Ryan Bentz

ECE 371

Design Project 2

12/4/17

List of Deliverables

Part I

A. Problem Description
B. High-Level Algorithm
C. Low-Level Algorithm
D. Project Code
E. Design Log
F. Supporting Documentation

Part II

G. Problem Description
H. High-Level Algorithm
I. Low-Level Algorithm
J. Project Code
K. Design Log
L. Supporting Documentation

Part III

M. Problem Description
N. High-Level Algorithm
O. Low-Level Algorithm
P. Project Code
Q. Design Log
R. Supporting Documentation

A. Part I - Problem Description

Study the BeagleBone Black System Manual to determine which GPIO pins are connected to the 4 USR LEDs and the logic level required to turn on one of the LEDs. Write the high level algorithm for a program that lights LED0 for 1 second, then adds LED1 so that two LEDS are on for one second, then adds LED2 so that 3 LEDs are on for one second, then adds LED3 so that all 4 LEDs are on for one second, turns off all the LEDs for one second and then repeats the pattern over and over and over. (Theaters used to have lights cycle through a rotating display like this.).

Carefully work through the section of Hall Chapter 4 that describes the AM3358 GPIO pins and the memory mapped registers that control them. As part of this, determine the registers that control the GPIO pins connected to the LEDs. Also study the section that shows how to set up bit templates for working with the GPIO registers. Set up the templates needed for the GPIO pins you are using. Determine values and addresses you need to output a high or output a low on a GPIO pin. Determine the RMW sequence of instructions and addresses required to program the GPIO pins for the LEDs as outputs.

Develop the low level algorithm, as shown in chapter 4, for your program, including the delay loop. Write and carefully check the assembly language program. Build, Load, Run, and Debug the program. Note that to single step through the program easily you can initially use a very small delay constant, so you don't have to single step forever to get from one LED to the next.

B. Part I – High-Level Algorithm

enable clock to GPIO peripheral
configure the GPIO module
set the GPIO pin states
configure the GPIO pin modes

REPEAT
    turn on LED 0
    wait one second
    turn on LED 1
    wait one second
    turn on LED 2
    wait one second
    turn on LED 3
    wait one second
    turn off all the LEDs
    wait one second
UNTIL forever

C. Part I – Low-Level Algorithm

create constants for the LED states and GPIO button
initialize the clock to GPIO 1
    load the address for CM_PER.GPIO1
    RMW the register to enable clocks
set the LED pin states as OFF
    pin states set to off as default
configure the LED pins as output
    load the address for GPIO1.OE
    RMW the register to configure as output

REPEAT
    load the address for GPIO1.DATAOUT
    write word to turn on LED 0
    branch to delay
    write the word to turn on LED 0, 1
    branch to delay
    write the word to turn on LED 0, 1, 2
    branch to delay
    write the word to turn on LED 0, 1, 2, 3
    branch to delay
    write the word to turn off all LEDs (0x00)
    branch to delay
UNTIL forever

DELAY_ROUTINE:
push registers on stack
load counter variable
LOOP:
    NOP
    subtract 1
    IF counter does not equal 0
        return to loop
    ELSE
        pop registers off stack
        return to main

D. Part I – Project Code

See attached code sections.

E. Part I – Design Log

11/10/17:

- Reviewed project specs and began sketching the algorithm for part 1

11/13/17:

- Reviewed the technical reference manual and printed out the GPIO section
- Reviewed the GPIO section made a list of registers and their addresses
- Decided that it would be best to create constants for the registers and their offsets in order to easily access the registers and not have to remember specific templates since I would have to access different GPIO modules and they have the same register offsets but different base addresses
- Began gathering the information of the needed registers such as their addresses and the values need to write to them in order to accomplish the goals
- Decided it would be easier to create constants for each LED since turning on the LED was the same as toggling a bit and it can be done by just writing a value with a 1 in the bit to the register

11/21/17:

- Typed up the high-level algorithm and the low-level algorithm for part 1
- Laid out the planned register usages for part 1 based on the low-level algorithm flow
- Revised the low-level algorithm for more optimal usages of registers
- Reviewed chapter 4 of the textbook for the delay loop

11/22/17:

- Converted low-level code to algorithm
- Had a problem lighting LEDs 1, 2, 3 because the Beagle Bone Black manual has the LEDs incorrectly shown to GPIO2 when they are all on GPIO1. Note that the user LEDs are on GPIO1 pins 21-24.

- Verified that all LEDs were being accessed and turned on
- Created a delay procedure to have more streamlined code
- Reviewed the clock management and reset section of the technical reference manual to understand what the clock frequency for the GPIO peripheral was
- Ended up eyeballing the delay loop and figuring a decent delay that is close to 1 second
- Finished a working version of the program for part 1

F. Part I - Supporting Documentation

Figure 1 shows the relevant registers, their addresses, and a general description of their purpose.

REGISTER	ADDRESS/OFFSET	DESCRIPTION
CM PER	0x44E0 0000	Base address for peripheral clock gating control
CM_PER_GPIO1_CLKCTRL	0xAC	Manages the GPIO1 clocks.
GPIO1	0x4804 C000	Base address for GPIO 1 peripheral registers
GPIO.OE	0x194	Set the input/output mode for each pin
GPIO.SETDATA	0x13C	Sets the state for each individual pin, 1 = HIGH, 0 = LOW

Figure 1 – List of relevant registers for Part 1

Figure 2 shows the GPIO.OE register and word required to program the LED pins as output. The pin states are configured as input on reset and a zero needs to be written in a bit to configure the pin as output.

GPIO.OE

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
State	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Note								LED3	LED2	LED1	LED0																					
Hex	F				E				1				F				F				F				F				F			

F	E	1	F	F	F	F	F

Figure 2 – Calculation for word to write to GPIO.OE for LED pins enable as output

G. Part II – Problem Description

Carefully read through Hall Chapter 5 and work through the development of the button service program. Your log should parallel the development steps described in the text section for the example program. Develop an initialization list, such as those in the text, with the values needed for this project where the pushbutton is on GPIO1_30 (Show all thinking, labeled templates, etc.). Use this detailed initialization list to modify the required steps of the assembly language program in Figure 5-14 as needed for this project. (Note that, as discussed in the text, you have to modify the startup file to intercept the system IRQ response, instead of using the usual method of hooking the interrupt vector, due to the way the BeagleBone Black is set up).

Write an algorithm for the Button Service procedure that will start the LED rotating action the first time it is called and turn the LED rotating action off the next time it is called. In general, one way to do this is to set aside a memory location that you toggle back and forth to keep track of whether the LEDs are pulsing or not. You then use this to determine whether to start or stop the pulsing when a button press produces an IRQ interrupt request. Write the assembly language program. To test the program, set a breakpoint at the start of the INT_DIRECTOR procedure and run the program. If all is well, execution should go to the breakpoint when the button is pushed. If execution doesn't make it to the breakpoint, mentally work through your code very carefully again to make sure all the bits in the control words, etc. are initialized as they should be. Your log should be helpful in doing this. When execution gets to the INT_DIRECTOR procedure correctly, set a breakpoint at the start of the BUTTON_SVC procedure and step execution to that point. If execution gets to the start of BUTTON_SVC correctly, then you can step through the rest of the program and back to the wait loop to wait for the next button push. When this is all working, save it and make a copy that you will use for developing the next addition to the program.

H. Part II – High-Level Algorithm

*hook the IRQ interrupt with the custom procedure
MAIN:
define the constants to be used in the program
initialize the stack frames
initialize GPIO pins for LEDs
initialize GPIO pin for external interrupt
initialize interrupt controller for GPIO interrupt
enable interrupts in the processor

REPEAT
    do nothing
Until forever

INT DIRECTOR:
check if GPIO was source of interrupt
IF GPIO was source
    check if button was source of interrupt
    IF button was source of interrupt
        go to button service routine
ELSE
    re-enable interrupts

Exit ISR

BUTTON SERVICE:
change LED status
reset interrupts
check LED status
IF LED status == ON
    go to LED flash routine
ELSE
    Exit ISR

LED FLASH:
REPEAT
    turn on LED 0
    wait 1 second
    turn on LED 1
    wait 1 second
    turn on LED 2
    wait 1 second
    turn on LED 3
    wait 1 second
    turn off all LEDs
    wait 1 second
UNTIL forever

I. Part II – Low-Level Algorithm

* hook the IRQ interrupt with the custom procedure

MAIN:
initialize constants for register control
initialize stack for ISR mode
initialize stack for supervisor mode
initialize peripheral clock for GPIO 1
set the GPIO pins for LEDs
    default state is already set
configure the GPIO pins as output
    RMW 0xFE1FFFFF to GPIO_OE
set the GPIO pin interrupt for falling edge detect
    RMW button pin to GPIO_FALLING_DETECT
initialize GPIO pin for external interrupt
    RMW button pin to GPIO_IRQ_STATUS_SET
initialize interrupt controller for GPIO pins
    RMW to INTC_MIR_CLEAR3
    interrupt 98 = bit 3
initialize interrupts in the processor
    copy CPSR to R3

clear bit 7
write back to CPSR

MAIN LOOP:
REPEAT
Do nothing
UNTIL forever

INT DIRECTOR:
push registers on to the stack
check if GPIO1 was source of interrupt
read the INTC_PENDING_IRQ3 register
mask GPIO1 interrupt bit
compare to see if GPIO1 was source
IF GPIO was source of interrupt
check to see if button was source of interrupt
read GPIO IRQ_STATUS register
compare with button pin to see if bit was the source
IF button was the source
go to button service procedure
ELSE
re-enable IRQ interrupts in the
restore register states
return from ISR
ELSE
re-enable IRQ interrupts in the
restore register states
return from ISR

BUTTON SERVICE PROCEDURE:
save registers and return address
push saved program status register on stack
mask lower priority interrupts
turn off GPIO interrupt request
reset interrupt controller IRQ requests
change LED flash status
reset interrupt controller IRQ request
check LED flash status
IF flash status is ON
go to led flash sequence
ELSE
disable IRQs in processor
pop recent saved program status register
pop recent INT DIRECTOR saved registers
pop stored registers from delay loop
pop first saved program status register
pop first INT DIRECTOR saved registers

restore register states
return from ISR

LED FLASH SEQUENCE:
REPEAT
load the address for GPIO1.DATAOUT
write word to turn on LED 0
branch to delay
write the word to turn on LED 0, 1
branch to delay
write the word to turn on LED 0, 1, 2
branch to delay
write the word to turn on LED 0, 1, 2, 3
branch to delay
write the word to turn off all LEDs (0x00)
branch to delay
UNTIL forever

DELAY_ROUTINE:
push registers on stack
load counter variable
LOOP:
NOP
subtract 1
IF counter does not equal 0
return to loop
ELSE
pop registers off stack
return to main

J. Part II – Project Code

See attached code sections.

K. Part II – Design Log

11/24/17:

- Read the relevant textbook sections to complete part 2
- Wrote first draft of the high-level algorithm for part 2
- Created a framework for the low-level algorithm from the high-level algorithm
- Read the interrupts section of the technical reference manual
- Per the datasheet, GPIO1 has two lines to the interrupt controller: POINTRPEND1 at #98 and POINTERPEND2 and #99. We are going to use POINTERPEND1 at #98. This means that the corresponding IRQ set/clear/read operation in the GPIO peripheral should all be at bank-0.

- Calculated POINTRPEND1 at #98 corresponds to bit-3 of bank-3 on the interrupt controller.

11/29/17:

- Made a decision about how the program should handle checking the button press. Determined that it is best to get the program up and running and it will poll the button variable in the main loop and then go to the LED flash sequence. Best way is to have the LED flash sequence called from the ISR but we can do that if we have time.
- Adjusted the high-level algorithm and wrote a draft of the low-level algorithm.
- Reviewed the textbook for how to hook the interrupt procedure and how to set up the ISR
- Reviewed the technical reference manual for the needed interrupt controller registers and gathered the information for the memory addresses and offsets needed to enable interrupts
- Modified the low-level algorithm to include the necessary steps to enable interrupts
- Finalized draft of the low-level algorithm by converting the algorithm into the assembly comments for the code
- Wrote the first draft of the code of the program for part 2
- Debugged problem with data error when initializing interrupt controller. I put the ending address of the interrupt registers instead of the starting address.
- Debugged problem with forgetting to initialize the falling edge detect. Forgot to put that in the low-level algorithm. Changed accordingly.
- Debugged problem with IRQ generation because I had calculated the interrupt as the wrong bit number. This was causing a memory access error.
- Debugged an issue with the IRQ interrupt generation because I had not finished the INT DIRECTOR routine and the interrupt was triggering before I could test it and not exiting properly. I finished the INT DIRECTOR routine and it enters and exits the ISR properly now.
- Changed the algorithm from setting a flag that was cleared at the start of the program to a variable that stores the status of the LED flashing sequence ON/OFF. Status is updated in the ISR and is checked in MAIN loop and checked again at the end of the flashing sequence. If, at the end of the sequence, the button was pressed during the sequence, it will exit the flash routine and go back to the do-nothing section of MAIN.
- Problem with the IRQ being generated at the start of the program and then not returning. Seems to be a data abort error and problems with the IRQ only happening once at the start of the program. Time to call it for the day and get some food.
- Reviewed the textbook and technical reference manual before bed.
- Reviewed the code and thought about the algorithm some more. Realized that I was not properly testing the button variable. I was moving things in registers and accessing memory that I was not supposed to.

11/30/17:

- Realized from Dr. Hall's explanation in class that part of the problem with the button service ISR was that I was not re-enabling the IRQ interrupt on the processor side. Added code at the end of the ISR routine to re-enable the IRQs. Button IRQ works great now. But, there is a problem with the LED flash sequence. It is not going on and off properly.
- The problem with the LED flash sequence was that when I pressed the button and the ISR changed the variable state, the end of the LED sequence would check the variable to determine if it should stop or not. When it stopped, it would go back to main and then check if it was off and

immediately start again. I changed the comparison in main from checking to see if it was off to checking to see if it was and it works great now.

12/2/17:

- The program design can be improved by making the MAIN LOOP do nothing and have the LED flash sequence run as a subroutine of the ISR. To do this, we need to make the button service routine re-entrant.
- Making the button service routine on itself means we need to re-enable interrupts in the ISR before going to the LED flash sequence. Figure 2 shows the new code flow
- I also fixed part 2 and got rid of the scheme where I pre-define all the register address and compute the memory address by adding base + offset. Now, I just write the direct register number. I think it looks better this way and saves unnecessary lines of instruction. Plus going back to the top of the code to check all the defines is getting tedious.
- Updated part 2 so that the button interrupt procedure interrupts itself. The first problem was figuring out the stack map. There was a bug where I forgot that the when the interrupt to turn off the LED's happens, there was one more push to the stack for the delay loop so we need to account for that and pop that off so we can restore registers and the link register properly.
- Last bug now is that when I stop the flash sequence and restart it, it restarts with the first LED already on. But I think at this point, its fine.
- Finished. All code works good.

L. Part II – Supporting Documentation

Figure 3 shows the relevant registers, their addresses, and a general description of their purpose.

REGISTER	ADDRESS/OFFSET	DESCRIPTION
CM PER	0x44E0 0000	Base address for peripheral clock gating control.
CM_PER_GPIO1_CLKCTRL	0xAC	Manages the GPIO1 clocks.
GPIO1	0x4804 C000	Base address for GPIO 1 peripheral registers.
GPIO.OE	0x194	Set the input/output mode for each pin.
GPIO.SETDATA	0x13C	Sets the state for each individual pin. 1 = HIGH, 0 = LOW
GPIO.FALLINGDETECT	0x14C	Enable/Disable the IRQ for falling edge detect.
GPIO.IRQSTATUS_SET_0	0x34	Enable/Disable IRQ generation.
GPIO.IRQ_STATUS_0	0x2C	Provides core status information for the interrupt handling, showing all active events which have been enabled.
INTCPS	0x4820 0000	Base address for interrupt controller registers.
INTC.MIR_CLEAR_3	0xE8	Unmasks the interrupt (Enable interrupt).
INTC.PENDING_IRQ3	0xF8	Contains the IRQ status after masking.
INTC.CONTROL	0x48	Contains the new interrupt agreement bits (reset IRQ generation).

Figure 3 – List of relevant registers for Part 2

Figure 4 shows an example stack map of what is being pushed and popped on to the stack during the programs normal operation. Everything shown in the stack map is pushed on to the stack and must be popped off before returning back to the MAIN loop.

0x0000			<- Start of program
0x0004	Link Register	1st call to INT DIRECTOR	
0x0008	R3		
0x000C	R2		
0x0010	R1		
0x0014	R0		
0x0018	SPSR	1st call to BUTTON SVC Routine	<- LED sequence turned ON
0x001C	Link Register	Used for the LED flash delay loop	
0x0020	R4		
0x0024	Link Register	2nd call to INT DIRECTOR	
0x0028	R3		
0x002C	R2		
0x0030	R1		
0x0034	R0		
0x0038	SPSR	2nd call to BUTTON SVC Routine	<- LED sequence turned OFF

Figure 4 – Sample stack map for Part 2

M. Part III – Problem Description

For this part of your program, you will use interrupts from Timer3 to determine when to switch from one LED to the next, instead of using a delay loop. The Timer 2 program discussed in text chapter 5 shows you how to set up a timer to produce interrupts at desired time intervals. In ECE 371 Design Project #2 part1, you learned how to use a dedicated memory location to keep track of whether an LED is on or off. For this program, you need to figure out some simple way to keep track of which of the 4 LEDs is on when a timer interrupt occurs and take appropriate action. The timer section of Chapter 5 tells you almost everything you need to add the timer capability to your button program, if you study it VERY carefully. Of course, you are using Timer 3 instead of Timer 2, so you have to modify those parts as needed. You also have to integrate your rotating LED actions into the Program.

N. Part III – High-Level Algorithm

MAIN:
Enable peripheral clocks
Wait for peripherals to be ready
initialize GPIO pins for LEDs
initialize GPIO pin for external interrupt
enable GPIO interrupts in interrupt controller
enable IRQ interrupt in the processor
initialize the timer
enable the timer interrupt in the interrupt controller

REPEAT
    DO nothing
UNTIL forever
END

INT DIRECTOR:
check if GPIO was source of interrupt
IF GPIO was source
    check if button was source of interrupt
    IF button was source
        go to button service routine
    ELSE
        exit ISR
ELSE
    exit ISR
re-enable interrupts
return to main

BUTTON SERVICE ROUTINE:
reset the IRQ flags
IF timer is enabled
    disable the timer
ELSE

enable the timer

TIMER SERVICE ROUTINE:
reset the IRQ flags
IF step variable == 1
    turn on LED0
IF step variable == 2
    turn on LED1
IF step variable == 3
    turn on LED2
IF step variable == 4
    turn on LED3
IF step variable == 5
    turn off all LEDs
increment step variable
IF step variable exceeds max
    reset step variable to 0
exit

O. Part III – Low-Level Algorithm

* Hook the IRQ interrupt with the custom procedure

MAIN:
define needed constants for register control
initialize stack for IRQ and supervisor modes
initialize clocks to peripheral
delay and wait for peripherals to be ready
initialize GPIO pins for LEDs
    configure pin states low
    configure pin modes as output
initialize GPIO pin for external interrupt
    configure pin as falling detect
initialize the timer
    enable auto reload
    set load register with counter value
    set the auto reload value
enable interrupts
    enable GPIO interrupt
    enable timer overflow interrupt
    unmask timer and GPIO interrupts in interrupt controller
    enable interrupts in processor
REPEAT
    Do nothing
UNTIL forever

INTERRUPT SERVICE ROUTINE:

```
push registers on stack
read interrupt controller IRQ status register
IF GPIO was interrupt source
    read GPIO IRQ status register
    IF button was source of interrupt
            go to button service routine
    ELSE
            re-enable IRQs on processor
            pop registers off stack
            exit ISR
ELSE
    read interrupt controller IRQ status register
    IF timer was source of interrupt
            go to timer service routine
    ELSE
            re-enable IRQs on processor
            pop registers off stack
            exit ISR
```

BUTTON SERVICE ROUTINE:

```
push saved program status register on stack
mask lower priority interrupts
reset GPIO IRQ interrupts
re-enable IRQs in processor
IF the timer is enabled
    disable the timer
ELSE
    enable the timer
reload the counter value
disable IRQs for critical region
unmask lower priority interrupts
restore the saved program status register
re-enable IRQ interrupts in processor
pop registers off stack
exit ISR
```

TIMER SERVICE ROUTINE:

```
Read the LED status
IF LED status = 0
    make word for LED 0
IF LED status = 1
    make word LED 0, 1
IF LED status = 2
    make word LED 0, 1, 2
IF LED status = 3
    make word LED 0, 1, 2, 3
```

IF LED status = 4
    make word all LEDs off
write the word to the GPIO DATOUT register
increment the status variable
IF status variable > 4
    reset status variable to 0
reset the timer IRQ flag
reset the interrupt controller IRQ flag
re-enable IRQs in processor

DELAY LOOP:
save used registers and link register on stack
load counter with count down value
REPEAT
    Subtract 1 from counter value
UNTIL counter reaches 0
pop registers and return to MAIN

P. Part III - Project Code

See attached code sections.

Q. Part III – Design Log

11/30/17:

- Read the timer section of the reference manual and created a list of registers that will need to be used and their addresses. Read the interrupts section of the manual to get the interrupt bit for the timer. Timer is at bit-69 which corresponds to bit-5 of bank-2 in the interrupt controller.
- Drafted the high-level algorithm and the low-level algorithm from the part 2 algorithms. Decided that I wanted to have the timer just read a memory location that contains the status of where we are at in the LED flash sequence.
- Beginning part 3 by working on the timer LED update routine. Decided it would be best to have a variable to keep track of the 5 different states of the LED flash sequence and cycle through that as the flash sequence goes. Working on the most efficient way to do the IF-ELSE statement, I think using a series of comparisons and conditional moves would be best to eliminate branch stalls.
- Realized that I could improve the efficiency of the routine by also compiling the end result of the word to write to the GPIO register. Even if an LED is on and the bit is HIGH, we can "write it again" with no effects. This makes it easier to compile the word to be ultimately written and makes the conditional moves more appropriate.
- Finished on the draft of the timer service routine with all conditional moves and now working on the initializing process. Testing the initialization process causes a data abort error. I forgot to add part of the initialization to enable the auto reload.
- Finished draft of the part 3 program. Initial test did not work correctly. First mistake found where I was accessing CM_PER for setting the 32kHz clock instead of CM_DLL.
- Debugged a mistake where I was conflating setting the 32kHz clock with setting the clock to the peripheral itself and was causing me to access incorrect memory locations.

- Debugging a problem with the timer not triggering an IRQ request. Tried changing the counter value but that did not fix the problem. Have not figured out the cause of the problem yet.
- I am experiencing a weird glitch where the program crashes the first time I try to run it after loading but works fine after that. And, I noticed that in all my instructions to push and pop registers, everything after the exclamation mark is greened out as if it is a comment. I am not sure if these are related but it is perplexing. I asked Kevin about the commented out portion of my code and he was not sure what the problem could be. He thinks that the reason my code crashes on first try but runs OK after might have something to do with hanging interrupts.
- Code Composer just crashed and now the weird greened out text of the comments are back to being black. Might have been a text editor error?
- Currently debugging my code to enable/disable the timer. I think I have started out with something too fancy. For some reason, my Boolean logic is not working. After doing some more work on this, I have decided to just do a direct write to the register. I know that auto-reload (bit-1) will always be enabled. So, in order to enable disable, I should just write 0x02 or 0x03 to the register.
- Fixed the issue where I wasn't enabling the timer but now it is still not triggering the interrupt. Was thinking about using timer in compare mode but realized I have to set it up differently for overflow mode.

12/1/17:

- There seems to be a problem with initializing the 32kHz clock causing a data abort error.
- For some reason, it was loading 0x00 for my constant for DM_PLL. Cleaned the project and rebuilt it and it works good.
- Seems to be a problem with being able to construct the addresses to write to registers with the scheme that I set up by making a huge list of constants for every register. It's possible that I am exceeding the program counter offset range so I am getting rid of the constants for every register and putting the exact address in instead.
- Still experiencing a problem with the program not running correctly the first time I connect to target and load the program. Program hangs in Supervisor Call Handler. If I step through the program on the first load, it does fine. If I break it up with period breakpoints it does fine. But running all at once it crashes.
- The timer is working and triggering the interrupt but the INT DIRECTOR was never getting to that part of the routine. Realized I hadn't updated the branch statement. It was directing the routine to exit instead of going to check if the timer was the source.
- Now we are into the actual timer routine to light the LEDs. On first try the LEDs did light but the sequence is not running properly.
- Trying to figure out the bug where every time I load a fresh program it crashes. ITR0 is set which means the raw interrupt flags have been set even though the interrupts are masked. Those interrupts are 0x00000082 are bits 1 and 7. Which is NMI external pin active low and COMMTX MPU subsystem.
- I have decided to start from scratch and slowly introduce sections of code until I find the source of the problem. Part 1 and Part 2 worked fine without this glitch so I suspect it must be something timer related. Now working on just initializing the clocks to the peripherals. It works OK when we just enable the clocks. It works OK when we write to a GPIO register after enabling clocks. But, accessing Timer registers after enabling the peripheral clock is causing problems.
- I think the problem with the crashing every time on restart has something to do with accessing registers rights after enabling the clock to the peripheral. It's possible that the clock signal needs

to time to be set up in the peripheral? It shouldn't need this but it would also explain why it works if I step through and it works if I restart the program without restarting the device. I will have to ask about this. I added a small delay after enabling the clocks and everything is OK now and it runs solid the first time.

- The problem is now that inside the timer service ISR I cannot update any registers to clear the IRQs. I can write to registers inside of INT DIRECTOR, but within the TIMER routine nothing works. There should be no cause for this but I cannot think of what the solution might be. I have double checked all the clocks are enabled correctly and I have checked the code is writing to the correct address and is writing the correct value but for some reason I cannot clear the IRQ flag. This is causing the program to be perpetually interrupted. By the timer overflow. In fact, once the timer overflows, it keeps interrupting even if it hasn't reached overflow. I thought it might be something with the timer continuing to run even when the program was halted in debug, but that bit in the timer register says that the timer will stop on halt in debug so that can't be it.

12/2/17:

- Came in this morning and everything suddenly works perfectly. Timer is working and LEDs are lighting. There is a last little bit with the code because once it turns on all the LEDs it doesn't restart. There is a small bug in the logic. I need to write to a different register for the GPIO.
- I was accessing SETDATAOUT AND CLEARDATAOUT. Easier to just access DATAOUT directly then I just have to worry about one register address.
- Fixed the LED flash sequence logic with the step variable. Everything works great now.
- Last bug to fix is something to do with my enable/disable timer logic. The first time I press the button it is not turning on the timer but every time after that it works OK.
- Adjusting the timer period to get closer to 1s periods.
- Everything works good now to tidy things up. Adjusted the registers used in the timer service routine to be more compact and stay within R0-R3.
- Want to make the button service routine re-entrant for the timer. Added code to make the button service routine re-entrant so that it can be interrupted by the timer.
- Going back to part 2 to make it better and put the LED flash sequence in the interrupt and have the routine be able to interrupt itself.
- Finished. All code works good.

R. Part III – Supporting Documentation

Figure 5 shows an excerpt from the technical reference manual showing the discussion of calculating the timer register value for the desired clock period.

The timer rate equation is as follows:

(FFFF FFFFh – TLDR + 1) × timer Clock period × Clock Divider (PS)

With timer Clock period = 1/ timer Clock frequency and PS = 2(PTV + 1).

As an example, if we consider a timer clock input of 32 kHz, with a PRE field equal to 0, the timer output period is:

Table 20-8. Value and Corresponding Interrupt Period

TLDR	Interrupt period
0000 0000h	37 h
FFFF 0000h	2 s
FFFF FFF0h	500 us
FFFF FFFEh	62.5 us

Figure 5 – Timer calculation discussion from technical reference manual

Calculating the timer load register value for a 1 second clock period is as follows:

$$TLDR = FFFFFFFFh - \left(\frac{8000h}{01h}\right) - 1 = FFFF7FFEh$$

Per the example in the datasheet, a clock source with 32kHz clock and no prescaler division has a 2s interrupt with a TLDR value of FFFF0000h. FFFFh converts to 65535_{10} which is exactly two times the clock source 32768_{10}. Therefore, if we use a timer period value of 8000h, we get the word to write for a 1 second delay in TLDR.

Figure 6 shows the relevant registers, their addresses, and a general description of their purpose.

REGISTER	ADDRESS/OFFSET	DESCRIPTION
CM PER	0x44E0 0000	Base address for peripheral clock gating control.
CM_PER_GPIO1_CLKCTRL	0xAC	Manages the GPIO1 clocks.
GPIO1	0x4804 C000	Base address for GPIO 1 peripheral registers.
GPIO.OE	0x194	Set the input/output mode for each pin.
GPIO.SETDATA	0x13C	Sets the state for each individual pin. 1 = HIGH, 0 = LOW
GPIO.FALLINGDETECT	0x14C	Enable/Disable the IRQ for falling edge detect.
GPIO.IRQSTATUS_SET_0	0x34	Enable/Disable IRQ generation.
GPIO.IRQ_STATUS_0	0x2C	Provides core status information for the interrupt handling, showing all active events which have been enabled.
INTCPS	0x4820 0000	Base address for interrupt controller registers.
INTC.MIR_CLEAR_3	0xE8	Unmasks the interrupt (Enable interrupt). (BUTTON)
INTC.PENDING_IRQ3	0xF8	Contains the IRQ status after masking. (BUTTON)
INTC.MIR_CLEAR_2	0xC8	Unmasks the interrupt (Enable interrupt). (TIMER)
INTC.PENDING_IRQ2	0xD8	Contains the IRQ status after masking. (TIMER)
INTC.CONTROL	0x48	Contains the new interrupt agreement bits (reset IRQ generation).
CM_DLL.CLKSEL_TIMER3_CLK	0x0C	Selects the mux select line for TIMER3 clock.
TIMER3	0x4804 2000	Base address for Timer3 registers.
TIMER.IRQENABLE_SET	0x2C	Enables the timer IRQ for event generation.
TIMER.IRQSTATUS	0x28	Component interrupt request status.
TIMER.TCCR	0x3C	Stores the current value of the timer counter.
TIMER.TCLR	0x38	Timer control register
TIMER.TLDR	0x40	Value loaded on overflow in auto-reload mode.

Figure 6 – List of relevant registers for Part 3

I developed and wrote this program by myself with NO help from anyone except the instructor and/or the T.A. and I did not give any assistance to anyone else.