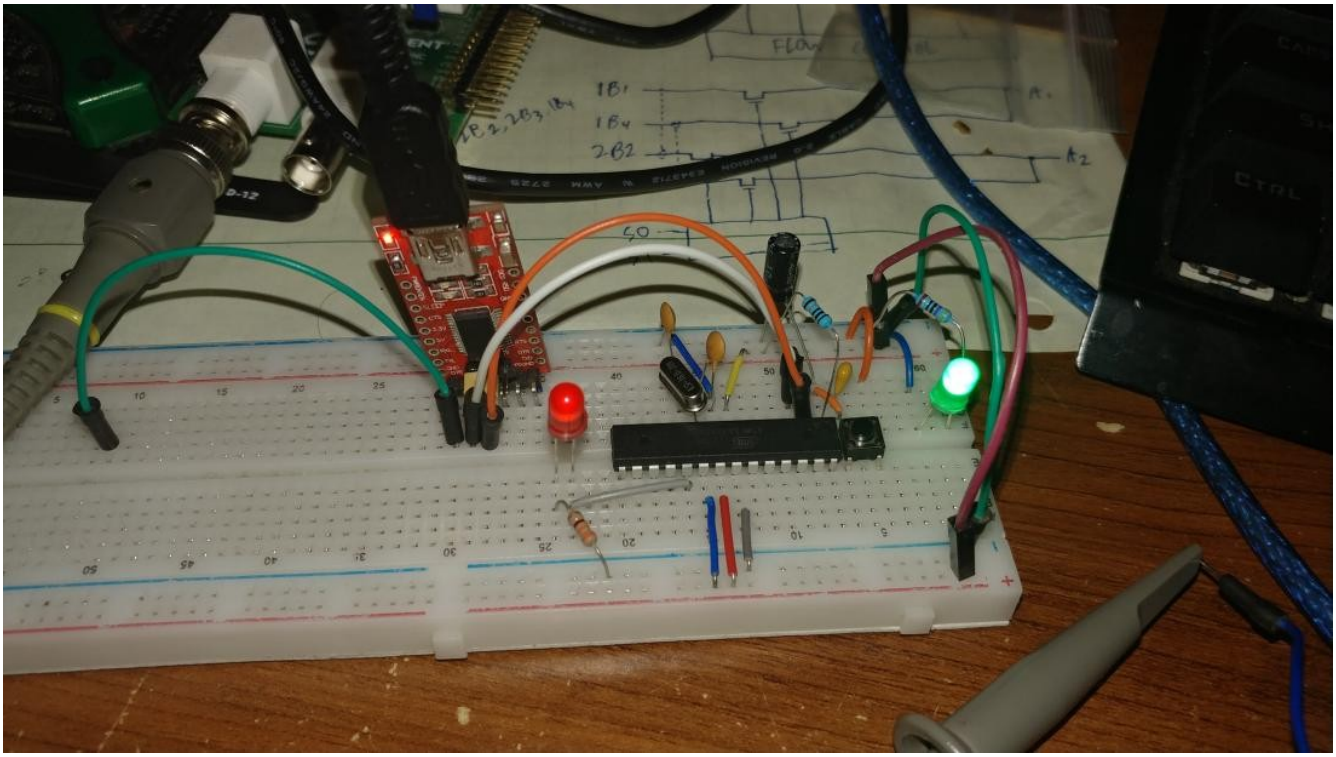
Table of Contents

Lab 1

Introduction

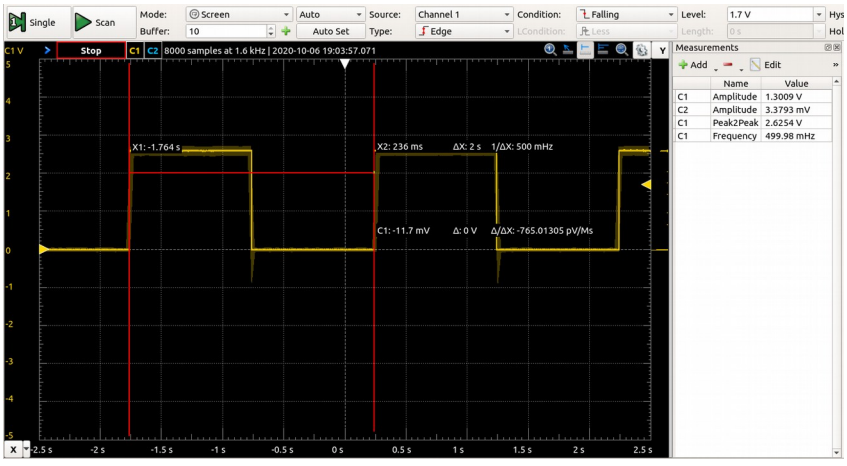
Objective: To build and test an ATMega328P based microcontroller on a breadboard. Basically, build an Arduino Uno, on a breadboard.

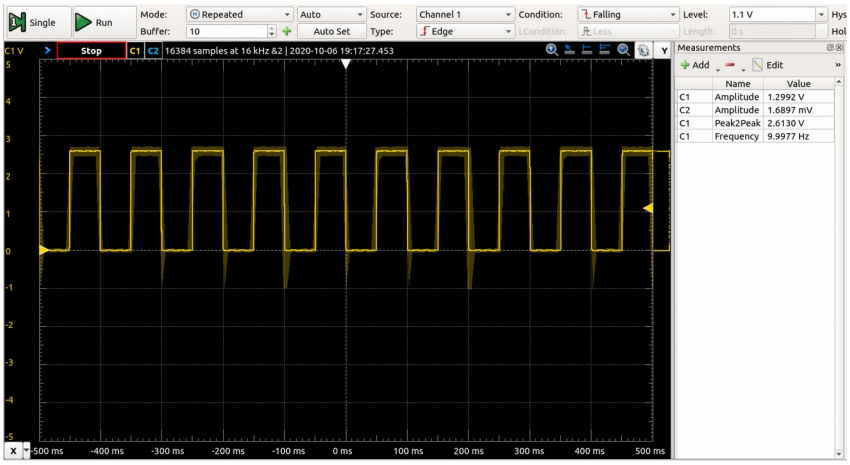
Breadboarding



Oscilloscope Captures – Breadboard

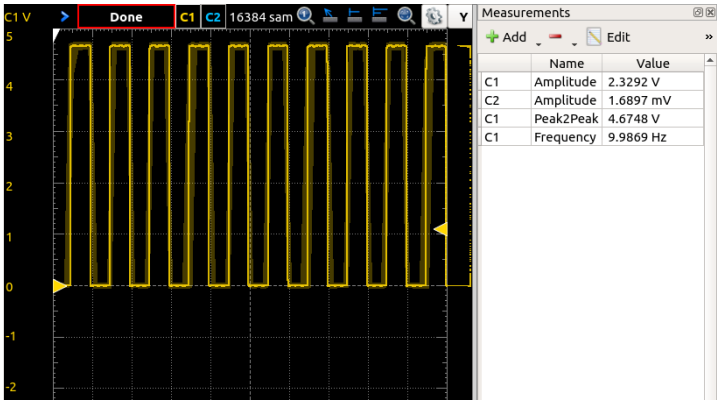
(Look at “C1 Frequency” under ‘Measurements’ on the right)

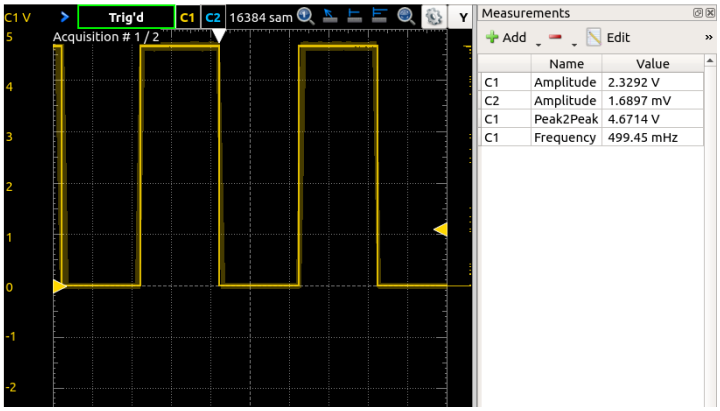


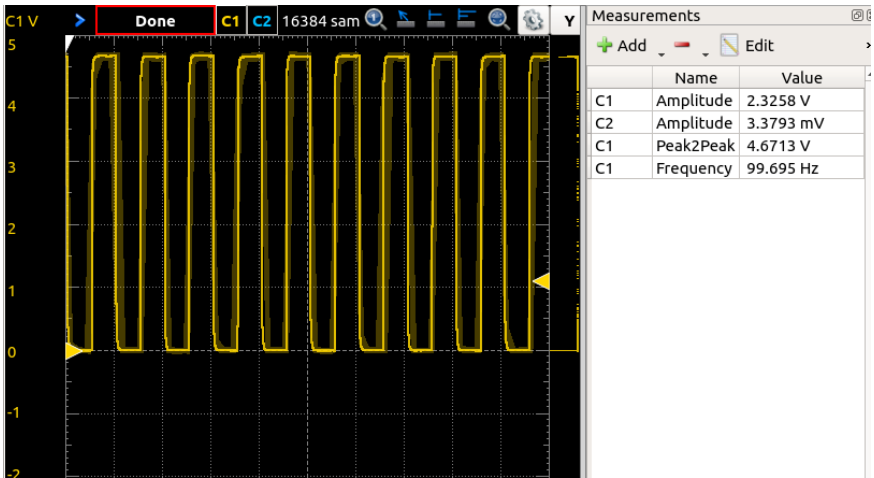


Oscilloscope Captures – Uno

(Look at “C1 Frequency” under ‘Measurements’ on the right)







Functions Descriptions

pinMode: Will set the specified pin as an input or output. Can also enable input pullup.

digitalWrite: Writes the specified digital pin to either 0V or 5V. Pin should first be configured with `pinMode`

delay: Will pause the program execution for the amount of time specified, in milliseconds. Useful in the blink sketch for changing the frequency.

Conclusion / Summary

In this lab I learned how to assemble and program an ATMega328p (the brain of the arduino Uno MCU) on a breadboard. I wasn't sure I had ATMega328p that had a bootloader already loaded on them, so at some point during the process I also burned the arduino bootloader using a bus pirate. I also discovered I had forgetten to tie my power rails together on the breadboard. Eventually, I got the Arduino to program with the FTDI USB adapter and, still having to press the reset button, can upload Arduino sketches to the MCU.

The Arduino maintains the code because it's flash memory gets programmed by the FTDI USB adapter. AVR has a different memory structure than other MCUs, like ARM based ones.

Extra Credit (Done on Linux)

```
#! /bin/bash
avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p -c -o led.o led.c
avr-gcc -mmcu=atmega328p led.o -o led
avr-objcopy -O ihex -R .eeprom led led.hex
avrdude -F -V -c arduino -p ATMEGA328P -P /dev/ttyACM0 -b 115200 -U flash:w:led.hex
```

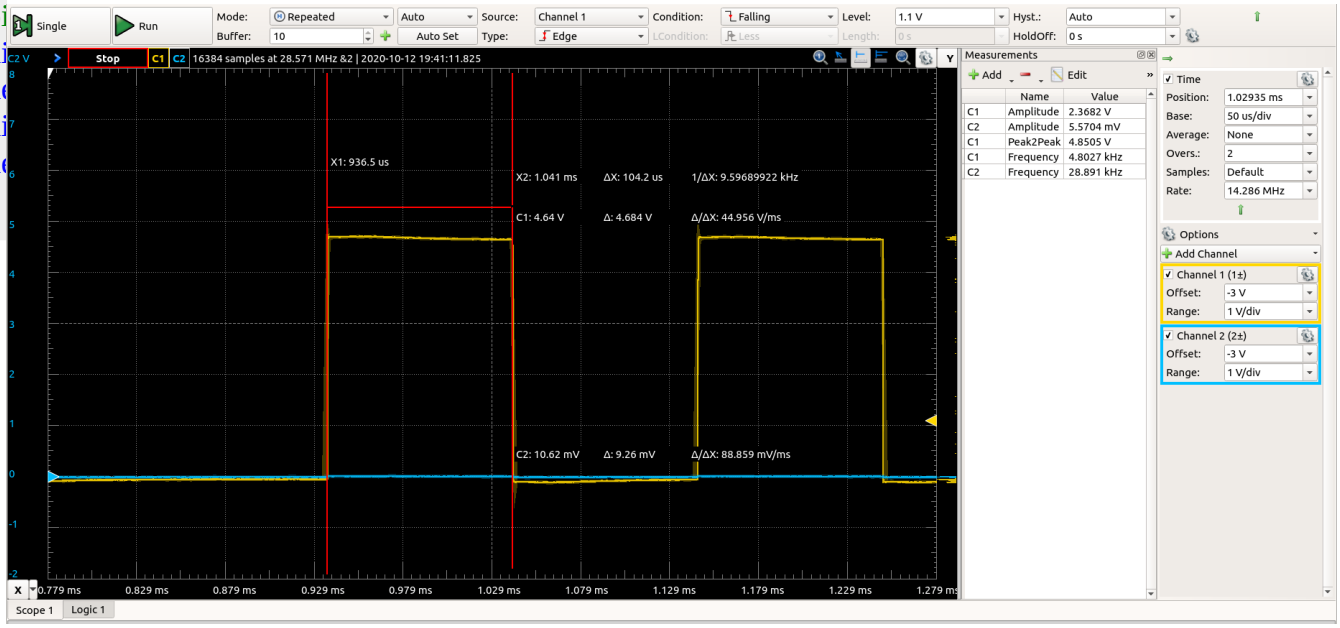
```
#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 123

int main (void)
{
 /* set pin 5 of PORTB for output*/
 DDRB |= _BV(DDB5);

 while(1) {
  /* set pin 5 high to turn led on */
  PORTB |= _BV(PORTB5);
  _delay_ms(BLINK_DELAY_MS);

  /* set pin 5 low to turn led off */
  PORTB &= ~_BV(PORTB5);
  _delay_ms(BLINK_DELAY_MS);
 }
}
```

Explanations / Summaries

Bitwise Operators: In C, bitwise operators contrast inputs bit-by-bit, the output being the literal logical comparison (01 & 01 = 01).
Macros: Macros in C are like bits of code that can be named and have their value changed, differently than how regular variables work.
Includes: Included files will essentially replace their '#include' code in your main program with the of the code in the header file.
The '**avr-gcc**' file will compile your arduino script with the correct parameters associated with it (Clock speed, MCU, etc). The '-o' specifies the output file, the '-c' specifies to not run the linker, and '-mmcu' is to specify the AVR instruction set or MCU type. In this case '**avr-objcopy'** converts the ELF program into a IHEX file ('-O' specifies output type), '-R' will remove the section name given ('.eeprom'). Finally, '**avrdude'** itself uploads the hex data to the arduino's embedded flash, with the correct protocols that the bootloader is expecting. The parameters '-F', '-V', and '-c' will force-override checking the device signature, disable automatically verifying upload, and specify the programmer id. '-p' specifies the part number, '-P' specifies the port, '-b' specifies the baud rate, and '-U' specifies the memory operation.

Appendix – Blink Code

```
#define BLINK_SPEED 5


// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
voi
  di
  de
  di
  de
}
```



Capture 1: ~9600 baud rate

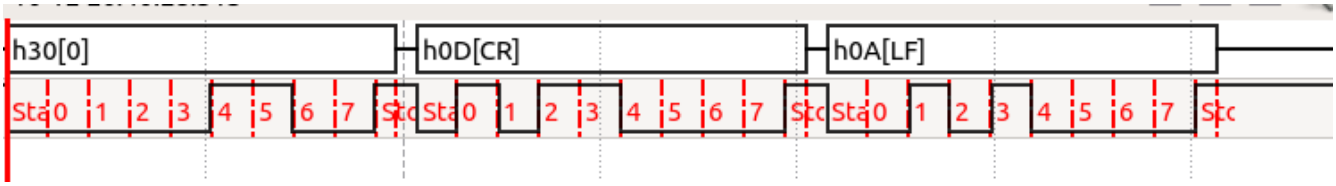
Lab 2

Introduction

Part 1a – RS232 Serial Transmit

First, using a slightly modified version of the “DigitalreadSerial” program, digital pin 2 of the Arduino is read and the state is written to the serial port and read on the PC. A ‘1’ appears when you wire pin 2 to HIGH (5v) and a ‘0’ appears when you tie pin 2 to LOW (GND, 0V).

With the oscilloscope, the TX signal from the Arduino is captured and the signal baud rate is measured. As expected, the bit rate is 9600, or about 104.2uS per bit.



Capture 2: Serial capture

It was discovered that besides the “’0”’ that the Arduino was sending, it would also send a carriage return and line feed (newline). A bit patterns of 0011_0000 (0x30) represents an ASCII ‘0’ character, and a 0x31 represents a ‘1’.

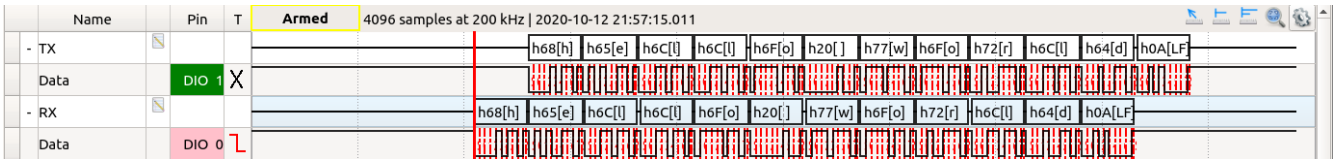
```
/*
  DigitalReadSerial

  Reads a digital input on pin 2, prints the result to the Serial Monitor

  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/DigitalReadSerial
*/

// digital pin 2 has a pushbutton attached to it. Give it a name:
int pushButton = 2;

// the setup routine runs once when you press reset:
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  // make the pushbutton's pin an input:
  pinMode(pushButton, INPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  // read the input pin:
  int buttonState = digitalRead(pushButton);
  // print out the state of the button:
  Serial.println(buttonState);
  delay(100);        // delay in between reads for stability
}
```

Code 1: DigitalReadSerial 100ms

Part 1b – RS232 Serial Transmit and Receive

This part of the lab involves sending and receiving data from the Arduino through it's (UART) serial port. The Analog Discover 2 from Digilent was used as a logic analyzer to spy on the serial protocol and decrypt the data being sent to and from the Arduino.

The sketch works by first initializing a serial connection to the PC 'Serial.begin(9600)', and then the loop function is repeatably executed. In the loop function, the 'if' statement checks if there is data available at the serial port which is buffered (64 byte buffer size). If data is available ('Serial.available()'), it's assigned to our 'data' variable with Serial.read() and then echo'd back to the serial port as a transmission ('Serial.print(data)';.



Capture 3: Data caught, and then sent back

Notice that there is an approximately 1.014ms delay between the start of the received data and the start of the echo'd data. This delay is probably due to the fact that the Serial.available() buffer waits for an entire transmission (8 bits) before returning a '1'. With this program, the transmission from the Arduino is precisely the same as what is sent to the device.

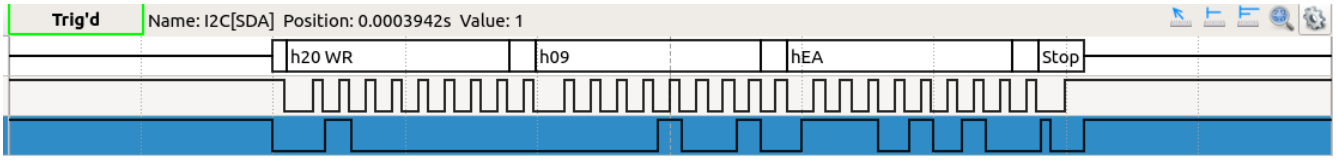
```
char data;

void setup() {
  Serial.begin(9600);

}

void loop() {
  /* Check if data has been sent from the computer: */
  if (Serial.available()) {
    /* read the most recent byte */
    data = Serial.read();

    /* ECHO the value that was read, back to the serial port. */
    Serial.print(data);
  }

}
```

Code 2: Echoing serial

Part 2 – I²C Port Expander

In this part of the lab, the objective is to connect and use a I/O (input/output) expander for the Arduino. The expander communicates with the two-wire I2C interface on the Arduino and provides an additional 8 GPIO pins that can be written to and read from. The address pins on the MCP23008 were tied low to make the device address 010 0000 (0x20, 32).

All eight of the GPIO pins on the MCP23008 chip were wired to LEDs with 330 ohm resisters in series. The program on the Arduino tells the MCP23008 to light the LEDs in a counter fashion from zero to 256 (0 LEDS to all 8 lit). This operation was verified and worked as expected.



Capture 4: I2C Capture of count update

```
#include "Wire.h";

void setup() {
  Wire.begin(); // Initializes the wire library

  Wire.beginTransmission(32); // transmit to device #32 (0x20)
  //device address in datasheet)
  Wire.write(byte(0x00)); // Address of the direction register
  Wire.write(byte(0x00)); // Write 0x00 to register (set to output)
  Wire.endTransmission(); // stop transmission

}

byte val = 0;

void loop() {
  Wire.beginTransmission(32); // Talk to our device
  Wire.write(byte(0x09));     // Specify the GPIO register
  Wire.write(val);            // write our value to the GPIO register
  Wire.endTransmission();     // End transmission

  val++;                      // Increment value by 1
  if (val == 256) {           // Check if value is greater than 8 bits
    val = 0;                  // Reset back to 0 if maxed out
  }
  delay(500);                 // End If statement

  /* It should be noted that the 'val' variable will overflow at 256 anyway,
   *  and probably doesn't need to be reset back to 0 manually.
   */
}
```

Code 3: I2C sketch

Part 3 – SPI Port Expander

In this part of the lab, we replace the I2C port expander with a near-identical device that uses the SPI serial protocol instead of I2C. I wired up the Arduino to the device with pin 13 going to SCK of the device, pin 12 going to SO, pin 11 going to SI, and pin 10 going to CS of the pin.

All eight of the GPIO pins on the MCP23S08 chip were wired to LEDs with 330 ohm resisters in series. The program on the Arduino tells the MCP23S08 to light the LEDs in a counter fashion from zero to 256 (0 LEDS to all 8 lit). This operation was verified and worked as expected.

```
#include <SPI.h>; // Include the SPI library from Arduino

const uint8_t csPin = 10;

void setup() {
  SPI.begin(); // Start up the SPI library
  pinMode(csPin, OUTPUT);       // Set digital pin 10 to an OUTPUT pin

  digitalWrite(csPin, LOW);     // Ground our chip select pin
  SPI.transfer(0x40);           // Specify the address of the device
  SPI.transfer(byte(0x00));     // Address of the direction register
  SPI.transfer(byte(0x00));     // Write 0x00 to register (set to output)
  digitalWrite(csPin, HIGH);    // Set our chip select pin high

}

byte val = 0; // Our incrementing counter variable.

void loop() {
  digitalWrite(csPin, LOW);     // Ground our chip select pin
  SPI.transfer(0x40);           // Specify the address of the device
  SPI.transfer(byte(0x09));     // Specify the GPIO register
  SPI.transfer(val);            // write our value to the GPIO register
  digitalWrite(csPin, HIGH);    // Set our chip select pin high

  val++;                       // Increment value by 1
  if (val == 256) {            // Check if value is greater than 8 bits
    val = 0;                   // Reset back to 0 if maxed out
  }                            // End If statement
  delay(500);                  // Delay by 500ms
}
```

Code 4: MCP23S08 SPI Sketch

Lab 3

The objective of this lab is to learn how to interface a microcontroller (Arduino Uno) with an LCD display and a Real-Time Clock IC.

Part 1

The example code (Hello World) was taken from the class website and uploaded to the Uno after wiring the microcontroller to the LCD module as per the instructions found on sparkfun's website and in the lab book. (https://learn.sparkfun.com/tutorials/basic-character-lcd-hookup-guide/all)

Figures 1 and 2 show the captured waveform from startup to the code loop in which the display is updated to show seconds since the last reset.

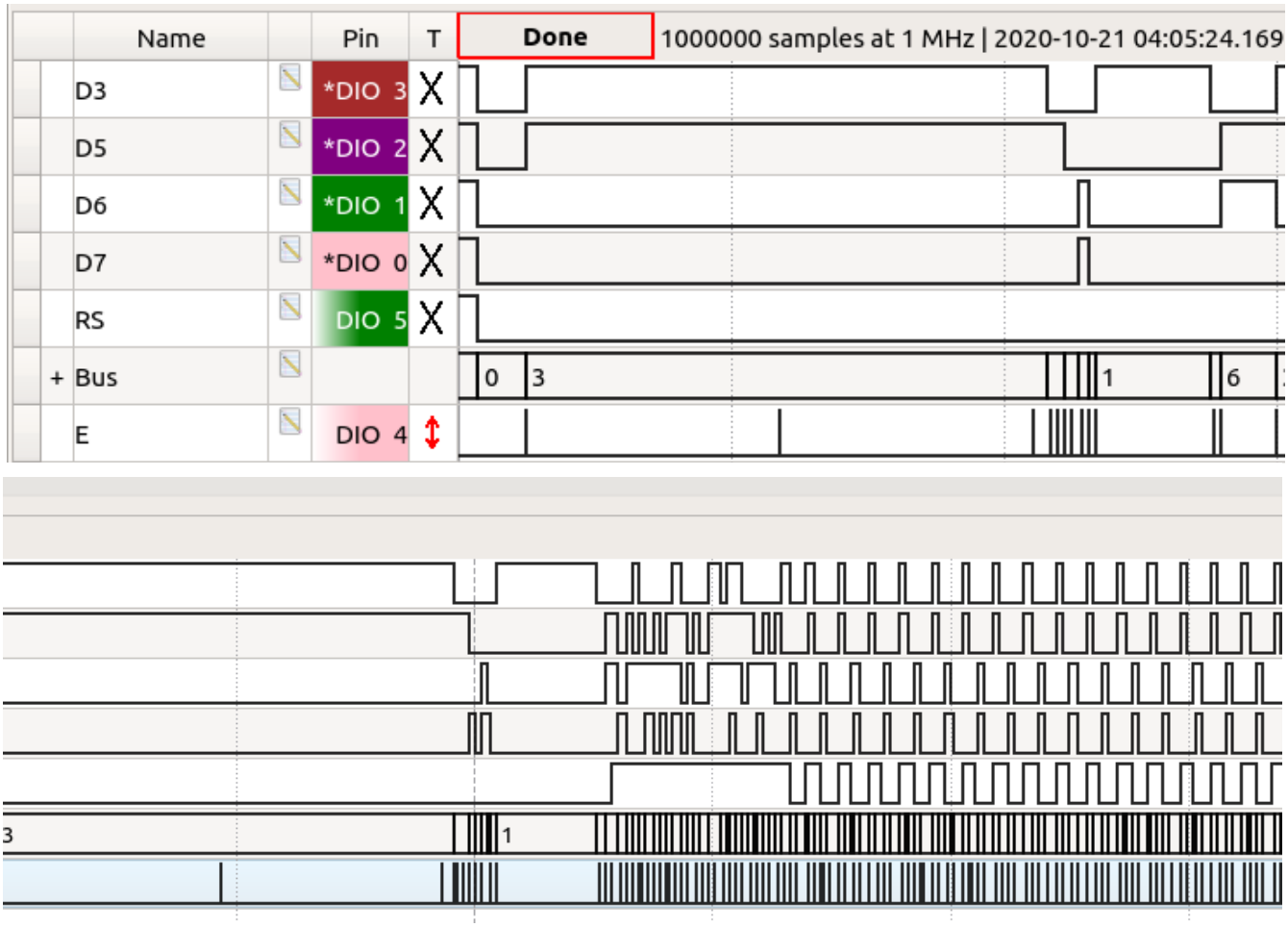


Figure 1: LCD Logic Waveform

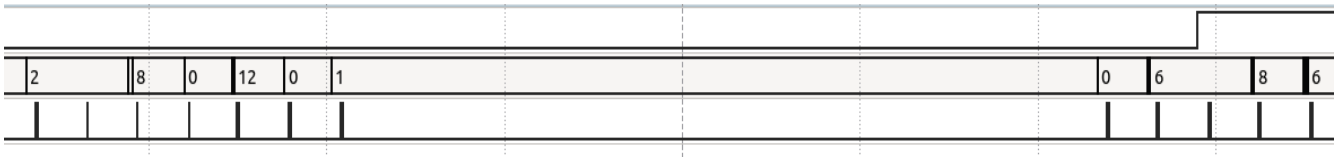


Figure 2: Detailed Capture

Function Descriptions

`lcd.begin(16, 2);`

Prior to figure 2, the command 0x3 is sent three times, which is part of the device initialization routine. Specifically this is 'set function, 8-bit data'. The last single nibble sent is 0x2, which signifies to the device that it is operating in 4-bit mode. The next nibbles (0x28 together), are a 'Function set' defining a two-line display, 4-bit, and 5x8 font. 0x0C (0-12) turns ON the display, but leaves off the cursor and blinking of cursor position. 0x01 (0-1) clears the display. Finally, 0x06 sets the entry mode of the display to have the cursor increment by one every write and not shift the display.

`lcd.print("hello, world!")`

The first nibble sent with RS high are 0x8 and 0x6, which reference the 'h' character in GCRAM, displaying it on the screen, every other character specified to the function is printed similarity.

`lcd.setCursor(0, 1)`

This function asserts the RS line LOW, addresses the DDRAM instruction and specifies that the cursor is now at position 0x40. Because the instruction is 0x8, the data sent becomes 0xC0 (12-0).

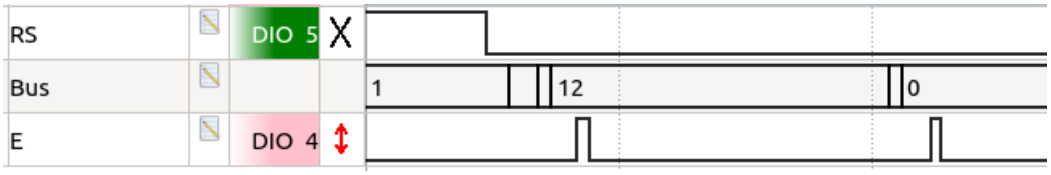
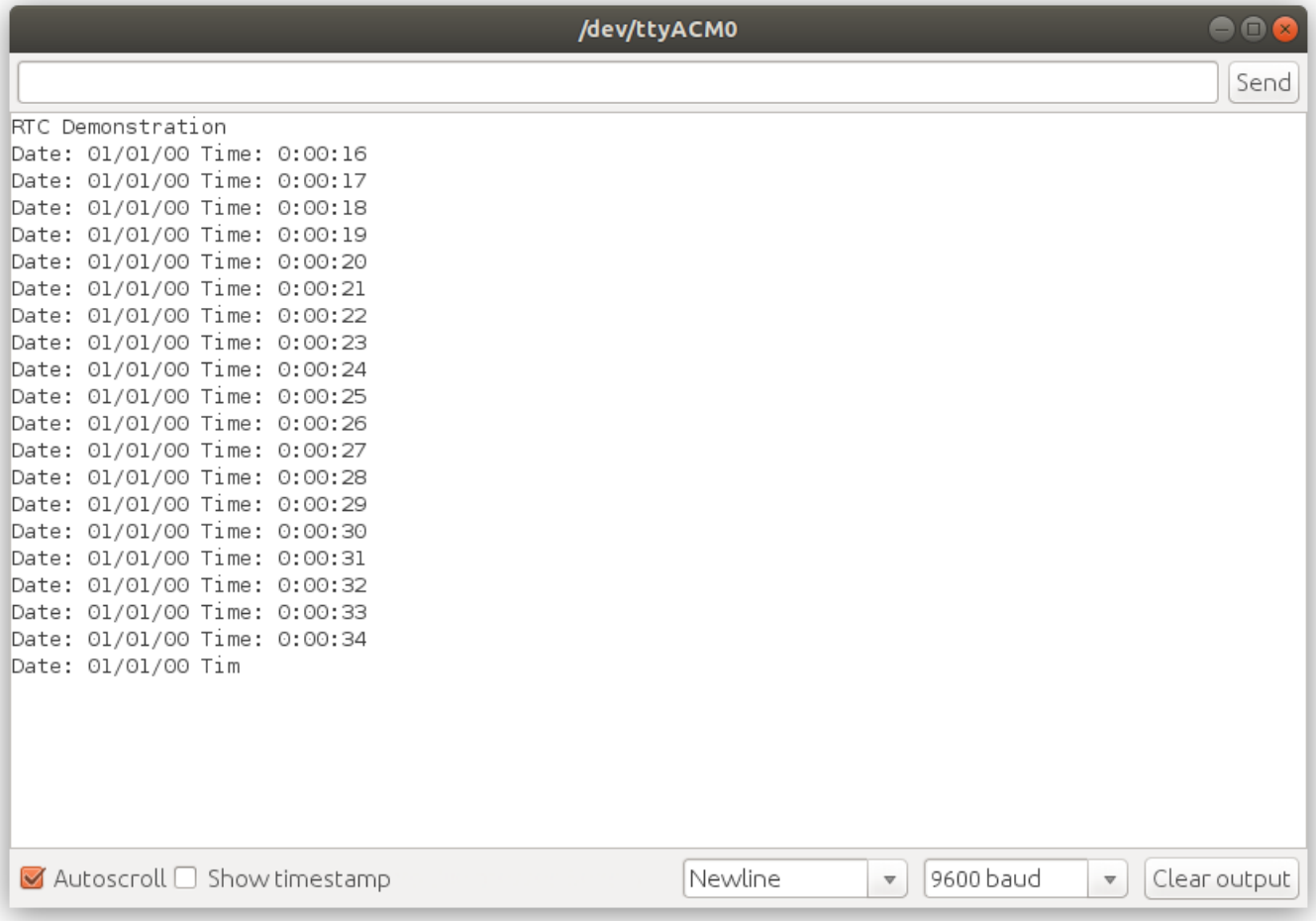


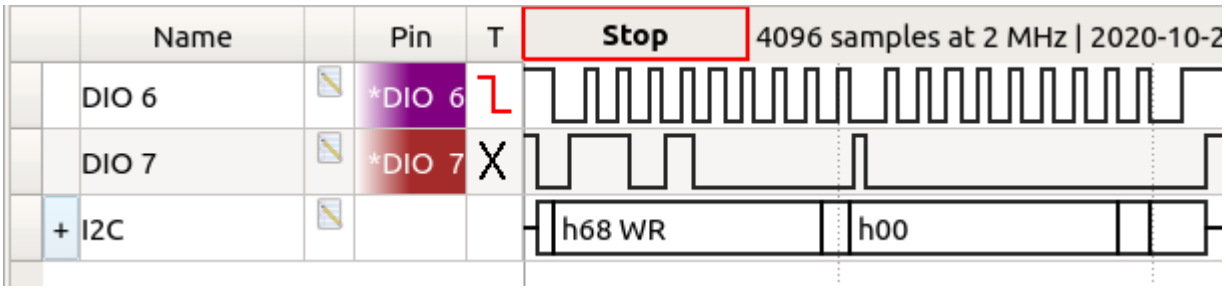
Figure 3: Next line

Part 2

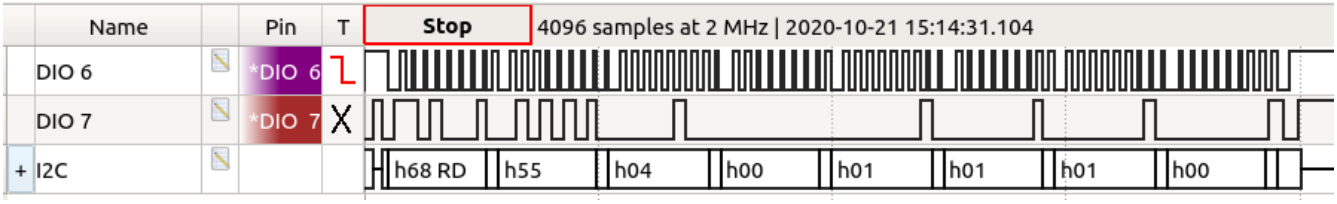
In this part of the lab we setup and serially communicate with a Real Time Cock integrated circuit. The example code (Real_Time_Clock.ino) was taken from the class page, minors errors fixed, and loaded into the Uno.



Capture 5: Time incrementing from RTC



Capture 6: Loop Start



Capture 7: Loop

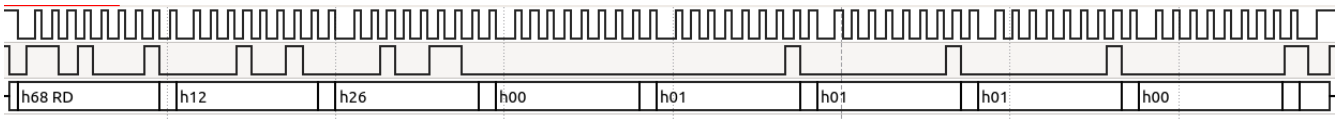
In capture 6, the 'loop' is repeating again. 0X68 is the address of the device, and the WR can means the LSB bit is clear (LOW), specifying a WRITE operation to the device. The data written is simply 0x00, or the address of seconds register in the RTC. This effectively sets up the RTC to sequentially read through all of it's timekeeping registers in the following READ operations.

The packet is 0x68, RD, which has the LSB bit set (HIGH), which signifies that the Arduino is reading data from the RTC. The 9th bit after the data is set to LOW by the RTC as an acknowledge, this happens after every read.

Capture 8: Seconds incrementing

Capture 8 is the next byte transmitted after the READ request, this data being the seconds register inside of the RTC. As expected, when operating normally, this data increments by 1 every second in the lower nibble. The upper nibble is the 10s digit in the seconds. In this case the seconds were 38 and 39.

The following bytes represent the minutes, hours, day of the week, date, month, and year values from the real time clock.



Capture 9: All reads

Notice that in capture 9 everything aside from the minutes and seconds is either zero or one. This is because when the device initializes, it has to be programmed with the current date and time. In order, 0x00 is hours, 0x01 is the day of the week, 0x01 is the date, 0x01 is the month, 0x00 is the year. Capture 10 shows the relevant post from the Uno.

```
Date: 01/01/00 Time: 0:26:11
Date: 01/01/00 Time: 0:26:12
Date: 01/01/00 Time: 0:26:13
```

Capture 10: Serial print from the microcontroller

Bit Operations

In the code, there are some cryptic lines involved in decoding the data from the RTC.

```
// Print date to serial port
Serial.print("Date: ");
Serial.print((month & 0x10)>>4);
Serial.print(month & 0x0F);
Serial.print("/");
Serial.print((date & 0x70)>>4);
Serial.print(date & 0x0F);
Serial.print("/");
Serial.print((year & 0x70)>>4);
Serial.print(year & 0x0F);
```

'`(month & 0x10) >> 4`' converts the month's "10" bit to an integer by masking the month data with 0b0001_0000, which when AND'd with the month data, will result in either 0b0001_0000 or 0b0000_0000. These two options represent if a month is before October (10th month), or after. These two options are then shifted to the right by 4 digits to place the singular bit (1 or 0) at the start of the byte to be printed to the serial monitor as a 0 or 1 instead of a 0 or 16. This is per the datasheet's description of the registers.

05h	0	0	0	10 Month	Month	Month	01–12

'`month & 0x0F`' similarly masks the month data (which is essentially 0b000X_XXXX) to select the lower four bits to represent 0b0000_XXXX. The Xs represent either a 0 or a 1, and are defined by the current month as reported from the RTC. For example, 0b0000_0001 would be January, 0b0000_0010 would be February, and so on. Once the data is converted to 0b0000_XXXX, it's simply printed to the serial console as an integer. 0X0000_0100 would get printed as 4, and so on.

For the month, date, and year, the same code is used to select the bits representing the 10s and 1s of the respective values. In the date and year, the bits describing how many 10s there are have two bits instead of one. Instead of 0b000X_0000 like the month, they use 0b00XX_0000, because there can be only 12 months, but 31 days and 99 years. There is actually probably an error in the example code here, as the 10s for the year should be 0xF0 instead of 0x70 like the date. 0x07 only allows a year up to 0b0111_1001, or 0x3F, or 79. Similarly, the date's mask may be too large.

The seconds, minutes, and hours data is all decoded similarly.

ADDRESS	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	FUNCTION	RANGE
00h	CH	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM 00–23
		24	PM/ AM							
03h	0	0	0	0	0	DAY			Day	01–07
04h	0	0	10 Date		Date				Date	01–31
05h	0	0	0	10 Month	Month				Month	01–12
06h	10 Year				Year				Year	00–99
07h	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08h–3Fh									RAM 56 x 8	00h–FFh

Capture 11: Registers

```
Wire.beginTransmission(0x68);
Wire.write(0);             // set the address
Wire.endTransmission();    // stop transmitting

byte sec;

Wire.requestFrom(0x68,1);  // read one byte
if(Wire.available() == 1)
  sec = Wire.read();  // read a byte from the buffer

Wire.beginTransmission(0x68);
Wire.write(0);             // set the address
Wire.write(sec & 0x7F);    // clear CH bit
Wire.endTransmission();    // stop transmitting
```

Code 5: Setup code

In Code 5, the device is configured so that the loop may operate correctly. First, the byte 0x00 is written to the device to set the register pointer to the 'seconds' register. Then, the line '`Wire.requestFrom(0x68,1);`` states the Uno is reading data from the RTC. Data is then read with '`sec = Wire.read();`', which puts the value of the seconds register in the '`sec`' variable. The Uno then sets the register back to 0x00 (seconds) and writes '`sec & 0x7F`', which is 0b0111_1111 & 0bXXXX_XXXX. This operation results in 0b0XXX_XXXX, which clears the CH bit. The CH bit, if set, stops the clock from operating.

Part 3

In this part of the lab, I took what I learned from the other two parts and made a digital clock using the LCD and RTC IC. The code can be found in the appendix, and Image 1 shows the clock in action, with the image perfectly timed with the seconds increment from 46 to 47.

In addition to reading from the RTC, functions were written to set the date and time on the device as well to make it a fully functional clock.

Image 1: Digital Clock

Extra Credit

In addition to setting the time and date on the RTC, a battery was connected to the Vbat pin on it to keep time even when the arduino was shut off.

Image 3: Before Shutdown

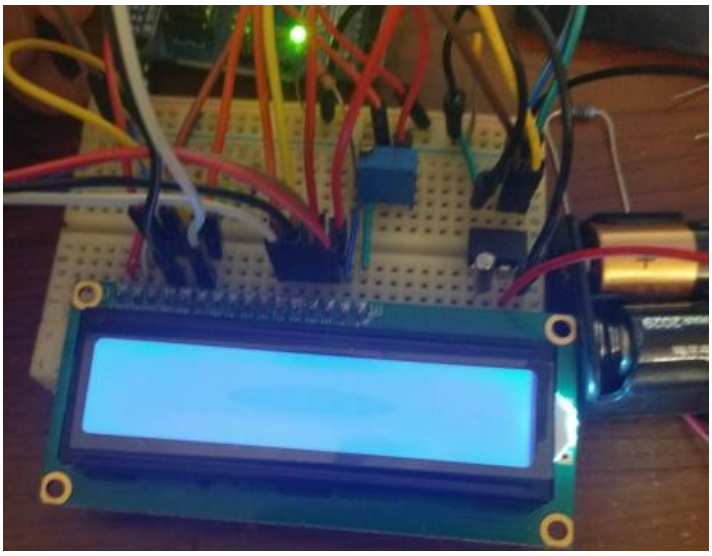


Image 2: During Shutdown

It's evident from Images 3 thru 4 that the device did not lose time during the power loss from the Arduino.

The two AA batteries together are the Vbat source and are wired to pin 3 of the RTC and to the common GND of the project.

The RTC can continue to operate without the Arduino UNO because of internal circuitry which disables the I2C bus if Vcc drops within 1.25 * Vbat, which is 4V in my case. The RTC continues to operate off of the battery even if the power fails and can operate for up to 10yrs on just 48mAh according to the datasheet.

Image 4: After Shutdown

Conclusion

Altogether, I learned how to program and use a standard LCD module and a real time clock. Both separately, and in use together to form a digital clock complete with offline timekeeping. In addition to the programs displays and discussed here, a sketch was written to directly initialize and display characters on the LCD by simple use of the digitalWrite() function in Arduino and the LCD's datasheet.

Appendix

DigitalClock.ino

```
#include <Wire.h>
#include <LiquidCrystal.h>

#define RTC_ADDR 0x68

#define RTC_SEC     0x00
#define RTC_MIN     0x01
#define RTC_HRS     0x02
#define RTC_DAY     0x03
#define RTC_DATE    0x04
#define RTC_MONTH   0x05
#define RTC_YEAR    0x06

void I2C_write(byte addr, byte regAddr, byte data, bool sendData = true);

const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

void setup() {
  Wire.begin();
  Serial.begin(9600);
  Serial.println("Digital Clock Init");

  lcd.begin(16, 2);
  lcd.print("hello, world!");

  I2C_write(RTC_ADDR, RTC_SEC, 0); // Set seconds to 0 to clear the CH bit
```

```
// Set time (Seconds, Minutes, Hours)
RTC_SetTime(30, 30, 16);

// Set date (Date, month, year)
I2C_write(RTC_ADDR, RTC_SEC, 0, false); // Set RTC register pointer to 0x00;

// Read in our values
byte seconds = I2C_readNextByte(RTC_ADDR);    // receive a byte as character
byte minutes = I2C_readNextByte(RTC_ADDR);    // receive a byte as character
byte hours   = I2C_readNextByte(RTC_ADDR);    // receive a byte as character
byte day     = I2C_readNextByte(RTC_ADDR);    // receive a byte as character
byte date    = I2C_readNextByte(RTC_ADDR);    // receive a byte as character
byte month   = I2C_readNextByte(RTC_ADDR);    // receive a byte as character
byte year    = I2C_readNextByte(RTC_ADDR);    // receive a byte as character


// Print and decode our date
lcd.setCursor(0, 0);
lcd.print("Date: ");
lcd.print((month & 0x10) >> 4);
lcd.print(month & 0x0F);
lcd.print("/");
lcd.print((date & 0x70) >> 4);
lcd.print(date & 0x0F);
lcd.print("/");
lcd.print((year & 0x70) >> 4);
lcd.print(year & 0x0F);

// Print and decode our time
lcd.setCursor(0, 1);
lcd.print("Time: ");
lcd.print(hours);
lcd.print(":");
lcd.print((minutes & 0x70) >> 4);
lcd.print(minutes & 0x0F);
lcd.print(":");
lcd.print((seconds & 0x70) >> 4);
lcd.print(seconds & 0x0F);

delay(1000);
```

```
}

// Write a byte to the I2C bus, optionally just write the register
// address
void I2C_write(byte addr, byte regAddr, byte data, bool sendData) {
  Wire.beginTransmission(addr);
  Wire.write(regAddr);
  if (sendData)
    Wire.write(data);
  Wire.endTransmission();
}

// Read a byte from the I2C bus
byte I2C_readByte(byte addr, byte regAddr) {
  byte ret;

  Wire.beginTransmission(0x68);
  Wire.write(regAddr);               // set the address
  Wire.endTransmission();     // stop transmitting
  Wire.requestFrom(addr, 1);  // read one byte
  if (Wire.available() == 1)
    ret = Wire.read();   // read a byte from the buffer
  return ret;
}
  RTC_SetDate(21, 10, 20);
}

void loop() {
  lcd.clear();

// Continue reading bytes from the I2C bus
byte I2C_readNextByte(byte addr) {
  byte ret;
  Wire.requestFrom(addr, 1);  // read one byte
  if (Wire.available() == 1)
    ret = Wire.read();   // read a byte from the buffer
  return ret;
}

// Set the time in the RTC IC
```

```
void RTC_SetTime(uint8_t seconds, uint8_t minutes, uint8_t hours) {
  uint8_t secTens = (seconds / 10 > 5) ? 0 : seconds / 10;
  uint8_t secOnes = seconds % 10;

  uint8_t minTens = (minutes / 10 > 5) ? 0 : minutes / 10;
  uint8_t minOnes = minutes % 10;

  Wire.beginTransmission(RTC_ADDR);
  Wire.write(0x00);
  Wire.write((secTens << 4) | secOnes);
  Wire.write((minTens << 4) | minOnes);
  Wire.write(hours);
  Wire.endTransmission();
}

// Set the date in the RTC IC
void RTC_SetDate(uint8_t date, uint8_t month, uint8_t year) {
  uint8_t dateTen = (date / 10 > 5) ? 0 : date / 10;
  uint8_t dateOnes = date % 10;

  uint8_t monthTens = (month / 10 > 5) ? 0 : month / 10;
  uint8_t monthOnes = month % 10;

  uint8_t yearTens = (year / 10 > 5) ? 0 : year / 10;
  uint8_t yearOnes = year % 10;

  Wire.beginTransmission(RTC_ADDR);
  Wire.write(0x04);
  Wire.write((dateTen << 4) | dateOnes);
  Wire.write((monthTens << 4) | monthOnes);
  Wire.write((yearTens << 4) | yearOnes);
  Wire.endTransmission();
}
```

Real_Time_Clock.ino

```
/*
 Real Time Clock Demonstration
 Allan Douglas
 Oregon Tech, 2017
 */
```

```
#include <Wire.h>

void setup() {

  Wire.begin();
  Serial.begin(9600);
  Serial.println("RTC Demonstration");

  Wire.beginTransmission(0x68);
  Wire.write(0);               // set the address
  Wire.endTransmission();      // stop transmitting

  byte sec;

  Wire.requestFrom(0x68,1);   // read one byte
  if(Wire.available() == 1)
    sec = Wire.read();   // read a byte from the buffer

  Wire.beginTransmission(0x68);
  Wire.write(0);               // set the address
  Wire.write(sec & 0x7F);      // clear CH bit
  Wire.endTransmission();      // stop transmitting
}

void loop() {
  Wire.beginTransmission(0x68);
  Wire.write(0);               // sends value byte
  Wire.endTransmission();      // stop transmitting

  Wire.requestFrom(0x68, 7);    // request 7 bytes from the RTC

  if(Wire.available() == 7)       // slave may send less than requested
  {
    byte seconds = Wire.read();    // receive a byte as character
    byte minutes = Wire.read();    // receive a byte as character
    byte hours   = Wire.read();    // receive a byte as character
    byte day     = Wire.read();    // receive a byte as character
    byte date    = Wire.read();    // receive a byte as character
    byte month   = Wire.read();    // receive a byte as character
```

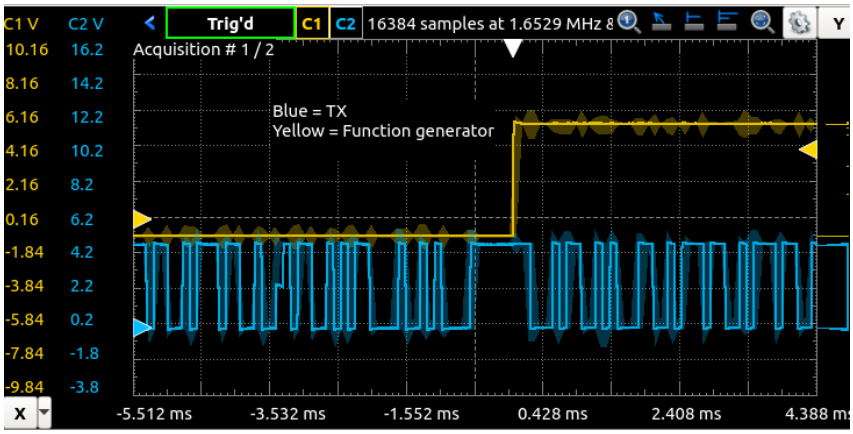
```
    byte year    = Wire.read();    // receive a byte as character

    // Print date to serial port
    Serial.print("Date: ");
    Serial.print((month & 0x10)>>4);
    Serial.print(month & 0x0F);
    Serial.print("/");
    Serial.print((date & 0x70)>>4);
    Serial.print(date & 0x0F);
    Serial.print("/");
    Serial.print((year & 0x70)>>4);
    Serial.print(year & 0x0F);

    // Print time to serial port
    Serial.print(" Time: ");
    Serial.print(hours);
    Serial.print(":");
    Serial.print((minutes & 0x70)>>4);
    Serial.print(minutes & 0x0F);
    Serial.print(":");
    Serial.print((seconds & 0x70)>>4);
    Serial.print(seconds & 0x0F);
    Serial.println("");
  }

  delay(1000);
}
```

Lab 4

The objective of this lab is to use the interrupts and timers on the Arduino Uno. This allows us to have the microcontroller stop normal execution to deal with external changes in a circuit, or to stop normal execution to deal with a timed task.

Part 1 – Interrupt Speed Test

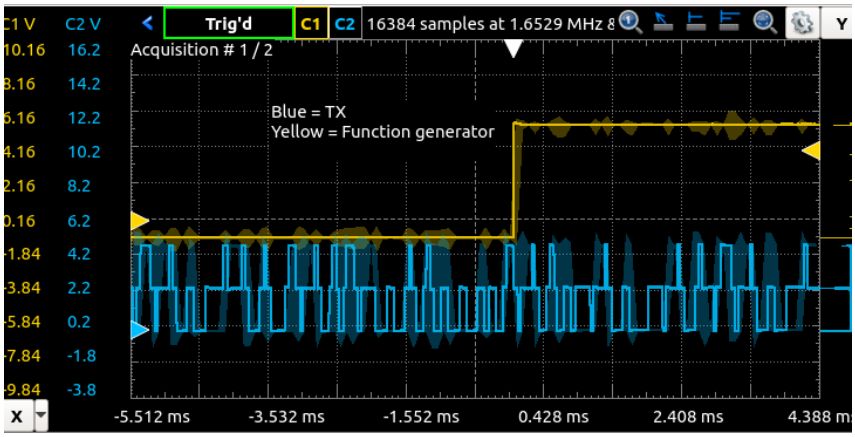
The Uno was programmed to increment a count by one when a pin changed values. The change on digital pin 2 is sent as an interrupt to the Uno and handled in a special function.



Capture 12: Right before skipping occurs

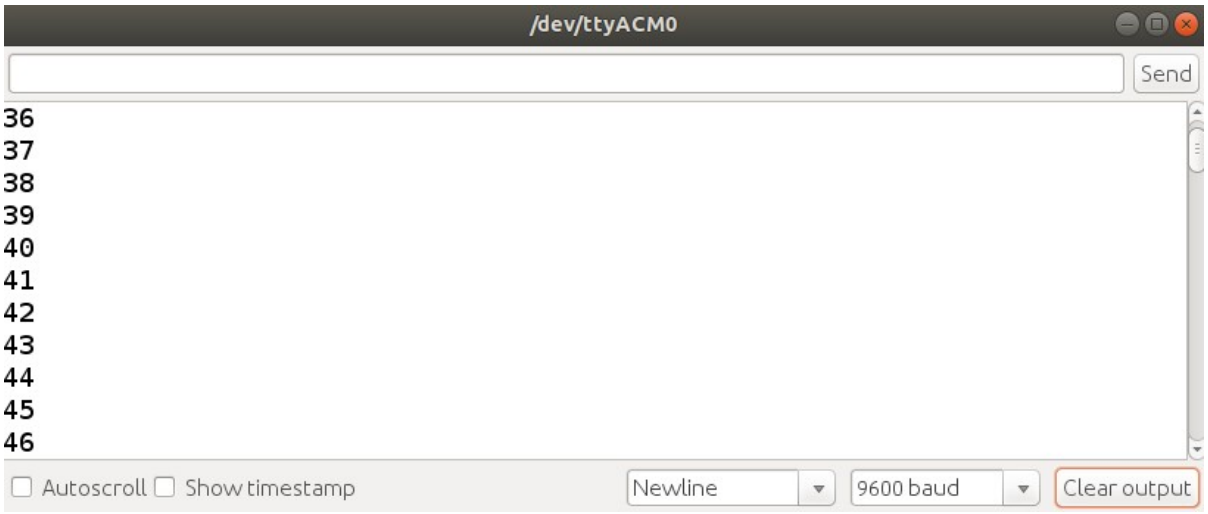
The **interrupts()** function will re-enable the Arduino's interrupts if they've been disabled. The **attachInterrupt()** function allows you to tie one of your functions to a specific interrupt source, such as an external pin state.

The **count** and **prev_count** variables in the code must be defined as volatile because otherwise the compiler might assume that, per normal operation, the variable might never change. By making the variables volatile we make sure the compiler knows that the variables can be changed at any time (due to the interrupts primarily).

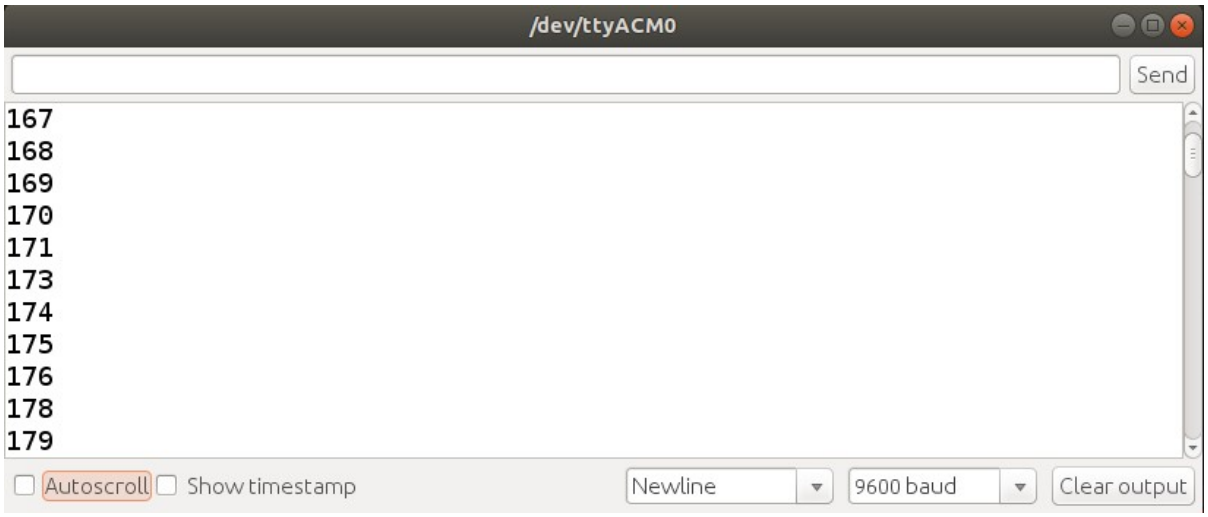


Capture 13: After Skipping

It's observed in the operation of the code that there's a maximum speed at which the function generator can run before the count on the serial monitor starts to skip values. Captures 15 and 16 show the serial monitor's values, and captures 12 and 13 show the function generator waveform (yellow) and the Arduino's TX waveform (blue).



Capture 14: Serial monitor without skipping



Capture 15: Serial monitor with skipping

There are skips in the count because the process of sending data to the computer eventually takes longer than the time it takes to increment the counter in the interrupt. The interrupt starts firing more than once during the time when the serial bus is active, and has not yet finished sending the last number. The baud rate will effect how fast the serial data is communicated back to the computer, and if we increase it, we should be able to increase the interrupt rate as well without any skips in the count.

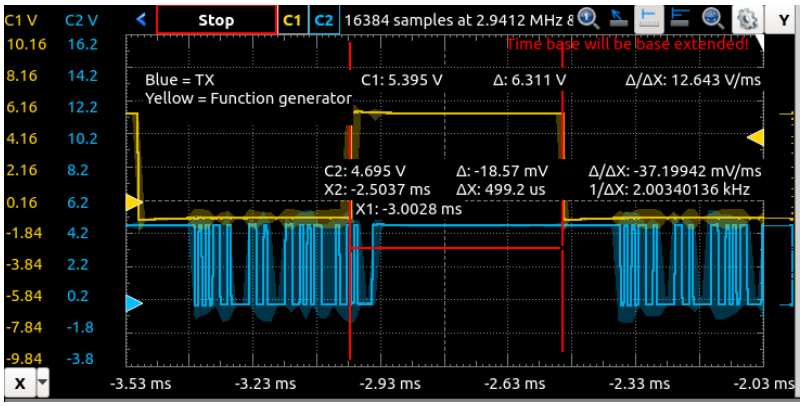
If we set the baud rate to 115200, we can increase the function generator frequency to approximately 690Hz before skipping occurs again. Again, the entire number is communicated to the computer much faster faster with the higher baud rate and so the interrupt rate can be much higher before triggering twice in a single serial transmission.

Part 2 – Measuring Pulse Duration

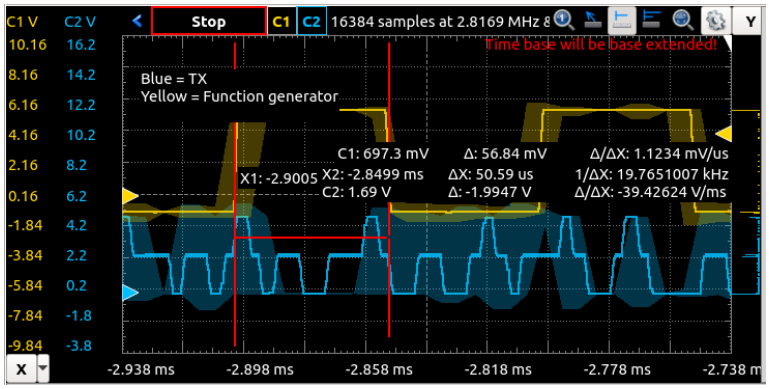
In this part of the lab, I look at using the internal timers or interrupts of the Arduino to measure the duration of a pulse from a function generator. The provided 'pulse_timer' sketch was taken from the class website and used.

The **pulse()** function serves to read the sense pin and set the RISING and FALLING edge times with use of the **micros()** function. The **setup()** function initializes a higher speed serial connection [**Serial.begin(115200)**], sets the input pin mode, attaches the ISR to interrupt 0, and sets the update flag to zero. The command **attachInterrupt(0, pulse, CHANGE)** will call the function '**pulse**' whenever the '**0**' interrupt is called, in this case a '**CHANGE**' on INT0, which is digital pin 2. The update flag is used in the **loop()** function to know when to send the pulse width to the computer via the Serial port. Once the loop is finished sending the data to the computer, the update flag is reset. The update flag is only set once the pulse goes HIGH and then LOW again. Timer/Counter0 is used for the **delay()**, **millis()**, and **micros()** functions.

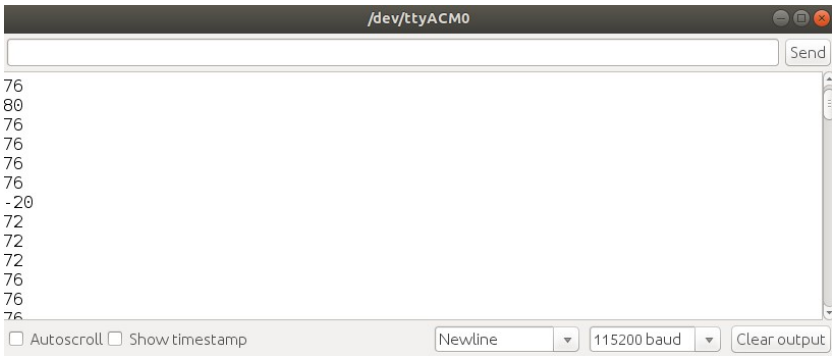
We set the function generator to a 1kHz 5VPP square wave with a 50% duty cycle. In measuring the waveform ON time with the oscilloscope, we find it to be exactly 500uS, which is what the value is reported by the Arduino (Capture 16, delta X). If we adjust the function generator frequency to 10kHz, the oscilloscope reports an ON time of 50uS, but the Arduino reports values between 48 and 52 (Capture 17, delta X).



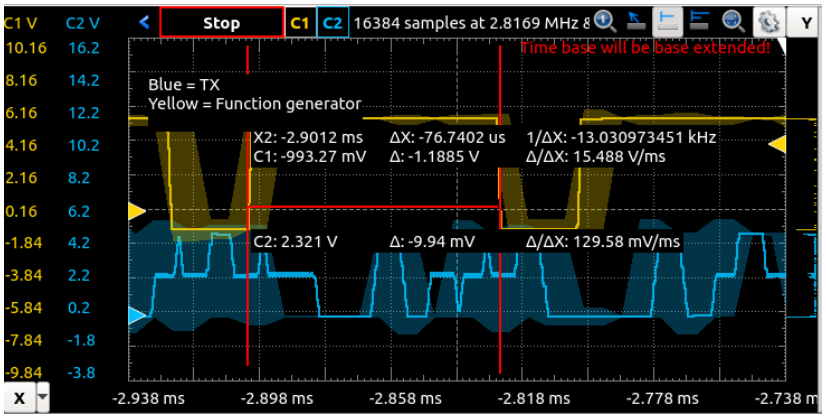
Capture 16: 1kHz



Capture 17: 10kHz



Capture 18: Negative numbers



Capture 19: Oscilloscope capture

If the duty cycle on the function generator is increased to approximately 75%, we begin seeing negative numbers reported through the serial monitor. Most other measurements are nearly correct, with 75uS being the actual ON time and 76-80 being the reported values by the Arduino. A negative number is reported when the Arduino calculates that the **endTime** was BEFORE the **startTime**. This condition can occur if the Arduino first reads a FALLING edge before it reads a RISING edge in the interrupt service routine.

Conclusion

In this lab I learned the basics of setting up and using the external interrupts on the Arduino Uno. The hardest part of the lab was probably in figuring out how the second program worked and understanding how negative numbers could be reported. In addition to learning about pin interrupts, it was useful to research how the timers are used in delay(), micros(), and millis().

Appendix – Part 1

```
/*  Pulse Duration Timer
 *  Oregon Institute of Technology, 2015
 *
 *  Allan A. Douglas
 */

volatile boolean update_flag;
volatile unsigned long startTime;
volatile unsigned long endTime;
int sensePin = 2;  // interrupt source

// interrupt service routine
void pulse() {
    if (digitalRead(sensePin)) {
      startTime = micros();
      update_flag = 0;
    }
    else {
      endTime = micros();
      update_flag = 1;
    }
}

void setup() {
  Serial.begin(115200);
  pinMode(sensePin, INPUT);
  attachInterrupt(0, pulse, CHANGE);
```

```
  update_flag = 0;
}

void loop() {
  if (update_flag) {
    Serial.println (long(endTime - startTime));
    update_flag = 0;
  }
}
```

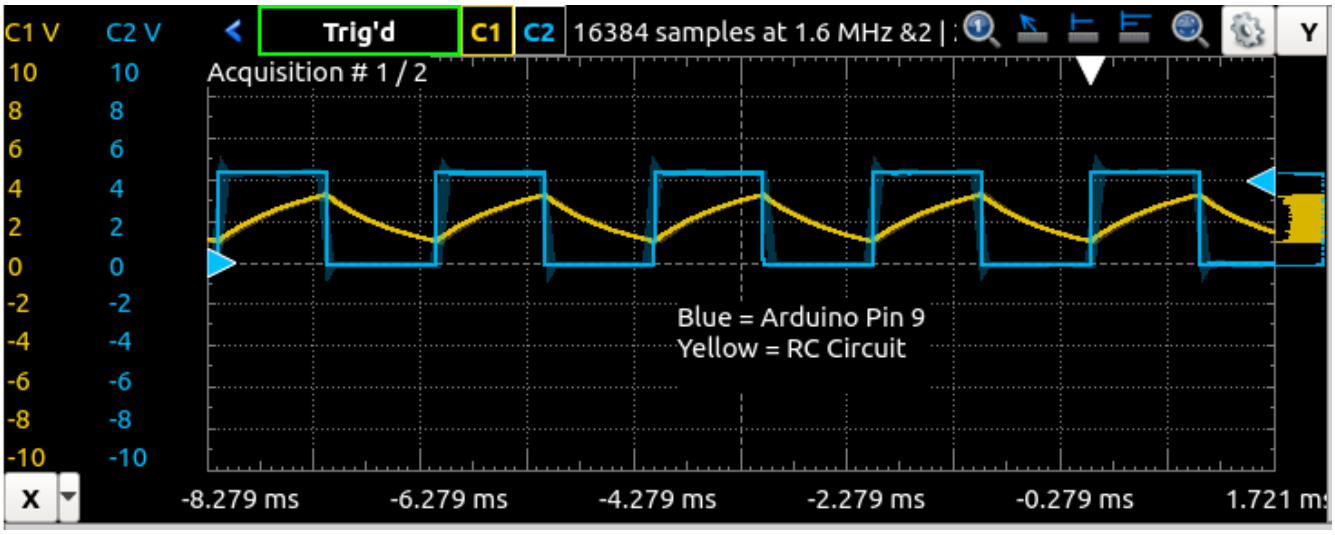
Lab 5

Introduction

The objective of this lab is to use the internal timers in the Arduino Uno to do Pulse Width Moduluation, controlling some external circuitry. We create a variable DC voltage, vary the intensity of an LED, and control the speed of a DC motor.

Part 1

In part 1 of this lab, two different methods of creating PWM are used and their accuracy compared. First, the Arduino function “analogWrite()” is used to create a 50% duty cycle wave.



Capture 20: Arduino Output

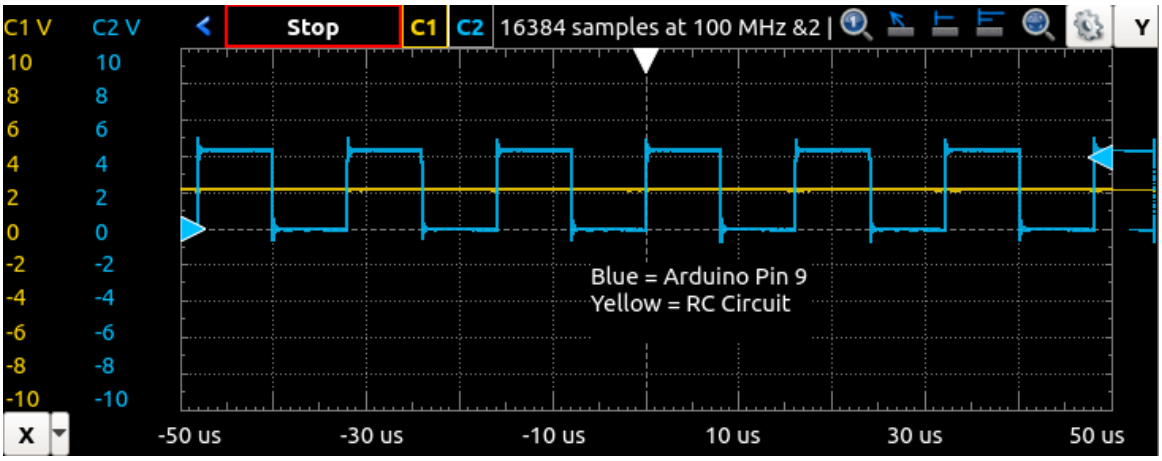
Afterwards, the PWM value was stepped from 0 to 255 (0-100% Duty Cycle) in steps of 15. The average value of voltage over the capacitor was measured and recorded in Table 1. A higher duty cycle resulted in a higher capacitor voltage, because the average ON time of the output wave was greater, charging the capacitor more than it was discharging it.

Taking into account the output voltage being less than 5V, we calculate the expected capacitor voltage with 4.6V instead of 5V. 4.6V was the maximum output voltage from the timer pin. Tables 1 and 2 both used this for consistency.

Duty Cycle	Expected Capacitor Voltage	Measured Capacitor Voltage
0	0	0
15	0.271	0.265
30	0.541	0.531
45	0.812	0.795
60	1.082	1.063
75	1.353	1.33
90	1.624	1.596
105	1.894	1.864
120	2.165	2.13
135	2.435	2.394
150	2.706	2.66
165	2.976	2.922
180	3.247	3.186
195	3.518	3.445
210	3.788	3.715
225	4.059	3.975
240	4.329	4.235
255	4.6	4.5

Table 1: AnalogWrite

A second Arduino sketch was written (part 1b) making use of the internal timer control registers to setup Timer1 of the Atmega328p. The sketch code is in the appendix. The frequency was approximately 62kHz instead of 490Hz, making a much smoother capacitor output voltage. Capture 21 shows the highspeed PWM setup from register manipulation.



Capture 21: Highspeed PWM

Duty Cycle	Expected Capacitor Voltage	Measured Capacitor Voltage
0	0	0
32	0.577	0.6
64	1.155	1.18
96	1.732	1.76
128	2.309	2.34
160	2.886	2.93
192	3.464	3.51
224	4.041	4.09
256	4.618	4.63

Table 2: Manual timer manipulation

Using the fast-PWM method, Table 2 was generated by stepping the duty cycle in steps of 32 (12.5%). The capacitor voltage is greatly smoothed by the higher frequency because the resistor limits the rate of change of voltage, and when the frequency is so high, the capacitor doesn't have time to change very much at at all.

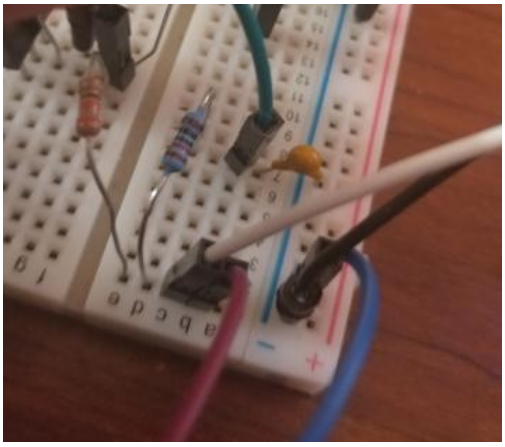


Image 5: RC Circuit

Part 2

In this part of the lab, the PWM waveform is used to sinusoidally change the brightness of an LED. The RC circuit from the previous part of the lab was disconnected and the output pin was connected to an LED through a 330 ohm resistor. This can be seen in image 6.

The code for this part of the lab is in the appendix.

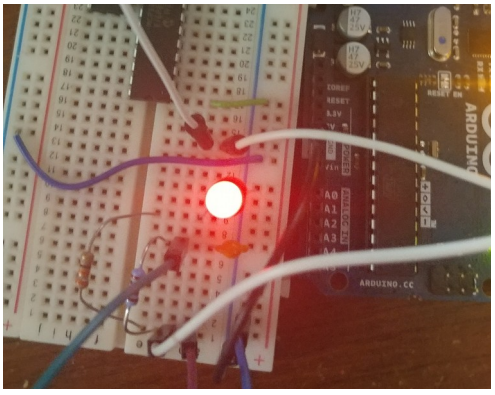


Image 6: LED Circuit in action

Part 3

In this part of the lab, a DC motor is controlled via PWM from the Arduino.

The diode is required because when the motor is turned OFF (the transistor is driven off by the PWM signal), the current in the coils in the motor keeps flowing. Without a diode, large voltages can be built up by the motor as the current attempts to leave the motor. Image 7 shows the motor circuit in action.

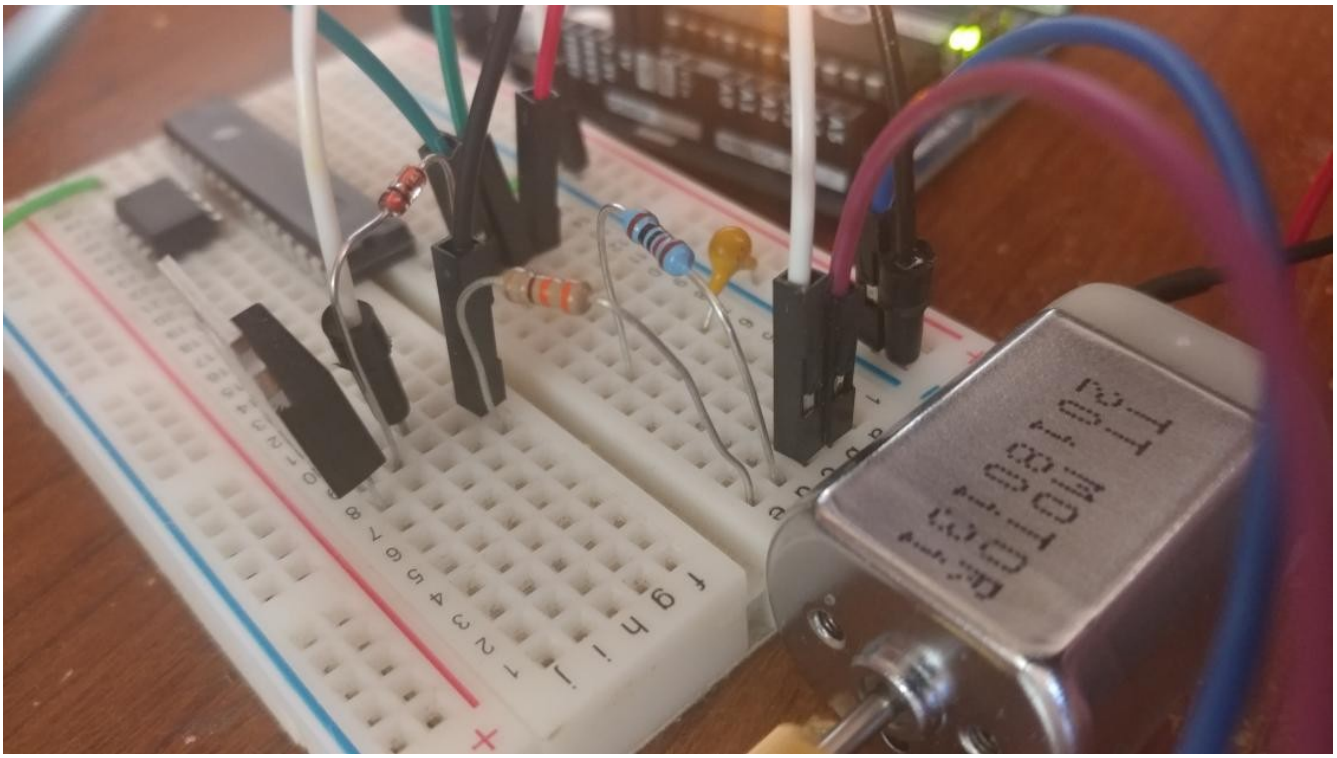


Image 7: Motor circuit

The circuit operates by buffering the PWM output from the Arduino with a Bipolar Junction Transistor. The Transistor is capable of handling much higher currents than the Arduino itself and is necessary for running big loads like a motor. The current entering the Base terminal of the transistor is supplied by the Arduino, and is then multiplied by the BJT to run the motor.

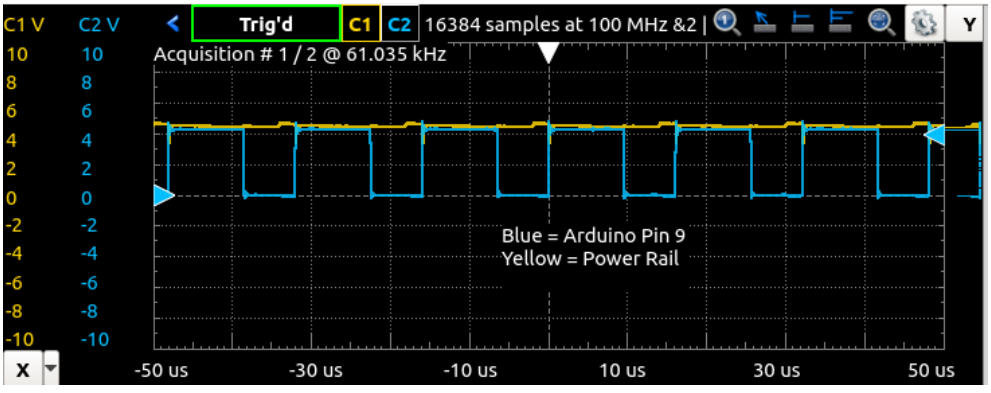


Image 8: DC motor PWM

Because of the high load of the motor, the Arduino's 5v rail had to be verified and care taken to make sure it wasn't dropping too low. In Image 8, the output voltage rail is shown in Yellow and PWM to the BJT is shown in blue.

Conclusion

At first, a small motor was used, and I think it must have been a 3.3v device or similar because it took huge amounts of current and it was very, very fast. It was in the order of almost half an amp at 5v and was worrisome at full speed. The motor pictured in Image 7 is another motor that took about 100mA at full load and was much nicer to work with.

Code Appendix

Part 1a

```
const int motorPin = 9;

int motorSpeed = 127;

void setup() {
  analogWrite(motorPin, motorSpeed);
  delay(1000);
}

void loop() {
}
```

Part 1b

```
/*
 * Author: Zachary Whitlock
 * Date: 03/11/2020
 *
 * Written to control timer1 for EE-333, lab 5, part 1b.
 */
const int motorPin = 9; // OC1A is PB1, which is digital pin 9...
int motorSpeed = 127;

void setup() {
  noInterrupts();
```

```
  pinMode(13, OUTPUT);

  // Set pins 9 and 10 to OUTPUT
  DDRB |= 0b00000110;

  byte R1A_mask = 0x00;
  byte R1B_mask = 0x00;

  // Toggle OC1A, might as well use OC1B
  R1A_mask |= (0b1010 << 4);
  // Enable fast-pwm, 8-bit mode (WGM bits)
  R1A_mask |= 0b01;

  // Enable fast-pwm, 8-bit mode (WGM bits)
  R1B_mask |= (0b01 << 3);
  // Clock I/O, no prescaler
  R1B_mask |= 0b001;

  //R1A_mask = 0b10000001;
  //R1B_mask = 0b00001001;
  TCCR1A = R1A_mask;
  TCCR1B = R1B_mask;

  TIMSK1 = 0b00000000;

  TCNT1 = 0;

  interrupts();

  OCR1A = 127;
  OCR1B = 240;

  delay(1000);
}

void loop() {
  if (TIFR1 != 0x00) {
    TIFR1 = 0x00;
  }
```

```
  // Increase the duty cycle once every 7.5 seconds
  for (int x = 0; x < 255;) {
    OCR1A = x;
    x += 32;
    delay(7500);
  }
  delay(7500);

}
```

Part 2

```
/*
 * Author: Zachary Whitlock
 * Date: 03/11/2020
 *
 * Written to control timer1 for EE-333, lab 5, part 2.
 */

double sin_step = 0.0;

void setup() {
  noInterrupts();
  // Set pins 9 and 10 to OUTPUT
  DDRB |= 0b00000110;

  byte R1A_mask = 0x00;
  byte R1B_mask = 0x00;

  // Toggle OC1A, might as well use OC1B too
  R1A_mask |= (0b1010 << 4);
  // Enable fast-pwm, 8-bit mode (WGM bits)
  R1A_mask |= 0b01;

  // Enable fast-pwm, 8-bit mode (WGM bits)
  R1B_mask |= (0b01 << 3);
  // Clock I/O, no prescaler
  R1B_mask |= 0b001;

  TCCR1A = R1A_mask;
  TCCR1B = R1B_mask;
```

```
  // Disable interrupts
  TIMSK1 = 0b00000000;
  interrupts();
}

// Vary the duty cycle with a sinusoidal periodic function
void loop() {
  // sin() appears to work without the math library.
  OCR1A = (unsigned int)(127.0 * sin(sin_step) + 128.0);
  delay(15); // 15 seems like the sweet spot
  sin_step = sin_step + 0.1;
}
```

Part 3

```
/*
 * Author: Zachary Whitlock
 * Date: 03/11/2020
 *
 * Written to control timer1 for EE-333, lab 5, part 2.
 */

#define MOTOR_MIN 150
#define MOTOR_MAX 255

double sin_step = 0.0;


void setup() {
  noInterrupts();
  // Set pins 9 and 10 to OUTPUT
  DDRB |= 0b00000110;

  byte R1A_mask = 0x00;
  byte R1B_mask = 0x00;

  // Toggle OC1A, might as well use OC1B too
  R1A_mask |= (0b1010 << 4);
  // Enable fast-pwm, 8-bit mode (WGM bits)
  R1A_mask |= 0b01;
```

```
  // Enable fast-pwm, 8-bit mode (WGM bits)
  R1B_mask |= (0b01 << 3);
  // Clock I/O, no prescaler
  R1B_mask |= 0b001;

  TCCR1A = R1A_mask;
  TCCR1B = R1B_mask;

  // Disable interrupts
  TIMSK1 = 0b00000000;
  interrupts();

  OCR1A = 250;
}

// Vary the duty cycle to speed up/down the motor
void loop() {
  for (int x = MOTOR_MIN; x < MOTOR_MAX; x++) {
    OCR1A = x;
    delay(50);
  }
  for (int x = MOTOR_MAX; x > MOTOR_MIN; x--) {
    OCR1A = x;
    delay(50);
  }

}
```

Lab 6

Introduction

The purpose of this lab is to demonstrate the use of the Arduino Uno's internal Analog to Digital Converter (ADC). The ADC will be used to measure an analog voltage from a potentiometer and from a function generator, as well as the temperature from a temperature sensor. Readings are displayed both in the Arduino Serial Monitor and on an LCD display.

Part 1

In this part of the lab, the ADC is utilized to measure a variable voltage supplied by a voltage divider in the form of a potentiometer.

The registers internal to the ATmega328p MCU are manipulated directly to enable, and configure the 10-bit ADC. The ADC is enabled and configured with a prescaler value of 4 from the internal clock speed. The complete circuit is shown in image 4.

Image 9: LCD and Potentiometer

The ADC is 10-bits, which means there are 1024 possible values that it will display. To map this to a voltage reading, we divide the ADC reading by the number of possible state (1024) and multiply by our max voltage (5V). This results in about 4.9mV per LSB (least-significant-bit), or 0.0049V. Due to this, I've decided that the appropriate precision past the decimal point is to show three places, since the lowest step is in millivolts, and the third decimal point represents that.

Additionally, the voltage was stepped in approximately 0.5V increments using the potentiometer. Each step was read with a Fluke multimeter and the ADC. Table 3 compares the readings of the ADC vs the multimeter.

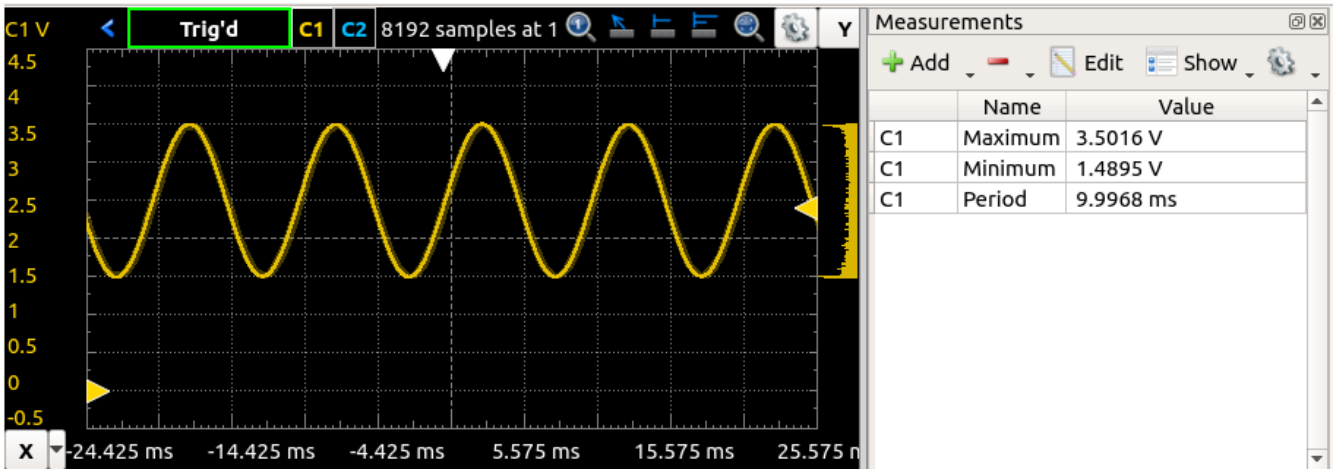
Multimeter	LCD Displayed
0.088	0.137
0.5	0.552
1	1.094
1.518	1.558
2.028	2.109
2.526	2.573
3.052	3.12
3.538	3.628
4.011	4.048
4.507	4.609
4.828	4.922

Table 3: Multimeter Vs ADC

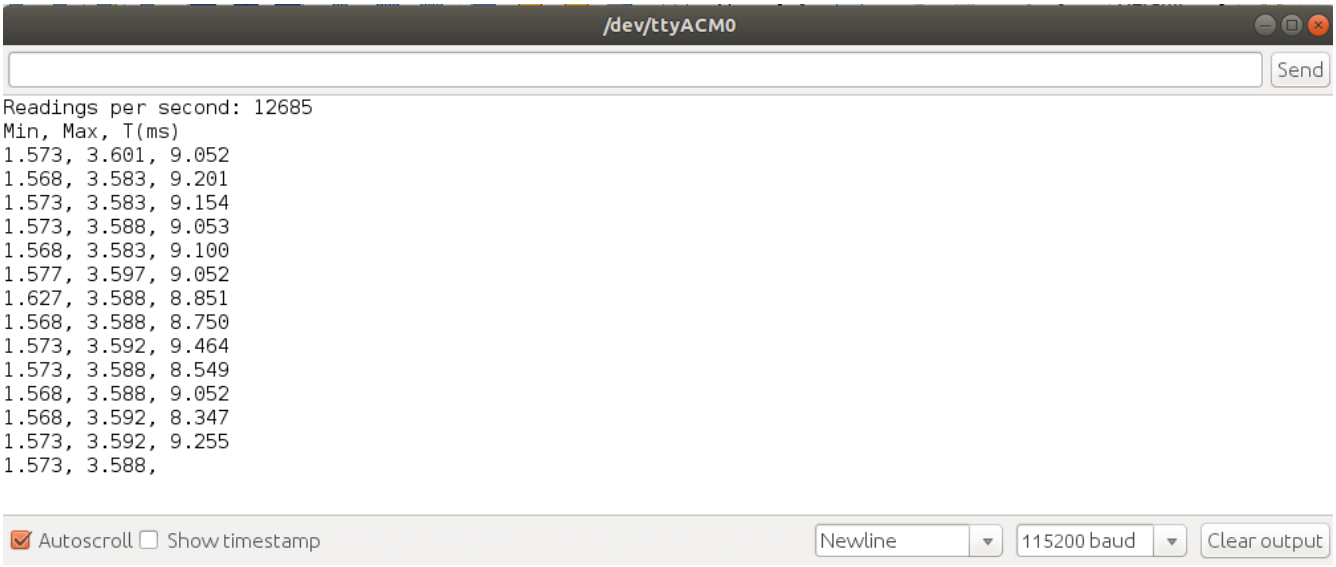
Part 2

In this part of the lab we use the ADC to measure a sinusoidal wave created by a function generator. The Arduino is programmed to measure and report the minimum voltage, the maximum voltage, and the period of the sinewave.

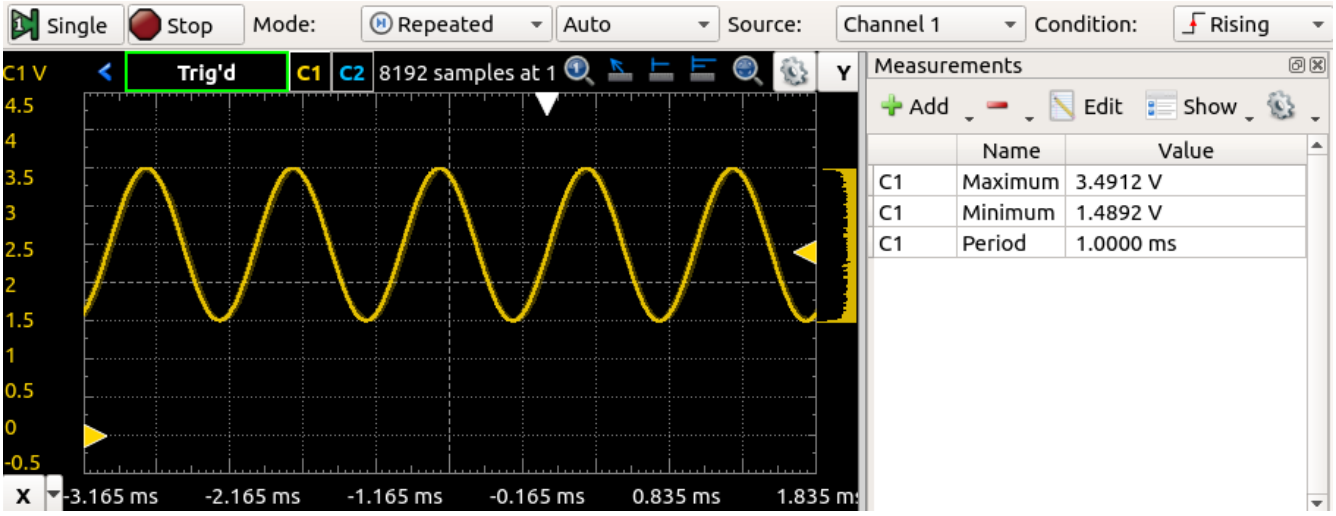
The ADC was setup such that it's prescaler would provide it a frequency of 125kHz, 16Mhz/128. This should result in a rough sampling frequency of 9615 samples per second. Captures 22-27 show the ADC's reading for the minimum, maximum, and period of the sine wave, along with the oscilloscope's reading for comparison. Not evident in Captures 22-27 is the actual sampling frequency of approximately 8987. The actual sampling frequency would have been shown except for time requirements on submitting the lab.



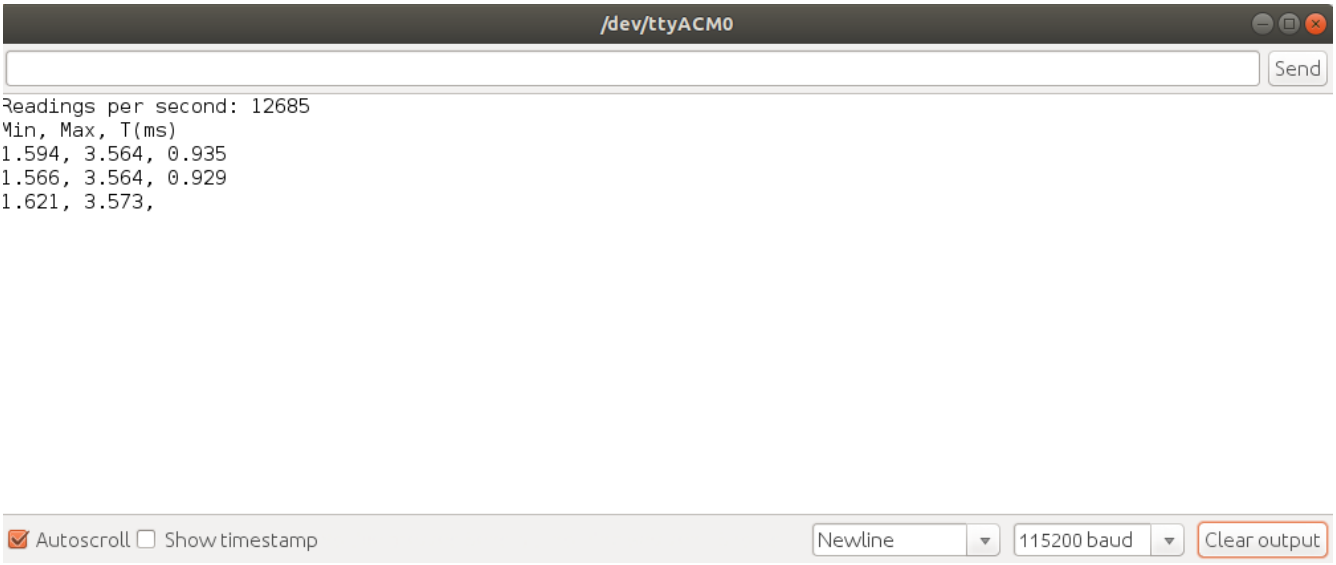
Capture 22: 100Hz - Oscilloscope



Capture 23: 100Hz - Serial Monitor

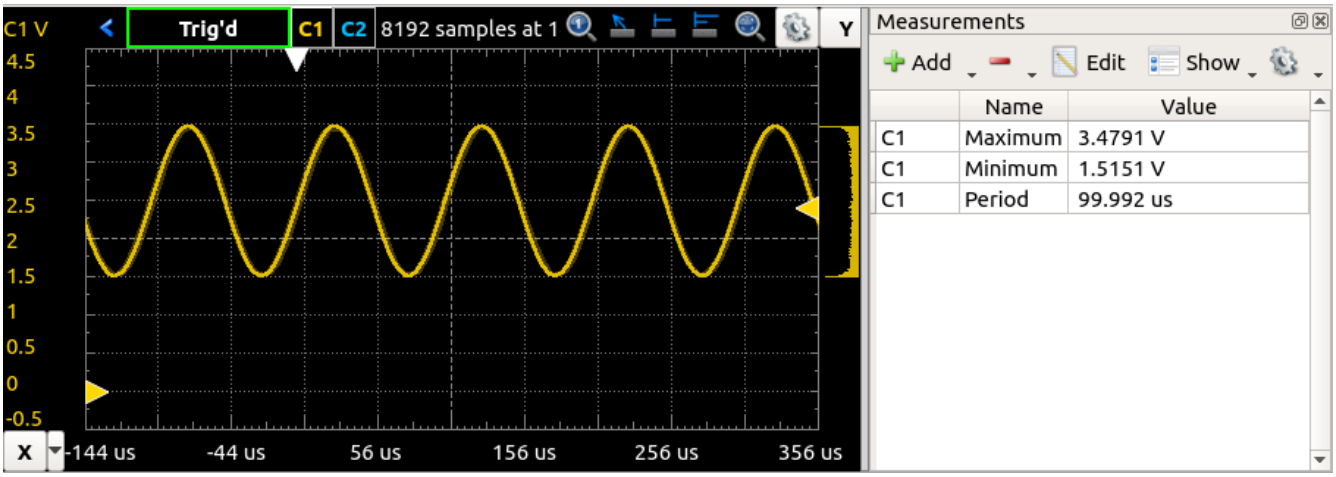


Capture 24: 1kHz - Oscilloscope

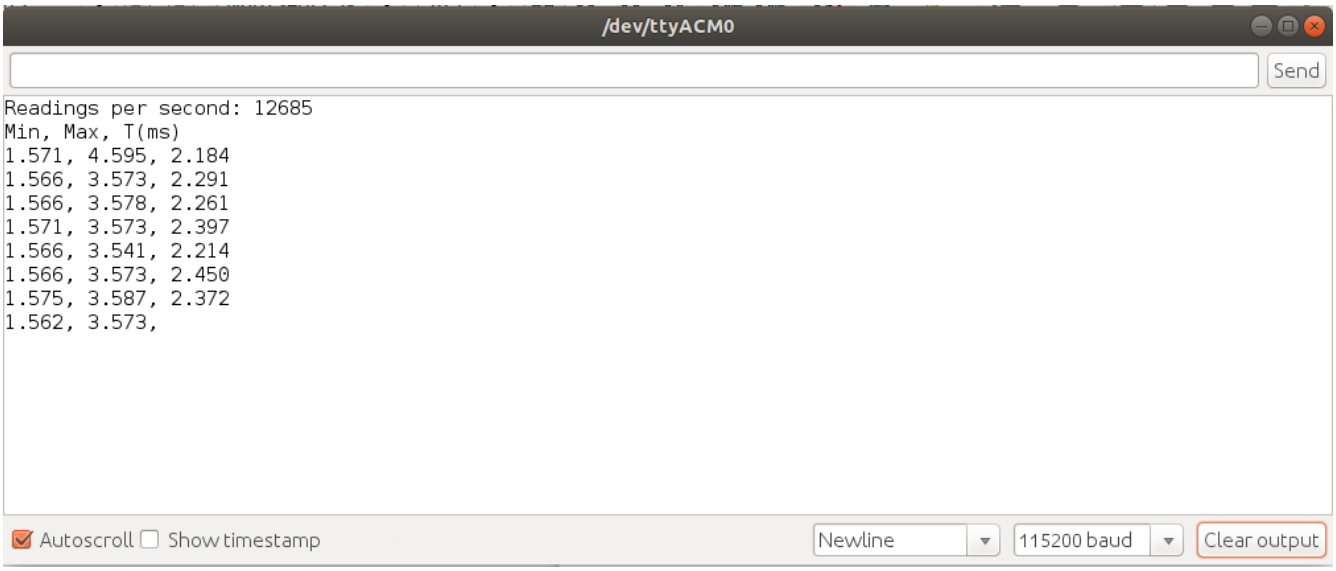


Capture 25: 1kHz - Serial Monitor

In the interest of time and of personal preference, I've set the peak-to-peak value of the sine wave to 2V instead of 5, and it varies between ~1.5v and ~3.5v. As mentioned later in the document, the Arduino's reference voltage was significantly lower than 5V and the lower measurement was more interesting to me than a clipped value.



Capture 26: 10kHz - Oscilloscope



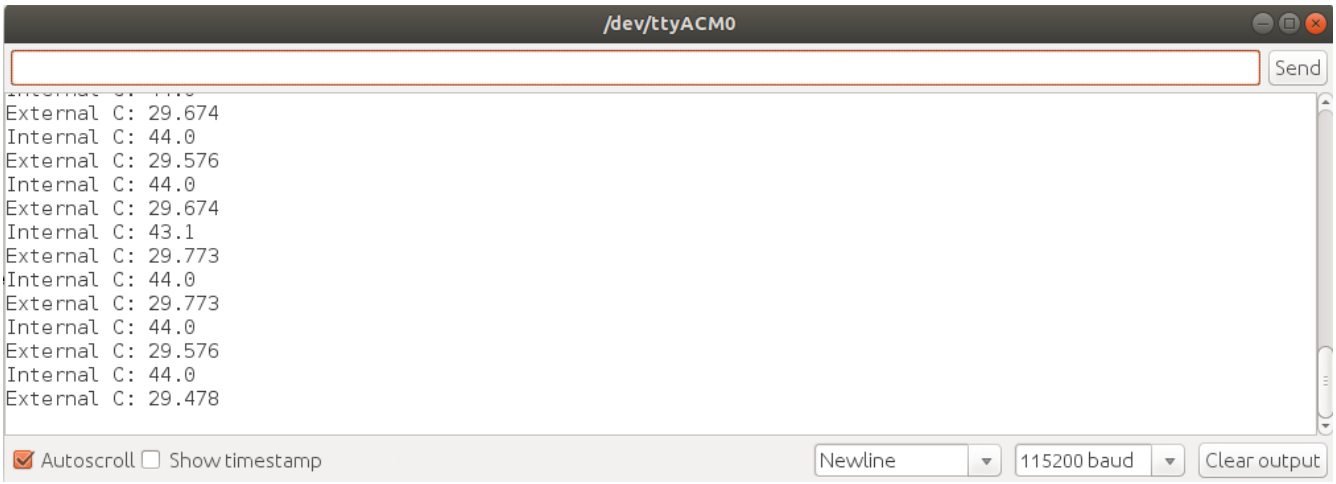
Capture 27: 10kHz - Serial Monitor

It's evident in Captures 22-27 that the sampling frequency of ~8987 is not fast enough to accurately measure the period of signals faster than 1kHz using the code designed by me. If the ADCSRA register is set so that the prescaler is running as fast as possible, the samples per second increases to nearly 200k, which is the stated limit in the datasheet.

The minimum and maximum functions were re-written several times to be as accurate as possible. Because this was done without interrupts, it was especially difficult to produce a robust measurement. Additionally, code was added to (attempt to) calculate the actual Vcc value of the Arduino, since the ADC's reference is roughly equal to Vcc. My Vcc was significantly less than 5v, which made measurements wildly inaccurate. This code requires the measurement of the internal "1.1v" reference, which required the one-time measurement of the actual Vcc. My internal 1.1v reference is closer to ~1.076V.

Part 3 – Temperature Measurement

In this part of the lab, the Arduino is used to measure it's internal temperature as well as an external temperature sensor. The ADC is used like in the other two parts of the lab, but this time it's not used in free-running mode. The ADC's interrupt is enabled, and for better readings the Arduino is put into a low power more to decrease the amount of noise on the ADC. Once the ADC sample is complete, the Arduino wakes and continues with the rest of the program, calculating the temperatures from the measurements and displaying them. Capture 28 shows the result shown on the Serial Monitor.



Capture 28: Serial Monitor

Although the lab calls for a DMM to measure the temperature of the temperature probe, I didn't have one on hand that supported a temperature measurement. Instead, I used a cooking thermometer secured to the TMP36 temperature measuring IC.

Image 10: Temperature verification

Thermometer	**Arduino**
25.5	24.8
31	32.0
35	34.5
40	39.1
45	44.2
50.5	50.3
55	50.5

This table compares the cooking-thermometer measurement to that of the Arduino. The Arduino was fairly accurate until values over ~50C, which was the maximum value that could be read with the 1.1V reference. Should the ADC reference be switched to Vcc, a much higher temperature could be measured, with a penalty to precision.

Image 11 shows the circuit that the lab made use. Additional capacitors were added to the Aref pin of the Arduino and the center pin of the TMP36 to further increase accuracy and make it less necessary to run averages in software.

The internal 1.1V reference was found to be much closer to 1V, and so that's what I used in the calculations for temperature. When the 1.1V reference is selected in the ADC configuration, the Arduino 'Aref' pin outputted it for measuring with the oscilloscope. The value was averaged over about 100ms

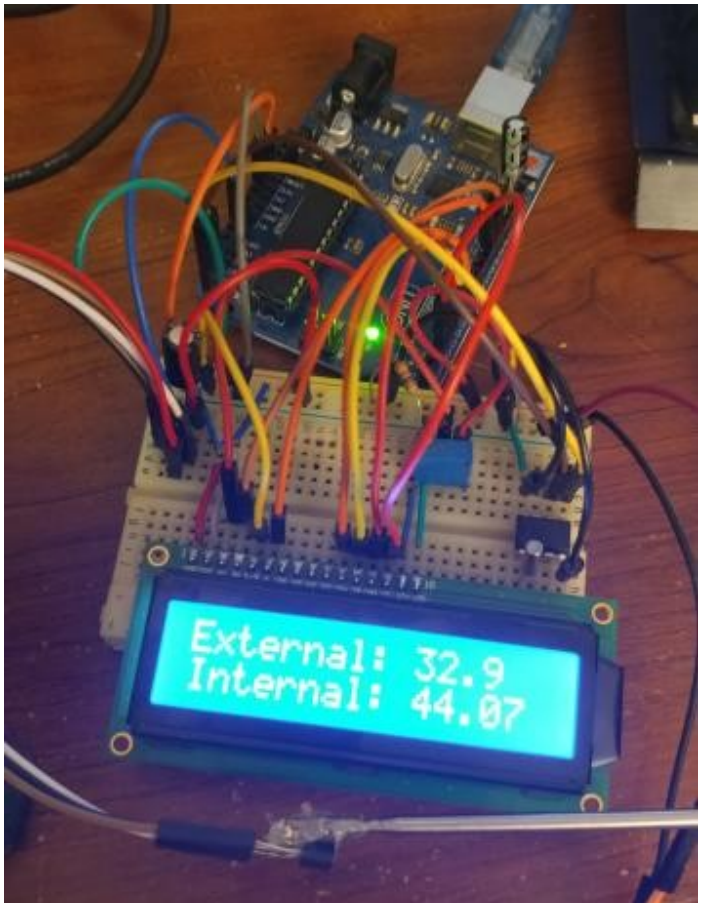


Image 11: Circuit

Conclusion & Discussion

In part 2, the sampling frequency was significantly decreased to just over 1kHz if the code only checked the ADIF bit without resetting to 0. While it made sense that the limited sampling frequency would be inaccurate when measuring a 10kHz signal, there was a length of time spent figuring out why 1kHz signals were inaccurate.

In part 3, much time was spent on trying to read the TS_OFFSET and TS_GAIN values mentioned in the Atmel datasheet. A unique value was determined for the TS_OFFSET to be 149, but strange issues were encountered when attempting to gleam the TS_GAIN value. If the value was used in any way except as an unsigned integer, it seemed to crash the microcontroller in the setup function. I did learn a little bit about the flash programming procedures and AVR assembly in the process, and got a glimpse at what it would take to make a boot-loader. For brevity, the non-working code is not included in the appendix for part 3.

In conclusion, this lab was great practice for using the ATmega328p's internal, 10-bit, ADC. The limits of precision were fairly well explored and multiple measurement ranges were explored. In the future, using the ADC in the Arduino or other devices using the ATMega328p will be much easier, and now I have some reference code for the future.

Appendix

Code – Part 1

```
/*
 * Author: Zachary Whitlock
 * Program: lab6-part1
 * Date: 10/11/20
 * Description:
 *     Reads a voltage using the built-in ADC of an ATMega328p and displays
 *     the result on the serial monitor and an LCD.
 */

#include <Wire.h>
#include <LiquidCrystal.h>

const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
void setup() {
```

```
  Serial.begin(115200);

  //      ADMUX
  // REF = VCC
  // MUX, ADC0, 0b0000
  ADMUX = 0b01000000;

  //      ADCSRA
  // ADEN - enable, 1
  // ADPS - prescale, 4, 0b010
  ADCSRA = 0b11100010;

  //      ADCSRB
  // ADTS - free running, 0b000
  ADCSRB = 0b00000000;

  // Read ADCL and then ADCH (0x78, 0x79)
}

void loop() {
  int ADCReading;
  float voltage;

  ADCReading = ADCL;
  ADCReading |= ADCH << 8;

  voltage = (ADCReading / 1024.0) * 5.0;

  lcd.setCursor(0, 1);
  lcd.print("Voltage: ");
  lcd.print(voltage, 3);
  Serial.println(voltage, 3);
  delay(100);
}
```

Code – Part 2

```
/*
   Author: Zachary Whitlock
   Program: lab6-part2
   Date: 10/11/20
```

```
   Description:
       Reads a min/max voltage and period using the built-in ADC of an ATMega328p and
displays
       the result on the serial monitor and an LCD.
*/

#include <Wire.h>
#include <LiquidCrystal.h>

#define REQUIRED_COUNT 3
#define PERIOD_HYST 200

const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

// Measured internal "1.1V" reference (in millivolts)
const int internalRef = 1076;

int readVcc;

void setup() {
  long calculatedOffset;
  Serial.begin(115200);
  lcd.begin(16, 2);

  //      ADMUX
  // REF = VCC
  // MUX, 1.1Vref, 0b1110
  ADMUX = 0b01001110;

  //      ADCSRA
  // ADEN - enable, 1
  // ADSC - start, 1
  // ADPS - prescale (125kHz) 128, 0b111
  ADCSRA = 0b11100111;

  //      ADCSRB
  // ADTS - free running, 0b000
  ADCSRB = 0b00000000;
```

```
  // Idea from
  // https://hackingmajenkoblog.wordpress.com/2016/02/01/making-accurate-adc-readings-on-the-arduino/
  // Calculate the real VCC
  delay(20);
  calculatedOffset  = ADCL;
  calculatedOffset |= ADCH << 8;
  readVcc = internalRef * 1024L / calculatedOffset;



  // MUX, ADC1, 0b0001
  ADMUX  = 0b01000001;
  ADCSRA = 0b11100111;

  lcd.setCursor(0, 1);
  lcd.print("Vcc: ");
  lcd.print(readVcc / 1000.0, 3);

  Serial.print("Readings per second: ");
  Serial.println(readingsPerSecond());

  Serial.println("Min, Max, T(ms)");


}

void loop() {
  float maxVoltage;
  float minVoltage;


  maxVoltage = (getMax() / 1024.0) * readVcc / 1000.0;
  minVoltage = (getMin() / 1024.0) * readVcc / 1000.0;

  lcd.setCursor(0, 0);
  lcd.print("Max: ");
  lcd.print(maxVoltage, 3);
  lcd.setCursor(0, 1);
  lcd.print("Min: ");
```

```
  lcd.print(minVoltage, 3);


  Serial.print(minVoltage, 3);
  Serial.print(", ");
  Serial.print(maxVoltage, 3);
  Serial.print(", ");
  Serial.println(determinePeriod() / 1000.0, 3);



  delay(1000);

  // randomly clear LCD
  if (micros() % 100 == 0)
    lcd.clear();
}

// Returns min voltage in millivolts
int getMin() {
  int count = 0;
  int minValue = makeReading();
  int oldReading = 0;
  int oldMin = 0;
  int newReading;

  // Wait for falling edge
  newReading = makeReading();
  while (makeReading() >= newReading);

  // Sample until we stop getting values any lower
  do {
    newReading = makeReading();

    // Discard duplicate readings
    if (newReading == oldReading)
      continue;


    // If our min value is lower than our current value,
```

```
    // update the min value.
    if (minValue > newReading) {
      minValue = newReading;
    }

    // If our old min value is the same as the current min value
    // and we aren't going up
    // increment counter by 1.
    if (minValue == oldMin && !(newReading >= oldReading)) {
      count++;
    }

    oldMin = minValue;
    oldReading = newReading;

  } while (count < REQUIRED_COUNT);


  return minValue;
}

// Returns max voltage in millivolts
int getMax() {
  int count = 0;
  int maxValue = makeReading();
  int oldReading = 0;
  int oldMax = 0;
  int newReading;

  // Wait for rising edge
  newReading = makeReading();
  while (makeReading() <= newReading);

  // Sample until we stop getting values any higher
  do {
    newReading = makeReading();

    // Discard duplicate readings
    if (newReading == oldReading)
      continue;
```

```
    /*
      Serial.print(newReading);
      Serial.print(", ");
      Serial.println(maxValue);
    */

    // If our max value is lower than our current value,
    // update the max value.
    if (maxValue < newReading) {
      maxValue = newReading;
    }

    // If our old max value is the same as the current max value
    // and we aren't going down
    // increment counter by 1.
    if (maxValue == oldMax && !(newReading <= oldReading)) {
      count++;
    }

    oldMax = maxValue;
    oldReading = newReading;

  } while (count < REQUIRED_COUNT);


  return maxValue;
}

// Return period in microseconds
long determinePeriod() {
  long timeNow;
  long elapsedTime = 0;
  int count = 0;
  int oldReading;
  int newReading;;

  // Wait for a reading at a middle value
  int middle = (getMin() + getMax()) / 2;
  do {
```

```
    newReading = makeReading();
  } while (!(newReading < (middle + PERIOD_HYST)) || !(newReading > (middle -
PERIOD_HYST)));

  oldReading = makeReading();

  // Wait for identical reading (2x)
  timeNow = micros();
  do {
    newReading  = makeReading();
    //Serial.print(oldReading);
    //Serial.print(", ");
    //Serial.println(newReading);

    if (newReading < (oldReading + PERIOD_HYST) && newReading > (oldReading -
PERIOD_HYST)) {
      elapsedTime += (micros() - timeNow);
      timeNow = micros();
      count++;
      // wait for a new reading
      while (makeReading() < (oldReading + PERIOD_HYST) && makeReading() > (oldReading -
PERIOD_HYST));
    }

  } while (count < 100);

  // Report elapsed time
  return (elapsedTime / (count - 1));
}

// Take a reading from the ADC
int makeReading() {
  int ADCReading;
  while (!bit_is_set(ADCSRA, ADIF));
  ADCReading = ADCL;
  ADCReading |= ADCH << 8;
  ADCSRA |= 1 << ADIF;
  return ADCReading;
}
```

```
long readingsPerSecond() {
  int timeNow = micros();
  long i = 0;
  while ((timeNow + 1000000) > micros()) {
    makeReading();
    i++;
  }

  return i;
}
```

Code – Part 3

```
/*
    Author: Zachary Whitlock
    Program: lab6-part3
    Date: 10/11/20
    Description:
        Measure an internal MCU temperature and an external temperature sensor
        using the ADC.
*/

#include <LiquidCrystal.h>

const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

// Measured internal "1.1V" reference (in millivolts)
const int internalRef = 1006;

void setup() {
  Serial.begin(115200);
  pinMode(13, OUTPUT);
  lcd.begin(16, 2);

  //      ADMUX
  // REF = 1.1V
  // MUX, 1.1V, 0b1110
  ADMUX = 0b11001110;

  //      ADCSRA
```

```
  // ADEN - enable, 1
  // ADSC - start, 1
  // ADPS - prescale 2, 0b000
  ADCSRA = 0b11000000;

  //      ADCSRB
  // ADTS - free running, 0b000
  ADCSRB = 0b00000000;
}

void loop() {
  interrupts();
  float internalTemp;
  float externalTemp;

  // Measure internal temperature
  // MUX, ADC8 (temp.), 0b1000
  ADCSRA = 0b10001111;
  ADMUX  = 0b11001000;
  internalTemp = lowNoiseRead();
  internalTemp = (internalTemp / 1024.0) * (internalRef / 1000.0);
  internalTemp = (internalTemp - 0.314)/0.001 + 25;
  Serial.print("Internal C: ");
  Serial.println(internalTemp, 1);

  delay(10);

  // Measure TMP36 (IC) temperature
  // MUX, ADC0, 0b0000
  ADCSRA = 0b10001111;
  ADMUX  = 0b11000000;
  externalTemp = lowNoiseRead();
  externalTemp = (externalTemp / 1024.0) * (internalRef / 1000.0);
  externalTemp = (externalTemp - 0.5) / 0.01;
  Serial.print("External C: ");
  Serial.println(externalTemp, 3);



  lcd.setCursor(0, 0);
```

```
  lcd.print("External: ");
  lcd.print(externalTemp, 1);
  lcd.setCursor(0, 1);
  lcd.print("Internal: ");
  lcd.print(internalTemp, 1);

  delay(500);
}



ISR(ADC_vect) {
  // Disable sleep
  SMCR = 0b00000010;
};

int lowNoiseRead() {
  int ADCReading;
  // Goto sleep for low-noise read
  SMCR = 0b00000011; // enable sleep
  asm("SLEEP");
  // Will awake once the conversion is done.

  ADCReading = ADCL;
  ADCReading |= ADCH << 8;
  return ADCReading;
}
```

Lab 7

Introduction

In this lab, Infrared (IR) light will be used to do wireless communication between two Arduinos. Ken Shirrif's IR library (IRremote) is used for the IR communication, and the hardware consists of 2x Arduino Unos, an IR reciever, and an IR LED.

Part 1

In this part of the lab, the first Arduino is connected to the IR receiver and the IRremote example code called "IRreceiveDumpV2.ino". The signal pin of the IR reciever module is attached to digital pin 11 of the Arduino. Additionally, an oscilloscope is hooked up to the output of the IR module as well.

Code Explanation

IRrecv: The code starts by initializing a new C++ class structure called 'IRrecv', which provides easy access to the functions available for getting and decoding data from the IR receiver.

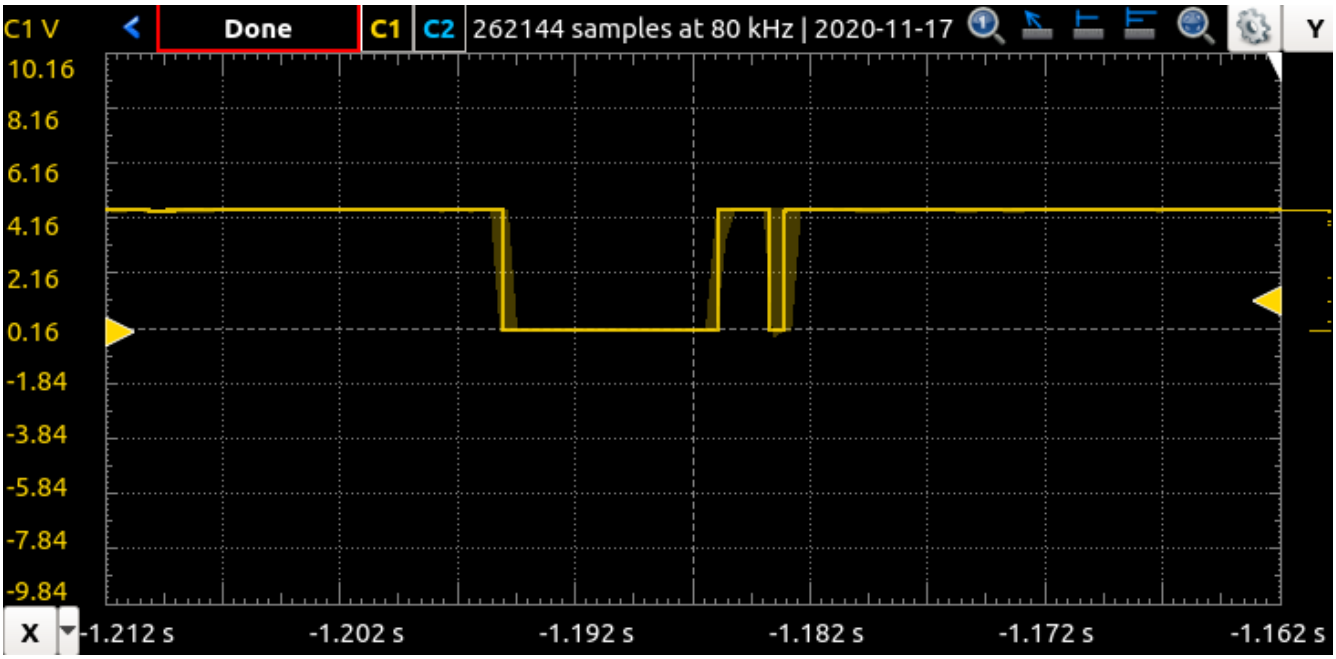
Setup: The program enables the built-in LED of the Arduino, starts the serial bus at 115200, prints out some text about the program, starts receiving with "IrReceiver.enableIRIn()", ties the built-in LED to the IRecv with "IrReceiver.blink13(true);", and prints out more text about the program on the serial monitor.

Loop: The loop section, from a bird's eye view, checks if there is data ready to decode and prints out the results if so. An 'if' statement checks 'IrReceiver.decode()' before attempting to print out the data. 'IrReceiver.decode()' will check the state machine of the IRecv class and make sure the state is stopped (IR_REC_STATE_STOP). If the state machine hasn't reached the end point yet, 'IrReceiver.decode()' will return 'false'.

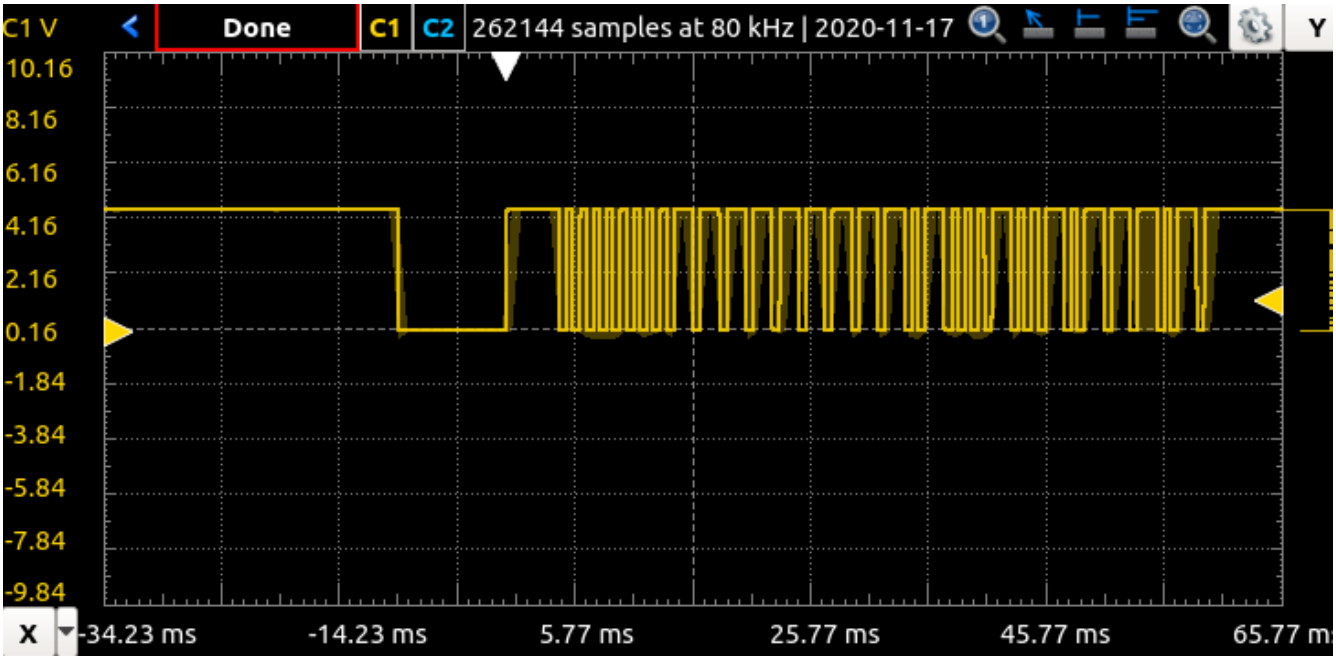
The code uses 'IrReceiver.results.overflow' to see if the received data was too large for the buffer, and advises the user to increase the buffer size if necessary. If actually ready to decode, the data is printed out using class methods of the 'IRrecv' class such as 'printResultShort'. These print functions are internally complicated and difficult to explain, but they essentially they will print out the captured data in several different ways.

At the end of the loop, the class's method 'IrReceiver.resume()' is called. This function literally just restarts the state machine and places it in idle mode (IR_REC_STATE_IDLE).

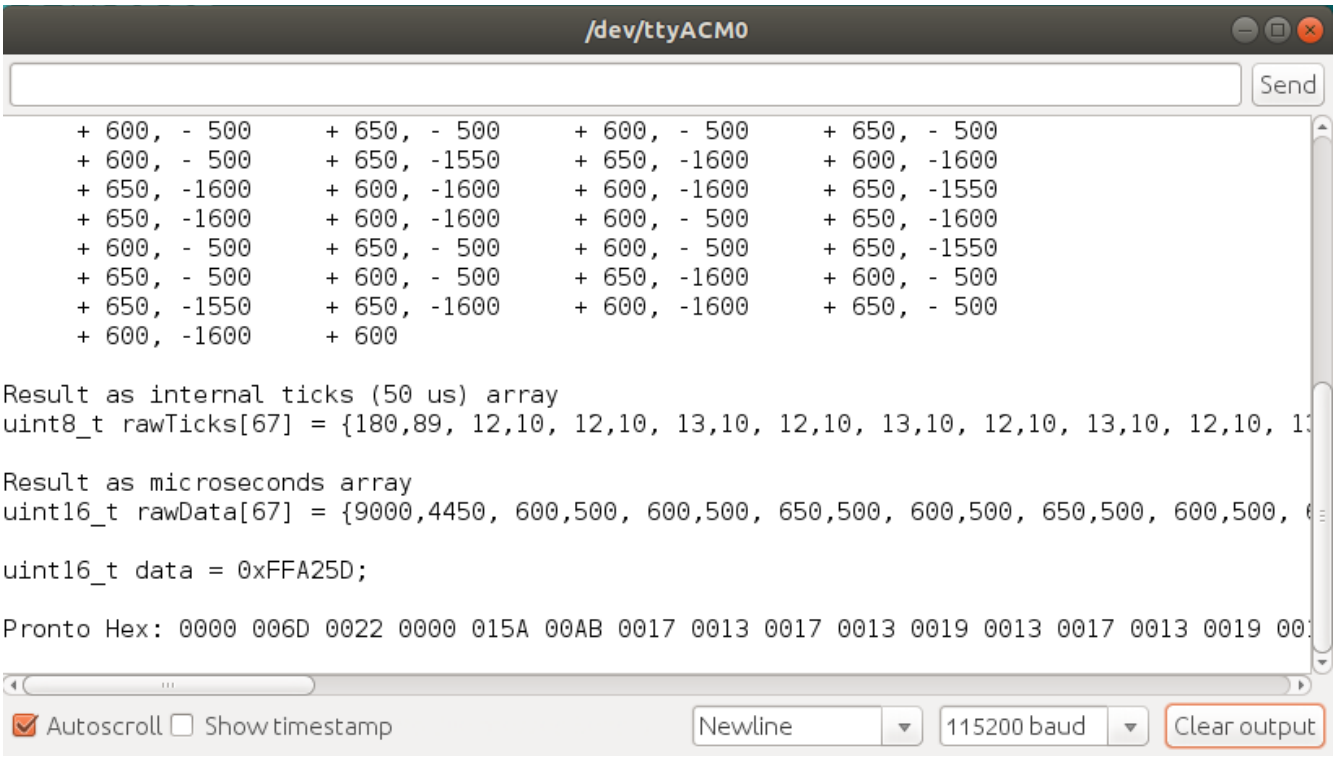
Captures



Capture 29: Repeat Code



Capture 30: Power button code



Capture 31: Serial monitor capture of power button code

Decoding

The oscilloscope waveform shown in Capture 30 is analyzed to find the code that the remote is sending. The Arduino program claims the protocol is NEC, which will be used to decode the waveform.

In NEC, a logical “Zero” is a 562.5uS pulse followed by a 562.5uS space. A logical “One” is the same 562.5uS pulse but it’s followed by a 1.6875mS space. The protocol starts off with a 9mS leading pulse and a 4.5ms space. It should be noted that the oscilloscope shows that the data is inverted, the ACTIVE state is 0V, not 5v. Hence a pulse is when the oscilloscope shows 0v, not 5.

The address of the receiving device is sent first, with the standard logical inverse sent directly following. In this case, there are eight pulses with 562.5uS spaces following them, which means there are 8x ‘8’s. Directly following this pattern are eight pulses with ~1.63mS spaces after them, meaning 8x ‘1’s. This means the address of the receiving device is supposed to be 0x00.

The command sent is 0b1010001001011101 according to the NEC protocol. Illustration 1 shows how the second half of the pattern is the logical inverse of the first half. This is represented by 0xA25D in hex, which is exactly what the program says is the command of the reciever.

0b1010 0010
0b0101 1101

Illustration 1: Power Command

Part 2

In this part of the lab, a second Arduino is used to send a optical signal to the receiving Arduino. An IR LED is attached to the second microcontroller through a 100 ohm resistor, and the “IRSendDemo” code is uploaded.

Code Explanation

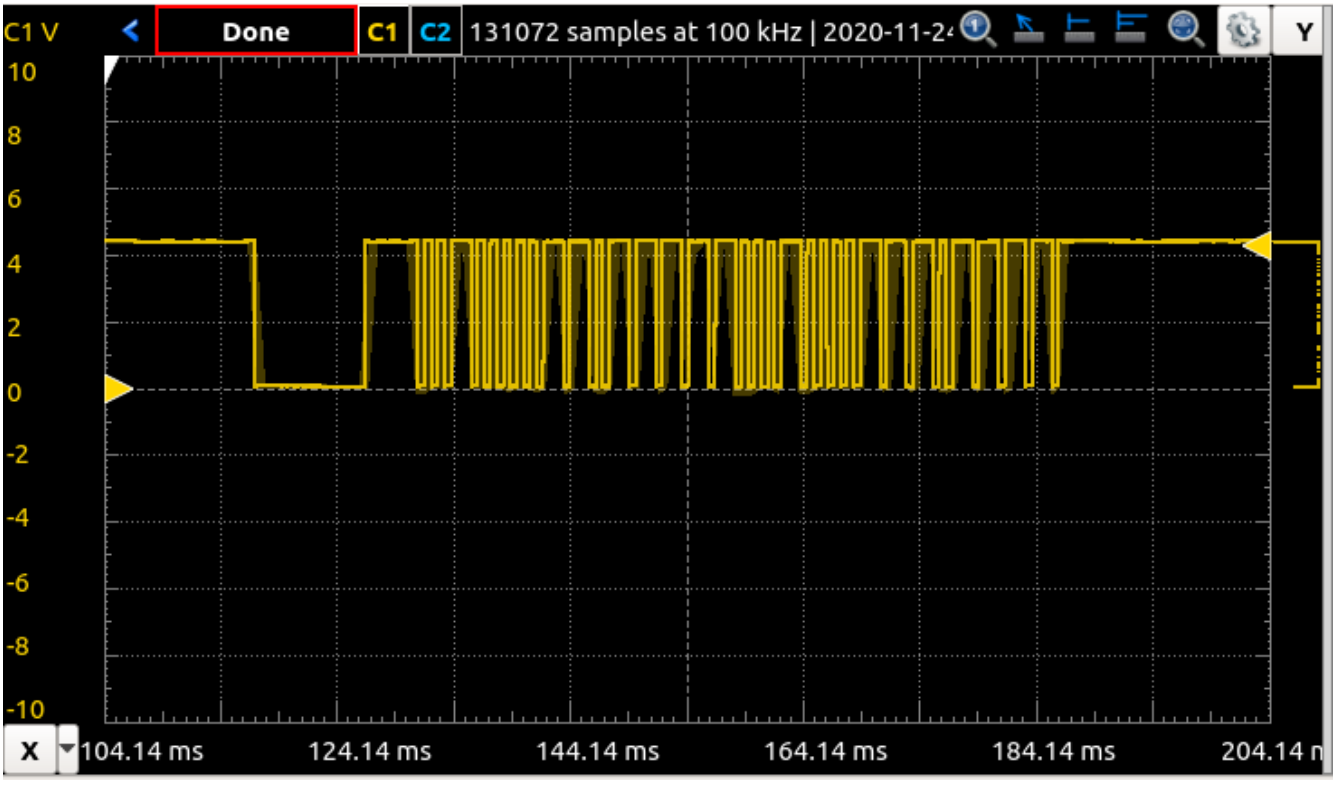
Setup: The program starts by initializing the serial connection to the PC, and enables the built-in LED.

Loop: In the program loop, several different data sets are sent every two seconds, with a delay of 3 seconds at the end of the loop. First, raw data is send via the **sendRaw()** function, which uses an array of 16 bit integers to define the amount of microseconds between pules. The first set of data sent is 0xFB04.

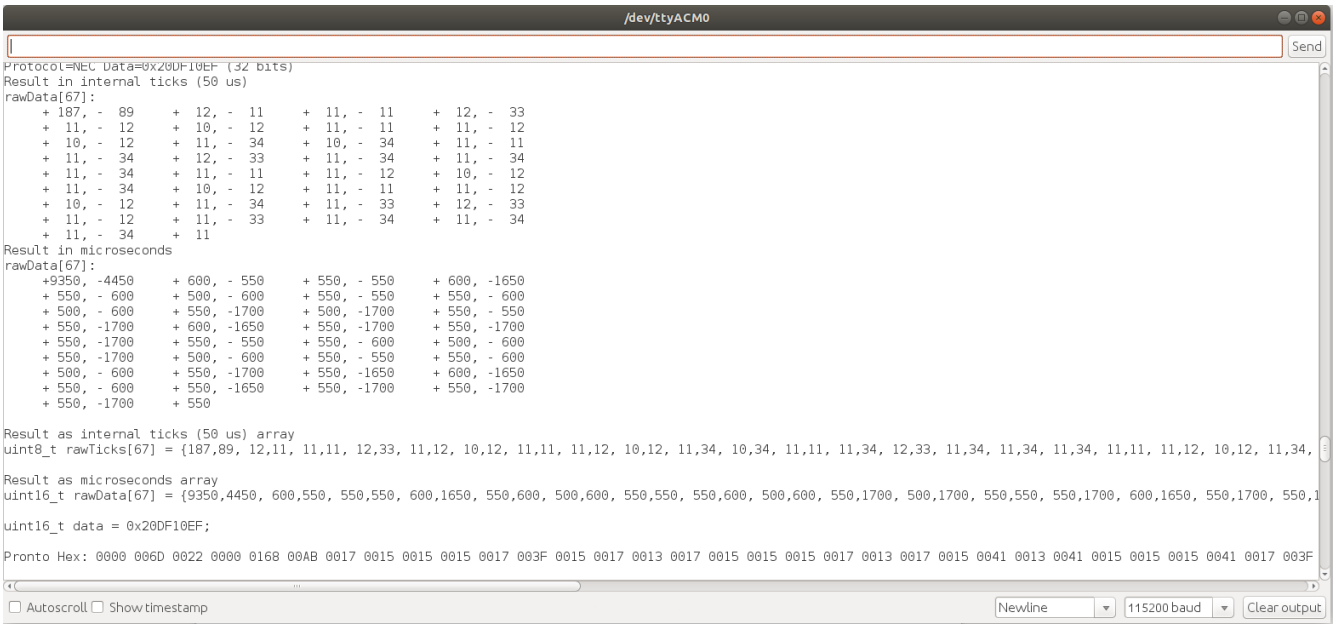
The second piece of data is 0xFB0C and is sent using the **sendRaw_P()** function. This send function takes in ticks instead of microseconds to decide when to send a pulse.

The third and final piece of data sent is 0xFF00, and the **sendNECStandard()** function is used. The standard send function simply accepts the actual hexadecimal data to be sent.

Captures



Capture 32: Oscilloscope Capture of Recieved Data



Capture 33: Serial Monitor of Recieved Data

Image 12 shows the actual bread-boarded circuits.

Decoding

Following the same decoding method as in part 1, the decoded bits from the oscilloscope waveform are: 0b00100000110111110001000011101111, or 0x20DF10EF. This is the first set of data sent in the loop, the 'NEC 0xFB04', and it matches the **time signatures written in the sketch.**

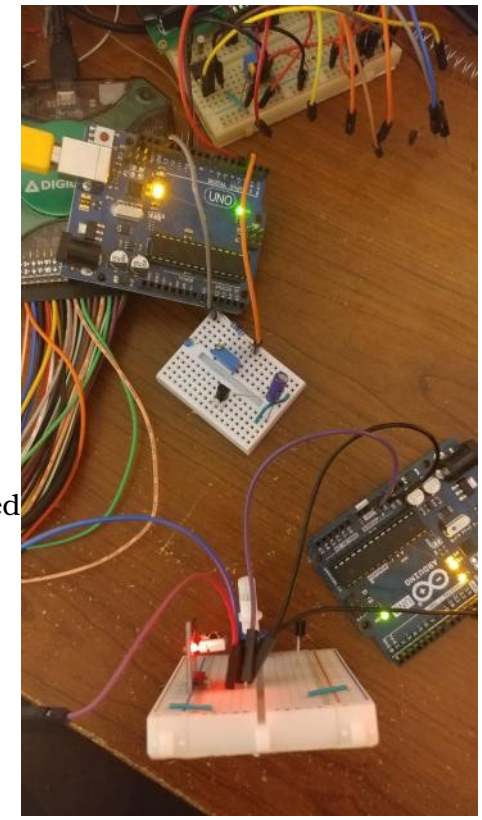


Image 12: IR LED and reciever

Conclusion

It was confusing to me the way the code given for part 2 listed commands like "FB04" and FB0C". The bytes seemed to match what I would expect out of NEC, but the rest didn't. I don't know if this is an error on my part, or the author of the code.

In this lab, I learned about the NEC IR protocol and how it's used to transmit data using Infrared Light. I also learned about the library I was using ('IRremote') which will be useful in future projects.

Appendix

Code – Part 1

```
//------------------------------------------------------------------------
// Include the IRremote library header
//
#include <IRremote.h>

//------------------------------------------------------------------------
// Tell IRremote which Arduino pin is connected to the IR Receiver (TSOP4838)
//
#if defined(ESP32)
int IR_RECEIVE_PIN = 15;
#else
int IR_RECEIVE_PIN = 11;
#endif
IRrecv IrReceiver(IR_RECEIVE_PIN);

//
+=============================================================================
===
// Configure the Arduino
//
void setup() {
    pinMode(LED_BUILTIN, OUTPUT);

    Serial.begin(115200);   // Status message will be sent to PC at 9600 baud
#if defined(__AVR_ATmega32U4__) || defined(SERIAL_USB) ||
defined(SERIAL_PORT_USBVIRTUAL)
    delay(2000); // To be able to connect Serial monitor after reset and before first printout
#endif
    // Just to know which program is running on my Arduino
    Serial.println(F("START " __FILE__ " from " __DATE__));

    IrReceiver.enableIRIn();  // Start the receiver
    IrReceiver.blink13(true); // Enable feedback LED

    Serial.print(F("Ready to receive IR signals at pin "));
    Serial.println(IR_RECEIVE_PIN);
```

```
}

//
+=============================================================================
===
// The repeating section of the code
//
void loop() {
    if (IrReceiver.decode()) {  // Grab an IR code
        // Check if the buffer overflowed
        if (IrReceiver.results.overflow) {
            Serial.println("IR code too long. Edit IRremoteInt.h and increase
RAW_BUFFER_LENGTH");
            return;
        }
        Serial.println();                       // 2 blank lines between entries
        Serial.println();
        IrReceiver.printResultShort(&Serial);

        Serial.println(F("Result in internal ticks (50 us)"));
        IrReceiver.printIRResultRawFormatted(&Serial, false); // Output the results in RAW
format
        Serial.println(F("Result in microseconds"));
        IrReceiver.printIRResultRawFormatted(&Serial, true);  // Output the results in RAW
format
        Serial.println();                       // blank line between entries
        Serial.println(F("Result as internal ticks (50 us) array"));
        IrReceiver.printIRResultAsCArray(&Serial, false);   // Output the results as uint8_t
source code array of ticks
        Serial.println(F("Result as microseconds array"));
        IrReceiver.printIRResultAsCArray(&Serial, true);    // Output the results as uint16_t
source code array of micros
        IrReceiver.printIRResultAsCVariables(&Serial);  // Output address and data as source
code variables
        IrReceiver.printIRResultAsPronto(&Serial);

        IrReceiver.resume();                    // Prepare for the next value
    }
}
```

Code – Part 2

```
/*
 * IRremote: IRsendRawDemo - demonstrates sending IR codes with sendRaw
 * An IR LED must be connected to Arduino PWM pin 3.
 * Initially coded 2009 Ken Shirriff http://www.righto.com
 *
 * IRsendRawDemo - added by AnalysIR (via www.AnalysIR.com), 24 August 2015
 *
 * This example shows how to send a RAW signal using the IRremote library.
 * The example signal is actually a 32 bit NEC signal.
 * Remote Control button: LGTV Power On/Off.
 * Hex Value: 0x20DF10EF, 32 bits
 *
 * It is more efficient to use the sendNEC function to send NEC signals.
 * Use of sendRaw here, serves only as an example of using the function.
 *
 */

#include <IRremote.h>

IRsend IrSender;

// On the Zero and others we switch explicitly to SerialUSB
#if defined(ARDUINO_ARCH_SAMD)
#define Serial SerialUSB
#endif

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);

  Serial.begin(115200);
#if defined(__AVR_ATmega32U4__) || defined(SERIAL_USB) ||
defined(SERIAL_PORT_USBVIRTUAL)
  delay(2000); // To be able to connect Serial monitor after reset and before first printout
#endif
  // Just to know which program is running on my Arduino
  Serial.println(F("START " __FILE__ " from " __DATE__));
  Serial.print(F("Ready to send IR signals at pin "));
  Serial.println(IR_SEND_PIN);
```

```
}

/*
 * NEC address=0xFB0C, command=0x18
 *
 * This is data in byte format.
 * The uint8_t/byte elements contain the number of ticks in 50 us
 * The integer format contains the (number of ticks * 50) if generated by IRremote,
 * so the integer format has the same resolution but requires double space.
 */
const uint8_t irSignalP[] PROGMEM
= { 180, 90 /*Start bit*/, 11, 11, 11, 11, 11, 34, 11, 34/*0011 0xC of address LSB first*/,
11, 11, 11, 11, 11, 11, 11, 11/*0000*/,
       11, 34, 11, 34, 11, 11, 11, 34/*1101 0xB*/, 11, 34, 11, 34, 11, 34, 11, 34/*1111*/,
11, 11, 11, 11, 11, 11, 11,
       34/*0001 0x08 of command LSB first*/, 11, 34, 11, 11, 11, 11, 11, 11/*1000 0x01*/,
11, 34, 11, 34, 11, 34, 11,
       11/*1110 Inverted 8 of command*/, 11, 11, 11, 34, 11, 34, 11, 34/*0111 inverted 1 of
command*/, 11 /*stop bit*/};



void loop() {
   int khz = 38; // 38kHz carrier frequency for the NEC protocol
   /*
    * Send hand crafted data from RAM
    * The values are NOT multiple of 50, but are taken from the NEC timing definitions
    */
   Serial.println(F("Send NEC 0xFB04, 0x08 with exact timing (integer format)"));
   const uint16_t irSignal[] = { 9000, 4500, 560, 560, 560, 560, 560, 1690, 560, 560, 560,
560, 560, 560, 560, 560, 560, 560,
          560, 1690, 560, 1690, 560, 560, 560, 1690, 560, 1690, 560, 1690, 560, 1690, 560,
1690, 560, 560, 560, 560, 560, 560,
          560, 1690, 560, 560, 560, 560, 560, 560, 560, 560, 560, 1690, 560, 1690, 560,
1690, 560, 560, 560, 1690, 560, 1690, 560,
          1690, 560, 1690, 560 }; // Using exact NEC timing
   IrSender.sendRaw(irSignal, sizeof(irSignal) / sizeof(irSignal[0]), khz); // Note the approach
used to automatically calculate the size of the array.

   // 0010 0000 1101 1111 0001 0000 1110 1111 (0x20DF10EF)
```

```
    delay(2000);

    /*
     * Send byte data direct from FLASH
     * Note the approach used to automatically calculate the size of the array.
     */
    Serial.println(F("Send NEC 0xFB0C, 0x18 with tick resolution timing (byte format) "));
    IrSender.sendRaw_P(irSignalP, sizeof(irSignalP) / sizeof(irSignalP[0]), khz);

    // 0011 0000 1101 1111 0001 1000 1110 0111 (0x30DF18E7)


    delay(2000);

    Serial.println(F("Send NEC 0xFF00, 0x17 generated"));
    IrSender.sendNECStandard(0xFF00, 0x17, 0);

    delay(3000);
}
```

Lab 8

Introduction

The objective of this lab is to interface the Arduino with a WiFi module. The specific module in this case is the popular ESP8266 device, which will be used as a serial-to-WiFi adapter for creating a web server.

Part 1

Setup

The ESP8266 itself is a 3.3V device that does not work at all with 5V. The Arduino, an Atmega328p, operates on 5V. To allow communication between the Arduino and the ESP8266, additional circuitry is required to convert the logic levels. I didn't have a logic level converter so a simple one was created. I referenced the schematic of Sparkfun's MOSFET logic-level converter and created a BJT version for 2N3904 BJTs.

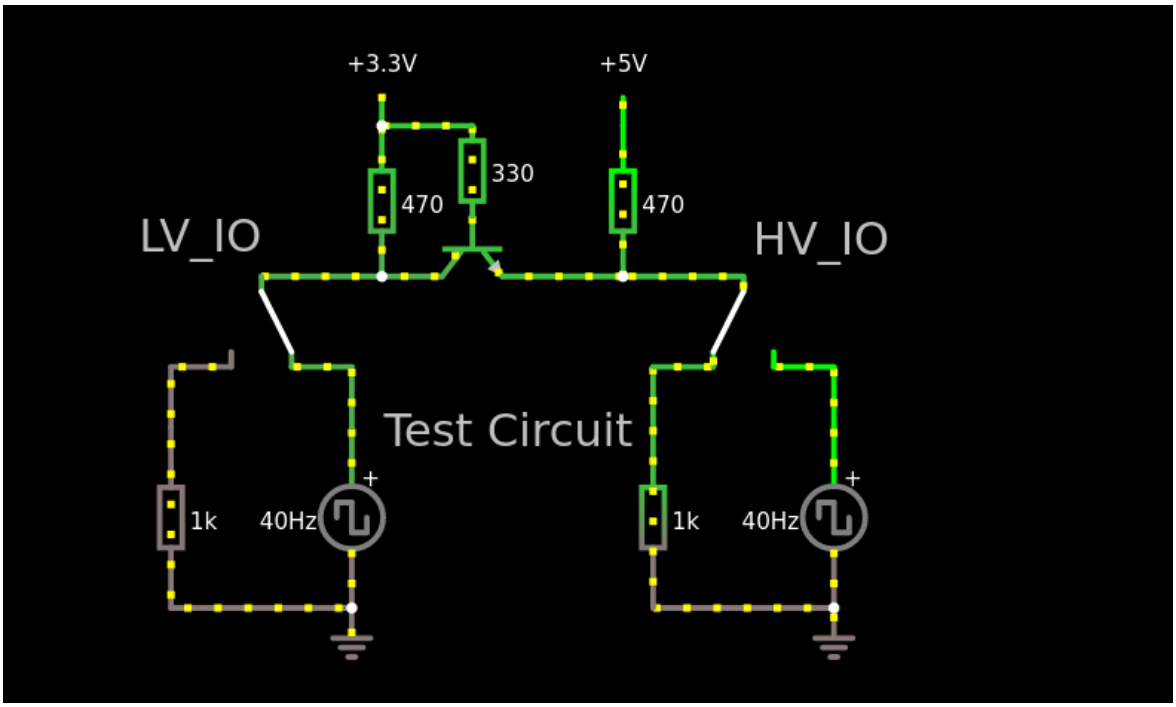


Figure 4: BJT Level Shifter

This didn't seem to work particularly well for 5v-3.3v in practice. For that, a simple common-collector PNP circuit was used.

Figure 5 is the final circuit including both level shifters.

The level shifters are required to prevent damage to the 3.3V ESP8266. Additionally, the 5V device may not properly read the logic signals from the 3.3 device.

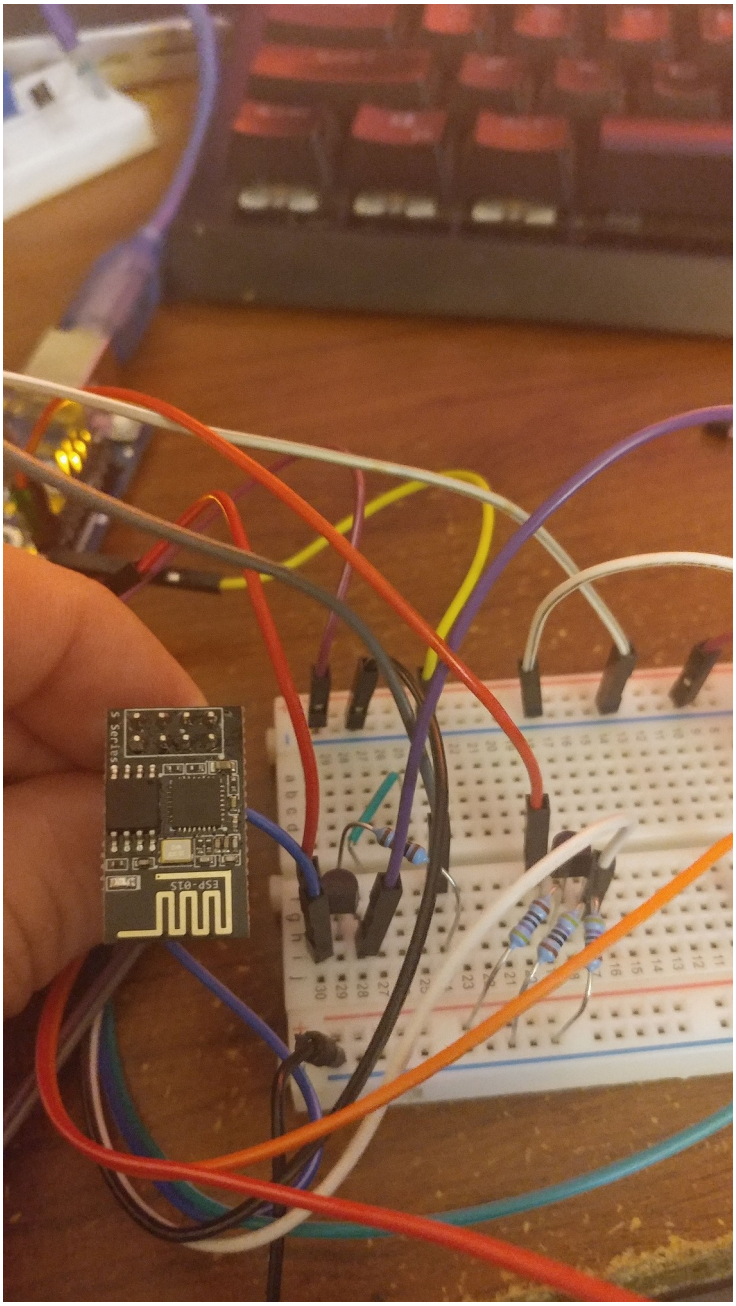


Figure 5: Circuit

Part 2

Firmware Update

Updating the firmware was difficult. I did the procedure on Linux, Ubuntu to be precise. I used a tool called “esptool.py”, with a command-line interface. Uploading the binary from the class website was no problem, but every time I tried to use the ESP8266 afterwards it would send a checksum error and not respond to any AT commands. Eventually, after trying half a dozen other binaries from other sources (including the epressif official binaries), I found that the recommended mode for most ESP8266s didn’t work for me. Originally the SPI flash module on the ESP8266 board uses the “dio” mode, but I had to use the “dout” mode in order for any binary to work.

Once I figured that out, I re-flashed the original binary from the class website and verified that the AT commands worked.

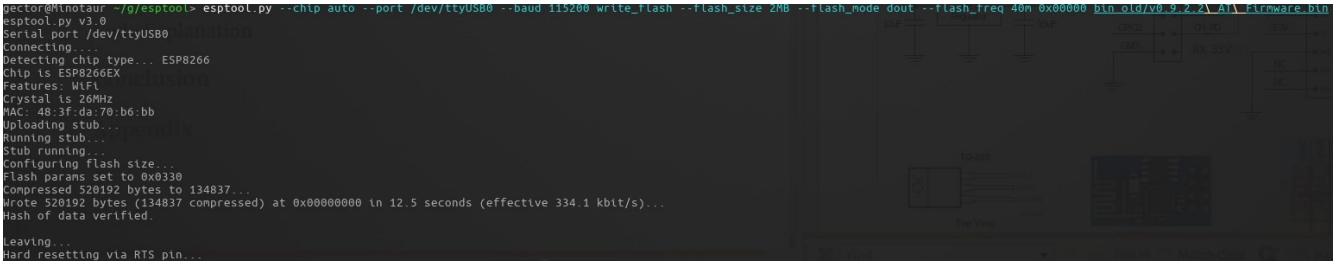


Figure 6: Flasher Output

Part 3

Website Connection

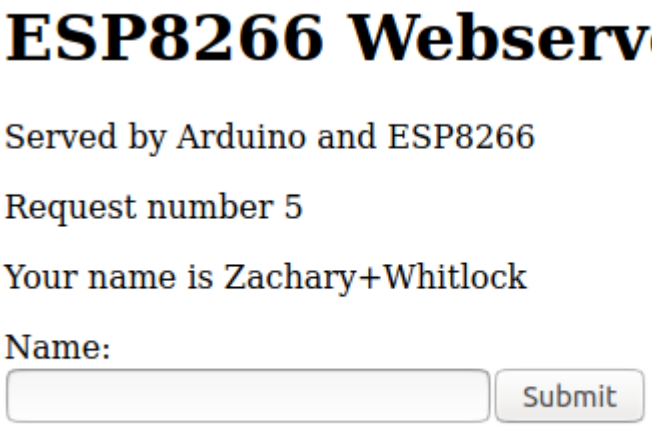


Figure 7: Website after submitting name

Figure 8: Webpage before submitting name

Code Explanation

The **SoftwareSerial.h** file is required in this program for creating an additional RX/TX port. The Arduino originally has just the one which is used for communicating with the PC, so software is used to write and receive data from digital pins 8&9.

The setup of the code sets up the ESP8266 to provide a webpage after connecting to my network.

The loop code checks to see if there is data ready to be read from the ESP. If the message from the ESP starts with “IPD” then the rest is printed to the serial monitor. If the message starts with “name”, the code reads the text that was sent from the ESP and places it in a variable called “name”. The code will supply HTML when a new client connects to the ESP, and if given a name the code will embed it in the HTML.

The function ‘strcpy’ is used liberally to copy pieces of HTML code into the “html” string. In addition to the generic HTML, an text entry box and a button are sent to the querying computer for submitting the name to the ESP.

The getReply() function is required for getting text from the ESP. It takes a time in milliseconds and attempts to read data from the software RX/TX for that duration. Upon a reply read from the software RX pin, it is printed to the Serial monitor. Additionally, the data is copied to the ‘reply’ global string so it can be used in other places in the program.

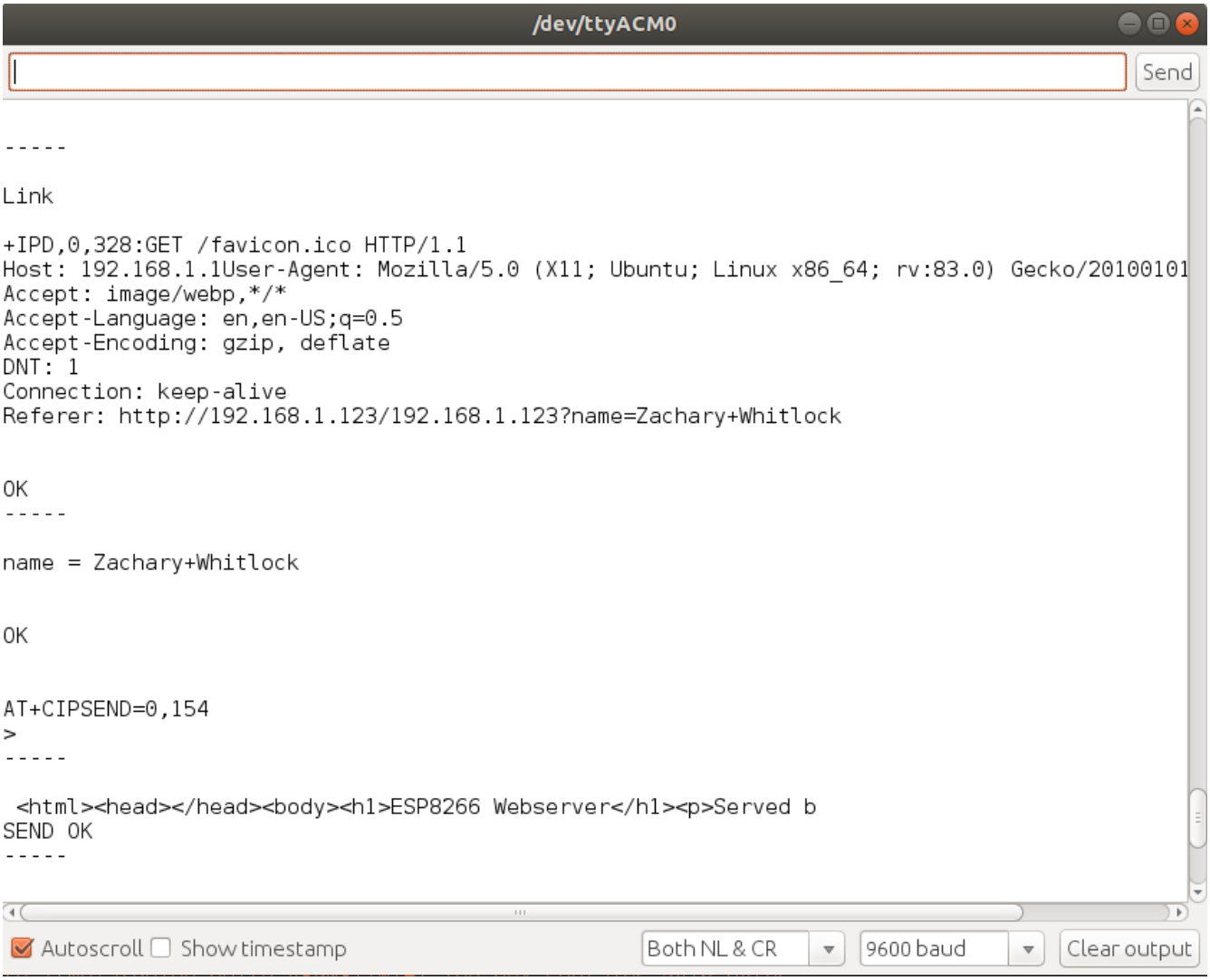


Figure 9: Serial Monitor

Conclusion

In this lab I learned how to use the ESP826 as a wifi interface for the simpler Arduino. I know it's possible to program these by themselves and use them as a I/O limited microcontroller with built-in WiFi capabilities. They have a lot of flash and more can be added using SPI flash modules which are commonly used. They are more complex than Arduinos and have a different architecture which makes them not so easy to develop for, though.

It was educational to debug the firmware process since I had to try so many binaries. It gave me a good idea as to what is available for the device and what people use it for. It was useful to have a 3.3V/5V FTDI converter so I could test my connections and circuit. I also used my oscilloscope to check the output of my level shifters and to verify their output. I believe that the design didn't work very well because it seemed like the RX pin of the ESP8266 was pulled high fairly stiffly. Finally, when I got the firmware working, the code supplied for the Arduino worked without a problem and I was able to open the IP address given to the ESP in a web browser.

Appendix

```
// Basic Arduino & ESP8266 webserver
//
// uses AltSoftSerial download from
https://www.pjrc.com/teensy/td_libs_AltSoftSerial.html
// this can be replaced with the normal software serial


#include <SoftwareSerial.h>
// Arduino pin 08 for RX
// Arduino Pin 09 for TX

SoftwareSerial espSerial(8, 9);


const bool printReply = true;
const char line[] = "-----\n\r";
int loopCount=0;

char html[256];
char command[20];
char reply[500]; // you wouldn't normally do this

char ipAddress [20];
char name[30];
```

```
int lenHtml = 0;
char temp[5];


void setup()
{
      Serial.begin(9600);
      Serial.println("Start\r\n\r\n");

      espSerial.begin(9600);

      // reset the ESP8266
      Serial.println("reset the module");
      espSerial.print("AT+RST\r\n");
      getReply( 1000 );


      // configure as a station
      Serial.println("Change to station mode");
      espSerial.print("AT+CWMODE=1\r\n");
      getReply( 1500 );
      getReply( 1000 );
      getReply( 1000 );

      // List networks I can see
      //Serial.println("Listing networks: ");
      //espSerial.print("AT+CWLAP\r\n");
      //getReply(600);

      // connect to the network. Uses DHCP. ip will be assigned by the router.
      Serial.println("Connect to a network ");

     // Enter the SSID and password for your own network
      espSerial.print("AT+CWJAP=\"SSID\",\"password\"\r\n");
      getReply( 6000 );


      // get ip address
      Serial.println("Get the ip address assigned by the router");
      espSerial.print("AT+CIFSR\r\n");
      getReply( 5000 );

      // parse ip address.
      int len = strlen( reply );
      bool done=false;
      bool error = false;
```

```
int pos = 0;
while (!done)
{
    if ( reply[pos] == 10) {
       done = true;
    }
    pos++;
    if (pos > len) {
       done = true;
       error = true;
    }
}

if (!error)
{
     int buffpos = 0;
     done = false;
     while (!done)
     {
        if ( reply[pos] == 13 ) {
           done = true;
        }
        else {
           ipAddress[buffpos] = reply[pos];
           buffpos++; pos++;
        }
     }
     ipAddress[buffpos] = 0;
}
else {
  strcpy(ipAddress,"ERROR");
}



// configure for multiple connections
Serial.println("Set for multiple connections");
espSerial.print("AT+CIPMUX=1\r\n");
getReply( 1500 );


// start server on port 80
Serial.println("Start the server");
espSerial.print("AT+CIPSERVER=1,80\r\n");
getReply( 1500 );
```

```
    Serial.println("");


    Serial.println("Waiting for page request");
    Serial.print("Connect to ");
    Serial.println(ipAddress);
    Serial.println("");
}

int web_delay = 2000;

void loop()
{
    if(espSerial.available()) // check if the ESP8266 is sending data
    {
        // this is the +IPD reply - it is quite long.
        // normally you would not need to copy the whole message in to a variable you can
copy up to "HOST" only
        // or you can just search the data character by character as you read the serial port.
        getReply( 2000 );


        bool foundIPD = false;
        for (int i=0; i<strlen(reply); i++)
        {
            if (  (reply[i]=='I') && (reply[i+1]=='P') && (reply[i+2]=='D')   ) { foundIPD = true;    }
        }


        if ( foundIPD  )
        {

            loopCount++;
            // Serial.print( "Have a request.  Loop = ");  Serial.println(loopCount);
Serial.println("");


            // check to see if we have a name - look for name=
            bool haveName = false;
            int nameStartPos = 0;
            for (int i=0; i<strlen(reply); i++)
            {
                if (!haveName) // just get the first occurrence of name
                {
                    if (  (reply[i]=='n') && (reply[i+1]=='a') && (reply[i+2]=='m') && (reply[i+3]=='e')
&& (reply[i+4]=='=') )
```

```
            {
                haveName = true;
                nameStartPos = i+5;
            }
        }
    }

    // get the name - copy everything from nameStartPos to the first space character
    if (haveName)
    {
        int tempPos = 0;
        bool finishedCopying = false;
        for (int i=nameStartPos; i<strlen(reply); i++)
        {
            if ( (reply[i]==' ') && !finishedCopying ) {
              finishedCopying = true;
            }
            if ( !finishedCopying ) {
              name[tempPos] = reply[i];
              tempPos++;
            }
        }
        name[tempPos] = 0;
    }

    if (haveName) {
      Serial.print( "name = ");
      Serial.println(name);
      Serial.println("");
    }
    else {
      Serial.println( "no name entered");
      Serial.println("");
    }



    // start sending the HTML

    strcpy(html,"<html><head></head><body>");
    strcat(html,"<h1>ESP8266 Webserver</h1>");
    strcat(html,"<p>Served by Arduino and ESP8266</p>");
    strcat(html,"<p>Request number ");
    itoa( loopCount, temp, 10);
    strcat(html,temp);
    strcat(html,"</p>");
```

```
if (haveName)
{
    // write name
    strcat(html,"<p>Your name is ");
    strcat(html, name );
    strcat(html,"</p>");
}

 // Send the command to ESP8266
 lenHtml = strlen( html );
 itoa(lenHtml, temp, 10);

 strcpy(command,"AT+CIPSEND=0,");
 strcat(command,temp);
 strcat(command,"\r\n");

 espSerial.print(command);
 getReply( web_delay );
 espSerial.print(html);
 getReply( web_delay );


 // Construct the name entry box
 strcpy(html,"<form action=\"");
 strcat(html, ipAddress);
 strcat(html, "\" method=\"GET\">");
 strcat(html,"Name:<br><input type=\"text\" name=\"name\">");  // new
 strcat(html,"<input type=\"submit\" value=\"Submit\"></form>");  //new
 strcat(html,"</body></html>");  // new

  // Send the command to ESP8266
 lenHtml = strlen( html );
 itoa( lenHtml, temp, 10);
 strcpy(command,"AT+CIPSEND=0,");
 itoa( lenHtml, temp, 10);
 strcat(command, temp);
 strcat(command, "\r\n");

 espSerial.print(command);
 getReply( web_delay );
 espSerial.print(html);
 getReply( web_delay );

 // close the connection
 espSerial.print( "AT+CIPCLOSE=0\r\n" );
```

```
            getReply( web_delay );


        } // if(espSerial.find("+IPD"))
      } //if(espSerial.available())


      delay (100);

      // drop to here and wait for next request.

}


void getReply(int wait)
{
    int tempPos = 0;
    long int time = millis();
    char c;
    boolean complete_flag = 0;

    while( (time + wait) > millis() && !complete_flag)
    {
        while(espSerial.available())
        {
            c = espSerial.read();
            if (tempPos < 500) {
              reply[tempPos] = c;
              tempPos++;
            }
            if((reply[tempPos-2] == 'O' && reply[tempPos-1] == 'K') || (reply[tempPos-2] == '\n'
&& reply[tempPos-1] == '>')) {
              complete_flag = 1;
              break;
            }
        }
        reply[tempPos] = 0;  //NULL
    }


    //Serial.println(millis()-time);


    if (printReply) {
      Serial.println( reply );
```

```
        Serial.println(line);      }
}
```