RoboJoint

Project Addendum

Colin McCarthy
cwm@wpi.edu
9/29/2009

RoboJoint Project Addendum

Contents

1. Introduction

This addendum is in addition to a previously submitted MQP report "Design of Low Cost Modular Robotic Manipulator Joints" (09D044M) submitted May 2009 by J. Baldiga, S. Fitzell, C. McCarthy, and T. Watson. This addendum details additional work on the joint controller, firmware, and user software performed after the submission of the original project. This addendum also elaborates on several sections of the report and includes some test results pertaining to the electrical and control systems that were omitted in the original submission.

2. Changes to Project Deliverables

Due of the scope of the RoboJoint project, the final deliverables included only the critical parts of the planned project. The primary components remain a joint controller and a computer interface. The joint controller went through 3 generations. The first utilized a MSP430 microcontroller interface with a stepper motor. After a decision was made to use a standard DC motor, the next revisions of the controller utilized an H-bridge driver and a simpler PIC microcontroller. The first revision of the PIC based controller utilized a simple serial interface with a PC. Using a terminal emulator, the user could communicate with the controller through a series of prompts, giving the joint the speed and direction in which it should move, and providing feedback of the current location. The third generation utilized a dsPIC, which included an onboard quadrature encoder interface (QEI) module to optimize the operation of the microcontroller, and move the most significant operations to a dedicated peripheral. The dsPIC based controller was again brought to the point where a user could interact with it through a terminal. However, this generation also brought a change in architecture to provide for a USB interface with a PC.

In order to provide a USB interface, it was initially determined that an additional module would be needed in order to communicate with the host PC via USB. This module would then store and forward commands to the individual joint controller. The USB module required customized drivers which became time consuming to develop. Additionally, in order to provide a visual interface for the user, an attempt to develop a control application using Visual C++ was made. The actual development of the Visual C++ application quickly consumed much of the available time for the project's completion and was eventually halted. While the PC application was able to successfully send commands via USB to the base it was never able to receive any usable form of feedback from the microcontrollers. This meant the speed and direction of individual joints could be set, it would not provide the current position and status of the joint, a critical part of the project. Additionally, the USB based prototype never successfully demonstrated control of more than one joint at a time. Because the focus of the project was not the development of the software user interface, but the actual microcontroller development, a new solution had to be found.

In order to simply the project for completion and to meet the critical project requirements, work on a final generation began. This utilized working components from all earlier generations and combined them to bring the project to an effective completion. The PIC microcontroller based controllers had been proven to control the joints and to provide accurate positioning data. As a result, much of this earlier code was utilized and reconfigured to work with a new serial interface. To eliminate the troubles associated with developing a USB controller and PC interface, a USB-to-UART bridge was

utilized. This bridge, manufactured by FTDI, provides serial port emulation over USB to the PC and TTL logic to the microcontroller. As a result, the user interface now had to communicate with a serial protocol as opposed to using a USB interface, greatly reducing the development complexity. Additionally, in order to eliminate the time consuming development and completion of a Visual C++ application, National Instruments LabVIEW was utilized. This rapid software development tool, greatly simplified the creation of a user interface. As a result, the project was able to meet the primary goals of the project.

3. Joint Controller

The final joint controller consisted of a microcontroller, H-Bridge Driver, and voltage regulator. The PIC and logic functions of the Driver IC were powered off of the regulator (specs.), allowing the joint controller to function over a wide range of input voltages. This provides great flexibility to the motor utilized, since the motor drive voltage, off which the regulator is fed, can be widely varied as well. This allows for a sole DC power source to the joint and its controller. The microcontroller utilizes a 4 MHz crystal as a system clock and has full in-circuit programming capability. An interface to the H-bridge driver is provided in the form of a PWM signal. A drive enable provides complete shutdown of the motor and associated drive circuitry, and a standard logic line allows for directional control of the motor.

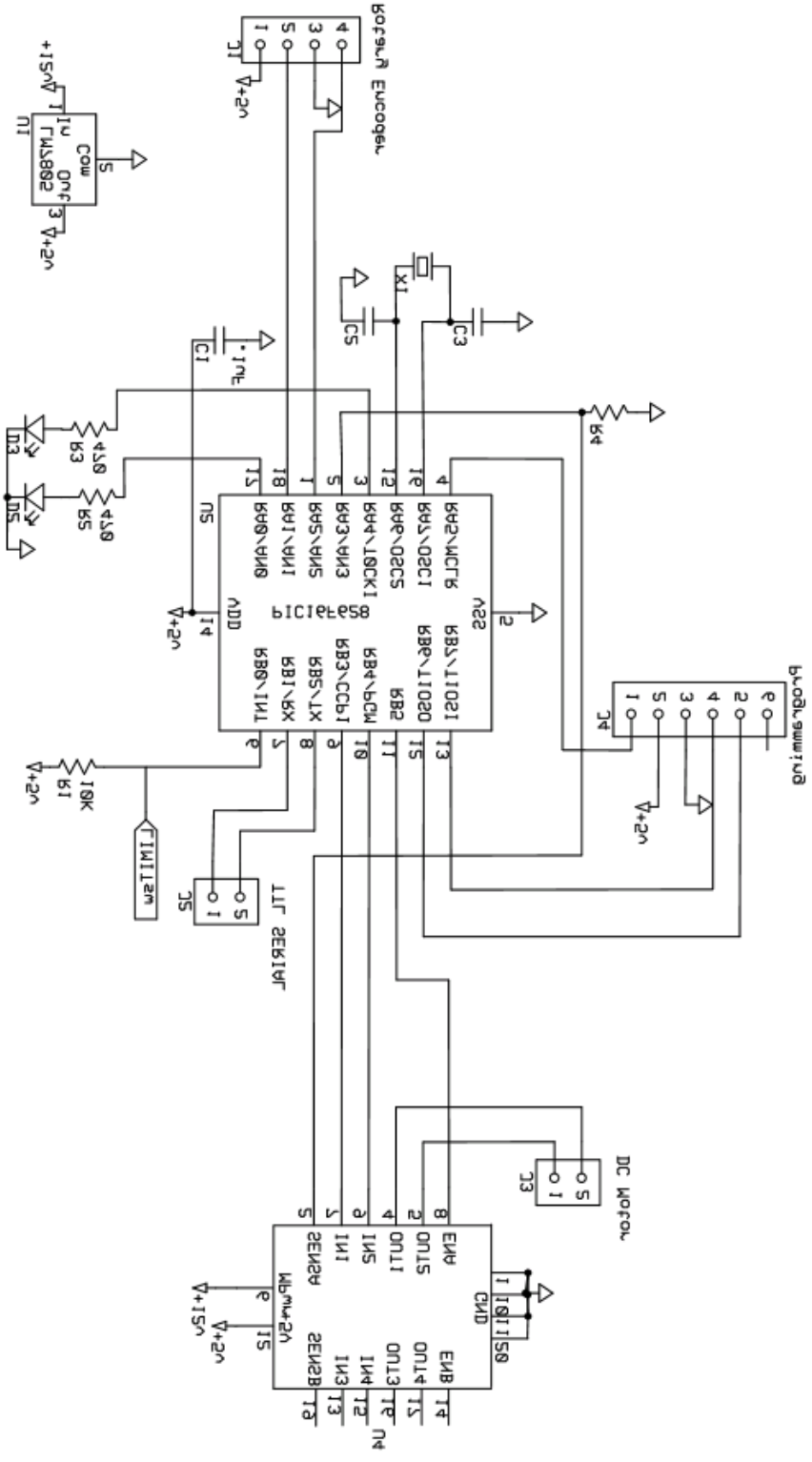

Figure 1: Final Schematic

3.1. 16F688 Microcontroller Code

The PIC microcontroller has 3 portions of code. The main C file, interrupt code, and USART library. The main C source variable initializes the microcontroller, configuring inputs and outputs appropriately and configuring TMR0 to allow for monitoring of the shaft position. An onboard PWM module is configured to provide speed and direction control and an integrated Universal Serial Asynchronous Receiver/Transmitter (USART) to provide a 19200 baud communication link with the PC running the LabVIEW control software. When data destined for the joint is detected an interrupt sets a flag forcing any needed change in the joint's operation, on the next iteration of the program loop. During this update, the speed byte is used to set the duty cycle of the PWM signal. If the command is a request for a status update from the controller, the information is immediately returned to the PC. A command to place the joint into a freerunning mode will result in the direction command being used to appropriately set the drive lines of the H-Bridge driver and result in the subsequent enabling of the driver. Lastly, a positioning command will defer primary control of the joint to the interrupt routine which already is monitoring the position of the joint. In this case, the destination is extracted from the received data and stored into memory onboard the controller for reference by the positioning routine.

Two interrupts are used by the controller. The first monitors the serial data stream for commands. A second is triggered every 512 µS by a peripheral timer. When a valid command for the joint is recognized, the data stream is loaded into a byte array, where a pointer can be used to select a particular element in the array which relates to a particular byte in the received data stream from the PC. The routine ignores and resets the array whenever the hex byte 0xFF is received, effectively using it as a SYNC command. When valid data is loaded into the array, the Update bit is set, causing the main loop to enter into a routine which parses the data and acts appropriately based on the received commands. The Timer0 interrupt is also responsible for the position monitoring of the shaft, utilizing the rotary encoder, whose operation is described in Section 3.2. The interrupt compares the current input state of two photosensors to their previous value. Utilizing a state table the direction of the shaft's rotation can be determined. The integer value holding the position of the joint is incremented or decremented as necessary. This timer is also utilized to monitor the state of the limit switch on the elbow joint. If a low logic condition is detected it will force a shutdown of the joint.

Lastly, a library provided by HITECH, the creator of the compiler used in the project greatly simplifies the use of the USART allowing for common C commands, such as putch(), getch(), and printf() to map to the serial port, and provide easy access to the data. Additional included libraries provide delay and timing routines that are occasionally called upon.

3.2. Optical Encoder Module

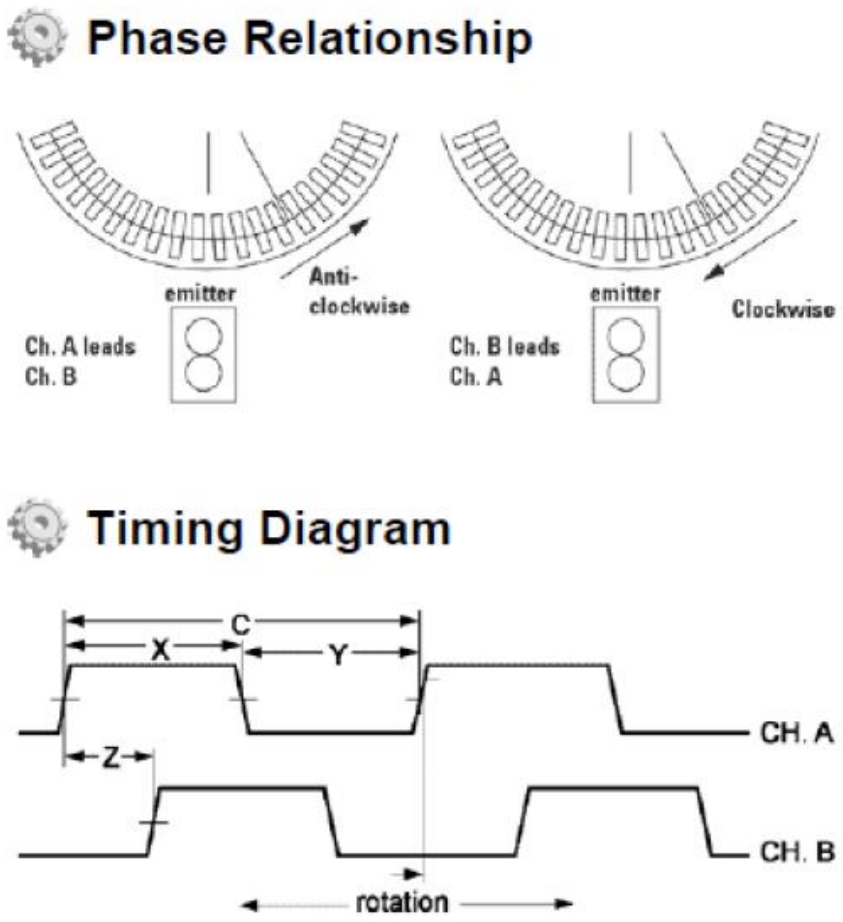

Figure 2: Rotary Encoder

The RoboJoint utilizes off-the-shelf optical encoders to determine the joint's current position. The encoder is made up of a graduated codewheel, fixed to the shaft, and a stationary circuit board with a LED emitter and detector. As the graduations pass over the emitter/detector, a pulse is sent out of the encoder. The encoder is comprised of two channels, which carry pulses corresponding to the detection of black marks and reflective gaps on the rotary wheel, as shown in Figure 2. Based on the phase between these two channels, and determining which channel is leading, it is possible to determine the direction of rotation in the shaft. A timed interrupt constantly monitors the state of these lines, and increments or decrements a position counter accordingly. This position is then accessed elsewhere as needed throughout the program.

3.3. Limit Switch

While the rotation joint was not designed to have any limits on its range of motion, the elbow joint only has the ability to move roughly 180°. For this reason it was necessary to implement a method of shutting

down power to the joint to prevent a collision and damage to the mechanism. While the original intent was to embed miniature switches into the body of the joint, the rapid prototype manufacturing methods made this not feasible. In order to achieve this on the prototype, a copper plate was machined to go on the base of the elbow, which was tied to digital ground. Two additional plates, one on either side of the rotating mechanism where tied to the limit switch input on the controller. When the rotating mechanism nears the base on either side of its 180° sweep, the copper plates make contact triggering the limit input. The input on the controller is tied by a pull-up resistor to V_{CC}. As a result logic high signifies a normal condition, while any short to ground will result in a limit error. As soon as a limit error condition is detected, the controller initiates a shutdown of the drive. It then reverses the drive until the limit condition has been cleared. This eliminates any strain the drive gears or the housing might be under from binding or other forces due to a collision. As soon as the rotating mechanism has been backed off from the housing base, the drive is shutdown until further input from the PC. When queried for its status, the controller will respond it is in a limit error condition until a new command has been received clearing the condition.

The limit switch is also used during the initialization of the joint on power-up. In order to provide additional positioning feedback to the PC, the joint controller determines the extent of its drive before initializing normal operation. It will drive the motor forward until it contacts the limit switch, at which point it will record its position and reverse the motor. Again the motor is driven until contact with the limit switch made. A position measurement is again made, and the joint drive is shutdown. The controller then passes to the PC the measured limits of the elbow mechanism. This can later be used to determine an approximate graphical representation of the joint's position.

4. PC LabVIEW Application

In order to allow rapid development of a user interface, LabVIEW was utilized. The tool uses graphical representations of commands, which can be interconnected to program the desired application. The RoboJoint application, when running constantly monitors the user inputs and forms command strings that are sent to the joint via a serial link. Additionally, the application sends a status request command to each joint, waits for a reply, and parses the data so that it can be displayed for the user. The constructed user interface is shown in Figure 3, and can be packaged into a standalone executable file for distribution and operation on other machines. All that is needed are the drivers for the USB to serial converter.

Figure 4 shows the portion of the application responsible for monitoring user input. A timed loop checks for changes to any of the controls at a frequency of 1kHz. If a button is pressed, or other change is detected, a string of commands is formed. This contains the joint address, speed controls and direction controls. After a string of hexadecimal values is generated, it is passed onto a serial write command, which sends it to the appropriate port, and onto the joint. Figure 5 shows the configuration of the port, and the simplicity of the LabVIEW tool. The configuration of the same port in Visual C++ took over 30 lines and several hours to troubleshoot, while this graphical representation makes it simple to configure and utilize the port.

Figure 3: RoboJoint User Interface

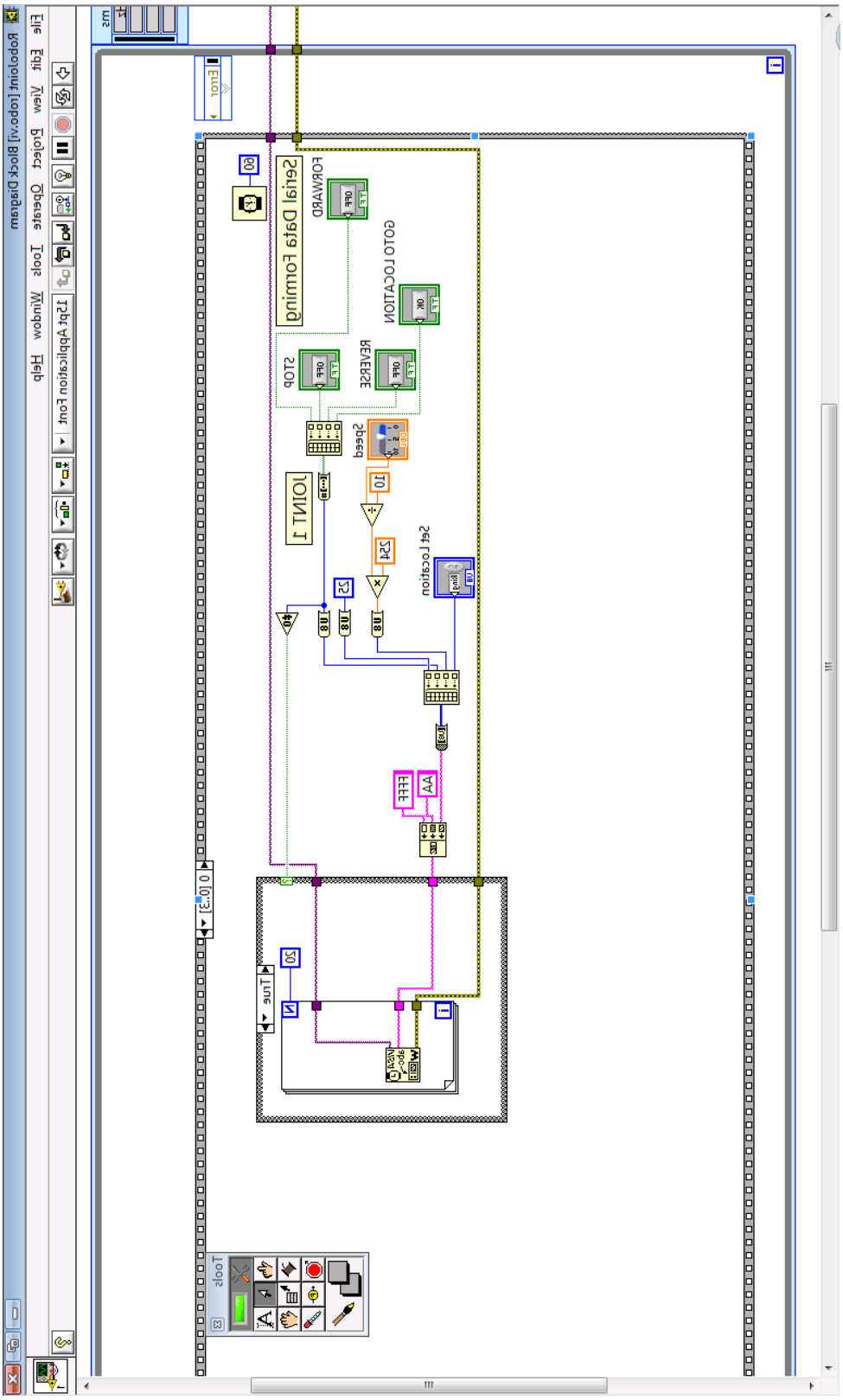

Figure 4: Command Forming

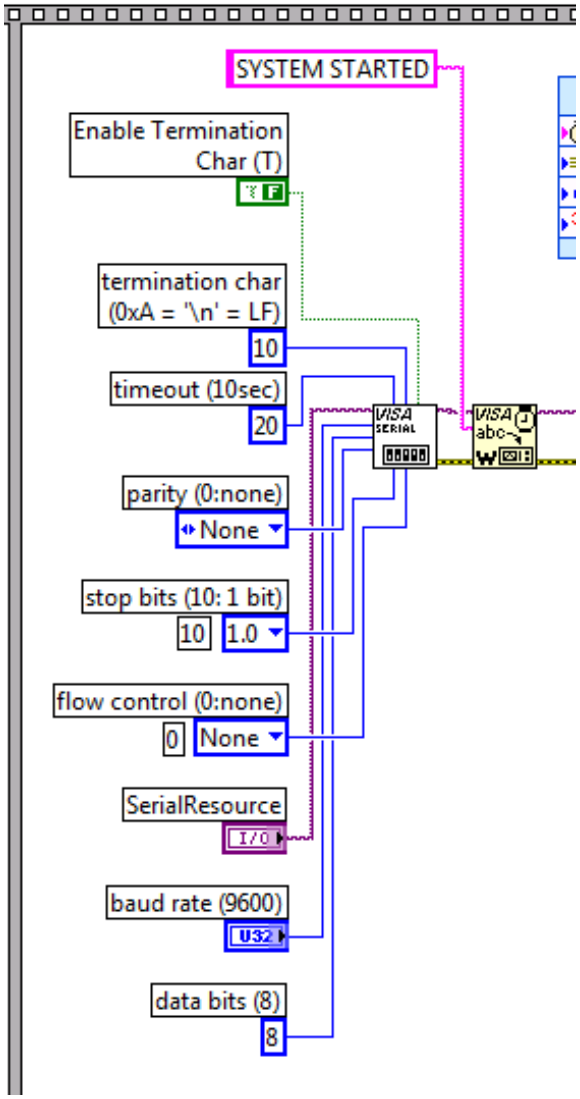

Figure 5: Serial Configuration

An important part of the user interface is to provide the current status and position of the joint. The application obtains this information, by forming a request consisting of the command 0xAB and the joint address, and then waiting for a response. Any response is stored into an array for parsing by the application. A timeout will eventually force the application to move to the next joint if no response is received. Figure 6 shows the creation of the status request, the read command, and the loading of any received data into an array. This received data is then parsed, and if valid displayed for the user. Figure 7 shows a case structure utilized to relay to the user the current mode of operation the joint is in. After receiving the status update, the LabVIEW application shades the button corresponding to the current motion red. The current reported position is extracted from the update, converted to a string and displayed. If the controller is in an error state, as happens when the limit switch is triggered, the status indicator will display any pertinent error messages. In the case of a limit error, the application also flashes the controls of the joint in error.

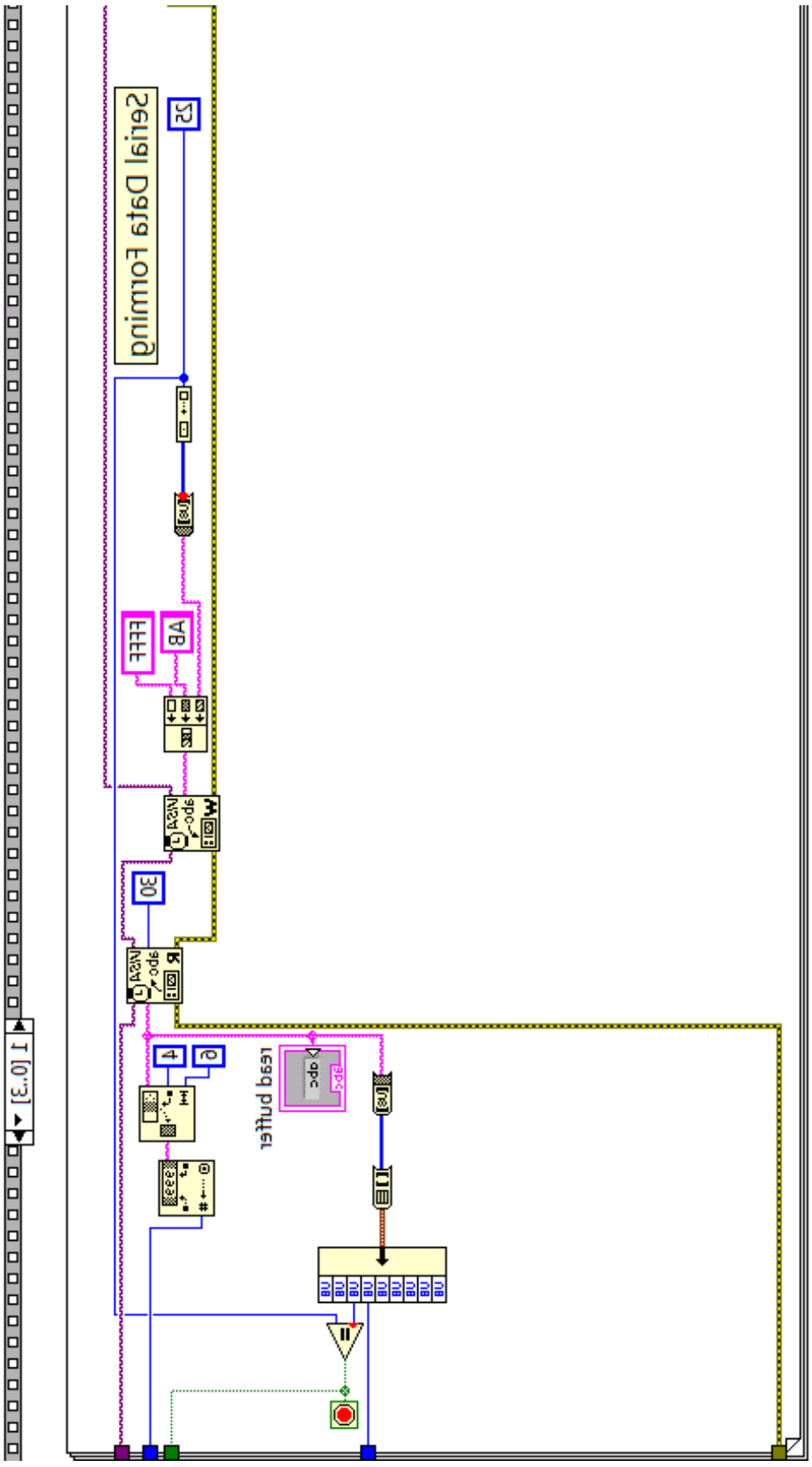

Figure 6: Formation of Status Request

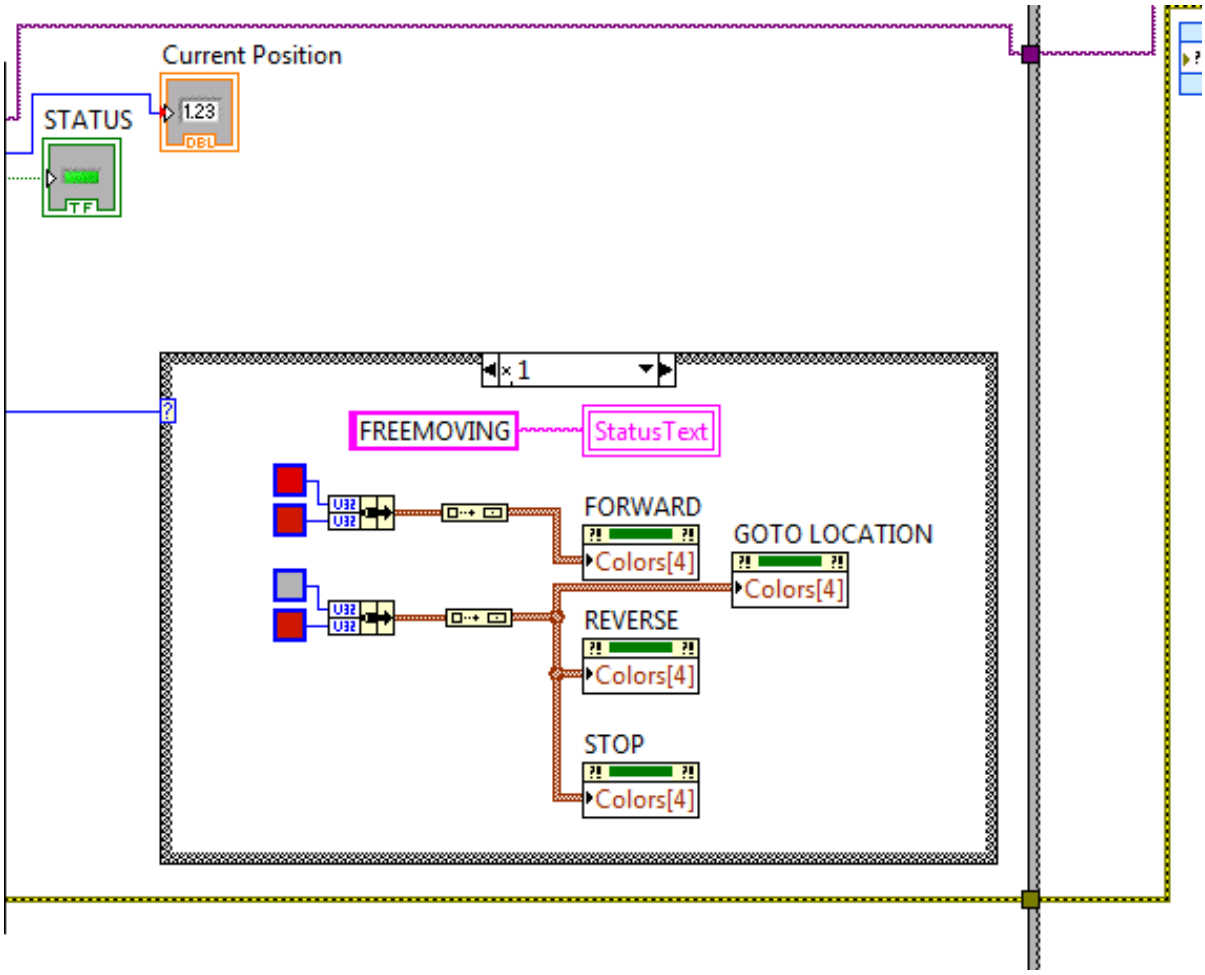

Figure 7: Display of Received Data

An additional feature of the user interface is the graphical representation of the elbow joint's position. One of the reasons for the initialization sequence described in section 3.3 is to determine the full range of the joint. One extreme of its movement is defined as 0, while the other will be set upon traveling the full motion range. This maximum value is sent back to the user interface and used to set the maximum value of the scale on the graphical gauge. The actual position is then represented with the needle on the gauge and the joints position will be represented due to the adjusted scale.

The LabVIEW tool has work very well to achieve the desired user interface for the RoboJoint system. There are some limitations to the tool, however as the primary purpose of the user interface is the testing and demonstration of the joint controllers, it suits the project well. Without this rapid development tool, it would have been exceeding hard to get a working user interface operational within the required time.

4.1.Serial Data Commands

The command structure utilized by the latest generation of controller is fairly straightforward. After a series of SYNC bits, a packet is constructed containing the command, joint address, direction, speed, and location parameter, as shown in Figure 8. For all commands except 0xAB, the packet contains parameters that are to be loaded into the controller to produce the desired function. The address byte is used by each controller to determine if the data should be stored into memory or simply discarded. The direction and speed bytes will be stored into the controller and override any previous settings. The valid direction commands are shown in Figure 9. The speed byte can range from 0x00 to 0xFE, with 0x00 representing 0% and 0xFE running the motor at full power. This value is used by the controller to calculate the duty cycle the motor will be driven with. The location data is used in conjunction with the 0xAC command and simply conveys the location the joint should be moved to.

Upon receiving the 0xAB command, with a valid address, from the PC, the controller forms a response utilizing the same packet structure. The response simply echo's back the values stored in memory. Command, Direction, and Speed will be identical to the last movement command received. The address byte is that of the current joint, and becomes a source identifier, and the Location bytes are used to return the current position of the joint, as determined by the rotary encoder.

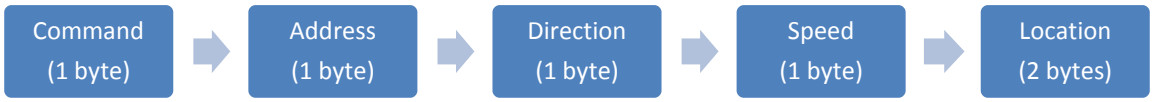

Figure 8: Packet Structure

Command	Function
0xAA	Run
0xAB	Request Status/Position
0xAC	Goto Position
0xAD	Halt

Figure 9: Commands

Direction	Function
0x00	Idle (Command yet to be received)
0x01	Forward
0x02	Stopped
0x04	Reverse
0xEE	Limit Error

Figure 10: Valid Direction Parameter

5. 1st Generation Controller

The initial electrical design consisted of a microcontroller to supervise all functions of the motor control as well as provide an interface to the user's computer. This interface was in the form of RS-232 serial communication to the host PC. At the time, the system was designed to drive a stepper motor, and as a result, a MSP430 microcontroller was chosen, as it had on-board peripherals designed explicitly to interface with servo and stepper motors, as well as a UART capable of serial and IrDA communications (to accommodate the needs of the rotator-joint). The MCU chosen was an MSP430F2132 and had the capability, in addition to IrDA functionality, to provide serial communications using its hardware UART.

Although this would not be an issue for production, when it came to prototyping it proved to be problematic. As a result, the processor was mounted on a surface-mount interface board. A preliminary prototype was assembled to establish serial communications between the host and the control board, and to test the functionality of the IR transceiver. Although work had begun on implementing stepper motor control, it was never completed due to electromechanical design changes which resulted in a change from stepper motors to conventional brushed DC motors.

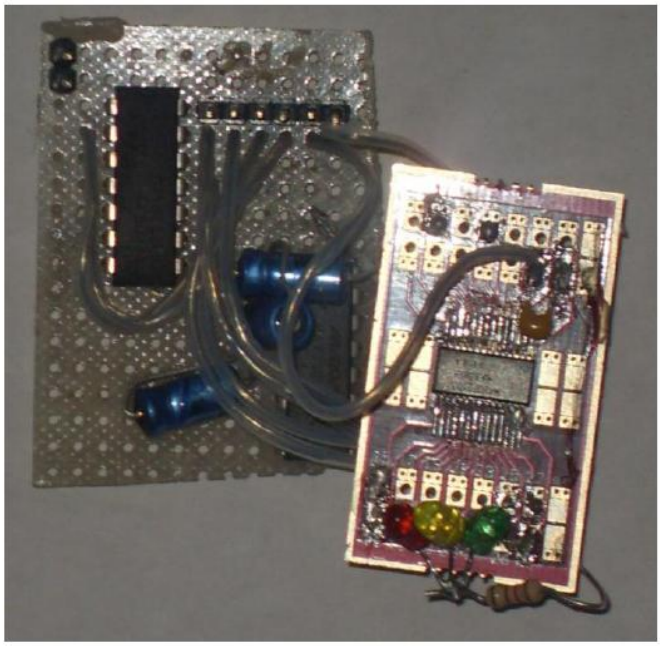

Figure 11: 1st Generation Protoboard

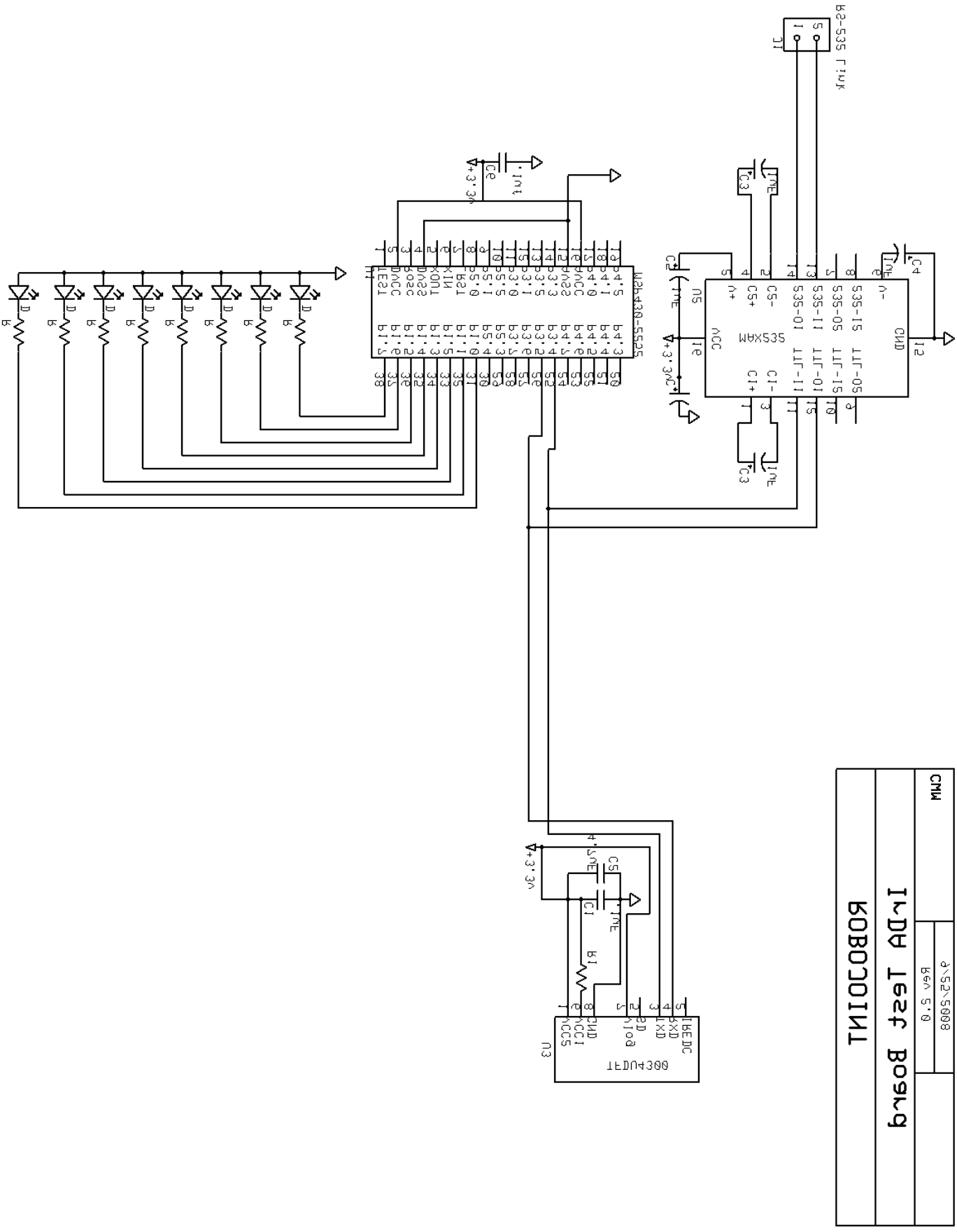

Figure 12: 1st Generation Schematic

6. 2nd Generation Controller

Upon selection of a standard brushed DC motor, it was decided to completely redesign the controller instead of adapting the MSP430 design. Instead, a PIC microcontroller was chosen due to its simplicity and ease of interface with a brushed DC motor. Like the MSP430 design, a serial link was created to communicate commands to the host PC. While the emphasis of the first generation was on communications and creating a reliable method of delivering the commands to the joint module, the next step was to focus on motor control. While the project as a whole focused on communication from the host to a number of joints, for simplicity and to encourage design on the motor controller, this iteration provided the capacity to communicate with a sole joint through an RS-232 interface. Communication with the module was achieved through a serial terminal, which enabled the user to access a text-based prompt system to issue relevant commands for manipulating the joint. This prompt allowed the user to move the joint either forwards or backwards for a set amount of time or to a rough location. The relevant commands were formed from these text responses and executed by the joint.

The PIC microcontroller used in this iteration is a 16F876. This microcontroller is a TTL based device, powered by a 5v regulated source. An PWM (Pulse-Width Modulation) interface integrated into the PIC is utilized to vary the duty cycle of a TTL drive signal. This drive signal is fed to an H-Bridge driver onboard the controller, in this case a SN75441. This H-bridge driver contains a series of MOSFETs which utilizing the TTL input logic, switches the high voltage, high current source provided separately for the purpose of driving the motor. Figure 13 shows a simplified depiction of an H-bridge circuit, while Figure 14 shows the common logic states and their accompanying function.

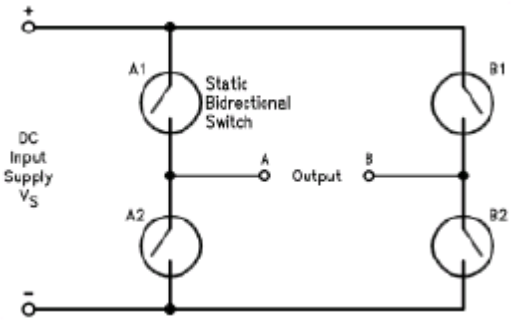

Figure 13: Typical H-Bridge Circuit (National Semiconductor)

C = H ; D = L	Forward
C = L ; D = H	Reverse
C = D	Fast Motor Stop
C = X ; D = X	Free Running Motor Stop

Figure 14: H-Bridge Logic Table (STMicroelectronics)

Figure 15 also shows the external RS232 transceiver which translated the TTL (0volts , 5volts) levels to true RS-232 (-15volts, 15volts) allowing for the circuit's direct interface with the PC.

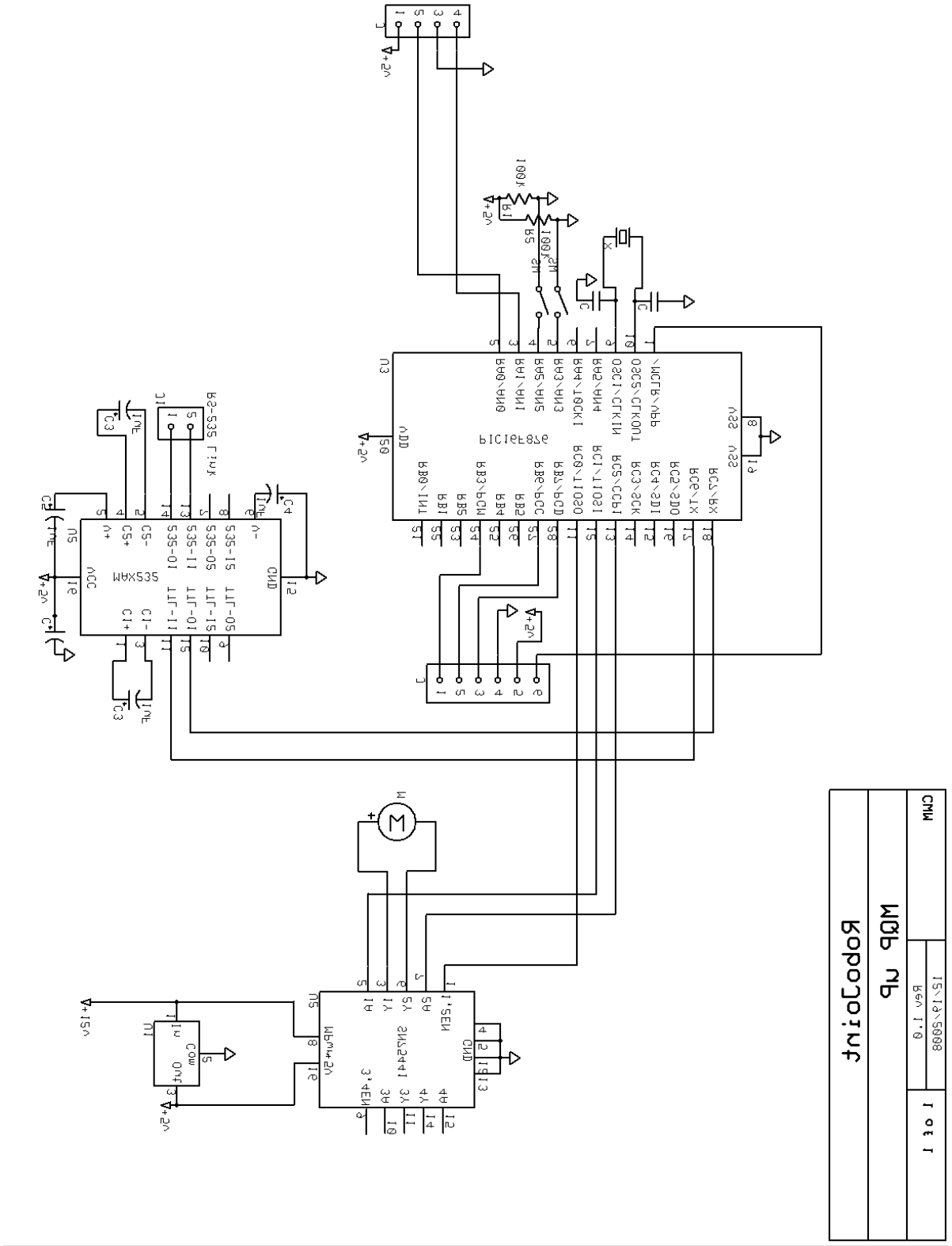

Figure 15: 2nd Generation Schematic

7. 3rd Generation Controller

The 3rd Generation control system was to be made up of two primary components, the base controller and the joint controller, as shown in Figure 16. The base controller served as a link between a PC and the RoboJoint system. Receiving commands from the PC via a USB connection, it translates the instructions from the PC into commands the joint controllers can interpret. Additionally, the base controller kept track of all joint controllers' operations and status, allowing the PC to poll for updated data from the controller when necessary. After reviewing the results of Iteration 2, especially in regards to the terminal based user interface, it was decided that it would be important to provide the user with an easily operable interface for testing and demonstration. This was attempted using Visual C++, but eventually would be completed with LabVIEW. The program was to communicate directly with the base controller, which would then relay the commands to the appropriate joint. Joint controllers, which reside in each individual joint, provide the actual motion functionality. An onboard microcontroller controls motor speed and direction, maintains a record of the current shaft position, monitors the status of necessary limit switches and provides two-way communication back to the controller. These controllers are designed to work with both the elbow and rotation joints. In order to accommodate the differences between the joints, a jumper must be set on the PC board, which will trigger the embedded software to react appropriately.

The interrupt also controls driving the joint to a known location. Because it has already determined the position of the joint, via the shafts cumulative rotation, the interrupt simply compares the current and desired position, and spins the shaft either clock or counterwise to try to force the two integers to be equal. Ideally the move would stop when the position difference is 0. However, because of the low pitch of the markings on the encoder shaft, it is extremely hard to precisely spin the shaft to the desired location. To combat this, it is assumed the shaft will usually over or undershoot the destination. As a result the rotation of the shaft will not stabilize on the correct position, but oscillate around it. To combat this, an additional counter monitors the number of times the joint moves past the destination and when a nominal amount of crossings have been made, the controller assumes the shaft is relatively close and will not be able to reliably move any closer to the desired position. Finally, the interrupt constantly monitors the limit switch input, which represents an error condition with the joint necessitating a reset or intervention.

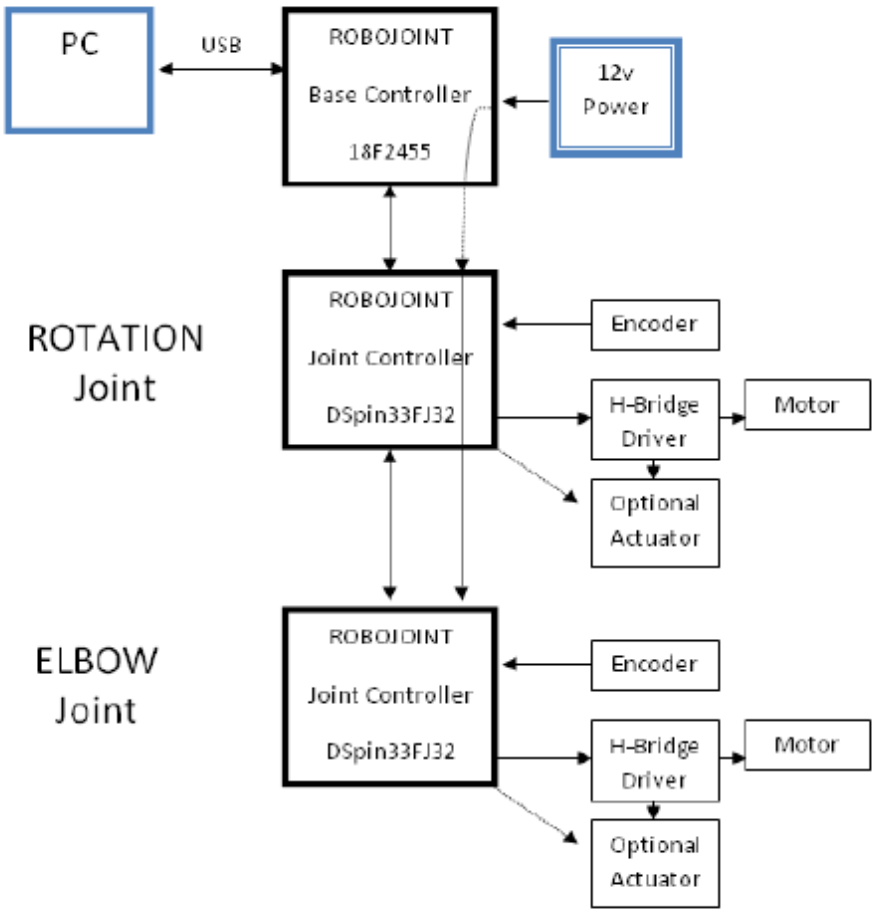

Figure 16: 3rd Generation System Outline

In addition to the microcontroller, a dual-channel H-bridge driver resides on each board. Both channels of this driver can be utilized by the controller. A set of header pins provide connectivity to the both the output and input of this second channel to any peripherals that may need it. As a result it is possible to use the controller board to drive an end effector. Additionally, these pins could potentially be used as logic level input/outputs, to either activate relays or interface with feedback systems, such as additional limit switches.

7.1.Communications

Please refer to section 6.3.2 (Communications) of the submitted MQP report for additional details on the design of the 3rd generation system.

7.2.Slip-Joint / IrDA Link

One of the unique ideas originally part of the 3rd generation system was an integrated slip joint to allow for continuous rotation while providing power across the joint. Additionally, an IrDA communications link was installed to maintain reliable communications as the joint was rotated.

7.2.1. Original IrDA Design

In order to reduce the cost of the joint, it was decided that an infrared data link would be used to transmit data across the infinite rotation joint. While it would be possible to integrate a slip-ring device capable of transmitting power and signal through the shaft, the cost or these devices grows as more conductors are added and higher signal integrity is needed. Instead of purchasing a high-quality 4-conductor device, capable of transmitting both power and signal, the design could utilize a cheaper 2-conductor part. Additionally, by only transmitting power, the device does not need to be of high quality, as voltage regulators and capacitive filtering on each board will be able to flatten out any spikes or dropouts in the power supply. The communication link which would be very sensitive to any interruptions is then provided through the use of IR transceivers, integrated into the mechanical package of the RoboJoint. The TFDU4300 Infrared Transceiver was chosen due to its small size and high modularity. Capable of an 115.2kbps transmission rate, the module is fully IrDA compliant. As a result, it can be easily integrated into designs using a wide range of microcontrollers, such as the MSP430 and Microchip PICs.

The TFDU4300 at its core, is little more than an IR LED and IR photo detector. It takes logic bits and triggers the LED accordingly and does not have any processing ability of its own. Because of this, serial data streams must be processed before arriving at the device. The IrDA specifications call for the nominal pulse to be T/*12*, where T is the duration of a typical UART bit at that transmission rate. During testing, a 9600 bits/second transmission rate was used. This equates to a bit duration of 104 µS and a resulting IrDA pulse of approximately 19.5 µS. Further calculations, referenced by the IrDA specifications, detail that the pulse length can deviate to a minimum of 1.41 µS and maximum of 22.13 µS and still be considered valid. A diagram comparing a typical UART serial stream to its IrDA compliment is shown in Figure 17. It is also important to note that pulses only occur for logic lows, effectively inverting the signal.

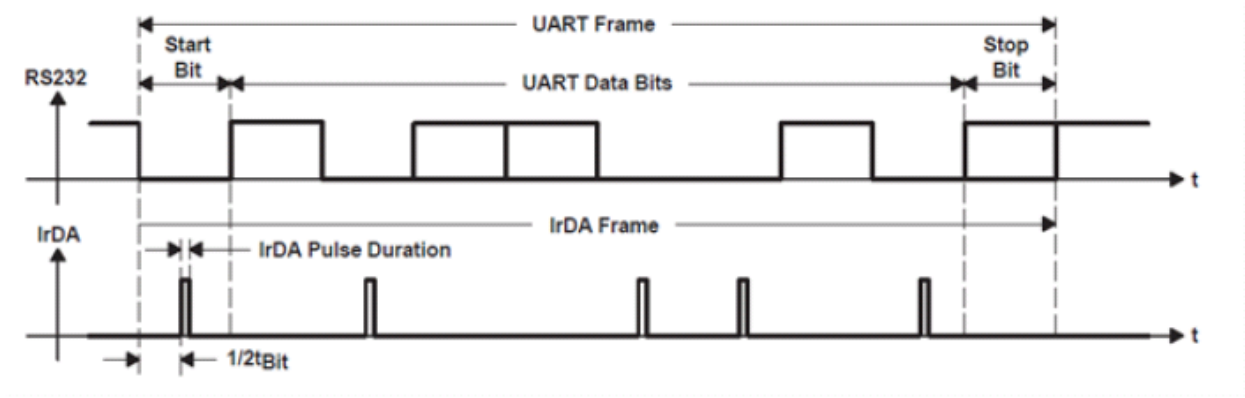

Figure 17: IrDA Protocol (Texas Instruments)

7.2.2. Results

The utilization of the TFDU4300 IR Transceiver began early in the design phase, appearing in the first design iteration. Initial testing consisted of using the MSP430 microcontrollers outfitting with an IR link to toggle LEDs, and progressed to testing the rotary joints' integrated transceivers. The TFDU4300 module proved to be a simple device to use and integrate with both the early MSP4300 microcontroller and the final dsPIC used on the controller board.

The IrDA link was tested in conjunction with the slip joint, but independent of the rest of the system. Tests showed that the link functioned reliably as the joint was rotated. The IR transceivers were simply connected to a 5V supply as well, with one side additionally being attached to a function generator producing a 1 kHz sine wave. The output of the device being utilized as a receiver was connected to an oscilloscope to verify the integrity of the signal. The two plates were rotated, stopping at 4 key points to capture the waveform. The signal did not appear to be affected by the rotation at any point.

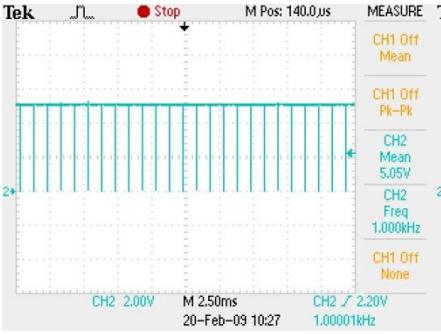

Figure 18: IR Transceivers over each other

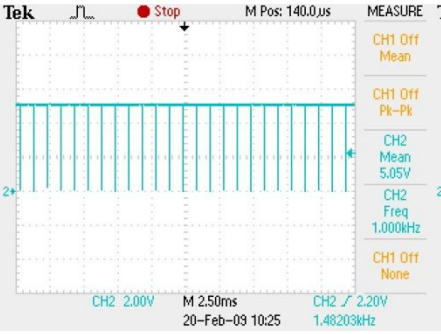

Figure 19: IR Transceivers 180° apart

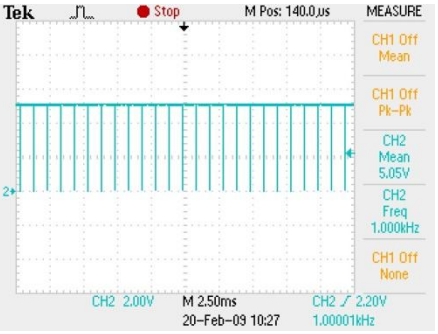

Figure 20: IR Transceivers 90° apart

Many designs for the slip joint were tested and compared. The design utilized, is described in detail in Section 6.2 of the original MQP report. Testing under controlled conditions showed that the slip joint

design might be feasible with better materials and manufacturing. As soon as any significant load was added, the power transfer became extremely unreliable. Figure 21 shows the result of simple testing. A noticeable dropout occurs when the direction of the joint is changed. When contact between the plates was maintained, there was a relatively clean power supply, however because the slip joint was fairly unreliable, it was eliminated from the project.

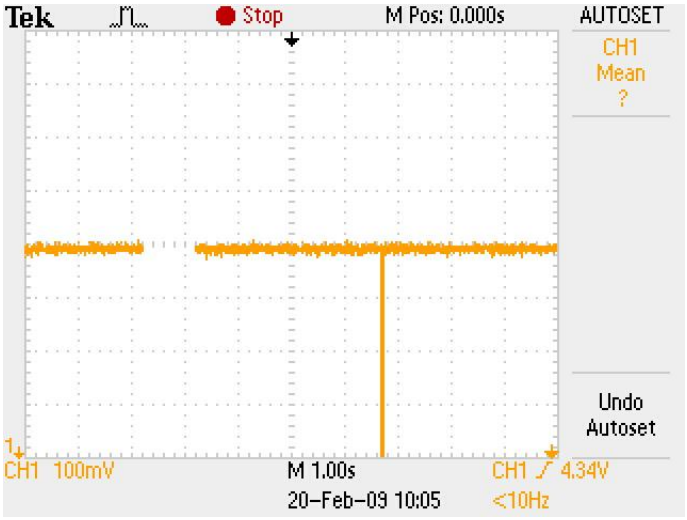

Figure 21: Power Transmission over 1 second

8. Conclusion

It is clear that the original scope of the project was too large to have been successfully completed in the given time period. While several aspects of the project where tested, their integration into one system proved to be unwieldy. As a result a clearer set of requirements and deliverables had to be defined. At its core, the RoboJoint project was to provide a modular joint platform. On the controller side, any electronics had to be small, yet powerful enough to drive a decent sized motor, and easily integrated into the joint. A user interface was also sought to provide testing and demonstration capabilities of the joints.

The latest results of this project have yielded a simple controller, which can be easily mounted inside a joint, and allow for integral monitoring of position via a COTS optical encoder. The controller utilizes an architecture that is expandable to 255 devices, with no change to the controller other than addressing. An H-bridge driver can provide up to 4 amps of DC power, allowing for the attachment of a wide variety of motors, and allow for precise speed and motion control. A simple serial interface can easy be interfaced with a number of wired and wireless protocols to provide functionality in a number of different environments. Lastly a small part count makes the controller small and cheap to build.

The provided software allows the user to operate the joints and demonstrate their controller's positioning and feedback ability. While the software is not designed to be an end solution, the controllers simple command structure can be integrated into a number of end-user applications

customized to a specific application. Section 10.4 of the submitted MQP report gives more insight to possible applications and features worth future consideration.

Overall, the final deliverables address the most critical design requirements and provide a platform for further improvement and refinement.

Appendix - PIC Code – main.c

```
#include <htc.h>
#include <stdio.h>
#include "usart.h"


#define _XTAL_FREQ  4000000          //Crystal Frequency for timing routines
volatile unsigned char RXhold;       //for Serial RX routine
volatile unsigned char cmds[20];     //Byte array of received data
volatile int ind;
volatile unsigned char LQA;          //QEI states
volatile unsigned char LQB;
volatile unsigned char QA;
volatile unsigned char QB;
volatile int position;               //current Position (0-65536)
unsigned char mode;                  //variable containing current mode state
unsigned char dir;                   //variable containing current direction
unsigned char speed;                 //variable containing current speed
volatile int destination;            //destination position  for goto cmd(0-65536)
volatile unsigned char addr;         //placeholder for joint address
unsigned char update;
volatile bit limit;                  //limit switch trigger
volatile bit home;                   //home routine active
volatile int feedback;               //feedback PID
volatile int limita;                 //placeholder for limit switch
int convert;                         //char to word conversion holder



void main(void)
{
        unsigned char input;
        init_comms();
        limit=0;
        addr=0x19; //DECIMAL 101     //joint Address
        update=0;                    //update flag (TRUE if new command to be processed)
        dir=0x00; //0x00-no cmd recieved
                                     //0x01-forward
                                     //0x02-stop
                                     //0x04-reverse
                                     //0xEE-limit error
        mode=0x00; //MODE=0x00- no command recieved
                        //MODE=0xAA- freerunning
                        //MODE=0xAB- Status Check
                        //MODE=0xAC- position fix
                        //MODE=0xAD- Stopped
                        //MODE=0xAE- Limit Error
```

```
//SETUP SYSTEM VARIABLES

        LQA=0;
        LQB=0;
        PR2=0xff; //SET timer 2 period, timer has prescale of 4, for roughly 1.22kHZ PWM
//      CCPR1L=0xcc; //50% duty cycle...ignore LSBs
        T2CON=0b00000101; //INIT TMR2

        CCP1CON=0b00001100; //PWM MODE ON STARTUP KEEP MOTOR STOPPED
                                CCPR1L=0;
                                RA0=0;
                                RB5=0;



/////////////////INITIALIZATION SEQUENCE/////////////////
                        home=1;
                        RB5=1; //enable drive
                        RA0=0;//MOVE FORWARD @ 50% speed
                        CCPR1L=128;
                        while(!limit); //move until limit

                        RA0=1;//MOVE BACKWARDS @ 50% speed

                        while(limit){ //move until limit triggered
                                __delay_ms(1);
                        }
                        limita=position; //save extent as position
                        while(!limit); //move again
                        position=0;//SAVE THE POSITON, use as origin
                        RA0=0;
                        while(limit){
                                __delay_ms(1);
                        }
                        RA0=1; //STOP THE MOTOR
                        RB5=0; //DISABLE DRIVE
                        CCPR1L=0;
                        home=0;
                        __delay_ms(5);



        printf("\rCONTROLLER INITIALIZED\n");String to debug/test serial connection
        //
```

```
while(1){                       //MAIN PROGRAM LOOP

        if(cmds[2]==addr){      // check to see if right joint
                update=1;       //FLAG change in command
                cmds[1]=0;
        }

        if(update){     //start routine to capture new values

                if (cmds[1]==0xAA&&cmds[1]!=0x00){
                        dir=cmds[3];            //retrieve cmds from array
                        speed=cmds[4];
                        mode=cmds[3];
                        convert = cmds[6];      //store intended desitination
                        convert += (cmds[5]<<8);
                        if(dir==0x08){
                                mode=0xAC;
                                dir=0x00;
                                feedback=0;
                        }
                        else
                                mode=0x00;
                }
                if (cmds[1]==0xAB){             //report status to PC
                        putch(0xFF);
                        putch(0xFF);
                        putch(addr);
                        putch(dir);
                        printf("%i",position);
                        putch(0xEF);
                        printf("%i",destination);
                        //printf("SPEED %c",speed);
                }

        //      else if (mode==0xAC)            //for debugging
                //      printf("Freerunning:[%c]",dir);

                update=0;
        }


if(limit&&dir!=0xEE){   //limit error handler
        RB5=1;
        if(RA0)                 //this reverses drive
                RA0=0;
        else
                RA0=1;
        CCPR1L=128;
```

```
                    for(int i=0;i<600;i++){     //move away from extent for about a second
                        if(limit){
                            __delay_ms(5);
                        }
                    }
                //  RA0=0;
                    RB5=0;          //disable drive
                    CCPR1L=0;
                    limit=0;        //clear controller limit error
                    dir=0xEE;       //set controller condition to limit error, to relay to PC
                    mode=0xAD;      //set controller mode to halt
                }
            limit=0; //clear limit condition, will be reset on next interrupt if condition still exists


                if(dir==0x01){          //FORWARD
                    CCPR1L=speed;
                    RA0=0;          //direction toggle
                    RB5=1;          //Joint Enabled
                }
                else if(dir==0x02){     //STOPPED
                    CCPR1L=0;
                    RA0=0;
                    RB5=0;          //Joint Disabled
                }
                else if(dir==0x04){
                    CCPR1L=255-speed;       //Invert PWM, direction inversion from RA0
                    RA0=1;
                    RB5=1;          //joint enabled
                }


//DEBUG STREAM
putch(RA1);
putch(RA2);
putch(0xAA);
putch(LQA);
putch(LQB
        }
}
```

Interrupt Routine

```
void interrupt my_isr(void){

    /***** Usart Code *****/
    if((RCIE)&&(RCIF)){
        RXhold=RCREG;
        RCIF=0; // clear event flag
```

```
            if(RCREG==0xFF)         //Consider 0xFF to be a SYNC bit
                ind=0;              //ignore by keeping array pointer @ 0
            else if (ind<20)        //Ignore any extraneous bits (more than 20 chars thrown out)
                cmds[ind]=RXhold; // Otherwise store char(byte) into array
            else
                ind=0;              //if data is not valid, or more than 20 entries, reset pointer

            ind++;                  //progress pointer
            //DEBUG
//          putch(0xEE);
//          putch(ind);
//          putch(cmds[ind]);

            //Command Array Format
            //0-Command type
            //1-Address
            //Direction
            //Speed
            //Destination

        }

        //POSITIONING COMMANDS
        if(mode==0xAC){
            if(destination==position){
                CCPR1L=0;
                RA0=0;
                RB5=0;
                feedback++;     //Count zero crossings
                speed--;
            }
            else if(destination>position){     //move joint towards 0
                CCPR1L=speed;
                RA0=0;
                RB5=1;

            }
            else if(destination<position){     //move joint towards 0
                CCPR1L=255-speed;
                RA0=1;
                RB5=1;
            }
            if(feedback>80)         //If too many zero crossings, stop motor to avoid oscillations
                mode=0x00;
        }
```

```
        /* QEI State Table from http://www.sxlist.com/techref/io/sensor/pos/enc/quadrature.htm
 (A;B)          Current:
Previous ||(0;0) (0;1) (1;0) (1;1)
       |+----------------------
 (0;0)  |   NC    CW   cCW   Err       NC=No change
 (0;1)  |  cCW    NC   Err    CW       CW=Clockwise
 (1;0)  |   CW   Err    NC   cCW      cCW=Counter-clockwise
 (1;1)  |  Err   cCW    CW    NC      Err=Error (ignored)
*/
        QA=RA1;          //STORE current states
        QB=RA2;
//DEBUG
//      printf("A\n");
//      putch(RA1);
//      putch(RA2);

//Use new states to determine phase, which will give shaft direction
//row1
        if((LQA==0&&LQB==0)&&(QA==0&&QB==1))
                position++;
        else if((LQA==0&&LQB==0)&&(QA==1&&QB==0))
                position--;
//row2
        else if((LQA==0&&LQB==1)&&(QA==0&&QB==0))
                position--;
        else if((LQA==0&&LQB==1)&&(QA==1&&QB==1))
                position++;
//row3
        else if((LQA==1&&LQB==0)&&(QA==0&&QB==0))
                position++;
        else if((LQA==1&&LQB==0)&&(QA==1&&QB==1))
                position--;
//row4
        else if((LQA==1&&LQB==1)&&(QA==0&&QB==1))
                position--;
        else if((LQA==1&&LQB==1)&&(QA==1&&QB==0))
                position++;

        LQA=QA;  //store new states to be used on next iteration
        LQB=QB;

                T0IF=0; // clear interrupt flag
        }
}
```