Lab 12

Implementing a Combination Lock

Due: Week of December 9, 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. Specifically, you will implement an electronic combination lock.

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. 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 combination lock so that only authorized people can get into our lab facilities."

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 **??**–7 in sequence. The assignment is also written so that you and your partner can add the new hardware and work on Section **??** together, then split up with one student working on Section **??** and the other working on Section **??**, 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 **??** over Section **??**, 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 not use any libraries, functions, macros, types, or constants from the Arduino core. This prohibition includes, but is not limited to, the **`delay()`**, **`delayMicroseconds`**, **`millis()`**, **`micros()`**, **`tone()`**, **`noTone()`**, **`pulseIn()`**, and **`pulseInLong()`** functions.

1.1.2 Constraints on MBED OS

You may not use any MBED-specific functions, macros, types, or constants except as provided in the starter code.

1.1.3 Constraints on the CowPi library

You may use any functions provided by the CowPi[1] and you may use any data structures[2] 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 *nuros.unl.edu*. Once downloaded, unpackage the file and open the project in your IDE.

2.1 Description of RangeFinder Files

2.1.1 combolock.c, interrupt_support.h, interrupt_support.cpp, display.h, display.cpp

Do not edit *combolock.c*, *interrupt_support.h*, *interrupt_support.cpp*, *display.*, or *display.cpp*.

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

2.1.2 rotary-encoder.h & rotary-encoder.c

Do not edit *rotary-encoder.h*

The *rotary-encoder.c* file is where you will process inputs from the rotary encoder.

2.1.3 servomotor.h & servomotor.c

Do not edit *servomotor.h*

The *servomotor.c* file is where you will control the servomotor.

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

2.2 lock-controller.h & lock-controller.c

Do not edit *lock-controller.h*

The *lock-controller.c* file is where you will implement the logic for the combination lock.

2.3 Assemble the Hardware

BEFORE YOU PROCEED FURTHER:

( ) Add the new hardware to your Cow Pi as described in Appendix B.

After you have assembled the hardware, you can work on the lab entirely with your partner, perhaps pair programming, or you can decide to have one partner work on Section 4 and the other work on Section 5. Regardless, you will need to work together on Section 7, as this is when you will integrate the work from Sections 4–5.

3 Test Mode Specification

If the **right switch** is in the *left position* when the system boots, then the system shall enter its test mode. The test mode is used to demonstrate the correct handling of the **rotary encoder** and of the **servomotor**.

When the system is in its test mode:

1. The system shall display the number of times that the **rotary encoder** is turned clockwise and the number of times that the **rotary encoder** is turned counterclockwise.
2. If the **right pushbutton** is *pressed*, the servomotor shall take its *center position*.
3. If the **right pushbutton** is *not pressed*, the servomotor shall follow the **left switch**:
   - If the **left switch** is in the *left position*, the servomotor shall rotate *fully clockwise*.
   - If the **left switch** is in the *right position*, the servomotor shall rotate *fully counterclockwise*.
4. The system shall display "SERVO: center", "SERVO: left", or "SERVO: right" according to the command given to the servo per Requirements 2–3.

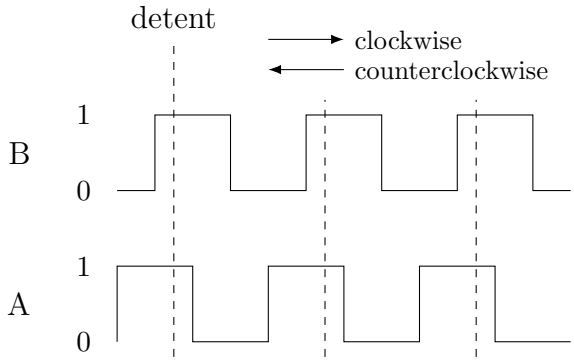


Figure 1: Quadrature encoding of the rotational position in a rotary encoder.

4 Decoding the Rotary Encoder's Direction

4.1 Theory of Operation

The rotary encoder consists of a shaft that can rotate without stop in either direction, and detents that hold the shaft in position when you are not rotating it. Electrically, it has a pair of wipers, each of which is connected to a pin at one end, and which share a common pin at the other end. By connecting the common pin to ground (as we have) and connecting the wipers' pins to pulled-up input pins on the microcontroller (as we have), then we can read the logic values of the wipers.

As the shaft rotates, the pair of wipers each cycle through a square wave. The two wipers' square waves form a *quadrature*; that is, they are 90°out of phase with each other, as shown in Figure fig:quadrature. By tracking which pin changes value first or second, we can determine which direction the shaft is rotating.

4.2 Changes to *rotary-encoder.c*

Open *rotary-encoder.c*. Locate the **`get_quadrature()`** function. The `A` component of the signal from Figure fig:quadrature is on input pin 16, and the `B` component is on input pin 17.

( ) Read the quadrature from the input pins into a variable.

( ) Shift the bits in that variable so that the `A` component is in bit 0 and the `B` component is bit 1, and return the shifted variable.

We will track the movement of the rotary encoder's shaft using the state machine depicted in Figure 2. The convention we will use is that the states are named after the logic values of the quadrature's two components: *`B_A`*. For example, if the bits returned by **`get_quadrature()`** are 0b0000'0010 then the state machine should be in state `HIGH_LOW`.

Locate the **`initialize_rotary_encoder()`**

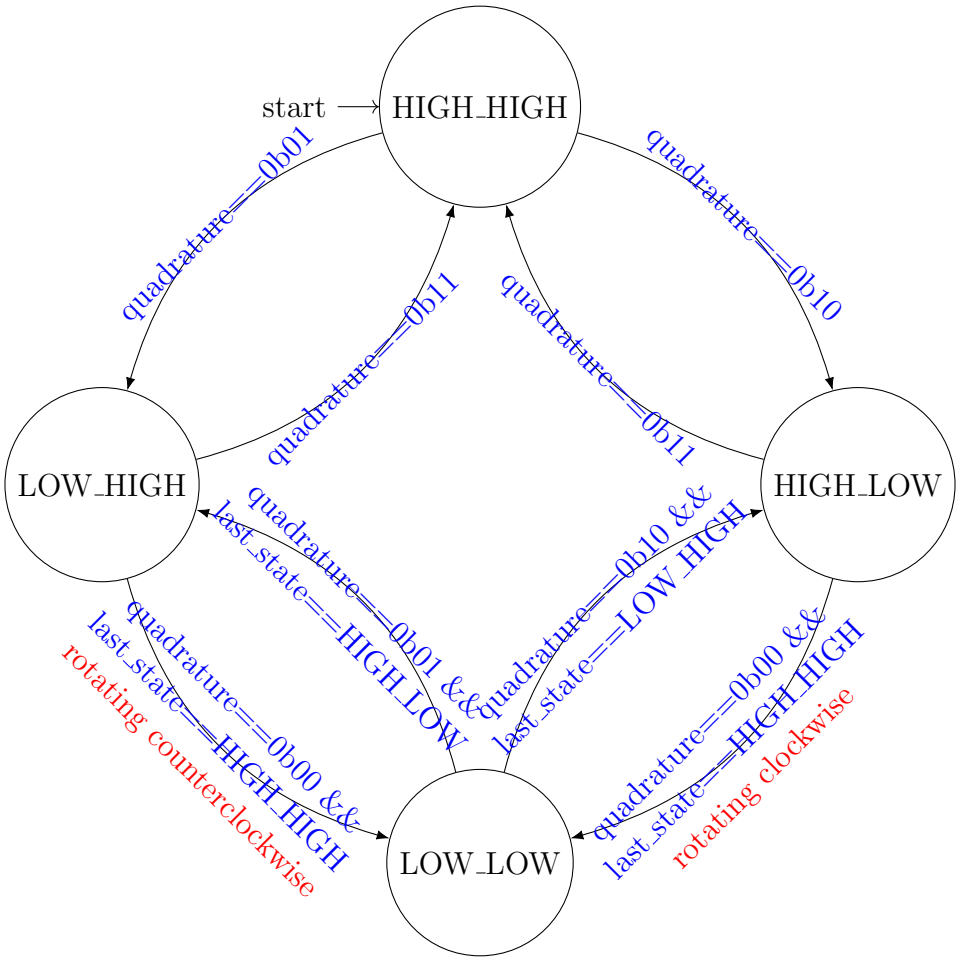


Figure 2: State machine to determine a rotary encoder's direction of rotation.

( ) Initialize the `state` variable.

For test mode, we'll use `clockwise_count` and `counterclockwise_count` to count the number of times that the shaft is turned in each direction for Requirement 1. For the combination lock, we'll use `direction` to track which direction the shaft was turned. Once the direction has been provided to the lock controller, we want to set `direction` back to `STATIONARY` so that a single turn isn't interpreted as many turns.

Location **`count_rotations()`**.

( ) Add code to populate the buffer with the counts as required by Requirement 1. Clearly label which count is which.

- Due to the limited amount of space on the display, you'll probably want to abbreviate, such as "CW" for clockwise and "CCW" for counterclockwise.

( ) Compile and upload your code, and confirm that a descriptive string is displayed with the counts (both of which should be 0 for now).

Locate **`get_direction()`**.

( ) Add code to save a copy of `direction`.

( ) Add code to set `direction` to `STATIONARY`.

( ) Return the copy of `direction`.

Locate **`handle_quadrature_interrupt()`** and examine the state machine depicted in Figure 2. You will implement the state machine in **`handle_quadrature_interrupt()`**, but first make sure that you understand how the state transitions driven by the quadrature signal in Figure 1 allow us to determine the direction of rotation.

But what about debouncing?

You *should* be wondering about debouncing. After all, the rotary encoder relies on mechanical contacts. In fact the `A` and `B` wipers *do* experience switch bounce. Fortunately, the state machine provides us a way to implement debouncing without using **`cowpi_debounce_byte()`** or **`debounce_interrupt()`**.

Consider the transition from `HIGH_HIGH` to `HIGH_LOW`:

- This could be due to the user turning the shaft clockwise, causing the `A` component to drop from 1 to 0, or
- This could be due to switch bounce
  - The user has turned the shaft counterclockwise
  - The last quadrature change from that action is the `A` component rising from 0 to 1

  - Switch bounce has caused the `A` component to drop from 1 to 0

Now consider the transition from `HIGH_LOW` to `HIGH_HIGH`:

- This could be due to the user rotating the shaft counterclockwise, causing the `A` component to rise from 0 to 1, or
- This could be due to the `A` wiper bouncing back as part of the aforementioned switch bounce

The reasoning is similar for transitions between `HIGH_HIGH` and `LOW_HIGH`.

Now consider the transition from `HIGH_LOW` to `LOW_LOW`. Continuing the clockwise turn, the `B` component drops from 1 to 0. If the change in `B`'s value were due to switch bounce, then we shall deny the state transition. How? By remembering which state the state machine was in before it was in `HIGH_LOW`.

If the previous state was `HIGH_HIGH` then `we know that the shaft is turning clockwise`. On the other hand, if the previous state was `LOW_LOW` then we know that the `B` wiper is bouncing. Thus, we will only allow a transition from `HIGH_LOW` to `LOW_LOW` if the previous state was `HIGH_HIGH`. We will similarly limit all transitions into and out of `LOW_LOW`.

(We cannot similarly limit the transitions into and out of `HIGH_HIGH` because after the shaft stops at the detent, the user can continue to turn the shaft in the same direction, or they could turn the shaft in the opposite direction.)

( ) Implement the state machine from Figure 2 in **`handle_quadrature_interrupt()`**.

( ) Add code to the transitions into `LOW_LOW` to update `clockwise_count`, `counterclockwise_count`, and `direction` as appropriate.

Locate the **`initialize_rotary_encoder()`** function:

( ) Uncomment this line:
```
// register_pin_ISR((1 << A_WIPER_PIN) | (1 << B_WIPER_PIN), handle_quadrature_int
```

( ) Compile and upload your code, and confirm that the clockwise count and counterclockwise count update on the display as specified by Requirement 1.

- It is acceptable for the counter to fail to update for the occasional turn if you turn the shaft quickly, but slow turns should always register, and most fast turns should register.

5 Controlling the Servomotor

5.1 Theory of Operation

The servomotor is controlled with a signal that consists of periodic pulses. The desired position of the servomotor is encoded as with width of each pulse. This encoding, *pulse*

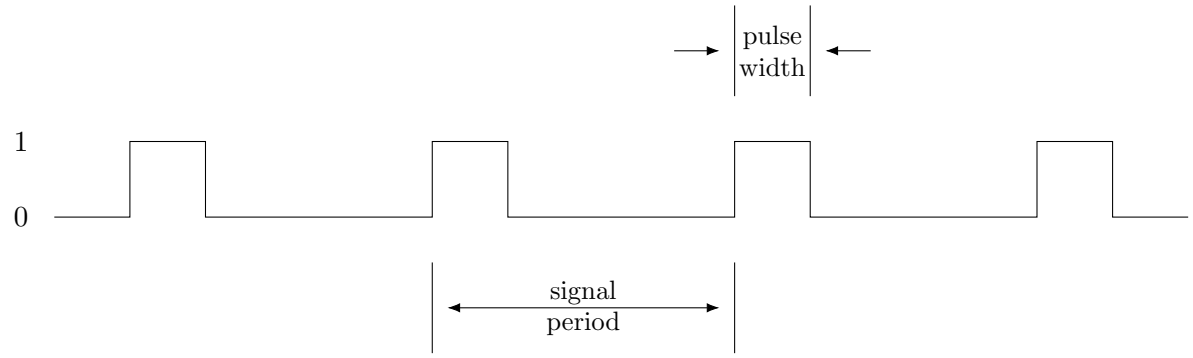


Figure 3: A PWM signal periodically sends a pulse. The *pulse width* encodes the information being transmitted; the *signal period* is how often the pulse is sent.

width modulation (PWM) is sufficiently useful that most microcontrollers can be configured to generate such a signal by setting a few of their memory-mapped I/O registers. We will not do that. Instead, we will directly form the signal by setting an output pin to 1 or 0 as needed. Figure 3 shows the relevant parameters of a PWM signal.

A servomotor expects a pulse to be sent every 20ms (20,000μs); this is the *signal period*. The *pulse width* can vary between 500μs and 2500μs. A 500μs pulse directs the servomotor to rotate fully clockwise, and a 2500μs pulse directs the servomotor to rotate fully counter-clockwise. The angle varies linearly with the pulse width (for example, 1500$\mu$is the central position).

5.2 Changes to *servomotor.c*

Open *servomotor.c*.

Locate these lines of code:

```
1 #define PULSE_INCREMENT_uS   (INT32_MAX)
2 #define SIGNAL_PERIOD_uS     (INT32_MAX)
```

( ) For `SIGNAL_PERIOD_uS`, replace `INT32_MAX` with the period (measured in microseconds) that the servomotor requires of its signal.

( ) Select a factor common to the signal period and to each of the pulse widths necessary to realize Requirements 2–3.

( ) For `PULSE_INCREMENT_uS`, replace `INT32_MAX` with that factor (measured in microseconds). This will be your timer interrupt period.

Locate the **`test_servo()`** function.

( ) Add code to call the **`center_servo()`**, **`rotate_full_clockwise()`**, and **`rotate_full_counterclockwise()`** functions as necessary for Requirements 2–3.

( ) Add code to populate `buffer` with the strings required by Requirement 4.

( ) Compile and upload your code, and confirm that the correct strings are displayed.

Locate the **`center_servo()`**, **`rotate_full_clockwise()`**, and **`rotate_full_counterclockwise(`** functions.

( ) Populate these functions with code to set the value of `pulse_width_us` to the necessary pulse width as necessary to encode the servo's desired rotational position into the pulse width.

Locate the **`handle_timer_interrupt()`** function.

( ) Add variables to track the time until the next rising edge of the pulse and the next falling edge of the pulse.

( ) Add code so that when the time until the next rising edge is 0μs:

- Start the pulse by setting output pin 22 to 1.
- Update your variable that tracks the time until the next rising edge, to the time until the *next* pulse should start.
- Update your variable that tracks the time until the next falling edge, to the time until *this* pulse should finish.

( ) Add code so that when the time until the next falling edge is 0μs:

- Finish the pulse by setting output pin 22 to 0.

( ) Add code to update the time remaining until the next rising edge and the time until the next falling edge, to reflect the time that has elapsed between timer interrupts.

Locate the **`initialize_servo()`** function:

( ) Uncomment this line:
`// register_periodic_timer_ISR(0, PULSE_INCREMENT_uS, handle_timer_interrupt);`

Pre-emptive debugging. Consider:

( ) Multiply the definitions of `SIGNAL_PERIOD_uS` and `PULSE_INCREMENT_uS` by 1000.

( ) Multiply the assignments to `pulse_width_us` by 1000.

( ) In **`handle_timer_interrupts()`**, replace the updates to pin 22 with updates to pin 20.

This will have the effect of replacing "microseconds" with "milliseconds" and of replacing the servomotor with the right LED.

( ) Compile and upload your code, and confirm that the correct right LED lights up and dims in the correct pattern, considering that you have replaced "microseconds" with "milliseconds".

( ) Restore the original definitions, assignments, and pin number.

( ) Compile and upload your code, and confirm that the servomotor rotates to the correct positions as specified by Requirements 2–3.

6 Combination Lock Specification

If the **right switch** is in the *right position* when the system boots, then the system shall enter its combination lock mode.

When the system is in its combination lock mode:

5. A combination shall consist of three numbers in a particular order.
   (a) Each of the three numbers shall consist of exactly two decimal digits.
   (b) It shall be possible for any two of the numbers to be duplicates of each other. It shall be possible for all three numbers to be duplicates of each other. For example, 12-12-12 is a valid combination.
   (c) Single-digit numbers must have leading zeroes. For example, 01-02-03 is a valid combination, but 1-2-3 is not.
   (d) Each of the three numbers can assume any value between 00 and 15, inclusive.
      - We are artificially limiting the range of possible values to simplify verifying the solution's correctness.
6. When displayed on the **display module**, the combination's first number shall occupy the two leftmost digits; the second number shall occupy the middle two digits; and the third number shall occupy the two rightmost digits. Dashes shall be displayed between the numbers, for example: `12-03-10`. If one or more of the positions has not had a number entered, then the digits for the unentered positions(s) shall be blank. For example, if the first number has been entered but not the second and third, then the display will show `12-  -  `.

7. The combination's numbers shall be entered using the **rotary encoder**.
   (a) When entering a number, the initial number shall be the last number that had been entered. If no number had previously been entered, then the initial number shall be 00.
   (b) When the **rotary encoder** is turned *clockwise*, the number shall increase. If the number increases past 15, it shall overflow to 00.
   (c) When the **rotary encoder** is turned *counterclockwise*, the number shall decrease. If the number decreases past 00, it shall underflow to 15.
8. When the system is powered-up, and after the system is reset, it shall be LOCKED, regardless of its state before losing power, and the display shall have all numbers blank: – – .
9. When the system is LOCKED, the **left LED** shall be *on*, and the **right LED** shall be *off*.
10. When the system is LOCKED, the **servomotor** shall be turned fully *clockwise*.
11. When the system is LOCKED, the system shall display the combination-entry display.
12. When the system displays the combination-entry display, it shall display the currently-entered combination (which might not be the correct combination), and the user shall be able to enter and change the currently-entered combination.
    (a) The first number shall be entered by turning the **rotary encoder** *clockwise*.
    (b) After entering the first number, when the user turns the **rotary encoder** *counterclockwise*, the first number shall be fixed, and the second number shall be entered.
    (c) The second number shall be entered by turning the **rotary encoder** *counterclockwise*.
    (d) After entering the second number, when the user turns the **rotary encoder** *clockwise*, the second number shall be fixed, and the third number shall be entered.
    (e) The third number shall be entered by turning the **rotary encoder** *clockwise*.
    (f) After entering the third number, if the user turns the **rotary encoder** *counterclockwise*, then the entered combination shall be cleared.
    (g) After entering the third number, if the user presses the **left pushbutton**, then the entered combination shall be evaluated.
13. The entered combination is *correct* if and only if the following are true:
    (a) The first number is the correct first number, the second number is the correct second number, and the third number is the correct third number.

(b) The first number was entered by passing the correct first number at least twice and then stopped at the correct first number on the *third* (or greater) time that number had been visible.

(c) The second number was entered by passing the correct second number once and then stopped at the correct second number on the *second* time that number had been visible.

(d) The third number was entered by stopping at the correct third number the *first* time that number had been visible.

14. If the entered combination is correct, then the system shall be UNLOCKED.

15. When the system is UNLOCKED, the **right LED** shall be *on*, and the **left LED** shall be *off*.

16. When the system is UNLOCKED, the **servomotor** shall be turned fully *counterclockwise*.

17. When the system is UNLOCKED, the system shall *not* display the combination but instead shall display "OPEN".

18. If the entered combination is incorrect, a warning message shall be displayed, both LEDs shall blink twice, the entered combination shall be cleared, and the system shall remain LOCKED.

    - The warning message shall be "bad try 1" or "bad try 2", depending on whether it was their first or second attempt.
    - The blinks may be implemented with timed busy-waits.

19. If the user makes three incorrect attempts, the system shall be ALARMED.

    (a) When the system is ALARMED, it shall display `alert!` and both **LEDs** shall blink on-and-off every quarter-second.

        - The blinks may be implemented with timed busy-waits.

    (b) When the system is ALARMED, it shall not accept further input until it has been reset or has been powered-down and then powered-up.

20. Optionally, a partially-entered combination can be abandoned, causing the partially-entered combination to be cleared, under these conditions:

    - When entering the second number, 00 is displayed three times
    - When entering the third number, 00 is displayed twice

21. When the system is UNLOCKED, the user shall be able to change the combination.

(a) When the user places the **left switch** in the *right position* and then presses the **right pushbutton**, the system shall be CHANGING.

(b) When the system is CHANGING, it shall display `enter` and the combination-entry display, and the user shall be able to enter the proposed new combination by pressing six digits on the **numeric keypad**.

(c) After the user has entered a new combination, a second combination-entry display shall be shown, and the user shall be able to re-enter the proposed new combination by pressing six digits on the **numeric keypad**.

(d) When the user places the **left switch** in the *left position*:

i. If either of the two entered combinations are incomplete, the system shall display "no change".

ii. If the two entered combinations do not match, the system shall display "no change".

iii. If any of the numbers in the combination is greater than 15, the system shall display "no change".

iv. Otherwise, the system shall display "changed", and the correct combination shall change to the new combination.

v. The system shall be UNLOCKED.

22. When the system is UNLOCKED, the user shall be able to lock the system by pressing both **pushbuttons** simultaneously.

- After doing so, the system shall be LOCKED.

23. The correct combination shall be persistent even when the system is reset. For example, if the user changes the combination and presses the **RESET button**, then the correct combination shall still be the combination that they had changed it to before resetting the system.

7 Implementing the Combination Lock

Open *lock-controller.c*.

7.1 Limited Starter Code

The starter code in *lock-controller.c* is sparse: an array to store the combination, a function to retrieve the combination, and a function to force the combination to take on a pre-defined setting (think of it as a "factory reset"). There are also stubs for two additional functions.

For your lock controller, you won't need to use **`get_combination()`** because you can directly access the `combination` array, and you should not use **`force_combination_reset()`** because it serves no purpose in the context of the system's specification.

The functions you *will* use are **`initialize_lock_controller()`** and **`control_lock()`**, as well as any additional functions that you write and call from these functions. When the system is in its combination lock mode, the effective **`main`** code looks like:

```
1    int main() {
2        initialize_rotary_encoder();
3        initialize_servo();
4        initialize_lock_controller();
5        for(;;) {
6            control_lock();
7        }
8    }
```

(If you look at *combolock.c*, you'll see that there's more to it than that, but this simplification is enough to make the point that...) Any code that you want to run *once* should go in **`initialize_lock_controller()`**, and any code that you want to run repeatedly in the main control loop should go in **`control_lock()`** (or be called by **`control_lock()`**.)

7.2 Storing the Combination Between Resets, and Resetting the Combination

As long as you store the correct combination in the `combination` array (and store the user entries in other arrays), you will have satisfied Requirement 23.

An earlier version of this assignment used a different microcontroller that had a small EEPROM built into it, and so students were able to save the combination and access it even after power had been removed and restored. The RP2040 microcontroller doesn't have a built-in EEPROM for data, so we will get as close as possible to persisting the combination.

The declaration of the `combination` array includes a directive that it should be placed in memory that doesn't get initialized during a system boot. The effect of this is that, as long as the microcontroller continues to have power, the `combination` array will retain its contents even if the microcontroller gets reset.

What if the microcontroller loses power?

If the microcontroller loses power, the combination *will* change. If you place the system in its test mode, then the currently-stored combination will be displayed. Because the combination after restoring power might not have numbers that are strictly less than 16, if you press the **right pushbutton** while in test mode, the combination will reset to 05-10-15.

7.3 Recommendations

I recommend that you use polling to get human-scale inputs (*i.e.*, button & key presses and switch toggles). I recommend that you use the functions from the CowPi library to do so (**`cowpi_get_keypress()`**, **`cowpi_left_button_is_pressed()`**, etc.) Debounce these

inputs if you need to – see the PollingLab code for examples of how to use the debounce functions, and/or read the datasheet (https://cow-pi.readthedocs.io/en/latest/CowPi/inputs.html#debouncing).

You can implement blinking using timed busy-waits, similar to the busy-wait you placed in **`get_keypress()`** in PollingLab, but considerably longer than 1μs. Other than when you're blinking the LEDs, the system must be responsive to valid user inputs. (You may ignore invalid inputs.)

Use **`display_string()`** to send strings to the display module.

You do **not** need to set up globally-shared variables to communicate between *lock-controller.c*, *rotary-encoder.c*, and *servomotor.c*. If you believe that you do, see me and if I agree that you do then I'll show you how to set that up. However:

- The lock controller should not need to expose any information to the rotary encoder and the servomotor.
- The lock controller needs to find out which direction the rotary encoder's shaft had been turned, if it had been turned. This can be obtained through the **`get_direction()`** function.
- The lock controller needs to instruct the servomotor to turn one direction or another. This can be accomplished through the **`rotate_full_clockwise()`** and **`rotate_full_counterclockwise()`** functions.

8 Turn-in and Grading

When you have completed this assignment, upload *lock-controller.c*, *rotary-encoder.c*, and *servomotor.c* 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.[3] 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

[3] 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.

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 tentatively worth 50 points. (I might increase it to 60.)

TODO

Students' scores may be adjusted up or down as necessary if the team had an inequitable distribution of effort.

Epilogue

After fastening the new electronic combination lock to the lab door, 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 if you complete your assignment early and get it checked-off by a TA or the professor during office hours.

TODO

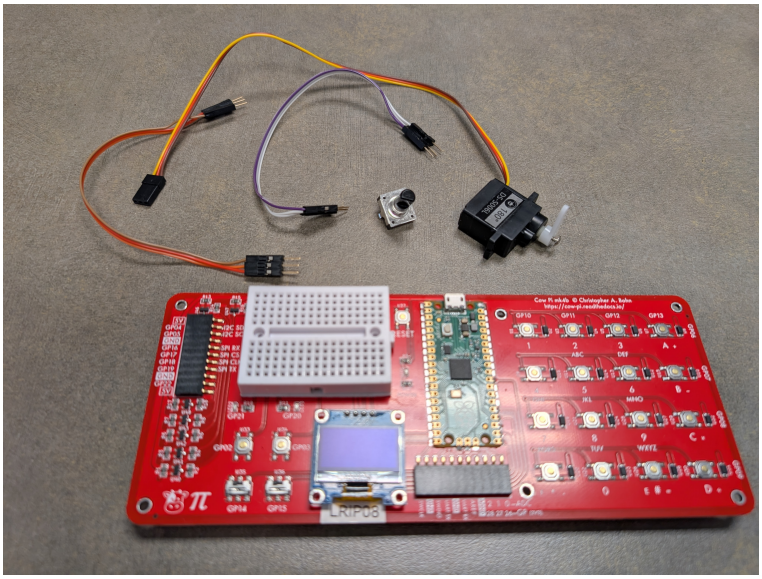


Figure 4: Components needed for the Range Finder

B Installing Necessary Hardware Components

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 4 shows the components you will need for the electronic combination lock.

You will need:

- Your Cow Pi hardware circuit
- A rotary encoder
- A servomotor
- Six 20cm male-to-male wires

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.

B.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 5). Each group of five is a *terminal strip*.

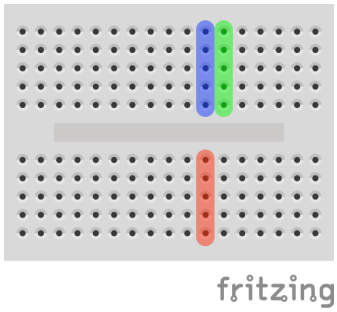
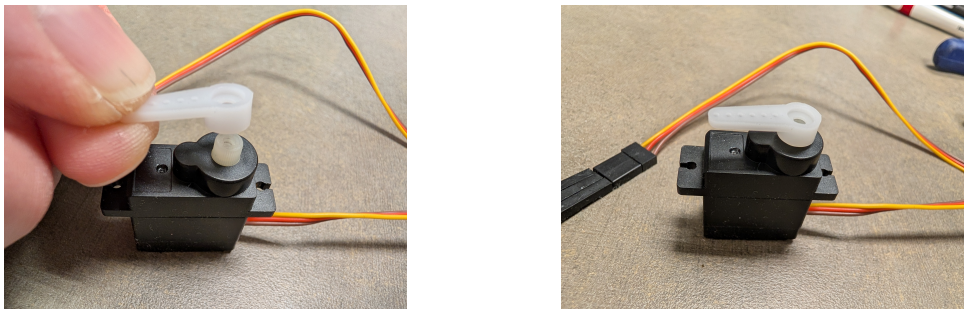


Figure 5: Terminal strips on a mini-breadboard



(a) A servo arm can be attached to a servo motor. (b) A servo arm has been attached to a servo motor.

Figure 6: Attaching a servo arm to a servo motor.

The five holes in a terminal strip are electrically connected to each other but are electrically isolated from the other terminal strips.[4] For example, in Figure 5, 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.

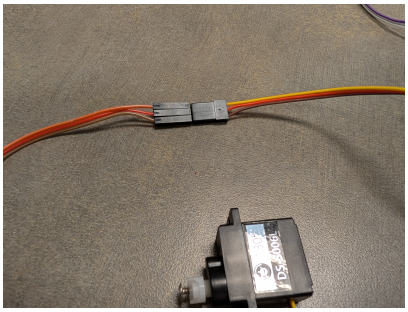
A consequence of this is that any components' connectors that are inserted into a terminal strip are connected to the connectors of other components that are inserted into the same terminal strip.

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

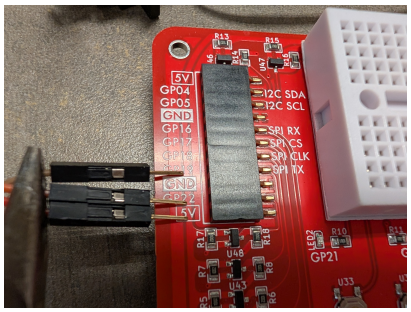
B.3 Connecting the servomotor

If your servomotor does not already have a servo arm attached, attach a servo arm to the motor's shaft. See Figure 6. The orientation does not matter since we will not connect anything to the servo – we are simply using the servomotor to simulate a lock's deadbolt mechanism, and the arm will allow us to see the motion. *Do not screw the arm to the motor's shaft.* Simply let it fit snugly.

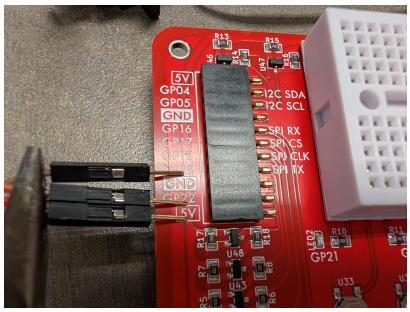
[4] They are isolated for DC signals, and parasitic reactance is negligible for AC signals below about 10 kHz.



(a) Attaching 20cm wires to servo wires.



(b) Determining where to insert each wire.



(c) All servo wires connected.

Figure 7: Wiring a servo motor to the Cow Pi.

( ) Attach three to the servo's wires (Figure 7a). The colors do not need to match, but you do need to pay attention to which 20cm wire you attached to which servo wire.

( ) Locate the 20cm that is attached to the servo's **red** wire. Insert the other end into a `5V` slot.

( ) Locate the 20cm that is attached to the servo's **brown** wire. Insert the other end into a `GND` slot.

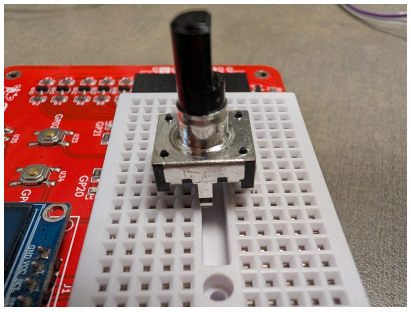
( ) Locate the 20cm that is attached to the servo's **brown** wire. Insert the other end into the `GP22` slot.

( ) Have someone verify that you have each wire inserted into the correct slot.

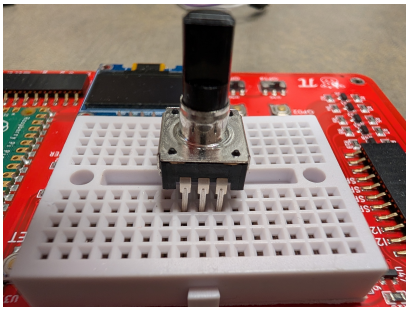
The servo motor is now connected to the Cow Pi's `GP22` pin. The starter code will configure `GP22` to be an output pin.

B.4 Connecting the Rotary Encoder

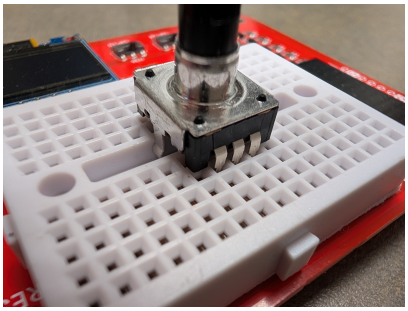
( ) Position the rotary encoder on the breadboard so that the prongs on either side of the encoder will fit in the breadboard's gutter (Figure 8a), and with the three pins positioned in the wells of three contact points (Figure 8b).



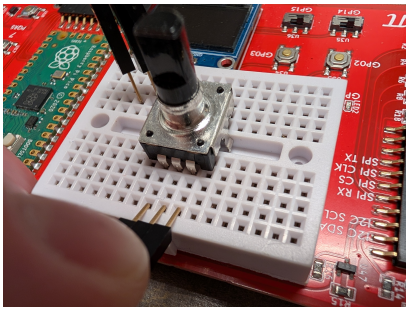
(a) Position the prongs into the breadboard gutter.



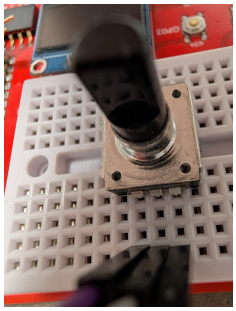
(b) Placing the pins in contact point wells.



(c) The rotary encoder in the breadboard.

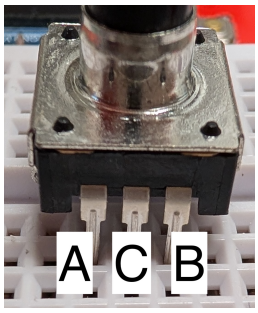


(d) Lining up the wires.

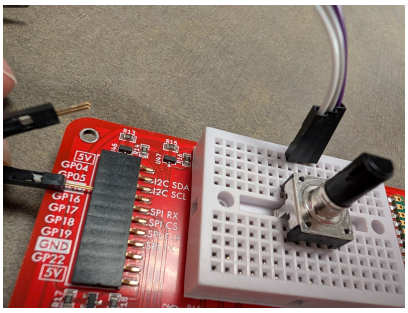


(e) Wires connected to the rotary encod

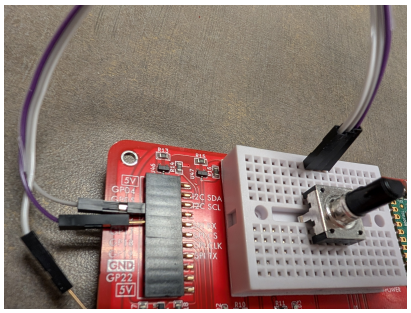
Figure 8: Inserting the rotary encoder in the breadboard.



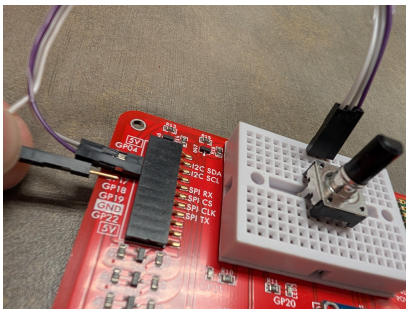
(a) Labeled pins



(b) Inserting the ground wire



(c) Inserting the A wiper's wire



(d) Inserting the B wiper's wire

Figure 9: Wiring a rotary encoder to the Cow Pi.

( ) Gently but firmly, press on the rotary encoder until the bottom of the encoder is flush with the top of the breadboard (Figure 8c).

( ) Insert three 20cm wires, one into each of the same terminal strips that the rotary encoder uses (Figure 8d–8e). Make a note of which color wire corresponds to which of the sensor's pins. ⋮

( ) 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`

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

The three pins are connected to the rotary encoder's `A` and `B` wipers, and to a `C`ommon ground. See Figure 9a for a labeling of the pins.

( ) Locate the 20cm that is attached to the rotary encoder's **`C`** pin. Insert the other end into a `GND` slot (Figure 9b).

( ) Locate the 20cm that is attached to the rotary encoder's **`A`** pin. Insert the other end into the `GP16` slot.

( ) Locate the 20cm that is attached to the rotary encoder's **B** pin. Insert the other end into the `GP17` slot.

( ) Have someone verify that you have each wire inserted into the correct slot.

The rotary encoder's quadrature can now be read through the Raspberry Pi Pico's pins `GP16`–`GP17`. The starter code will configure `GP16` & `GP17` to be pulled-up input pins.

Your Cow Pi is now ready for you to design and code the software for an electronic combination lock.

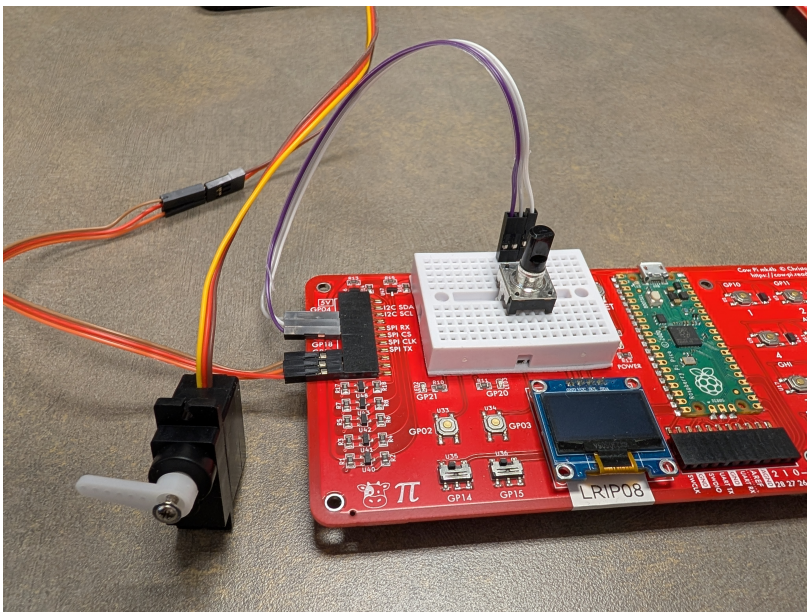


Figure 10: A fully-assembled combination lock.