# logo_bw.eps*Worcester Polytechnic Institute*

# *Robotics Engineering Program*

Lab 3 - Sensors

Submitted By  
Joel Sutherland  
Adam Panzica

**Date Submitted : 2/23/2010**

**Date Completed : 2/23/2010**

**Course Instructor : Profs. Fischer & Ciaraldi**

**Lab Section : C02**

# Abstract

This lab explored the use of various sensors, both ranging and non-ranging, and techniques for integrating them with the AVR. The sensors tested included a 3 axis accelerometer, quadrature encoders, an IR range finder, and an ultrasonic sensor. The EDUarm was used where appropriate to generate performance data for the sensors, and analysis on the advantages and limitations of each sensor type was performed.

Contents

[Abstract 1](#_Toc254854568)

[Introduction 3](#_Toc254854569)

[Methodology 4](#_Toc254854570)

[Code 4](#_Toc254854571)

[IR Sensor 4](#_Toc254854572)

[Accelerometer 4](#_Toc254854573)

[Encoder 5](#_Toc254854574)

[Ultrasonic Sensor 6](#_Toc254854575)

[Results and Analysis 7](#_Toc254854576)

[Discussion 8](#_Toc254854577)

[Appendix A: IR\_Sensor.h/.c 9](#_Toc254854578)

[Appendix B: Accelerometer Code 11](#_Toc254854579)

[Appendix C: Encoder.h/.c 15](#_Toc254854580)

[Appendix D: Ultrasonic code 21](#_Toc254854581)

# Introduction

In this lab exercise, the 2-link arm configuration and control system from lab two was modified to test various sensors beyond the potentiometers used in previous labs. There were four additional sensors that needed to be interfaced: a Parallax 3-axis accelerometer board featuring the Hitachi H48C and an onboard MCP3204 ADC, the Globe quadrature encoders built into the EDUarm, a Sharp Inferred Range Finder, and an Ultrasonic sensor that had no built in range processing.

There were several objectives for this lab. The first was to continue to expand our knowledge and experience with interfacing to external perfidies. Each of the four sensors had a different method of communication. The Parallax accelerometer board’s built in ADC uses three-wire SPI. Therefore, the hardware SPI capabilities of the AVR are not compatible, as they do not tri-state outputs when not in use (in the AVR, it clocks in data on MISO as it is clocking out data on MOSI, so they are technically both always in use). Therefore, it was necessary to write our own ‘bit banged’ SPI routines to communicate with it. The Encoders, on the other hand, are interfaces through a dedicated external hardware encoder processing IC. This IC communicates through four-wire SPI, and is thus compatible with the hardware SPI ports. However, it uses a different clock polarity and phase than the DAC on the daughter card, and has a fairly complex instruction set, complicating communication to it. The IR sensor is the easiest to interface to, as it simply returns an analog voltage and thus can be hooked to an ADC port. The trick is that the voltage is not linearly (or even approximately linearly) proportional to range, so additional processing must be done to turn the signal into useful information. The ultrasonic sensor was handled differently. Instead of returning data the AVR, its output was hooked to the oscilloscope. This is because the sample frequency needed to do accurate waveform analysis is so high (on the order of MHz) that it would render the AVR unable to do anything else. The AVR was simply used as the pulse trigger.

Beyond teaching the ins and outs of peripheral integration, the lab also served as an analysis of the advantages and disadvantages of the various sensor types. For example, the potentiometers, which natively measure absolute angular position, the encoders, which in this case natively measure incremental angular position and angular velocity, and the accelerometer, which natively measures linear acceleration, were used to measure the position, velocity and acceleration of the robot arm via integration/derivation. These results were compared so that it became clear that certain sensors should be used for certain measurements, and that while they can be used for other measurements through clever math than the confidence of that information is reduced. Understanding what sensors to use when is a vital skill for any robotics engineer, so this lab was designed to hammer that point home.

# Methodology

## Code

For this lab, there were several new libraries that needed to be added to our code to integrate the new sensors. In addition, some of the old libraries needed to be modified to make them more flexible so that they could be used in conjunction with the new sensors, or to simply add functionality to them when the new sensor fit into a category which already had a library, such as the MCP3024 ADC.

## IR Sensor

The library for the IR sensor is quite straightforward. It contains a single primary function which utilizes a lookup table and linear interpolation to process the nonlinear data returned by the sensor. As the AVR’s ADC is 10bit, and thus has a range of 0-1024, the lookup table has 11 locations, corresponding to increments of 100 ADC counts. The ranges corresponding to these values were determined experimentally by manual calibration of the sensor. Since the sensor is interfaced directly through the AVR’s ADC, it uses the functions already developed in the ADC library to interface to the sensor. Code for the IR sensor can be seen in appendix A.

## Accelerometer

There were two tasks that needed to be completed to interface with the accelerometer. The first was interfacing to the sensor itself. The Parallax accelerometer has a built in MCP3024 ADC which converts the analog output from the actual accelerometer IC to a digital output convenient for microprocessors to read. The MCP3024 on the board has its MOSI and MISO pins hooked together to form what is commonly called three-wire SPI. This saves I/O pins on the host microcontroller, but it has the downside of making the sensor incompatible with the hardware SPI ports on the AVR, as they expect four wire SPI.

Therefore, to communicate with the MCP3024, we needed to develop our own, ‘bit banged’ SPI library. ‘Bit banging’ is the process of emulating something- generally an I/O controller- which is normally handled by dedicated hardware (in this case, generating an SPI clock and shifting in/out data) in software. Our bit banged SPI controller is not modular, as it is hard coded to the clock-phase, polarity and clock speed dictated by the MCP3024. Keeping the code simple and application specific was done to reduce development time, as well as decrease the likelihood of bugs in the code due to incorrectly handled configurations.

Once communication was established with the MCP3024, additional functions were needed to turn the 12bit value returned from each channel into a useful number. The documentation for the parallax accelerometer provides a useful function for converting the ADC value into G’s:

The 0.0022 is a ‘magic constant’ which takes into account the resolution of the ADC as well as the dynamic range of the accelerometer operating at 3.3V. and are the ADC values red from the desired measurement axis, and the voltage reference channel (essentially a calibration point outputted by the accelerometer to the ADC), respectively. To allow the use of integer math, as well as to output data in the more useful form of 100ths of a G, the .0022 number is scaled appropriately. Full code for the accelerometer can be seen in appendix B.

## Encoder

For the encoders, we had to use an external interface chip, the LS7366R, which is an ASIC designed to monitor a quadrature encoder. The interface chip was needed because while the AVR is perfectly capable of monitoring a quadrature encoder using pin-change interrupts, the high resolution of the encoders on the EDUarm combined with the high gear reduction on the output meant that the encoders would require about 90KHz of bandwidth each, leaving very little time for the processor to do other tasks between interrupts. The LS7366R communicates over standard 4-wire SPI, so it was compatible with the AVR’s hardware SPI port. However, it uses a different clock polarity and phase angle than the DAC on the daughter board. This meant that it was necessary to write a separate SPI initialization for it, and to call the respective encoder SPI or DAC SPI initialization before calls to either device were made.

In addition, unlike other devices that we had interfaced with before, the LS7366R does not have a set message length, and has commands which are both mixed read/write or write only. It has a simple API consisting of 2-5 bytes. The first byte is always a transmit from the master, and is the command word. The next 1-4 bytes can be either transmits or receives depending on what the contents of the command word were. Only one command can be performed per communication cycle, so the SS pin must be clocked before the next command can be sent.

The LS7366R has four different operations: Read, Write, Clear to Zero, and Load. It also has X registers: MDR0, MDR1, DTR, CNTR, OTR, and STR. However, not every command can be performed on every register. The command word consists of, in order from MSB to LSB, 2 bits to denote the operation, 3 bits to denote the register to perform the op on, and then 3 bits of don’t care which are set to 0. A union, ENCODER\_COMMAND, as well as a function Encoder\_Generate\_Command(), with creates an ENCODER\_COMMAND union from a single ‘command’ parameter, was utilized to simplify the generation of command words.

MDR0 and MDR1 are the LS7366’s mode registers. They tell the chip how it should interpret the output from the encoder, how it should increment/de-increment the counter, as well as enabling and disabling counting. As with the command word, unions were used to simplify the creation of MDR words: ENCODER\_MDR0 and ENCODER\_MDR1. The function Encoder\_Initialize() generates the two MDR’s and writes them to the LS7366, and then clears the counter as it is not cleared unless the encoder chip loses power or is expressly told to clear it.

The final function for communication with the encoder is Encoder\_Read(). It takes a ‘command’ parameter (which is simply a number, but to make programming it more verbose symbolic constants for the commands are provided in the header) and a pointer to a long to store the result read back from the LS7366R. This function handles all of the communication over SPI as well as handling the variable message lengths, so the programmer can simply call this function and not worry about the low level details. Full code for the Encoder functions can be seen in appendix C.

## Ultrasonic Sensor

Interfacing to the ultrasonic sensor was a bit different than the other sensors in this lab. The sensor had no onboard signal processing. Therefore, it returns raw analog values corresponding to the intensity of the sound received by the sensor. It does this at an incredibly high frequency; sampling the waveform would require bandwidth on the order of several MHz. Beyond the fact that the ADC on the AVR cannot sample faster than 200KHz, attempting to sample the signal would use up so much processor time that it would leave little computing power for anything else. Therefore, the only thing that the AVR was required to do was trigger the sensor by sending a 1ms pulse every 31ms. To do this, we were required to utilize timer 0. Timer 0 is an 8bit timer, and with the maximum prescale it has a minimum frequency of 35Hz, higher than the ~31Hz needed by the sensor. Therefore, we had to implement a counter system which would increment at a rate of 1ms, which with a prescale of 1024, is exactly every 9 timer counts. On the 31’s increment, it pulls the OC0A pin high for 1ms, then resets. Complete code can be seen in appendix D.

# Results and Analysis

# Discussion

# Appendix A: IR\_Sensor.h/.c

/\*\*

\*@file IR\_Sensor.h

\*

\*@date Feb 16, 2010

\*@author Adam Panzica

\*/

**#ifndef** IR\_SENSOR\_H\_

**#define** **IR\_SENSOR\_H\_**

**int** ***ReadIR***( **int** **channel** )**;**

**#endif** /\* IR\_SENSOR\_H\_ \*/

/\*\*

\* @file IR\_Sensor.c

\* @brief Functions for utilizing the Sharp IR range finding sensor

\* @date Feb 16, 2010

\* @author Adam Panzica

\* @version 1.0 Initial version with ReadIR() and interpolate()

\*/

**#include** "IR\_Sensor.h"

**#include** "Simple\_Serial.h"

**#include** "ADC.h"

**#include** <avr/io.h>

/\*\*@fn int IR\_interpolate(int xA, int yA, int xB, int yB, int value)

\* @brief Performs a linear interpolation

\* @param [in] xA the lower bound along the input axis

\* @param [in] yA the lower bound along the output axis

\* @param [in] xB the upper bound along the input axis

\* @param [in] yB the upper bound along the output axis

\* @param [in] value the input value to be interpolated

\* @returns the output value of the interpolated input

\*

\* This function performs a linear interpolation between two points and a given input

\*/

**int** ***IR\_interpolate***(**int** **xA**, **int** **yA**, **int** **xB**, **int** **yB**, **int** **value**)

{

**long** ***temp*;**

**temp** **=** (**long**)**yA+**(((**long**)**value-**(**long**)**xA**)**\***((**long**)**yB** **-** (**long**)**yA**))**/**((**long**)**xB-**(**long**)**xA**)**;**

**return** (**int**)**temp;**

}

/\*\*@fn int ReadIR( int channel )

\* @brief Gets a range formt the IR sensor

\* @param [in] channel the desired channel to read from

\* @returns the range to target, in mm

\*

\* This function reads a value from the sharp IR sensor on the specified channel and returns a range in millimeters

\*/

**int** ***ReadIR***( **int** **channel** )

{

**int** ***rawVol*;**

**int** ***interRangeHigh*;**

**int** ***interVolHigh*;**

**int** ***interRangeLow*;**

**int** ***interVolLow*;**

**int** ***reduced*;**

**long** ***temp*;**

**int** ***lookup*** [20] **=** {622, 521, 451, 387, 333 ,298, 270, 241, 219, 203, 184, 168, 156, 143, 130, 120, 114, 110, 105, 98}**;**

**rawVol** **=** **ADC\_Get\_Value**((**char**)**channel**)**;**

**temp** **=** ((((**long**)475 **\*** (**long**)**rawVol**)**/**100) **-** 475) **/** 100**;**

**reduced** **=** (**int**)**temp;**

**interRangeLow** **=** **lookup**[**reduced**]**;**

**interRangeHigh** **=** **lookup**[**reduced**+1]**;**

**temp** **=** ((((**long**)**reduced\***(**long**)100)**+**(**long**)475) **\*** (**long**)100)**/**(**long**)475**;**

**interVolLow** **=** (**int**)**temp;**

**temp** **=** (((((**long**)(**reduced**+1))**\***(**long**)100)**+**(**long**)475) **\*** (**long**)100)**/**(**long**)475**;**

**interVolHigh** **=** (**int**)**temp;**

**return** **IR\_interpolate**(**interVolLow**, **interRangeLow**, **interVolHigh**, **interRangeHigh**, **rawVol**)**;**

}

# Appendix B: Accelerometer Code

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SYMBOLIC CONSTANTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*@def BBCLK

\* Pin number for Bit Banged SPI clock

\*/

**#define** **BBCLK** PD7

/\*\*@def BBD

\* Pin number for Bit Banged SPI data I/O line

\*/

**#define** **BBD** PD6

/\*\*@def BBSS

\* Pin number for Bit Banged SPI Slave Select

\*/

**#define** **BBSS** PD5

/\*\*@def BBPERIOD

\* Clock period for Bit Banged SPI

\*/

**#define** **BBPERIOD** 98

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MCP3024 COMMAND UNION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**typedef** **union** MCP3024\_COMMAND{

**struct** {

**unsigned** **:**3**;**

**unsigned** \_D0 **:**1**;**

**unsigned** \_D1 **:**1**;**

**unsigned** \_D2 **:**1**;**

**unsigned** \_SGDF **:**1**;**

**unsigned** \_STRB **:**1**;**

}**;**

**struct** {

**unsigned** \_w **:**8**;**

}**;**

} MCP3024\_COMMAND**;**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PUBLIC FUNCTIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*@fn char ADC\_MCP3204\_Send\_Command(MCP3024\_COMMAND \*command)

\* @brief sends a command word to the MCP3024

\* @param [in] \*command Pointer to an MCP3024 command struct

\* @returns 1 if successful

\*

\* This function sends a command word to the MCP3024 ADC via bit banged SPI

\*/

**char** ***ADC\_MCP3204\_Send\_Command***(MCP3024\_COMMAND **\*command**)

{

//set up direction registers

DDRD **|=** (1**<<BBCLK**)**|**(1**<<BBD**)**;**

PORTD **&=** (**~**((1**<<BBCLK**)**|**(1**<<BBD**)))**;**

//send start bit of the command

**Pin\_Set**('D', **BBD**, **command->**\_STRB)**;**

**\_delay\_us**(1)**;**

**Pin\_Set**('D', **BBCLK**, 1)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

**Pin\_Set**('D', **BBCLK**, 0)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

//send Single/Diff bit of the command

**Pin\_Set**('D', **BBD**, **command->**\_SGDF)**;**

**\_delay\_us**(1)**;**

**Pin\_Set**('D', **BBCLK**, 1)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

**Pin\_Set**('D', **BBCLK**, 0)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

//send D2 bit of the command

**Pin\_Set**('D', **BBD**, **command->**\_D2)**;**

**\_delay\_us**(1)**;**

**Pin\_Set**('D', **BBCLK**, 1)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

**Pin\_Set**('D', **BBCLK**, 0)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

//send D1 bit of the command

**Pin\_Set**('D', **BBD**, **command->**\_D1)**;**

**\_delay\_us**(1)**;**

**Pin\_Set**('D', **BBCLK**, 1)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

**Pin\_Set**('D', **BBCLK**, 0)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

//send D0 bit of the command

**Pin\_Set**('D', **BBD**, **command->**\_D0)**;**

**\_delay\_us**(1)**;**

**Pin\_Set**('D', **BBCLK**, 1)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

**Pin\_Set**('D', **BBCLK**, 0)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

**return** 1**;**

}

/\*\*@fn char ADC\_MCP3204\_Recive\_Data(int \*receive)

\* @brief Reads a value from the MCP3024

\* @param [in] \*receive Pointer to an integer to write the value to

\* @returns 1 if successful

\*

\* This function reads a data segment from the MCP3024 ADC via bit banged SPI

\*/

**char** ***ADC\_MCP3204\_Recive\_Data***(**int** **\*receive**)

{

**char** ***i*;**

**char** ***temp*;**

//set up direction registers to receive data

DDRD **&=** **~**(1**<<BBD**)**;**

//clock in the null bit

**Pin\_Set**('D', **BBCLK**, 1)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

**Pin\_Set**('D', **BBCLK**, 0)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

//clock in data

**for** (**i=**0**;** **i<=**12**;** **i++**)

{

**temp** **=** **Pin\_Read**('D', **BBD**)**;**

**\*receive** **|=** **temp;**

**\*receive** **=** **\*receive** **<<** 1**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

**Pin\_Set**('D', **BBCLK**, 0)**;**

**\_delay\_us**(**BBPERIOD/**2)**;**

**if**(**i<**12) **Pin\_Set**('D', **BBCLK**, 1)**;**

}

//Pin\_Set('D', BBCLK, 0);

**return** 1**;**

}

/\*\*@fn int ADC\_MCP3204\_Get\_Value(unsigned char channel, unsigned char justify)

\* @brief Reads a value from the MCP3024 from the specified channel

\* @param [in] channel ADC channel to read from

\* @param [in] justify 0 if the returned 12bit value should be right justified, 1 if it should be left justified

\* @returns A 12bit value from the ADC

\*

\* This function uses bit banged SPI to communicate with the MCP3024 ADC. It gets a measurement from the requested channel and, if desired, left justifies it.

\*/

**int** ***ADC\_MCP3204\_Get\_Value***(**unsigned** **char** **channel**, **unsigned** **char** **justify**)

{

MCP3024\_COMMAND ***commandByte*;**

**int** ***dataPacket*=**0**;**

//set up SS line direction register

DDRD **|=** (1**<<BBSS**)**;**

**commandByte.**\_STRB **=** 1**;**

**commandByte.**\_SGDF **=** 1**;**

**switch** (**channel**)

{

**case** 'a'**:**

**case** 'A'**:**

**case** 0**:**

**commandByte.**\_D2 **=** 0**;**

**commandByte.**\_D1 **=** 0**;**

**commandByte.**\_D0 **=** 0**;**

**break;**

**case** 'b'**:**

**case** 'B'**:**

**case** 1**:**

**commandByte.**\_D2 **=** 0**;**

**commandByte.**\_D1 **=** 0**;**

**commandByte.**\_D0 **=** 1**;**

**break;**

**case** 'c'**:**

**case** 'C'**:**

**case** 2**:**

**commandByte.**\_D2 **=** 0**;**

**commandByte.**\_D1 **=** 1**;**

**commandByte.**\_D0 **=** 0**;**

**break;**

**case** 'd'**:**

**case** 'D'**:**

**case** 3**:**

**commandByte.**\_D2 **=** 0**;**

**commandByte.**\_D1 **=** 1**;**

**commandByte.**\_D0 **=** 1**;**

**break;**

**default:**

**commandByte.**\_D2 **=** 0**;**

**commandByte.**\_D1 **=** 0**;**

**commandByte.**\_D0 **=** 0**;**

**break;**

}

**Pin\_Set**('D', **BBSS**, 0)**;**

**ADC\_MCP3204\_Send\_Command**(**&commandByte**)**;**

**ADC\_MCP3204\_Recive\_Data**(**&dataPacket**)**;**

**Pin\_Set**('D', **BBSS**, 1)**;**

//if justify was set, left justify the data

**if**(**justify** **==** 1)

{

**dataPacket** **=** **dataPacket** **<<** 4**;**

}

**return** **dataPacket;**

}

/\*\*@fn int ADC\_MCP3204\_Count\_to\_Gs(int ADCVal, int VRef)

\* @brief Converts a 12bit ADC value to 100ths of a G

\* @param [in] ADCVal 12bit ADC value to convert

\* @param [in] VRef The VRef value from the ADC

\* @returns a value in 100ths of a G

\*

\* This function converts the 12bit ADC value from the MCP3024 on the Parallax accelerometer board and converts it into 100ths of a G.

\*/

**int** ***ADC\_MCP3204\_Count\_to\_Gs***(**int** **ADCVal**, **int** **VRef**)

{

**long** ***temp*;**

**temp** **=** ((**long**)**ADCVal** **-** (**long**)**VRef**)**\***(**long**)22**/**(**long**)100**;**

**return** (**int**)**temp;**

}

# Appendix C: Encoder.h/.c

/\*\*

\*@file Encoder.h

\*

\*@date Feb 19, 2010

\*@author Adam Panzica

\*/

**#ifndef** ENCODER\_H\_

**#define** **ENCODER\_H\_**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ENCODER COMMANDS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* Write to MDR0\*/

**#define** **WRITEMDR0** 9

/\*\* Write to MDR1\*/

**#define** **WRITEMDR1** 10

/\*\* Read from MDR0\*/

**#define** **READMDR0** 4

/\*\* Read from MDR1\*/

**#define** **READMDR1** 5

/\*\* Read from CNTR\*/

**#define** **READCNT** 6

/\*\* Clear CNTR\*/

**#define** **CLEARCNT** 2

**void** ***Encoder\_Initialize***()**;**

**char** ***Encoder\_Read***(**char** **command**, **long** **\*result**)**;**

**#endif** /\* ENCODER\_H\_ \*/

/\*\*

\*@file Encoder.c

\*

\*@date Feb 19, 2010

\*@author Adam Panzica

\*/

**#include** "Encoder.h"

**#include** "Simple\_Serial.h"

**#include** "SPI.h"

**#include** "PortIO.h"

**#include** <avr/interrupt.h>

**#include** <avr/io.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ENCODER COMMAND UNION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**typedef** **union** ENCODER\_COMMAND{

**struct** {

**unsigned** \_DC **:**3**;**

**unsigned** \_REGSEL**:**3**;**

**unsigned** \_COMSEL**:**2**;**

}**;**

**struct** {

**unsigned** \_COMMAND**:**8**;**

}**;**

} ENCODER\_COMMAND**;**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ENCODER MDR UNIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**typedef** **union** ENCODER\_MDR0{

**struct** {

**unsigned** \_QCM **:**2**;**

**unsigned** \_FRCM **:**2**;**

**unsigned** \_INDEX **:**2**;**

**unsigned** \_AI **:**1**;**

**unsigned** \_CLKDIV**:**1**;**

}**;**

**struct** {

**unsigned** \_MDR0**:**8**;**

}**;**

} ENCODER\_MDR0**;**

**typedef** **union** ENCODER\_MDR1{

**struct** {

**unsigned** \_CMODE **:**2**;**

**unsigned** \_ENABLE**:**1**;**

**unsigned** \_RESER **:**1**;**

**unsigned** \_FLAGS **:**4**;**

}**;**

**struct** {

**unsigned** \_MDR1**:**8**;**

}**;**

} ENCODER\_MDR1**;**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ENCODER OPCODES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*Selects no register\*/

**#define** **RSNONE** 0b000

/\*\*Selects MDR0 register\*/

**#define** **RSMDR0** 0b001

/\*\*Selects MDR1 register\*/

**#define** **RSMDR1** 0b010

/\*\*Selects DTR register\*/

**#define** **RSDTR** 0b011

/\*\*Selects CNTR register\*/

**#define** **RSCNTR** 0b100

/\*\*Selects OTR register\*/

**#define** **RSOTR** 0b101

/\*\*Selects STR register\*/

**#define** **RSSTR** 0b110

/\*\*Perform a clear command\*/

**#define** **CCLR** 0b00

/\*\*Perform a read command\*/

**#define** **CRD** 0b01

/\*\*Perform a write command\*/

**#define** **CWR** 0b10

/\*\*Perform a load command\*/

**#define** **CLOAD** 0b11

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PRIVATE FUNCTIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*@fn char Encoder\_Initialize\_SPI()

\*@brief Initializes the AVR's hardware SPI to communicate with the LS7366R encoder chip

\*/

**char** ***Encoder\_Initialize\_SPI***()

{

**SPI\_Initialize**(**MSTRSEL**,**MSBF**,**CPLLH**,**CPHSL**)**;**

**return** 1**;**

}

/\*\*@fn void Encoder\_Generate\_Command(unsigned char encCommand, ENCODER\_COMMAND \*genCom)

\*@brief Generates a valid LS7366R encoder chip command

\*@param [in] encCommand Command number to be generated

\*@param [in] \*genCom Pointer to an ENCODER\_COMMAND union to store the generate command to

\*/

**void** ***Encoder\_Generate\_Command***(**unsigned** **char** **encCommand**, ENCODER\_COMMAND **\*genCom**)

{

**genCom->**\_DC **=** 0**;**

**switch**(**encCommand**)

{

**case** 0**:** //Generates a command to clear MDR0

**genCom->**\_REGSEL **=** **RSMDR0;**

**genCom->**\_COMSEL **=** **CCLR;**

**break;**

**case** 1**:** //Generates a command to clear MDR1

**genCom->**\_REGSEL **=** **RSMDR1;**

**genCom->**\_COMSEL **=** **CCLR;**

**break;**

**case** 2**:** //Generates a command to clear CNTR

**genCom->**\_REGSEL **=** **RSCNTR;**

**genCom->**\_COMSEL **=** **CCLR;**

**break;**

**case** 3**:** //Generates a command to clear STR

**genCom->**\_REGSEL **=** **RSSTR;**

**genCom->**\_COMSEL **=** **CCLR;**

**break;**

**case** 4**:** //Generates a command to read MDR0

**genCom->**\_REGSEL **=** **RSMDR0;**

**genCom->**\_COMSEL **=** **CRD;**

**break;**

**case** 5**:** //Generates a command to read MDR1

**genCom->**\_REGSEL **=** **RSMDR1;**

**genCom->**\_COMSEL **=** **CRD;**

**break;**

**case** 6**:** //Generates a command to read CNTR

**genCom->**\_REGSEL **=** **RSCNTR;**

**genCom->**\_COMSEL **=** **CRD;**

**break;**

**case** 7**:** //Generates a command to read OTR

**genCom->**\_REGSEL **=** **RSOTR;**

**genCom->**\_COMSEL **=** **CRD;**

**break;**

**case** 8**:** //Generates a command to read STR

**genCom->**\_REGSEL **=** **RSSTR;**

**genCom->**\_COMSEL **=** **CRD;**

**break;**

**case** 9**:** //Generates a command to write MDR0

**genCom->**\_REGSEL **=** **RSMDR0;**

**genCom->**\_COMSEL **=** **CWR;**

**break;**

**case** 10**:** //Generates a command to write MDR1

**genCom->**\_REGSEL **=** **RSMDR1;**

**genCom->**\_COMSEL **=** **CWR;**

**break;**

**case** 11**:** //Generates a command to write DTR

**genCom->**\_REGSEL **=** **RSDTR;**

**genCom->**\_COMSEL **=** **CWR;**

**break;**

**case** 12**:** //Load DTR to CNTR in 'parallel'

**genCom->**\_REGSEL **=** **RSCNTR;**

**genCom->**\_COMSEL **=** **CLOAD;**

**break;**

**case** 13**:** //Load CNTR to DTR in 'parallel'

**genCom->**\_REGSEL **=** **RSOTR;**

**genCom->**\_COMSEL **=** **CLOAD;**

**break;**

**default:** //Generates an empty command (operates on no register)

**genCom->**\_REGSEL **=** **RSNONE;**

**genCom->**\_COMSEL **=** **CRD;**

**break;**

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PUBLIC FUNCTIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*@fn void Encoder\_Initialize()

\*@brief Initializes the LS7366R encoder chip for use with the EDUarm

\*/

**void** ***Encoder\_Initialize***()

{

ENCODER\_COMMAND ***init***[3]**;**

ENCODER\_MDR1 ***mdr1*;**

ENCODER\_MDR0 ***mdr0*;**

**char** ***packets***[5]**;**

**long** ***tmdr0*** **=** 0**;**

**long** ***tmdr1*** **=** 0**;**

**long** ***tcnt*** **=** 1232**;**

**Encoder\_Generate\_Command**(**WRITEMDR0**, **&init**[0])**;**

**Encoder\_Generate\_Command**(**WRITEMDR1**, **&init**[1])**;**

**Encoder\_Generate\_Command**(**CLEARCNT**, **&init**[2])**;**

**Serial\_Print\_String**("\r\nInitialization Commands Generated")**;**

**mdr0.**\_CLKDIV**=** 0**;** //Sets the clock divider in mdr0

**mdr0.**\_AI **=** 0**;** //Sets the Index Syncornisity in mdr0

**mdr0.**\_INDEX **=** 0b00**;** //Sets the Intedx mode in mdr0

**mdr0.**\_FRCM **=** 0b00**;** //Sets the free-run mode in MDR0

**mdr0.**\_QCM **=** 0b10**;** //Sets the quadrature mode in MDR0

**mdr1.**\_CMODE **=** 0**;** //Sets the count mode in MDR1

**mdr1.**\_ENABLE **=** 1**;** //Enables the counter

**mdr1.**\_FLAGS **=** 0**;** //Sets the flags in MDR1

**mdr1.**\_RESER **=** 0**;** //filler for the reserved bit

**packets**[0]**=init**[0]**.**\_COMMAND**;**

**packets**[1]**=mdr0.**\_MDR0**;**

**packets**[2]**=init**[1]**.**\_COMMAND**;**

**packets**[3]**=mdr1.**\_MDR1**;**

**packets**[4]**=init**[2]**.**\_COMMAND**;**

//disable interrupts to ensure that there will be no conflict with DAC SPI that is called on interrupts

**cli**()**;**

//send initialization packets

**Encoder\_Initialize\_SPI**()**;**

**SPI\_Send\_Packet**(**&packets**[0],2,**SSPIN\_ENC**)**;**

**SPI\_Send\_Packet**(**&packets**[2],2,**SSPIN\_ENC**)**;**

**SPI\_Send\_Packet**(**&packets**[3],1,**SSPIN\_ENC**)**;**

//debug

**Serial\_Print\_String**("\r\nSet MDR0 To: ")**;**

**Serial\_Print\_Int**((**int**)**mdr0.**\_MDR0,2)**;**

**Serial\_Print\_String**("\r\nSet MDR1 To: ")**;**

**Serial\_Print\_Int**((**int**)**mdr1.**\_MDR1,2)**;**

**Encoder\_Read**(**READMDR0**, **&tmdr0**)**;**

**Encoder\_Read**(**READMDR1**, **&tmdr1**)**;**

**Encoder\_Read**(**READCNT**, **&tcnt**)**;**

**Serial\_Print\_String**("\r\nMDR0: ")**;**

**Serial\_Print\_Int**((**int**)**tmdr0**, 2)**;**

**Serial\_Print\_String**("\r\nMDR1: ")**;**

**Serial\_Print\_Int**((**int**)**tmdr1**, 2)**;**

**Serial\_Print\_String**("\r\nCNTR: ")**;**

**Serial\_Print\_Int**((**int**)**tcnt**, 2)**;**

//re-enable interrupts

**sei**()**;**

}

/\*\*@fn char Encoder\_Read(char command, long \*result)

\*@brief Reads from a register on the LS7366R encoder chip

\*@param [in] command A valid encoder command. Use the symbolic constants in Encoder.h

\*@param [in] \*result pointer to a long value to store the result to

\*

\*This function reads from a register on the LS7366R encoder chip over SPI. It stores the result to the location pointed to by \*result.

\*NOTE: Not all registers are longs (and depending on configuration, none of them might be), so it is up to the user to interpret the

\*returned value correctly. For READCNT commands, if the CNTR register is set to be less than 4 bytes, it will store the returned data

\*in the upper n bytes of the long, MSBF, where n is the number of bytes CNTR is set up to be. For all other commands, it stores the value

\*in the lower bytes.

\*NOTE: Interrupts are disabled while this operation communicates over the SPI bus. Care should be taken that critical interrupts do not

\*happen during the operation, as they will be ignored.

\*/

**char** ***Encoder\_Read***(**char** **command**, **long** **\*result**)

{

ENCODER\_COMMAND ***genCom*;**

**Encoder\_Generate\_Command**(**command**, **&genCom**)**;**

**Serial\_Print\_String**("\r\nGenerated Command: ")**;**

**Serial\_Print\_Int**((**int**)**genCom.**\_COMMAND, 2)**;**

**cli**()**;**

**Encoder\_Initialize\_SPI**()**;**

//bring encoder SS pin low to enable communications

**Pin\_Set**('B', **SSPIN\_ENC**, 0)**;**

**switch**(**command**)

{

**case** **READMDR0:**

**case** **READMDR1:**

**SPI\_Send\_Byte**(**genCom.**\_COMMAND)**;**

**\*result** **=** **SPI\_Read\_Byte**()**;**

**break;**

**case** **READCNT:**

**SPI\_Send\_Byte**(**genCom.**\_COMMAND)**;**

**\*result** **=** **SPI\_Read\_Byte**()**;**

**\*result** **=** **\*result<<**8**;**

**\*result** **|=** **SPI\_Read\_Byte**()**;**

**\*result** **=** **\*result<<**8**;**

**\*result** **|=** **SPI\_Read\_Byte**()**;**

**\*result** **=** **\*result<<**8**;**

**\*result** **|=** **SPI\_Read\_Byte**()**;**

}

**Pin\_Set**('B', **SSPIN\_ENC**, 1)**;**

**sei**()**;**

**return** 1**;**

}

# Appendix D: Ultrasonic code

/\*\*@fn void Timer8\_Initialize\_US\_Pulse()

\* @brief Sets up the timer to generate a pulse at set intervals

\*

\* This function sets up the AVR's 8bit Timer 0 to generate a 1ms pulse every 31ms. It generates this continuously, trapping the processor in an infinate loop.

\*/

**void** ***Timer8\_Initialize\_US\_Pulse***()

{

**int** ***i*** **=** 0**;**

**int** ***startTime*;**

**int** ***currTime*;**

DDRB **|=** (1**<<**PB3)**;**

TCCR0A **=** 0**;**

TCCR0B **=** (1**<<**CS02)**|**(0**<<**CS01)**|**(1**<<**CS00)**;**

**while**(1)

{

**Pin\_Set**('B', PB3, 1)**;**

**while**(**i** **<=** 31)

{

**startTime** **=** TCNT0**;**

**currTime** **=** TCNT0**;**

//if there will not be a valid frame window without roll over, reset the timer

**if** (**startTime** **>** 245)

{

TCNT0 **=** 0**;**

**startTime** **=** 0**;**

**currTime** **=** 0**;**

}

//idle waiting for 1ms to pass

**while**((**currTime-startTime**) **<=** 18 )

{

**currTime** **=** TCNT0**;**

}

//increment the ms counter

**i++;**

}

**i** **=** 0**;**

**Pin\_Set**('B', PB3, 0)**;**

TCNT0 **=** 0**;**

**startTime** **=** 0**;**

**currTime** **=** 0**;**

//idle waiting for 1ms to pass

**while**((**currTime-startTime**) **<=** 18 )

{

**currTime** **=** TCNT0**;**

}

}

}