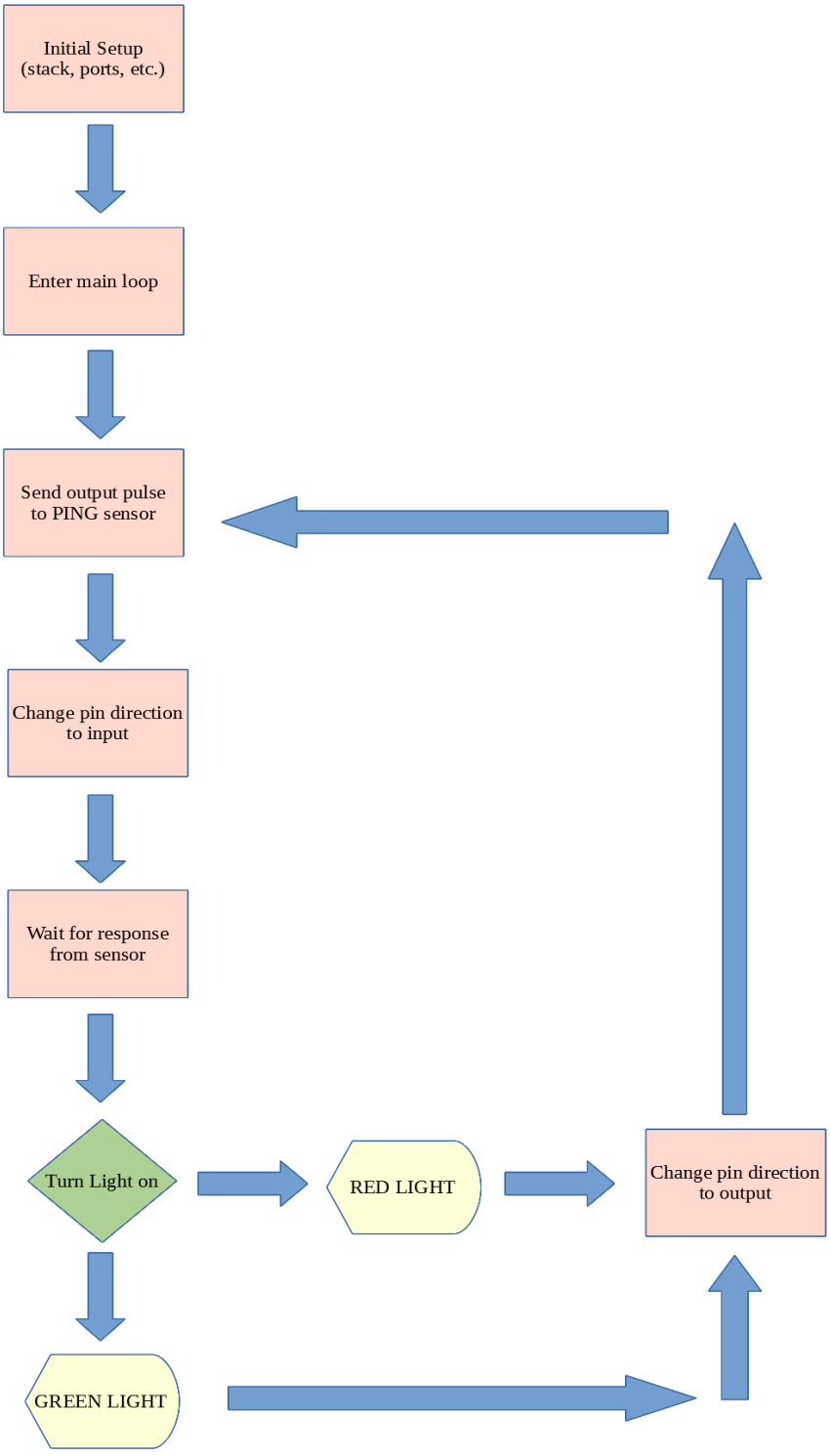
EECSX497.2 – Final Project
Matt Logan

To start, I decided to make a garage “traffic light” to let me know when I am far enough in my garage so that I can shut the garage door while still leaving myself enough room to walk in front of my car. I’ve never hit the wall in my garage while trying to pull in, it’s not quite that tight. But I have on many occasions left myself with no room to walk between my car and the back wall. When this happens, I usually start to walk around toward the front of my car thinking I can squeeze through, only to find out I can’t and I’ve now wasted a few precious steps while also leaving myself with an extra 15-20 steps to go around the back of the car. *Oh, the horror*. So to combat this very serious problem, I decided to make the garage traffic light.

I had originally planned on just simulating this in Atmel Studio, but the next class I signed up for required the ATSTK600 board, so I went ahead and got one of those from Microchip. The board came with an ATmega2560 so that will be my controller. It is somewhat pricey, but when you get your hands on it and realize all that it’s capable of, it starts to make sense. While I was at it, I went ahead and grabbed a PING sensor off the internet to detect the distance of the car from the wall. This ended up being sort of a bad choice, which I will go into a little more detail in the next paragraph. I took advantage of a relationship my work has with Grainger to get a simple 24V red/green traffic light. Since the ATSTK600 board can’t put out 24V, I borrowed a surplus 24V power supply from my work as well. The last thing I needed was a way to control the 24V going to the traffic light. To do this, I got some small mechanical relays off the internet. The relays I used had a switching voltage of 5V so that the board easily can switch them off and on, and they are more than capable of handling the current draw of the traffic light at 24V, so they were ideal for the slow switching speed of the traffic light circuit.

Once I had all the components picked out, it was time to start putting them together. A full circuit schematic can be found at the end of this document. To turn the lights on, the mechanical relay must be activated and allow current to pass through. We want the lights to turn on and off independent of each other so we needed two relays for our circuit. This would in turn mean we need two output pins to control the relays. The PING sensor is a 3 pin sensor: 5V, GND, and Input/Output Signal. The fact that this sensor has only 3 pins made it challenging to use with the ATmega2560 assembly language. We would have to use one pin from our board and switch it between input and output. This is simple enough to do, but it still presented a challenge. The documentation on the PING sensor wasn’t the best, but their website does provide a small example application[1]. This sensor is an ultrasonic sensor, meaning it relies on very high frequency sound waves to determine the distance an object is in front of it. It sends an ultrasonic pulse and waits for the reflection of the pulse back to the sensor and calculates the time in microseconds it takes to receive the pulse. Using the ATSTK600, we must send a pulse to the sensor signal pin, then switch our microcontroller pin to an input, and finally we will wait for a response pulse from the sensor which will provide our time in microseconds that the sensor read. This didn’t sound too bad at first, but as I sat and worked on it, it became apparent that this wasn’t as easy as it sounded. The operation flow on the next page shows the steps we will need to take to read the sensor.

EECSX497.2 – Final Project
Matt Logan



Now that we know the general flow of the program, we can start writing the AVR assembly code to achieve our goal. The first thing we would do is set up our stack. The following snippet of code shows how we do this:

```
38  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
39  ; Set up the stack by pointing the end of SRAM (the stack grows down toward lower addresses)
40  ; This means our stack pointer must be 2 bytes, in order to reach every memory location
41  ; we will use LOW() and HIGH() to move the end of SRAM (SRAMEND) into the low and high bytes of SP respectively
42  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
43  ldi     TEMP, low(RAMEND)               ; move the low byte of RAMEND into r16
44  out     SPL, TEMP                       ; move the low byte of RAMEND (r16) into the low byte of SP
45  ldi     TEMP, high(RAMEND)              ; move the high byte of RAMEND into r16
46  out     SPH, TEMP                       ; move the high byte of RAMEND (r16) into the high byte of SP
47  clr     TEMP                            ; clear the contents of r16
```

After we have our stack initialized, we can move on to setting port directions. As mentioned before, we will need two outputs. One for our red light relay, and one for our green light relay. The following snippet sets up PORTA0 and PORTA1 as output pins.

```
49  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
50  ; Set PORTA as outputs
51  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
52  ldi     TEMP, 0x03                      ;
53  out     DDRA, TEMP                      ; set PORTA1:0 as output
54  clr     TEMP                            ;
55  out     PORTA, TEMP                     ; write all outputs low
56
```

To make a pin an output, we must set it's direction bit in its DDRn register. In the case of PORTA, we want to set bits 0 and 1 (0x03) in DDRA to make these pins outputs.

The next thing to do is choose a pin to connect to the sensor signal pin. I ended up choosing PORTL0 as my signal pin. I will discuss the reason for this choice in a little more detail soon. For now, we just want to send an output pulse from PORTL0 to the signal pin of the sensor. To do this, we first need to set PORTL0 as an output. This is done in much the same way as the PORTA0 and PORTA1 pins, with one pretty big difference. The DDRA register is at location 0x01 (this can be found by searching through the m2560def.inc file in Atmel Studio), so it can be written to using the OUT instruction of the AVR assembler language. DDRL however is memory mapped at 0x10A. This means it is out of range of the possible OUT instruction operands[2], so we must use STS to store the information in DDRL. The following code shows how to do this:

```
194 SET_DDRL0_OUTPUT:
195     ; Subroutine to set PORTL0 as an output
196     push    TEMP                    ; save TEMP
197     ldi     TEMP, 0x01              ; PORTL0 direction must be high
198     sts     DDRL, TEMP              ;
199     pop     TEMP                    ; restore TEMP
200     ret                             ;
```

I decided to make this into a subroutine since we would be doing this action pretty often during normal operation.

Now that PORTL0 is set as an output, we can send our output pulse to turn on the sensor. The following code shows how I did this. I also decided to make this a subroutine since I would be doing it often in the main loop.

```
211  PULSE_OUTPUT_PL0:
212      ; Subroutine to send a pulse output to PING sensor (PORTL0)
213      push    TEMP                        ; save temp before starting
214      clr     TEMP                        ; Now set the output low to clear the line
215      sts     PORTL, TEMP                 ; PORTL is outside the range of the OUT instruction so we must use STS
216      ldi     TEMP, (1 << PORTL0)         ; Send a high pulse to the PING sensor
217      sts     PORTL, TEMP                 ;
218      nop                                 ; a few nop's to let the output pulse settle at the input pin
219      nop                                 ; of the distance sensor
220      nop                                 ;
221      clr     TEMP                        ; Clear the output line again by setting the output low
222      sts     PORTL, TEMP                 ; PORTL is outside the range of the OUT instruction so we must use STS
223      pop     TEMP                        ; restore temp
224      ret                                 ;
```

Hopefully the comments help to clearly indicate what each line does. Lines 218 through 220 are simple NOP instructions. I used these to give the output pulse some time to settle. The example application provided by the vendor shows a 10us pulse to the sensor. I did not calculate a delay, so to allow the signal to settle at the sensor, I added these NOP instructions.

After the pulse to the sensor, we must quickly change our PORTL0 direction from output to input so that we can be ready for the response pulse from the sensor. Again, I wrote a subroutine that we can call to do this for us:

```
202  SET_DDRL0_INPUT:
203      ; Subroutine to set PORTL0 as an input
204      push    TEMP                        ; save TEMP
205      clr     TEMP                        ; PORTL0 direction must be low
206      sts     DDRL, TEMP                  ; PORTL is outside the range of the OUT instruction so we must use STS
207      pop     TEMP                        ; restore TEMP
208      ret
```

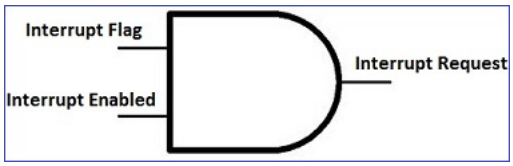
We now have to read the response pulse from the sensor on PORTL0 and time how long the pulse lasts.

This sounded simple at first. We simply need to read PORTL0, and while it's high, we will run a timer. Once this goes signal goes low, we will turn the timer off and read our time value to determine whether we need to turn on the red light or turn on the green light. I ended up doing almost exactly this, but doing this turned out to be much more complicated than I was initially thinking it would be.

There are no timing instructions in the AVR assembler instruction set. To find a solution to the problem, I went down the rabbit hole that is the ATmega2560 Datasheet [3]. What I eventually found out after a few hours of reading was that the ATmega2560 has what is called an Input Capture Interrupt.

Before we understand what this is, let me quickly describe what an interrupt is. One definition of an interrupt is "an event that requires immediate attention by the microcontroller" (http://www.avr-tutorials.com/interrupts/about-avr-8-bit-microcontrollers-interrupts). When an interrupt is enabled, if

an interrupt is received, the interrupt flag will be set. The interrupt enable bit and the interrupt flag bit are AND together as shown by the circuit here:



This then produces our interrupt request. When an interrupt request is received, the associated interrupt vector is looked up. An interrupt vector is basically a location in memory that stores the address of the interrupt service routine used to service the interrupt request. The interrupt service routine does what we need it to do much like any other normal subroutine in assembly language and then the ISR returns control to the main program, resetting the interrupt flag in the process. This is all explained in much better detail in the AVR Turorial link above. An Input Capture Interrupt is an interrupt that occurs when an input is detected on a given input capture pin. The input capture pins are all listed in the ATmega2560 datasheet. I chose Input Capture Pin 4, or ICP4, which is located on the PORTL0 pin. The interrupt is tied to a 16-bit timer/counter, Timer4. When the input capture event occurs, this 16-bit timer value is stored. The 16-bit timer value is located in TCNT4, which is broken into a high and a low byte, TCNT4H and TCNT4L respectively. During the input capture event, the values of these bytes are moved to the ICR4H and ICR4L bytes to store the timestamp of the interrupt event. The input capture interrupt can be configured in many ways. One of the main ways it can be configured is to detect a rising or a falling edge of a signal. This, along with the time stamp of the timer register TCNT4, will be important when we try to measure the time of an input pulse. Remember the duration of the input pulse is proportional to the distance of an object from the sensor.

Using the input capture event solves the first half of our problem of catching an input pulse. But a pulse will have two distinct events. A rising edge and a falling edge. So after we catch the rising edge of the pulse, we need to then catch the falling edge of the same pulse. The time between the rising edge of the pulse and the falling edge of the pulse will be our pulse duration, and it will help us figure out how far away an object is from the sensor. This means that after the interrupt event caused by the rising edge of the sensor response pulse, we need to switch the detection method of the input capture interrupt from rising edge detection to falling edge detection. To get a meaningful pulse duration from the TCNT4 value, we should clear it at every input capture interrupt. Remember that TCNT4 is a 16-bit value, with its high byte stored in TCNT4H and its low byte stored in TCNT4L. Page 135 of the ATmega2560 datasheet [3] explains how to write and read data to/from 16-bit registers. In short, when we are writing to the 16-bit register, we must write the high byte first, and the low byte second. Reading from the 16-bit register is just the opposite. We must read from the low byte first, and then read from the high byte second. This is all given in detail in the ATmega2560 datasheet.

With all of that information regarding interrupts in mind, we can start to put it together. First, we must tell our program to jump to our ICP service routine. This is done by pointing the given interrupt vector to our service routine. The following code shows how to do this:

```
18  .org            0x0052                  ; Inputer Capture 4 Interrupt
19  jmp             ISR_TIMER4_CAPT         ;
```

The interrupt vector addresses for all available interrupts on the ATmega2560 can be found in Table 14.1 on page 101 of the datasheet. The m2560def.inc file also includes definitions for all of the interrupt vectors available that would make this line slightly more readable.

The next thing we have to do is setup the interrupt, clear the interrupt flag, and then enable the interrupt. This involves 3 registers: TCCR4B for the setup, TIFR4 for the interrupt flag, and TIMSK4 to enable the interrupt. TCCR4B is described on page 156 of the datasheet, TIFR4 is on page 162 of the datasheet, and TIMSK4 is on page 161 of the datasheet. The code to set these up is as follows:

```
68  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
69  ; Setup Input Capture Interrupt 4 (ICP4) on PORTL0
70  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
71  ldi     TEMP, (1 << ICNC4) | (1 << ICES4) | (1 << CS40) ; 1/1 prescaler, rising edge detection,
72                                                          ; noise cancellation
73  sts     TCCR4B, TEMP                    ; TCCR4B = 0b11000001
74  ldi     TEMP, (1 << ICF4)               ; write a 1 to ICF4 to clear any pending interrupts that may be there
75  sts     TIFR4, TEMP                     ;
76  ldi     TEMP, (1 << ICIE4)              ; Input Capture Interrupt Enable 4
77  sts     TIMSK4, TEMP                    ;
```

The bit descriptions of each bit in the given register can be found on the pages given above. These registers are all memory mapped, so they must be accessed and written to using the lds/sts instructions.

There is one other bit that must be enabled to allow the interrupt events to have any effect. That bit is the global interrupt bit in the status register, SREG. AVR assembly has two instructions that are directly related to this bit: **SEI** and **CLI**. **SEI** enables the global interrupt bit, and **CLI** disables it. These two instructions will prove to be pretty important in the main loop of the program.

Now that we know how to setup and enable the input capture interrupt, it's time to write the interrupt service routine. The interrupt service routine will be called immediately when an interrupt event occurs. This means the main program will stop and turn over control to the ISR (interrupt service routine). In order for the ISR to return control to the main program, a special return instruction is used at the end. **RETI** must be used instead of the normal RET instruction to return from an ISR. Any other method of exiting the ISR can lead to many problems. My input capture ISR is shown in the code on the next page.

```
174  ISR_TIMER4_CAPT:
175      ; Timer4 Input Capture Interrupt Service Routine
176      ; Here we will capture our sensor read values.  Even though we only use the high byte, we must read the low
177      ; byte too.  More can be read about this in section 17.3 of the ATmega2560 datasheet.
178      ; After we get our sensor value, we will increment the IC4_FLAG.  We don't bother to check its value here
179      ; since we do all that logic in the main loop.  This will keep our interrupt routine shorter.
180      ; Lastly, we will restart the TCNT4 register to 0 so we can get a meaningful result from the measurement.
181      lds     CAPTURE_LOW, ICR4L          ; read low byte first (see 17.3 Accessing 16 Bit Registers in ATmega2560
182      lds     CAPTURE_HIGH, ICR4H         ;
183      inc     IC4_FLAG                    ; increment the IC4 flag
184      push    TEMP                        ; save TEMP register
185      clr     TEMP                        ; clear TCNT4
186      sts     TCNT4H, TEMP                ; write to TCNT4H first (Section 17.3 ATmega2560 datasheet)
187      sts     TCNT4L, TEMP                ; write to TCNT4L only after writing to the high byte
188      pop     TEMP                        ; restore TEMP register
189      reti
```

The first thing we do in our ISR is read the captured timer value in ICR4. Again, to access a 16-bit register, we first read the low byte and then the high byte. At the start of the program, I have defined CAPTURE_LOW and CAPTURE_HIGH to be two successive registers. That can be found in the code provided. The TEMP register is also defined along with them. After we've read our timer values, we increment our input capture flag. This was also defined at the start of the program to hold a value that tells us whether we are looking for a rising edge or a falling edge. The ISR increments this value only. It does not use any logic to determine what kind of edge we have detected. That logic, along with clearing the flag, is reserved for the main program to keep the ISR short and manageable. The next thing to do is restart the timer. To do this, we save the TEMP register contents and then clear it out before we write this register value to both the high and low bytes of TCNT4H. Again, high byte is written first and then the low byte. We restore TEMP and return from the ISR using the RETI instruction. Notice that there is not a lot of logic in the ISR. It is simply capturing and acting on variables that are used by the main program logic to determine what to do.

Now that we have the interrupt setup and enabled and the ISR is defined, it is time to move on to the main program. This is somewhat long so I will go over it in sections. The first part of the main program is:

```
70   MAIN_LOOP:
71       rcall   SET_DDRL0_OUTPUT            ; First set the sensor in/out pin as an output
72       rcall   PULSE_OUTPUT_PL0            ; Send a pulse to the sensor
73       rcall   SET_DDRL0_INPUT             ; Change the sensor pin to an input
74
```

Here we have defined the label of the main loop as, fittingly, MAIN_LOOP. The first thing we do in our main loop is set PORTL0 as an output using the DDRL register. I have this action defined in the SET_DDRL0_OUTPUT subroutine. Once PORTL0 is set as an output, we can send the output pulse to the sensor. Again I've defined this in the subroutine PULSE_OUTPUT_PL0. After the output pulse has been sent, we need to switch PORTL0 to an input pin to receive the response pulse from the sensor. We do this with the SET_DDRL0_INPUT subroutine.

The next thing to do is wait for a rising edge interrupt on PORTL0/ICP4. This code snippet shows how I chose to do that:

```
75      sei                             ; enable global interrupts
76      WAIT_FOR_RISING_EDGE:
77      cpi     IC4_FLAG, 0x01          ; check our interrupt flag
78      brne    WAIT_FOR_RISING_EDGE    ; If we didn't encounter an interrupt, keep waiting
```

First, the global interrupt flag must be enabled. This is done with AVR instruction SEI which I previously described. The next thing we do is define a label that will allow us to branch to it if we have no yet received an input capture event. At this point in the program, the IC4_FLAG value is 0 (0x00). We compare the IC4_FLAG to the immediate value of 1 (0x01). If we haven't received an input capture interrupt, the IC4_FLAG will remain at 0 and the program will branch back to the WAIT_FOR_RISING_EDGE label. Once the interrupt occurs, the IC4_FLAG will be incremented to 1 (along with everything else that occurs in the ISR) and the program will continue on.

The next thing to do is change the edge detection method of the input capture interrupt to a falling edge detection. To do this, we disable the global interrupt flag and then call our subroutine SET_FALLING_EDGE_DETECTION, as seen here:

```
75      sei                             ; enable global interrupts
76      WAIT_FOR_RISING_EDGE:
77      cpi     IC4_FLAG, 0x01          ; check our interrupt flag
78      brne    WAIT_FOR_RISING_EDGE    ; If we didn't encounter an interrupt, keep waiting
```

The SET_FALLING_EDGE_DETECTION is a subroutine I defined to help with readability. That subroutine is as follows:

```
160  SET_FALLING_EDGE_DETECTION:
161      ; Subroutine to change the edge detection method of Timer4 input capture to falling edge
162      push    TEMP
163      ldi     TEMP, (1 << ICNC4) | (1 << CS40) ; 1/1 prescaler, falling edge detection, noise cancellation
164      sts     TCCR4B, TEMP                ;
165      pop     TEMP                        ;
166      ret
```

This is not part of the ISR. It is its own subroutine called during normal operation so it uses the standard RET instruction. Setting falling edge detection is similar to the initial setup for rising edge detection with the exception of the ICES4 bit. When this bit is 1 in the TCCR4B register, rising edge detection is used. When this bit is 0, falling edge detection is used. Since we will be routinely switching between edge detection modes, I also wrote a subroutine that switches to rising edge detection:

```
152   SET_RISING_EDGE_DETECTION:
153       ; Subroutine to change the edge detection method of Timer4 input capture to rising edge
154       push    TEMP
155       ldi     TEMP, (1 << ICNC4) | (1 << ICES4) | (1 << CS40) ; 1/1 prescaler, rising edge detection,
156                                                               ; noise cancellation
157       sts     TCCR4B, TEMP              ; TCCR4B = 0b11000001
158       pop     TEMP                      ;
159       ret
```

This is done in almost the same way as the initial setup at the start of the program, it is just done within a subroutine.

Back to the main loop. After we have disabled the global interrupt flag and then set falling edge detection, we are ready to receive the falling edge input capture event. This is done very similarly to waiting for the rising edge event:

```
85        sei                               ; reenable global interrupts
86        WAIT_FOR_FALLING_EDGE:
87        cpi     IC4_FLAG, 0x02            ; check the interrupt flag again
88        brne    WAIT_FOR_FALLING_EDGE     ; check to see if we received a falling edge interrupt
89        cli                               ; disable global interrupts
90        rcall   SET_RISING_EDGE_DETECTION ; switch to rising edge detection mode
```

Again, we have to re-enable the global interrupt flag using the SEI instruction. We also create another label, this one called WAITING_FOR_FALLING_EDGE. At this point in the program, the IC4_FLAG value should be 1. Another interrupt will increment this value to 2 and we will break out of the loop just as before. After the interrupt event, we disable global interrupts and reset the edge detection to rising edge detection so we don't have to do it at the start of the program. Disabling global interrupts also ensures that no unwanted interrupts occur and halt the main program.

Now that both a rising edge and a falling edge interrupt has occurred, the CAPTURE_HIGH and CAPTURE_LOW bytes should hold our sensor response pulse time in microseconds. This means it's time to decide whether to turn on the red light or the green light. The logic for that decision is shown in the following code:

```
 93        ; Decide whether to turn red or green light on
 94        cpi     CAPTURE_HIGH, 0x0E          ;
 95        brlo    RED_ON                      ;
 96        GREEN_ON:
 97        ldi     TEMP, 0xFD                  ; turn PA1 on
 98        out     PORTA, TEMP                 ;
 99        rjmp    START_RETURN                ;
100        RED_ON:
101        ldi     TEMP, 0xFE                  ; turn PA0 on
102        out     PORTA, TEMP                 ;
103        START_RETURN:
104        clr     IC4_FLAG                    ; restart the IC4 flag register at 0
105        rjmp    MAIN_LOOP                   ;
```

EECSX497.2 – Final Project
Matt Logan

The first thing we do is compare our CAPTURE_HIGH byte to a magic number. The right way to do this would be to use both the high and low bytes as well as some math to change the value from time in microseconds to a distance value using the speed of sound. I chose the brute force method instead. Through trial and error, I ended up using the magic number shown 0x0E. This led to a distance of roughly 21 inches from the sensor. That should be enough room for me to walk through while also making sure the car is fully in the garage. If our captured high byte is lower than this, the BRLO instruction tells us to branch to the RED_ON label, which simply turns the red light on and all other pins off. If the captured high byte is higher than this magic number, we simply go on to the GREEN_ON label which turns on the green light and turns off all other pins. This section then jumps to the START_RETURN label, which is the main return point of the MAIN_LOOP section. To restart the program, we clear the IC4_FLAG register so that we don't get stuck in any loops. The last thing to do is jump to the MAIN_LOOP label and start over.

To prove to myself that the program worked, I decided to simulate it before running it on the real hardware. The following sections will show the process of stepping through the program and visualizing the simulated registers to ensure proper function.

The first part of the program is rather uninteresting. We simply setup the interrupt vectors and define names for some registers we will commonly use:

```
 9   .include "m2560def.inc"
10   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
11   ; Vector Interrupt Table
12   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
13   .org            0x0000                  ; RESET Vector
14   jmp             RESET                   ;
15
16   .org            0x0052                  ; Inputer Capture 4 Interrupt
17   jmp             ISR_TIMER4_CAPT         ;
18
19   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
20   ; Define some names for common registers we will use
21   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
22   .def            TEMP = r16              ; r16 will be our temporary register
23   .def            CAPTURE_LOW = r18       ; r18 will store our low counter value when we capture input
24   .def            CAPTURE_HIGH = r19      ; r19 will store our high counter value when we capture input
25   .def            IC4_FLAG = r22          ; r22 will hold our IC4_FLAG value
26                                           ; 0x01 ==> we caught rising edge from the sensor
27                                           ; 0x02 ==> we caught falling edge from the sesor
```

I have not set any breakpoints here. The next part of the program is also rather uninteresting. In this section, we set the stack, initialize the IC4_FLAG register to 0, and set PORTA1:0 as outputs.

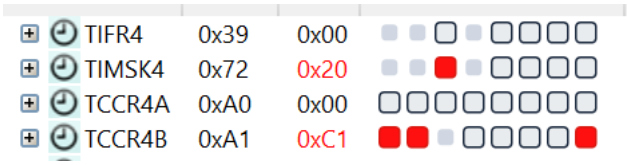
```
29  RESET:
30  clr     IC4_FLAG                    ; Make sure our IC4_FLAG register is clear to start
31
32  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
33  ; Set up the stack by pointing the end of SRAM (the stack grows down toward lower addresses)
34  ; This means our stack pointer must be 2 bytes, in order to reach every memory location
35  ; we will use LOW() and HIGH() to move the end of SRAM (SRAMEND) into the low and high bytes of SP respectively
36  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
37  ldi     TEMP, low(RAMEND)           ; move the low byte of RAMEND into r16
38  out     SPL, TEMP                   ; move the low byte of RAMEND (r16) into the low byte of SP
39  ldi     TEMP, high(RAMEND)          ; move the high byte of RAMEND into r16
40  out     SPH, TEMP                   ; move the high byte of RAMEND (r16) into the high byte of SP
41  clr     TEMP                        ; clear the contents of r16
42
43  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
44  ; Set PORTA as outputs
45  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
46  ldi     TEMP, 0x03                  ;
47  out     DDRA, TEMP                  ; set PORTA1:0 as output
48  clr     TEMP                        ;
49  out     PORTA, TEMP                 ; write all outputs low
```

The only real slightly interesting part is the “RESET:” label. This allows us to push the RESET button on the STK600 board and restart the program by calling the reset interrupt vector and starting the program at this location.

The next thing to do is setup and enable the input capture interrupt as described earlier:

```
51  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
52  ; Setup Input Capture Interrupt 4 (ICP4) on PORTL0
53  ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
54  ldi     TEMP, (1 << ICNC4) | (1 << ICES4) | (1 << CS40) ; 1/1 prescaler, rising edge detection,
55                                                          ; noise cancellation
56  sts     TCCR4B, TEMP                ; TCCR4B = 0b11000001
57  ldi     TEMP, (1 << ICF4)           ; write a 1 to ICF4
58  sts     TIFR4, TEMP                 ;
59  ldi     TEMP, (1 << ICIE4)          ; Input Capture Interrupt Enable 4
60  sts     TIMSK4, TEMP                ;
```

If we stop after line 60 when all the registers have been written to, we can view those registers in the I/O tool of Atmel Studio and see the following:



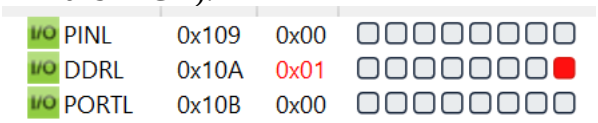
This shows us that all of our register have been correctly setup. Notice the TIFR4 register is 0 even though we wrote a 1 to one of its bits. This is because a 1 means there is an interrupt pending, but writing a 1 to the register clears that interrupt. So by writing to TIFR4, we have effectively cleared any interrupts on it.

Now we have reached the main loop of the program:

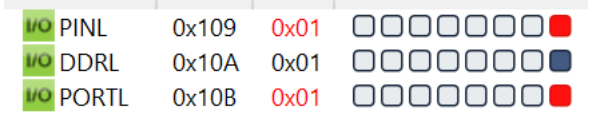
```
63   MAIN_LOOP:
64       rcall   SET_DDRL0_OUTPUT            ; First set the sensor in/out pin as an output
65       rcall   PULSE_OUTPUT_PL0            ; Send a pulse to the sensor
66       rcall   SET_DDRL0_INPUT             ; Change the sensor pin to an input
67
```

These subroutines will set our PORTL0 pin as an output, and then send an output pulse before setting the pin direction back to input. I will set breakpoints in the subroutines to show the pin as an output, then one to show the pin sending the pulse, and finally one to show the pin is back to an input:

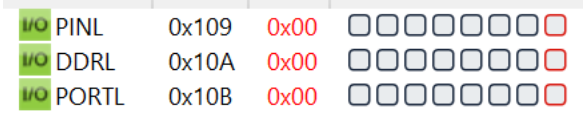
PORTL0 set as an output (DDRL0 is HIGH):



PORTL0 is sending an output pulse:



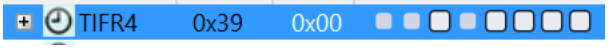
And finally, PORTL0 is reset as an input (DDRL0 is LOW):



So that portion is working well. The next part of the code waits for the rising edge interrupt to occur. There's nothing too interesting about this part specifically, but I will pause the program during the comparison instruction so that I can manually turn on the input capture interrupt flag. This will simulate an interrupt occurring. I will also set a break point in the ISR and step through that to show the captured value.

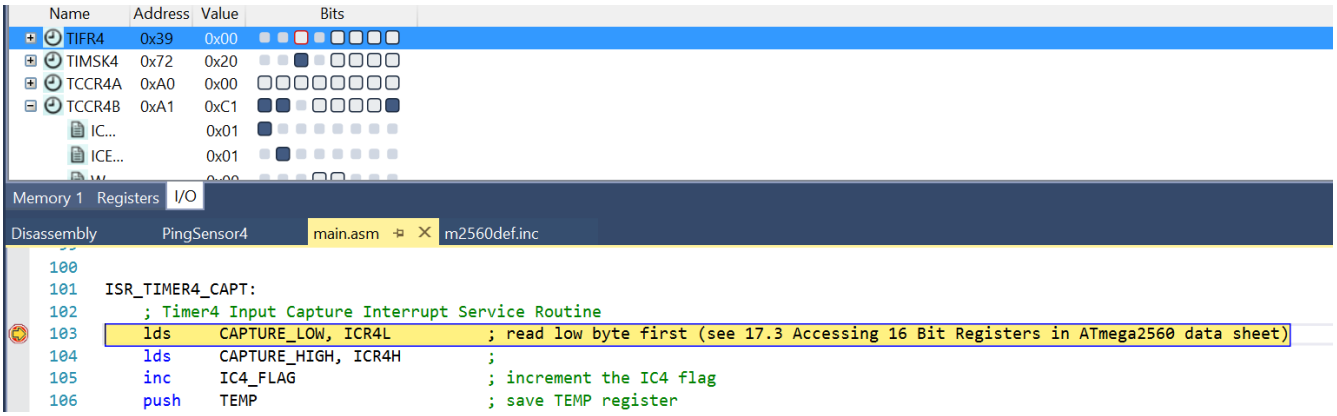
```
69       WAIT_FOR_RISING_EDGE:
70       cpi     IC4_FLAG, 0x01             ; check our interrupt flag
71       brne    WAIT_FOR_RISING_EDGE       ; If we didn't encounter an interrupt, keep waiting
```

Pausing the program here, we will see the TIFR4 register like this:



The ICP4 flag is bit 5, which can be found in the datasheet. If we turn this on, we will receive an interrupt and our program will go to the ISR.

EECSX497.2 – Final Project
Matt Logan



Also notice this clears the interrupt flag. I'll now step the program to line 106 so we can view the CAPTURE_LOW, CAPTURE_HIGH, and IC4_FLAG registers:

```
R00 = 0x00 R01 = 0x00 R02 = 0x00 R03 = 0x00 R04 = 0x00 R05 = 0x00 R06 = 0x00 R07 = 0x00 R08 = 0x00 R09 = 0x00 R10 = 0x00 R11 = 0x00 R12 = 0x00
  R13 = 0x00 R14 = 0x00 R15 = 0x00 R16 = 0x20 R17 = 0x00 R18 = 0x26 R19 = 0x00 R20 = 0x00 R21 = 0x00 R22 = 0x01 R23 = 0x00 R24 = 0x00 R25 = 0x20
   R26 = 0x00 R27 = 0x00 R28 = 0x00 R29 = 0x00 R30 = 0x00 R31 = 0x00

CYCLE_COUNTER = 0x00000066 FP = 0x0000 PC = 0x00000082 SP = 0x21FC X = 0x0000 Y = 0x0000 Z = 0x0000
```

IC4_FLAG is just register R22, and we can see that value is 0x01 just like we would expect. CAPTURE_LOW is R18 and CAPTURE_HIGH is R19. We can see these values and put them together to see our 16-bit counter value 0x0026.

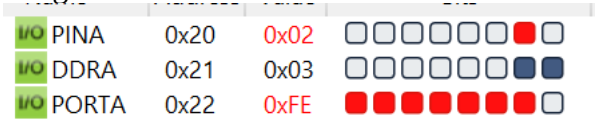
From here, we can let the program run until it gets stuck waiting for the falling edge. We again simulate the falling edge interrupt by turning on bit 5 in the TIFR4 register. This will bring us to the ISR as shown in the snippet above, this time however our register variable values will be different:

```
Registers
 R00 = 0x00 R01 = 0x00 R02 = 0x00 R03 = 0x00 R04 = 0x00 R05 = 0x00 R06 = 0x00 R07 = 0x00 R08 = 0x00 R09 = 0x00 R10 = 0x00 R11 = 0x00 R12 = 0x00
   R13 = 0x00 R14 = 0x00 R15 = 0x00 R16 = 0xFE R17 = 0x00 R18 = 0x41 R19 = 0x00 R20 = 0x00 R21 = 0x00 R22 = 0x02 R23 = 0x00 R24 = 0x00 R25 = 0x20
    R26 = 0x00 R27 = 0x00 R28 = 0x00 R29 = 0x00 R30 = 0x00 R31 = 0x00

 CYCLE_COUNTER = 0x00005873 FP = 0x0000 PC = 0x00000082 SP = 0x21FC X = 0x0000 Y = 0x0000 Z = 0x0000
```

Now, IC4_FLAG (R22) is 0x02, CAPTURE_LOW (R18) is 0x41, and CAPTURE_HIGH (R19) is still 0x00. This means shows that I did not let the simulation run long and the high byte value was left at 0. That's okay. After the falling edge interrupt is received, control is returned the main program, the global interrupt bit is disabled, and the edge detection method is reset to rising edge as previously described.

Now we will quickly go through the logic to turn one of the lights on. I will step through this portion of the program until PORTA has an output written to it. Since our measured value is less than our magic number, the red light, or PORTL0 should turn on. To actually turn a pin on, we must write it low. This will allow current to flow from the 5V board supply through the pin to ground. A high bit turns off the pin. Our PORTA register looks like this:



This shows us that the PORTA0 pin is on and the red light would be illuminated as desired. If we were to let the timer run longer before simulating the interrupt, we would be able to see PORTA1 turn on simulating the green light being on.

Now that I have gone through the simulation, I was ready to load it to the hardware. When I finally got the opportunity to do this, I was feeling very confident everything was going to work well. Reality was quite the opposite. Everything worked but it only worked once. It was like the program wasn't looping through properly and instead it was most likely getting stuck on one of the inner loops waiting for the IC4_FLAG to be incremented. Without any real debug hardware, it was hard to tell. I tried doing a few different things but nothing was working, leaving me frustrated. The light correct light would turn when I was in front of the sensor, but it only worked once. I could reset the STK600 and the program would restart, and again the proper light would light up but only one time. The simulation worked correctly. The proper light was turning on. But it would only loop through once.

To solve this issue, I enlisted the help of another interrupt vector. When I first discovered interrupts in the ATmega2560 datasheet, I started with a little example using the Timer0 Overflow Interrupt. This allowed me to flash the LEDs at a given interval based on the clock of the interrupt. The clock would overflow, trigger the interrupt, and the LEDs would toggle. It wasn't much, but it was exciting getting my first interrupt to work. One night I was working on my project trying to solve my loop-only-runs-once issue and I remembered the timer overflow interrupt that allowed me to do stuff at given intervals. I went ahead and setup the Timer0 Overflow Interrupt again:

```
15   .org            0x002E                  ; Timer0 Overflow Interrupt Vector
16   jmp             ISR_TOV0                ;
```

Enabling and setting it up:

```
57   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
58   ; Set up Timer0 Overflow interrupt
59   ;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
60   ldi     TEMP, (1 << CS02) | (1 << CS00) ; Set clk to be sysclk / 1024.  We OR the individual bits together
61                                           ; to set each bit in the register.
62   out     TCCR0B, TEMP                    ;
63   ldi     TEMP, (1 << TOV0)               ;
64   out     TIFR0, TEMP                     ; clear pending interrupts flag
65   ldi     TEMP, (1 << TOIE1)              ;
66   sts     TIMSK0, TEMP                    ;
```

I also created another flag to hold a value that would enable the timer overflow interrupt only when we weren't looking for an input capture interrupt:

```
27   .def            TOV0_FLAG = r20         ; This will allow our sensor to take measurements at a specific interval
28                                           ; 0x00 ==> Wait for timer overflow then start measurement
29                                           ; 0x01 ==> We have received ICR4 interrupt, now wait for timer overflow
30                                           ;          and then clear this flag and restart the main loop
```

The ISR for the timer overflow interrupt was even simpler than my input capture ISR:

```
167   ISR_TOV0:
168       ; Timer0 Overflow Interrupt Service Routine
169       ; This will simply increment our TOV0_FLAG register to let the main loop proceed
170       inc     TOV0_FLAG                   ;
171       reti
```

All we do here is increment the TOV0_FLAG register. All logic is handled in the main loop of the program as shown here:

```
148       ; Disable global interrupts so we can clear our IC4 flag and enable the timer0 overflow interrupt again
149       cli
150       MAIN_LOOP_RETURN:
151       clr     IC4_FLAG                    ; restart the IC4 flag register at 0
152       rcall   ENABLE_TOV0                 ;
153
154       ; Reenable the global interrupts yet again, and wait for our timer0 overflow interrupt.
155       ; These delays will give our circuit time to settle and prevent the relays from seizing up.
156       ; A common problem of mechanical relays is they can't switch very fast, so if our circuit is constantly
157       ; trying to switch these, there is a very high probability that they will become stuck on.
158       sei
159       OVF2:
160       cpi     TOV0_FLAG, 0x01             ; is the TOV0 flag still 1?
161       breq    OVF2                        ; If so, keep waiting for our timer0 overflow interrupt
162       clr     TOV0_FLAG                   ; Otherwise, restart the timer0 overflow flag
163       cli                                 ; Disable global interrupts to restart the main loop
164       rjmp    MAIN_LOOP                   ;
```

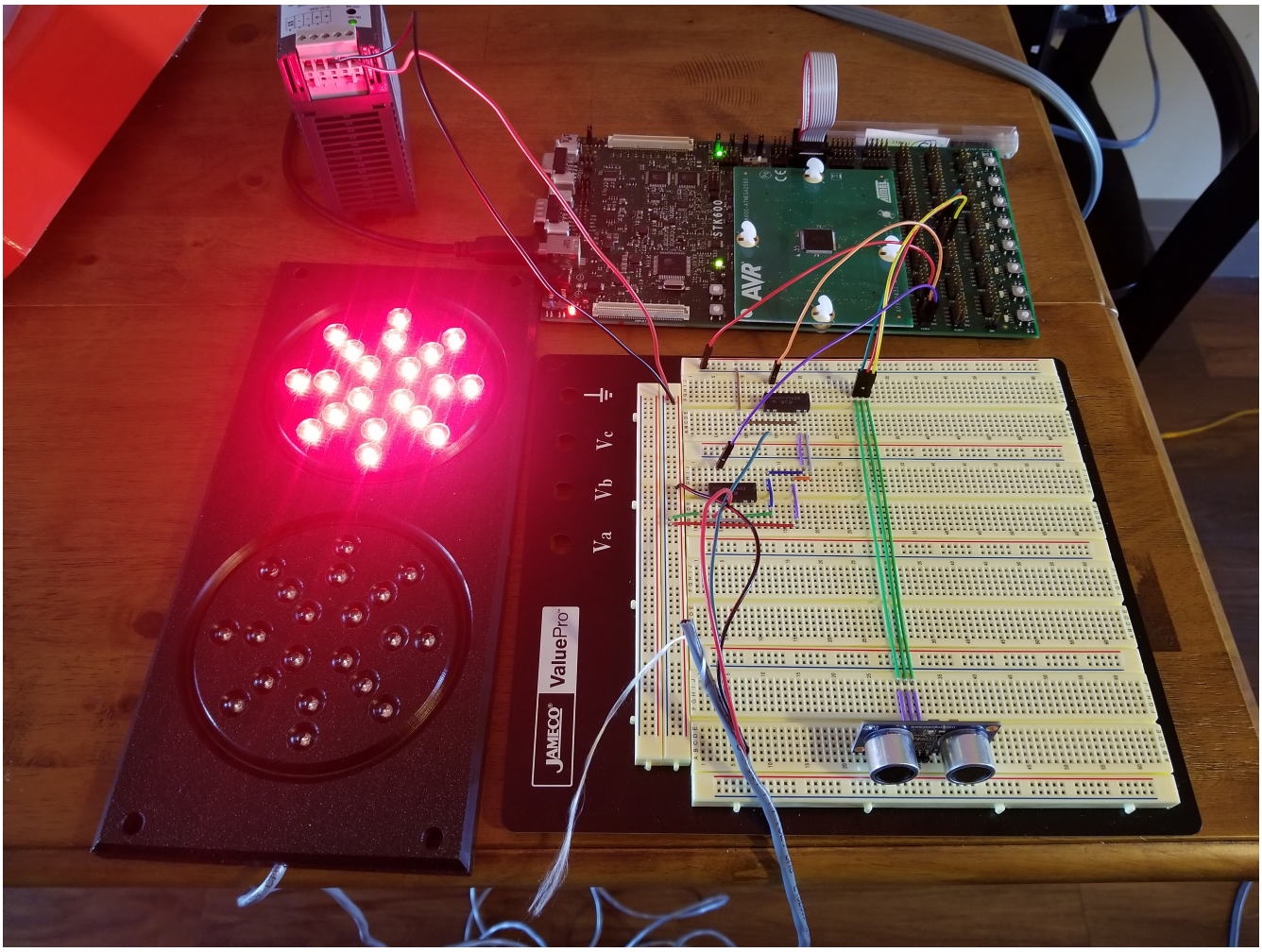
Notice this is similar to how we waited for the input capture interrupt. We simply loop until the TOV0_FLAG value changes like we would expect. I used the SEI/CLI instructions very frequently to make sure I was only receiving the interrupts I was expecting.

I again simulated this and everything appeared to work well. It did take a lot longer to simulate though and there was more to be aware of so the solution definitely got more complicated and still showed the same results. This made me nervous but I went ahead and uploaded it to the real hardware. Holding my breath, I tested it. Sure enough the red light turned on when I held my hand close. I removed my hand from the line of the sensor and… the red light turned off and the green light turned on. How exciting. My program worked and the circuit acted as desired, letting me know whether or not I was able to pull my car forward or whether I was far enough already.

In conclusion, this was a very fun and challenging project that proved to be more than I was expecting. I think the challenge of it really helped me learn a lot about the ATmega2560 specifically but also about some general concepts such as interrupt vectors and interrupt service routines. One thing I would have changed if I were to restart right now, I would have used a different distance sensor. The 3-pin sensor I believe made this more difficult than I was expecting. After looking around the internet some, it turns out there are indeed 4-pin ultrasonic sensors. Starting over I would have chosen one of these so I could have avoided switching the ATmega pin.

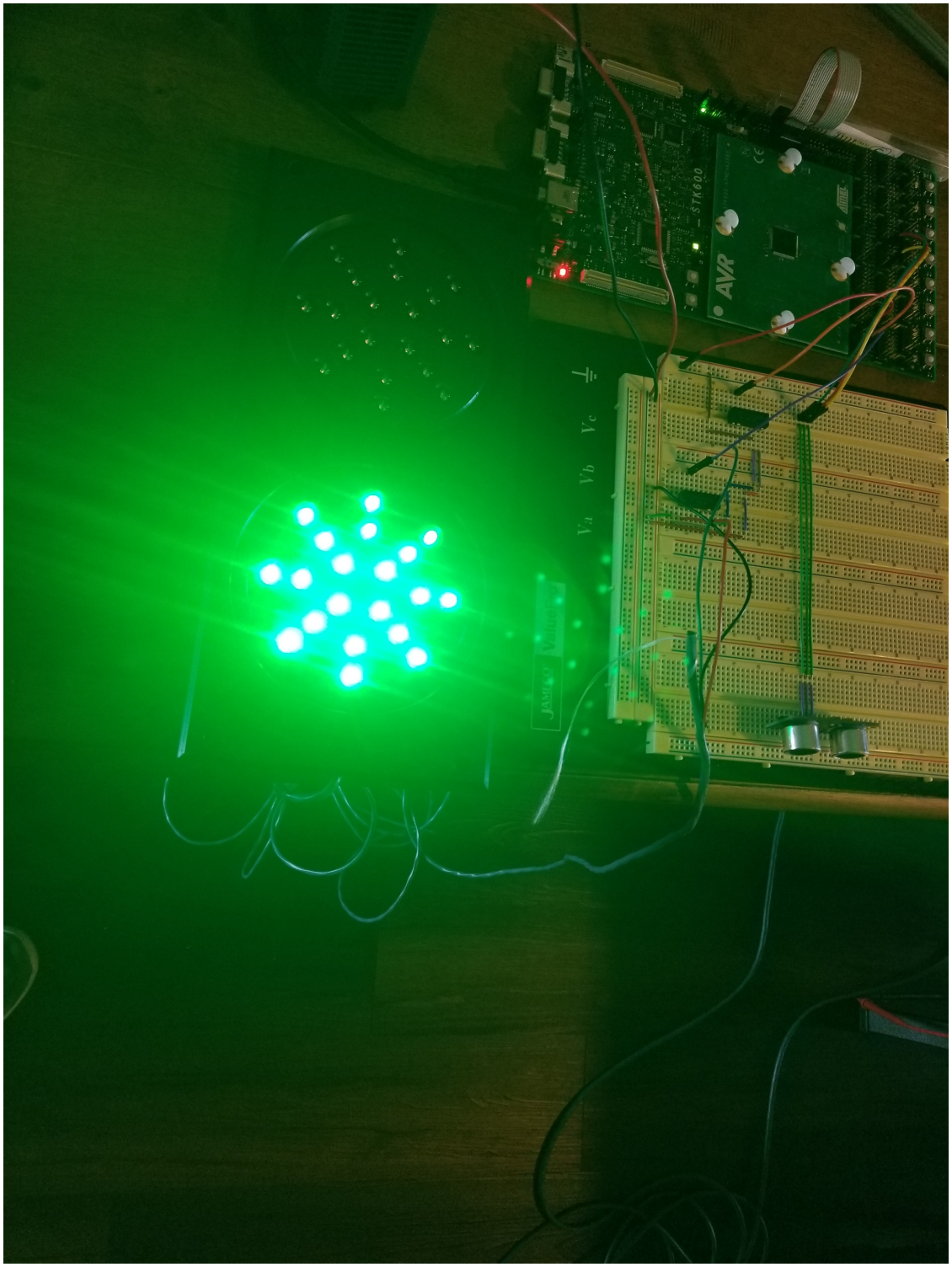
EECSX497.2 – Final Project
Matt Logan

Red Light (standing in front of sensor):



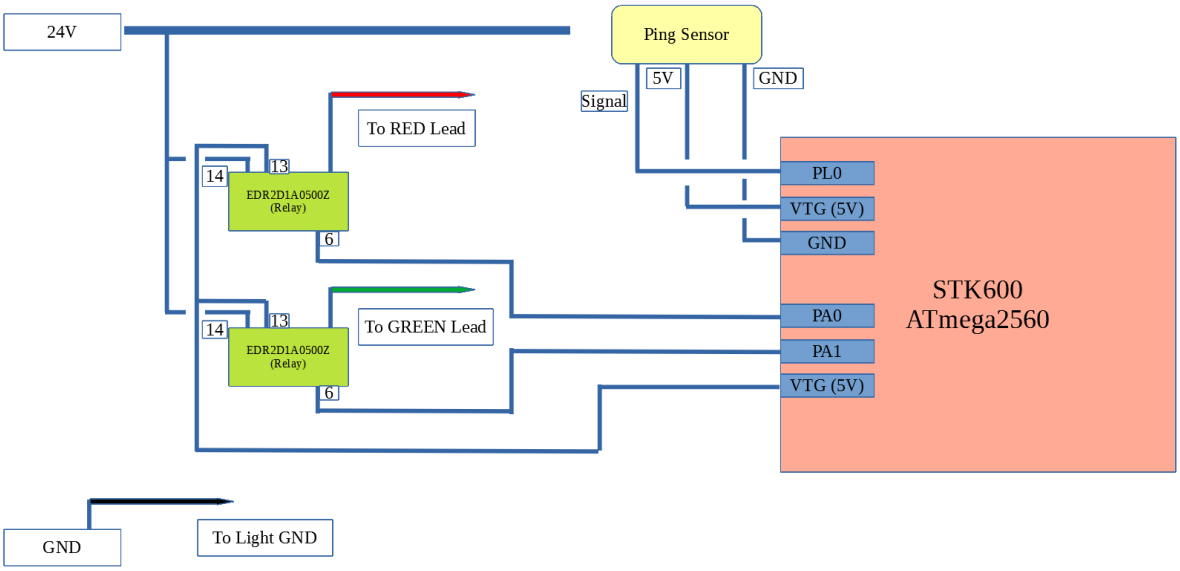
EECSX497.2 – Final Project
Matt Logan

Green Light (standing to side of sensor):



EECSX497.2 – Final Project
Matt Logan

Circuit Diagram



References

1. Detecting Distance with the Ping))) Ultrasonic Sensor
https://www.parallax.com/sites/default/files/downloads/28015-PING-Detect-Distance.pdf

2. AVR Assembler Instruction Set
http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

3. ATmega2560 Datasheet
https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf