**CMPE-250 Laboratory Exercise Ten**

**Timer Driver Input Timing**

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students; however, other than code provided by the instructor for this exercise, all code was developed by me.

Joel Yuhas

Performed 4/14/16

Submitted 4/21/16

Lab Section L2

Instructor: Shaaban, Muhammad

TA: Peter Muller

Stephen Moore

Connor Goss

Lecture Section 01

Professor: Sarra, Alessandro

**Abstract**

A timer for the Freescale Freedom Board KL46Z was developed and tested. The timer was implemented using an interrupt-based design, which utilized PIT in order to measure time to within 0.01s. The PIT timer was used in tandem with UART0, as well as other previously generated UART0 subroutines such as UART\_ISR, which was responsible for printing data to the terminal window. Ultimately, the timer was implemented in order to measure the amount of time it would take for a user to input an access code. The program would check to see if the code that the user inputted equaled a predetermined code, and would also check to see if it was done so within 5 seconds. If the user met both those requirements, then the “access was granted” and the time was printed. The exercise provided an opportunity to develop and further the understanding of assembly as well as to learn and implement PIT alongside of UART0. Overall, the implementation of PIT ended up being a success and the rest of the program worked flawlessly as well.

**Procedure**

Two subroutines were written, Init\_PIT\_IRQ and PIT\_ISR that would work alongside the code that was created in exercise 09. The program that was being implemented used an interrupt-based timing system that was offered by PIT and would measure the amount of time it took for the user to type in an access code.

*Init\_PIT\_IRQ*

Init\_PIT\_IRQ was responsible for initializing the PIT mechanisms. This was done through a series of steps that followed in this order:

1. Enabling pit clock
2. Disabling timer
3. Setting pit interrupt priority
4. Clearing any pending PIT interrupts
5. Unmasking PIT interrupts
6. Enabling PIT Modula
7. Setting interrupt for every 0.01s
8. Enabling PIT timer to channel 0

Once the initialization was complete, the program would always call the PIT\_IRQ interrupt subroutine. In order to tell the program where to PIT\_IRQ subroutine was, it had to be “installed” which involved placing PIT\_IRQ into the vector table at location 38.

*PIT\_IRQ*

The PIT\_IRQ was the subroutine that incremented a counter every time PIT\_IRQ was called. Because of the functionality of PIT, this meant that PIT\_IRQ would be called every 10ms, which was determined based on the code implemented in the Init\_PIT\_IRQ. This was done based off of two variables, RunStopWatch and Count.

RunStopWatch is what started/stopped the counter. Whenever RunStopWatch was ‘1’, the PIT\_IRQ would branch into an if statement that allowed it to add to a counter, which was precisely what the variable Count was used for.

Count stored the amount of times PIT\_IRQ was called while RunStopWatch was ‘1’, and since PIT\_IRQ was called every 10ms, count effectively stored the amount of time between when RunStopWatch was turned on and off. Using this functionality, the program now had a timer it could use to measure the amount of time it took the user to input a code.

Next Init\_UART\_IRQ was used to initialize the KL46 for interrupt based serial I/O. This was the exact same technique that was used in exercise 09 to receive and print items from/to the terminal window. This also included “installing” the UART\_ISR just like PIT\_ISR, by placing its label in the vector table at 28 for UART (38 for PIT).

With I/0 and the timer effectively ready, the main program was written, which involved loading 0 into RunStopWatch as well as into Count to initialize them, calling Init\_UART\_IRQ and Init\_PIT\_IRQ and then printing the initial string to the terminal. After the initial string had been printed, the timer was started (RunStopWatch = ‘1’) and the program waited for the user to input a value using GetStringSB. Once GetStringSB had returned a value, the timer was stopped (RunStopWatch = ‘0’) and the value from GetStringSB as well as the time stored in Count were checked to see if they met the requirements to earn the user the “access granted” notification. If the user did not satisfy those requirements “access denied” was printed.

The program did not have to loop back, but a loop was instantiated anyway, that way the user could have more than one try if they missed the code.

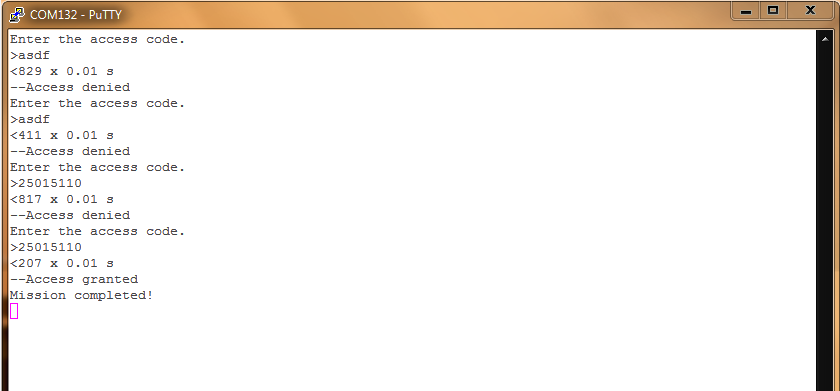
Once the program was written, it was tested to ensure it produced the correct results, which it did.

**Results**

Therewere only two areas from the exercise from which results were gathered. The first was a screen capture of the terminal window, showing:

* Wrong code input before 5s
* Wrong code input after 5s
* Correct code input after 5s
* Correct code input before 5

Figure 1 shows the screen capture that was gathered and contains the previously stated items. The access-denied/access-granted statuses as well as the time to enter the codes can be clearly seen after the code had been entered by the user.

****

**Figure 1: Screen Capture Access Code**

Results were also gathered from the memory map, where memory ranges of were gathered and listed. These memory ranges involved the ranges of:

* Executable code
* PIT ISR code
* Constants in Rom
* RAM used

Table 1 shows the memory ranges of the code listed above.

**Table 1: Exercise 10 Memory Map**

|  |  |  |
| --- | --- | --- |
|  | **Start Address** | **Last Byte** |
| **Executable Code** | 0x00000410 | 0x000000913 |
| **PIT\_ISR Code** | 0x0000076B | 0x0000076B |
| **Constants in Rom** | 0x00000204 | 0x000002CF |
| **Ram used** | 0x1FFFE000 | 0x1FFFE237 |

(Ram used = 0x00000237 in size)

Overall the exercise was a success at producing the desired results.

**Conclusion**

A timer was implemented using an interrupt-based design, which utilized PIT in order to measure time to within 0.01s. The PIT timer was used in tandem with UART0, as well as other previously generated UART0 subroutines such as UART\_ISR that was responsible for printing data to the terminal window.

The implementation was an outstanding success and provided insight into a new mechanism of Assembly that was PIT and also created a scenario where two interrupt handlers, PIT and UART, were able to be implemented alongside one another. The code worked exactly as intended and allowed the user to study exactly how timers could be implemented using Assembly and overall was a success.