

Assignment 6: Microcontroller Programming Part 2

ECE 3056 Fall 2016

October 25, 2016

This assignment expands on Assignment 5, introducing two additional devices: the `systick` timer and the phase-locked loop clock generator. Along with the GPIO ports and external interrupts introduced in the previous assignment, this provides a basic introduction to the STM32F100RB device.

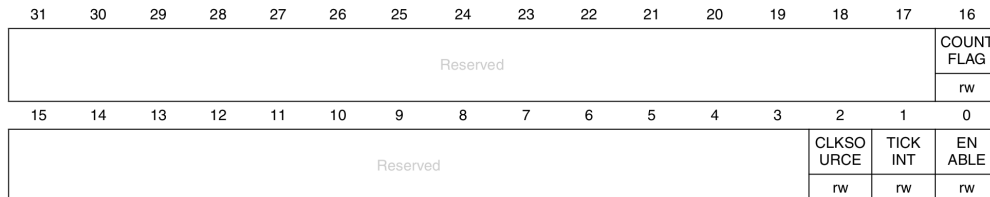
1 A Simple Timer

Timers are a basic and essential peripheral in every kind of computer from high-end servers to microcontrollers. In general, these timers provide a way to trigger an interrupt after a precisely specified amount of time has elapsed. Several types of timers are included in the STM32F100 devices, the simplest of which is the `systick` timer. This timer will periodically trigger an interrupt, at a period which can be configured in software. It operates as a count-down counter that, upon reaching zero, triggers an interrupt and is automatically reloaded with a user-provided value. This is a feature of the ARM Cortex M3 processor core used by the microcontroller, and is present to provide a simple avenue for any operating system that may be present to support preemptive multitasking.

The two memory-mapped registers that must be used to control the `systick` timer are `STK_LOAD` at `0xe000e014`, which is used to select the reset value, and `STK_CTRL` at `0xe000e010`, which is used to start and stop the timer. These are explained in full on PM0056 page 150. A brief summary of the operation of the `systick` timer is that a 24-bit value (the upper byte of the 32-bit word must be 0) is stored into `STK_LOAD` and then 3 is stored into `STK_CTRL` to enable the interrupt and the timer itself. The complete control register, as described on PM0056 page 151, is composed of the following bit fields:

4.5.1 SysTick control and status register (STK_CTRL)

Reset value: 0x0000 0000



Bits 31:17 Reserved, must be kept cleared.

Bit 16 COUNTFLAG:

Returns 1 if timer counted to 0 since last time

Bit 2 **CLKSOURCE**: Clock source selection


Selects the clock source.

0: AHB/8 1: Processor clock (AHB)

Bit 1 **TICKINT**: SysTick exception request enable

Bit 0 **ENABLE**: Counter enable

The interrupt vector for the `systick` timer is at offset `0x3c`, as seen in the vector table layout on PM0056 page 36. Just like the `EXTI0` interrupt vector at `0x005c`, this contains a pointer to the handler for this interrupt.

 Add a label for your `systick` interrupt handler to your code and add an entry for this handler to your interrupt vector table. Add the following code to your reset handler to initialize the `systick` timer:

```
ldr r0, =0xe000e010 ; systick base; PM0056 p. 150
mov r1, #0x100000    ; systick reload value; PM0056 p. 152
str r1, [r0, #4]     ; SYSTICK->load
ldr r1, [r0, #0]     ; SYSTICK->ctrl; PM0056 p. 151
orr r1, #3           ; Set to 3: enable interrupt and counting
str r1, [r0, #0]
```


This will trigger a call to the interrupt handler every 2^{20} counts, or, at the 4 MHz default clock rate with the $\times 8$ divider, approximately once every two seconds. You may add code to your `systick` handler to toggle the state of the LEDs as a debugging measure to verify that the handler is being called at the expected rate.

Remember that returning from the interrupt service routine can be accomplished with a `bx lr` instruction, or a `push {lr}` at the entry to the handler followed by a `pop {pc}` at exit. Also note that Demo 3 is a good reference for a working program using `systick`.

1.1 Questions

1. How would you change the provided code to cause a timer interrupt at a rate of 100Hz, a common operating system tick rate for desktop and server operating systems?
2. What is the function of bit 2 of the `STK_CTRL` register?
3. What problem might arise if the `STK_LOAD` register were loaded with a small value, say 3?

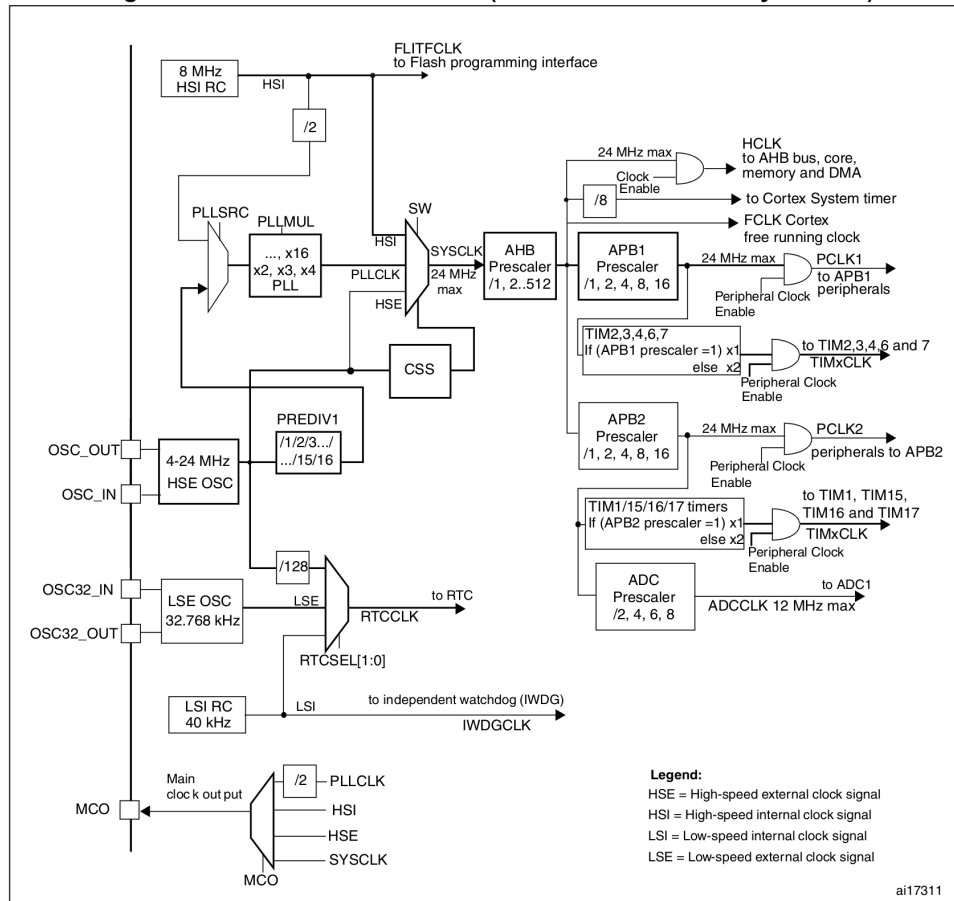
1.2 Assignment

 Using the `systick` timer this time, write a program that will wait for the button to be pressed and released, and then flash the LEDs simultaneously when the button has been released. The duration of the LED's flashing should be approximately equal to the amount of time the button was held, within 10 milliseconds. While waiting for the button to be pressed, and while the button is being held, the LEDs should flash in an alternating pattern, timed by a delay loop (and not the `systick` timer). You do not have to handle any case where the button is held for more than 100 seconds.


2 Using the Phase-Locked Loop

Until this point, we have been using the STM32F100RB's internal 8MHz resistor/capacitor oscillator. This is a quite low-power oscillator, but it is not particularly stable, nor will it enable the device to run at the 24MHz clock rate it is capable of. The following diagram from RM0041 page 71 shows the entire clock network of this microcontroller:

Figure 8. STM32F100xx clock tree (low and medium-density devices)



The external crystal resonator, connected to the `OSC_OUT` and `OSC_IN` pins, resonates at 8MHz, providing a more-stable 8MHz HSE (high-speed external) clock to complement the RC-generated 8MHz HSI (high-speed internal) clock. It is necessary to multiply this by 3 in order to get the desired 24MHz core clock. Phase-locked loops use a control loop to use a slower oscillator to govern a faster one, effectively creating a frequency multiplier. These are very common in modern CPUs, since the frequency stability of crystal oscillators is desired, but it is not possible to build crystal resonators that operate in the GHz range. To run at 24MHz using the phase-locked loop, it is necessary to enable the HSE oscillator, select the HSE oscillator with `PLLSRC`, select an appropriate multiplier `PLLMUL`, and select `PLLCLK` with no prescaler as the system clock source. To simplify this process and avoid re-setting `PREDIV1`, which is a $\times 2$ divider by default, we set the PLL multiplier to $\times 6$.

 Place the following code at the beginning of your reset handler. This enables the external clock, sets the phase-locked loop multiplier to $\times 6$, and selects the PLL as the source of the system clock:

```
ldr r0, =0x40021000
ldr r1, =0x100010 ; RCC->cfgr, PLL mul x6, pll src ext; RM0041 p. 80
str r1, [r0, #4]

ldr r1, [r0] ; RCC->cr, turn on PLL, RM0041 p. 78
orr r1, #0x1000000
```

```

str r1, [r0]


ldr r1, [r0, #4] ; RCC->cfgr, switch system clock to PLL
bic r1, #0x3      ; RM0041 p. 81
orr r1, #2
str r1, [r0, #4]

```

2.1 Questions

1. Both the crystal-controlled oscillator and the internal RC oscillator oscillate at 8MHz. Why might the crystal-controlled oscillator be preferred to the internal one?
2. Say our external crystal was specified at 4MHz instead of 8. How would we modify our code to still run the internal clock at 24MHz?
3. Note that the `systick` timer uses the main processor clock as well. How would the behavior of the code you wrote for the previous section change if you ran the system at 24MHz instead of 8MHz?
4. According to the diagram on RM0041 page 71, what is the slowest clock rate that can be configured?

2.2 Assignment


 For this one, simply turn in the same code as the previous section, but running at 24MHz instead of 8MHz. If the modification is working correctly, the alternation of the lights should be noticeably faster.

3 Putting it All Together

The final demo, Demo 5, provides an example that uses the `systick` timer to perform pulse-width modulation, displaying the result on the LEDs. This uses the GPIO, the `systick` timer, and the interrupts, and sets the clock to 24MHz using the PLL and the external crystal resonator. The pulse-width modulation is achieved by resetting the timer in the `systick` interrupt handler to change the delay. In this part, you may use the same strategy to time the alternate flashing of the LEDs.

In this section, you will build a simple game. The object of the game is to press the pushbutton switch while the green LED is lit. Doing this causes a point to be scored and the game to become more difficult by lighting the green LED for less time. There is no penalty for not pressing the pushbutton when the green LED is lit, but pressing the pushbutton when the blue LED is lit ends the game, and the LEDs will not light again until the reset button is pressed.

3.1 Assignment

 Using an EXTI interrupt for the pushbutton and a `systick` timer interrupt to time the flashing of the LEDs, implement the following simple timing game:

- Initially, the blue LED flashes for approximately one second, followed by the green LED flashing for approximately one second, then back to the blue LED, in an alternating pattern.

- If the pushbutton switch is pressed while the green LED is lit, a point is scored. Both LEDs flash n times when a point is scored, where n is the total number of points that have been scored in the game.
- The blue LED always remains lit for one second, but the green LED remains lit for $\frac{1}{n}$ seconds, where n is the number of points that have been scored. Each point scored makes the game more difficult.
- If the pushbutton is pressed while the blue LED is lit, the game is over and neither LED will light again until the microcontroller is reset.
- There is no penalty for “missing” the green LED. The player may wait as many cycles as they want before pressing the button.

4 What to Turn In

Just like last time, please turn in `.axf` files for each part of the assignment, named `6-1.axf`, `6-2.axf`, and `6-3.axf`, assembly files called `6-1.s`, `6-2.s`, and `6-3.s`, and a text file containing answers to all of the “Questions” subsections.