**ECSE 324 Lab 2 Report**

**Part 1: Basic I/O**

Approach

For this part of the lab, I learned how to access, read, and write to certain I/O devices of the virtual board, and from there develop a useful application. That application is using switches and push buttons to display certain things on seven-segment HEX displays.

|  |  |
| --- | --- |
| Pseudocode | Assembly |
| switch(r6) {  case 0:  r5 = 0b0111111  break  case 1:  r5 = 0b0000110  break  // …  } | HEX\_write\_0:  cmp r6, #0  bne HEX\_write\_1  mov r5, #0b0111111  b HEX\_write\_break  HEX\_write\_1:  cmp r6, #1  bne HEX\_write\_2  mov r5, #0b0000110  b HEX\_write\_break  // … |
| Figure 1: Writing switch-case code in Assembly. | |

First, I wrote the code for reading which slider switches are on then writing those bits to the LED address to light up the corresponding LEDs.

For the HEX displays, my three subroutines are very similar. In fact, the only difference between my HEX clear and write is that for clear, I set targeted display segments to 0, while for flood, it is 1. For write, I had to first clear the targeted displays, and then use switch-case style programming (Figure 1) to encode the R1 argument into a seven-segment encoding. For example, if the user wants to display an E, the encoding for it is 0b1111001 (Figure 2).

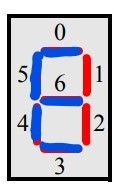
For push buttons (PB), I wrote functions for determining which PB were pressed, which were pressed then released (edgecapture), clearing the edgecapture, and toggling the interruptmask bits (that is meant for part 3, but the given code for the part already does it). Clearing the edgecapture is important, because once a button is released, there is no way of figuring out that the button is released later on without clearing edgecapture (I could poll read\_PB\_data\_ASM and check when a 1 is set back to 0, but that is cumbersome).

Figure 2: Encoding the letter E

Most of the inputs and outputs for the subroutines in part 1 are based around one-hot encoding. For example, 0b0010 could refer to PB1 (second push button). And 0b0110 could refer to PB1 and PB2. For inputs other than switches, a 1 signifies that we want to change the value in this index, and for outputs, a 1 signifies that it is on/pressed for that index.

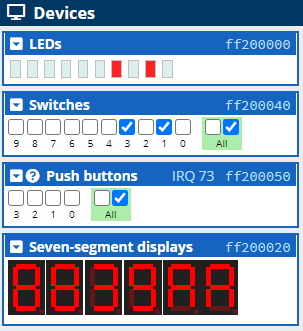
As for the application, I first flooded HEX 4 and 5. Then I wrote an endless loop, which checks multiple things. If SW9 (the leftmost switch) is on, then clear all the 4 displays and keep it that way until SW9 is off. Then if an edgecapture is detected, write the value to the corresponding HEX based on SW0-3.

Challenges

The concept of accessing specific parts of the memory to get the devices to work was still new to me, so I had to spend some time getting used to it. There was a lot more documentation reading than the previous lab.

For pushbuttons, I thought that they were the same as switches. So, I did not see the point of edgecaptures, and thought that the checkmark meant that the push button was “on”. It was not until I thought of the physical board that I realized that releasing a push button meant physically letting go, not turning off the button. “On” and “pressed” are different.

Shortcomings/Improvements

* The application is polling based, so performance-wise it will never be as good as using interrupts.
* I could refactor my three HEX subroutines to massively reduce code duplication.
* Use 8 PBs and HEX displays.

Testing

My testing procedure while running my part1.s code:

1. Check that HEX4 and HEX5 are flooded at the start
2. Turn each switch on and off to see if it changes the LEDs
3. Press then release each button to see if it displays 0 on the first four HEX displays
4. Turning switch 9 (SW9) on clears all the displays, and they stay cleared even when I release more push buttons (until I turn SW9 off)
5. Press all the buttons, then releasing them one at a time also works as intended

Figure 3: Testing the HEX displays.

1. Set SW0-3 to a random value, then seeing if that number is displayed after releasing a PB. Repeat this process for different values to display (e.g. 3, 9, A, A) on different HEX displays (Figure 3). Check if they can even be overwritten by a different value.
2. Make sure that the IDE did not give me any warnings or errors

**Part 2: Timers**

Approach

I learned how to create a stopwatch for this part of the lab, and it was very satisfying to watch it run when it finally worked. The subroutines I had to write were configuring the timer, reading the F bit from the timer register, and clearing that bit. The F bit is set to 1 when the timer has reached 0.

For configuring the timer, I gave it an initial count value and set E to 1 (E=1 means the timer can begin). It is important to note that the timer has a clock frequency of 200MHz, which basically means, if I want to set a timer for 1 second, the initial count has to be 200Ms (not millisecond but megasecond).

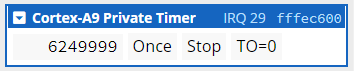
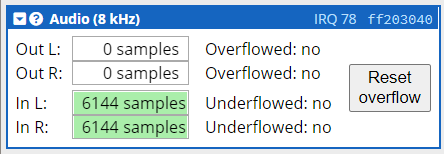
I first wrote code that could count from 0 to 15 in hexadecimal. Then from 15, it loops back to 0 and continues counting. Every time the count changes, I would write it to HEX0. I created an infinite loop that checked if the F bit is 1. If it is, it would set F to 0, then reconfigure the timer to restart (Figure 4).

Figure 4: TO is the F bit. 0 means that the timer is not done yet.

With that code working, I had a strong platform to start on the stopwatch. I first wrote 0 to all the HEXes. Then I used registers R4-9 to keep track of units of 10ms-10min. If the reset push button is released, it would branch all the way back to the stopwatch label, effectively resetting the HEXes and R4-9 back to 0. The stopwatch only loops when PB0 was released and F=1, and PB0 corresponds to the start button. The stop button clears the edgecapture, effectively preventing the contents inside the loop from being run. Finally, I increment R4 everytime the timer reaches 0. When R4 reaches 10, I set it to 0 and increment R5, and that logic follows to R9 (except for R3, I set it to 0 when it reaches 6, since 60s = 1min). R10 keeps track of which of R4-R9 were changed, so that the HEX displays are updated only when necessary.

Challenges

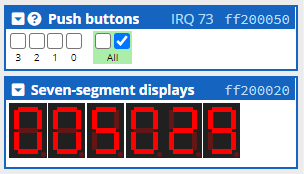
This part was quite straightforward once I understood how the timer device worked from the documentation. I had to come to terms with the fact that I am making a stopwatch, but the timer counts down, so I need a separate up-counter register. The first draft of my code had screen flickering, which was resolved when I introduced R10 into the code (its purpose was mentioned earlier).



Shortcomings/Improvements

* Using interrupts will perform better than polling (hence part 3)
* Make a timer instead of a stopwatch, then have it produce a sound when it reaches 0 (See Figure 5)

Figure 5: This device is probably what is needed to produce sound. Perhaps it will be in lab 3.

Testing

For the 0 to 15 code, testing it just meant making sure that it looped from 0 to F forever onto HEX0 in increments of 1 second.

For the actual stopwatch (Figure 6), I made sure:

1. All the HEX are set to 0 at the start
2. It begins only when I release PB0

Figure 6: The stopwatch in action.

1. It increments like an actual stopwatch, where the numbers to the right move faster
2. It is the same speed as an actual stopwatch
3. There is no HEX flickering
4. PB1 stops the timer
5. Releasing PB0 and PB1 consecutive times separately does not have any additional effect
6. Releasing PB2 resets the displays to 0 and pauses it. Releasing PB0 starts it again and PB1 stops it

I have programmed it so that once it reaches 100 minutes, it will loop back to 0 seconds and continue there. Unfortunately, I do not have the patience to test that out.

**Part 3: Interrupts**

Approach

This part is very similar to the previous part, except that instead of polling the I/O device memory, I check two other memory addresses instead – PB\_int\_flag and tim\_int\_flag. Here were the things that had to be done for this part:

1. Initialize a vector table, which includes IRQ interrupts
2. Disable IRQ interrupts in the processor
3. Configure the GIC so that the timer is identified by interrupt ID 28 and for the pushbuttons, ID 73. This way, I can identify what kind of interrupt I am dealing with.
4. Enable interrupts for the timer and push buttons by setting the I bit field and interruptmask register, respectively
5. Re-enable IRQ interrupts
6. Define KEY\_ISR and ARM\_TIM\_ISR, which basically write to their respective flag addresses then clear the interrupt
7. Modify SERVICE\_IRQ to handle the above two subroutines
8. Copy-paste my part 2 stopwatch code under the IDLE label, then tweak it to check for the flag addresses instead in the main loop

Challenges

There is so much boilerplate involved with this part. There were new concepts and commands that I was not fully familiar with, like vector tables, SVC, CPSR\_c, and MSR. Fortunately, since the code was already there for me, I did not need to understand it fully. Once I managed to get the emulator to trigger SERVICE\_IRQ, it was smooth sailing from there, basing off the code from part 2.

Shortcomings/Improvements

There is still polling in the sense that I keep checking for PB\_int\_flag and tim\_int\_flag. The alternative is writing the timer functionality into the ISR, but that would cause the ISRs to take too long, which would be a problem if higher priority interrupts would like to intervene.

There is a lot of shared code with the other parts. If I could place common subroutines in a library, that would be a lot more space efficient and easier to manage.

Testing

I tested the interrupt-based stopwatch the same way I tested the polling-based stopwatch, since functionality wise, they are the same.