Lab 12

Implementing a Range Finder and Alarm

Due: Week of May 6, before the start of your lab section*

This is a team-effort project. You may discuss concepts and syntax with other students, but you may discuss solutions only with your assigned partner(s), the professor, and the TAs. Sharing code with or copying code from a student who is not on your team, or from the internet, is prohibited.

In this assignment, you will write code for your Cow Pi that will use new electronic devices to interact with the physical world.

Contents

*See Piazza for the due dates of teams with students from different lab sections.

Learning Objectives

After successful completion of this assignment, students will be able to:

- Work collaboratively on a hardware/software project
- Design and implement a simple embedded system
- Expand their programming knowledge by consulting documentation

During Lab Time

During your lab period, coordinate with your group partner(s) to decide on your working arrangements. Unless you're only going to work on the assignment when you're together, you may want to set up a private Git repository that is shared with your partner(s). With your partner(s), add the new hardware as described in Appendices B–C. Then, think through your system's design and begin implementing it. The TAs will be available for questions.

No Spaghetti Code Allowed

In the interest of keeping your code readable, you may *not* use any **`goto`** statements, nor may you use any **`continue`** statements, nor may you use any **`break`** statements to exit from a loop, nor may you have any functions **`return`** from within a loop.

Scenario

"I have various teams working on different projects around here to improve security," Archie reminds you. He glances toward the Zoo's labs, where there's now a guy who looks like

the actor who portrayed the fictional actor who portrayed the Norse god Odin, trying to avoid children while wistfully talking about raising rabbits in Montana. You briefly wonder why there are children someplace where there are also carnivorous megafauna, and then you remember that you work at a petting zoo. "What I need your team to do," Archie continues, "is make a range finder that will alert us when someone – or some*thing* – gets too close to someplace they shouldn't be."

1 Assignment Summary

Please familiarize yourself with the entire assignment before beginning. There are multiple parts to this assignment.

If one of the partners does not make an equitable contribution, we will adjust both partners' scores accordingly.

The assignment is written so that if you and your partner decide to pair program, you can complete Sections 4–7 in sequence. The assignment is also written so that you and your partner can add the new hardware and work on Section 4 together, then split up with one student working on Section 5 and the other working on Section 6, and then get back together again to work on Section 7.

If you are in the unfortunate position of having a partner who does not contribute to the project, we do not expect you to complete the full assignment by yourself and will take the circumstances into account when grading. In this situation, you should prioritize Section 5 over Section 6, as you will be able to complete more of Section 7 with a working distance sensor but no alarm, than the other way around.

1.1 Constraints

You may use the constants and functions provided in the starter code. You may use any features that are part of the C standard if they are supported by the compiler. You may use code written by you and/or your partner.

1.1.1 Constraints on the Arduino core

You may use **`digitalWrite()`**[1] to write to the `TRIGGER` and `BUZZER` pins (or you may use memory-mapped I/O).

You may not use any other libraries, functions, macros, types, or constants from the Arduino core. This prohibition includes, but is not limited to, the **`micros()`**, **`tone()`**, **`noTone()`**, **`pulseIn()`**, and **`pulseInLong()`** functions.

1.1.2 Constraints on AVR-libc and on MBED OS

You may not use any AVR-specific or MBED-specific functions, macros, types, or constants.

[1] https://www.arduino.cc/reference/en/language/functions/digital-io/digitalwrite/

1.1.3 Constraints on the CowPi library

You may use any functions provided by the CowPi[2] and the CowPi_stdio[3] libraries, and you may use any data structures[4] provided by the CowPi library.

1.1.4 Constraints on other libraries

You may not use any libraries beyond those explicitly identified here.

2 Getting Started

Download the zip file or tarball from Canvas or ~cbohn2/csce231 on *cse-linux-01.unl.edu*. Once downloaded, unpackage the file and open the project in your IDE.

2.1 Description of RangeFinder Files

2.1.1 RangeFinder.cpp, interrupt_support.h, interrupt_support.cpp, outputs.h, outputs.cpp

Do not edit *RangeFinder.cpp*, *interrupt_support.h*, *interrupt_support.cpp*, *outputs.h*, or *outputs.cpp*.

These files contain code to simplify interrupt management and to Functions to place text on the display module.

2.1.2 user_controls.h & user_controls.c

Do not edit *user_controls.h*

The *user_controls.c* file is where you will get inputs from the user.

2.1.3 sensor.h & sensor.c

Do not edit *sensor.h*

The *sensor.c* file is where you will control the ultrasonic echo sensor and compute any detected objects' distance and speed.

2.1.4 alarm.h & alarm.c

Do not edit *alarm.h*

The *alarm.c* file is where you will control the piezoelectric disc and manage the chirping and strobing of a proximity alarm.

[2] https://cow-pi.readthedocs.io/en/latest/library.html
[3] https://cow-pi.readthedocs.io/en/latest/stdio.html
[4] https://cow-pi.readthedocs.io/en/latest/microcontroller.html

2.1.5 shared_variables.h

The *shared_variables.h* header file is where you will place any types that you define and where you will externalize any global variables that need to be used by more than one *.c* file.

It also contains a structure that you may use to access an analog-digital converter's registers, and it contains meaningfully-named constants to refer to specific pins you will use in this assignment.

2.2 Combining Code for Old and New Hardware

If you and your partner are both using the old hardware, you do not need to worry about combining code for old and new hardware. Similarly, if you and your partner are both using the new hardware, you do not need to worry about combining code for old and new hardware.

If, however, one partner is using the old hardware and the other is using the new hardware, then you will need to include compiler preprocessor directives to detect which microcontroller the program will run on, and execute the correct code for that microcontroller. These differences primarily will be configuring the timers and calculating distances.

C's preprocessor accepts directives that modify a temporary copy of a file before compiling it. Two that you've already seen are **#include** and **#define**. The **#if** directive and its relatives can be used to include/exclude code based on compile-time conditions. If one partner is using the old hardware and the other is using the new hardware, use the pattern

```
1      // code for both old and new hardware goes here
2  #if defined (ARDUINO_AVR_NANO)
3      // code for the old hardware goes here
4  #elif defined (ARDUINO_RASPBERRY_PI_PICO)
5      // code for the new hardware goes here
6  #endif
7      // code for both old and new hardware goes here
```

A concise example can be found in **refresh_display()** in *outputs.cpp*:

```
108 void refresh_display(void) {
109     if (display_needs_refreshed) {
110 #if defined (ARDUINO_AVR_NANO)
111         fprintf(display, "\f%s\n%s\f", rows[0], rows[1]);
112 #elif defined (ARDUINO_RASPBERRY_PI_PICO)
113         obdDumpBuffer(&display, backbuffer);
114 #endif
115         display_needs_refreshed = false;
116     }
117 }
```

2.3 Assemble the Hardware and Edit *platformio.ini*

BEFORE YOU PROCEED FURTHER:

( ) Edit *platformio.ini* so that the section for your hardware (old/new hardware) is uncommented, and the other section (old/new hardware) is commented-out.

( ) Add the new hardware to your Cow Pi as described in Appendices B–C.

3 System Specification

1. Definitions

   Threshold Range The distance at which a detected object will cause the system to produce an audible alarm

   Strobe An illumination of the right LED for 50ms

   Chirp A 5kHz tone lasting 50ms

2. The slide-switches shall control the mode of operation.

   (a) Both switches in the *left position*: Normal Operation

   (b) The **left switch** in the *right position* and the **right switch** in the *left position*: Single-Pulse Operation

   (c) Both switches in the *right position*: Threshold Adjustment

   (d) The **left switch** in the *left position* and the **right switch** in the *right position*: Continuous Tone

3. When the range finder and alarm system is in **Continuous Tone** mode:

   (a) The system shall not detect the range of nearby objects: it shall neither emit nor detect ultrasound

   (b) The system shall not illuminate the right LED

   (c) The system shall produce a continuous 5kHz audible tone

4. When the range finder and alarm system is in **Threshold Adjustment** mode:

   (a) The system shall not detect the range of nearby objects: it shall neither emit nor detect ultrasound

   (b) The system shall not produce an audible tone

   (c) The system shall not illuminate the right LED

   (d) The system shall prompt the user on the **display module** to enter the threshold range, in centimeters

- You may assume that the user understands what the system does: the prompt must be *meaningful*, but it may be *succinct*

(e) The user shall be able to enter the threshold range, in centimeters, using the numeric keypad

i. The user will enter the range in decimal

ii. The system shall echo the user's input, digit-by-digit, on the **display module** as they type their input

iii. The user will indicate that they have finished entering their input by pressing the '#' key

(f) After the user has completed their input, then any value less than 50cm or greater than 400cm shall be rejected as an invalid threshold; the system shall display a helpful error message on the **display module** and re-prompt the user to enter the threshold range

(g) After the user has completed their input, and if the input is valid, then the system shall display "Threshold (value)cm" on the **display module**, where "(value)" shall be the numeric threshold range in centimeters; this shall be displayed until the user takes the system out of the Threshold Adjustment mode

5. When the range finder and alarm system is in **Single-Pulse Operation** mode:

(a) The system shall neither emit nor receive ultrasound, shall not produce an audible tone, and shall not illuminate an LED, *until* the pushbutton has been pressed

(b) Whenever the user presses the **left pushbutton**, the system shall emit one ultrasonic pulse

(c) If the system does not receive that pulse's echo, the system shall resume waiting for the user to press the pushbutton

(d) If the system does receive that pulse's echo:

i. The system shall strobe the **right LED** once

ii. If the object's distance is less than the threshold range then the system shall emit one chirp

iii. The system shall then resume waiting for the user to press the pushbutton

6. When the range finder and alarm system is in **Normal Operation** mode, the system shall repeatedly:

(a) Emit an ultrasonic pulse and take no further action until either the system receives an echo or the system can establish that it will not receive an echo

(b) If the system can establish that it will not receive an echo:

i. The system shall stop strobing the right LED and chirping if it previously was doing so

distance	LED on / Piezo sounding	LED off / Piezo silent	total period
$distance \geq 250cm$	$50ms$	$2450ms$	$2500ms$
$200cm \leq distance < 250cm$	$50ms$	$1950ms$	$2000ms$
$150cm \leq distance < 200cm$	$50ms$	$1450ms$	$1500ms$
$100cm \leq distance < 150cm$	$50ms$	$950ms$	$1000ms$
$50cm \leq distance < 100cm$	$50ms$	$700ms$	$750ms$
$25cm \leq distance < 50cm$	$50ms$	$450ms$	$500ms$
$10cm \leq distance < 25cm$	$50ms$	$200ms$	$250ms$
$distance < 10cm$	$50ms$	$75ms$	$125ms$

Table 1: Strobe and Chirp periods for various distances to an object

ii. The system shall repeat the action in Requirement 6a
- Repeating the action may be postponed until after necessary quiescent periods

(c) If the system receives an echo:

i. The system shall compute and display the object's distance, in centimeters

ii. The system shall compute and display the object's rate of approach, in centimeters per second

iii. The system shall strobe the **right LED** at the rate described by Table 1

iv. If the object's distance is less than the threshold range, the system shall emit chirps at the rate described by Table 1

v. After the system has completed the speed and distance calculations, it shall repeat the action in Requirement 6a
- Repeating the action may be postponed until after necessary quiescent periods

7. All mechanical inputs shall be properly debounced

8. The system shall always be responsive to user input

- The user will never press two keys on the numeric keypad at the same time
- The system may ignore changes to the slide-switches while the user enters a new threshold range
- The system may block while displaying an error message
- But for those exceptions, there shall be no noticeable lag when responding to an input

4 Initial Software Changes

4.1 Examining the Starter Code

user_controls.c The **`initialize_controls()`** function is where you'll place any alarm-related code that needs to be run once when the program starts. The **`manage_controls()`** function is where you'll place any alarm-related code that needs to run with every iteration of the program's main loop. You may, of course, add helper functions.

shared_variables.h This is the only header file you will turn in, so if you need to share any **`enum`**s, **`struct`**s, or variables between *.c* files, place them in here and not in the other header files, so that we can compile your code. When you need to share a variable between *.c* files, declare it in exactly one *.c* file (preferably the one that the variable best coheres to) without the **`extern`** keyword, and then externalize it by declaring it again with the **`extern`** keyword in *shared_variables.h*. This approach will create just one global symbol for the variable in the program while making it "visible" to the code in the other *.c* files.

4.2 A Quick Note about Creating State Machines

A not-uncommon pattern in embedded systems is to design the system as a state machine, or as multiple state machines. The states are represented as the value of an enumerated type. A couple of key advantages of writing the system as a state machine or as a collection of state machines is that it's easy to change the state based on inputs, and it's easy for the parts of the system that take action to do so based on what state the state machine is in.

4.3 Read the Switches

Create a way to track which mode the system is in (Requirement 2). Be sure to declare it not only in *user_controls.c* (without the **`extern`** modifier) but also in *shared_variables.h* (with the **`extern`** modifier). Add code to **`manage_controls()`** to poll the positions of the slide-switches and set the system's mode accordingly.

4.4 Read the Pushbutton

As noted in Requirement 5, if the system is in Single-Pulse Operation mode and the user presses the left pushbutton, they want to emit exactly one ultrasound pulse to get the distance to an object.

( ) Create a variable to indicate that the user has requested a ping.

- Be sure to declare it not only in *user_controls.c* (without the **`extern`** modifier) but also in *shared_variables.h* (with the **`extern`** modifier).

( ) Add code to **`initialize_controls()`** to set this variable initially to `false`.

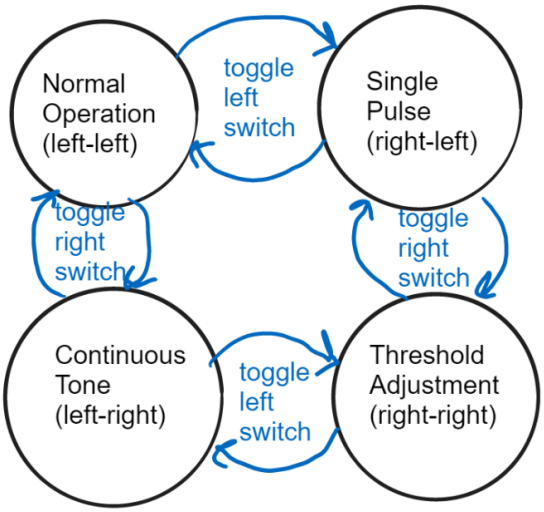


Figure 1: State machine describing the rangefinder's modes of operation.

Figure 2: Verify our range to target. One ping only.
Image by Paramount Pictures Corporation

( ) Add code to **`manage_controls()`** to set this variable to `true` when and only when the user presses the button while in Single-Pulse Operation mode.

- The variable should be set to `true` only *once* per press.
- Do *not* set it to `true` just because the user is still holding the button down.

Do not worry about setting this variable to `false` yet; that will come later.

You can now continue to work through the remainder of the lab with your partner, perhaps pair programming, or you can decide to have one partner work on Section 5 and the other work on Section 6. Regardless, you will need to work together on Section 7, as this is when you will integrate the work from Sections 5–6.

5 Measuring Distance

You will use the ultrasonic echo sensor to determine the distance to an object.

5.1 Theory of Operation

The ultrasonic echo sensor has a simple interface. If your program places a logic-high signal on the `TRIGGER` pin for 10μs and then drop it to logic-low, then the device will emit a short burst of ultrasound. The device will then raise the `ECHO` pin's logic level to high. If there is a nearby object, the ultrasound will reflect off of it, and the sensor will detect the echo. After detecting the echo, the device will drop the `ECHO` pin's logic level to low. If no echo is received, then the device will eventually time-out and drop the `ECHO` pin's logic level to low. After affording the device a quiescent period, the cycle can be repeated.

If your program measures the time between the `ECHO` pin's logic going high and subsequently going low, then it knows how long the ultrasound travelled to the object and back again. The classical relationship $distance = speed \times time$ requires that we know how fast the ultrasound travels.

The speed of sound depends on the medium it travels through and the temperature of that medium. We will assume the medium is air.

Old Hardware If you are using the old hardware, you may assume that the air temperature is 70° Fahrenheit (21.1° Celsius). The speed of sound for 70° air is $348.8\frac{m}{s}$.[5]

New Hardware If you are using the new hardware, you will treat the speed of sound as a function air temperature. Assuming the temperature T is measured in °C, the speed of sound in air is[6]

$$331.228 \times \sqrt{1 + \frac{T}{273.15}}\frac{m}{s}$$

Thus, by knowing the round-trip travel time, we can determine the distance that the ultrasonic pulse travelled. The distance to the object it echoed off of will be half of the round-trip distance.

5.2 The Wrong Calculation

The internet is littered with example code for measuring distance that instructs you to divide the ultrasonic pulse's round-trip travel time by 58 to obtain the distance in centimeters. Few of these explain the origin of that particular calculation; however, if you look carefully, you can find one that does.[7] Like approximating gravitational acceleration as $10\frac{m}{s^2}$, the calculation of "divide microseconds by 58" is simple enough for someone to easily use and provides an adequate approximation. Like approximating gravitational acceleration as

[5] https://www.weather.gov/epz/wxcalc_speedofsound
[6] https://www.weather.gov/media/epz/wxcalc/speedOfSound.pdf
[7] https://docs.arduino.cc/built-in-examples/sensors/Ping/

$10\frac{m}{s^2}$, the calculation of "divide microseconds by 58" provides you with the wrong answer if "approximately correct" is insufficient.

The errors stem from three sources:

1. The formulation states that the speed of sound is $340\frac{m}{s}$, but this is the case only when the air temperature is 14.66° Celsius (58.4° Fahrenheit). While there are situations in which that *will* be the air temperature, students using the new hardware will measure the temperature, and students using the old hardware will assume the temperature is the thermostat set-point for Avery Hall.

2. Even if the temperature were 14.66° Celsius, the formulation approximates the speed of sound as $\frac{1}{29}\frac{\mu s}{cm}$, but $\left(340\frac{m}{s}\right)^{-1} = \frac{1}{29.41}\frac{\mu s}{cm}$. This rounding error adds up quickly.

3. On the ATmega328P microcontroller (which most of the examples are targeting), the timer typically used to measure time is only accurate to within 4μs. Even if $\frac{1}{29}\frac{\mu s}{cm}$ were the correct expression, this will produce an error of about 1mm. If we were using floating point arithmetic, this could be dismissed as a rounding error, but we are using integer arithmetic. Because integer division truncates the fractional portion of the quotient, there will be times in which "round-towards-zero" produces an error of a full centimeter relative to the answer that a more-careful computation would have produced.

5.3 The Correct Calculation

5.3.1 Practical Considerations

Arithmetic As neither of our microcontrollers have a floating point unit (FPU), we want to avoid floating point calculations, which are computationally expensive when performed entirely in software. Further, the ATmega328P cannot perform division in hardware, and so we want to avoid integer division unless the divisor is a power of two when using the old hardware. The RP2040 has an integer divider that requires 8 processor clock cycles to perform division, so we might allow integer division with the new hardware – but we have an equation that will not require the integer divider.

Time After we configure a timer to manage the sensor, we will be able to use its counter to determine the round-trip travel time. On the old hardware, the "tick" will be a half-microsecond. On the new hardware, the "tick" will be one microsecond.

5.3.2 The Equations

These equations can be used to accurately compute the distance to an object. That is, the computed distance is half of the ultrasonic pulse's round-trip distance. The derivation of these equations can be found in Appendix D.

Old Hardware

$$distance = time_{\text{half}\mu s} \times \frac{18,025cm}{\text{half}\mu s} \div 2^{21}$$

Even though *distance* will be less than 500 cm, you will need at least 31 bits to represent the intermediate products when performing the arithmetic.

New Hardware

$$distance = time_{\mu s} \times (256,108,888 - 121,907 \times ADC\_register\_value) \frac{cm}{\mu s} \div 2^{33}$$

Even though *distance* will be less than 500 cm, you will need at least 44 bits to represent the intermediate products when performing the arithmetic.

Obtaining the *ADC_register_value* will be treated as extra credit. If you do not wish to pursue that extra credit, then you may hard-code *ADC_register_value* as 889, which is the value corresponding to 21.05°C (69.9°F).

5.4 Examining the Starter Code

The **`initialize_sensor()`** function is where you'll place any alarm-related code that needs to be run once when the program starts. The **`manage_sensor()`** function is where you'll place any alarm-related code that needs to run with every iteration of the program's main loop. You will, of course, add code outside these functions too: an interrupt service routine for a timer, an interrupt handler for the distance sensor, and possibly helper functions.

5.5 Single-Pulse Operation

( ) Create a variables to:

- indicate whether an object has been detected
- indicate the object's distance (if the object is detected)
- indicate the object's rate of approach (this will only be useful in Normal Operation mode)

Be sure to declare them not only in *sensor.c* (without the **`extern`** modifier) but also in *shared_variables.h* (with the **`extern`** modifier).

( ) In **`initialize_sensor()`**, initialize the object detection variable to indicate that an object has not been detected.

A fully-correct implementation of Single-Pulse Operation mode has the alarm chirp if an object is detected closer than the threshold range (Requirement 5). For now, your focus will be on determining the distance to an object.

( ) In *sensor.c*, create a temporarily-empty interrupt handler that will be used to respond to the sensor timer's interrupts. Don't forget to pre-declare this functions above **`initialize_sensor()`**.

( ) In **`initialize_sensor()`**, register that function as an ISR for a timer interrupt that has a period of 32,768μs

- On the old hardware, use TIMER1.
- On the old hardware, this should configure TIMER1 to have a "tick" of $\frac{1}{2}\mu s$ (which we will refer to as 1halfμs). On the new hardware, the timer's "tick" will be $1\mu s$

( ) Create a state machine for the sensor that can be "initial-start", "powering-up", "ready," "active-listening," "active-detected," and "quiescent."

( ) Initialize that state machine to be "initial-start".

If you are using the new hardware, you will need to be able to read from an analog-digital converter (ADC).

( ) Create a pointer to an `adc_t` structure and point it to `0x4004c000`.

( ) In **`initialize_sensor()`**, configure the ADC using its `control` register:

- Set bit20 to `1` indicating that we are interested in ADC channel 4.
- Set bits14..12 to `4` indicating that channel 4 is the next channel to convert.
- Set bits1..0 to `3` indicating that the temperature sensor should be powered, and that the ADC should be enabled.

( ) In *sensor.c*, create a temporarily-empty interrupt handler that will be used to detect the rising and falling edge of the pulse. Don't forget to pre-declare this functions above **`initialize_sensor()`**.

- **Do *NOT* use `manage_sensor()` as your interrupt handler!**

( ) Place in **`initialize_sensor()`** code to register the handler for a CHANGE on the `ECHO` pin.

Allow Time for the System to Initialize When the system starts up, the input pins and the timers may fire "phantom" interrupts. We will handle this taking no action until the sensor timer has fired at least two interrupts.

In the timer interrupt handler:

( ) If the sensor's state machine is in its "initial-start" state, place it in its "powering-up" state

( ) If the sensor's state machine is in its "powering-up" state, place it in its "ready" state

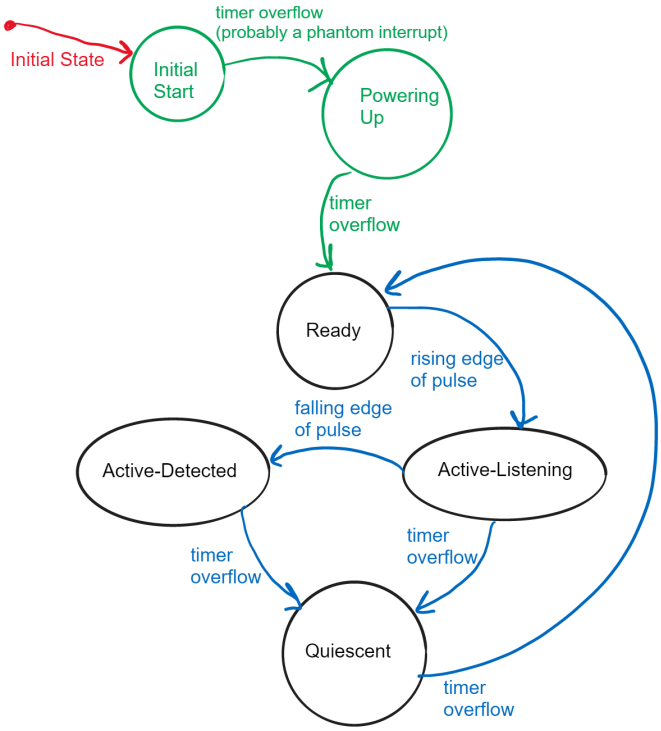


Figure 3: State machine describing the control of the distance sensor.

Initiate a Pulse Place code in **`manage_sensor()`** that, whenever a ping is requested (see Section 4.4), will:

( ) First, set Arduino pin D2 to logic-high

( ) Then, set the variable indicating that a ping is requested to `false`

( ) Next, delay for 10μs

Old Hardware

- You will need a pointer to TIMER1
- Use the timer's counter to busy-wait for 20halfμs

New Hardware

- You will need a pointer to the general-purpose timer
- Use the timer's counter to busy-wait for 10μs

( ) Finally, set pin D2 to logic-low

Several microseconds later, the sensor will emit its ultrasound pulse and raise its `Echo` line to logic-high.

Handle the Start of a Pulse In the function that you registered to handle the pulse edges on the `ECHO` pin, you want to keep track of whether the interrupt handler was triggered for a rising or falling edge. While you *could* do so by checking the logic level on the `ECHO` pin, you can also assume that rising and falling edges alternate – you will never have two rising edges in a row, and you will never have two falling edges in a row.

If the pin interrupt handler was triggered for a rising edge, you want to start the process of determining exactly how much time passes between emitting the pulse and receiving the echo.

( ) First, reset the sensor timer's

( ) Then, *if you are using the new hardware*, copy the value of the timer's counter into a **`volatile`** global variable

- You will need a pointer to the general-purpose timer
- *For the old hardware*, the timer's counter will be 0 after resetting the timer, so copying it is not necessary.

( ) Finally, place the sensor state machine in its "active-listening" state

Two things will happen in the next 38 (or fewer) milliseconds: the signal on the `ECHO` pin will fall to logic-low, and the sensor timer will fire an interrupt. Whichever happens first, the signal falling low or the timer interrupt, will tell us whether there's an object.

Handle the End of a Pulse If the signal on the `ECHO` pin falls to logic-low first, then it's because there's an object that reflected the ultrasound pulse.

If the pin interrupt handler was triggered for a falling edge, and if the sensor is "active-listening," then you want to capture the information needed to compute the distance.

( ) First, copy the value of the timer's counter into a **`volatile`** global variable

Old Hardware

- You will need a pointer to TIMER1
- The value that you copy is the number of half-microseconds between the pulse emission and the echo's return

New Hardware

- You will need a pointer to the general-purpose timer

- This value, minus the value you copied at the start of the pulse, is the number of microseconds between the pulse emission and the echo's return

( ) Then, place the sensor state machine in its "active-detected" state

Handle Timer Interrupt If the sensor timer overflows before the signal on the `ECHO` pin falls low, then there is not an object within detectable range. The sensor will time-out after 36,000–38,000μs, and the sensor timer will fire an interrupt after 32,768μs. Any object whose echo might have been detected after this must be beyond the sensor's detection range.

If the sensor is "active-listening" then no object was detected:

( ) First, indicate that an object has not been detected

( ) Then, place the sensor in its "quiescent" state

On the other hand, if the sensor is "active-detected" then an object was detected:

( ) First, indicate that an object has been detected

( ) Then, place the sensor in its "quiescent" state

As noted above, the sensor requires quiescent period between pulses. We shall allow 65.536ms between pulses, which is ample time for a quiescent period.

If the sensor is "quiescent" when the timer interrupt fires:

( ) Place the sensor in its "ready" state

Compute and Display the Distance Add code to the **`manage_sensor()`** function that, if an object has been detected, computes the distance (as a whole number of centimeters) to the object.

Old Hardware If an object has been detected, then from handling the end of the pulse, you have the number of half-microseconds that the pulse was high.

( ) Use the equation from Section 5.3.2 to compute the distance.

- Do not use **`pow()`** to compute 2^{21} – remember, we're trying to avoid floating point calculations
- If you think back to chapter 3, you'll realize that you do not need to compute 2^{21}

New Hardware If an object has been detected, then you have the time that the pulse initiated (from handling the start of the pulse) and the time that the pulse ended (from handling the end of the pulse).

( ) Compute the length of time that the pulse was high.

If you are not pursuing the temperature extra credit

Hard-code *ADC_register_value* as 889

If you are pursuing the temperature extra credit Read the ADC channel that the temperature sensor is connected to. Using your `adc_t` pointer:

( ) Set `control` register's bits14..12 to `4` indicating that channel 4 is the next channel to convert.

( ) Set the `control` register's bit2 to 1, indicating that we want a conversion.

- The ADC will automatically set `control`'s bit8 to become 0, indicating that a conversion is in progress.

( ) Busy-wait while `control`'s bit8 is 0

- When the ADC automatically sets `control`'s bit8 to 1, it indicates that it is ready for the next conversion.

( ) Copy the ADC result from the `result` register.

( ) Use the equation from Section 5.3.2 to compute the distance.

- Do not use **`pow()`** to compute 2^{33} – remember, we're trying to avoid floating point calculations
- If you think back to chapter 3, you'll realize that you do not need to compute 2^{33}

( ) Display the clearly-labeled distance on the display module.

( ) Now add code to the **`manage_sensor()`** function that, if an object has *not* been detected, displays on the display module a clear indication that there is no detected object.

As a small optimization, you might have your code make these updates only when the sensor is "quiescent."

Test your code. I recommend that you place your Cow Pi on top of its food-container carrying case, or some other object, to reduce the likelihood of the sensor receiving an echo from your worktable.

Most of the work to configure the alarm for Single-Pulse Operation takes place in Section 6.5. You will finish implementing Single-Pulse Operation in Section 7.1 by integrating the detection code from Section 5.5 with the alarm code from Section 6.5. This will require small changes to the code.

6 Generating Sound

You will use the piezoelectric disc to generate the required tones.

6.1 Theory of Operation

Piezoelectric crystals have a property that causes them to turn mechanical energy into electrical energy, and vice-versa. We will apply electricity to the crystal inside the buzzer to cause a thin plate to deform, and then remove the electricity to allow the plate to relax. By repeatedly doing this, we will cause the piezobuzzer to produce sound.

The specification calls for a 5kHz tone. This means that the audio wave must reach its peak 5,000 times per second. Put another way, during every second there must be 5,000 peaks – dividing that out, we see that there must be 200μs between the wave's peaks.

6.2 Practical Considerations

The piezobuzzers have enough internal resistance that we can safely drive the crystal directly from a microcontroller pin.

As we will be driving the piezobuzzer with a digital output pin, we cannot produce a pure sine wave; instead we will produce a square wave. Because it will be a square wave, in addition to the 5kHz tone, there will also be a 15kHz harmonic, plus other harmonics beyond the range of human hearing. We will not attempt to suppress the harmonics.

To produce the exact 5kHz square wave, you need to use a timer interrupt. At first glance, you might think to generate an interrupt every 200μs; however, the wave needs to have a peak *and* a trough every 200μs. (Without troughs, there are no peaks.) Therefore, you want to generate an interrupt every 100μs, alternatingly setting the `BUZZER` pin logic-high and logic-low.

6.3 Examining the Starter Code

The **`initialize_alarm()`** function is where you'll place any alarm-related code that needs to be run once when the program starts. The **`manage_alarm()`** function is where you'll place any alarm-related code that needs to run with every iteration of the program's main loop. You will, of course, add code outside these functions too: an interrupt service routine for a timer, and possibly helper functions.

You'll also notice two variables in the starter code. The `on_period` variable will be used to control how long a tone is generated and the right LED is illuminated when in Single Pulse mode and when in Normal Operation mode. The `total_period` variable will be used to control the time between alarms when in Normal Operation mode.

6.4 Continuous Tone

( ) In *alarm.c*, create an interrupt handler that will be used to respond to the alarm timer's interrupts. Don't forget to pre-declare this functions above **`initialize_alarm()`**.

( ) In **`initialize_alarm()`**, register that function as an ISR for a timer interrupt that has a period of 100μs

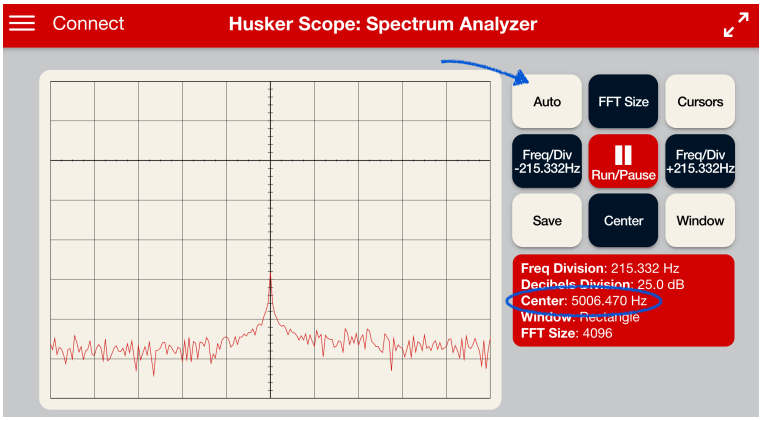


Figure 4: Spectrum Analyzer showing a peak near 5kHz

- On the old hardware, use TIMER2.

( ) In that interrupt handler, add code so that on every other invocation will place a 1 on the `BUZZER` pin and will place a 0 on the alternate invocations.

Test your code

( ) Run your code to check whether it generates a tone on the piezobuzzer, and correct any errors.

( ) After you have code that generates a tone, test that it generates a 5kHz tone.

- ( ) In a web browser, load the Husker Scope Spectrum Analyzer.[8]
- ( ) Place your Cow Pi near your computer's microphone.
- ( ) While the piezobuzzer is producing a tone, click on the "Auto" button in the control-cluster on the right-side of Husker Scope.
- ( ) Husker Scope should then display something similar to Figure 4.
- ( ) Confirm that the center frequency on the display is at or very near 5,000Hz.
- ( ) Correct any errors.

After your code generates the correct tone:

( ) Add code so that the continuous tone is generated only when the system is in Continuous Tone mode (Requirements 2 & 3).

[8] https://cse.unl.edu/~jfalkinburg/husker-scope/app/page/SpectrumAnalyzer

6.5 Single-Pulse Operation

A fully-correct implementation of Single-Pulse Operation mode has the alarm chirp only if an object is detected closer than the threshold range (Requirement 5). *If you and your partner are working synchronously together – if you have a working distance sensor – then temporarily comment-out the code in* `manage_sensor()` *that initiates an ultrasonic pulse.* For the purposes of getting your alarm to chirp, you are temporarily going to have the alarm chirp whenever the pushbutton is pressed.

( ) Add another variable that counts the number of times the ISR has been triggered since the start of an alarm.

( ) Add code to increment that counter each time the timer interrupt handler runs.

- You will actually use this variable as a measure of time for the alarm – if the counter's value is n, then there have been $100n$ milliseconds since the start of an alarm.

( ) Since the `total_period` variable in the starter code is the amount of time *between* alarms when the system is in Normal Operations mode, add code to reset the counter to 0 when it reaches `total_period`.

( ) Add another variable that indicates that an alarm should be sounded.

( ) For now, add code that, whenever a ping is requested (see Section 4.4):

- The variable indicating that an alarm should be sounded becomes `true`
- The variable that counts the number of times the ISR has been triggered becomes 0
- The variable indicating that a ping is requested becomes `false`

Recall that the `on_period` variable is used to determine how long to generate a tone (and illuminate the LED) after detecting an object. (Since you don't yet have a working distance sensor, you're temporarily choosing to generate a tone whenever the user requests a ping.)

( ) Introduce code that will generate the tone whenever these two things are true: an alarm should be sounded, and when the counter is less than `on_period`.

( ) Since the piezo should only chirp once, add code to set the "alarm should be sounded" variable to `false` when the counter exceeds `on_period`.

( ) Test your code.

You may have noticed that when you press the pushbutton, the piezobuzzer generates a tone for considerably longer than 50ms – it can hardly be described as a "chirp."

( ) Change the value assigned to `on_period` to the value that will correctly have the tone generated for only 50ms.

- **Hint** What is $50ms \div 100\mu s$?

( ) Now add code to illuminate the right LED for the same 50ms that the piezobuzzer generates a tone and then deluminates the LED just as the tone is silenced.

Most of the object detection work for Single-Pulse Operation takes place in Section 5.5. You will finish implementing Single-Pulse Operation in Section 7.1 by integrating the detection code from Section 5.5 with the alarm code from Section 6.5. This will require small changes to the code.

7 Putting it all Together

The remaining portion of this assignment integrates the work you put into Sections 4, 5, and 6.

If you and your partner decided to separately work on Sections 5 and 6, and if you're still waiting for your partner to finish their section, there is some work in this section that you can get started on. For example, the user interface portion of Section 7.2.1 can be while waiting for your partner to finish; however, completing the Threshold Adjustment mode will require the "distance" variable from Section 5.5, and making use of the threshold range in Section 7.2.2 requires a working alarm from Section 6.5. Similarly, if you have a working distance sensor, you can do Sections 7.3.1 and 7.3.2 without a working alarm, but you will need a working alarm for Section 7.3.3.

7.1 Single-Pulse Operation

Your distance sensor code responds to a ping request by initiating an ultrasonic pulse, which is what it should do. Your alarm code responds to a ping request by chirping the piezobuzzer and strobing the right LED; however, the alarm should only sound if an object has been detected. There is another incompatibility between the two subsystems: both set the ping request variable to `false` after responding, which means that only one can actually respond to the ping request.

You can resolve this by introducing another shared variable that represents an alarm request.

( ) When an object is detected, the sensor subsystem should request an alarm.

( ) When an alarm is requested, the alarm subsystem should chirp the piezobuzzer, strobe the LEDs, and set the alarm request to `false` (instead of the ping reqeust).

7.2 Threshold Adjustment

When in Single-Pulse Operation mode, the system now alarms once whenever an object is detected. Requirement 5d, however, says that the LEDs should strobe whenever an object is detected, but the piezobuzzer should chirp only when the detected object is closer than the threshold range.

( ) Introduce a shared variable to store the threshold range with an initial value of 400cm, the greatest valid threshold range.

7.2.1 Obtaining the Threshold Range

Requirement 4 describes the user interaction for inputting the threshold range. Implement this requirement in *user_controls.c*.

(Note: while you *should* write your code to safely handle a user attempting to enter more than three digits, we will not attempt to do so during grading.)

( ) After the user has entered a valid threshold range, assign that value to the shared variable.

7.2.2 Applying the Threshold Range

You now have the threshold range, externalized from *user_controls.c*, and the distance to an object, externalized from *sensor.c*.

( ) Modify the alarm code so that, when in Single-Pulse Operation mode, the alarm strobes the LEDs whenever an object is detected, but it only chirps the piezobuzzer when the distance is no greater than the threshold range.

7.3 Normal Operation

You have now completed every operating mode except "Normal Operations." Look over Requirement 6.

7.3.1 Repeated Ultrasonic Pulses

When in Normal Operation mode, the sensor does not wait for a ping request. Instead, it can (and should) send a ping whenever the sensor is in its "ready" state.

Modify your code so that:

( ) When the system is in Normal Operation mode, it initiates a pulse whenever the sensor is "ready."

( ) Re-compute the distance and update the display with each pulse.

7.3.2 Compute the Rate of Approach

Since the distance sensor cannot determine the change of direction to the detected object (indeed, it cannot determine the direction), you cannot calculate the object's speed. You can, however, calculate the longitudinal component of its speed – that is, how fast it is approaching the sensor. (A retreating object would, of course, have a negative rate of approach.) Since the sensor does not detect doppler shift, you must calculate the rate of approach based on the difference between two distance measurements. There are two options to compute the rate of approach in centimeters per second. Choose one. (Or, if you can think of a third approach, you can choose it.)

Compare two distances separated in time by 1 second

If you have a working alarm subsystem, then you have a timer interrupt firing every 100μs. For every 10,000 times that the alarm timer's ISR is triggered, 1 second has passed. If you externalize the number of times that the ISR runs, then your code in *sensor.c* can subtract the current distance from the distance calculated 1 second ago. The specification does not require that the rate of approach be updated more frequently than once per second, so this is an acceptable approach. The only complication is that you'll need to make sure that the user doesn't see an erroneous speed value in the second between obtaining the very first distance after detection and obtaining the second distance.

Compare the pulse duration for two adjacent measurements

Alternatively, you can update the speed every 65,536μs. The actual distance travelled will be too small to measure with an integer number of centimeters, so we shall instead rely on the differences in the pulse durations $\tau_1 and \tau_2$. Regardless of whether you're using the old hardware or the new hardware, this calculation will need to use 64-bit integers for the intermediate terms. If we assume that the wall-clock times of reflection are exactly 65,536μs apart,[9] and that the air temperature does not change appreciably between detections then:

Old Hardware

$$speed = \frac{\Delta distance}{time} = \frac{(\tau_2 - \tau_1) \times 18,025 cm}{2^{21} \times 65,536 \mu s}$$

$$= (\tau_2 - \tau_1) \times 281,640,625 \div 2^{31} \ \frac{cm}{s}$$

New Hardware

[9] If the object is moving, then the actual difference in the wall-clock times of reflection won't be 65,536μs apart; however, the error resulting from this simplifying assumption is less than the rounding error.

$$speed = \frac{\Delta distance}{time} = \frac{(\tau_2 - \tau_1) \times (256,108,888 - 121,907 \times ADC\_register\_value)\, cm}{2^{33} \times 65,536 \mu s}$$

$$= 1,000,000 \times (\tau_2 - \tau_1) \times (256,108,888 - 121,907 \times ADC\_register\_value) \div 2^{49}\ \frac{cm}{s}$$

( ) Add code to **`manage_sensor()`** to calculate the rate of approach when the system is in Normal Operations.

( ) Whenever you have calculated a new rate of approach, update the display with that rate of approach.

7.3.3 Repeated Alarm

Your remaining task is to repeatedly activate the alarm when in Normal Operations. You probably noticed the `total_period` variable in *alarm.c*, immediately below the `on_period` variable. Just as `on_period` is used to determine how long the piezobuzzer should emit its tone and the LEDs should be illuminated, the `total_period` is used to determine how much time should transpire between activations of the alarm. (If you wish, you can imagine a `off_period` variable whose value is always `total_period - on_period`.)

Repeatedly activate the alarm

( ) Add code so that, when an object is detected while the system is in Normal Operations mode, the alarm will repeatedly activate.

- We recommend that you use the `total_period` variable as part of that solution.

Vary the time between activations

( ) Using Table 1 and the distance to the detected object, change `total_period`'s value so that the time between alarm activations is correct.

8 Turn-in and Grading

When you have completed this assignment, upload *alarm.c*, *sensor.c*, *user_controls.c*, and *shared_variables.h* to Canvas.

No Credit for Uncompilable Code

If the TA cannot create an executable from your code, then your code will be assumed to have no functionality.[10] Before turning in your code, be sure to compile and test your code on your Cow Pi with the original driver code and the original header file(s).

Late Submissions

This assignment is due before the start of your lab section. The due date in Canvas is five minutes after that, which is ample time for you to arrive to lab and then discover that you'd forgotten to turn in your work without Canvas reporting your work as having been turned in late. We will accept late turn-ins up to one hour late, assessing a 10% penalty on these late submissions. Any work turned in more than one hour late will not be graded.

Rubric

This assignment is worth 60 points.

User Controls

_____ **+1** The switches control the mode of operation as specified

_____ **+2** The user can request a ping when the system is in Single Pulse mode

_____ **+2** The user can request another ping when the system is in Single Pulse mode

_____ **+1** The user is prompted to enter a new threshold range when the system is in Threshold Adjustment mode

_____ **+2** The user can enter a new threshold range when the system is in Threshold Adjustment mode

_____ **+2** Valid threshold ranges are those between 50cm and 400cm, inclusive

_____ **+1** The user is given a helpful error message after entering an invalid threshold range

_____ **+1** The user is re-prompted to enter a threshold range after entering an invalid threshold range

_____ **+1** The user is shown a confirmation message after entering a valid threshold range

Sensor

[10] At the TA's discretion, if they can make your code compile with *one* edit (such as introducing a missing semicolon) then they may do so and then assess a 10% penalty on the resulting score. The TA is under no obligation to do so, and you should not rely on the TA's willingness to edit your code for grading. If there are multiple options for a single edit that would make your code compile, there is no guarantee that the TA will select the option that would maximize your score.

_____ **+2** There is code to initiate an ultrasound pulse

_____ **+3** There is code to detect the length of the pulse

_____ **+3** The pulse's length is measured to a precision of no greater than 1μs[11]

_____ **+3** The pulse's length is measured as accurately as possible[12]

_____ **+3** The code correctly recognizes the that no object has been detected, if no object reflects the ultrasound pulse

_____ **+2** The distance to an object is correctly calculated from the pulse's length

_____ **-16** The Arduino **`pulseIn()`** and/or **`pulseInLong()`** function, or a third-party library, is used to time the ultrasonic echo sensor's pulse.

Alarm

_____ **+1** There is code to generate an audible tone

_____ **+1** The system continuously generates the tone when in Continuous Tone mode

_____ **+2** The audible tone has a frequency of 5kHz

_____ **+2** A chirp is an audible tone lasting 50ms

_____ **+2** A chirp occurs for a detected object that is closer than the threshold range

_____ **+2** A chirp only occurs for a detected object that is closer than the threshold range

_____ **+2** A strobe is an illumination of the right LED for 50ms

_____ **+2** A strobe occurs for any detected object

_____ **+2** A strobe only occurs for a detected object

_____ **-16** The Arduino **`tone()`** and/or **`noTone()`** function, or a third-party library, is used to generate tones on the piezoelectric disc.

Object Detection

_____ **+2** When there is no in-range object, the system displays a message to that effect

_____ **+2** When an object is detected (in Single-Pulse mode or Normal Operation mode), the system displays the distance to the object

[11] The assignment describes obtaining a precision of $\frac{1}{2}\mu$s, which is, of course, acceptable. The 4μs precision offered by the Arduino **`micros()`** function is *not* acceptable.

[12] This means you use interrupts to detect the rising and falling edges of the pulse.

_____ **+2** When an object is detected in Normal Operation mode the rate of approach is displayed

_____ **+2** When an object is detected in Normal Operation mode, the rate of approach is updated at least once every second

_____ **+2** When an object is detected in Single-Pulse mode, the system generates exactly one alarm

_____ **+2** When an object is detected in Normal Operation mode, the system repeatedly generates alarms

_____ **+2** When the system repeatedly generates alarms, the time between alarms is as specified in Table 1

Code Quality

_____ **+1** The code is clean, well-organized, has good variable and function names, and is otherwise understandable

_____ **-1** for each **`goto`** statement, **`continue`** statement, **`break`** statement used to exit from a loop, or **`return`** statement that occurs within a loop.

Bonuses

_____ **Bonus +1** Use the actual *ADC_register_value* when performing distance and speed calculations.

_____ **Bonus +2** Get assignment checked-off by TA or professor during office hours before it is due (you cannot get both check-off bonuses)

_____ **Bonus +1** Get assignment checked-off by TA at *start* of your scheduled lab immediately after it is due (your code must be uploaded to Canvas *before* it is due; you cannot get both bonuses)

Epilogue

A technician installing a new range finder outside the lab door briefly sets off the alarm, but then the range finder falls quiet and faithfully reports that nothing is approaching. As reports come in of facilities getting secured with Cow Pi-based locks, and of accurate speciment counts accomplished with Cow Pi-based calculators, Archie smiles and tells you that this was a job well done. With all of the excitement neatly wrapped-up and arriving at a satisfactory conclusion, you look forward to a boring career in which there's absolutely no screaming and running for your life.

The end...?

A Appendix: Lab Checkoff

You are not required to have your assignment checked-off by a TA or the professor. If you do not do so, then we will perform a functional check ourselves. In the interest of making grading go faster, we are offering a small bonus to get your assignment checked-off at the start of your scheduled lab time immediately after it is due. Because checking off all students during lab would take up most of the lab time, we are offering a slightly larger bonus if you complete your assignment early and get it checked-off by a TA or the professor during office hours.

( ) Position your Cow Pi's storage box upright, a little more than 1 meter from the Cow Pi.

( ) Place both switches in the left position.

( ) Upload your code to your Cow Pi, and leave your code open in the IDE.

( ) Confirm that the system detects the box and not something closer (such as a computer or the table surface).

1. ( ) Show and explain to the TA how your code generates a tone with a frequency of 5kHz; that is, it has a period of 200μs.

2. ( ) Place the right switch in the right position, putting the system in Continuous Tone mode. The system generates a continuous 5kHz tone.

   (TA, confirm that the tone is 5kHz by code inspection and by ear; confirm with the HuskerScope spectrum analyzer if you aren't sure.)
   +1 There is code to generate an audible tone
   +1 The system continuously generates the tone when in Continuous Tone mode
   +2 The audible tone has a frequency of 5kHz

3. ( ) Place the left switch in the right position, putting the system in Threshold Adjustment mode. The system prompts for a new threshold range.
   +1 The user is prompted to enter a new threshold range when the system is in Threshold Adjustment mode

4. ( ) Enter a range of 25, using the '#' key to indidicate that you have fininished entering the value. The system displays a helpful error message explaining that this is not a valid threshold range. The system then prompts the user for a new threshold range.
   +1 The user is given a helpful error message after entering an invalid threshold range
   +1 The user is re-prompted to enter a threshold range after entering an invalid threshold range

5. ( ) Enter a range of 450, using the '#' key to indidicate that you have fininished entering the value. The system displays a helpful error message explaining that this is not a valid threshold range. The system then prompts the user for a new threshold range.

6. ( ) Enter a range of 75, using the '#' key to indidicate that you have fininished entering the value. The system displays a message confirming the new threshold range.
   +2 Valid threshold ranges are those between 50cm and 400cm, inclusive
   +1 The user is shown a confirmation message after entering a valid threshold range
   +2 The user can enter a new threshold range when the system is in Threshold Adjustment mode

7. ( ) Place the right switch in the left position, putting the system in Single Pulse mode. The system might indicate that no object has been detected yet; however, this is not required information before initiating a ping.

8. ( ) Show and explain to the TA how your code initiates a pulse.

9. ( ) Show and explain to the TA how your code measures the length of a pulse.

10. ( ) Show and explain to the TA how your code achieves the required precision (no greater than 1μs) and accuracy (immediately detect pulse edges without waiting for code in the main loop to poll the pin).

11. ( ) Press the pushbutton to initiate a pulse. The right LED strobes once. The piezodisc does not chirp. The system displays the correct distance to the wall (or book or other object).
    +2 There is code to initiate an ultrasound pulse
    +3 There is code to detect the length of the pulse
    +3 The pulse's length is measured to a precision of no greater than $1\mu s$
    +3 The pulse's length is measured as accurately as possible
    +2 The user can request a ping when the system is in Single Pulse mode
    +2 The distance to an object is correctly calculated from the pulse's length
    +2 When an object is detected, the system displays the distance to the object
    +2 When an object is detected in Single-Pulse mode, the system generates exactly one alarm
    +2 A strobe is an illumination of the right LED for 50ms
    +2 A strobe occurs for any detected object

12. ( ) Slightly change the distance between the Cow Pi and the target object. Press the pushbutton to initiate a pulse. The LED strobes once. The piezodisc does not chirp. The system displays the new distance to the wall (or book or other object).
    +2 The user can request another ping when the system is in Single Pulse mode

13. ( ) Place the left switch in the left position, putting the system in Normal Operation mode. The system displays the distance to the target object, and it displays an

approach rate of 0cm/s. The LED strobes once per seccond (100ms), but the piezodisc does not chirp.
+1 The switches control the mode of operation as specified
+2 When an object is detected in Normal Operation mode, the system repeatedly generates alarms

14. ( ) Slowly move the Cow Pi closer to the wall, or slowly move the book (or other object) closer to the Cow Pi. As you do so, vary the rate of approach slighly to demonstrate that the rate of approach updates. The displayed distance changes with the decreasing distance to the target object. The system displays a plausible, positive rate of approach that updates at least once every second.
+2 When an object is detected in Normal Operation mode the rate of approach is displayed
+2 When an object is detected in Normal Operation mode, the rate of approach is updated at least once every second

15. ( ) As the distance between the Cow Pi and the target object decreases, note that:

When the distance falls below 100cm the LED strobes more frequently, once every 750ms ($\frac{3}{4}$sec)

When the distance falls below 75cm the piezodisc chirps every 750ms

When the distance falls below 50cm the LED strobes, and the piezodisc chirps, every 500ms ($\frac{1}{2}$sec)

When the distance falls below 25cm the LED strobes, and the piezodisc chirps, every 250ms ($\frac{1}{4}$sec)

When the distance falls below 10cm the LED strobes, and the piezodisc chirps, every 125ms ($\frac{1}{8}$ sec)

+2 A chirp is an audible tone lasting 50ms
+2 When the system repeatedly generates alarms, the time between alarms is as specified

16. ( ) Place the both switches in the right position, putting the system in Threshold Adjustment mode. The system prompts for a new threshold range.

17. ( ) Enter a range of 55. The system displays a message confirming the new threshold range.

18. ( ) Place the both switches in the left position, putting the system in Normal Operation mode. The system displays the distance to the target object, and it displays an approach rate of 0cm/s.

19. ( ) Slowly move the Cow Pi away from the wall, or slowly move the book (or other object) away from the Cow Pi. The displayed distance changes with the decreasing

distance to the target object. The system displays a plausible, negative rate of approach.

20. ( ) As the distance between the Cow Pi and the target object decreases, the alarms become less urgent. Note that as the distance increases above 55cm, the piezodisc stops chirping but the LED continues to strobe.
    +2 A chirp occurs for a detected object that is closer than the threshold range
    +2 A chirp only occurs for a detected object that is closer than the threshold range

21. ( ) Reorient the Cow Pi, or remove the book (or other object) so that there are no in-range objects to detect. The system displays a message indicating that no object is detected. The LED does not strobe, and the piezodisc does not chirp.
    +3 The code correctly recognizes the that no object has been detected, if no object reflects the ultrasound pulse
    +2 When there is no in-range object, the system displays a message to that effect
    +2 A strobe only occurs for a detected object

22. ( ) Show the TA any code they have not yet examined.
    +1 The code is clean, well-organized, has good variable and function names, and is otherwise understandable

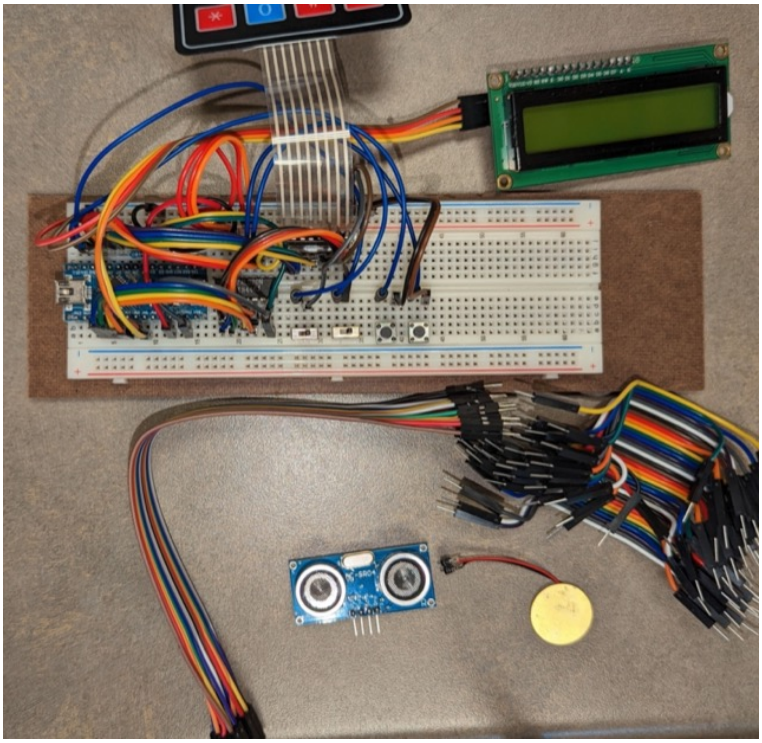


Figure 5: Components needed for the Range Finder

B Installing Additional Hardware Components (Cow Pi mk1f)

You will need to add a couple of hardware components to your Cow Pi circuit before you can start this lab.

B.1 Necessary Components

Figure 5 shows the components you will need for the range finder and alarm.

You will need:

- Your Cow Pi hardware circuit
- A piezoelectric disc
  - Piezoelectric devices can convert electric energy into mechanical energy, and vice-versa
  - You will use a piezoelectric disc to create an audible alarm
- An ultrasonic echo sensor

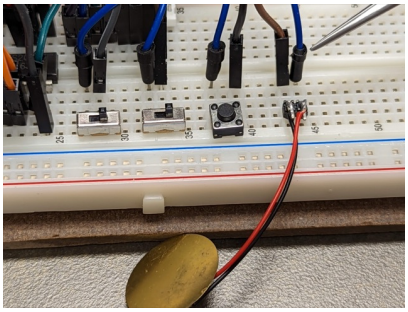


Figure 6: Connecting the Piezoelectric Disc.

  - The two prominent drums on this device are ultrasonic transducers, one of which converts electricity to 40kHz sound (well above the range of human hearing), and the other of which converts 40kHz sound into electricity
  - You will measure the time between the ultrasound being transmitted and its reflecting being received to determine the distance to whatever is reflecting the ultrasound

- Six male-to-male wires (three of these must be 20cm, and the other three can be 10cm or 20cm)

You and your partner only need to modify one of your Cow Pis (but you may modify both).

B.2 Connecting the Piezoelectric Disc

The Arduino Nano's pin D13 is used for the Arduino Nano's internal LED, the "left" LED. In the group project, we will use it to control the piezodisc.

( ) Insert the piezodisc's header into unused rows on the breadboard (Figure 6).

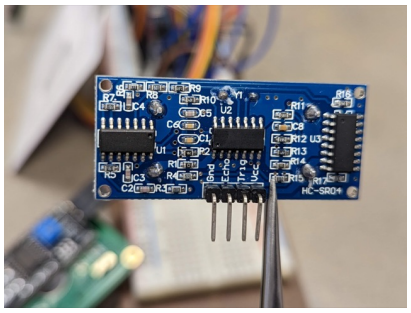
( ) Use a 20cm wire to connect the piezodisc's red lead to the Arduino Nano's pin D13 (contact point a1 is a good choice).

( ) Use another wire to connect the piezodisc's black lead to the upper ground (–) rail.

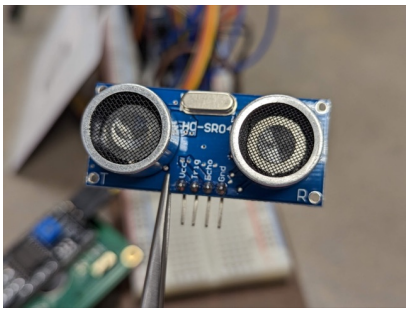
The piezodisc can now be operated through the Arduino Nano's pin D13, which is the pin that the built-in LED is connected to.

B.3 Connecting the ultrasonic echo sensor

Take a look at the ultrasonic echo sensor (Figure 7). Notice that it has four pins, labeled `Gnd`, `Echo`, `Trig`, and `Vcc`.



(a) The back side of the ultrasonic sensor.



(b) The front side of the ultrasonic sensor.

Figure 7: The ultrasonic echo sensor.

( ) Insert the ultrasonic echo sensor into unused rows on the breadboard, pointing away from you. The ultrasonic transducers should point toward the upper power/ground rails.

( ) Insert a 20cm wire into contact point j11 (Figure 8b). Make a note of the wire's color for future reference. This is your **D2 wire**.

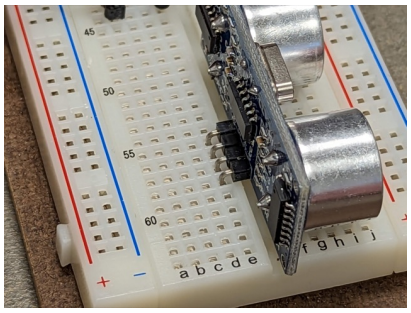
( ) Insert a 20cm wire into contact point j10 (Figure 8c). Make a note of the wire's color for future reference. This is your **D3 wire**.

( ) Insert the D2 wire into the same row as the sensor's `Trig` pin, and the D3 wire into the same row as the sensor's `Echo` pin (Figure 8e).

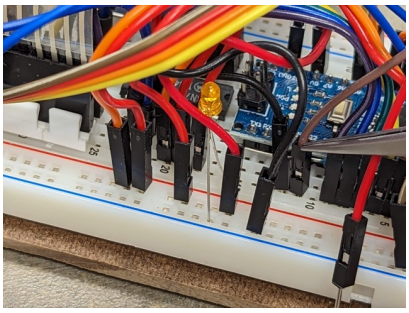
( ) Use a wire to connect the sensor's `Gnd` pin to the upper ground (−) rail. Use another wire to connect the sensor's `Vcc` pin to the upper power (+) rail. See Figures 8f–8g. For best performance, position these wires so that they are not directly in front of the ultrasonic transducers.

The ultrasonic echo sensor is now connected to the Arduino Nano's pins D2 & D3. The starter code will configure D2 (`TRIGGER`) to be an output pin and D3 (`ECHO`) to be an input pin.

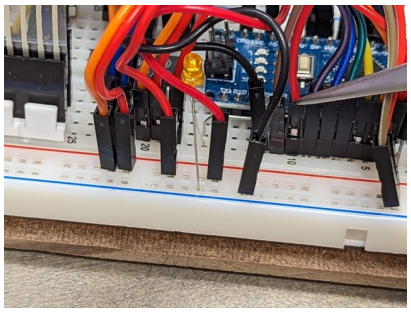
Your Cow Pi circuit is now ready for you to design and code the software for a range finder and alarm.



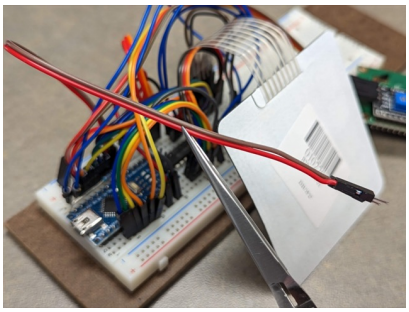
(a) The ultrasonic sensor inserted into the breadboard.



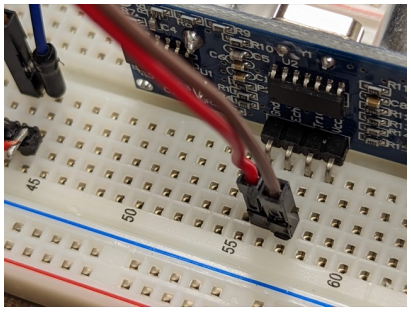
(b) Inserting a wire into contact point j11.



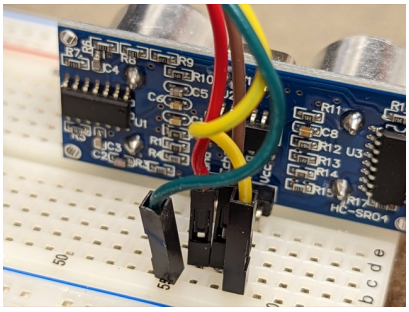
(c) Inserting a wire into contact point j10.



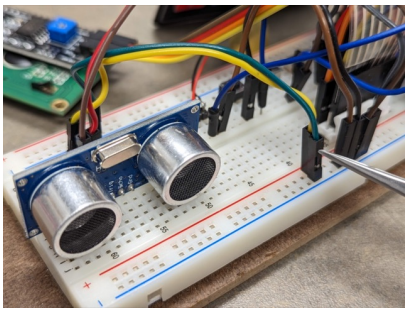
(d) The two 20cm wires, ready to be used.



(e) The ultrasonic sensor connected to the D2 and D3 wires.



(f) Inserting one end of the power and ground wires.



(g) Inserting the other end of the power and ground wires in the power and ground rails.

Figure 8: Connecting the ultrasonic echo sensor.

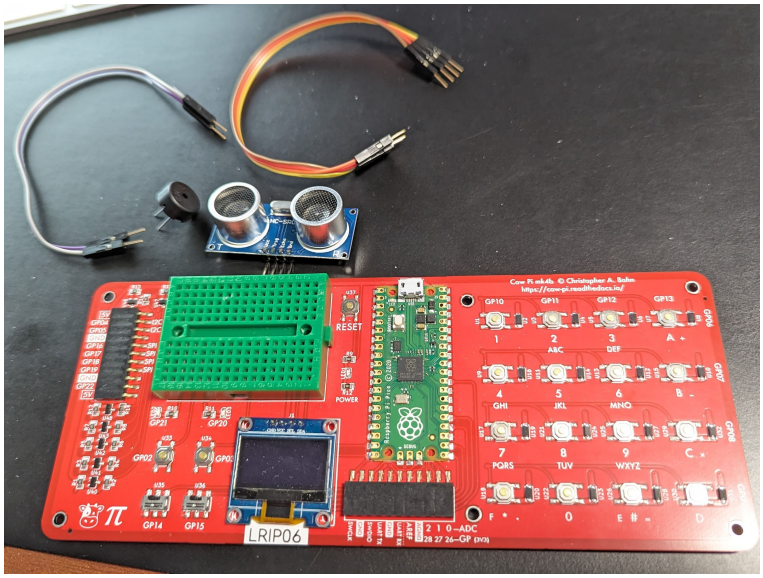


Figure 9: Components needed for the Range Finder

C Installing Necessary Hardware Components (Cow Pi mk4b)

You will need to add a couple of hardware components to your Cow Pi circuit before you can start this lab.

C.1 Necessary Components

Figure 9 shows the components you will need for the range finder and alarm.

You will need:

- Your Cow Pi hardware circuit
- A piezoelectric "passive buzzer"
  - Piezoelectric devices can convert electric energy into mechanical energy, and vice-versa
  - You will use a piezoelectric device to create an audible alarm
- An ultrasonic echo sensor
  - If your sensor is still in the red bubblewrap packaging, there are also two resistors in the packaging. We will not use these, but we might have a use for them in a future semester. Please place them in the Cow Pi's storage box.

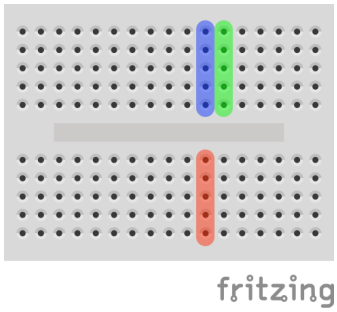


Figure 10: Terminal strips on a mini-breadboard

  - The two prominent drums on this device are ultrasonic transducers, one of which converts electricity to 40kHz sound (well above the range of human hearing), and the other of which converts 40kHz sound into electricity
  - You will measure the time between the ultrasound being transmitted and its reflecting being received to determine the distance to whatever is reflecting the ultrasound

- Six 20cm male-to-male wires (you can separate these wires from a multi-wire cable)

There is a labeled header on the left side of the Cow Pi; we will use this to connect the hardware components to the RP2040 microcontroller.

C.2 The Mini-Breadboard on the Cow Pi

A key feature of solderless breadboards, such as the mini-breadboard on your Cow Pi, are the groups of 5 holes (Figure 10). Each group of five is a *terminal strip*.

The five holes in a terminal strip are electrically connected to each other but are electrically isolated from the other terminal strips.[13] For example, in Figure 10, all holes in the terminal strip that is highlighted in blue are connected to each other, but they are not connected to the holes in the adjacent terminal strip highlighted in green. Similarly, they are not connected to the holes in the red terminal strip on the other side of the gutter.

A consequence of this is that any components' connectors that are inserted into a terminal strip are connected to the connectors of other components that are inserted into the same terminal strip.

If you want to learn more, a very good overview of solderless breadboards can be found here: https://learn.adafruit.com/breadboards-for-beginners?view=all

Figure 11 shows where you will insert the hardware components for the group project. (The precise placement is not critical, so long as you make a note of which terminal strips you use.)

[13] They are isolated for DC signals, and parasitic reactance is negligible for AC signals below about 10 kHz.

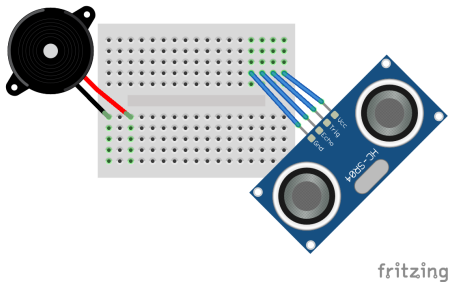


Figure 11: Terminal strips on a mini-breadboard

C.3 Connecting the ultrasonic echo sensor

Take a look at the ultrasonic echo sensor (Figure 7). Notice that it has four pins, labeled `Gnd`, `Echo`, `Trig`, and `Vcc`.

( ) Insert the ultrasonic echo sensor into unused rows on the breadboard, pointing away from you. Leave room behind the sensor to connect a wire into each of the same terminal strips that the sensor uses.

( ) Insert four 20cm wires, one into each of the same terminal strips that the sensor uses. Make a note of which color wire corresponds to which of the sensor's pins.

( ) Insert the other end of the `Vcc` wire into a `5V` slot

( ) Insert the other end of the `Gnd` wire into a `GND` slot

( ) Insert the other end of the `Echo` wire into the `GP16` slot

( ) Insert the other end of the `Trig` wire into the `GP17` slot

The ultrasonic echo sensor is now connected to the Raspberry Pi Pico's pins `GP16`–`GP17`. The starter code will configure `GP17` (`TRIGGER`) to be an output pin and `GP16` (`ECHO`) to be an input pin.

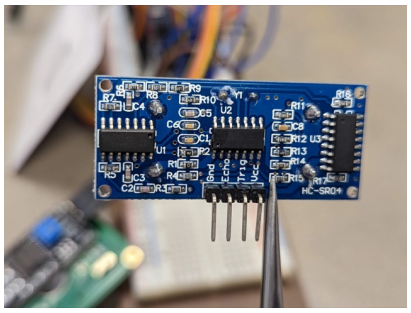
C.4 Connecting the Piezobuzzer

( ) Insert the piezobuzzer into unused rows on the breadboard. Leave room to connect a wire into each of the same terminal strips that the piezobuzzer.

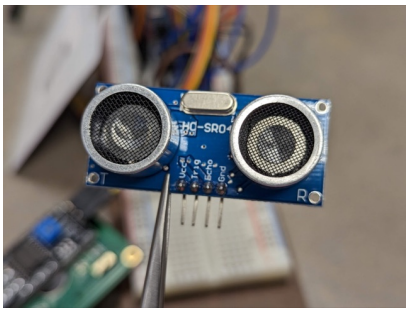
( ) Insert two 20cm wires, one into each of the same terminal strips that the piezobuzzer uses.

( ) Insert the other end of one of the wires into a `GND` slot

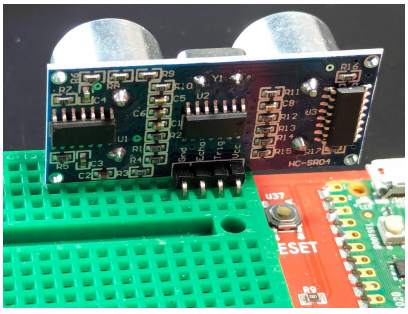
- For *this* particular device, the polarity does not matter, so it doesn't matter which wire is connected to `GND`



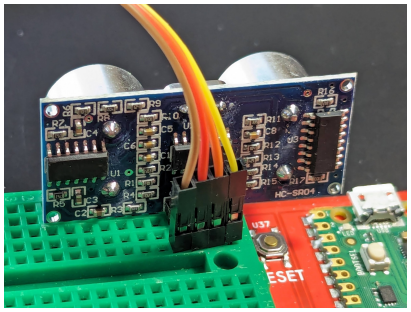
(a) The back side of the ultrasonic sensor.



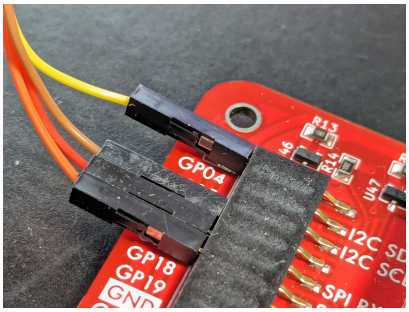
(b) The front side of the ultrasonic sensor.



(c) Inserting the ultrasonic sensor.

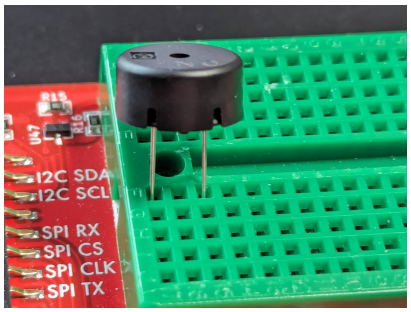


(d) One end of the board's connection to the sensor.

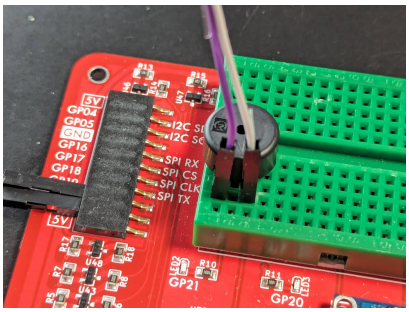


(e) The other end of the board's connection to the sensor.

Figure 12: The ultrasonic echo sensor.



(a) Inserting the piezobuzzer



(b) The front side of the ultrasonic sensor.

Figure 13: The piezoelectric "passive buzzer".

( ) Insert the other end of the other wire into the `GP22` slot

The piezodisc can now be operated through the Raspberry Pi Pico's pin `GP22`. The starter code will configure `GP22` (`BUZZER`) to be an output pin.

Your Cow Pi is now ready for you to design and code the software for a range finder and alarm.

D Distance Equation Formulation

D.1 Old Hardware

Assuming that the air temperature is 70° Fahrenheit (21.1° Celsius), the speed of sound is $348.8\frac{m}{s}$.[14]

The distance to an object, being half of the round-trip distance, is

$$distance = \frac{1}{2} \times time_{\mu s} \times \frac{343.8m}{1s} \times \frac{100cm}{1m} \times \frac{1s}{1,000,000\mu s} = time_{\mu s} \times 0.01719\frac{cm}{\mu s}$$

Because the timer will be configured with a "tick" of a half-microsecond, we can re-express this equation as

$$distance = time_{\text{half}\mu s} \times 0.008595\frac{cm}{\text{half}\mu s}$$

In real-number arithmetic, multiplying the speed of sound by a constant non-zero finite value and by time, and dividing by the same constant value, will produce the same distance as if the speed of sound were multiplied by time. In bit-limited integer arithmetic, the same is true *if* the original speed were an integer, *if* the product doesn't overflow, and *if* all multiplications occur before division. Because we're treating the speed of sound as a constant, and the conversion factor is also a constant, part of the multiplication can occur at compile-time, ensuring that both the run-time multiplier and multiplicand are integers. Normally, integer division without a hardware divider is only slightly more desirable than floating point arithmetic; however, if the constant factor is a power of two, then the division can be accomplished with a bitshift operation.

$$distance = time_{\text{half}\mu s} \times 0.008595\frac{cm}{\text{half}\mu s} \times \frac{2^{21}}{2^{21}} \approx time_{\text{half}\mu s} \times \frac{18,025}{2^{21}}\frac{cm}{\text{half}\mu s}$$

when the air temperature is 70° Fahrenheit (21.1° Celsius).

D.2 New Hardware

D.2.1 Calculating the Speed of Sound

The speed of sound in air is[15]

$$speed = 331.228 \times \sqrt{1 + \frac{T}{273.15}}\frac{m}{s} = 0.0331228 \times \sqrt{1 + \frac{T}{273.15}}\frac{cm}{\mu s}$$

[14] https://www.weather.gov/epz/wxcalc_speedofsound

[15] https://www.weather.gov/media/epz/wxcalc/speedOfSound.pdf

The presence of that square root would seem to be an obstacle to our goal of avoiding floating point calculations; however, the expression's Taylor series gives us

$$\begin{aligned} speed &= 0.0331228 \times \left(1 + \frac{T}{546.30}\right) \frac{cm}{\mu s} \\ &= 0.0331228 \times \left(\frac{546.30 + T}{546.30}\right) \frac{cm}{\mu s} \\ &= \frac{546.30 + T}{16493.17} \frac{cm}{\mu s} \end{aligned}$$

D.2.2 Calculating the Temperature

The RP2040 has a temperature sensor connected to one of its analog-digital converter (ADC) inputs. Section 4.9.5 of the RP2040 datasheet[16] shows that the temperature, in °C, is:

$$T = 27 - \frac{ADC\_voltage - 0.706}{0.001721}$$

The ADC is a 12-bit ADC with a reference voltage of 3.3V, and so

$$\begin{aligned} T &= 27 - \frac{\frac{3.3 \times ADC\_register\_value}{2^{12}} - 0.706}{0.001721} \\ &= 27 - \frac{3.3 \times ADC\_register\_value - 0.706 \times 2^{12}}{0.001721 \times 2^{12}} \\ &= \frac{0.046467 \times 2^{12} - (3.3 \times ADC\_register\_value - 0.706 \times 2^{12})}{0.001721 \times 2^{12}} \\ &= \frac{0.752467 \times 2^{12} - 3.3 \times ADC\_register\_value}{0.001721 \times 2^{12}} \\ &\approx \frac{3.3 \times 0.22802 \times 2^{12} - 3.3 \times ADC\_register\_value}{0.001721 \times 2^{12}} \\ &\approx \frac{0.22802 \times 2^{12} - ADC\_register\_value}{0.0005215 \times 2^{12}} \end{aligned}$$

[16] https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

D.2.3 Calculating the Distance

Substituting the expression for the temperature into the expression for the speed of sound, we get:

$$\begin{aligned}
speed &= \frac{546.30 + \frac{0.22802 \times 2^{12} - ADC_register_value}{0.0005215 \times 2^{12}}}{16493.17} \frac{cm}{\mu s} \\
&= \frac{546.30 \times 0.0005215 \times 2^{12} + 0.22802 \times 2^{12} - ADC_register_value}{16493.17 \times 0.0005215 \times 2^{12}} \frac{cm}{\mu s} \\
&= \frac{0.5129 \times 2^{12} - ADC_register_value}{8.601438 \times 2^{12}} \frac{cm}{\mu s} \\
&\approx \left(0.05963 - \frac{ADC_register_value}{8.601438 \times 2^{12}}\right) \frac{cm}{\mu s} \\
&= \left(0.05963 \times \frac{2^{32}}{2^{32}} - \frac{ADC_register_value}{8.601438 \times 2^{12}} \times \frac{2^{20}}{2^{20}}\right) \frac{cm}{\mu s} \\
&\approx \left(\frac{256,108,888}{2^{32}} - \frac{121,907 \times ADC_register_value}{2^{32}}\right) \frac{cm}{\mu s} \\
&= \frac{256,108,888 - 121,907 \times ADC_register_value}{2^{32}} \frac{cm}{\mu s}
\end{aligned}$$

The distance to an object, being half of the round-trip distance, is

$$\begin{aligned}
distance &= \frac{1}{2} \times time_{\mu s} \times \frac{256,108,888 - 121,907 \times ADC_register_value}{2^{32}} \frac{cm}{\mu s} \\
&= time_{\mu s} \times \frac{256,108,888 - 121,907 \times ADC_register_value}{2^{33}} \frac{cm}{\mu s}
\end{aligned}$$