Project Report

Matt Fleetwood
ECE 371 - 2017
Project 2

Item	Feedback	Item Score	Item Total
Demo: Student demonstrates thorough understanding of their own code, and their project produces the correct output / performs as specified in assignment.	Good understanding	36	40
Source Code: Code is unobfuscated, easy to follow and clearly commented. Generally, comments should not be english descriptions of assembly language instructions	Good comments and structure	15	15
Log: Detailed, was written "as-you-go". Shows steps taken throughout entire project to arrive at final algorithms and source code.		20	20
Report: Includes (at a minimum) the 5 sections specified in assignment. Algorithms are clear and produce desired results. Testing was thorough and handled common/rare cases. Results are correct.		25	25
Total		96	100
Percentage		96	

Abstract: Interrupts represent one method for responding to events in microprocessors. For the Beaglebone Black (BBB) microprocessor this means altering the dynamic memory such that the system responds to a certain event of interest in a desirable way. This project shows how to respond to pushbutton and timer interrupt events. When a button connected to the BBB is pressed, it causes an interrupt that in turn causes a timer to cause interrupts that occur at half-second intervals. Ultimately this timer sets the sequence for the lighting of the four USR LEDs on the BBB similar to how KITT's lights animate in the show Knight Rider.

Methods

In order to accomplish the overall goal of this project, it was useful to consider the overall control structure as several distinct parts. The program starts the light sequence when the pushbutton is pressed and turns it off when the button is pressed again, using the timer to control when the lights turn on and off. This means the control structure for this program can be described in three distinct parts: controlling the LEDs to animate properly (without any interrupts), the pushbutton interrupt, and the timer interrupt.

The project was solved in this sequence as recommended in the project requirements documentation. In general working with these peripherals means finding the base address of the module that contains the control register and using an offset to begin accessing them. By doing so one can manipulate such devices, especially since they are memory-mapped. One method known as Read Modify Write (RMW) was used to access these registers. This means reading or loading a pointer to the correct address that has the register of interest (for instance the INTC_MIR_CLEAR2, GPIO1_FALLINGDETECT, and GPIO1_OE registers), modifying the state of that register such that the new state represents the desired behavior, and then storing the new state back into its location in memory.

A high-level algorithm for the mainline (called MAIN) of this program can be described as follows:

```
Load pointers to GPIO1, GPIO1_CLEARDATAOUT, GPIO1_FALLGDETECT, INTC Config,
INTC_MIR_CLEAR2, CM_PER_TIMER2_CLKCTRL, and Timer 2 IRQENABLE_SET registers.
Initialize aforementioned modules (program LEDs for output, enable falling edge on the
pushbutton, initialize INTC, turn on Timer2CLK and registers,  enable IRQ in CPSR)
MAIN:
        Read button flag state from memory
        IF the button flag state is 0 THEN
            Read timer flag state from memory
            IF timer flag state is 0 THEN
                Return to (MAIN) loop
            ELSE
                Go to UPDATE_LEDS label   (start the LED animation)
        ELSE
            Go to TOGGLE_ANIMATION (read LED_state flag)
```

The main feature missing from the algorithm above are interrupts. If an interrupt happens, the mainline (in this case called MAIN) halts execution wherever it is and the processor switches from user (USR) mode to interrupt (IRQ) mode. The interrupt control structure begins with the code labeled INT_DIRECTOR. This portion of code checks whether an occuring interrupt is from the pushbutton, the timer, or something else. Since the BBB only causes interrupts that are enabled, only the pushbutton or the timer happen for this project. Figure 5 - 3 on page 213 of Microprocessors by Hall, shown below, highlights a useful way to consider how the control structure for this program works.

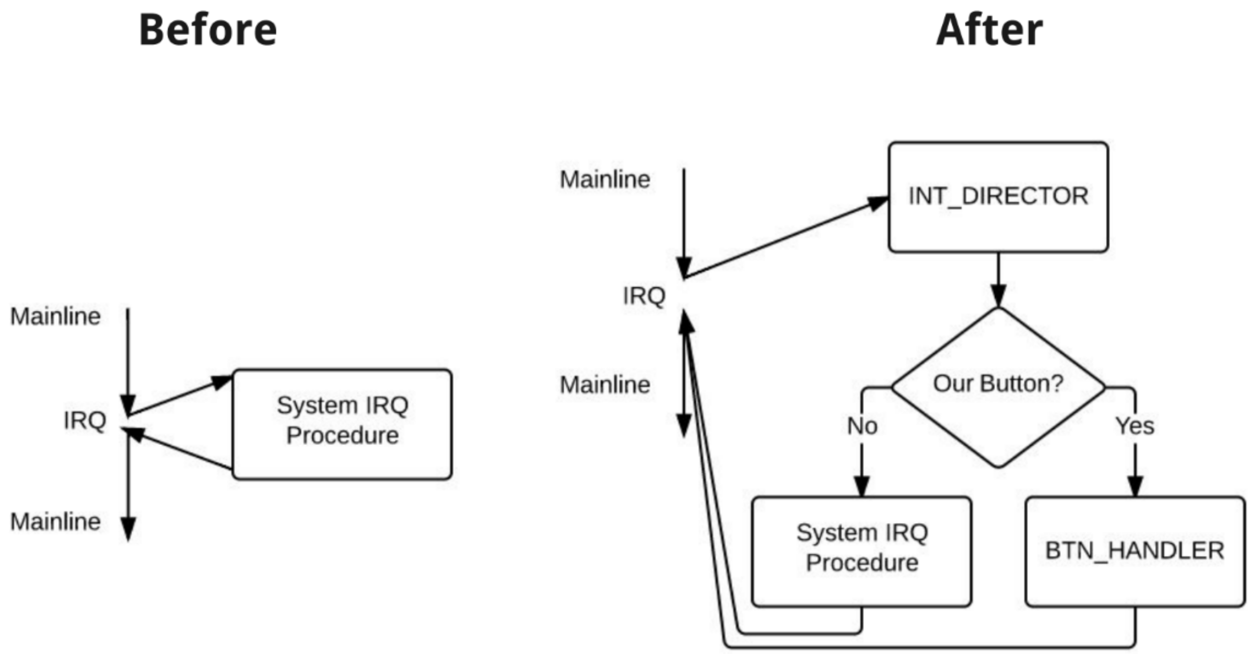


Figure 5 - 3: A) Before dynamic memory is modified to use the INT_DIRECTOR for this program.

B) After dynamic memory is altered to include and use the intended INT_DIRECTOR.

In figure part A, the system operates without the desired control structure for the interrupts that cause and stop the KITT-like animations. Part B shows how the system can respond to an interrupt, such as the button being pressed. For this project the above diagram would have a second decision checking if the interrupt occurred from the timer.

The lights and interrupts for the timer and button ultimately function as a result of using data in memory such as BTTN_PRS, LED_state, TIMER, and SEQUENCE. The BTTN_PRS and TIMER are flags used by the interrupt routines. They have a state of either 0 or 1 and get set inside the interrupt routines. This solves one way to communicate between the different states of the processor. The flags are set inside the routines and cleared in the mainline. The LED_state and SEQUENCE are variables used by the mainline. The LED_state keeps track of whether the LEDs are on or off, and have a state of either 0 or 1. SEQUENCE is used in order to step through the animation sequence as it should appear on the BBB, i.e. the first LED is lit (for half of a second), then the second, then the third, then the fourth, then the third, in the same way until the first LED is reached again. Using flags and variables like this is efficient and excellent advice given by the TA, Fransisco Lopez.

The dynamic memory is altered by modifying the startup_ARMCA8.s file. This is done by replacing the system's IRQ interrupt response with the desired control structure (fig. 5 - 3 part B, plus decision making for the timer). In particular, the ldr pc, [pc, #-8] @0x18 instruction gets replaced with b INT_DIRECTOR. This makes the system go to the new interrupt procedure, but return to the system's procedure if the interrupt was not caused by the button or the timer.

When the program is loaded and run, there are no errors given by Code Composer Studio (CCS). This means the program is syntactically correct. The program executes until power is lost to the BBB, which means that it will cycle between the animation sequence and turning off the LEDs as the pushbutton is pressed.

Testing

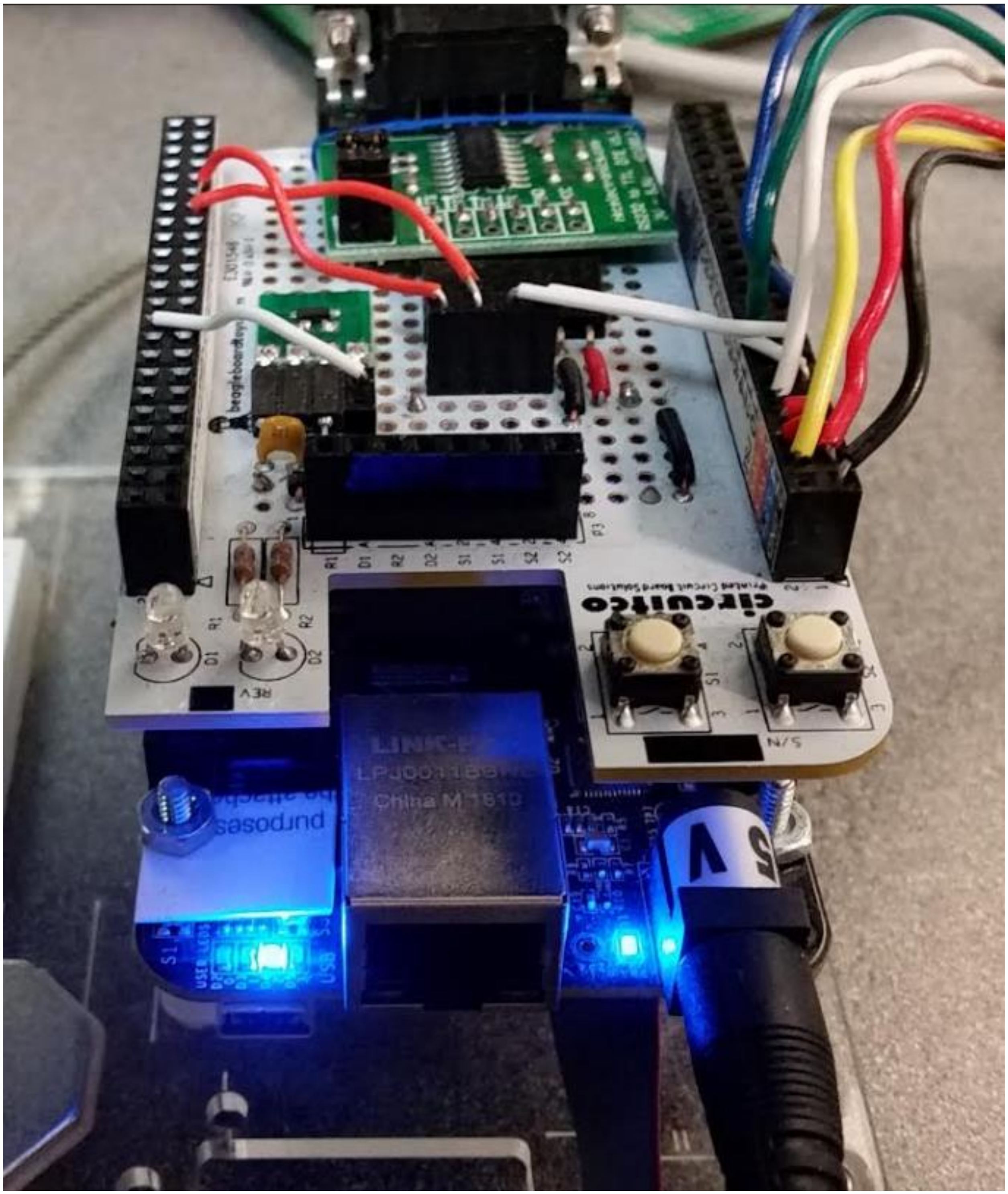
As mentioned previously, this program was developed using the idea of unit testing. Considering the program as a whole to develop is complicated. Breaking up parts of the program into dedicated units results in relatively easier testing than debugging everything at once. The LEDs were made to animate using a delay loop as before in the previous project. Then the pushbutton was enabled to cause interrupts and the startup file was modified to respond to these interrupts. Finally, the timer was incorporated into the other pieces.

The debugger was used extensively to verify the program. This means stepping through the memory browser, evaluating the contents of registers (such as the counter for the SEQUENCE variable and memory flags), and using breakpoints while the program plays so that parts of the code (such as INT_DIRECTOR, BUTTON_SVC, TCHK) could be confirmed as to be executing properly. For instance, the SEQUENCE variable was determined to be functioning correctly as the UPDATE_LEDS label was stepped through.

Results

The product of executing the code below for this project is that the four USR LEDs remain off, until the pushbutton connected to pin 20 (GPIO1_31), i.e. bit 31, is pressed. This causes an interrupt to happen, which is responded to by the new INT_DIRECTOR. In turn, this also causes the timer to begin counting in half-second intervals. The counting in half-seconds is responsible for the timing of when the LEDs turn on and captures the rapid animation of KITT's lights. When the pushbutton is pressed again, the timer is stopped and the LEDs turn off, until the button is pressed again and the process resumes. The pictures below shows one USR LED on the BBB working:


LINK-PP
LPJ0011BBN
China M 1610
USER LEDS
USB
S1


circuitco

Contract

Please type your name below to acknowledge the following contract. This contract must be included exactly as it is written below for all projects submitted in 37x.

This project is exclusively the product of my own work. I designed, developed and tested it myself with no help from anyone except the instructor and/or the T.A, and I did not give help to anyone else. I understand that this project will be cross-analyzed against the projects of all current and previous 37x students for plagiarism, and that evidence of copying, plagiarizing, or otherwise cheating will result in a score of 0 on the project.

Matthew Etcyl Fleetwood

Design Log

Sunday, March 12th - Read the project description. Confused about timers, and how an interrupt works. Started reading through Ch. 5 to get a better understanding. What helped was reading the lecture notes, because eventually it becomes extremely clear how we're supposed to use the interrupt. (It's in bright, shiny letters as a giant disclaimer not to use the hook-and-chain method, but instead just modify RAM in the startup file).

Monday, March 13st - Started working on the high-level algorithm(s) for the first part, part 1. This was somewhat easy because project 1 used similar code to turn on the GPIO1_21 - 24 USR LEDs using the pushbutton connected to pin 20 on bit 31. Ran into some issues getting the pattern for the animation sequence correct (even though it's spelled out in part 1 of the project description…). Originally I thought to set a counter to 1, then logical shift left until 16, then reset the counter to 8. Then logical shift right until 0, and reset the counter to 1. This proved to be less useful later on because of needing to keep track of the LED states.

Wednesday, March 15th - Tested part 1 and it worked. I watched the counter values in the memory debugger, ran my code instruction by instruction and also on play (full-speed). Started designing part 2 and reading through Ch. 5 and the two examples in order to understand how syntactically the interrupt works. Program worked eventually, but only when the code was stepped through and not on play.

Thursday, March 16th - Started running into issues with the computers freezing up. Additionally, the ghost of a tormented TI engineer possessed some of the BBBs such that they would freeze my BUTTON_SVC routine on an LDR instruction. These issues decided to haunt and halt as much progress until Saturday. I e-mailed Eric and Francisco about it, and Francisco kindly agreed to meet on his day off tomorrow to help. I tried saving my code in text files, deleting all of the CCS projects in my current directory, and creating a new project using the saved code. I think at this point I accidently did not configure the .gel file correctly, which is what started to cause new problems.

Friday, March 17th - Met with Francisco but naturally we encountered an error that was an aside to my code's original problem. We fixed the new issue that started today in the initialization section of example 5 - 14 from Hall. Francisco was very helpful and nice, helped me confirm that ultimately I would have done the same steps he did (eliminate lines or add them until something works or breaks, verify addresses using the BBB manual). My error persisted after he left.

Saturday, March 18th - Met with Francisco for his TA hours in order to receive additional help debugging my LDR instruction problem. He was helpful but also became busy helping and signing off others for their demos. Other students helped me get the example working from 5 - 14. Being an example from the book that everyone has access to, and considering students are allowed to discuss examples from the book, this was not collaboration for the project. This was was purely to execute the example, because I claimed that it didn't work for me but they claimed the example did work. Eventually we realized my startup file was not replacing the system's IRQ (I think perhaps because I deleted my project(s) previously and didn't connect the .gel file correctly). The example then worked.

Sunday, March 19th - The example stopped working eventually as I added my own code to the example. Soon I realized my code did work. It was just the boards or the computers I was using (considering CCS and the taskbar would freeze). For instance, I started getting mildly esoteric alerts such as one that seemed to indicate the board was not responding, had lost power, or a combination of that and other bad or possibly worse things occurred. I switched computers again, and began making progress on the pushbutton. It didn't take long for the code to work. What took awhile was the freezes and Memory Mapped Errors, and re-configuring my part 1 for the LED status because somehow, those stopped working correctly. Still, once the computers and boards started working, I was able to start the timer interrupts. I read the section towards the end of the book, which includes a useful example. Francisco explained to me and another student how the general logic for the timers should work. He sat with me for hours helping me debug issues that kept occurring, as well as giving me excellent advice about using the flags and variable(s). The last issue I had was writing an incorrect address during the end of my pushbutton routine. By far, the trickiest part of the project was dealing with the boards becoming trapped on the Memory Mapped errors. It hindered progress, but my advice to others would be: to start as early as humanly possible.

Project Code

@Part 3 of Project 2
@Lights the four USR LEDs on the Beaglebone Black (BBB) such that they animate like KITT's lights from the show
@Knight Rider, when the pushbutton connected to pin 20 (GPIO1_31, or bit 31 in GPIO1) is pushed. The pushbutton

@causes an interrupt, which causes timers to start counting.The timers count for half-second intervals and cause
@interrupts, which serves for the timing to light the LEDs at the right moment.
@Matt Fleetwood, Portland, OR

```
.text
.global _start
.global INT_DIRECTOR
_start:

        LDR     R0, =0x4804C000         @Base address for GPIO1 registers
        ADD     R4, R0, #0x190          @Address of GPIO1_CLEARDATAOUT register
        LDR     R7, =0x01E00000         @Load value to turn off LED on GPIO1_21 - 24
        STR     R7, [R4]                @Write to GPIO1_CLEARDATAOUT register

        @Program GPIPO1_21 - 24 as output
        ADD     R1, R0, #0x0134         @Make GPIO1_OE register address
        LDR     R6, [R1]                @READ current GPIO1 OE register
        LDR     R7, =0xFE1FFFFF         @Word to enable GPIO1_21 - 24 as output (0 enables)
        AND     R6, R7, R6              @Clear bit 21 - 24 (MODIFY)
        STR     R6, [R1]                @WRITE to GPIO1 OE register

        @Detect falling edge on GPIO1_31 and enable to assert POINTRPEND1
        ADD     R1, R0, #0x14C          @R1 = address of GPIO1_FALLINGDETECT register
        MOV     R2, #0x80000000         @Load value for bit 31
        LDR     R3, [R1]                @Read GPIO1_FALLINGDETECT register
        ORR     R3, R3, R2              @Modify (set bit 31)
        STR     R3, [R1]                @Write back
        ADD     R1, R0, #0x34           @Address of GPIO1_IRQSTATUS_SET_0 register
        STR     R2, [R1]                @Enable GPIO1_31 request on POINTRPEND1
```

```
        @Initialize INTC
        LDR     R1, =0x48200000         @Base address for INTC
        MOV     R2, #0x2                @Value to reset INTC
        STR     R2, [R1, #0x10]         @Write to INTC Config register
        MOV     R2, #0x10               @Unmask INTC INT 68, Timer2 interrupt
        STR     R2, [R1, #0xC8]         @Write to INTC_MIR_CLEAR2 register
        MOV     R2, #0x04               @Value to unmask INTC INT 98, GPIOINTA
        STR     R2, [R1, #0xE8]         @Write to INTC_MIR_CLEAR3 register

        @Turn on Timer2 CLK
        MOV     R2, #0x2                @Value to enable Timer2 CLK
        LDR     R1, =0x44E00080         @Address of CM_PER_TIMER2_CLKCTRL
        STR     R2, [R1]                @Turn on
        LDR     R1, =0x44E00508         @Address of PRCMCLKSEL_TIMER2 register
        STR     R2, [R1]                @Select 32 KHz CLK for Timer2

        @Initialize Timer 2 registers, with count, overflow interrupt generation
        LDR     R1, =0x48040000         @Base address for Timer2 registers
        MOV     R2, #0x1                @Value to reset Timer2
        STR     R2, [R1, #0x10]         @Write to Timer2 CFG register
        MOV     R2, #0x2                @Value to enable Overflow interrupt
        STR     R2, [R1, #0x2C]         @Write to Timer2 IRQENABLE_SET
        LDR     R2, =0xFFFFC000         @Count value for 0.5 seconds
        STR     R2, [R1, #0x40]         @Timer2 TLDR load register (Reload value)
        STR     R2, [R1, #0x3C]         @Write to Timer2 TCRR count register
```

```
                @Enable IRQ in CPSR
                MRS         R3, CPSR                @Copy CPSR to R3
                BIC         R3, #0x80               @Clear bit 7
                MSR         CPSR_c, R3              @Write back to CPSR
                MOV         R0, #0                  @Init the LED counter to 1

MAIN:           @Mainline for the program, loops until an interrupt occurs
                LDR         R11, =BTTN_PRS                  @Set ptr to the
memory flag
                LDR         R2, [R11]               @Get the memory flag
                CMP         R2, #0                  @Is the flag 0?
               BNE          TOGGLE_ANIMATION                @No, turn on LED
               LDR          R11, =TIMER             @Get the TIMER memory
flag
               LDR          R2, [R11]               @Get state of TIMER mem
flag
               CMP          R2, #0                  @Check if TIMER flag is 0
               BNE          UPDATE_LEDS                     @No, so update
LEDs
               B            MAIN                    @Loop to main


TOGGLE_ANIMATION:      @Toggles the animation status flag
                MOV         R2, #0                  @Clear button flag
                STR         R2, [R11]               @Write to button flag
                LDR         R11, =LED_state                 @Load ptr to
LED_state flag
                LDR         R2, [R11]               @Get LED_state value
                CMP         R2, #0                  @Is the LED_state value 0?
                BEQ         START_ANIMATION                 @Yes, so start LED
animation
                B           STOP_ANIMATION                  @No, so stop the
LED animation


START_ANIMATION: @Start the timer and change the LED_state flag to 1
                MOV         R2, #1                  @Set new LED_state to 1
                STR         R2, [R11]               @Write to LED_state
                MOV         R2, #0x03               @Load val to auto reload
timer and start
                LDR         R11, =0x48040038        @Address of Timer2 TCLR
register
                STR         R2, [R11]               @Write to TCLR register
                B           MAIN                    @Return to the mainline
```

```
STOP_ANIMATION:         @Turns off lights, sets LED_state to 0, auto reloads timer and
starts it
                BL      OFF                     @Turn all LEDs off
                MOV     R2, #0                  @Set new LED_state to 1
                STR     R2, [R11]               @Write to LED_state
                MOV     R2, #0x0                @Load val to auto reload
timer and start
                LDR     R11, =0x48040038        @Address of Timer2 TCLR
register
                STR     R2, [R11]               @Write to TCLR register
                B       MAIN                    @Return to mainline

UPDATE_LEDS:            @Clears TIMER flag, turns off lights, then begins the animation
sequence
            LDR     R11, =TIMER             @Get TIMER flag address
              MOV       R2, #0                  @Value to clear flag
              STR       R2, [R11]               @Write back to the flag
              BL    OFF                     @Turn off the LEDs
              LDR   R11, =SEQUENCE              @Get the first element in
SEQUENCE
              LDR       R2, [R11, R0]           @Add the offset in R0 to get
to the next element in SEQUENCE
              LSL       R5, R2, #21             @Shift counter left by 20 to
map to GPIO pins
                LDR     R6, =0x4804C194             @Load address of
GPIO1_SETDATAOUT register
                STR     R5, [R6]                @Write to
GPIO1_SETDATAOUT register
                ADD     R0, R0, #4              @Increment the counter
                CMP     R0, #24                     @Check if the
counter has reached the end
                MOVEQ       R0, #0                  @Yes, so reset it with
0
                B       MAIN                    @Return to the mainline


OFF:            @Turns the LEDs off
                STMFD   SP!, {R4 - R10, LR}         @Save all used
registers
                LDR     R8, =0x4804C000             @Base address for
GPIO1 registers
                ADD     R4, R8, #0x190              @Address of
GPIO1_CLEARDATAOUT register
```

```
            LDR     R7, =0x01E00000         @Load value to turn
off LED on GPIO1_21 - 24
            STR     R7, [R4]            @Write to
GPIO1_CLEARDATAOUT register
            LDMFD   SP!, {R4 - R10, PC}     @restore all used
registers

INT_DIRECTOR:       @Control structure for responding to interrupts (pushbutton and
timer used here)
            STMFD       SP!, {R0-R3, LR}        @Push registers on
stack
            LDR     R0, =0x482000F8         @Address of INTC-
PENDING_IRQ3 register
            LDR     R1, [R0]            @Read INTC-PENDING-
IRQ3 register
            TST     R1, #0x4            @TEST bit 2
            BEQ     TCHK                @Not GPIOINT1A, check if
Timer2, else
            LDR     R0, =0x4804C02C
    @GPIO1_IRQstatus_0
            LDR     R1, [R0]            @Read STATUS register
            TST     R1, #0x80000000         @Check if bit 31 = 1
            BNE     BUTTON_SVC          @If bit 31 = 1, then
button pushed
            LDR     R0, =0x48200048         @INTC_CONTROL
register
            B     RET_MAIN          @If bit 31 = 0, then go back
to wait loop

RET_MAIN:       @Label handles code returning from the interrupt routine(s) to the
mainline
            LDR     R0, =0x48200048         @Address of
INTC_CONTROL register
            MOV     R1, #01             @Value to clear bit 0
            STR     R1, [R0]            @Write to INTC_CONTROL
register
            LDMFD       SP!, {R0-R3, LR}        @Restore registers
            SUBS        PC, LR, #4          @Pass execution on
to wait loop (main) for now

TCHK:       @Checks if the interrupts was from the timer, if true also checks for
overflow (toggling if there is overflow)
            LDR     R1, =0x482000D8         @Address of INTC
PENDING_IRQ2 register
```

```
                LDR         R0, [R1]                @Read value
                TST         R0, #0x10               @Check if interrupt from
Timer2
                BEQ         RET_MAIN                @Not from Timer2 so return
to MAIN loop
                LDR         R1, =0x48040028                 @Address of Timer2
IRQSTATUS register
                LDR         R0, [R1]                @Read value
                TST         R0, #0x2                @Check bit 1
                BNE         LED                     @If Overflow, then toggle
status flag for LED animation
                B           RET_MAIN                @If here, no overflow so
return to MAIN loop

LED:            @Turn off Timer2 interrupt request and enable INTC for next IRQ
                LDR         R1, =0x48040028                 @Address of INTC
PENDING_IRQ2 register
                MOV         R2, #0x2                @Value to turn off
                STR         R2, [R1]                @Write back
                @Toggle status flag
                LDR         R6, =TIMER              @Load ptr to LED_state flag
                MOV         R4, #0x1                @Value to start the timer
                STR         R4, [R6]                @Set the flag
                B           RET_MAIN                @Return to MAIN loop

BUTTON_SVC:                 @Handles interrupt requests from the button
                LDR         R0, =0x4804C02C
    @GPIO1_IRQstatus_0
                LDR         R1, =0x80000000                 @Value turns off
GPIO1_31 Interrupt request
                STR         R1, [R0]                @Write to
GPIO1_IRQSTATUS_0 register

                @Toggle memory flag
                LDR         R0, =BTTN_PRS                   @Load ptr to memory
flag
                MOV         R1, #1                  @Set button flag to 1
                STR         R1, [R0]                @Store the new memory flag
state
                B           RET_MAIN                @Go back to the mainline

.data
```

```
BTTN_PRS:      .word 0x0                                       @Memory flag is 0 if lights
are off, 1 if lights are on
LED_state:        .word 0x0                                     @Memory flag is 0 if the
timer was cleared in MAIN, 1 if it was set in the Timer2 routine
TIMER:                          .word 0x0                                   @Memory flag for the
timer
SEQUENCE:    .word 1,2,4,8,4,2                                 @SEQUENCE represents
the order in which to light the LEDs such that they animate like
                                                                @KITT's lights from Knight
Rider
.end
```