SOEN 422/2 Fall 2016
LAB REPORT 2-3

Harley McPhee 27003226

1. Introduction

1.1 Problem Statement

The purpose of these two laboratory experiment was to begin using the ATMEGA328 microcontroller and to program it in the C language using the AVR platform. The programs in lab two introduced us to interrupts and timers on the ATMEGA328 microcontroller and the programs in lab three showed us various communication protocols such as FTDI, SPI, and I2C for communicating between processes running on multiple microcontrollers. The following programs were developed

1. Toggling an LED by Switch using interrupt
2. Timer Controlled LED
3. Button Controlled Timers
4. FTDI Greetings
5. SPI Master/Slave Infrared read on slave and transmit to master
6. I2C Infrared read on one device and other device receives from it

1.2 Abbreviations and Acronyms

^=	Bitwise XOR assignment operator
|=	Bitwise OR assignment operator
&=	Bitwise AND assignment operator
<<	Bitwise shift left
~	Bitwise negation operator
#define	Creates a macro
#include	Tells the preprocessor to include libraries into the source code
<avr/io.h>	Includes the required input and output definitions
<avr/interrupt.h>	Includes the required interrupt definitions
<util/delay.h>	Includes functions for busy-wait delay loops
ADCSRA	ADC Control and Status Register A
ADIF	AD Interrupt Flag
ADMUX	ADC Multiplexer Selection Register
ADSC	AD Start Conversion
CPU_16MHz	Sets the CPU clock speed to 16 megahertz
COM0A1	Compare Output Mode
CS0i	Clock Select

DDRx	Data Direction Register, where x is the letter of the port
I2C	Inter-Integrated Circuit
ISR	Defines a function to register to interrupt vectors
MISO	Master In / Slave out for SPI
MOSI	Master Out / Slave In for SPI
OCR0x	Output Compare Register of port x
Pxi	Pin i of port x (e.g. PB7, PD6)
PINx	PORTx Input Pin Register, where x is the letter of the port
PORTx	Port x Data Register. Teensy++ 2.0 has six ports: PORTA, PORTB, PORTC, PORTD, PORTE, and PORTF.
SCL	I2C clock signal line
SCK	SPI clock signal line
SDA	I2C data signal line
Sei	Executes an assembler instruction to enable interrupts
SPI	Serial Peripheral Interface Bus
TCCR1x	Timer Counter Control Register, where x is the letter a of the port
TIFR	Timer Interrupt Flag Register
TIMER1_OVF_VECT	Timer Counter Overflow
TIMSK	Timer Interrupt Mask Register
TCCR1A	Timer Counter Control Register A
TCCR1B	Timer Counter Control Register B
TCNT1	Timer Counter Register
TOIE1	Timer/Counter0 Overflow Interrupt Enable
TX	Transmit pin used in UART
RX	Recieved pin used in UART
WGM0i	WaveForm Generation Mode

2. Resources

2.1 Hardware Resources

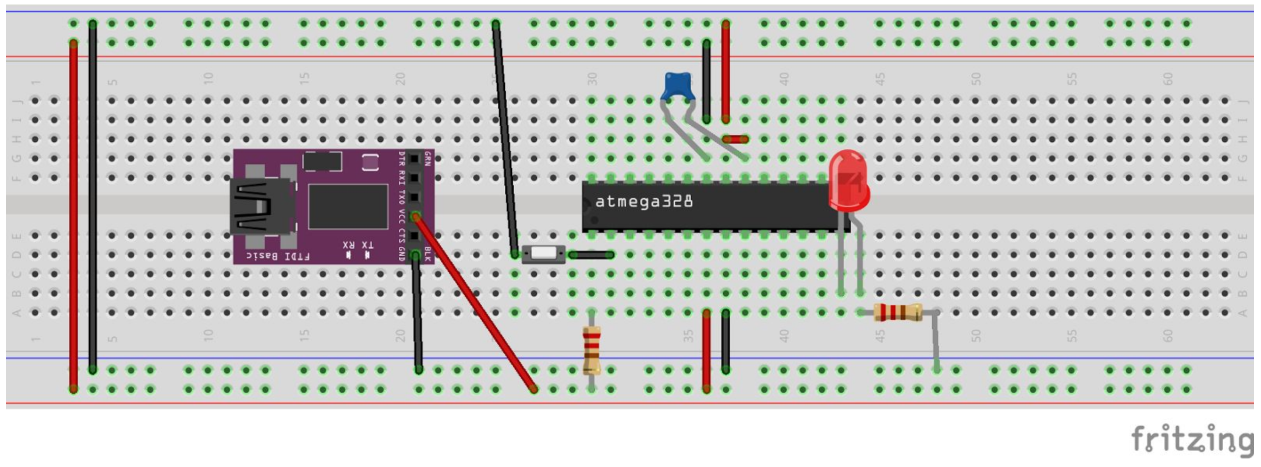
- ATMEGA328 x 2
- GP2Y0A02YK0F, distance measuring sensor unit
- USB cable
- Breadboard
- Light emitting diode (LED) pack
- Anode side of LED pack
- Dual inline package (DIP)
- Wires
- 4.7k Resistor x 2
- DEV-09716 FTDI
- usbtiny programmer

2.2 Hardware Setup

2.2.1 Lab 2

2.2.1.1 Programs 1, 2, 3

Programs one to three pretty much used a very similar setup so only one diagram was created. The LED is connected to pin PB0, while the INT0 pin is connected to a switch for the second and third programs. The FTDI was simply used for power.

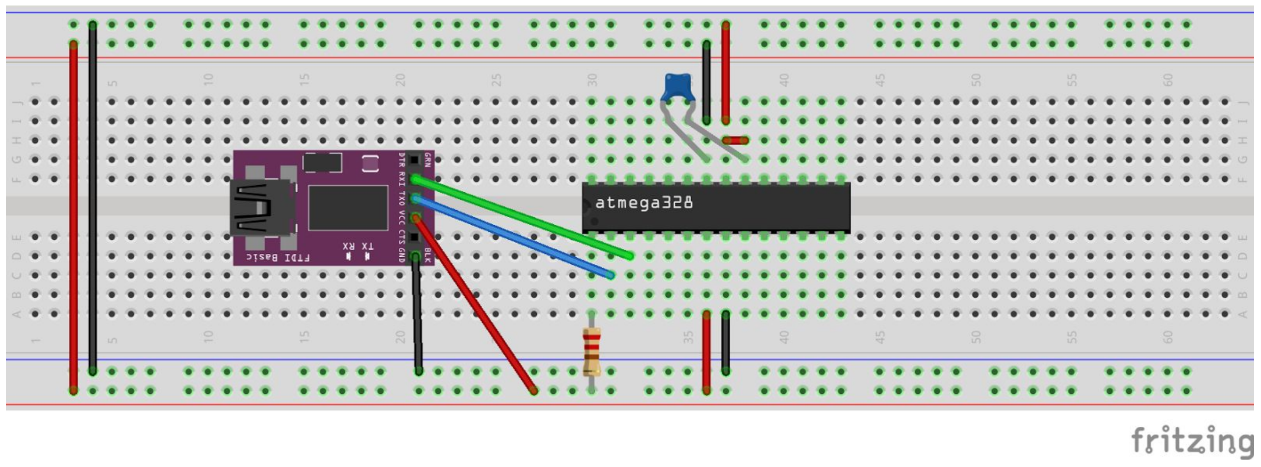


2.2.2 Lab 3

2.2.2.1 Program 1

The circuit for program 1 of lab 3 was pretty simple, the TX on the FTDI was connected to the RX pin on the ATMEGA and the RX on the FTDI was connected to the TX pin on the ATMEGA.

- The green wire is the FTDI TX to the ATMEGA RX
- The blue wire is the FTDI RX to the ATMEGA TX

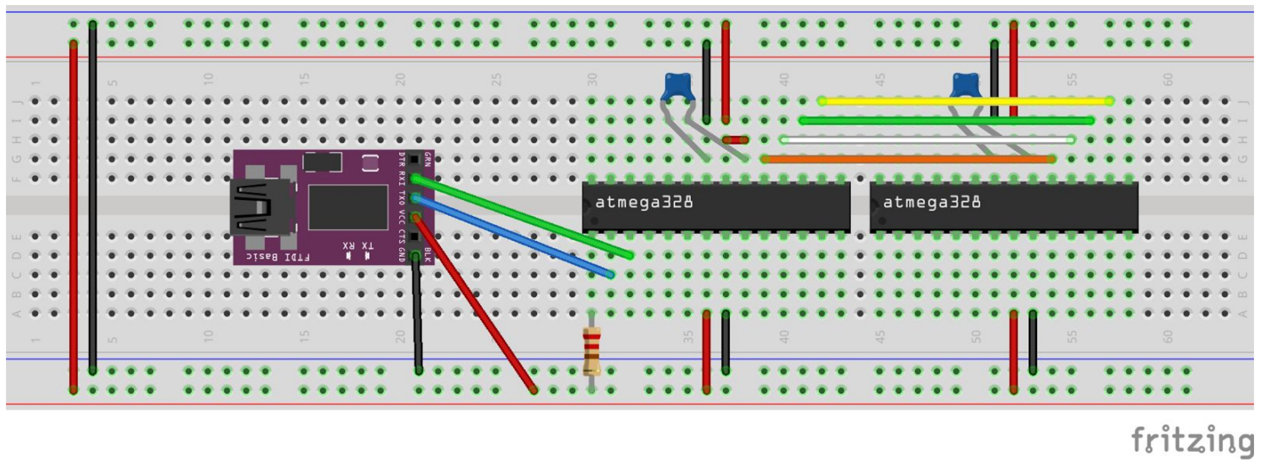


2.2.2.2 Program 2

Program required two ATMEGAS, the leftmost one was the master which was connected to the FTDI basic to output the values received from the slave on a computer. The ATMEGAS were connected together with four wires to allow them to use the SPI communication protocol.

- The orange wire is the SCK
- The white is the MISO
- The green is the MOSI

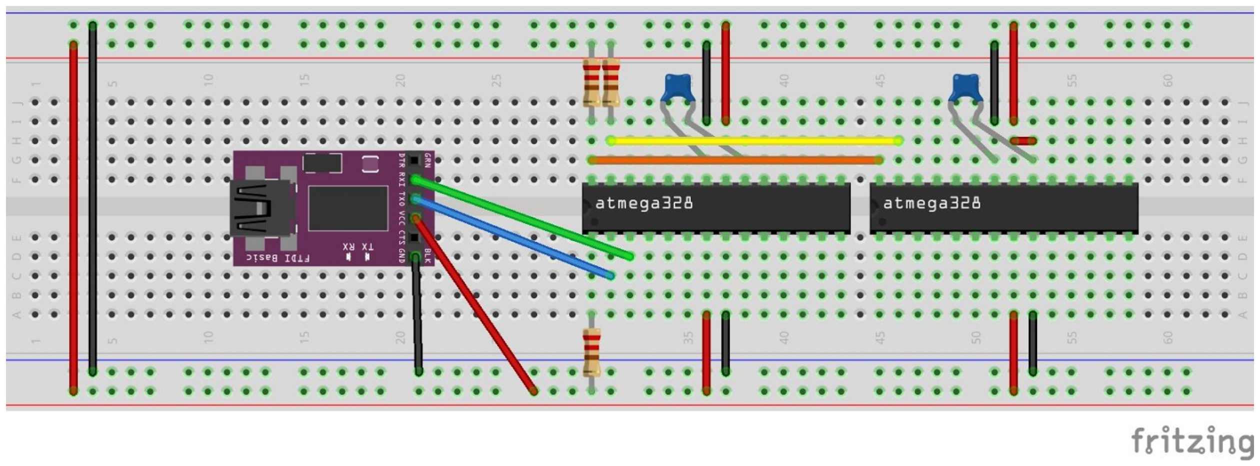
The yellow is the SS



2.2.2.3 Program 3

Program required two ATMEGAS, the leftmost one was the master which was connected to the FTDI basic to output the values received from the slave on a computer. The ATMEGAS were connected together with two wires to allow them to use the I2C communication protocol, on these two wires were pull-up resistors as required by the I2C protocol.

- The orange wire is the SCL
- The yellow is the SDA



2.2 Software Resources

- AVR-C development environment
- AVR-GCC for compiling then generate hex file from object file
  AVR-C programs can be loaded using the Teensy Loader through the
- Arduino IDE
- Fritizing

2.3 Software Setup

i. Install AVR-GCC, for Ubuntu Linux:

```
sudo apt-get install gcc-avr binultils-avr gdb-avr avr-libc make -y
```

ii. Include this library for AVR-C programs

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

iii. Compile the C code as:

```
avr-gcc -mmcu=atmega328 -O -o {output.o} {input.c}
```

iv. Generate hex file by

```
avr-objcopy -O ihex {output.o} {output.hex}
```

v. Load the HEX file by teensy loader file menu, press reset button on Teensy

To speed up this process I created a bash script to do the following steps above to any file. The script can be found below

```
avr-gcc -O1 -o out.o -mmcu=atmega328 $1.cpp
avr-objcopy -O ihex out.o out.hex
avrdude -c usbtiny -p m328 -U flash:w:out.hex
rm out.hex out.o
```

3. Programs

3.1 Part 1: Toggling an LED By Switch

3.1.1 Task

The first task is to toggle an LED by toggling a switch. The while loop in your main() function should be empty. The button must trigger INT0. You may use either an internal or external pull up resistor.

3.1.2 Results

This program was pretty simple, when the switch is closed the INT0 pin goes low which causes the interrupt to be called and toggles the LED, an internal pull-up resistor was also used.

3.1.3 Program

```
#define F_CPU 1000000
#include <avr/io.h>
#include <avr/interrupt.h>

int main()
{
	DDRB = (1 << 0); // Set PB0 as an output
	DDRD &= ~(1 << DDD2);     // Clear the PD2 pin
	// PD2 (PCINT0 pin) is now an input

	PORTD |= (1 << PORTD2);    // turn On the Pull-up

	EICRA |= (1 << ISC00);    // set INT0 to trigger on falling edge
	EIMSK |= (1 << INT0);     // Turns on INT0

	sei();                    // turn on interrupts

	while(1)
	{

	}
}
ISR(INT0_vect)
{
	cli();
	PORTB ^= (1 << 0); // Toggle the LED
	sei();
}
```

3.2 Part 2: Timer Controlled LEDs

3.2.1 Task

The next task is to toggle an LED by using a timer and an interrupt. The LED should toggle state every 5 seconds.

3.2.2 Results

Mode 4 was used with timer1, timer 1 was needed because timer 0 only counted up to 8 bits so it wasn't enough to generate a 5 second timer. The highest prescaler option was also used which was 1024, mode 4 which is CTC allowed the OCR1A to be set and compared against the timer's counter.

3.2.3 Program

```
#define F_CPU 1000000
#include <avr/io.h>
#include <avr/interrupt.h>


int main(void)
{
	DDRB = (1 << 0);
	// 4882 will give us a 5 second timer because
	// 1 000 000 the clock speed / 1024 prescaler will give us the required count for 1
second
	// times the 1 second value by 5 will give us a 5 second timer
	OCR1A = F_CPU / 1024 * 5;

	// set mode 4, CTC on OCR1A
	TCCR1B |= (1 << WGM12);

	//set interrupt on compare match
	TIMSK1 |= (1 << OCIE1A);

	// set prescaler to 1024 and start the timer
	TCCR1B |= (1 << CS12) | (1 << CS10);

	// enable interrupts
	sei();

	while (1)
	{
		// we have a working Timer
```

```
       }
}

ISR (TIMER1_COMPA_vect)
{
       // toggle led
       PORTB ^= (1 << 0);
}
```

3.3 Part 3: Button Controlled Timers

3.3.1 Task

The last task is to implement a method to toggle the timer used in Part 2. Assume that when the timer is stopped, it is reset back to 0.

3.3.2 Results

This program was just a combination of the other two, in the INT0 interrupt it simply checks if timer1 is enabled by checking if any bits in TCCR1B is set. If they're then the timer is running and we set TCCR1B to off to stop the timer, and if it's not then we just reset the timer info and start it again.

3.3.3 Program

```
#define F_CPU 1000000
#include <avr/io.h>
#include <avr/interrupt.h>



int main(void)
{
       DDRB = (1 << 0); // Set PB0 as an output
       DDRD &= ~(1 << DDD2);     // Clear the PD2 pin

       PORTD |= (1 << PORTD2);    // turn On the Pull-up
                                           // PD2 is now an input with pull-up
enabled

       EICRA |= (1 << ISC01);    // set INT0 to trigger on falling edge
       EIMSK |= (1 << INT0);     // Turns on INT0

       sei();                    // turn on interrupts

       OCR1A = F_CPU / 1024;
```

```c
        // set mode 4, CTC on OCR1A
        TCCR1B |= (1 << WGM12);

        //set interrupt on compare match
        TIMSK1 |= (1 << OCIE1A);

        // set prescaler to 1024 and start the timer
        TCCR1B |= (1 << CS12) | (1 << CS10);

        // enable interrupts
        sei();


        while (1)
        {
                // we have a working Timer
        }
}



ISR(INT0_vect) {
        cli();
        if (TCCR1B) {
                // set prescaler to none to stop the timer
                TCCR1B = (0 << CS12) | (0 << CS11) | (0 << CS10);
                // reset the counter value to zero
                TCNT1H = 0;
                TCNT1L = 0;
        }
        else {
                OCR1A = F_CPU / 1024;

                // set mode 4, CTC on OCR1A
                TCCR1B |= (1 << WGM12);

                //set interrupt on compare match
                TIMSK1 |= (1 << OCIE1A);

                // set prescaler to 1024 and start the timer
                TCCR1B |= (1 << CS12) | (1 << CS10);
        }
        sei();
}
ISR (TIMER1_COMPA_vect)
{
        PORTB ^= (1 << 0);
}
```

3.4 Part 1: USB/UART Communication

3.4.1 Task

The objective of this part of the lab is to be able to display 3 dierent greetings on the Serial Monitor. The user should be able to send a number and the ATmega328 should respond with a greeting. You may use the Serial Monitor in the Arduino IDE to interact with the ATmega328.

3.4.2 Results

The program simply waits for a value in the UDR0 register, if a value is received it checks the value to see if it's 0,1 or 2. If it is then the program will respond with a greeting by placing it in the UDR0 register and if an invalid value is received the program will respond with a message of valid values.

3.4.3 Program

```
#define F_CPU 1000000
#include <avr/io.h>

#define BAUDRATE 4800
#define BAUD_PRESCALLER (((F_CPU / (BAUDRATE * 16UL))) - 1)

void usart_init(void);
unsigned char usart_receive(void);
void usart_send( unsigned char data);
void usart_putstring(char* StringPtr);


char greet1[]="Greetings\n";
char greet2[]="Hello\n";
char greet3[]="Hey\n";

int main(void){
	usart_init();
	char info[] = "Enter a number from 0 to 2 for a greeting!\n";
	usart_putstring(info);
	while(1){
		// Read in a value from usart
		unsigned char recieve = usart_receive();
		if (recieve == '0')
			usart_putstring(greet1);
		else if (recieve == '1')
			usart_putstring(greet2);
		else if (recieve == '2')
			usart_putstring(greet3);
		else if (recieve != 10 && recieve != 13)
```

```
                        usart_putstring(info);
        }

        return 0;
}

void usart_init(void){
        // usart initialization code
        UBRR0H = (uint8_t)(BAUD_PRESCALLER>>8);
        UBRR0L = (uint8_t)(BAUD_PRESCALLER);
        UCSR0B = (1<<RXEN0)|(1<<TXEN0);
        UCSR0C = (3<<UCSZ00);
}

unsigned char usart_receive(void){
        while(!(UCSR0A & (1<<RXC0)));
        return UDR0;
}

void usart_send( unsigned char data){
        //wait for register to be ready
        while(!(UCSR0A & (1<<UDRE0)));
        UDR0 = data;
}

void usart_putstring(char* StringPtr){
        while(*StringPtr != 0x00){
                usart_send(*StringPtr);
                StringPtr++;}
}
```

3.5 Part 2 A: SPI Communication

3.5.1 Task

The Master/Slave configuration is a common device configuration. When multiple devices communicate, they tend to share a bus to communicate on. There are many types of communication protocols and buses available but in this lab, we will focus on two: I2C and SPI. The goal is to implement a slave microcontroller that will report the ADC value obtained from an IR sensor to the master. The master will then display that value on a serial monitor.

Implement the Master/Slave configuration using SPI.

3.5.2 Results

There are two programs, one for the master Atmega328 that simply reads the values received from the slave while sending a value due to the nature of SPI that requires a write and read to be done that the same time. When a value is received from the slave on the master, the master sends the value over uart so that it can be viewed on a computer. The slave program reads the sensor values and then converts it to ascii values so that it can transmit it to the master. This was a bit tricky to get working as at first I didn't understand that when I did a read from the master I also had to write a value to be able to read. This program could've used interrupts but I opted to use polling as it was a bit simpler and there was no need for interrupts as the program wasn't doing anything else besides communicating with each other and reading the sensor value.

3.5.3 Programs

```
spi_master.cpp
#define F_CPU 1000000
#include <avr/io.h>
#include <util/delay.h>

#define BAUDRATE 4800
#define BAUD_PRESCALLER (((F_CPU / (BAUDRATE * 16UL))) - 1)

void usart_init(void);
unsigned char usart_receive(void);
void usart_send( unsigned char data);
void master_spi_init(void);
unsigned char read_slave(char write);

int main(void){
	usart_init();
	master_spi_init();

	while(1){
		_delay_ms(5000);
		while(1) {
			char c = read_slave('c'); // read value of slave
			usart_send(c); // output over usart
			if (c == '\n')
			{
				break;
			}

		}
	}

	return 0;
}

unsigned char read_slave(char write)
{
	SPDR = write;
	while(!(SPSR & (1 << SPIF)));
	return SPDR;
}

void master_spi_init(void)
{
	// set the register for master
	DDRB |= (1 << 2) | (1 << 3) | (1 << 5);
	DDRB &= ~(1 << 4);

	SPCR |= (1 << MSTR);
	SPCR |= (1 << SPR0) | (1 << SPR1);
	SPCR |= (1 << SPE);
}
void usart_init(void)
{

	UBRR0H = (uint8_t)(BAUD_PRESCALLER>>8);
	UBRR0L = (uint8_t)(BAUD_PRESCALLER);
	UCSR0B = (1<<RXEN0)|(1<<TXEN0);
	UCSR0C = (3<<UCSZ00);
}

unsigned char usart_receive(void)
{

	while(!(UCSR0A & (1<<RXC0)));
	return UDR0;

}

void usart_send( unsigned char data)
{
	while(!(UCSR0A & (1<<UDRE0)));
	UDR0 = data;
}
```

spi_slave.cpp

```
#define F_CPU 1000000
#include <avr/io.h>
#include <string.h>
#include <stdlib.h>

int read_ir_sensor(uint8_t adctouse);
int write_to_master(char write);
int main (void)
{
	DDRB &= ~((1<<2)|(1<<3)|(1<<5));   // SCK, MOSI and SS as inputs
	DDRB |= (1<<4);                    // MISO as output

	SPCR &= ~(1<<MSTR);                // Set as slave
	SPCR |= (1<<SPR0)|(1<<SPR1);       // divide clock by 128
	SPCR |= (1<<SPE);                  // Enable SPI

	while(1)
	{
		int a;
		char str[4];
		a = read_ir_sensor(1); // read ir value
		itoa(a, str, 10); // convert to string
		int count = 0;
		while (count < strlen(str)) {
			write_to_master(str[count]);
			count += 1;
		}
		write_to_master('\n');
	}
}
int write_to_master(char write)
{
	SPDR = write;
	while(!(SPSR & (1 << SPIF)));
	return SPDR;
}

int read_ir_sensor(uint8_t adctouse)
{
	int ADCval;

	ADMUX = adctouse;          // use #1 ADC
	ADMUX |= (1 << REFS0);     // use AVcc as the reference
	ADMUX &= ~(1 << ADLAR);    // clear for 10 bit resolution

	ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);    // 128 prescale for 8Mhz
	ADCSRA |= (1 << ADEN);     // Enable the ADC

	ADCSRA |= (1 << ADSC);     // Start the ADC conversion

	while(ADCSRA & (1 << ADSC));      // Thanks T, this line waits for the ADC to finish


	ADCval = ADCL;
	ADCval = (ADCH << 8) + ADCval;    // ADCH is read so ADC can be updated again

	return ADCval;
}
```

3.6 Part 2B: I2C Communication

3.6.1 Task

The Master/Slave configuration is a common device configuration. When multiple devices communicate, they tend to share a bus to communicate on. There are many types of communication protocols and buses available but in this lab, we will focus on two: I2C and SPI. The goal is to implement a slave microcontroller that will report the ADC value obtained from an IR sensor to the master. The master will then display that value on a serial monitor.

Implement the Master/Slave configuration using I2C.

3.6.2 Results

There are two programs again, one for the I2C master and one for the I2C slave. The I2C master program simply starts the I2C process by setting signaling start on the SDA by setting the necessary registers, it then writes out on the SDA line the address of the slave devices it wants to communicate with followed by a bit set to 1 to indicate it wants the slave devices to write to it. It then expects four bytes from the slave before terminating and sending the stop signal on the SDA line. The master program also sends the values over uart so that they can be viewed on the computer. The slave program simply just waits for it to be addressed and when it is, it reads the sensor values and converts it to a string then sends the four bytes to the master. This program could've also been done using interrupts but again polling was just simpler and either program wasn't doing much besides communicating with each other. The program is also not that robust, there is no error handling so if something goes wrong the program will simply crash and need to be restarted. The twi_get_status could've been used to determine when an error occurs and handle it appropriately but it was necessary for this lab as this program is rather simple.

3.6.3 Programs

```
I2C_master.cpp
#define F_CPU 1000000
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>

#define BAUDRATE 4800
#define BAUD_PRESCALLER (((F_CPU / (BAUDRATE * 16UL))) - 1)
```

```
//Declaration of our functions
void usart_init(void);
void usart_send( unsigned char data);
void twi_init(void);
void twi_start(void);
void twi_stop(void);
void twi_write(uint8_t daya);
uint8_t twi_read(void);
uint8_t twi_read_ack(void);
uint8_t twi_get_status(void);
void write_error_message(void);

// READ bit is set for SLA + R

int main(void){
	usart_init();		//Call the USART initialization code

	DDRB |= (1<<0);
	twi_init();
	while(1){
		twi_start(); // Send I2C start signal on SDA line
		twi_write(0xCD); // Write the devices address which is 0xCC + 0x01, 0x01 is for
read mode
		usart_send(twi_read_ack()); // read and ack
		usart_send(twi_read_ack());
		usart_send(twi_read_ack());
		usart_send(twi_read()); // last read we do, so don't send ack so slave switches
off
		twi_stop(); // send I2C stop signal on SDA line
		PORTB ^= (1 << 0); // blink LED, used for debugging
		usart_send('\n');
		_delay_ms(1000); // one second pause
	}

	return 0;
}

void twi_init(void) {
	//set SCL to 400kHz
	TWSR = 0x00;
	TWBR = 0x0C;
	TWBR = 0x0C;
	TWCR = (1<< TWEN); // clear interrupt bit
}

uint8_t twi_get_status(void)
{
	uint8_t status;
	status = TWSR & 0xF8;
	return status;
}

void write_error_message(void)
{
	uint8_t t = twi_get_status();
	char str[2];
	itoa(t, str, 16);
	usart_putstring(str);
	usart_send('\n');
}

void twi_start(void) {
	TWCR = (1<< TWINT) | (1<< TWSTA) | (1<< TWEN);
	while ((TWCR & (1<< TWINT)) == 0);

}

void twi_stop(void) {
	TWCR = (1 << TWINT) | (1 << TWSTO) | (1<< TWEN);
}

void twi_write(uint8_t data)
{
	TWDR = data; // set the data register to the passed value
	TWCR = (1<<TWINT) | ( 1<< TWEN); // set twint and twen to allow hardware to write value
to SDA line
	while ((TWCR & (1<<TWINT)) == 0);
}

uint8_t twi_read(void)
{
	TWCR = (1<<TWINT)|(1<<TWEN);
	while ((TWCR & (1<<TWINT)) == 0);
	return TWDR;
}

uint8_t twi_read_ack(void)
{
	TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWEA);
	while ((TWCR & (1 << TWINT)) == 0);
	return TWDR;
}

void usart_init(void){

	UBRR0H = (uint8_t)(BAUD_PRESCALLER>>8);
	UBRR0L = (uint8_t)(BAUD_PRESCALLER);
	UCSR0B = (1<<RXEN0)|(1<<TXEN0);
	UCSR0C = (3<<UCSZ00);
}
```

```
void usart_send( unsigned char data){

	while(!(UCSR0A & (1<<UDRE0)));
	UDR0 = data;

}
```

I2C_slave.cpp

```
#define F_CPU 1000000
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>

#define BAUDRATE 4800
#define BAUD_PRESCALLER (((F_CPU / (BAUDRATE * 16UL))) - 1)

//Declaration of our functions
void usart_init(void);
unsigned char usart_receive(void);
void usart_send(unsigned char data);
void usart_putstring(char* StringPtr);
void twi_slave_init(void);
void twi_write(uint8_t daya);
uint8_t twi_get_status(void);
void write_error_message(void);
int read_ir_sensor(uint8_t adctouse);

char str[4];

int main(void) {
	usart_init();
	DDRB |= (1 << 0);
	PORTB |= (1 << 0);
	twi_slave_init();
	while (1) {
		while ((TWCR & (1 << TWINT)) == 0);

		for (int i = 0; i < 4; i++) {
			str[i] = 0x00;
		}

		int a = read_ir_sensor(1);
		itoa(a, str, 10); // convert int to string
		// Write out the converted int to string value
		for (int i = 0; i < 4; i++) {
			twi_write(str[i]);
		}
		PORTB ^= (1 << 0); // Debug led
		TWCR |= (1 << TWINT); // Clear the interrupt
	}

	return 0;
}

void twi_slave_init(void) {
	//set SCL to 400kHz
	TWSR = 0x00;
	TWBR = 0x0C;

	TWAR = 0xCC; // Set the slave address
	TWCR = (1 << TWEA) | (1 << TWEN); // Set to slave mode

}

uint8_t twi_get_status(void)
{
	uint8_t status;
	//mask status
	status = TWSR & 0xF8;
	return status;
}


void write_error_message(void)
{
	uint8_t t = twi_get_status();
	char str[2];
	itoa(t, str, 16);
	usart_putstring(str);
	usart_send('\n');
}

void twi_write(uint8_t data)
{
	TWDR = data;
	TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWEA);
	while ((TWCR & (1 << TWINT)) == 0);
}


void usart_init(void) {

	UBRR0H = (uint8_t)(BAUD_PRESCALLER >> 8);
	UBRR0L = (uint8_t)(BAUD_PRESCALLER);
	UCSR0B = (1 << RXEN0) | (1 << TXEN0);
	UCSR0C = (3 << UCSZ00);
}
```

```
unsigned char usart_receive(void) {

        while (!(UCSR0A & (1 << RXC0)));
        return UDR0;

}

void usart_send(unsigned char data) {

        while (!(UCSR0A & (1 << UDRE0)));
        UDR0 = data;

}

void usart_putstring(char* StringPtr) {

        while (*StringPtr != 0x00) {
                usart_send(*StringPtr);
                StringPtr++;
        }

}


int read_ir_sensor(uint8_t adctouse)
{
        int ADCval;

        ADMUX = adctouse;          // use #1 ADC
        ADMUX |= (1 << REFS0);     // use AVcc as the reference
        ADMUX &= ~(1 << ADLAR);    // clear for 10 bit resolution

        ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);    // 128 prescale for 8Mhz
        ADCSRA |= (1 << ADEN);     // Enable the ADC

        ADCSRA |= (1 << ADSC);     // Start the ADC conversion

        while (ADCSRA & (1 << ADSC));      // Thanks T, this line waits for the ADC to finish

        ADCval = ADCL;
        ADCval = (ADCH << 8) + ADCval;    // ADCH is read so ADC can be updated again

        return ADCval;
}
```

4. Conclusion

4.1 Interrupts and Timers

This lab provided more insight into the uses of interrupts and timers. Although we have already used them in previous labs with the teensy, it showed us how to use them on the ATMEGA328 and how to configure a timer to go off at specific intervals. It also introduced interrupts that are triggered due to some logic changes on an external pin.

4.2 Communication Protocols

Lab three introduced us to various communication protocols used in the world of embedded systems. UART showed us how a direct connection between two devices can be made, specifically an ATMEGA328 and our computer using the Sparkfun DEV-09716 FTDI. This protocol was probably one of the easier ones to understand due to the fact that it's only requires two lines, the TX for transmitting and the RX for receiving, and that it's only used for communication between two devices whereas the SPI and I2C are used for communication between multiple devices. SPI and I2C showed the limitations that UART had though, mainly that two lines restricted you to two devices and to require the two devices to agree on the same BAUD rate so that the data could be understood. SPI and I2C both have clock lines that all devices used which can help solve a lot of possible synchronization issues. It also shows how a protocol can be used to talk to many different devices. I2C showed that SPI is quited limited though, in regards to the number of devices it can talk to as the master device will require one extra pin on top of the other required for each slave it will need to communicate with.

Going forward I am now aware of various communication protocols that exists and when to use them. UART will mainly be used when I only need two communicate between two devices and can agree on a BAUD rate before. SPI will be used when I need a lightweight protocol to talk to a few devices whereas I2C will be used when I need to talk to a lot of devices.