Assignment 5: Basic Microcontroller Programming

ECE 3056 Fall 2016

October 14, 2016

This assignment is intended to provide an understanding of bare metal microcontroller software development using assembly language, and some experience interacting directly with hardware devices. The text of this assignment closely parallels the first half of the demos presented in class, and both the slides and documentation for these demos, and the manuals and datasheets they reference, are a valuable resource for completing the assignment. Once you have finished this assignment, a good measure of understanding is the set of exercises presented at the end of each demo's documentation.

As mentioned in the slides, all of the figures here can be found in the following references:

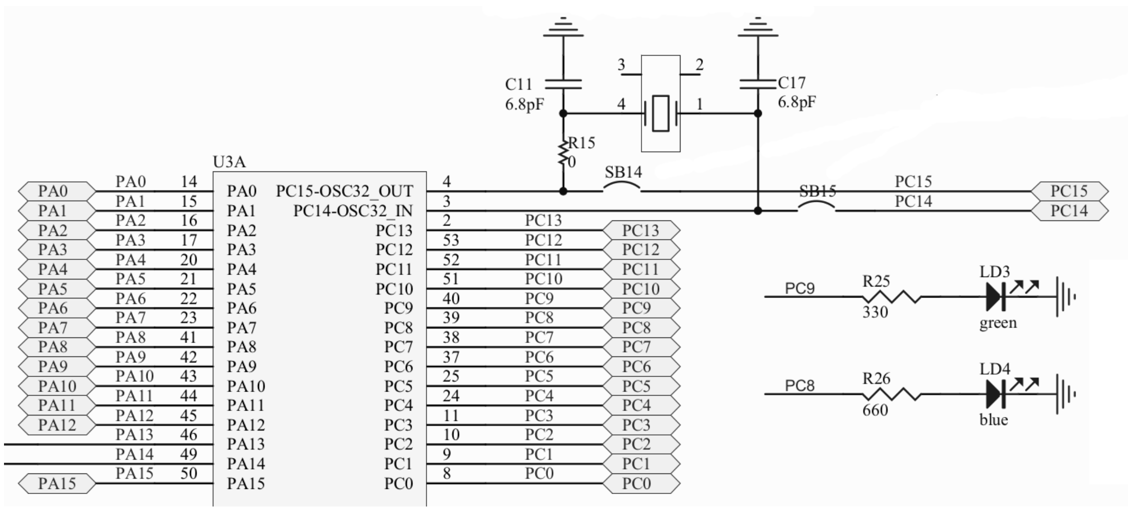
- ST Micro Programming Manual PM0056
  - Processor instruction set reference.
  - Details on NVIC interrupt controller.
- ST Micro Reference Manual RM0041
  - Extensive documentation of peripherals.
  - Memory maps of all peripherals, including GPIO.
  - Documentation of the EXTI interrupt interface.
- ST Micro User Manual UM0919
  - Good summary of board features and pinout.
  - Full schematic of board.
- ST Micro STM32F100RB Datasheet
  - Chip-specific memory map
  - List of chip features and pinout.

It is recommended that you start working through the guided portion of this assignment with the provided `boilerplate.s` assembly file. The word "boilerplate" in this context refers to code that appears frequently, unaltered. In this case, the initial `AREA` directive and mostly-empty vector table constitute boilerplate. The reason for this name is shrouded in mystery, and the current Wikipedia entry shares an uncited folk etymology. This contains a minimal amount of code needed to get started, including a interrupt vector table, which sets the initial stack to 1kB into the SRAM and an initial program counter set to the address of the `Reset_Handler` label.

Start a new project for the STM32F100RB containing only the included boilerplate code. Be sure to answer "no" to the dialogue box asking if you would like to link in the standard library. You will not need any of its functionality since you will be interfacing with the hardware directly.

1 Turning on the Lights

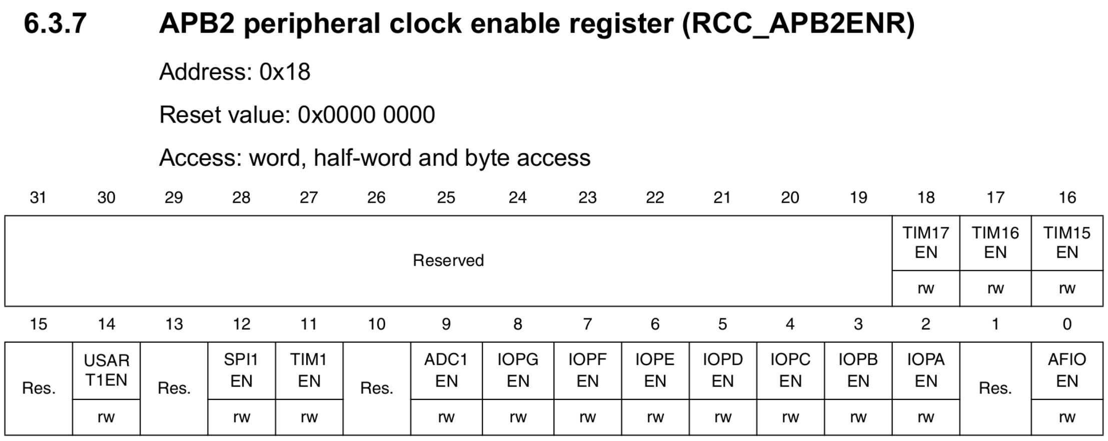
The first task is using the GPIO (General Purpose I/O) pins on our board as output, enabling us to control the LEDs on the board. The GPIO pins are arranged into *port*s of 16 bits each, identified by a letter. As shown in the schematic from UM0919,



the LEDs are connected to bits 8 and 9 of GPIO port C. These pins can be configured by software to be input or output, according to the needs of the application. They default to being inputs and must be reconfigured as outputs before they can be used.

1.1 Enabling the Clock

In order to use the GPIO pins, the clock for the port must be enabled. This clock is used to synchronize the copying of the data register contents to and from the pins and without the clock it is impossible to read or write to the GPIO pins. We can see in this image from RM0041 that this is done by setting bit 4 of `RCC_APB2ENR`:



6.3.7 APB2 peripheral clock enable register (RCC_APB2ENR)

Address: 0x18

Reset value: 0x0000 0000

Access: word, half-word and byte access

31–19	18	17	16
Reserved	TIM17 EN	TIM16 EN	TIM15 EN
	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	USAR T1EN	Res.	SPI1 EN	TIM1 EN	Res.	ADC1 EN	IOPG EN	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.	AFIO EN
	rw		rw	rw		rw	rw	rw	rw	rw	rw	rw	rw		rw

The full description of the function is available on page 84 of RM0041. How do we know which address corresponds to `RCC_APB2ENR`? Note that the image declares `0x18` as its address. This is not the absolute address of this register, but its offset from the start of the `RCC` address space. `RCC` starts at 0x40021000 according to the detailed peripheral memory map on RM0041 page 36. This is the address to which we add the offset of 0x18.

Add code to the beginning of your Entry_Handler function to set bit 4 of the RCC_APB2ENR register at address 0x40021018. The first three instructions of Demo 0 do this by simply storing 0x10 to this location:

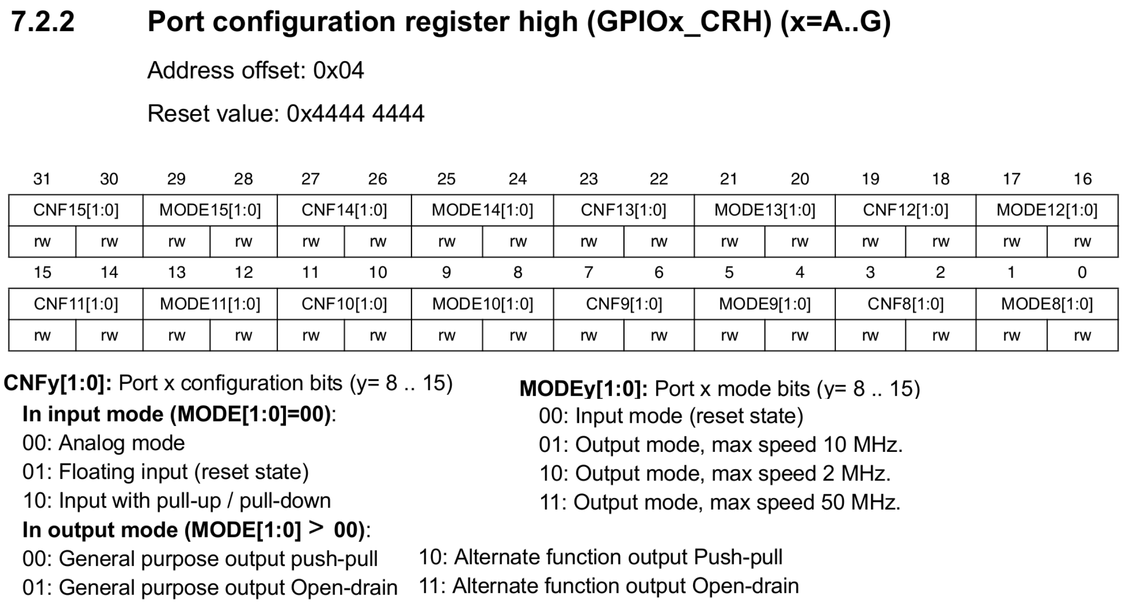
```
ldr r0, =0x40021018
mov r1, #0x10
str r1, [r0]
```

This can be done because, at initialization, none of the other bits are set. A subroutine that enabled the clock for GPIO port C while leaving all other bits in RCC_APB2ENR undisturbed would look like:

```
ldr r0, =0x40021018
ldr r1, [r0]
orr r1, #0x10
str r1, [r0]
```

1.2 Setting the Pin Mode

Within ports, individual pins can be confiured to have any of a varitety of modes by writing to the appropriate configuration register. Each bit's configuration is 4 bits, mapping nicely to a hexidecimal digit. Since each configuration register is 32 bits long, we can only configure 8 GPIO pins with each configuration register. For this reason, each GPIO port has two configuration registers, GPIOx_CRL for the lower 8 bits and GPIOx_CRH for the upper 8 bits. These are documented, on page 111 of RM0041, as:



7.2.2 Port configuration register high (GPIOx_CRH) (x=A..G)

Address offset: 0x04

Reset value: 0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

CNFy[1:0]: Port x configuration bits (y= 8 .. 15)
In input mode (MODE[1:0]=00):
00: Analog mode
01: Floating input (reset state)
10: Input with pull-up / pull-down
In output mode (MODE[1:0] > 00):
00: General purpose output push-pull
01: General purpose output Open-drain
10: Alternate function output Push-pull
11: Alternate function output Open-drain

MODEy[1:0]: Port x mode bits (y= 8 .. 15)
00: Input mode (reset state)
01: Output mode, max speed 10 MHz.
10: Output mode, max speed 2 MHz.
11: Output mode, max speed 50 MHz.

Note that the reset value is 0x44444444, corresponding to CNF being 01 and MODE being 00 for each bit, making all of the ports low-speed inputs by default. This is a sensible default configuration, since it minimizes power draw caused by contention on pins and excessive clock speeds.

Add code to your reset handler to configure port C bits 8 and 9 as low speed push/pull outputs. In Demo 0, this was done by writing `0x22` to `GPIOC_CRH`. This leaves all of the other pins in the upper 8 bits of port C as inputs, but changes their input mode. Since they are unused, this is unimportant. Again, the address to use to is the sum of the GPIO port C base address from UM0041 page 36 and the offset of `CRH` from UM0041 page 111. The instructions that do this in Demo 0 are:

```
ldr r0, =0x40011004
mov r1, #0x22
str r1, [r0]
```

1.3 Writing to the Pin

Once the GPIO port has had its clock enabled and been configured as an output, it becomes possible to change the state of the pins by writing to the `ODR` (output data register). Each GPIO port has a single 32-bit `ODR` at an address offset of `0x0c`. The lower bits of the `ODR` control the state of their corresponding pins and the upper 16 bits must remain at 0.

Add the following code to your reset handler after the GPIO port is configured. This is a simple loop which will flash both LEDs simultaneously:

```
    ldr r0, =0x4001100c ; GPIOC_ODR

loop
    mov r1, #0x300      ; Set bits 8 and 9
    str r1, [r0]

    mov r2, #0x80000    ; Do nothing 1M times
delay0
    sub r2, #1
    cmp r2, #0
    bne delay0

    mov r1, #0          ; Turn the LEDs off
    str r1, [r0]

    mov r2, #0x80000    ; Do nothing 1M times again
delay1
    sub r2, #1
    cmp r2, #0
    bne delay1

    b loop
```

1.4 Questions

1. What is the address of `GPIOA_ODR`?

2. What is the address of `GPIOD_CRH`?

3. What would I write, and to which address, to enable the clock for only ports B and D?

4. Why is it necessary for the GPIO port to have a clock?

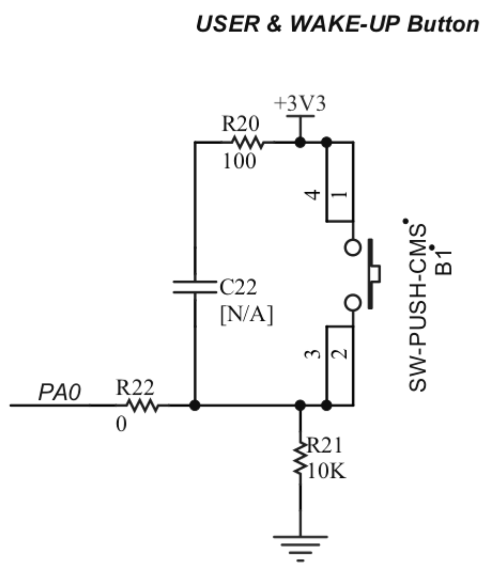
5. How would I configure port B bits 0, 3, and 15 to be outputs?

1.5 Assignment

Restructure the code however you like. It is suggested that you move the delay loop from the given code to a subroutine that takes the number of iterations in the delay as a parameter in a register. Rewrite the example so that it repeatedly flashes "ATL" in morse code (·- - ·-··), using the green LED for the dots and the blue LED for the dashes. The exact timing is unimportant, but the pause between letters should be longer than the pause between dashes or dots, and the dashes should be longer than the dots.

2 Reading Switches

There is a single user-configurable pushbutton on the board, the blue one labeled "user". This is, according to the following schematic cut from UM0919,



connected to bit 0 of GPIO port A. Since GPIO pins are configured by default ot be low-speed floating inputs, and there is already a pull-down on this pushbutton, on startup the pushbutton state can be read simply by reading the `IDR` (input data register) for port A, at address `0x40010808`, but only if the clock for port A is enabled in the `RCC` registers.

Create a new directory and a new Keil project based on your code from the previous part. Keep the code that enables the clock and sets the mode for port C. Modify this code to enable the clock for port A by writing `0x14` instead of `0x10` to `RCC_APB2ENR`. Remove everything past this initialization step.

2.1 Setting the Pin Mode as Input

For pedagogical purposes, Demo 1 includes code which is, strictly speaking, unnecessary to set the direction of all of the bits of port A to input. This is performed in just the same way as setting their states to output, by writing to the configuration registers `CRL` and `CRH`. Since the pin in question is bit 0, the register we use is port A `CRL`:

```
        ldr r0, =0x40010800  ; GPIOA_CRL
        ldr r1, =0x44444444  ; Low-speed floating inputs
        str r1, [r0]
```

If the desired mode was not the reset mode, but any of the other modes listed in Section , this would be a necessary step.

2.2 Reading the Pin State

The pin state can be read by loading from `GPIOA_IDR`, which is, according to page 112 of RM0041, 8 bytes from the base address of port A. In Demo 1, this is done as:

```
    ldr r0, =0x40010808
    ldr r0, [r0]
```

Add the following as the main loop of your entry handler. It simply reads the state of the pushbutton switch and lights the LEDs if it is pressed. Make sure this works as expected before moving on.

```
    loop
        ldr r0, =0x40010808     ; GPIOA_IDR
        ldr r0, [r0]            ; Load IDR
        and r0, #1              ; Light LEDs if pressed
        orr r0, r0, r0, lsl #1
        lsl r0, #8
        ldr r1, =0x4001100c
        str r0, [r1]
        b loop
```

2.3 Questions

1. How would the code in Section 2.2 be different if the pushbutton were attached to port A bit 1 instead of bit 0?

2. What values would you write, and to which addresses, to configure port D bits 2, 4, and 9 to be inputs with pull-down resistors?

2.4 Assignment

Rewrite the code to flash the green LED once when the button is pressed and the blue LED once when the button is released. The length of time the blue LED remains lit when the button is released should be proportional, but not necessarily equal, to the amount of time the button was held.

3 Interrupting the Processor

Periodically polling a register in an I/O device is one way to wait for an external device to finish a task, but as the number of devices increases and the desired response time decreases, the number of cycles the processor must spend repeatedly reading the same register instead of doing useful work increases significantly. What we would like is a mechanism that allows an external event to cause the processor to immediately run a subroutine, then continue executing the code it was originally running when it is finished. Nearly every commercially successful processor has had such a mechanism, provided by an `interrupt controller`.

3.1 Adding a Handler to the Interrupt Vector Table

The address of the procedure that gets called when an interrupt occurs is held in the interrupt vector table. This is the samer table that presently contains the initial stack pointer and initial program counter. When an interrupt handler (also known as an interrupt service routine) is called, the present registers `r0` through `r3` and `lr` are pushed to the stack, and the `pc` is moved to `lr`. This makes the process of entering and returning from an interrupt handler exactly like entering and returning from an ordinary subroutine. In fact, when using a C compiler, interrupt service routines can be written just like any other function.

Interrupts triggered by GPIO pins are handled by two devices, the `EXTI` (external interrupt controller), which is fully documented in RM0041, beginning on page 134, and the `NVIC`, (nested vectored interrupt controller), documented in PM0056, beginning on page 118. Interrupts may have priority levels, so that a higher-priority interrupt may only interrupt a lower-priority interrupt. This is beyond the scope of this assignment, and your interrupt will be at the highest priority level.

When an interrupt is triggered, a bit is set in both the `EXTI` and `NVIC` to mark it as pending. This bit must be cleared before the interrupt can be triggered again. The following code for an `ext0` handler does this.

Copy the boilerplate code, including all of the GPIO initialization code, from the previous section to a new assembly file and create a new project. Skip 80 bytes from the "`Reset_Handler + 1`" entry (using the `.space` directive) and add an "`ext0_handler + 1`" entry to the interrupt vector table. Create an `ext0_handler` label after your reset handler. This is the beginning of your interrupt handler. For now, just make it increment a counter at address `0x20000400` and reset the pending bit in both the `NVIC` and `EXTI`:

```
ext0_handler
  push {r4, lr}       ; Save r4 and the link register
  ldr r0, =0x20000400 ; The counter is after the stack.
  ldr r4, [r0]
  add r4, #1
  str r4, [r0]

  ; Clear pending-bit in EXTI; see RM0041 p. 140
  ldr r0, =0x40010414 ; EXTI->pr, see RM0041 p. 140
  mov r1, #1
  str r1, [r0]

  ; Clear pending-bit in interrupt controller
  ldr r0, =0xe000e280 ; NVIC->icpr0; see PM0051 p. 123
```

```
    mov r1, #0x40
    str r1, [r0]

    pop {r4, pc}        ; Restore r4 and return
```

3.2 Configuring the EXTI

The EXTI controller allows you to configure interrupts on the rising or falling edges of inputs, with the restriction that each of the EXTI channels can be triggered by its corresponding bit from only one of the GPIO ports. EXT0, for example, can be triggered by port A bit 0, port B bit 0, etc., and EXT1 can be triggered by port A bit 1, or port B bit 1, and so on. Since our pushbutton is on port A bit 0, we configure EXT0 to trigger, in this case on the rising edge. EXTI_CR is used to select which port causes the interrupt; this register is actually in the AFIO, alternate-function I/O, address space. See RM0041 page 115. A bit corresponding to the interrupt source is then set in EXTI_RTSR to cause an interrupt on the rising edge of the GPIO signal. The same bit would be set in EXTI_FTSR to cause an interrupt on the falling edge of the GPIO signal. EXTI_IMR is then used to unmask the corresponding

Add the following code to your reset handler, after the GPIO initialization. This will set up the EXTI to interrupt the processor when there is a rising edge of port A bit 0.

```
ldr r0, =0x40010008 ; AFIO->exticr1, see RM0041 p. 124
ldr r1, [r0]
bic r1, #0xf        ; Set LSBs to 0, Port A
str r1, [r0]

ldr r0, =0x40010400 ; EXTI base address
mov r1, #1
str r1, [r0, #8]    ; EXTI->rtsr; event 0 rising
str r1, [r0, #0]    ; EXTI->imr; unmask line 0
```

3.3 Configuring the NVIC

The nested vectored interrupt controller, NVIC is responsible for all interrupts in the system.

Add the following code after your EXTI configuration to configure the NVIC. This code sets the priority level for IRQ6, the interrupt request line corresponding to EXT0, to 0.

```
ldr r0, =0xe000e404 ; Address of NVIC->ipr1; PM0056 p. 128
ldr r1, [r0]        ; NVIC->ipr1; PM0056 p. 125
bic r1, #0xff0000   ; Clear bits for IRQ6
str r1, [r0]        ; Set IRQ6 priority to 0
```

3.4 Write a Main Loop

Your program still needs a main loop. Use the following code to reset the counter and then flash the LED for a length of time dependent on the counter value:

```
ldr r3, =0x20000400 ; Clear the counter.
mov r1, #1
str r1, [r3]
```

```
    ldr r0, =0x4001100c ; GPIOC_ODR

loop
    mov r1, #0x300      ; Set bits 8 and 9
    str r1, [r0]

    ldr r2, [r3]        ; Loop 64k * counter times.
    lsl r2, #16
delay0
    sub r2, #1
    cmp r2, #0
    bne delay0

    mov r1, #0          ; Turn the LEDs off
    str r1, [r0]

    mov r2, #0x80000    ; Do nothing 1M times again
delay1
    sub r2, #1
    cmp r2, #0
    bne delay1

    b loop
```

Try this program out. If it is functioning correctly, the length of time the LED is lit should depend on the number of times the button has been pressed.

3.5 Questions

1. How would you modify your code to increment the counter when the button is released instead of when it is pressed?
2. How might you go about making the counter increment twice for each button press, once when it is pressed and again when it is released?
3. What is the instruction to return from an interrupt handler that uses registers `r0`, `r1`, `r3`, and `r4`?

3.6 Assignment

Rewrite the main loop to flash the LEDs in an alternating pattern: green, blue, green, blue, green, blue, ad infinitum. Using an interrupt, make the lights flash simultaneously when the button is held and then return to flashing alternately when it is released. You can poll the pushbutton within your interrupt handler to determine when it is released.

4 What to Turn In

Please turn in:

- A text file containing answers to each of the questions from the "Questions" sections.
- Your assembly source from the project directory of each of the three subassignments, named `5-1.s`, `5-2.s`, and `5-3.s`.
- The final `.axf` file from the project directory for each of the three subassignments, named `5-1.axf`, `5-2.axf`, and `5-3.axf`.

The `.axf` file is an `ELF` binary image of the assembled and linked source code and will be used to verify that your code runs on the board. Please be sure you rename your files as specified, so that grading may proceed in a timely fashion.