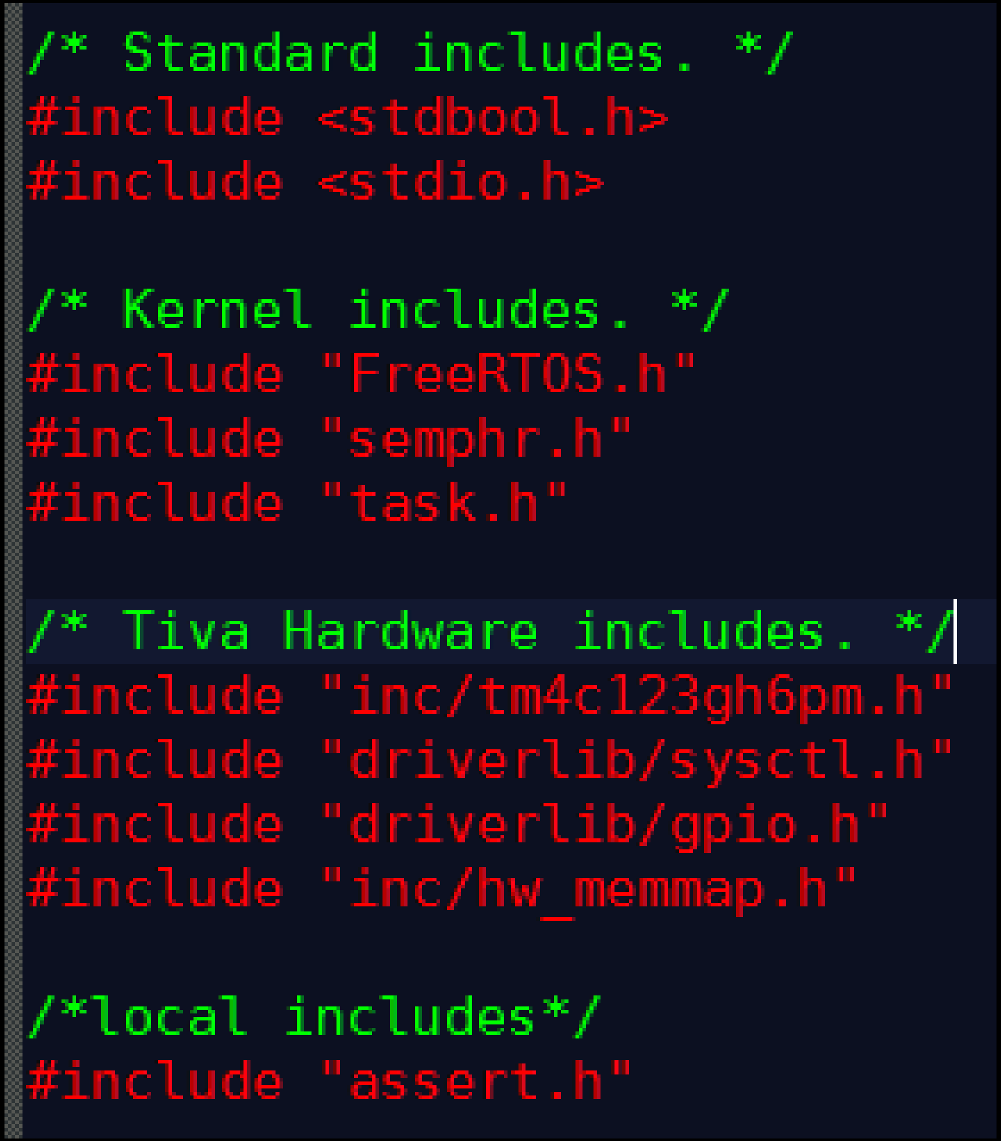
FreeRTOS

- In lab we are switching to RTOS based implementation at week2. This presentation goes over a minimal program.
- All of our remaining programs will build on this one in some way.
- This is the handout for LAB2, You will be completing the same requirements as Lab 1 after modifications.

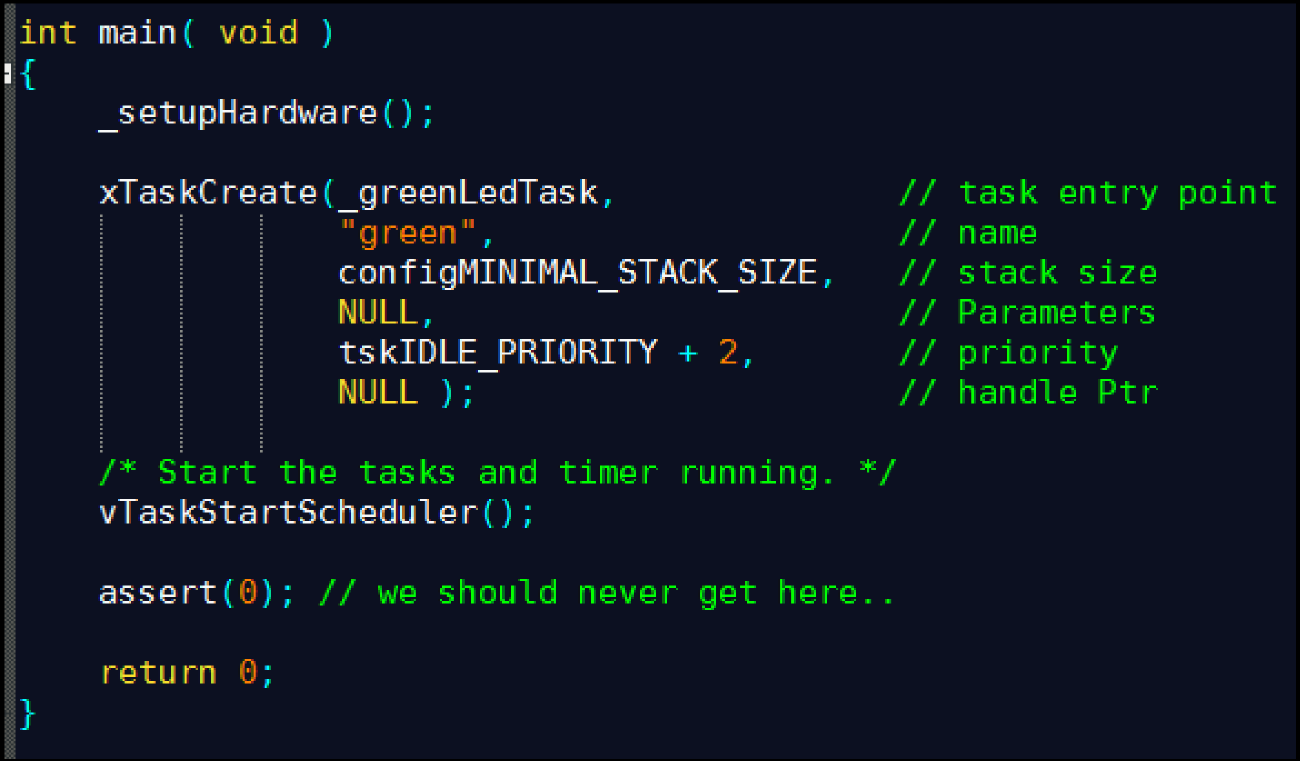
There is a fair amount of required stuff.



```
/* Standard includes. */
#include <stdbool.h>
#include <stdio.h>

/* Kernel includes. */
#include "FreeRTOS.h"
#include "semphr.h"
#include "task.h"

/* Tiva Hardware includes. */
#include "inc/tm4c123gh6pm.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "inc/hw_memmap.h"

/*local includes*/
#include "assert.h"
```

- Adds a whole directory of build dependencies
- Adds at least 5 C files to your build.
- This is the main reason that I wan to keep our builds off a common Makefile
- Will introduce basic RTOS tread concepts way ahead of the book.
- Later programs will add to this but its nearly the bare minimum.
- assert() is a valuable time saver that we will start to use to speed velocity.
- assert() can be a dangerous thing if used in production and should be thought of as temporary!

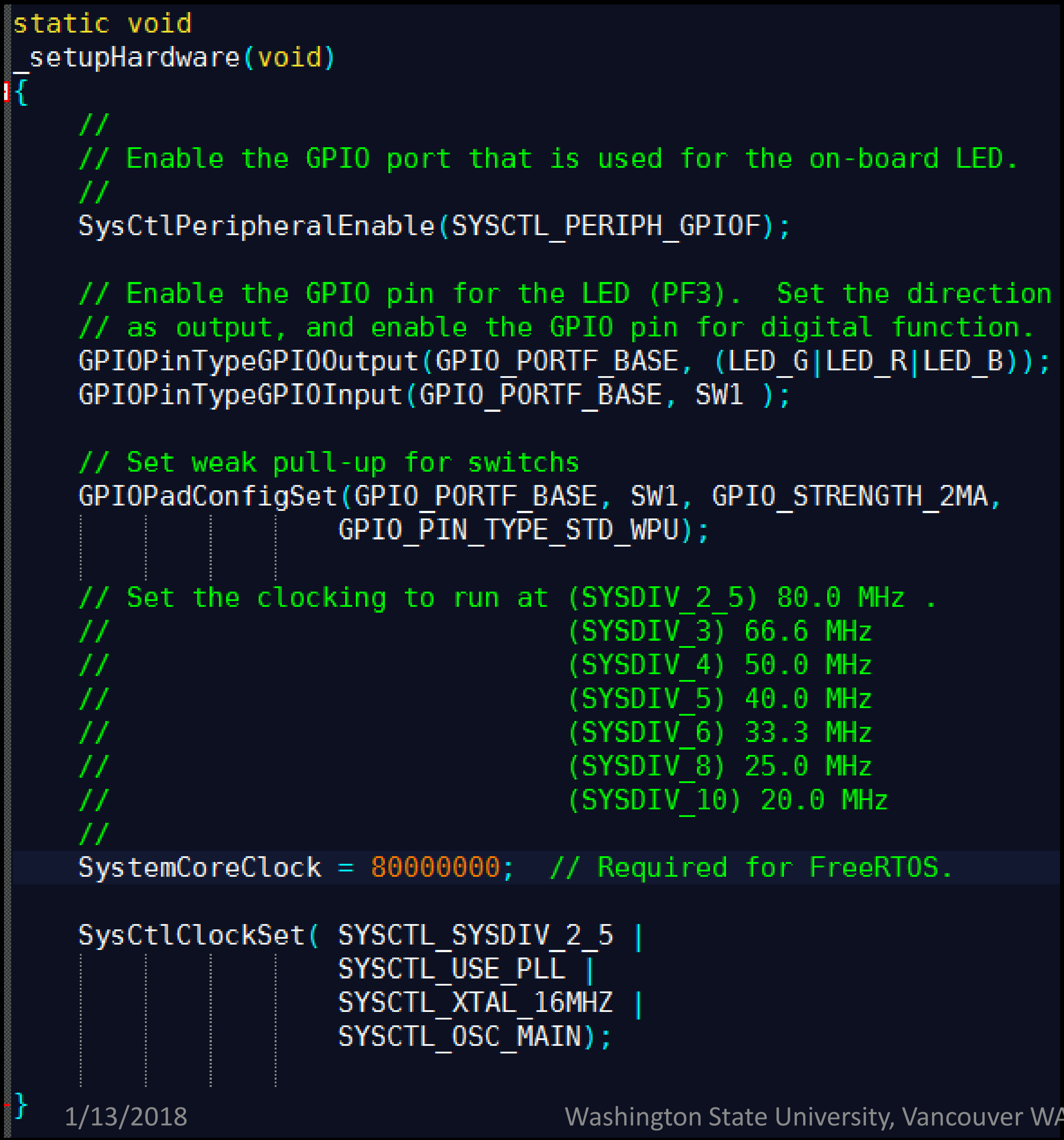
The main() function. (I prefer at the bottom of the module)



```
int main( void )
{
    _setupHardware();

    xTaskCreate(_greenLedTask,               // task entry point
                "green",                     // name
                configMINIMAL_STACK_SIZE,    // stack size
                NULL,                        // Parameters
                tskIDLE_PRIORITY + 2,        // priority
                NULL );                      // handle Ptr

    /* Start the tasks and timer running. */
    vTaskStartScheduler();

    assert(0); // we should never get here..

    return 0;
}
```

- Arduino has a Setup() and Loop() structure, what I refer to as 'superloop'
- Under an RTOS You can have a Setup as I do above in _setupHardware()
- Then I have an area where I setup tasks with multiple calls to the FreeRTOS library xTaskCreate(). This program will have one task
- Nothing happens in the tasks until you start the scheduler which should never return.
- Assert(0), is a programmers convention I encourage use of, Stay Tuned…

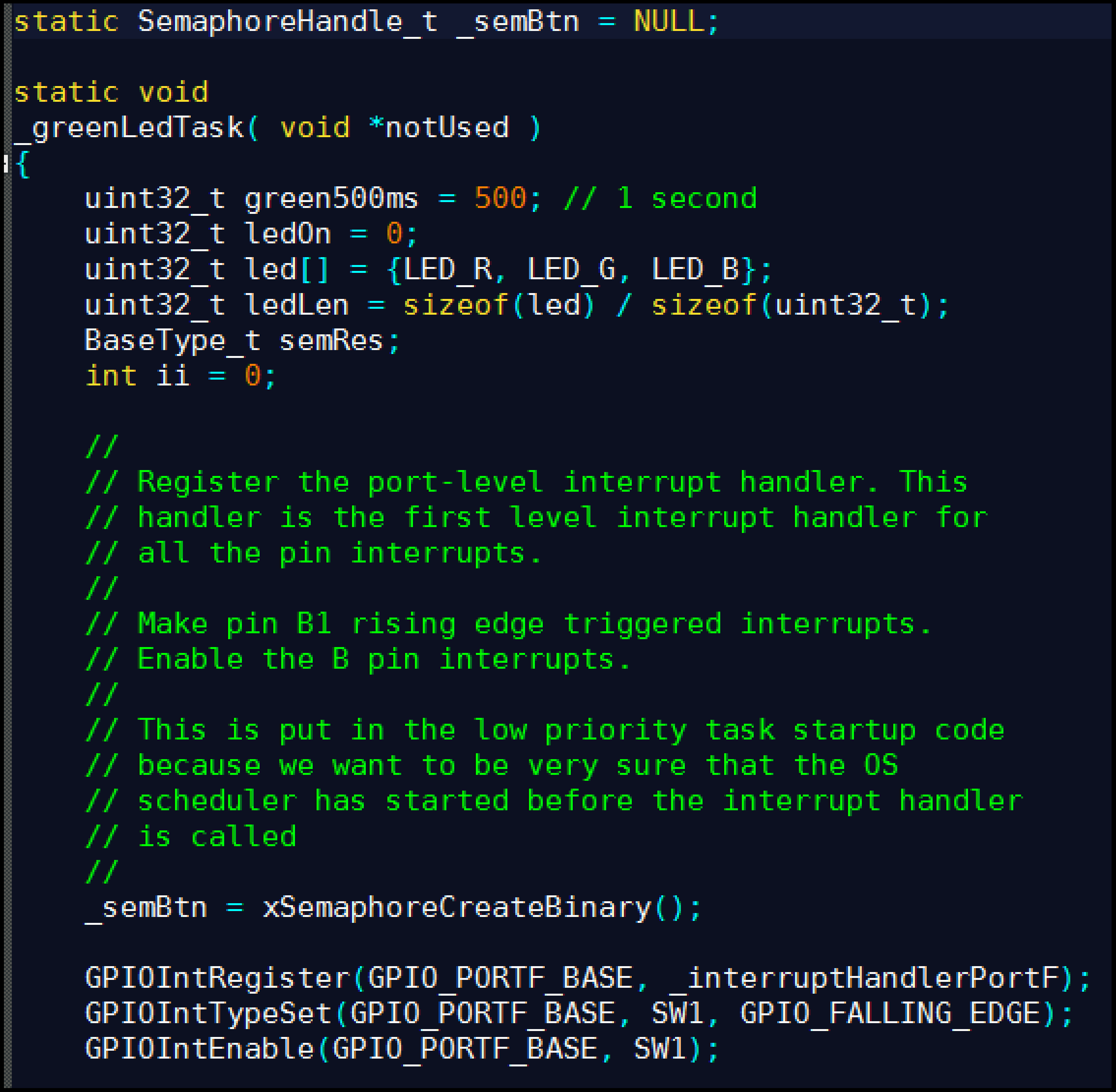
_setupHardware(void)



```
static void
_setupHardware(void)
{
    //
    // Enable the GPIO port that is used for the on-board LED.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

    // Enable the GPIO pin for the LED (PF3).  Set the direction
    // as output, and enable the GPIO pin for digital function.
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, (LED_G|LED_R|LED_B));
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, SW1 );

    // Set weak pull-up for switchs
    GPIOPadConfigSet(GPIO_PORTF_BASE, SW1, GPIO_STRENGTH_2MA,
                    GPIO_PIN_TYPE_STD_WPU);

    // Set the clocking to run at (SYSDIV_2_5) 80.0 MHz .
    //                            (SYSDIV_3) 66.6 MHz
    //                            (SYSDIV_4) 50.0 MHz
    //                            (SYSDIV_5) 40.0 MHz
    //                            (SYSDIV_6) 33.3 MHz
    //                            (SYSDIV_8) 25.0 MHz
    //                            (SYSDIV_10) 20.0 MHz
    //
    SystemCoreClock = 80000000;  // Required for FreeRTOS.

    SysCtlClockSet( SYSCTL_SYSDIV_2_5 |
                    SYSCTL_USE_PLL |
                    SYSCTL_XTAL_16MHZ |
                    SYSCTL_OSC_MAIN);

}
```

- Uses Driver Library Register Access Where Possible
- Sets up system clock
- SystemCoreClock is a required unresolved external of FreeRTOS.
- Setting the system clock via direct register access is difficult.

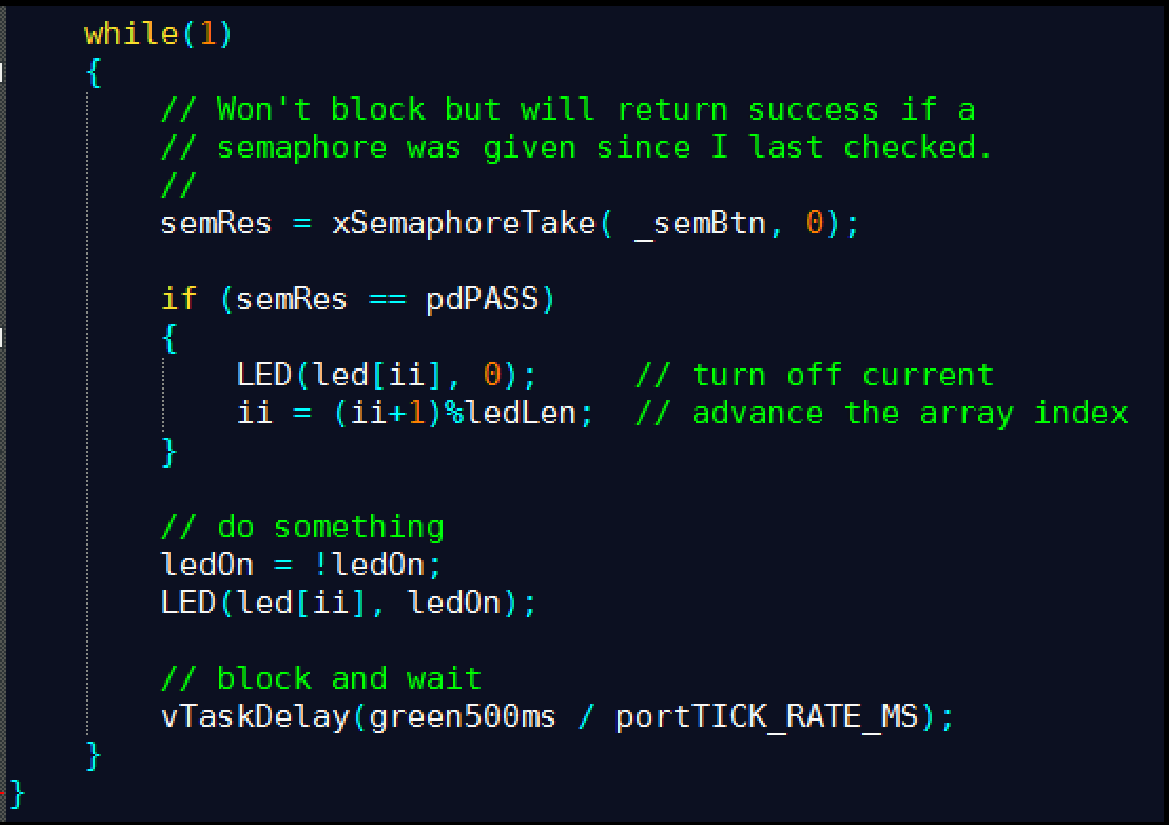
_greenLEDTask() Intro and startup (1 of 2)



```
static SemaphoreHandle_t _semBtn = NULL;

static void
_greenLedTask( void *notUsed )
{
    uint32_t green500ms = 500; // 1 second
    uint32_t ledOn = 0;
    uint32_t led[] = {LED_R, LED_G, LED_B};
    uint32_t ledLen = sizeof(led) / sizeof(uint32_t);
    BaseType_t semRes;
    int ii = 0;

    //
    // Register the port-level interrupt handler. This
    // handler is the first level interrupt handler for
    // all the pin interrupts.
    //
    // Make pin B1 rising edge triggered interrupts.
    // Enable the B pin interrupts.
    //
    // This is put in the low priority task startup code
    // because we want to be very sure that the OS
    // scheduler has started before the interrupt handler
    // is called
    //
    _semBtn = xSemaphoreCreateBinary();

    GPIOIntRegister(GPIO_PORTF_BASE, _interruptHandlerPortF);
    GPIOIntTypeSet(GPIO_PORTF_BASE, SW1, GPIO_FALLING_EDGE);
    GPIOIntEnable(GPIO_PORTF_BASE, SW1);
```

- This task gets called by the scheduler after the main thread starts it.
- I have local context and normal automatic scope which is 'threadsafe'
- Notice that variables at module scope are 'not threadsafe'
- In the bottom half of this screen I perform more hardware setup.. I did not put this in the _hardwareSetup() routine because I want it to occur after scheduling starts.
- _interruptHandlerPortF()

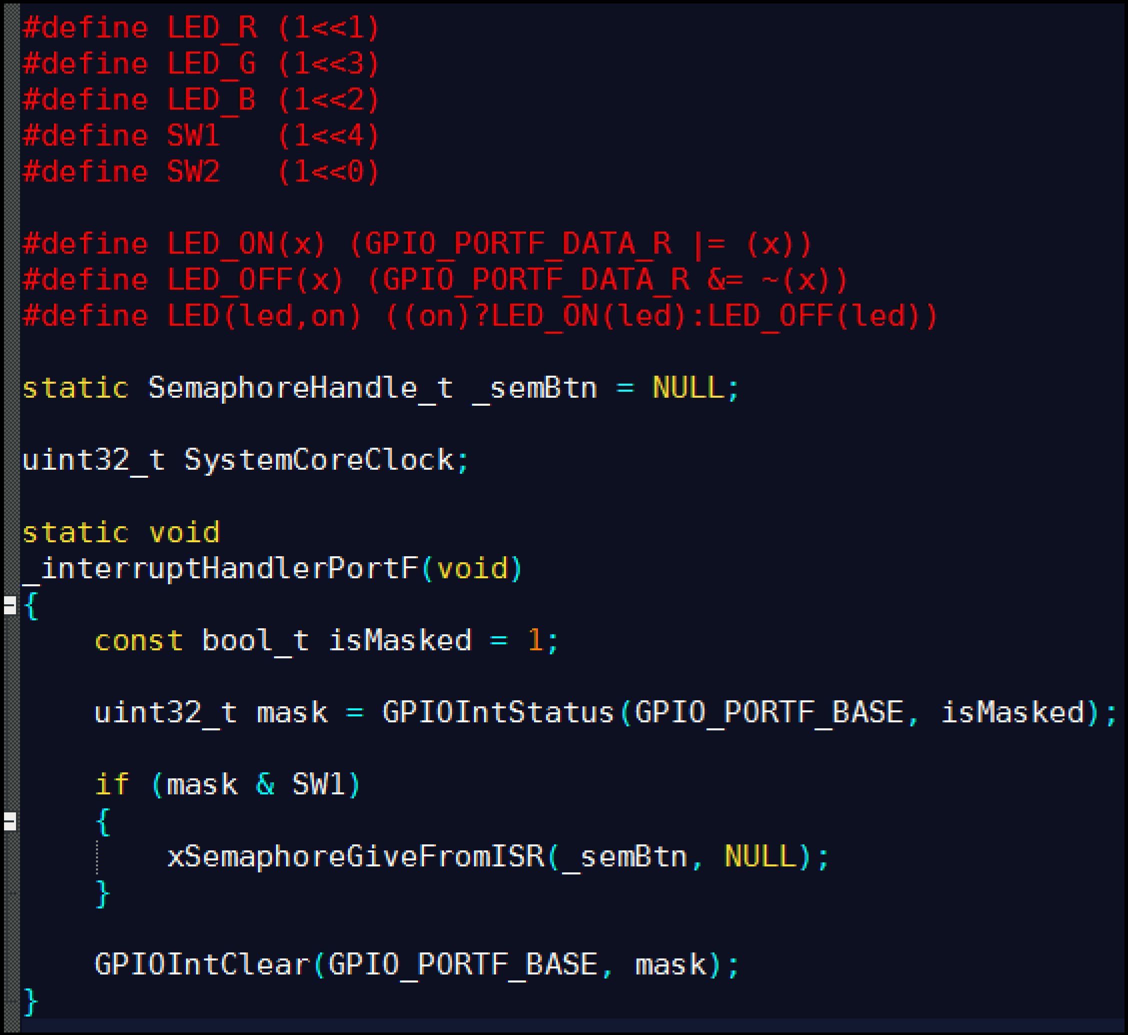
_greenLEDTask() Standard Loop Pattern (2 of 2)



```
    while(1)
    {
        // Won't block but will return success if a
        // semaphore was given since I last checked.
        //
        semRes = xSemaphoreTake( _semBtn, 0);

        if (semRes == pdPASS)
        {
            LED(led[ii], 0);      // turn off current
            ii = (ii+1)%ledLen;   // advance the array index
        }

        // do something
        ledOn = !ledOn;
        LED(led[ii], ledOn);

        // block and wait
        vTaskDelay(green500ms / portTICK_RATE_MS);
    }
}
```

- This is a standard (small) thread loop
- The more simple the better
- Since this one blinks the LED at a fixed rate we block inside the loop.
- The semaphore used is really acting as a mailbox more than a thread synchronization.
- When a button-press is detected `ii` will be incremented thus displaying the next color LED.

_interruptPortF()



```
#define LED_R (1<<1)
#define LED_G (1<<3)
#define LED_B (1<<2)
#define SW1   (1<<4)
#define SW2   (1<<0)

#define LED_ON(x) (GPIO_PORTF_DATA_R |= (x))
#define LED_OFF(x) (GPIO_PORTF_DATA_R &= ~(x))
#define LED(led,on) ((on)?LED_ON(led):LED_OFF(led))

static SemaphoreHandle_t _semBtn = NULL;

uint32_t SystemCoreClock;

static void
_interruptHandlerPortF(void)
{
    const bool_t isMasked = 1;

    uint32_t mask = GPIOIntStatus(GPIO_PORTF_BASE, isMasked);

    if (mask & SW1)
    {
        xSemaphoreGiveFromISR(_semBtn, NULL);
    }

    GPIOIntClear(GPIO_PORTF_BASE, mask);
}
```

- In our hardware, the way we configured it (Remember the beginning of the green LED function).
- When a falling edge is detected on the port F masked bits a processor interrupt will occur and direct control to this routine.
- Need to be sure to clear the interrupt or it will just fire again.
- If our bit was SW1 we give the _semBtn semaphore.