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.