ECE 371: Project 2

Painful blinking lights

Portland State University
Maseeh College of Engineering & Computer Science
Department of Electrical & Computer Engineering

Authors:
Stephen Johnston

March 18th, 2018

Contents

List of Figures

1900 SW 4^{th} Ave, suite 60-08, Portland, OR 97201 •
https://www.pdx.edu/ece/tektronix-circuit-design-and-testing-laboratory

1 Objective

The main objective of Project 2 is to design a led timer that counts in both reflective grey code, binary and does this using timer interrupts. The main problem to understand is documenting how to turn on the timer and how to make sure that the interrupt is distinctly from the timer. Interrupts are also tricky because, as was learned late in the project, having a distinct register output for an interrupt is unreliable and can over write information being used by other programs that were interrupted.

2 Approach

2.1 AM33x DMTIMER2

This wasn't necessarily easy, but was the least time (haha) consuming part of the project. Much of what was taken in terms of implementation was implemented just how it was in the book. Our implementation method is:

- Turn on the timer clock
- Set the clock to a frequency of 32KHz
- Reset any presets on timer 2
- Enable overflow interrupts on timer 2
- Determine count value
- Turn on timer

The way the timer works is by counting up to a specific value, sending a signal to IRQ and wait to be reset. This is called "Auto reload mode" on the DMtimer manual.

Table 1:DM TIMER addresses in use	
Timer clock	0x44E00080
Clock frequency selection	0x44E00508
Base address	0x48040000
Timer 2 CFG	0x48040010
Timer IRQ register	0x4804002C
TLDR register	0x48040040
TCRR register (count register)	0x4804003C

2.2 Interrupts

Table 2: Interrupt addresses in use	
Base address for controller	0x48200000
Interrupt config register	0x48200010
Interrupt flag reg	0x482000C8

Interrupts are like bar fights, if you slap someone with a drink in their hands, they can't easily fight back.

Figure 1: Chen, my name for the interrupt, slapping a program

This is because they might risk spilling their drink. However, if you manage to knock the drink out of their hands, the game has changed.

Similarly, if you during an interrupt knock a much needed register out of a programs hands (Perhaps R0, the default output), you risk crashing the entire system. This happened repeatedly in my design, where the interrupt would interfere with elements that were being used (over writing an address during a STR stage, for example) and I had no idea why. This caused much headache. Thus I have recognized the rules of interrupts.

RULES OF INTERRUPTING

1. Always write to RAM. Interrupts have no outputs. Do not assume they ever do or will.
2. ALWAYS. WRITE. TO. RAM. You have no idea where the program got interrupted.
3. Interrupts hate branching. Don't bother
4. Even if the code works now, it might not later.
5. Change one line of code at a time when doing interrupts. Refer to previous line for when reverting code.

Once these rules are followed, we then can actually understand what to do next. When an interrupt is called, we have a series of questions to ask. Namely: *How many interrupts have happened before?*

Was this interrupt from the timer?

Is there any other way the PC could have found its way here?

After these questions are answered we can then participate in the desired program. My first step after the interrupt is to increment or reset the interrupt counter, which is stored in ram as TIMERC. We load TIMERC and check to see its current value. If its less than 9 (incremented every 900ms), we add 1. If its greaterthan or equal to 9, we go to the next stage.

The next stage now is to load from our led counter, COUNTER, and increment it by 1. Then we return back to the scheduled programming. Note that the program not only should have no output, but restores all registers back to their original state after the interrupt. We absolutely need to make sure that nothing is changed during the interrupt.

3 Algorithm

For the sake of brevity, I'm omitting the original pseudocode that discusses how the lights are arranged or how the button is checked.

```
// == PROGRAM SECTION ==
.text
.global _start
.global INT_DIRECTOR
_start:


 //Interrupt controller
 LOAD R1, =0x48200000  //Base address for the controller
 COPY R2, #0x2         //This value resets the controller
 STORE R2, [R1, #0x10] //
 COPY R2, #0x10        //Then we put our own settings and make sure
   that Timer as
 STORE R2, [R1, #0xC8] //an interrupt is enabled


 // Set up timer2 clock
 COPY R2, #0x2
 LOAD R1, =0x44E00080 //Address of clock control
 STORE R2, [R1]     //Turn on clock
 LOAD R1, =0x44E00508    //Address of clock selection timer
 STORE R2, [R1]
 LOAD R1, =0x48040000 //Starting address
 COPY R2, #0x1 //Resets timer 2
 STORE R2, [R1, #0x10]     //Write to timer config register
 COPY R2, #0x2   //Enable overflow interrupt
 STORE R2, [R1, #0x2C] //Address for overflow interrupt
 LOAD R2, =0xFFFFF333  //Timer for 100ms
 STORE R2, [R1, #0x40] //Timer TLOAD load register
 STORE R2, [R1, #0x3C]      // Writes to TCRR


//Turn on the timer
```

```
33   COPY R2, #0x03        //This turns on the timer
34   LOAD R1, =0x48040038   //this is the address for TCLR
35   STORE R2, [R1]
36
37     //Enable IRQ
38
39   MRS R3, CPSR
40   BIC R3, #0x80
41   MSR CPSR_c, R3
42
43
44
45
46   LOAD R1, =COUNTER  //This makes sure that the value at COUNTER begins
       with 0
47   COPY R0, #0          //
48   STORE R0, [R1]       //
49
50
51 Main:
52 Main:
53   LOAD R1, =COUNTER      //We check to make sure that R1 is less than 15
54   LOAD R0, [R1]          //
55   COMPARE R0, #15           //
56
57   IF GREATER THAN {
58   COPY R0, #0          // If not, then we set it to 0
59   STORE R0, [R1]    }     //
60   MOV R1, R0            // Next we check to see if timerc is 9 or not.
       If so, we set R1 to 0 until the next interrupt.
61   LOAD R2, =TIMERC       //
62   LOAD R0, [R2]          //
63   COMPARE R0, #9            //
64   MOVEQ R1, #0          //
65   GOTO check              //Then we check if the button was pressed
66   COMPARE R0, [BIT TWO]
67     GOTO IF NOT EQUAL noteq     //Then we go in different directions
68     GOTO IF EQUAL eq
```

```
69
70
71
72      //This is the main loop, normally there's information on lighting
        the GPIO lights up, however I didn't feel like this was important
        for explaining interrupts and the timer function.
73
74
75  //Here's the interrupt director
76    INT_DIRECTOR:
77
78    STMFD SP!, {R0-R3,LR}
79    LOAD R1, [TIMER INTERRUPT ADDRESS]     //Checks if overflow happened
80    LOAD R0, [R1]              //
81    COMPARE R0, #0x2               //When overflow occurs, this flag is
        turned on, meaning
82    BNE Interrupt              //that we can check it just in case of a
        false positive
83
84
85    LOAD R1, =TIMERC    //Loads the address of the interrupt counter
86    LOAD R0, [R1]
87    COMPARE R0, #9          //If the interrupt counter is 9 or greater, we
        go to the next
88    ADDLT R0, R0, #1  //process. If not, we add 1 and flip the interrupt
        flag.
89    STORELT R0, [R1]
90    BLT Skip
91
92    COPYGE R0, #0       //We also reset the counter to 0.
93    STOREGE R0, [R1]
94
95    LOAD R1, =COUNTER    //I keep having problems with registers getting
        over written, so now I'm keeping it at a totally separate location.
96    LOAD R0, [R1]
97    ADD R0, R0, #1  //Increments by 1
98    STORE R0, [R1]
99  Skip:
```

```
100     LOAD R1, [ADDRESS FOR TIMER OV]  //Address for the timer overflow
        flag
101   COPY R2, #0x2     //Value to turn off the timer and its overflow.
102   STORE R2, [R1]
103
104
105   COPY R2, #0x03        //This turns on the timer
106   LOAD R1, [TIMER COUNTER ADDRESS]  //this is the address for TCLR
107   STORE R2, [R1]
108
109 Interrupt:
110   LOAD R0, [INTERRUPT CONTROLLER ADDRESS]  //address of the interrupt
        controller flag
111   COPY R2, #01      //Value to reset
112   STORE R2, [R0]
113   COPY R1, R5
114   LOAD R0-R3 AND LR
115   SUBTRACT PC, LR, #4
116
117 END:
118 // == DATA SECTION ==
119 // This section contains data that will be allocated in memory (RAM)
        when loaded.
120 // A pointer to this allocated space is loaded into R13 (SP).
121 .data
122 .align 4
123 TIMERC:  .word 0xFFFF
124 COUNTER:  .word 0xFFFF
125 STACK1:    .REPT 256
126         .BYTE 0x00
127         .ENDR
128 STACK2:    .REPT 256
129         .BYTE 0x00
130         .ENDR
131 .END
```

4 Testing

My program was filled quite literally with bugs on the first run through. Again. Electric Boogaloo. It wasn't welcome or easy to respond towards. Not only does the debugger not necessarily respond correctly to software breakpoints, but a lot of the interrupt programming is not intuitive. We can't predict when or where an interrupt will happen. Often because of this, my program would work sometimes and sometimes not work. What does this mean? A crash in the context of this report is when the normal loop no longer works, instead we find ourselves in the middle of a list of unknown assembly commands that are continuously looping back to a branch that is meant for responding to "Unknown data"

My initial thought was that because I am using two stacks, the registers held in each stack could not communicate. It turns out that despite using two stacks, the registers are the same. I'm not sure how to interpret this. At this point my program was crashing immediately after leaving the interrupt. This meant to me that the interrupt worked correctly.

I wrote at least 3 or 4 programs in hopes of rearranging the registers in such a way that the program wouldn't crash. I did this due to the fact that the program would crash during certain instructions and other times it did not. There was no stated reason as to why. Eventually after bashing my face into the table a few times, I realized that maybe the program can't use registers that have been modified by an interrupt. I decided to push all registers, not including R0 on to the stack. This resulted in the program working...then crashing. I then made sure that R0 was also pushed, meaning that the program had no register data modified and finally there were no crashes.

5 Conclusion

It works. All 4 lights blink and will respond to button input. It also blinks. I love blinking.

Probably the most impressive part for me was how incredibly important interrupts are, yet how horrible they are to implement. I can not imagine how a nested interrupt would ever work. Easily the most bugs came from just simply trying to implement it correct and I am very very very sure that if I rewrote the program, I would still get program crashes from interrupts.

6 References

1. ARM Architecture
2. "Binary to Gray Code Converter and Grey to Binary Code Converter", www.electrical4u.com
3. Doug Hall, Dr. "ECE 371 Microprocessor system design-hardware, programming and interfacing" Published by Doug Hall,Draft Edition-Winter 2017.
4. ncalculators.com
5. The table I bashed my face into, it told me to stop using interrupts in general.