Lab 3

Part 0

For part 0, I first looked into what algorithms to use to check for primes less than n. I choose to use the Sieve of Eratosthenes algorithm. It basically uses an boolean array of size n, that starts with all values false. You check all numbers less than √n. If their array value is true, then you increment the prime counter and mark all multiples of that number as false. Then, you traverse the back end of the array (all the indexes greater than √n), and if their corresponding value in the array is true, increment the primes counter. I chose to use this algorithm because it seemed to be the most efficient algorithm and simple algorithm out there for calculating the prime numbers smaller than a given n.

Design decisions to avoid unsafe rust features

To safely utilize multiple threads in my program, I protected the global counter "primes" and the "sieve" array by using the "Arc" and "Mutex" data types in rust. The Arc data type helps you to do safe reference counting, where multiple pointers can exist to the same reference. Then, the Mutex data type makes sure that only one thread is editing the data in the mutex at each time. Each thread only accessed the "primes" global counter at the end of their lifespan, to add their own local value to it. Each time access to the "sieve" array was not needed, it was "dropped"—meaning that the thread gave up its lock on the variable—so that another thread could use it. Using the recommendation from this Ed post, I used the nth thread to check if the nth number was prime or not. Then the number of "nth" primes less than the number provided to the program would be added to the global counter just before the thread ends. Then all threads are "joined" and the main thread waits for all other threads to finish.

I think the most difficult part of part 0 was figuring out how to manage ownership issues, which I have never had to deal with in C before. The most common error I kept getting was "value moved", when I tried to get a pointer to the mutex in each thread. I then realized that I had to get a reference to the global variable outside of the thread (which takes ownership of all values it uses given the move keyword), and then give that variable to the thread. Because I was using the value "primes" to get its reference, the thread was taking ownership of this value. It took me a bit to get used to these kinds of concerns when writing code.

Source Code for Part 0

For choosing and implementing the algorithm in Rust, I used the Sieve of Eratosthenes implementation of a prime number checker in this source. Since the implementation I modeled mine off of is in C++, the actual lines of code are different, but their meaning/function is the same. I also changed this code to make it more friendly to using multiple threads, by adding threads that performed the algorithm in the previous source on every nth number. I found the code for multi-threading and safe variable sharing in Rust in the Shared Concurrency section of the Rust Programming language book. I used their "Atomic Reference Counting with Arc<T>" example and replaced the code they had in their thread to "increment a counter" with the code from the previous source that checks if a number is prime. From the Accepting Command line arguments section in the Rust Programming book, I got lines 10-12 of my code for getting access to command line arguments like "n" and the number of threads the program should use. I lastly used this stack overflow poset to learn how to convert the string args to numbers for use in the rest of the program.

Part 1

The first challenge of Part 1 was figuring out how to get the example code given burned on to the board. Openocd was not working so I had to use a modification of the burn command used in part 2 of the lab. After compiling the code, I then called "cargo objcopy" to produce a raw binary file (similar to the one used in part 2) from the compiled code files. Then, I wrote this file, using the same command as in the makefile in Part 2, to the flash.

Another part of the initial set-up was editing the memory.x file to match the board specifications, since this stm32f407g board has different specs then the board I was using.

After that, I chose to use the hal interface used in the example code when writing my part1 code because I thought I would be more straightforward. One of the issues I ran into here was that many versions of the stmf4xx_hal crate have been released, with slight tweaks to the interface, so example code using one version of the stmf4xx_hal crate can not necessarily be combined with code using another version of that same crate and produce good results.

My general code structure was very similar to my Lab 2 part 1 code, except I chose to use systik instead of a timer interrupt to flash the leds. This allowed me to avoid putting the leds into

a mutex, which would have been necessary for the safe usage of those components. First I got an instance of the stmf4xx_hal and cortex_m peripherals, initialized the syscfg clock, and set the clock speed to 100mhz. Then I initialized the GPIOD pins corresponding to the leds and the PA0 (the pin connected to the user button). Connecting the EXTI0 to PA0, enabling the interrupt, and setting it to trigger on both rising and falling edges was actually quite straightforward with the hal_interface. I also had to set up my "delay" method via systik, which with the hal interface was only one line. Lastly, I had to enable and set the priority of the interrupts through NVIC, a cortex-m peripheral. Then, I used similar logic to Part 2 and used two global variables—playing and my_color—to control the state of the LED playing. When the EXTI0 handler registered 1 click (i.e. the button has been pressed), the handler changes the "playing" variable to false and the sequence stops. When a second press has been registered (i.e. the button has been unpressed), the "playing" is set to true and the "my_color" is set to a reset state, which turns off all LEDs, turns on the green LED and updates the state.

Reduction of unsafe rust features

First, I put all variables that needed to be accessed in more than one function in Mutexes. This ensured that only one process was accessing each shared resource at a time, which prevents race conditions. Since using mutexes can be cumbersome, I also use a local static variable—STATUS—to track the state of the button instead of using a global variable. This allowed the value to persist between function calls without potentially having another process access it and cause a race condition.

Explanation of lines 1-18 of the github example

The first line tells Rust not to use the standard main function as the entry point of the program. This is because Rust makes assumptions about the environment the standard main function is executed in. For example, Rust assumes that command line arguments exist for the main function, which would not work for code that was going to be run on the stm32f411 board (The Embedomicon). Because of this, the entry point of the program must be manually specified later.

The second line tells Rust that it should not load the standard library for this program. It is usually loaded automatically to deal with things like processing command line arguments and

creating the thread that will run the main method. To execute these types of processes, Rust makes the assumption that the environment that the program is being executed in has an OS, which on our stm32f411, is not the case. This causes the "core" library to be loaded instead which helps the code explicitly do some of the things that the std library would have taken care of in a normal environment(The Embedomicon, The Embedded Rust Book).

The next line sets the panic handler for the rust code. Rust panics when some process it's runtime checking for memory safety (ie array indexing) does something it's not supposed to do. Normally, the std library has a routine way of dealing with this, but since we didn't link the standard library to this code (per line 2), the code has to say what function should be called when a panic is raised "use panic_halt as _" means that this code will use the panic handler from the panic halt crate. This means that when the code panics, it will stop executing and enter an infinite loop. The reason the syntax is "use panic_halt as _;" is because you need to let the compiler know that it should include the panic handler in panic_halt in the code even though the import is not used in the rest of the code. This keeps the compiler from mistaking it as an unused import (The Embedded Rust Book).

The next line attaches "stm32f407g_disc" to the name "board". So now whenever you use "board" in the code, the compiler knows that you are referring to the "stm32f407g_disc" crate. This can help shorten long paths of things nested deep in large crates (Rust by Example). The stm32f407g_disc crate has functions, classes, and structs that contain information about the specifics of that board that can be used in the code.

Lines 8-12 do a similar thing to the previous line, it binds various shorter names to different things in the stm32f407g_disc crate, which ordinarily would have had to have been accessed through long paths. The keywords "use crate" instead of just use specifies that the path bound to the following names should be an absolute path instead of the relative path from where the binding is taking place. The curly brackets allow multiple items in the board section of the crate to be bound to different convenient names for later use in the code. For example, now the code has access to the Led class without having to specify the full path to the class in the board crate. Led class and LedColor can be used for initializing and lighting up the leds. The stm32 contains the hal interface information specific to that board, kind of like an advanced c header file. The Delay binding is later used to set up a systik based timer. Lastly, prelude::* allows for all "helper traits" of the things importanted by this crate to be imported without explicit

specification. For example, you don't have to worry about importing ".split()", the method used to get access to the hal GPIO representations (stm32f4xx_hal documentation).

Line 14 allows you to use the cortex_m peripherals by simply using "Peripherals". This gives you access to systik and NVIC methods, for example.

Line 16 imports the entry attribute from the cortex_m_r crate. This allows the main method to be declared as the entry point of the program by #[entry] on line 18. This crate, however, does require a memory.x file, which specifies the memory layout of the environment the code will be run on. This crate does a lot of the work that std would usually do for bare metal environments.

Source Code for Part 1

My major piece of source code for part 1 was this stm32f4xx_hal implementation of a stopwatch using the user button. From this code, I used their code for clock set-up (but set the clock to 100 mhz), PA0 set up, and EXTI0 interrupt and handler set up. I also used some of their mutex code and generally used their code as a model for how to set up and access data in mutexes. I additionally used their code to set up a systik based delay function used when blinking the LEDs. I added the LED blinking functionality, the global mutex-ed variables about board status, and the code to pause and reset the blinking. Although I didn't use other bits of this code directly, it helped me to write my LED blinking code and learn about the hal interface.

From this same repository, I also copied the relevant dependencies from their Cargo.toml file into mine. I used those unchanged.

I used and modified the memory.x file from the example repository given. Based on the board specs, I changed the flash from 1M to 512KB to fit the specs of my board and also removed the CCRAM part, since my board also doesn't have that.

I took the .config file from the example repository and used it unchanged.

Lastly, the makefile for this part used the debug and burn commands used in the armf4 makefile from lab 2. See Lab 2 Part 1 for more details on the source from that. I added a new build command that compiled the rust code and then copied the compiled code into a raw binary file, which could be flashed onto the board.

Part 2

For this part, the hal interface was no longer nicely suited for the problem. Since in my design you had to be able to change the mode of the pin and access it globally, I decided to just use the stmf4 crate and implement the code similar to how I did it in lab 2 part 2. I first get a copy to the stmf411 and coretx_m peripherals and initialize PC7 and the LEDs (PD12-PD15) to inputs and outputs respectively. Then, I enabled the SYSCFG, GPIOC, and GPIOD clocks. I set up the EXTI7 interrupt by linking it to GPIOC, unmasking the interrupt, and setting the trigger to falling edge. I then enabled the interrupt from NVIC's perspective, and set its priority. I then initialized the timers by setting their prescaler values, setting their auto refill values, and enabling them. One timer expired every 0.5 seconds and was used to blink the LEDs. The other timer expired every 0.3 seconds and was used to refresh the charge on PC7. When a finger touched PC7, the charge would discharge quickly, triggering a falling edge interrupt. The interrupt handler then paused the LED sequence on the first tap and restarted it on the second.

One of the issues I ran into here was that when you use the .write() method, bits that you haven't "mentioned" in the arguments of the .write method get affected. For example, gpiod.odr.write(|w| w.odr15().clear_bit()) will also clear the other odr bits. So when you use this method, you have to make sure if you want to enable 2 bits in one register, you have to use 1 call to write, or the previous one will be overwritten by the second.

Reduction of unsafe rust features

Like in Part 1, I used mutexes to allow multiple functions to have access to peripherals and global variables that they needed without using unsafe global mutable static variables. This means that only 1 piece of code can access the variable at a given time, which prevents a race condition. For state tracking, like in the EXTI9_5 handler, I used the local static variables STATUS. It was easier to deal with than a global variable in a mutex, and it allowed the value to persist between function calls.

Source code for Part 2

I used 3 main sources to write my Part 2 code, a blog post on interrupts, the Discovery book's section on one shot timers, and my Part 1 code. From the blog post on interrupts, I used their method of getting instances of both peripherals (the stm32f4 and the cortex_m), enabling

GPIO and SYSCFG clocks, connecting EXTI7 with GPIOC, unmasking the interrupt, and enabling the interrupt. In the interrupt handlers, I also used the blog post's method of clearing the pending bit. From the one shot timers section, I used their code to enable the timer clocks, set the prescaler and auto refill values, enable the timers, and clear the pending interrupt bit in the handlers. I took the method of enabling the interrupts and setting their priorities through the NVIC directly from my part 1 code, as that didn't depend on the hal interface. I also directly used the state machine in my EXTI0 handler in my EXTI9_5 handler here. I also took the logic of the LED blinker state machine, and moved it to a timer interrupt handler and re-wrote lines (like turning off and on the leds) without the help of the hal interface.

My memory.x and .config files were directly taken from my Part 1 code. I also took the makefile from Part 1 and just changed the target from part1_cargo to part2_cargo. I took the Cargo.toml file from my Part 1 code and removed the hal import.