ECSE 324 - Computer Organization
Lab session 003
Winter 2020

Lab Assignment 3 - Basic I/O, Timers and Interrupts

Group 05 :
Narry Zendehrooh Kermani 260700556
Imane Chafi 260847716

Basic I/O

For this laboratory, we were introduced to integrating C and subroutine codes in a more explicit way. The drivers, divided into 3 folders as asm, inc and src, are used in concordance with the main files. The asm folder held the subroutines in ARM and the inc folder held the .h header files for the function declarations to use between different files. The src folder has the int_setup file, which was provided to us with the code calling the different C functions.

Slider switches and LEDs program

For this code, we used ARM and C code to control the slider switches to switch the LED lights on and off the board depending on the state of the switches. We created a slider_switches.s assembly file that has a simple branch called read_slider_switches_ASM which reads the memory of the slider switches into a register, R1, and loads this value into R0. We do this such as to know which slider is on or off. The branch ends with a BX LR exit with link register instruction so that we can use this branch and get back to where we were left in the code with the link register. Furthermore, to make use of this branch, we needed to create a header file called slider_switches.h which provides the C function declaration for our assembly code. The code was provided to us, with a simple declaration of read_slider_switches_ASM as an extern declaration to call that branch from the assembly file created beforehand in the asm folder.

Secondly, to switch the lights of the LED on, we need to be able to read and write to the LED lights of the board. As such, we've created subroutines called write_LEDs_ASM and read_LEDs_ASM, which stores what is provided by the R0 register into the memory location of the LED light for each LED, and then we can branch back to LR to load the next LED light register. The other subroutine created is read_LEDs_ASM, which simply loads the value of the LED into R0. In the inc folder, we needed to create a LEDs.h header file with function declarations that accepts an argument for the value of the R0 register to write into. Indeed, the 2 function declarations added to this header file is int read_LEDs_ASM() that returns the value of the LED light memory, and void write_LEDs_ASM(int val) with the parameter int val. Finally, we've added the main.c code with the write_LEDs_ASM with parameter read_slider_switches_ASM() to load all the values of the sliders that are on and writing them into the LED lights to switch them on if the slider values have been switch on. One of the challenges of this code was mostly to understand how the C files would interact with the subroutines, and how to implement the subroutine calls from the main.c file. One of the improvements that we could make on this code would be to simply load the value of the switches base into R0 directly for the read_slider_switches_ASM subroutine instead of loading it into R1 before reloading this value into R0.

Entire basic I/O program

For the HEX displays and push buttons, the code was simply to create, using our basics learned from the previous program, a subroutine that could write to the HEX board displays hexadecimal numbers between 0 to 15, based on the 4 first switches and the push buttons pressed. The push buttons would provide the variables for the position of the HEX display that we want to add numbers to, and the sliders would provide the values from 0 to 15 that we want to add to the HEX display (the first slider on the left is 2^0, the second is 2^1, and so on). The code uses a similar structure than the first one, with the exception of introducing new subroutines called HEX_clear_ASM, HEX_flood_ASM and HEX_write_ASM. The HEX_clear subroutine will only take as parameter the enumeration that holds all the displays, and clear all the segments of the display. The HEX_flood also only has the enumeration as parameters, and it will make sure that all the segments are turned on. The last subroutine is the HEX_write_ASM, which will write the number from 0-15 to the corresponding display.

To go more into details, the HEX_clear_ASM pushes all the registers first into the stack, then we create another branch to clear the display, which loads the value right shifted by 2 for 8 bits and keeps an empty space. We then go to another branch called HEX_clear_DONE, which stores the new empty value to the registers that hold the HEX displays, pops the values from the stack, and clears the HEX displays with the empty values stored in the registers, then we go back to the link register to where we were left in the code. Most of the challenges we faced while writing this code was writing the main.c file and calling the subroutines created with the right pushbuttons for the right HEX displays. Another challenge was making sure that the clear function was iterating through each register that holds data from the sliders and HEX displays, as at first when the push button for clearing the displays was pressed, it only deleted the first HEX displays and not all of them. We then created a loop for iterating through each register and storing the empty value into their memory locations. One important improvement that we could make to this code is to reuse a branch that loops through the registers for each subroutine created, instead of having multiple branches that loop through the registers for each subroutine, as we did in our code. Indeed, this would lower the lines of code of our assembly file and make the code much more understandable.

Timers

This part of the code was a bit more complicated than the previous part, as we needed to understand how the timers worked to make sure that every time a minute passed, it would update the right HEX display and reset the HEX displays for the seconds accordingly, while incrementing the number of minutes passed. To achieve this clock, we set the push buttons as our stop, start and delete actions, and we could reuse the subroutine created earlier HEX_write_ASM to write to the HEX displays. Indeed, we first needed to configure our

timers(struct pointer) with the HPS_TIM_config_ASM subroutine, such that we can disable the timers before we can write the other configuration parameters such as timeout and enable. Furthermore, the HPS_TIM_read_INT_ASM subroutine was also created to return the value of the S-bit and takes as input the timer enum. The final subroutine to code was the HPS_TIM_clear_INT_ASM subroutine that clears the registers holding the timer base values from configuration if the push button for clearing the HEX displays.

One of the greatest challenges of this part of the code was clearing the right HEX displays based on the values of the timer. To resolve this, we created variables in the main file to hold the seconds, milliseconds and minutes, such as to make if statements that can update the HEX displays if a second or minute has passed. Furthermore, multiple improvements could be made on our code, especially in terms of branch use and optimization of writing to the HEX displays. When we wrote the main class that calls the HPS_TIM_clear_INT_ASM and HEX_write_ASM, we created variables that hold the count, the first HEX display and the second HEX display for seconds and minutes. However, we could have reused a count variable if the count for the number of minutes isn't being used until the number of minutes is updated. Indeed, another improvement would be to use interrupts instead as the processor, in this code, waits for the pushbuttons's actions constantly, wasting time that it would use on other processes. Using a code with interrupts, we would have a much faster and more efficient code.

Interrupts

This part of the lab is very similar to the polling stopwatch. The way the counters on the hex displays work is the same. In the polling stopwatch, a separate timer is polled to determine when to check whether a button had been triggered. In another word, the polling stopwatch uses two timers, one for the stopwatch and one polling the edge cap display. Instead, for the interrupt stopwatch, we only use one timer. Indeed, interrupts are used, and we implemented in the ISR a key routine. In the interrupt stopwatch, when the button is released, it triggers an interrupt that starts, stops or resets the timer. Another difference between polling and interrupt stopwatch is that the polling stopwatch reacts whenever the button is pressed, but the interrupt stopwatch reacts whenever the button is released. For this part, most of the codes were provided, and we only had to write the ISR.s and revise the main.c. In the ISR.s file, spaces in memory were set apart for flags for each of the timers and a simple interrupt service routine that handles timer selection is written. In the main.c, the structure is similar to the polling stopwatch. We only replaced the polling timer with a statement that watches for the interrupt flag to be raised.

The challenge that we faced for this part was the implementation of the subroutine for the pushbutton interrupt which was a new concept for us so that we needed to get familiar with it. An improvement for this part could be to combine the two stopwatch codes so that we can be able to switch between polling and interrupt stopwatch by using different pushbuttons for each so that the main.c file becomes more efficient.