Part 1: Basic I/O

To start the lab, we were given the code for the drivers of the LEDs and slider switches with which we wrote an infinite loop that turns the LEDs on or off depending on the pressed switches. For instance, if switch 4 is pressed, we turn on LED 4. We first use the given "read_slider_switches_ASM" subroutine to store the state of the switches in R0 and then we call "write_LEDs_ASM" which writes the content of R0 in the address of the LEDs' state. The code for this infinite loop is then shown in figure 1.

```
8  loop:
9      BL read_slider_switches_ASM
10     BL write_LEDs_ASM
11     B loop
```

Figure 1: Loop used to control the state of LEDs based on the switches pressed.

We continued the lab by writing a list of subroutines that serve as drivers for the pushbuttons and the HEX displays. All subroutines follow the subroutine calling convention by pushing the used arguments and the LR register at the beginning of the function and popping them before returning. We started with the HEX drivers by writing the "HEX_clear_ASM". The function first saves the content of the passed argument in R4 first because we will use R0 to call a helper function named "clearHEX". We then use the one-hot encoding by ANDing the value in R4 with each of the corresponding HEX display indices one by one. If the result is '1', the display must be cleared so we call the helper subroutine "clearHEX". This subroutine takes two arguments: the HEX display number to clear and the base address controlling the 7 segments of the corresponding display (for example if we want to clear HEX2, R0 contains #2). Figure 2 shows some initialized variables at the beginning of the program, including SEG 1 and SEG2 which are the base addresses controlling the segments of the displays: displays HEX0 to HEX3 are set in SEG1 while HEX4 and HEX5 in SEG2. Figure 3 shows the "clearHEX" helper function. Figure 4 shows this procedure for the HEX4 display, but the same code is applied for all segments.

Figure 2: Variables initialized for the HEX displays

```
.equ HEX0, 0x00000001
.equ HEX1, 0x00000002
.equ HEX2, 0x00000004
.equ HEX3, 0x00000008
.equ HEX4, 0x00000010
.equ HEX5, 0x00000020
.equ SEG1, 0xFF200020
.equ SEG2, 0xFF200030
```

Figure 3: "clearHEX" helper subroutine

```
213 clearHEX:
214     PUSH {R4, LR}
215     MOV R4, #0b0000000
216     STRB R4, [R1, R0]
217     POP {R4, LR}
218     BX LR
```

Figure 4: Code to check if HEX4 must be cleared

```
194     LDR R5, =HEX4
195     TST R4, R5
196     MOVNE R0, #0
197     LDRNE R1, =SEG2
198     BLNE clearHEX
```

To clear the display, we store a byte with 0 values in the base address of the register that controls the corresponding HEX display (stored in R1) to which we add an offset. This offset is stored in R0: thus, R0 will contain '3' if we want to clear HEX3 and not '0x00000008'. Figure 4 shows how we call the "clearHEX function" for HEX4: we pass '0' in the argument because HEX4's segments are controlled with the 7 least significant bits of the register stored at address in SEG2 (SEG2's value is also passed using R1). We then proceeded to write the "HEX_flood_ASM" subroutine, which works exactly like "HEX_clear_ASM". Indeed, the function makes a copy of the passed argument because it uses R0 to call a helper method called "floodHEX". The function also determines which HEX displays to flood by ANDing the argument passed with the indices of the HEX displays. If the ANDing result is '1', we call the "floodHEX" subroutine which works like the "clearHEX" helper function but instead of moving '0b0000000' in the address controlling the segments, we pass "0b1111111". Figure 5 shows how "HEX_flood_ASM" determines if HEX0 is selected and figure 6 shows the "floodHEX" subroutine.

```
226     LDR R5, =HEX0
227     TST R4, R5
228     MOVNE R0, #0
229     LDRNE R1, =SEG1
230     BLNE floodHEX
```

Figure 5: Code to check if we must flood HEX0's segments

```
269 floodHEX:
270     PUSH {R4, LR}
271     MOV R4, #0b1111111
272     STRB R4, [R1, R0]
273     POP {R4, LR}
274     BX LR
```

Figure 6: "floodHEX"

We then wrote the "HEX_write_ASM" function which accepts two arguments: the first one selects the HEX displays to write to (like the previous two subroutines), and the second one is the hexadecimal digit to display on these displays (0 to F). This function is also very similar to "HEX_flood_ASM" or "HEX_clear_ASM". We first check if the HEX display is selected using the same method, and we then call a helper subroutine named "writeHEX". The only difference is that this helper method takes three arguments: the first argument stores the number of the display to use (if R0 is 2, we write to the HEX2 display), the second argument stores the value to display, the third argument selects the register to write to (either the one stored at the address in SEG1 or SEG2). The hex digit we need to display is already stored in R1 when we call "HEX_write_ASM" so it will also be in R1 when we call "writeHEX". In "writeHEX", we use an array of 16 elements initialized at the beginning of the program to display to translate the hex

values to display to their 7-segment representations. The array at index 0 stores the 7-segment representation of 0 : "0x3F". Figure 8 shows the "writeHEX" subroutine.

```
329 writeHEX:
330     PUSH {R4, R5, R6, LR}
331     LDR R5, =values
332     LDRB R4, [R5, R1]
333     STRB R4, [R2, R0]
334     POP {R4, R5, R6, LR}
335     BX LR
```

Figure 8: "writeHEX" helper subroutine.

We then wrote the subroutine drivers for the pushbuttons. All the addresses of the registers used in the pushbutton parallel port are stored in variable at the beginning of the program, we just access the registers' contents using load and store operations. The "PB_data_is_pressed_ASM" subroutine receives a pushbutton index as argument and ANDs it with the content of the register storing the state of pushbuttons (whose address is stored in the PB variable). If the AND operation is '1', the pushbutton is pressed so we used the 'MOVNE' instruction to store '1' in R0, otherwise 'MOVEQ' moves '0' in R0. The "read_PB_edgecp_ASM" returns the indices of the pushbuttons that have been pressed and released. It reads the edgecapture register of the pushbuttons, keeps only its last 4 bits since the rest is unused using the AND operation and it stores the result in R0 before returning. The "PB_edgecp_is_pressed_ASM" subroutine receives a pushbutton index, ANDs it with the content of the edgecapture register which also uses one-hot encoding: if only pushbutton 3 was pressed and released, the edgecapture register will contain: '1000'. The result of this AND operation is stored in R0 before returning. The "PB_clear_edgecp_ASM" subroutine simply stores the value in the edgecapture register (stored in the 'PBEDGE' value) in itself to clear it. The "enable_PB_INT_ASM" subroutine receives the pushbutton for which we need to activate interrupts for. It loads the content of the interruptmask register whose address is stored in the 'PBINT' variable in R5. This register also uses one-hot encoding, so we use the OR operation to keep the already enabled interrupts for the other pushbuttons and put '1' for the pushbutton passed as argument. The result of this or operation is written in the interruptmaskregister. The "disable_PB_INT_ASM" disable the interrupts for the given pushbuttons. It first uses the NOT operation on the passed argument and ANDs the result with the content of the interruptmask register before storing the result in it. Thus, the other pushbuttons that were not passed as arguments are not disabled.

We then proceed to write the application using the drivers we just wrote. First, the application starts by clearing the edgecapture register using the "PB_clear_edgecp_ASM" subroutine because when we stop our program and re-run it in the emulator, the edgecapture register is not cleared. Moreover, we flood the HEX4 and HEX5 displays using the "HEX_flood_ASM" subroutine. We then enter an infinite loop in which we map the state of the switches to the LEDs by using the content of our loop from figure 1 and their corresponding drivers. The loop then checks the state of switch 9 using the content we just stored in R0 (the state of all switches): if the switch is on, we clear all displays (including HEX4 and HEX5), otherwise we make sure HEX4 and HEX5 displays are still flooded. This first part of the loop is shown in Figure 9. Afterwards, the loop checks the edgecapture register of the pushbuttons using the "read_PB_edgecp_ASM" subroutine. We directly call the "PB_clear_edgecp_ASM" after we read this value to make sure that the values displayed will not change if we modify the switches. The program then checks if each pushbutton was pressed and released using the content of the edgecapture register we just obtained by ANDing it with the index of the pushbuttons (uses one hot encoding). If the pushbutton was pressed and released, we store the index of the corresponding display and call the "HEX_write_ASM" subroutine. For instance, pushbutton 0 is associated with HEX0, so if pushbutton 0 is pressed and released, we display the state of the last four switches on HEX0. Notice that we copy the state of the last for switches in R1 at the beginning of the loop using an AND operation that keeps the last four bits of the state of all switches. Figure 10 shows the procedure to check if HEX0 should display the value of the last four switches, but the same procedure is used for HEX1, HEX2 and HEX3.

```
31     BL read_slider_switches_ASM
32     BL write_LEDs_ASM
33     AND R1, R0, #0x0000000F
34
35     MOV R2, #0x00000200
36     TST R0, R2
37     MOVNE R0, #0x000000FF
38     BLNE HEX_clear_ASM
39
40     // If the SW9 switch is off,
41     MOVEQ R0, #0x00000030
42     BLEQ HEX_flood_ASM
```

Figure 9: Beginning of the loop

```
40     // If the SW9 switch is
41     MOVEQ R0, #0x00000030
42     BLEQ HEX_flood_ASM
43
44     BL read_PB_edgecp_ASM
45     BL PB_clear_edgecp_ASM
46     MOV R3, R0
47
48     LDR R2, =PB0    // If t
49     TST R3, R2
50     LDRNE R0, =HEX0 // Load
51     BLNE HEX_write_ASM
```

Figure 10: Code to determine if HEX0 should display the switches' valu

When writing this application, I encountered issues when I first ran it because I was not clearing the edgecapture register of the pushbuttons, which made the values change on the displays. After correcting this issue, the application behaved as expected.

Part 2: Timers

This part of the lab consisted of writing subroutines to use the ARM A9 private timer and use them to create a stopwatch using the other previously written subroutines for the HEX displays and the pushbuttons. We started by writing the subroutines for the timer. Again, all subroutines respect the subroutine calling convention by pushing the used registers at the beginning of the function and popping them before returning. The ARM_TIM_config_ASM subroutine configures the private timer and takes two arguments: the initial count value that will be placed in the load register of the timer whose address is stored in the 'TIMLOAD' variable, and the configuration bits that will be set in the control register whose address is stored in 'TIMCOUNT' variable. The function uses load and store operations to get the corresponding registers and store the value of the arguments in them. Notice that for setting the configuration bits, we store a byte in the register because we do not want to modify the "Prescaler" section which starts after bit 7 in the control register of the timer. This subroutine is shown in figure 11. The "ARM_TIM_read_INT_ASM" subroutine reads the "F value" of the interruptstatus register of the timer, whose address is stored in the 'TIMSTATUS' variable. The function uses load and store instructions to read from the interruptstatus register and store its content in R0. We then isolate the "F value" by using the AND operation and storing the result in R0 before returning. This function is shown in figure 12. Finally, the "ARM_TIM_clear_INT_ASM" subfunction clears the "F value" in the interruptstatus register of the timer. It is similar to "ARM_TIM_read_INT_ASM" but instead of returning the value, we just store the value obtained back into the interruptstatus register to clear it.

```
131 ARM_TIM_config_ASM:
132     PUSH {R4, LR}
133     LDR R4, =TIMLOAD
134     STR R0, [R4]
135     LDR R4, =TIMCONT
136     STRB R1, [R4]
137     POP {R4, LR}
138     BX LR
```

Figure 11: "ARM_TIM_config_ASM" subroutine

```
142 ARM_TIM_read_INT_ASM:
143     PUSH {R4, LR}
144     LDR R4, =TIMSTATUS
145     LDR R0, [R4]
146     AND R0, R0, #0x00000001
147     POP {R4, LR}
148     BX LR
```

Figure 12: "ARM_TIM_read_INT_ASM" subroutine

We then wrote the stopwatch which consists of displaying the time on the HEX displays and controlling the timer using the edgecapture register of the pushbuttons. We start the program by configuring the private timer using the "ARM_TIM_config_ASM" subroutine: we set the start value to be '20000000' which we obtained using the following equation: 200MHz * 100ms. We also set the configuration bits to be '110': I = '1' to enable interrupts, A = '1' to make sure the timer restarts after it reached '0' and E = '0' because we don't want the timer to start now. The timer therefore counts down from '20000000' to '0' and then restarts at '20000000'. Then we use the "MOV" instruction to store '0' in registers R5 to R10. These registers are used to keep track of the stopwatch value: R5 holds the diciseconds, R6-R7 the seconds, R8-R9 the minutes and R10 the hours. The program then enters a loop and starts by reading the edgecapture register of the pushbuttons and clearing it after reading it by using the "read_PB_edgecp_ASM" and "PB_clear_edgecp_ASM" subroutines respectively. The loop then checks which pushbutton was pressed by ANDing the value we obtained by reading the edgecapture register and the index of each pushbutton. If pushbutton 0 was pressed and released, we use load and store instructions to set the configuration bits in the control register of the timer to '111' (E is now '1' so the timer starts). This part of the loop is shown in Figure 13. If pushbutton 1 was pressed and released, the configuration bits are set back to '110' which stops the counter (E='0'). If pushbutton 2 was pressed and released, we reset the counter by setting the configuration bits in the control register of the timer to '111' (E is '1' so timer starts if it was off) and we set all the registers holding the stopwatch value to '0'. Afterwards, we must check the "F value" of the timer which indicates that the timer has reached '0' and restarted using the "ARM_TIM_read_INT_ASM" subroutine. If F is '0' we go back to the beginning of the loop, otherwise we clear it using the "ARM_TIM_clear_INT_ASM" subroutine and we increment the timer by 100ms. This part of the code is shown in Figure 14. When the stopwatch value increments by 100ms, we make sure that if the value of the diciseconds is 10, that it goes back to 0 and that the seconds are incremented by '1'. This process goes on for tens of seconds, then minutes, then tens of minutes… The last part of the loop updates the HEX displays using the new value of the stopwatch.

When writing this part of the lab, I mistakenly used the interval timer instead of the private timer. My code worked but I modified it accordingly when the TA noticed my mistake. The stopwatch behaves as expected.

```
41 loop:
42     BL read_PB_edgecp_ASM
43     BL PB_clear_edgecp_ASM
44     MOV R3, R0
45
46     LDR R2, =PB0
47     TST R3, R2
48     MOVNE R1, #0x7
49     LDRNE R4, =TIMCONT
50     STRNE R1, [R4]
```

Figure 13: Beginning of the loop that shows how we determine if the pushbutton 0 was pressed and released.

```
70     // Check the "F" value
71     BL ARM_TIM_read_INT_ASM
72     CMP R0, #0x00000001
73     BNE loop
74     BLEQ ARM_TIM_clear_INT_AS
75     ADDEQ R5, R5, #1
```

Figure 14: Part of the loop that checks the "F value" of the timer.

Part 3: Interrupts

This part of the lab consisted of implementing the same application as part 2, but instead of periodically checking the pushbuttons and timer for an event, we configured interrupts and used them to implement the stopwatch. We used the provided code to set up the interrupts. We first initialized the exception vector table and then set the stack to the A9 on-chip memory in IRQ mode. This is done in the "start_" subroutine where we also enable both the interrupts for the pushbuttons using the "enable_PB_INT_ASM" subroutine and the interrupts for the private timer using the "ARM_TIM_config_ASM" subroutine as shown in Figure 15. We call the "ARM_TIM_config_ASM" subroutine using "0110" as configuration bits because we just set the I and A bits to '1' for the timer to restart after it reached '0' and to enable interrupts. We also call the "CONFIG_GIC" subroutine which configures the ARM GIC (Generic Interrupt Controller). We modified this "CONFIG_GIC" subroutine by configuring the interrupts for the pushbuttons and the timer by passing their corresponding IDs to the given "CONFIG_INTERRUPT" subroutine (id is '29' for the timer and '73' for the pushbutton). This is shown in Figure 16. We also set the content of the registers holding the stopwatch's value in the "start_" subroutine

```
55    MOV R0, #0x0000000F
56    BL enable_PB_INT_ASM
57
58    LDR R0, =time
59    MOV R1, #0x6
60    BL ARM_TIM_config_ASM
```

Figure 15: enable interrupts for the pushbuttons and private timer in the "start_" subroutine

```
300   MOV R0, #73
301   MOV R1, #1
302   BL CONFIG_INTERRUPT
303
304   MOV R0, #29
305   MOV R1, #1
306   BL CONFIG_INTERRUPT
307
```

Figure 16: Configure the interrupts for the pushbuttons and the private timer in the "CONFIG_GIC" subroutine

The configuration of the interrupts is therefore made at the very beginning of the code before we enter the "IDLE" subroutine which contains the stopwatch's operations. When an interrupt occurs (a pushbutton was pressed and released or the timer reached '0'), the "SERVICE_IRQ" subroutine which we modified to check for the pushbutton or private timer interrupts is called. The code in this subroutine reads the ICCIAR register to check the id of the interrupt that occurred. We use the "CMP" instruction to check if this interrupt was caused by either the private timer or the pushbutton as shown on Figure 17. If the id is '29', the interrupt is from the timer, so we branch to the "ARM_TIM_ISR" subroutine. This subroutine reads the "F value" of the timer, stores its value in the memory "tim_int_flag" and clears it by writing to the "F value" before returning as shown on Figure 18. If the id is '78', the interrupt is from one of the pushbuttons, so we branch to the given "KEY_ISR" subroutine. We modified this subroutine by removing the part that updated the HEX displays based on the pushbuttons pressed and released, and by only storing the content of the edgecapture register in the "PB_int_flag" memory. We also clear the interrupt by writing to the edgecapture register before returing. If the interrupt that occurred was not from the pushbuttons or the private timer, the "SERVICE_IQ" branche enters an infinite loop using the "UNEXPECTED" label.

```
271 Timer_check:
272     CMP R5, #29     // *
273     BNE Pushbutton_check
274     BL ARM_TIM_ISR
275     B EXIT_IRQ
276
277 Pushbutton_check:
278     CMP R5, #73     // Ch
279 UNEXPECTED:
280     BNE UNEXPECTED
281     BL KEY_ISR
```

```
377 ARM_TIM_ISR:
378     LDR R0, =TIMSTATUS
379     LDR R1, [R0]
380     AND R1, R1, #0x00000001
381     LDR R2, =tim_int_flag
382     STR R1, [R2]
383     STR R1, [R0]
384
```

In the "IDLE" subroutine, we modified the code we wrote in part 2 to use the interrupt configuration we just wrote. Indeed, when the pushbuttons are pressed and released, the "PB_int_flag" memory is updated to hold the indices of pushbuttons that were pressed and released. Thus, our loop doesn't have to check or clear the content of the edgecapture register of the pushbuttons, but just check the "PB_int_flag" memory. This is shown on Figure 19. Moroever, the loop doesn't have to call a subroutine to check the "F value" of the private timer, since the interrupt will set the "tim_int_flag" memory to '1' if the "F value" is '1'. Figure 20 shows how we check if the timer reached '0'.

```
73 IDLE:
74     LDR R4, =PB_int_flag
75     LDR R3, [R4]
```

```
103   // * Check for timer interrupts
104   LDR R1, =tim_int_flag
105   LDR R0, [R1]
106   CMP R0, #0x00000001
107   BNE IDLE                    //
108
109   MOV R0, #0x00000000         //
110   STR R0, [R1]
```

Since I based my code of part 3 on the second part, I had the issue of using the wrong timer (Intermediate timer instead of the private one). Therefore, my interrupts were not captured by the program and the “SERVICE_IRQ” always ended in the infinite loop of “UNDEFINED”. After correcting this mistake, my code worked as expected.