ELEC 327 Radar Gun Final Report

Report submitted: May 6th, 2025

Aden Briano, Brandon Stroud, Nick Hu, Marc Alaniz

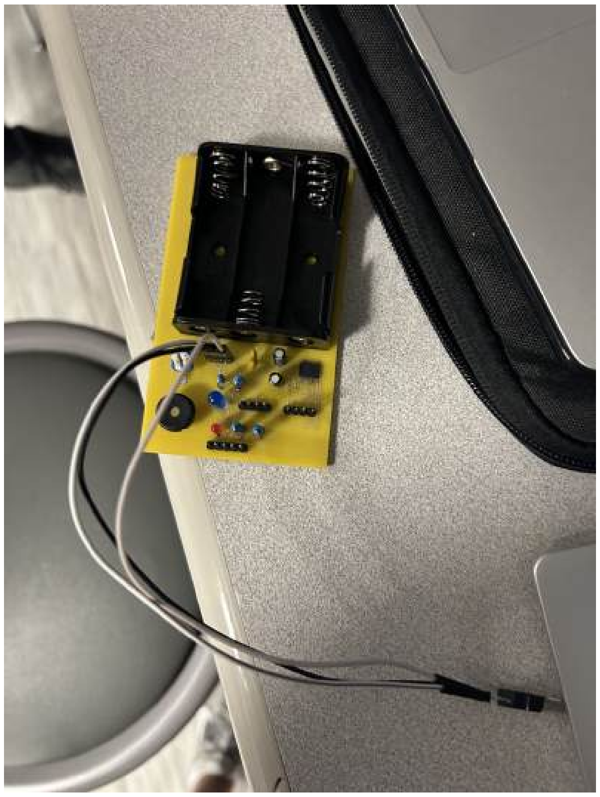
1 Introduction

For our final project, we wanted to design something that was both interesting and practical, so we decided to create a radar gun. We have always been fascinated by how the device worked, and thought it could be useful for our Beer Bike teams for them to measure the speed of the bikers. Since radar is expensive, we designed the device to use an ultrasonic sensor to measure the speed of the object, and a buzzer plus LEDs to indicate when it is above a certain speed threshold. Finally, for the device, we decided to use the components in the next section.

2 Project Design

Bill of Materials:

- Custom PCB



- HC-SR04 Ultrasonic Sensor (2 cm–450 cm)

- LD1117V33 Voltage Regulator

- 10 $k\Omega$ Potentiometer

- MSPS003F3 Microcontroller

- TDK PS1440PB02BT Through-hole Buzzer

- Multicolor LEDs

- Custom 3D Printed Case

2.1 Special Ultrasonic Sensor Used

The special sensor we decided to use for our project was the ultrasonic sensor. This works by sending a pulse of ultrasonic waves out and recording the time it takes for the waves to get back to the device. Because the waves travel at the speed of sound, we can calculate distance using this sensor. If we take the distance at two instances and divide by time, we calculate speed. Initially, we considered implementing two IR sensors a known distance apart, as they would measure the time it took to travel that distance, making both the concept and code implementation simpler. However, in the end, we decided to use the ultrasonic sensor as it allowed for a more flexible measurement of the object's speed rather than needing the object to move in a straight line. Additionally, it allowed us to measure the object at a further distance and with greater accuracy.

2.2 Design Process

2.2.1 Conceptual Design

To start off, designing and making a radar gun using only the supplies above caused us to iterate the design of the device several times. One of our first designs of the circuit can be seen in **Figure 1** below:

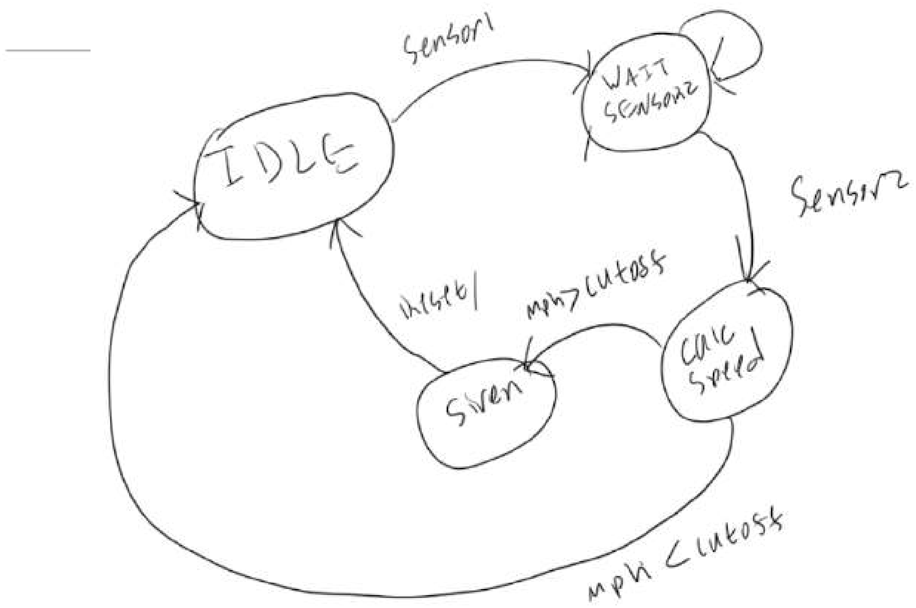


Figure 1: Conceptual Design of the Radar Gun FSM

In the diagram, we had four states (Idle, Wait For Sensor, Calculate Speed, and Siren). Note that our original conceptual design used the IR sensors; our next state machine design utilizes the simpler ultrasonic sensor. Here is a breakdown of each phase:

- **Idle:** The system remains quiet with no active outputs until the first object is detected (IR Sensor$_1$).
- **Wait for Sensor:** Upon activation of the IR Sensor$_1$, the system records the initial timestamp t_1 and transitions to this state, awaiting the next object detection (IR Sensor$_2$).
- **Calculate Speed:** When IR Sensor$_2$ triggers, the system captures the second timestamp t_2, computes the elapsed time, and calculates the vehicle speed

  $$v = \frac{d}{\Delta t},$$

  where d is the known sensor spacing. The result is then converted to meters per second.
- **Decision and Siren:** The calculated speed v is compared against the predefined threshold v_{cutoff}. If

  $$v > v_{\text{cutoff}},$$

  the FSM enters the *Siren* state, activating the buzzer and LED. After, it resets directly back to *Idle.*

2.2.2 Major Mistake

Despite working on the PCB together, our group forgot to add programming headers to our Speed Gun PCB. This caused us to have two options: either take three of the thinnest wires and solder them directly onto the pins we need access to, or reorder the PCBs and lose valuable time waiting for them to arrive. We spent a whole afternoon attempting the first option, but were not able to use this solution. Therefore, we decided to wait for the reordered PCB. In the meantime, we decided to finish the code of our project, but we couldn't be sure it would work as we couldn't even test without the PCB. The PCB was able to arrive in time, but they arrived on the last day of class, so we really had to rush to get our project done.

2.2.3 Final Design

For our final design, we simplified the FSM to only three stages of **Siren Off, Calculations, and Siren On**:

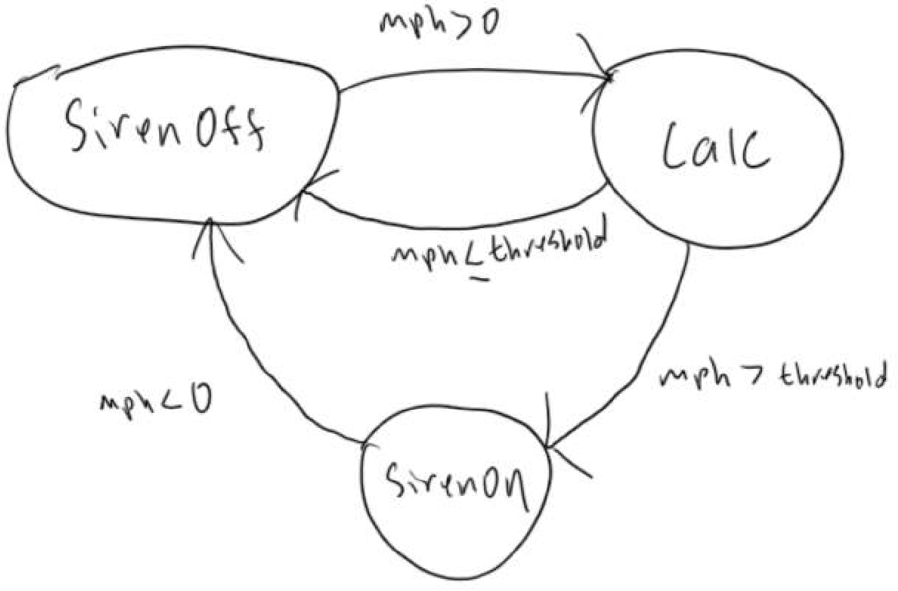


Figure 2: Final Design of the Radar Gun Circuit

Here's a breakdown of each stage:

- **Calculate (Calc):**
  - Compute v from the distance taken at each instance and the time elapsed between the instances.
  - If $v > v_{\text{th}}$, transition to `Siren On`.
  - Otherwise (i.e. $0 \leq v \leq v_{\text{th}}$), transition to `Siren Off`.
- **Siren On:**
  - Activate visual/auditory alerts.
  - As soon as $v < v_{th}$ (object speed is lower than threshold speed), transition to `Siren Off`.
- **Siren Off:**
  - Deactivate any alarms.
  - When $v > 0$ (new object detected), return to `Calculate`.

2.2.4 Final Hardware

We developed our PCB, used a breadboard for a voltage divider, and stuck it all inside a casing for ease of use.

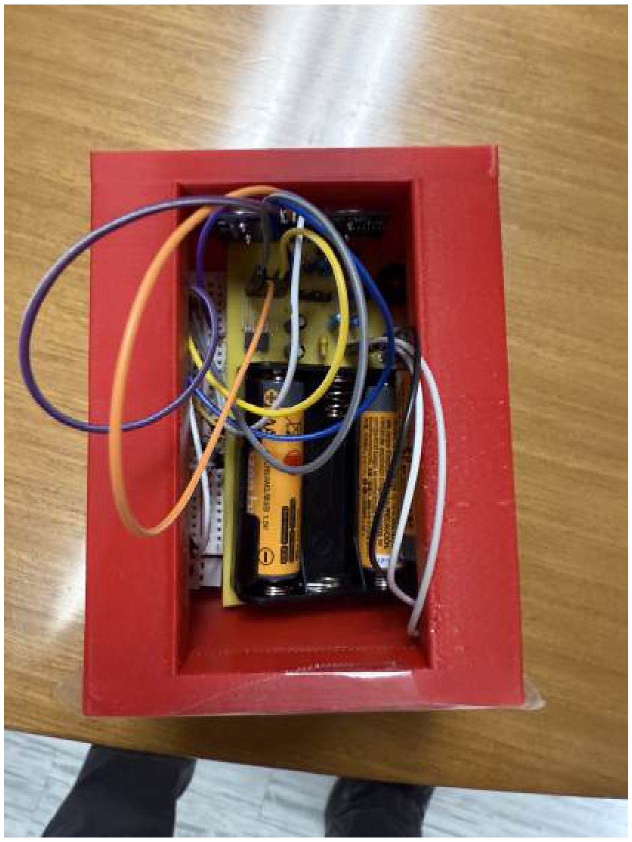


Figure 3: User view of radar-gun during use

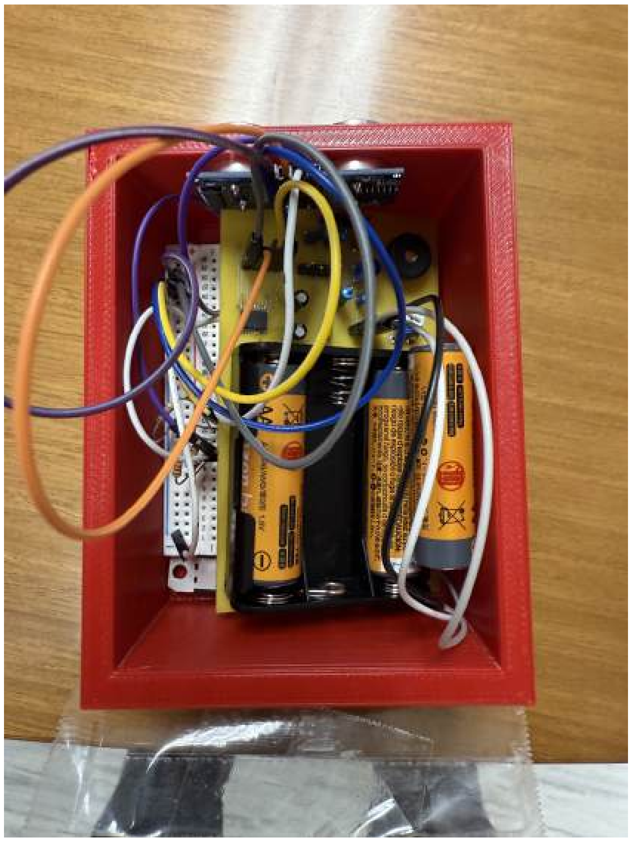


Figure 4: Internal view of radar-gun

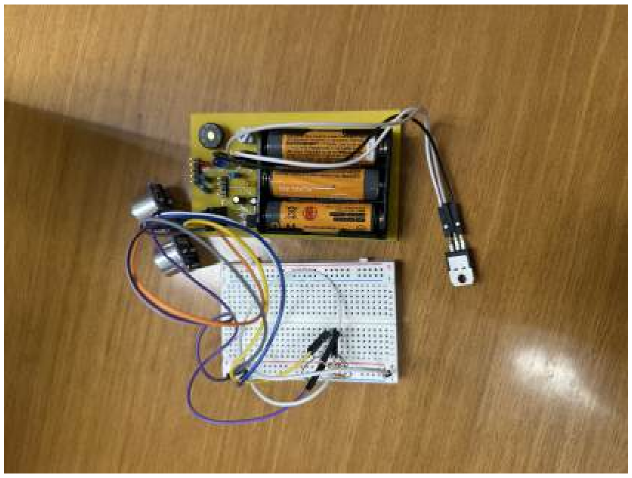


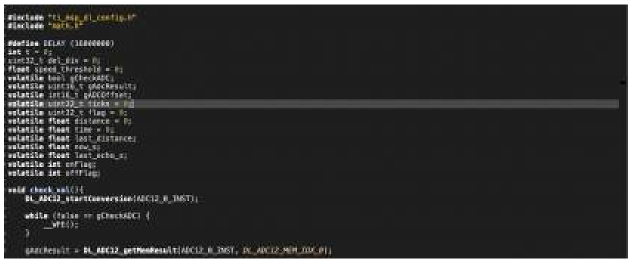
Figure 5: Circuit used for radar-gun

As seen in the circuit above, the PCB is connected to the ultrasonic sensor and a breadboard. The breadboard is used to convert the 5 V ECHO from the ultrasonic sensor into 3.3 V so we don't overpower our MSPS003F3. Everything else is housed on the PCB, except for the voltage regulator on the right that is used to convert 5 V to 3.3 V for the IC's power.

3 Code of Project

https://github.com/BrianoAden/RadarGun/tree/main

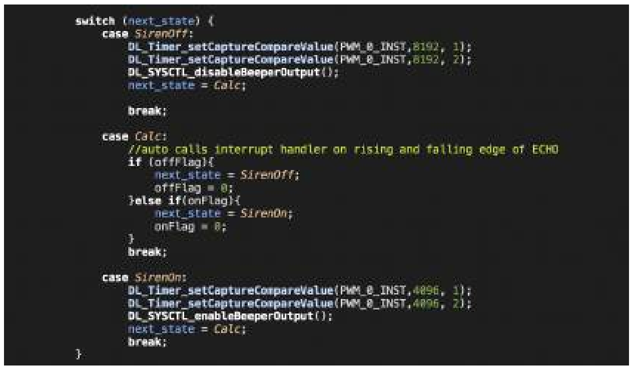
Our program consists of a state machine with 3 states to control the flow of our device. We utilized numerous includes and variables, seen below. Additionally, we need the potentiometer help function to read the value from the potentiometer and convert it to a speed threshold.



```c
#include "ti_msp_dl_config.h"
#include "math.h"

#define DELAY (16000000)
int t = 0;
uint32_t del_div = 0;
float speed_threshold = 0;
volatile bool gCheckADC;
volatile uint16_t gAdcResult;
volatile int16_t gADCOffset;
volatile uint32_t ticks = 0;
volatile uint32_t flag = 0;
volatile float distance = 0;
volatile float time = 0;
volatile float last_distance;
volatile float now_s;
volatile float last_echo_s;
volatile int onFlag;
volatile int offFlag;

void check_sol(){
    DL_ADC12_startConversion(ADC12_0_INST);

    while (false == gCheckADC) {
        __WFE();
    }

    gAdcResult = DL_ADC12_getMemResult(ADC12_0_INST, DL_ADC12_MEM_IDX_0);
```

Figure 6: Includes and variables used for our program, and potentiometer helper function

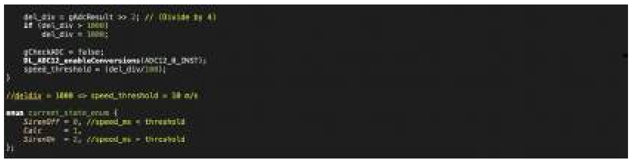
We utilized a state machine, and as seen below, SirenOff turns off the siren and buzzer, Calc handles whether we need to be in SirenOn or SirenOff, and SirenOn turns on our siren and buzzer.



```c
    switch (next_state) {
        case SirenOff:
            DL_Timer_setCaptureCompareValue(PWM_0_INST,8192, 1);
            DL_Timer_setCaptureCompareValue(PWM_0_INST,8192, 2);
            DL_SYSCTL_disableBeeperOutput();
            next_state = Calc;

            break;

        case Calc:
            //auto calls interrupt handler on rising and falling edge of ECHO
            if (offFlag){
                next_state = SirenOff;
                offFlag = 0;
            }else if(onFlag){
                next_state = SirenOn;
                onFlag = 0;
            }
            break;

        case SirenOn:
            DL_Timer_setCaptureCompareValue(PWM_0_INST,4096, 1);
            DL_Timer_setCaptureCompareValue(PWM_0_INST,4096, 2);
            DL_SYSCTL_enableBeeperOutput();
            next_state = Calc;
            break;
    }
```

Figure 7: State machine used

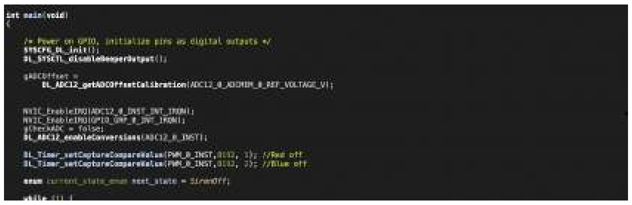
We used an enum to control which state we are in.



```c
    del_div = gAdcResult >> 2; // (Divide by 4)
    if (del_div > 1000)
        del_div = 1000;

    gCheckADC = false;
    DL_ADC12_enableConversions(ADC12_0_INST);
    speed_threshold = (del_div/100);
}

//deldiv = 1000 => speed_threshold = 10 m/s

enum current_state_enum {
    SirenOff = 0, //speed_ms < threshold
    Calc     = 1,
    SirenOn  = 2, //speed_ms > threshold
};
```

Figure 8: Enum used

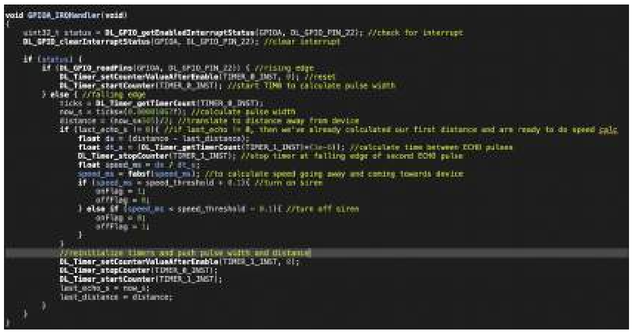
In our main function, we power on our GPIO and initialize pins. We enable the IRQs and begin our timers. Additionally, in our main function, we utilize the state machine described above.



```
int main(void)
{

    /* Power on GPIO, initialize pins as digital outputs */
    SYSCFG_DL_init();
    DL_SYSCTL_disableBeeperOutput();

    gADCOffset =
        DL_ADC12_getADCOffsetCalibration(ADC12_0_ADCMEM_0_REF_VOLTAGE_V);


    NVIC_EnableIRQ(ADC12_0_INST_INT_IRQN);
    NVIC_EnableIRQ(GPIO_GRP_0_INT_IRQN);
    gCheckADC = false;
    DL_ADC12_enableConversions(ADC12_0_INST);

    DL_Timer_setCaptureCompareValue(PWM_0_INST,0[illegible], 1); //Red off
    DL_Timer_setCaptureCompareValue(PWM_0_INST,0[illegible], 3); //Blue off

    enum current_state_enum next_state = SirenOff;

    while (1) {
```

Figure 9: Main function

In our interrupt handler, we check for an interrupt and clear it. We handle rising and falling edges separately. For rising edges, we reset and start TIM0 to calculate the pulse width. For falling edges, we receive the count of TIM0 and calculate the width of the pulse sent by the ultrasonic sensor. We translate to distance, and if we have calculated a previous distance, we go through our calculations to find speed. If speed is higher than our threshold, we turn on the siren, and if not, we turn off the siren. At the end, we reinitialize our timers and change our pulse-width and distance to be previous variables, so that we can calculate new ones using the device.



```
void GPIOA_IRQHandler(void)
{
    uint32_t status = DL_GPIO_getEnabledInterruptStatus(GPIOA, DL_GPIO_PIN_22); //check for interrupt
    DL_GPIO_clearInterruptStatus(GPIOA, DL_GPIO_PIN_22); //clear interrupt

    if (status) {
        if (DL_GPIO_readPins(GPIOA, DL_GPIO_PIN_22)) { //rising edge
            DL_Timer_setCounterValueAfterEnable(TIMER_0_INST, 0); //reset
            DL_Timer_startCounter(TIMER_0_INST); //start TIM0 to calculate pulse width
        } else { //falling edge
            ticks = DL_Timer_getTimerCount(TIMER_0_INST);
            now_s = ticks*([illegible]f); //calculate pulse width
            distance = (now_s*[illegible])/2; //translate to distance away from device
            if (last_echo_s != 0){ //if last_echo != 0, then we've already calculated our first distance and are ready to do speed calc
                float ds = (distance - last_distance);
                float dt_s = (DL_Timer_getTimerCount(TIMER_1_INST)*([illegible])); //calculate time between ECHO pulses
                DL_Timer_stopCounter(TIMER_1_INST); //stop timer at falling edge of second ECHO pulse
                float speed_ms = ds / dt_s;
                speed_ms = fabsf(speed_ms); //to calculate speed going away and coming towards device
                if (speed_ms > speed_threshold + 0.1){ //turn on siren
                    onFlag = 1;
                    offFlag = 0;
                } else if (speed_ms < speed_threshold - 0.1){ //turn off siren
                    onFlag = 0;
                    offFlag = 1;
                }
            }
            //reinitialize timers and push pulse width and distance
            DL_Timer_setCounterValueAfterEnable(TIMER_1_INST, 0);
            DL_Timer_stopCounter(TIMER_0_INST);
            DL_Timer_startCounter(TIMER_1_INST);
            last_echo_s = now_s;
            last_distance = distance;
        }
    }
}
```

Figure 10: Interrupt Handler

When programming our microcontroller, we took power into consideration by implementing an interrupt-driven design. We used Wait-For-Event (WFE) to make the device sleep until an interrupt occurs. Additionally, we utilized PWM for LEDs so that the CPU doesn't spend cycles toggling GPIO pins, and utilized Timer Capture to avoid software loops. We also used an enum-state machine to limit branching and wasted cycles, and optimized performance by keeping the CPU in sleep most of the time.

4 Conclusion

All in all, our final system worked really well with the ultrasonic sensor calculating the speed of the object with high accuracy. With further time, we could see future improvements such as adding a wireless aspect through adding Bluetooth or Wi-Fi modules or using a better/more expensive sensor so that it could be even more consistent and accurate. Through this project, we were able to learn a lot of information from all aspects of electrical engineering, from custom designing our PCB board to programming the necessary FSM for the project to function. This project was also great for us to see the real-world application of all that we have learned in class and really put our knowledge to the test.