Lab 4

Part 1

For part 1, I first researched the cortex_m crate to see if there was any support for safe pushing/popping to the stack. Although I didn't find anything to help me safely do that, I did find a cortex_m method that safely reads that main stack pointer, which I used in my code instead of reading the value within my unsafe assembly blocks to try to minimize my use of unsafe code. Then I used the programming manual to determine what instructions I needed to use to set up/operate and their syntax. One issue I came across quite early in the process was that 3 sets of curly braces were needed to format the "reglist" operand for the PUSH and POP when using the "in/out" register feature in the inline asm. For reading and writing to MSP, I used the MRS and MSR instructions respectively, as the MSP is a special register and therefore requires special instructions for reading and writing.

After finding the right instructions to use for operating the stack, I then had to choose a location for my initial stack pointer.

This was probably the most difficult part of Part 1 for me. At first, I thought that the range of SRAM we could use was from 0x24000000 to 0x3FFFFFFF as I didn't realize at first that the memory map on page 28 of the programming manual dictated what type of memory every possible address that could be created with 32 bits would point to. Once I realized this, and that on the board we use actually only had 128 kB of SRAM, it became clear that the actual range of addresses was 0x20000000 (the start of the SRAM addresses) to 0x20020000, which is 128 kB worth of addresses. Then, I picked 0x20000000 as my stack address. However by the time the interrupt handler was called, the led_state variable on the stack had been erased and replaced with 0xfff…. Assuming the board was using those memory addresses for other purposes, I switched to using 0x20000100 as my initial stack pointer. This finally ended up working.

The overall structure of the code was very simple. I first initialized the TIM3 interrupts and LEDs in main. I then also set up my stack in main by switching MSP to my initial stack pointer (0x20000100) , pushing the current led state (off), and switching the stack pointer back to its old value. Each time the TIM3 interrupt occurred (which was every second), I read the LED state with a similar procedure as above, but popped the led_state off the stack instead of pushing it. According to that state, I turned the red led on/off, updated the led_state variable, and stored it back on the stack.

In order to reduce my use of unsafe rust features, I put all variables that needed to be globally accessible in mutexes. This ensured that no race conditions would occur.

Source Code for Part 1

I used my Lab 3 Part 2 main.rs code to initialize the red LED, TIM3 interrupt, and the necessary clocks. I also used this code to help me put different variables into mutexes and access variables within the mutexes. I added the body of the TIM3 interrupt and the stack set up code in main to the previously mentioned parts of my Lab 3 Part 2 code to create my Lab 4 Part 1 main.rs.

I also used my makefile, Cargo.toml, memory.x, and .config file from my Lab 3 Part 2. I used all these files unchanged except the makefile and the .config file. I changed the target of my makefile to be "part1_cargo" and changed make debug to not load from a specific file. I also added a "runner" setting to my .config file so I could use "cargo run" to launch gdb. I additionally updated their version of the cortex_m crate I was using, so I added the new dependencies of that to my .config file.

Source code for my Lab 3 Part 2 code can be found in my Lab 3 report.

Part 2

For part 2, I created the function flash_blue, that flashes the blue LED every second and never returns. Within flash_blue (above the loop), I initialized a local variable led_state, to keep track of the current led_state. Since the function never returns, a local variable can be effectively used to store the state. Within the loop in flash blue, the blue LED is turned on/off, the led_state variable is updated, and the delay function is called. I used the cortex_m::asm::delay function to create my busy loop. This function delays the program for at least the number of CPU cycles given. Since the board defaults to a 8MHz clock speed, I used an 8000000 cycle delay to approximate a 1 second delay. I also then set up the TIM3 interrupt to get triggered every 10 ms using a similar approach to the one we used in Lab 3 for setting up timer interrupts in rust.

Explain what happens when the busy loop inside FlashBlue() is interrupted. When the interrupt handler returns, how does the interrupted busy loop resume? Apparently you didn't do anything to make sure the interrupted busy loop resumes properly. Who did it for you?

When the TIM3 interrupt is triggered, the processor pushes certain information about the current program context to the stack pointed to by the MSP. According to the programming manual, 8 words are pushed to the main stack: the contents of registers R0-R3, R12, and PC, a LR value, and a xPCR value. In the programming manual, this is referred to as "the stack frame". The processor saves the current PC value so that it knows what instruction to resume executing when the interrupt is over. It saves the LR value so that it will "remember" where the interrupted code will eventually return to. The overall purpose of saving these 8 words to the stack frame is so that when the interrupted program resumes, it will have all of the same information that it had when it was interrupted.

As this is happening, the processor is also looking up the interrupt handler in the vector table, so it can find the address at which to start executing.

After the stack frame has been saved, the processor also loads a special value to LR to indicate that when the interrupt handler returns, the information in the stack frame needs to be restored to the relevant registers. The special LR value pushed has information about the location of the stack frame (where all the information is that needs to be restored) and what mode the processor was in (handler or thread) when the code was interrupted.

Then the body of the interrupt is executed. In this case, all that includes is getting access to the tim3 variable in the mutex, assigning it to a local variable, and then clearing the pending interrupt bit. After this happens, the special value in LR is put into the PC register. This lets the processor know that it's exiting from an interrupt, so it not only has to pop the local variables created in the body of the interrupt handler off the stack, but it also has to restore the 8 words saved in the stack frame to their corresponding registers. After that, it can resume running the interrupted code in the correct processor mode (as indicated by LR) at the right place (indicated by the PC value that was just restored). All of this is done for me by the processor, so I don't have to worry about making sure the interrupted program resumes correctly.

Source code for Part 2 Code

I used my Part 1 code to initialize the LED, the TIM3 interrupt, and enable the relevant clocks. I added the flash_blue method and deleted the body of the TIM3 interrupt, as it didn't need to do anything for this part. I used a similar method of deciding whether to turn off or turn on my LED in my flash_blue method as I did in the body of the TIM3 interrupt in Part 1. I also used my makefile, Cargo.toml, memory.x, and .config file from my Part 1 code. See the Source code for Part 1 section for more details on the sources of those files.

Part 3

Part 3 was the most difficult part of this lab to complete. My first attempt at this part made me realize that I did not have a deep understanding of what a context switch was or how to implement one. However, through going to office hours, looking at the programming manual, and working on this pset, I vastly improved my understanding of what a context switch is and how to implement one.

From a high level, the program first sets up two stacks, one for flash red and one for flash blue. Each of these stacks contain an exception frame and some saved registers. The idea behind setting up these stacks with exception frames and saved registers is to facilitate the initial transition from the main stack to the task stack. As mentioned in part 2, the programming manual designates that the exception frame should contain (from the bottom to the top of the stack) the xPCR value and the contents of the PC, LR,R12, R3-R0 registers when the exception was triggered. Because the board can only run in "Thumb mode"—using instructions from the Thumb instruction set—the 24th bit of the xPCR value has to be 1 and the least significant bit of LR has to be 1 as well. Since these exception frames will be used to trigger the start of either flash_red or flash_blue, the PC values in the exception fram (which indicate where the exception should return to) were either the address of flash_red or flash_blue, depending on the stack. Since both flash methods never return, the LR value wasn't of much importance, it just needed to have a least significant bit of 1 to indicate that the processor was in Thumb mode. Initially, the rest of the values in the exception frame were just placeholders. In the saved registers portion, the LR value and registered R4-11 were saved as those are the callee saved registers. All the values at this point were fillers except for the LR value, which had to point to the trampoline function that "tricked" the processor into performing an exception return.

After setting up the stack, a timer interrupt had to be set up to fire every 10ms and enabled. Upon entering the timer interrupt global counters for each task were incremented. These were used to flash the leds every second, as once the interrupt fired 100 times 1 second had passed. After that, a local static variable WHOSE_RUNNING, was used to determine which task was running, and run the appropriate context switch function with the right variables. The main context switch function saved the LR value and registers R4-11, saved the current stack pointer to the correct variable, switched the stack pointer to the new context to be run, restored the LR and R4-11 registers from the new stack pointer, saved the new stack pointer, and returned from the function. Initially, this would return to the trampoline function, which would perform an exception return. This is necessary initially because the first time the exception return fires, it pushes all its variables to the main stack and the LR value initially present from the call to context switch will return it to the interrupt handler. Returning to the interrupt handler with a stack that contains none of the variables it thought it would will be problematic. Therefore, the trampoline function skips that and triggers an exception return so that either of the flash functions can be called.

I also have a helper function "context_switch_orig" which switches from the original stack to flash_blue. This method is almost identical to context_switch, except it doesn't save register values or record the location of the old stack.

This process happens again and again. The interrupt fires, the WHOSE_RUNNING variable is used to determine what method to switch to, and then the context_switch function is called.

After finding a design that could practically work, I still ran into many practical problems. The first problem I ran into was that the PC value in my exception frame was "corrupted", according to the board. What was actually happening was I was incorrectly storing the value of "old stack pointer" after saving register values to it during the contex_switch function. This meant when this task was switched back to after being interrupted, the restoring of the registers accidentally popped some values that were actually part of the exception frame. Since the exception frame got messed up, the processor couldn't find the correct PC value to load for the exception return.

Another problem I ran into was when I called the context_switch function inside of a mutex data access section during the initial transition to flash_red/blue, the interrupt ended up being permanently disabled. To fix this, I moved the call to the context switch function outside of the mutex data access section.

To minimize the use of unsafe rust functions, I used mutexes to store global variables when feasible (like GPIOD, TIM3, BLUE_COUNTER, and RED_COUNTER). I also used local static variables in the interrupt handler instead of a global variable to keep track of which function was running.

Source Code for Part 3

I used source code from part 2's main.rs to toggle the LED in flash_red and flash_blue. I also used that source code to set up the timer interrupt and initialize the LEDs. I used my Cargo.toml, .config, and makefile from Part 2 of this lab as well and just changed instances of "part2_cargo" to "part3_cargo". I also moved some dependencies from my .toml to my .config file to get rid of some compile time warnings. I also used the memory.x file from Part 2 unchanged. Please see the "Source code for Part 2 Code" for more details on the source of my Part 2 code.

Part 4

In order to demonstrate my task abstraction, I created 4 tasks (flash_red, flash_blue, flash_orange, and flash_green) which flash LEDs once every second. I created a few user facing methods to help provide nice features to any programmer using my task running software. I first created a method called initialize_tasks, which takes an array of function addresses of size NUM_TASKS. I choose to build my methods and programmatic logic around a constant named NUM_TASKS, which a user could set. Since the number of tasks are known at compile time, this makes it possible to use arrays instead of a data type like Vec, which would be significantly less memory efficient. The initialize_tasks function creates the number of stacks specified by NUM_TASKS and puts them in the TASK_STACK_POINTERS array. Each task is given a 2KB stack. I chose this number in the hopes of balancing the amount of individual tasks that could be run on this interface with how much memory each task has available. The stacks are created in a very similar way as they were for flash_red and flash_blue in Part 3.

The structure of the code that performs the timer interrupt is very similar to what I used in Part 3. However, stack pointers being fed into the context switch function are from the

TASK_STACK_POINTERS array and the task switching logic has been generalized to work for any number of tasks.

I also provided a potential user of my tasking switching software a method of measuring time. Each task has a TASK_TICK_COUNTER value (named like systick), which increments each time the context switch interrupt fires. The delay function checks which task is running, zeros the corresponding TASK_TICK_COUNTER value for that task, waits until the counter value has gotten larger than the given delay value, and then returns. I also provided some other helper methods for other more complex functions that may need raw access to the counter values. Namely, zero_task_time zeros the TASK_TICK_COUNTER value, and get_task_time returns it.

In order to reduce the use of unsafe rust features, I put global variables in a mutex when possible (TASK_TICK_COUNTER and TASK_RUNNING). I also used local static variables (WHOSE_RUNNING) in the interrupt to track the running task. Since in this part I needed global access to which task was running for my delay function, I used the value of WHOSE_RUNNING to update the value in the TASK_RUNNING mutex. This way, I could keep the context switch out of the mutex access section (see Part 3 for more details) but still have safe access to that global variable.

At first when I was developing the task abstraction interface, I updated WHOSE_RUNNING after calling the context switch function. After spending some time debugging, I soon realized that the WHOSE_RUNNING needed to be updated before the context switch function was called. The first context switch to each one of the tasks triggers the trampoline function, which then triggers an exception return. Therefore, the lines under the context switch function will not be called in those instances until after the previously interrupted task is restored, which is too late. I therefore had to update the value of WHOSE_RUNNING before calling the context switch function, even though it made my code a bit messier.

Source Code for Part 4

I used context_switch, context_switch orig and trampoline from my Part 3's main.rs unchanged. I also used Part 3 code to help me set up the stacks, timer interrupts, task switching logic, and example tasks (ie flash_red, flash_blue …). I changed the stack set up code so that it could be used for any task and generalized the task switching logic so that it could be used for any number of tasks. I also used my Cargo.toml, .config, and makefile, and memory.x file from Part 3 practically unchanged. Please see the "Source code for Part 3 Code" for more details on the source of my Part 3 code.