum of 60VDC.

The control signals that feed the two H-Bridge circuits are wired to two main 4 pin headers. All of these signals are TTL compatible (driven by +5V and 0V). These signals are of the following 4 types:

1) Direction – when a positive voltage is applied to this pin, the motor will turn in the reverse direction (current flows from the negative to positive terminal on the motor output). When a negative voltage is applied to this pin, the opposite will occur.
2) Brake – The brake is active high, therefore a high voltage will engage the brake.
3) PWM – A duty cycle modulated signal up to approximately 500 kHz.
4) Thermal Flag – This signal provides the thermal warning. This pin becomes active-low when the junction temperature of the device reaches 145$^{\circ C}$. The chip will automatically shut it self down when this temperature reaches 170$^{\circ C}$.

The following truth table outlines the various input and output signal combinations:

PWM	Direction	Brake	Motor Output	Status LED Output
H	H	L	Reverse	RED
H	L	L	Forward	GREEN
L	X	L	Terminals High	OFF
H	H	H	Terminals High	OFF
H	L	H	Terminals Low	OFF
L	X	H	NONE	OFF

Table 1 - Logic Truth Table (Motor Driver Module)

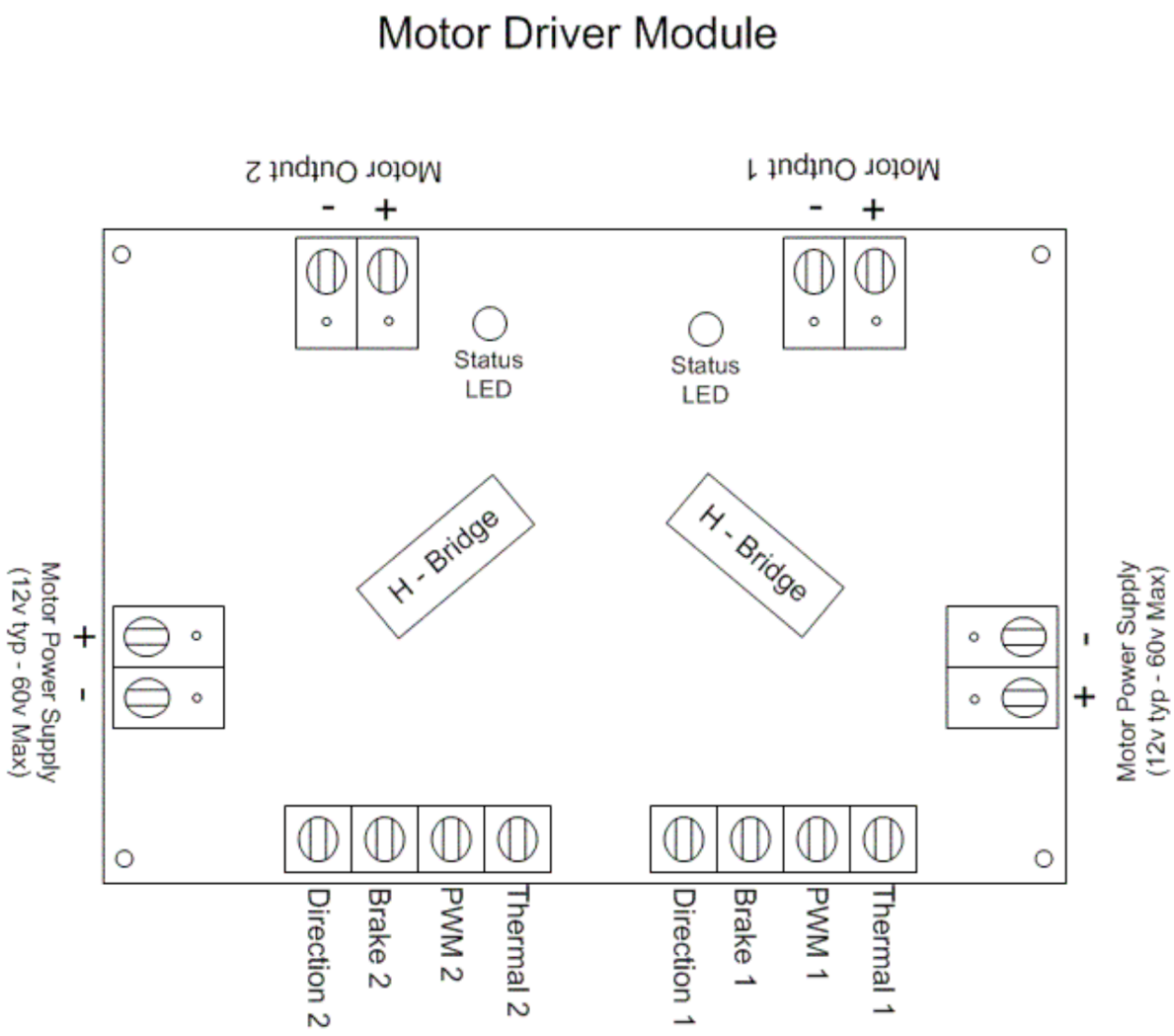


Figure 1 – Interface Board Schematic

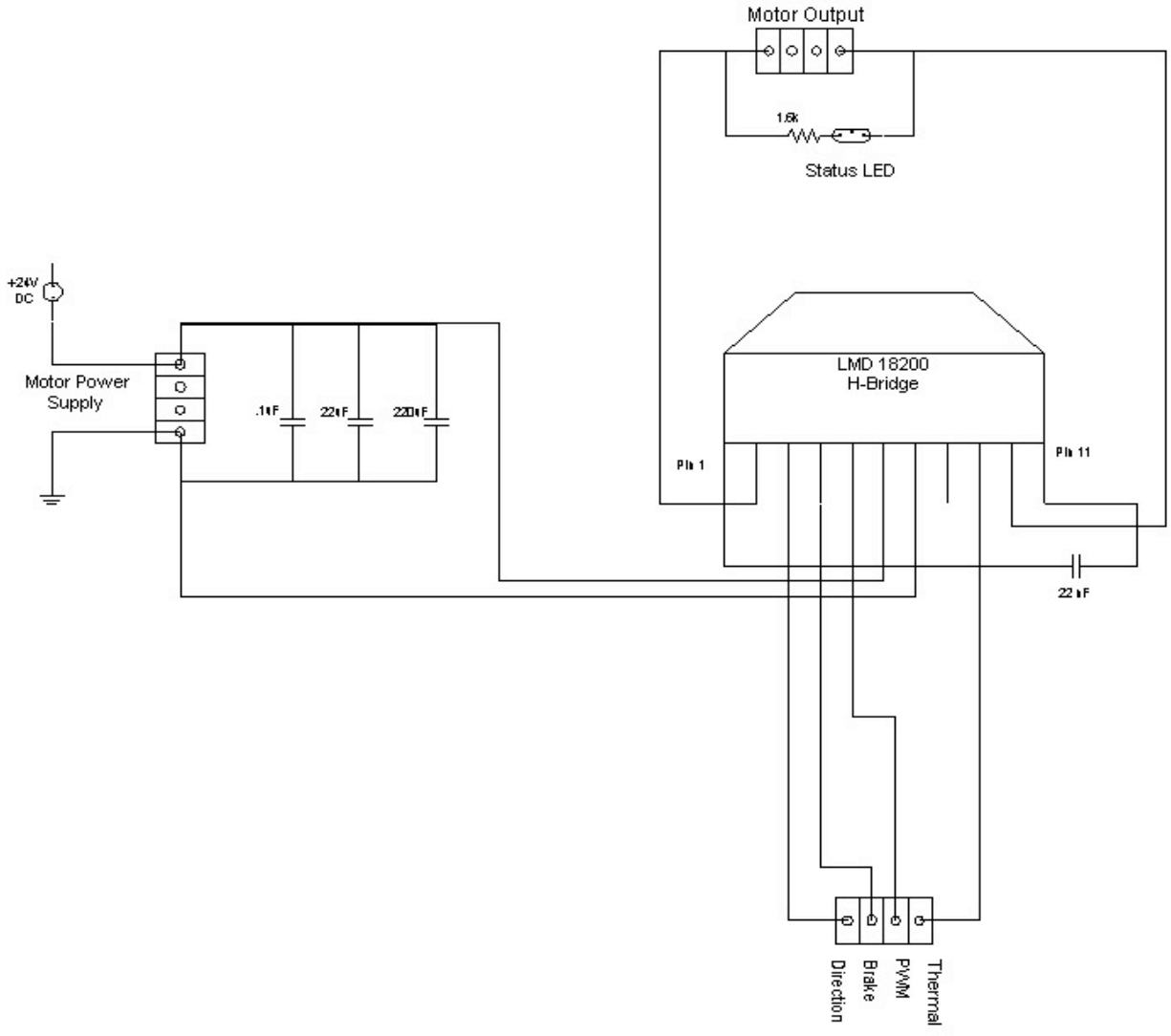


Figure 2 - Motor Driver Module Schematic

Motor Control Module (Master Controller)

The Motor Control Module is responsible for serial communication with the remote terminal (i.e. notebook computer in this case) and platform systems. Platform systems include: solenoid firing systems, motor driver circuitry, manual override controls, and communication with the slave processor which is responsible for monitoring the encoders.

The manual controls are used to move the platform without remote control and zeroing of the platform before operation. Communication with the remote terminal is performed on a remotely requested basis. The remote terminal has several commands that can be issued to the platform to perform various tasks. Upon receiving a request data command, the control module prepares and sends a packet containing pertinent system information such as axis angles and chamber pressure.

Axis motion control is performed using two different methods. The first is onboard closed-loop axis control and the second is manual motor power manipulation. Closed-loop control has been implemented on the theta axis. The input to the control is the desired angle and the outputs of the control are a PWM and direction signal to the motor driver circuitry. The controller runs a position PID routine to determine the outputs. While this control algorithm can control position, it cannot control the speed at which the final position is achieved.

The 68HC908 microcontroller offers up to two timer systems, a PWM timer system, eight ADCs, and two full ports of I/O as well serial communication. The microcontroller also contains 32K of Flash memory. Furthermore, the processor can run at up to 8 MHz with use of the onboard Phase Lock Loop (PLL). For these reasons, choosing the 68HC908 as the master controller offers a very complete system with code storage without battery backup. The development environment for this microcontroller was designed and completed before this project began. Thus, this report does not include the necessary details describing the development of this environment.

Encoder Module (Slave Controller)

The slave processor is responsible for keeping track both the theta and beta (x- & y- axes) absolute encoder position.

The A channels from the encoder for the theta and beta axes are fed into the Timer Input Capture (TIC) pins TCH0 (PTD4 Pin 19) and TCH1 (PTD5 Pin 18) respectively. The B channels are fed into PTD2 (Pin 17) and PTD3 (Pin 16). The TICs are configured to cause an interrupt on both the rising and falling edges. The direction of rotation can be determined by looking at the state of the B channel after an input capture interrupt occurs. Based upon the rotation direction, the global counter for that axis is either incremented or decremented. Consult the flow diagram below (Figure 4) below for the detailed operation of these interrupts.

The current encoder positions are transmitted from the slave controller to the master processor using a 5-bit data bus and three control signals. The control signals are the Read Strobe (Rs), Control Signal 0 (Cs0) and Control Signal 1 (Cs1). The current position of each axis is transmitted on the data bus using two nibbles. The slave processor determines which nibble to send based on the state of the two control signals, CS0 and CS1.

CS0	CS1	Nibble
0	0	Theta Low
0	1	Theta High
1	0	Beta Low
1	1	Beta High

Table 2 - Control Signal Decoding

The encoder position is valid while the Read Strobe is high. Reference the timing diagram (Figure 3) below for details.

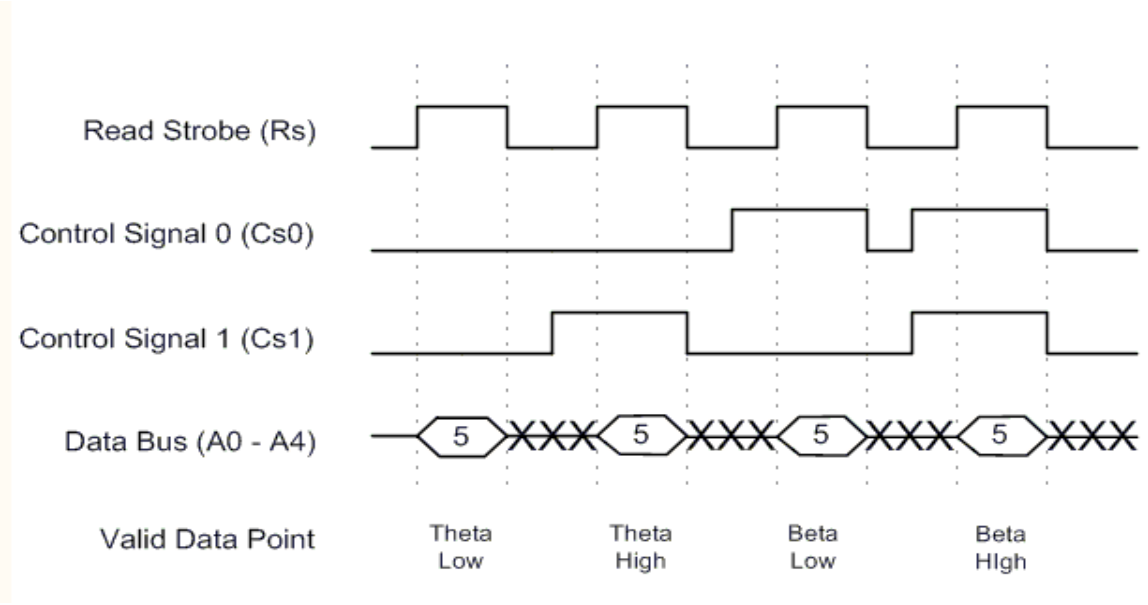


Figure 3 – Timing Diagram – Encoder Data Transfer

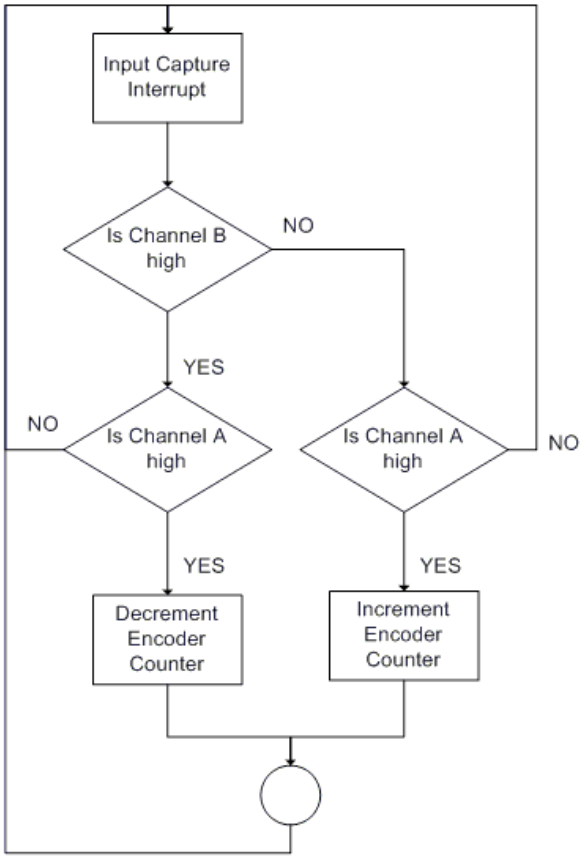


Figure 4 – Encoder Input Capture Flow Diagram

Communications Interface

The Motion Control Module, Encoder Module and the System Control Module all share a common RS-485 communications interface. Each module will have a unique address on the data link.

Theory of Operation

During typical usage for Time-On-Target shots, the "LoadData" command is issued once for each shot in the sequence. When all data points have been loaded, the Fire Control shall issue a "Fire" command. This will cause the motion control to execute the firing of each of the data points. From this point forward, the Fire Control Module can continue to issue "Fire" commands one after another on the same data points. If a new Time-On-Target shot is desired, the user shall issue a "ClearData" command followed by a sequence of "LoadData" commands done previously.

Before entering Single-Shot mode, a ClearData command shall be issued. Single-Shot mode shall continuously update the data for sequence number 0 until the desired data points have been reached. When the "Fire" command is issued, the Motion Control module will fire all the data points in memory (1 shot in this instance).

Message Protocol

Start Byte	0xAA
Source	
Destination	
Command	
Data1	
Data2	
Data3	
Data4	
Data5	
Data6	
Data7	
Checksum	0xEE
Stop Byte	0xDD

Table 3 - Message Protocol

Null Data should be filled with 0x11.

Commands

LoadData - This command shall be issued from the Fire Control Module to the Motion Control Module. This command will contain 5 data items.

1. Sequence number – (Used for multiple shots)
2. Beta Angle (Vertical Axis) – 2 Bytes
3. Theta Angle (Horizontal Axis) – 2 Bytes
4. Time (Measured in tenths of a second) – 1 Byte
5. Pressure (Measured in psi) – 1 Byte

Start Byte	0xAA
Source	Source of Command
Destination	Destination Module
Command	0xB1
Data1	Sequence #
Data2	BetaAngle LowByte
Data3	BetaAngle HighByte
Data4	ThetaAngle LowByte
Data5	ThetaAngle HighByte
Data6	Time
Data7	Pressure
Checksum	0xEE
Stop Byte	0xDD

Table 4 - LoadData Command

Fire – This command shall be issued from the Fire Control Module to the Motion Control Module. This command will cause the Motion Control Module to execute the firing sequence of all the data points stored in memory. If this command is issued to any other module it shall have no effect.

Start Byte	0xAA
Source	Source of Command
Destination	Destination Module
Command	0xC0
Data1	0x11
Data2	0x11
Data3	0x11
Data4	0x11
Data5	0x11
Data6	0x11
Data7	0x11
Checksum	0xEE
Stop Byte	0xDD

Table 5 - Fire Command

ClearData – This command shall be issued from the Fire Control Module to the Motion Control Module. This command will clear all of the stored data points in the Motion Control Module. If this command is issued to any other module it shall have no effect

Start Byte	0xAA
Source	Source of Command
Destination	Destination Module
Command	0xD0
Data1	0x11
Data2	0x11
Data3	0x11
Data4	0x11
Data5	0x11
Data6	0x11
Data7	0x11
Checksum	0xEE
Stop Byte	0xDD

Table 6 - ClearData Command

System Control Module

The System Control Module is essentially a LabVIEW Virtual Instrument (VI). A VI was chosen as the System Control Module because it allows for an intuitive graphical user interface (GUI) and allows for control via a generic notebook computer that has LabVIEW installed.

The System Control Module has three main Frame Control Structures that correspond to the three types of functionality of our system. The three methods of operation are: Single Shot, Time-On-Target, and Manual Control. Thus far, Single Shot and Manual Control are implemented completely. Time-On-Target will be implemented in the future as is discussed below in the Discussion section.

Once the program is running and the Single Shot tab is selected, the program prompts the user for Target Range in yards, Time to Target in seconds and Theta Angle in degrees. When the "Calculate" Button is pressed, the VI runs a MatLab script (see Appendix A) that calculates the Beta angle and the air pressure required to launch the projectile with the above constraints. The user then presses "Upload," and the VI sends the pertinent calculated parameters to the master controller. Finally, when the "Fire" button is pressed and the "Arm" button is active, the pressure is uploaded to the master controller and the projectile is launched.

The Manual Control tab allows a user to do just that, manually control the launcher. The first dial (Beta motor power) changes the pulse width of the motor control signal. Thus, as the dial is moved, the motor controlling the Beta axis moves until the dial is reset to zero. The Theta position dial controls the Theta angle. This dial is closed loop, i.e. it moves to the location specified by the dial. The last input is the desired pressure to launch the projectile at. From this point, operation occurs as specified above. When the "Arm" button is active and the "Fire" button is pressed, the pressure is uploaded and the projectile is launched.

Finally, the entire VI is surrounded by a while structure so that while the "Stop" button is not pressed the system will continue to run. Thus, there is a master emergency stop button available on the VI in the case that something malfunctions.

Below are a series of figures that depict the Virtual Instrument described above. Figure 5 shows the Front Panel of the Manual Control Frame of the LabVIEW VI. The front panel is the view seen by the user. Figure 6 shows the Block Diagram of the Single Shot frame discussed above.

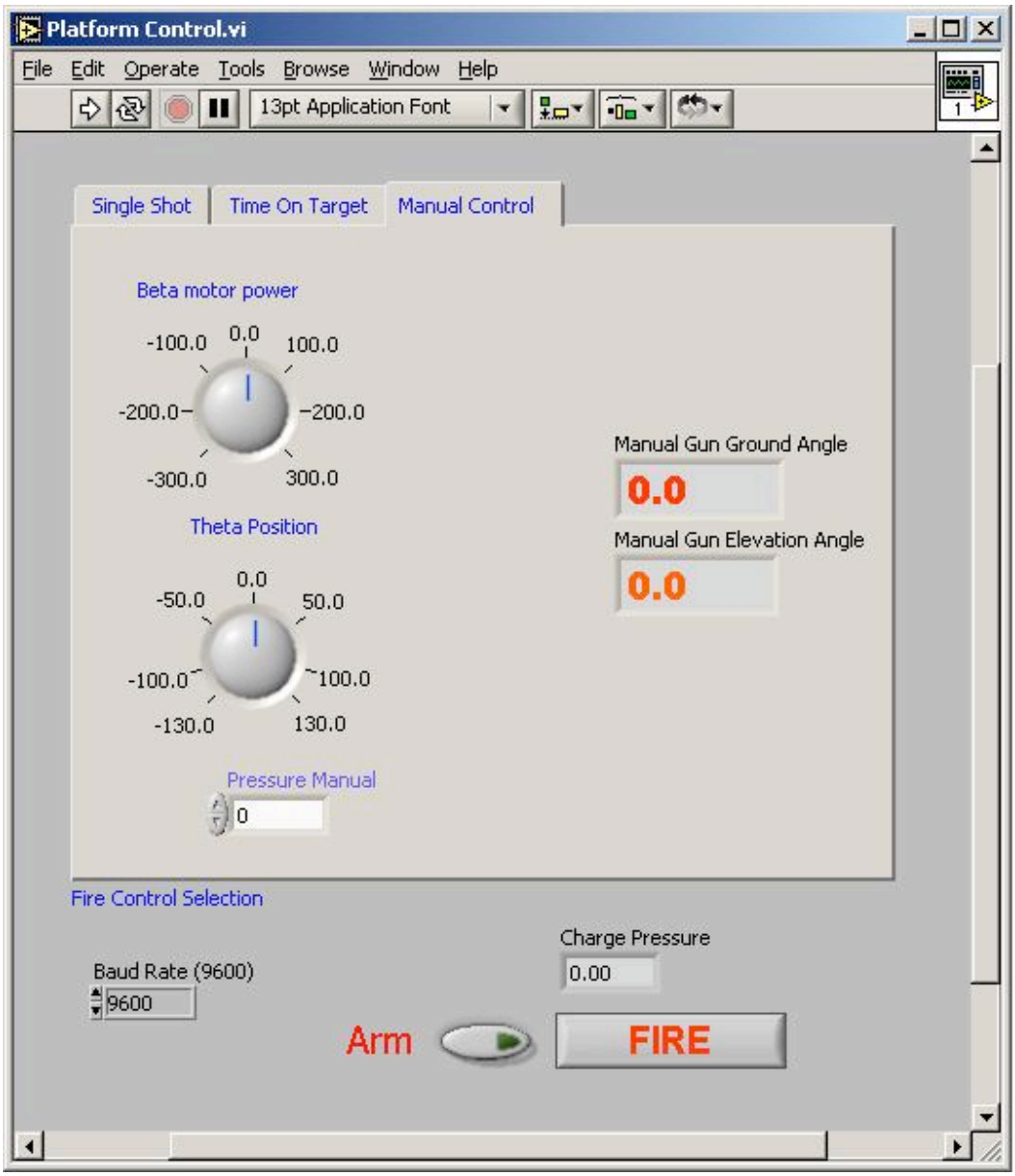


Figure 5 - VI Front Panel - Manual Control

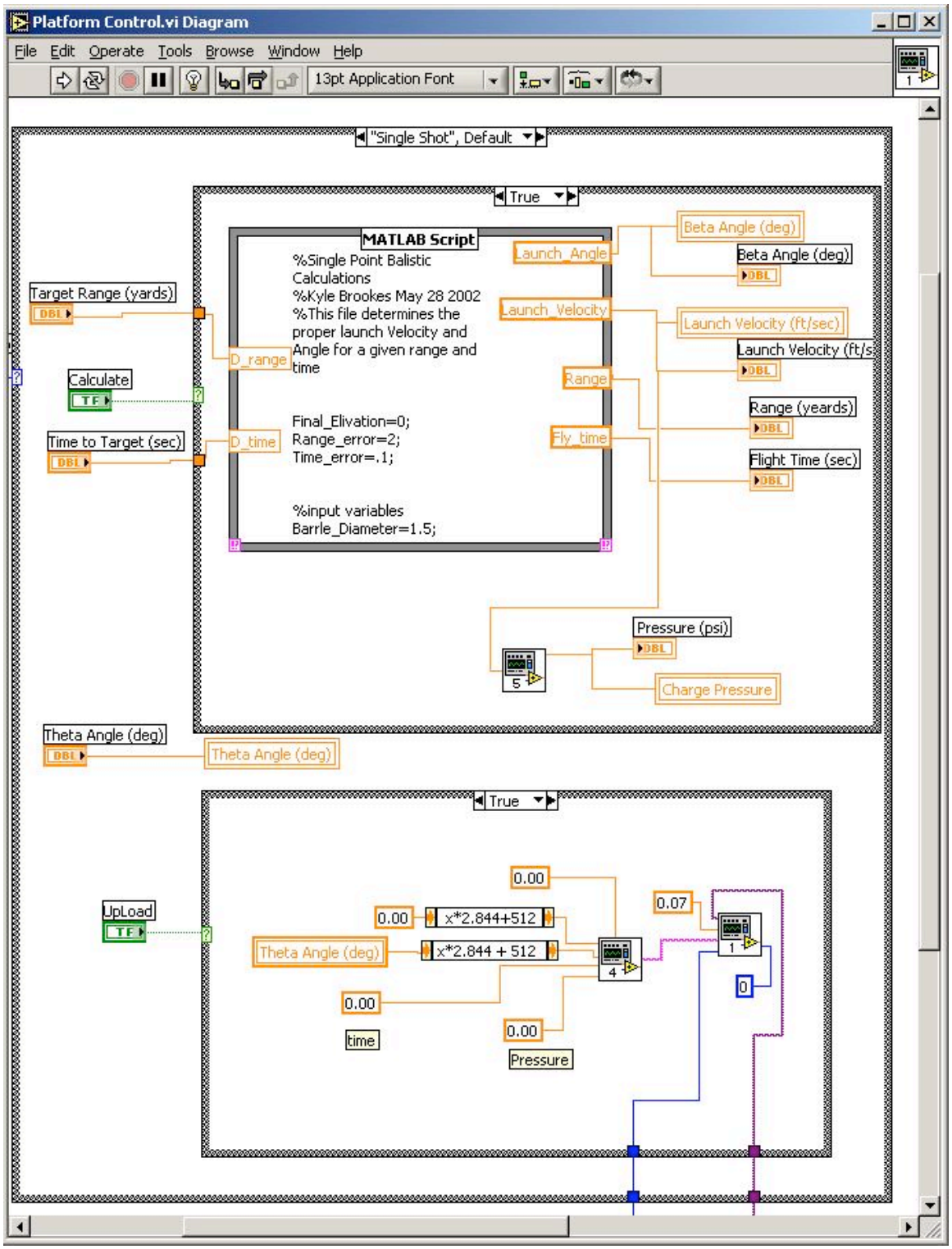


Figure 6 - VI Block Diagram - Single Shot Frame

Results

As the end of the term neared, much of what we originally set out to accomplish in our design has been completed. While there are admittedly bugs to be worked out, our core design is functional. Of our subcomponents, the motor control system is lacking the most. While our actual motor control circuit and PID feedback loop work correctly, mechanical limitations in our design account for much of the error. Our theta axis is a testament to our thoughtful design. Currently, it works consistently and accurately, moving the assembly to the desired angle. Our encoders provide the necessary feedback to accurately drive the motors to the inputted angle. If the mechanical problems were to be worked out of the beta axis, we are confident our design would work.

The communications protocol and wireless interface boards work as we had envisioned. The wireless boards allow us to easily control our launcher from up to seven miles away, under optimal conditions. Our communications protocol is efficient and stores all the necessary information to direct and fire the launcher. Our wireless boards, in conjunction with the communications protocol, make up a complete subsystem. The only area in which we would see further work would be to streamline the protocol to be more efficient. This would in turn speed up communications between the user and platform.

The overall system control of our launcher is essentially complete. While our design is fully implemented in LabVIEW, not all of our components are supported in hardware. Our user interface is complete in that a user can successfully enter in data points and choose to fire the launcher in various ways. LabVIEW is able to successfully analyze the inputted data, and then communicate with the various subsystems to translate this into a successful firing sequence. We had hoped to implement Time-On-Target capabilities in our design. Currently, the LabVIEW VI has structures to input data for Time-On-Target sequences and can correctly communicate with the launcher for a successful fire sequence. LabVIEW has the mathematical and functional capabilities to perform everything we set out to do initially. Mechanical aspects of our launcher, however, limit us in our ability to actually perform Time-On-Target functions. Using the LabVIEW GUI interface, a user can control the various subsystems to accurately fire a single shot at a desired target. Also, using the VI Front Panel, the user can also select from the proper menus to input the necessary data points for a single shot fire. We also chose to implement a menu so the user could manually choose to aim and fire the launcher.

While there is still more work to be done on our launcher design and some of its subsystems, it is almost fully operational. Much of what we initially set out to accomplish has been completed. Considerable progress has been made as much of our design is implemented as we had planned.

Discussion

Initially, when approaching this project we had many goals and features in mind. With the scant time allowed, we found ourselves having to scale back our original design. In the end, we were not able to fully accomplish all of the aspects of the project we had originally planned. We found time to be our biggest constraining force. With more time, we feel we could have had our design fully tested and operational. Due to the very unique nature of our project, considerable time was spent researching and procuring parts.

Many of the parts we used are being used in ways they had not been designed for. For instance, in our original implementation, our encoders were supposed to signal a bank of bidirectional counters that would capture the absolute position of the two axes. While this hardware worked in simulation, it did not translate to a working implementation. In the end, we chose to alter our original design and use a slave microprocessor to handle the encoder output, thereby doing much of the work in software. While this design worked efficiently, considerable time was wasted debugging our original design.

Although many of the components of our initial design are implemented, there is still considerable work that needs to be done. Much of the core hardware and components are in place but need fine tuning to be fully operational. We appreciate that there are many places in our design where things could be done more efficiently and technically correct. We did not have the resources, knowledge or budget to design this project as it would be done in industry. Our design in itself is robust, but the hardware we had available to us was lacking in some aspects. One of the major problems we encountered with our design was the beta axis motor control. Due to the gear ratio in our motor control system, there is considerable degradation in accuracy. There is also considerable backlash in our motor assemblies that also allows for "play" and further inaccuracies. Additional time is needed to complete the mechanical work for the beta axis.

Many of the physical components of our gun were machined here at RPI from stock aluminum. While many of the parts were designed in CAD and machined on CNC equipment, there are still flaws prevalent. Given more time and resources, many of the bugs could have been worked out further. Our initial design included the capabilities to do Time-On-Target firing. This would allow several shells to be fired so that they would hit the desired target at the same time. However, due to the inaccuracies in our beta axis, this is no longer feasible. Given that the various subcomponents were to work correctly we feel this feature would work properly. Additional work needs to be done to our communication protocol and systems to increase its speed and efficiency. Having spent considerable time and effort on this project, work on it will be continued during the upcoming semester.

References

Cady and Sibigtroth, "Software and Hardware Engineering"

MC68HC908MR32/MC68HC908MR16 Advance Information HCMOS Microcontroller Unit Manual.

Various Data Sheets for TTL Compatible ICs.

Appendices

Appendix A – MatLab .m files

```
%Pressure Matrix Loader
%Pressure Matrix Loader.m
%Kyle Brookes November 17 2002
%This file loads the pressure & velocity matrices

load Pressure_Matrix;
load Velocity_Matrix;

*************************************************************************

%Single Point Ballistic Calculations
%Single Point Ballistic Calculations.m
%Kyle Brookes November 17 2002
%This file determines the proper launch Velocity and Angle for a given range and
time



Final_Elivation=0;
Range_error=2;
Time_error=.1;



%input variables
Barrle_Diameter=1.5; %(inches)
Mp=.070;              %Projectile Mass (kg)
MVE=200; %Launch Velocity (f/sec)
Launch_Angle=37; %Degrees

%Static Parameters
rho=1.3;            %Density of air

%Calculated parameters
Ab=(Barrle_Diameter)^2*.7853*.00064516; %X-section Area of the Barrel (m^2)
MV=MVE*.3048;
Beta=pi/4;
Beta_H=pi/2;
Beta_L=0;
```

```
while 1
    while 1 %calculate max fly velocity

        dxi=cos(Beta)*MV; %Initial X Velocity
        dyi=sin(Beta)*MV; %Initial Y Velocity
        [t,x,y]=sim('Projectile'); % Run Simulation
        range_missmatch = D_range - Range;
        if abs(range_missmatch) < Range_error
            break
        end
        if range_missmatch > 30
            MV_gain = 2;
        elseif range_missmatch > 15 & range_missmatch < 30
            MV_gain = 1;
        elseif range_missmatch < 15
            MV_gain = .5;
        elseif range_missmatch <20 & Beta < .05
            MV_gain =.05;
        end
        MV = MV * (1 + (range_missmatch/D_range)*MV_gain);
    end

    time_missmatch = D_time - Fly_time;
    if abs(time_missmatch) < Time_error
        break
    end

    %Calculate new launch angle

    if time_missmatch < 0
        Beta_H = Beta;
        Beta = (Beta_H-Beta_L)/2 + Beta_L;
    else
        Beta_L = Beta;
        Beta = (Beta_H-Beta_L)/2 + Beta_L;
    end


end
```

```
Launch_Velocity = MV/.3048;
Launch_Angle = Beta*180/pi;
%Range;
%Fly_time;
```

Appendix B – System Overview Block Diagram

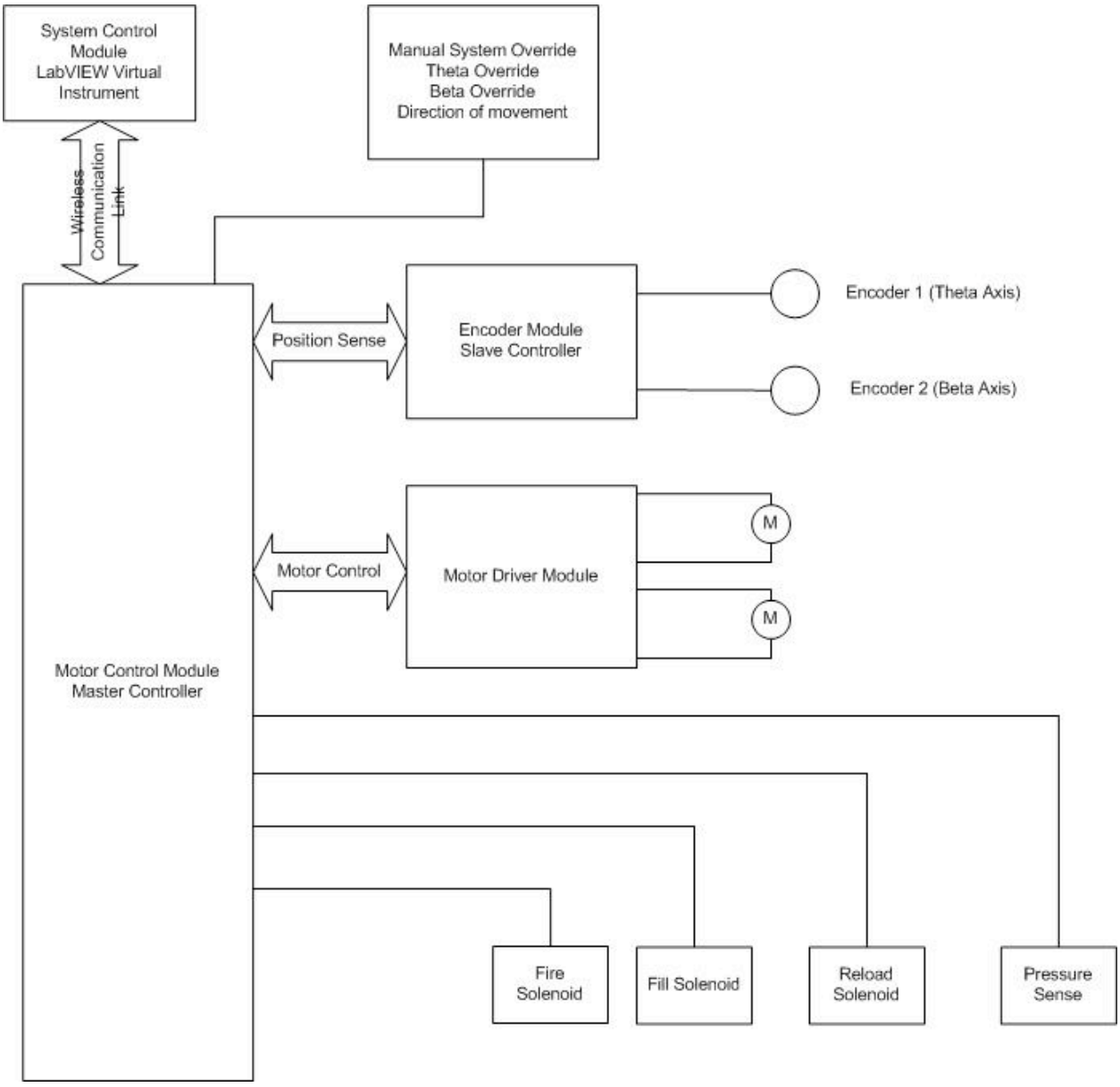


Figure 7 - System Overview Block Diagram

Appendix C – C Code

```
// setup.c
// Handles initial setup of HC908 Controllers

#include <AIomr32.h>


void _HC08Setup(void)
      {
      asm("CLI\n");       //Clear the Interrupt Mask Bit in the CCR

      PPG = 0x66;         //Set PLL for 7.3728 MHz opperation using 4.9152 MHZ
crystal
      PBWC |= 0x80;       //Use AUTO PLL bandwith control
      PCTL = 0x2f;              //Enable PLL
      while((PBWC & 0xc0)!= 0xc0);   //wait for PLL stabilization
      PCTL = 0x3f;       // select PLL as clock

      SCC1 = 0x40;       // enable SCI
      SCC2 = 0x2C;       // enable both TX and RX and RX interrupt
      //SCBR = 0x01;     // Baud Rate = 57600
      SCBR = 0x12;       // Baud Rate = 9600

      CONFIG = 0x091;    // Edge Aligned-independent PWMs,disable COP

      TBSC &= ~0x80;     //clear timer B overflow flag
      TASC &= ~0x80; //reset flag bit
      TASC3 &= ~0x80; //clear timer A channel 3 input Capture flag
      TASC2 &= ~0x80; //clear timer A channel 2 input Capture flag
      TASC1 &= ~0x80; //clear timer A channel 1 input Capture flag
      TASC0 &= ~0x80;  //clear timer A channel 0 input Capture flag
      ISCR = 0x04;             //reset irq latch

}

*****************************************************************************

// Spud Platform Control Code
// SpudControl.c
// Kyle Brookes, December 2 2002

#include <AIomr32.h>
#include <Avectors.h>

#pragma interrupt_handler timerB_overflow_handler
#pragma interrupt_handler SCIrecieve


void Initialize (void);
void UpdateEncoderValues(void);

extern char split_high(unsigned int high ); //returns high 8 bits
extern char split_low(unsigned int low);   //returns low 8 bits
extern void PrintChar(char c);
extern void PrintNum(long num);
extern void nlcr(void);
```

```
extern void ProcessIncomingMessage(void);

//Theta Motor Control Variables
char motor_T_register_flag = 1;
long new_T_motor_speed = 5;
long T_old_motor_speed = 0;
long T_delta_error = 0;
long T_encoder = 0;
long T_error = 0;
long T_error_1 = 0;
long T_error_2 = 0;
float T_Kp = 70;
float T_Kd=15;
float T_Ki =10;
long D_Theta=512;


//Beta Motor Control Variables
char motor_B_register_flag = 1;
long new_B_motor_speed = 5;
long new_B_temp_motor_speed = 0;
long B_encoder = 0;
long B_error = 0;
long D_Beta=512;



// Global Variables for Serial Com
extern char FlagIncomingMessage;
extern char FlagProcessingMessage;
extern char MessageProcessingBuffer[13];
extern char IncomingMessageBuffer[13];
extern char OutgoingMessageBuffer[13];


// Misc globals
long time=0; char Pressure=0; char Gun_P=0;
char T_manual_move = 0; char B_manual_move = 0;
char temp=0; char low_5=0;
char RTI_execute =1; char FIRE=0; char FIRING=0;
long fire_duration=0;
unsigned int high_5=0;
long mod_pot=0;
unsigned long i=0;

void main(void)
{
      Initialize();

      while(1)
      {
            if(FlagIncomingMessage)
            {
                  ProcessIncomingMessage();
            }

            if(PTC & 0x20)   //B switch not pressed
            {
                  B_manual_move =0; //enable remote conrol on Beta axis
```

```
            }
            else
            {
                  B_manual_move = 1;      //enable manual move for Beta axis
            }

            if(PTC & 0x40)    //T switch not pressed
            {
                  T_manual_move =0; //enable remote conrol on Theta axis
            }
            else
            {
                  T_manual_move = 1;      //enable manual move for Theta axis
            }
      }
}

//Realtime 10 msec interrupt
void timerB_overflow_handler (void)
{
      if(RTI_execute) //used to dissable RTI during message recieving
      {
      UpdateEncoderValues();

            if(T_manual_move)
            {
            ADSCR = 0x07;   //A/D Conversion on PB7
            while ((ADSCR & 0x80) == 0 );
            mod_pot = ADRH;
                  mod_pot = mod_pot*256 + ADRL;

                  D_Theta = mod_pot;  //from 0 to 1024
                  //assigns the desired position from 50 to 997
            if(D_Theta < 50)
            {
                  D_Theta = 50;
            }
            else if (D_Theta > 975)
            {
                  D_Theta = 975;
            }
            }


      /**** Calculat Theta Motion
*********************************************/
       T_error = D_Theta - T_encoder;
       T_delta_error = T_error-(T_error + T_error_1 + T_error_2)/3;   //runing
average of velocity

         //PID control routine for Theta axis
       new_T_motor_speed = T_old_motor_speed + T_error*(T_Kp+T_Ki) -
T_Kp*T_error_1 +T_Kd*T_delta_error;

            if(new_T_motor_speed > 0)
      {
                  PTA |= 0x01;      //forward

                  if(new_T_motor_speed > 997)  //check against limit
```

```
                {
                      new_T_motor_speed = 997;
                }
          T_old_motor_speed = new_T_motor_speed;  //store old dity value
          }
          else
          {
                PTA &= ~0x01;     //reverse

          if(new_T_motor_speed < (-997)) //check against limit
          {
                new_T_motor_speed = -997;
    }
      T_old_motor_speed = new_T_motor_speed; //store old dity value
        new_T_motor_speed = (-1)*new_T_motor_speed;  //Make duty value positive
      }


      //Write Theta Signal to TCH0 using Buffered PWM
      if(motor_T_register_flag)
      {
            TACH0H = split_high(new_T_motor_speed);
            TACH0L = split_low(new_T_motor_speed);
      motor_T_register_flag = 0;
      }
      else
      {
       TACH1H = split_high(new_T_motor_speed);
       TACH1L = split_low(new_T_motor_speed);
      motor_T_register_flag = 1;
      }
      T_error_2 = T_error_1;   //update error pipeline
      T_error_1 = T_error;



      /**** Calculat Beta Motion
**********************************************/
      if(B_manual_move)
            {
            ADSCR = 0x07;   //A/D Conversion on PB7
            while ((ADSCR & 0x80) == 0 );
            mod_pot = ADRH;
            mod_pot = mod_pot*256 + ADRL;
            new_B_motor_speed = (mod_pot*2) - 1024;  //from 1024 to -1024
            }
            else
            {
                  new_B_motor_speed = new_B_temp_motor_speed;
            }

        if(new_B_motor_speed > 0)
            {
                  PTA |= 0x02;      //forward

                  if(new_B_motor_speed > 996)  //check against limit
                  {
                        new_B_motor_speed = 996;
                  }
```

```
            }
            else
            {
            PTA &= ~0x02;      //reverse

            if(new_B_motor_speed < (-996)) //check against limit
            {
                  new_B_motor_speed = -996;
            }
            new_B_motor_speed = (-1)*new_B_motor_speed;
        }


      //Write Beta Signal to TCH2 using Buffered PWM
       if(motor_B_register_flag)
      {
       TACH2H = split_high(new_B_motor_speed);
       TACH2L = split_low(new_B_motor_speed);
      motor_B_register_flag = 0;
      }
      else
      {
       TACH3H = split_high(new_B_motor_speed);
       TACH3L = split_low(new_B_motor_speed);
      motor_B_register_flag = 1;
      }



            ADSCR = 0x06;   //A/D Conversion on PB6 for gun Pressure
            while ((ADSCR & 0x80) == 0 );
            mod_pot = ADRH;
            mod_pot = mod_pot*256 + ADRL ;
            Gun_P = (mod_pot*100)/1024; //Gun pressure on a scale of 0 -> 100

             if(Gun_P < Pressure)
            {
                  PTA |= 0x10;      //open fill valve
            }
            else
            {
                  PTA &= ~0x10;     //close fill valve
            }

            if(FIRE && ((Pressure - Gun_P) <= 2))      //wait for pressure to
come up
            {
                  FIRE = 0;
                  FIRING = 1;
                  PTA |= 0x20;            //Fire the gun
                  Pressure = 0;           //set the pressure to 0 so no reload
                  fire_duration = 0;//set timer to zero
            }
            if (FIRING && (fire_duration == 40))
            {
                  PTA &= ~0x20; //close FIRE valve
                  PTA |= 0x40;      //opend reload vlave
```

```
            }

            if (FIRING && (fire_duration == 80))
            {
                  PTA &= ~0x40;     //close reload valve
                  FIRING = 0;             //end firing sequence
            }

            if (FIRING)
            {
                  fire_duration++;  //incorment fire time counter
            }

      }
      else
      {
            RTI_execute = 1; //reenable the RTI for next time
      }

 TBSC &= ~0x80; //reset overflow flag bit
}

//external interrupt dummy handler
void external_handler (void)
{
 ISCR |= 0x04;
}

void SCIrecieve (void)
{
   char rec_ctr=0;
   char rec_ctr_2=0;
   char test=0;
   char temp;

    temp = SCS1; //Clear Flag
      RTI_execute = 0; //dissable the RTI for one cycle = 20 msec

      //shift message buffer to make room for new byte
      for(rec_ctr=0; rec_ctr<12; rec_ctr++)
      {
            IncomingMessageBuffer[rec_ctr] = IncomingMessageBuffer[rec_ctr+1];
      }

      // Get the Incoming data
      IncomingMessageBuffer[12] = SCDR;


  //valid message recieved
   if((IncomingMessageBuffer[0] == 0xAA) &&  (IncomingMessageBuffer[11] == 0xEE)
&& (IncomingMessageBuffer[12] == 0xDD))

   {
      //We have a valid message

        // If we are not currently processing a message
        if(!FlagProcessingMessage)
        {
           // Copy the incoming message to the processing buffer
```

```
            for(rec_ctr_2=0; rec_ctr_2<13; rec_ctr_2++)
              {
                 MessageProcessingBuffer[rec_ctr_2] =
IncomingMessageBuffer[rec_ctr_2];
              }

            //Set the Incoming Message Flag
            FlagIncomingMessage = 1;
         }
    }
}


//Initialize functoin
void Initialize (void)
{
      //Prot Directions
      DDRA = 0xff;          //Port A for output
      DDRB = 0x00;         //PTB used for A/D and encoder data
      DDRC = 0x0f;          //Port C 0-3 output, 4-6 input
      PTA = 0x00;              //set Port A LOW
      PTC = 0x00;              //set Port C LOW

      //Timer B
      TBMODH = 0x8f;           //Overflow at 20 msec, Used for RTI and Servo
Control
      TBMODL = 0xfe;

      //Timer A
      TAMODH=0x03;          //TICA0 capable of counting from (1 count) .1356
microsec to .868 msec (1000 count)
      TAMODL=0xe8;             //                        Driven at 7.37 kHz
      TACH0H=0x00;             //0.5% duty, motor off, first write to ch0 next to
ch1
      TACH0L=0x05;
      TASC0=0x2a;                 //Buffered PWM on TAC0, Theta motor control
      TACH2H=0x00;             //0.5% duty, motor off, first write to ch0 next to
ch1
      TACH2L=0x05;
      TASC2=0x2a;                 //Buffered PWM on TAC2, Beta motor control

      //A/D
      ADCLK = 0x54;   //A/D fop/8, Right Hand Mode

      TBSC = 0x02;              //Timer B counter prescaler of 4, turn counter on
      TASC = 0x00;             //Timer A counter prescarer of 1, turn counter on
      TBSC |= 0x40;            //enable Timer B overflow interrupt
}

void UpdateEncoderValues(void)
{
      /****** THETA ENCODER UPDATE ROUTINE ******/

       // Set C1 and C0 low (Theta low bits)
       PTC &= ~(0x0C);
      // Wait an extra cycle to sync up
       while((PTB & 0x20) != 0x00);// Make sure that Rs strobe low (previous
data is gone)
       while((PTB & 0x20) == 0x00);// Wait for the Rs high strobe
```

```
       while((PTB & 0x20) != 0x00);// Make sure that Rs strobe low (previous
data is gone)
       while((PTB & 0x20) == 0x00);// Wait for the Rs high strobe

       low_5 = (PTB & 0x1F);// Save the lower 5 bits of Theta

      // Set C1 low and C0 high
       PTC &= ~(0x08); // C1 low
       PTC |= 0x04; // C0 high

       while((PTB & 0x20) != 0x00);// Make sure that Rs strobe is low (previous
data is gone)
       while((PTB & 0x20) == 0x00); // Wait for the Rs high strobe

       high_5= PTB & 0x1F;// Save the upper 5 bits of Theta
       T_encoder = ((high_5 << 5) + low_5);
      /***********************************************/

      /****** BETA ENCODER UPDATE ROUTINE ******/
      // Set C1 high and C0 low (Beta low bits)
      PTC |= 0x08; //C1 high
      PTC &= ~(0x04); // C0 low

      while((PTB & 0x20) != 0x00);// Make sure that Rs strobe low (previous data
is gone)
      while((PTB & 0x20) == 0x00);  // Wait for the Rs high strobe

      low_5 = (PTB & 0x1F);  // Save the lower 5 bits of Theta
      PTC |= 0x0C; // C0 high C1 high (Beta high bits)

      while((PTB & 0x20) != 0x00);// Make sure that Rs strobe is low (previous
data is gone)
      while((PTB & 0x20) == 0x00);// Wait for the Rs high strobe

      high_5= PTB & 0x1F;// Save the upper 5 bits of Theta
      B_encoder = ((high_5 << 5) + low_5); //encoders are
      /***********************************************/
}
```

```
*************************************************************************

// serial_com.c
// Handles serial communication in system

extern void PrintNum(long num);
extern void PrintChar(char c);
extern void nlcr(void);

// Global Variables
char FlagIncomingMessage=0;
char FlagProcessingMessage=0;
char MessageProcessingBuffer[13];
char IncomingMessageBuffer[13];
char OutgoingMessageBuffer[13];
```

```
extern char FIRE;
extern long T_encoder;
extern long D_Theta;
extern long B_encoder;
extern long D_Beta;
extern long new_B_temp_motor_speed;
extern long time;
extern char Pressure;
extern char Gun_P;

void ProcessIncomingMessage(void)
{
   int *Theta;
   int *Beta;

   /*
     Signal that we are processing a message, this will prevent
     the SCI interrupt from overwriting the current message while
     it is still in use
   */
   FlagProcessingMessage = 1;

   // Clear the incoming message flag
   FlagIncomingMessage = 0;

   switch(MessageProcessingBuffer[3])
   {
      // Load Data Command
      case 0xB1:


           //Theta and beta are pointers to integers
             //This will grab both bytes from the buffer
             Theta = (int *)&MessageProcessingBuffer[7];
           D_Theta = *Theta;
             Beta = (int *)&MessageProcessingBuffer[5];
             new_B_temp_motor_speed  = (*Beta *2) - 1024;  //from 1024 to -1024
             time = (char)MessageProcessingBuffer[9];
             break;

           // Fire Command
           case 0xC0:
           FIRE = 1;
           Pressure = (char)MessageProcessingBuffer[10];
           break;

           //Request Data Command
           case 0xD1:
           /*
                 PrintNum(MessageProcessingBuffer[0]);
                 PrintChar('-');
                 PrintNum(MessageProcessingBuffer[1]);
                 PrintChar('-');
                 PrintNum(MessageProcessingBuffer[2]);
                 PrintChar('-');
                 PrintNum(MessageProcessingBuffer[3]);
                 PrintChar('-');
                 PrintNum(MessageProcessingBuffer[4]);
                 PrintChar('-');
```

```
                PrintNum((char) (B_encoder % 256));
                PrintChar('-');
                PrintNum((char) (B_encoder / 256));
                PrintChar('-');
                PrintNum((char) (T_encoder % 256));
                PrintChar('-');
                PrintNum((char) (T_encoder / 256));
                PrintChar('-');
                PrintNum(Pressure);
                PrintChar('-');
                PrintNum(Gun_P);
                PrintChar('-');
                PrintNum(MessageProcessingBuffer[11]);
                PrintChar('-');
                PrintNum(MessageProcessingBuffer[12]);


            */

            PrintChar(0xaa);
            PrintChar(2);
            PrintChar(1);
            PrintChar(209);          //report Data
            PrintChar(0);
            PrintChar((char) (B_encoder % 256));       //Theta encoder Low byte
            PrintChar((char) (B_encoder / 256));       //Beta encoder High byte
            PrintChar((char) (T_encoder % 256));       //Theta encoder Low byte
            PrintChar((char) (T_encoder / 256));       //Theta encoder High byte
            PrintChar(Pressure);
            PrintChar(Gun_P); //Gun Pressure
            PrintChar(0xee);
            PrintChar(0xdd);

            break;


        // Clear Data
      case 0xD0:
           // Set to home position
           D_Theta = 512;
             D_Beta = 512;
             break;

        default:
           break;
   }
   FlagProcessingMessage=0;
}

*****************************************************************************



// Prints.c


#include <AIomr32.h>
#include <AStdlib.h>
```

```
//PrintChar and PrintNum used for debugging purposes

void PrintChar(char c)
{
    while ((SCS1 & 0x80) == 0)
        ;
    SCDR = c;
}



void PrintNum(long num)
{
 char digit; long place=1000000000; char test=1;

 if (num<0)
 {
  PrintChar('-');
  num=abs(num);
 }

 if (num==0)
 {
  test=0;
  place=0;
  PrintChar('0');
 }

 while (test)
 {

  if (num/(place/10))
             test=0;
  else
             place=place/10;
 }

 while (place >1)
 {
  digit = (num % place)/(place/10);
  PrintChar(digit+48);
  place = place/10;
  }
}
void nlcr(void)
{
PrintChar(10);PrintChar(13);
}


// split_high & split_low are used for motor control

char split_high(unsigned int high)
{
 unsigned int hs;
 hs = high;
 return (hs >> 8);
```

```
}
char split_low(unsigned int low)
{
 unsigned int ls;
 ls = low;
 ls = ls << 8;
 return (ls >>8);
}

*****************************************************************************

// AIomr32.h

/*    IO DEFINITIONS FOR MC68HC908MR32
 */

/*    PORTS --UPDATED
 */
#define PTA       *(volatile unsigned char *)0x00 /* port A */
#define PTB       *(volatile unsigned char *)0x01 /* port B */
#define PTC       *(volatile unsigned char *)0x02 /* port C */
#define PTD       *(volatile unsigned char *)0x03 /* port D */
#define DDRA      *(volatile unsigned char *)0x04 /* data direction port A */
#define DDRB      *(volatile unsigned char *)0x05 /* data direction port B */
#define DDRC      *(volatile unsigned char *)0x06 /* data direction port C */
#define PTE       *(volatile unsigned char *)0x08 /* port E */
#define PTF       *(volatile unsigned char *)0x09 /* port F */
#define DDRE      *(volatile unsigned char *)0x0c /* data direction port E */
#define DDRF      *(volatile unsigned char *)0x0d /* data direction port F */

/*    SPI --UPDATED
 */
#define SPCR      *(volatile unsigned char *)0x44 /* SPI control register */
#define SPSCR     *(volatile unsigned char *)0x45 /* SPI control/status register
*/
#define SPDR      *(volatile unsigned char *)0x46 /* SPI data register */

/*    SCI --UPDATED
 */
#define SCC1      *(volatile unsigned char *)0x38 /* SCI control register 1 */
#define SCC2      *(volatile unsigned char *)0x39 /* SCI control register 2 */
#define SCC3      *(volatile unsigned char *)0x3a /* SCI control register 3 */
#define SCS1      *(volatile unsigned char *)0x3b /* SCI status register 1 */
#define SCS2      *(volatile unsigned char *)0x3c /* SCI status register 2 */
#define SCDR      *(volatile unsigned char *)0x3d /* SCI data register */
#define SCBR      *(volatile unsigned char *)0x3e /* SCI baud rate */

/*    PWM  --UPDATED
 */
#define PCTL1     *(volatile unsigned char *)0x20 /* PWM control register 1 */
#define PCTL2     *(volatile unsigned char *)0x21 /* PWM control register 2 */
#define PWMOUT    *(volatile unsigned char *)0x25 /* PWM Output control register
*/
#define PCNT      *(volatile unsigned int *)0x26 /* PWM counter register */
#define PCNTH     *(volatile unsigned char *)0x26 /* PWM counter register high*/
#define PCNTL     *(volatile unsigned char *)0x27 /* PWM counter register low*/
#define PMOD      *(volatile unsigned int *)0x28 /* PWM counter modulo
register*/
```

```
#define PMODH     *(volatile unsigned char *)0x28 /* PWM counter modulo register
high*/
#define PMODL     *(volatile unsigned char *)0x29 /* PWM counter modulo register
low*/
#define PVAL1     *(volatile unsigned int *)0x2a /* PWM1 value register */
#define PVAL1H    *(volatile unsigned int *)0x2a /* PWM1 value register high*/
#define PVAL1L    *(volatile unsigned int *)0x2b /* PWM1 value register low*/
#define PVAL2     *(volatile unsigned int *)0x2c /* PWM2 value register */
#define PVAL2H    *(volatile unsigned int *)0x2c /* PWM2 value register high*/
#define PVAL2L    *(volatile unsigned int *)0x2d /* PWM2 value register low*/
#define PVAL3     *(volatile unsigned int *)0x2e /* PWM3 value register */
#define PVAL3H    *(volatile unsigned int *)0x2e /* PWM3 value register high*/
#define PVAL3L    *(volatile unsigned int *)0x2f /* PWM3 value register low*/
#define PVAL4     *(volatile unsigned int *)0x30 /* PWM4 value register */
#define PVAL4H    *(volatile unsigned int *)0x30 /* PWM4 value register high*/
#define PVAL4L    *(volatile unsigned int *)0x31 /* PWM4 value register low*/
#define PVAL5     *(volatile unsigned int *)0x32 /* PWM5 value register */
#define PVAL5H    *(volatile unsigned int *)0x32 /* PWM5 value register high*/
#define PVAL5L    *(volatile unsigned int *)0x33 /* PWM5 value register low*/
#define PVAL6     *(volatile unsigned int *)0x34 /* PWM6 value register */
#define PVAL6H    *(volatile unsigned int *)0x34 /* PWM6 value register high*/
#define PVAL6L    *(volatile unsigned int *)0x35 /* PWM6 value register low*/
#define DEADTM    *(volatile unsigned int *)0x36 /* Dead-time Write-once
register */
#define DISMAP    *(volatile unsigned int *)0x37 /* PWM disable mapping Write-
once register */

/*    FAULT
 */
#define FCR       *(volatile unsigned char *)0x22 /* Fault control register */
#define FSR       *(volatile unsigned char *)0x23 /* Fault status register */
#define FTACK     *(volatile unsigned char *)0x24 /* Fault acknowledge register
*/


/*    CONFIG --UPDATED
 */
#define ISCR      *(volatile unsigned char *)0x3F /* IRQ status/control register
*/
#define CONFIG    *(volatile unsigned char *)0x1F /* configuration register */


/*    TIMER A  --UPDATED
 */
#define TASC      *(volatile unsigned char *)0x0e /* timer A status/ctrl
register */
#define TACNT     *(volatile unsigned int *)0x0f /* timer A counter register */
#define TACNTH    *(volatile unsigned char *)0x0f /* timer A counter high */
#define TACNTL    *(volatile unsigned char *)0x10 /* timer A counter low */
#define TAMOD     *(volatile unsigned int *)0x11 /* timer A modulo register */
#define TAMODH    *(volatile unsigned char *)0x11 /* timer A modulo high */
#define TAMODL    *(volatile unsigned char *)0x12 /* timer A modulo low */
#define TASC0     *(volatile unsigned char *)0x13 /* timer A chan 0 status/ctrl
*/
#define TACH0     *(volatile unsigned int *)0x14 /* timer A chan 0 register */
#define TACH0H    *(volatile unsigned char *)0x14 /* timer A chan 0 high */
#define TACH0L    *(volatile unsigned char *)0x15 /* timer A chan 0 low */
#define TASC1     *(volatile unsigned char *)0x16 /* timer A chan 1 status/ctrl
*/
```

```
#define TACH1     *(volatile unsigned int *)0x17 /* timer A chan 1 register */
#define TACH1H    *(volatile unsigned char *)0x17 /* timer A chan 1 high */
#define TACH1L    *(volatile unsigned char *)0x18 /* timer A chan 1 low */
#define TASC2     *(volatile unsigned char *)0x19 /* timer A chan 2 status/ctrl
*/
#define TACH2     *(volatile unsigned int *)0x1a /* timer A chan 2 register */
#define TACH2H    *(volatile unsigned char *)0x1a /* timer A chan 2 high */
#define TACH2L    *(volatile unsigned char *)0x1b /* timer A chan 2 low */
#define TASC3     *(volatile unsigned char *)0x1c /* timer A chan 3 status/ctrl
*/
#define TACH3     *(volatile unsigned int *)0x1d /* timer A chan 3 register */
#define TACH3H    *(volatile unsigned char *)0x1d /* timer A chan 3 high */
#define TACH3L    *(volatile unsigned char *)0x1e /* timer A chan 3 low */

/*    TIMER B  --UPDATED
 */
#define TBSC      *(volatile unsigned char *)0x51 /* timer B status/ctrl
register */
#define TBCNT     *(volatile unsigned int *)0x52 /* timer B counter register */
#define TBCNTH    *(volatile unsigned char *)0x52 /* timer B counter high */
#define TBCNTL    *(volatile unsigned char *)0x53 /* timer B counter low */
#define TBMOD     *(volatile unsigned int *)0x54 /* timer B modulo register */
#define TBMODH    *(volatile unsigned char *)0x54 /* timer B modulo high */
#define TBMODL    *(volatile unsigned char *)0x55 /* timer B modulo low */
#define TBSC0     *(volatile unsigned char *)0x56 /* timer B chan 0 status/ctrl
*/
#define TBCH0     *(volatile unsigned int *)0x57 /* timer B chan 0 register */
#define TBCH0H    *(volatile unsigned char *)0x57 /* timer B chan 0 high */
#define TBCH0L    *(volatile unsigned char *)0x58 /* timer B chan 0 low */
#define TBSC1     *(volatile unsigned char *)0x59 /* timer B chan 1 status/ctrl
*/
#define TBCH1     *(volatile unsigned int *)0x5a /* timer B chan 1 register */
#define TBCH1H    *(volatile unsigned char *)0x5a /* timer B chan 1 high */
#define TBCH1L    *(volatile unsigned char *)0x5b /* timer B chan 1 low */

/*    PLL --UPDATED
 */
#define PCTL      *(volatile unsigned char *)0x5c /* PLL control register */
#define PBWC      *(volatile unsigned char *)0x5d /* PLL bandwidth control
register*/
#define PPG       *(volatile unsigned int *)0x5e /* PLL programing register */


/*    ANALOG/DIGTAL --UPDATED
 */
#define ADSCR     *(volatile unsigned char *)0x40 /* A/D status/ctrl register */
#define ADR       *(volatile unsigned int *)0x41 /* A/D data register */
#define ADRH      *(volatile unsigned char *)0x41 /* A/D data register high */
#define ADRL      *(volatile unsigned char *)0x42 /* A/D data register high */
#define ADCLK     *(volatile unsigned char *)0x43 /* A/D input clock register */

/*    SIM --UPDATED
 */
#define SBSR      *(volatile unsigned char *)0xfe00 /* SIM break status register
*/
#define SRSR      *(volatile unsigned char *)0xfe01 /* SIM reset status register
*/
#define SBFCR     *(volatile unsigned char *)0xfe03 /* SIM break control
register */
```

```
#define FLCR      *(volatile unsigned char *)0xfe08 /* FLASH control register */
#define BRK       *(volatile unsigned int *)0xfe0c /* BREAK address register */
#define BRKH      *(volatile unsigned char *)0xfe0c /* BREAK address register
low */
#define BRKL      *(volatile unsigned char *)0xfe0d /* BREAK address register
high */
#define BRKSCR    *(volatile unsigned char *)0xfe0e /* BREAK status/ctrl
register */
#define LVICR     *(volatile unsigned char *)0xfe0f /* LVI control register */
#define FLBPR     *(volatile unsigned char *)0xff7e /* FLASH block protect
register */
#define COPCTL    *(volatile unsigned char *)0xffff /* COP control register */

****************************************************************************

//Avectors.h

//HC08 Vector File

extern void timerB_overflow_handler (void);
extern void external_handler (void);
extern void SCIrecieve(void);



#pragma interrupt_handler isrDummy

void isrDummy(void)
      {
      }

#pragma abs_address:0xffd2

void (* const _vectab[])(void) = {
      isrDummy,               /* SCI transmit   0xffd2*/
      SCIrecieve,             /* SCI recieve    0xffd4*/
      isrDummy,               /* SCI error      0xffd6*/
      isrDummy,               /* SPI transmit   0xffd8*/
      isrDummy,               /* SPI recieve    0xffda*/
      isrDummy,               /* A/D      0xffdc*/
      timerB_overflow_handler,            /* TIMB OVERFLOW  0xffde*/
      isrDummy,               /* TIMB channel 1  0xffe0*/
      isrDummy,               /* TIMB channel 0  0xffe2 Servo output*/
      isrDummy,               /* TIMA OVERFLOW  0xffe4*/
      isrDummy,               /* TIMA channel 3  0xffe6*/
      isrDummy,               /* TIMA channel 2  0xffe8*/
      isrDummy,               /* TIMA channel 1  0xffea*/
      isrDummy,               /* TIMA channel 0  0xffec*/
      isrDummy,               /* PWMMC        0xffee*/
      isrDummy,               /* FAULT 4        0xfff0*/
      isrDummy,               /* FAULT 3        0xfff2*/
      isrDummy,               /* FAULT 2        0xfff4*/
      isrDummy,               /* FAULT 1        0xfff6*/
      isrDummy,               /* PLL          0xfff8*/
      external_handler,             /* IRQ      0xfffa*/
      isrDummy                      /* SWI      0xfffc            */
/*    RESET defined in crt08.o */
      };
#pragma end_abs_address
```

```
*****************************************************************************



// Iojk3.h

/*    IO DEFINITIONS FOR MC68HC908JK3  */

#define _IO_BASE 0
#define _REGISTER(a) *(volatile unsigned char *)(a)
#define _BIT(a,b) (((vbitfield *)(_IO_BASE + a))->_BIT##b)
#define UP          1
#define DOWN        0
#define ON         1
#define OFF        0
#define SET        1
#define RESET      0
#define CLEAR      0
#define ENABLED    1
#define DISABLED 0
#define OUTPUT     1
#define INPUT      0

#define ENABLE_INTERRUPTS      asm("cli");
#define DISABLE_INTERRUPTS     asm("sei");
/*    PORTS
 */
#define PTA                    _REGISTER(0x00)
#define PTA_BIT0          _BIT(0x00,0)
#define PTA_BIT1          _BIT(0x00,1)
#define PTA_BIT2          _BIT(0x00,2)
#define PTA_BIT3          _BIT(0x00,3)
#define PTA_BIT4          _BIT(0x00,4)
#define PTA_BIT5          _BIT(0x00,5)
#define PTA_BIT6          _BIT(0x00,6)

#define PTB                     _REGISTER(0x01)
#define PTB_BIT0          _BIT(0x01,0)
#define PTB_BIT1          _BIT(0x01,1)
#define PTB_BIT2          _BIT(0x01,2)
#define PTB_BIT3          _BIT(0x01,3)
#define PTB_BIT4          _BIT(0x01,4)
#define PTB_BIT5          _BIT(0x01,5)
#define PTB_BIT6          _BIT(0x01,6)
#define PTB_BIT7          _BIT(0x01,7)

#define PTD                     _REGISTER(0x03)
#define PTD_BIT0          _BIT(0x03,0)
#define PTD_BIT1          _BIT(0x03,1)
#define PTD_BIT2          _BIT(0x03,2)
#define PTD_BIT3          _BIT(0x03,3)
#define PTD_BIT4          _BIT(0x03,4)
#define PTD_BIT5          _BIT(0x03,5)
#define PTD_BIT6          _BIT(0x03,6)
#define PTD_BIT7          _BIT(0x03,7)
```

```
#define DDRA                    _REGISTER(0x04)
#define DDRA_BIT0               _BIT(0x04,0)
#define DDRA_BIT1               _BIT(0x04,1)
#define DDRA_BIT2               _BIT(0x04,2)
#define DDRA_BIT3               _BIT(0x04,3)
#define DDRA_BIT4               _BIT(0x04,4)
#define DDRA_BIT5               _BIT(0x04,5)
#define DDRA_BIT6               _BIT(0x04,6)

#define DDRB                          _REGISTER(0x05)
#define DDRB_BIT0               _BIT(0x05,0)
#define DDRB_BIT1               _BIT(0x05,1)
#define DDRB_BIT2               _BIT(0x05,2)
#define DDRB_BIT3               _BIT(0x05,3)
#define DDRB_BIT4               _BIT(0x05,4)
#define DDRB_BIT5               _BIT(0x05,5)
#define DDRB_BIT6               _BIT(0x05,6)
#define DDRB_BIT7               _BIT(0x05,7)

#define DDRD                          _REGISTER(0x07)
#define DDRD_BIT0               _BIT(0x07,0)
#define DDRD_BIT1               _BIT(0x07,1)
#define DDRD_BIT2               _BIT(0x07,2)
#define DDRD_BIT3               _BIT(0x07,3)
#define DDRD_BIT4               _BIT(0x07,4)
#define DDRD_BIT5               _BIT(0x07,5)
#define DDRD_BIT6               _BIT(0x07,6)
#define DDRD_BIT7               _BIT(0x07,7)

#define PDCR                          _REGISTER(0x0A)
/*Port A Pull-UP Enable _REGISTER*/
#define PTAPUE                        _REGISTER(0x0D)


/*    KEYBOARD
 */
#define KBSCR     _REGISTER(0x1A)
#define KBIER     _REGISTER(0x1B)

/*    CONFIG
 */

#define INTSCR    _REGISTER(0x1D)
#define CONFIG2   _REGISTER(0x1E)
#define CONFIG1   _REGISTER(0x1F)

/*    TIMER
 */
/* Timer Status and Control _REGISTERs */
#define TSC                   _REGISTER(0x20) /* timer 1 status/ctrl
_REGISTER */
#define TSC_PS0                 _BIT(0x20, 0)
#define TSC_PS1           _BIT(0x20, 1)
#define TSC_PS2           _BIT(0x20, 2)
#define TSC_TRST          _BIT(0x20, 4)
#define TSC_TSTOP         _BIT(0x20, 5)
#define TSC_TOIE          _BIT(0x20, 6)
#define TSC_TOF                 _BIT(0x20, 7)
```

```
/*Timer Counter _REGISTER */
#define TCNTH                    _REGISTER(0x21)
#define TCNTL                    _REGISTER(0x22)

/*Timer Modulo _REGISTER */
#define TMODH                    _REGISTER(0x23)
#define TMODL                    _REGISTER(0x24)

/*Timer Status and Control _REGISTER Channel 0 */

#define TSC0                     _REGISTER(0x25)
#define TSC0_CH0MAX              _BIT(0x25,0)
#define TSC0_TOV0          _BIT(0x25,1)
#define TSC0_ELS0A               _BIT(0x25,2)
#define TSC0_ELS0B               _BIT(0x25,3)
#define TSC0_MS0A          _BIT(0x25,4)
#define TSC0_MS0B          _BIT(0x25,5)
#define TSC0_CH0IE               _BIT(0x25,6)
#define TSC0_CH0F          _BIT(0x25,7)

/*Timer Channel 0 _REGISTER */

#define TCH0H     _REGISTER(0x26)
#define TCH0L     _REGISTER(0x27)

/*Timer Status and Control _REGISTER Channel 1*/

#define TSC1   _REGISTER(0x28)
#define TSC1_CH1MAX                  _BIT(0x28,0)
#define TSC1_TOV1              _BIT(0x28,1)
#define TSC1_ELS1A                   _BIT(0x28,2)
#define TSC1_ELS1B                   _BIT(0x28,3)
#define TSC1_MS1A              _BIT(0x28,4)
#define TSC1_CH1IE                   _BIT(0x28,6)
#define TSC1_CH1F              _BIT(0x28,7)

/*Timer Channel 1 _REGISTER */
#define TCH1H     _REGISTER(0x29)
#define TCH1L     _REGISTER(0x2A)


/*    ANALOG/DIGTAL
 */
#define ADSCR     _REGISTER(0x3C)
#define ADR       _REGISTER(0x3D)
#define ADCLK     _REGISTER(0x3E)

/*    SIM
 */
#define BSR       _REGISTER(0xFE00)
#define RSR       _REGISTER(0xFE01)
#define BFCR      _REGISTER(0xFE03)
#define INT1      _REGISTER(0xFE04)
#define INT2      _REGISTER(0xFE05)
#define INT3      _REGISTER(0xFE06)
#define FLCR      _REGISTER(0xFE08)
#define FLBPR     _REGISTER(0xFE09)
#define BRKH      _REGISTER(0xFE0C)
```

```
#define BRKL      _REGISTER(0xFE0D)
#define BRKSCR  _REGISTER(0xFE0E)

#define COPCTL    _REGISTER(0xFFFF)


typedef volatile struct {
      volatile unsigned int _BIT0 : 1;
      volatile unsigned int _BIT1 : 1;
      volatile unsigned int _BIT2 : 1;
      volatile unsigned int _BIT3 : 1;
      volatile unsigned int _BIT4 : 1;
      volatile unsigned int _BIT5 : 1;
      volatile unsigned int _BIT6 : 1;
      volatile unsigned int _BIT7 : 1;
} vbitfield;

typedef union
{
    struct{
            volatile unsigned char timer_high;
            volatile unsigned char timer_low;
      }count;
      unsigned short timer_count;
}TIMER;

TIMER timer_ch0;
TIMER timer_ch1;

*****************************************************************************

/* vectors08.h
/* Dummy Interrupt Handler
 * Place a Breakpoint here in case you are looking for spurious Interrupts
 */

extern void TIM_1_handler(void);
extern void TIM_0_handler(void);

#pragma interrupt_handler isrDummy
void isrDummy(void)
{
}

#pragma abs_address:0xfff2
void (* const _vectab[])(void) = {

      isrDummy,                     /* TIM Overflow*/
      TIM_1_handler,                      /* TIM Channel1*/
      TIM_0_handler,                      /* TIM Channel0 */
      isrDummy,                     /* NOT USED */
      isrDummy,                     /* IRQ  */
      isrDummy,                     /* SWI */
      /*    RESET defined in crt08.o */
};

#pragma end_abs_address
```

Appendix D – Visual Documentation

This appendix contains a visual representation of our project. A series of photographs documenting our work follows.

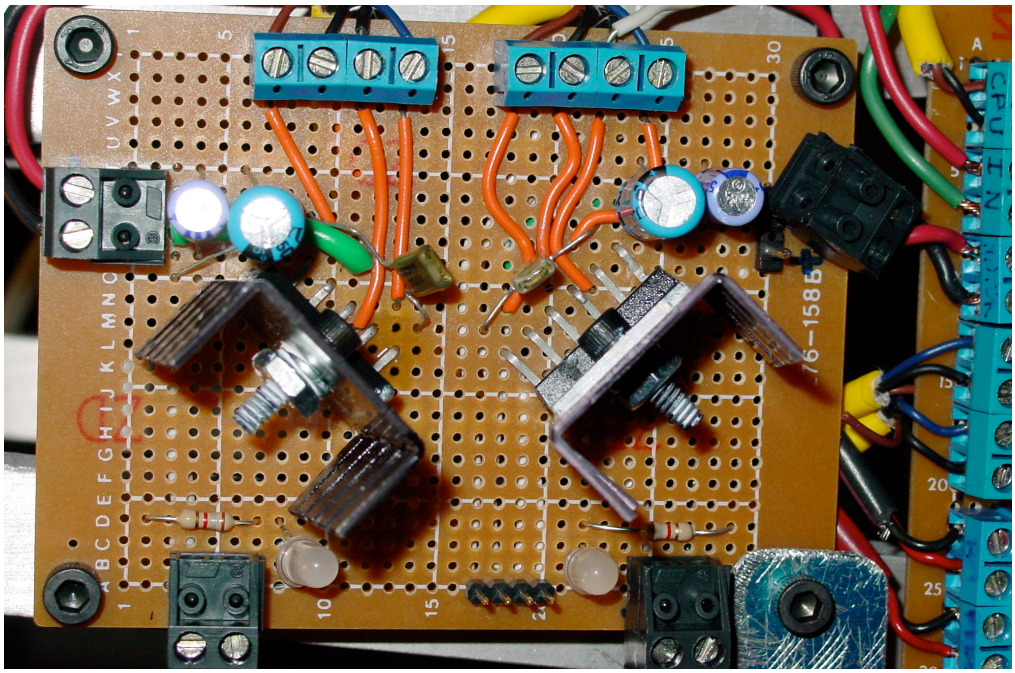


Figure 8 - Motor Driver Circuitry

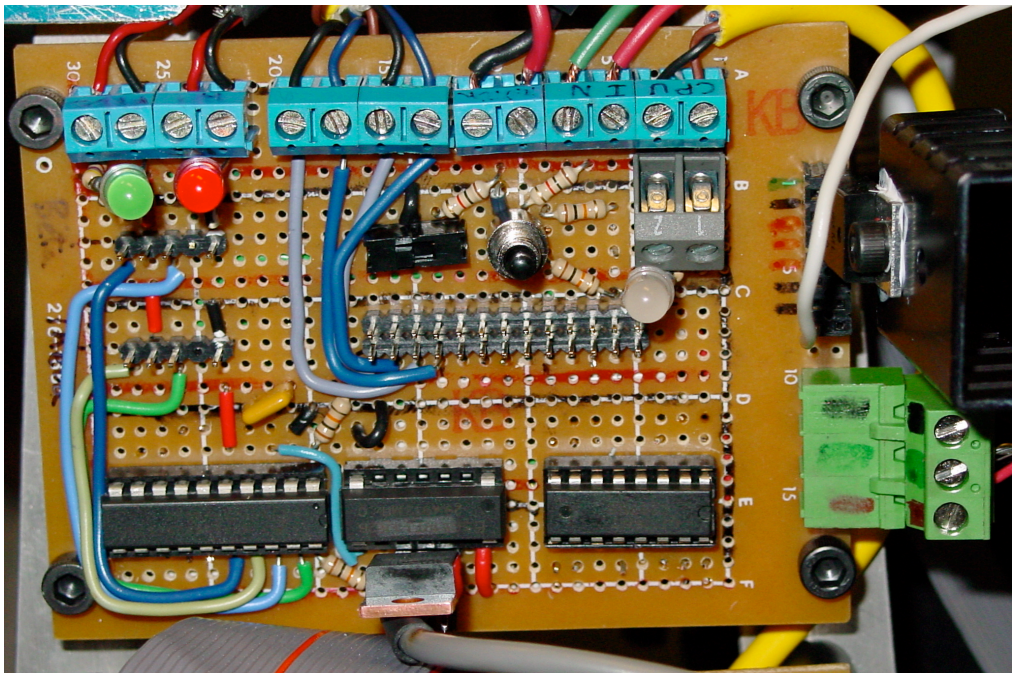


Figure 9 - Interface Circuitry

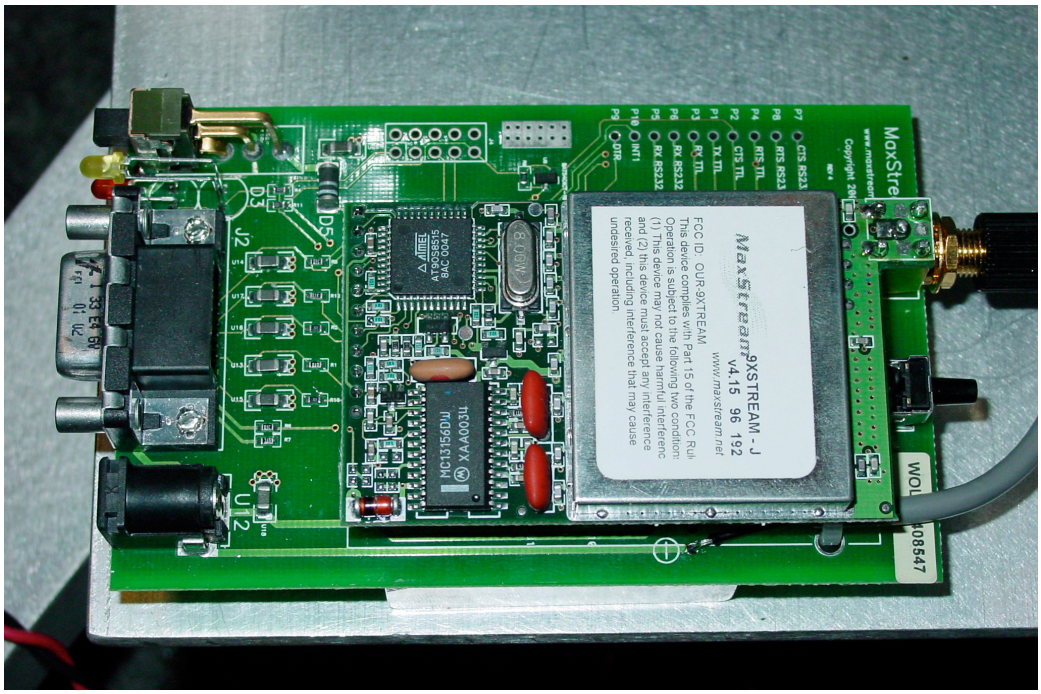


Figure 10 - Wireless Serial PC Board