Lab 02 Report: Task Management with FreeRTOS on Zybo

Peter Mowen
peter.mowen@temple.edu

Summary

Lab 02 introduces task management in FreeRTOS on the Zybo Board. Two tasks are created. One task controls the LEDs. The other task suspends and resumes the LED task depending on the switch state.

Introduction

The main objective of Lab 02 is to become familiar with managing tasks in FreeRTOS on the Zybo Board. FreeRTOS is a real time operating system. According to Wikipedia, "A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time applications that process data as it comes in" (Wikipedia, 2020). In this lab, the incoming data is the switch state and the outgoing data is the number displayed on the LEDs.

In FreeRTOS, the developer defines tasks, then FreeRTOS's scheduler controls when each task runs based on the priority assigned to each task by the developer and the tick time. The default tick time of 10ms is used for this lab. A task can be in a running state or a not-running state. This lab will introduce the suspended state, which is a specific type of not-running state.

To achieve the main objective, we are asked to write code for FreeRTOS based on the following design requirements from the lab manual (Dennis Silage, 2020):

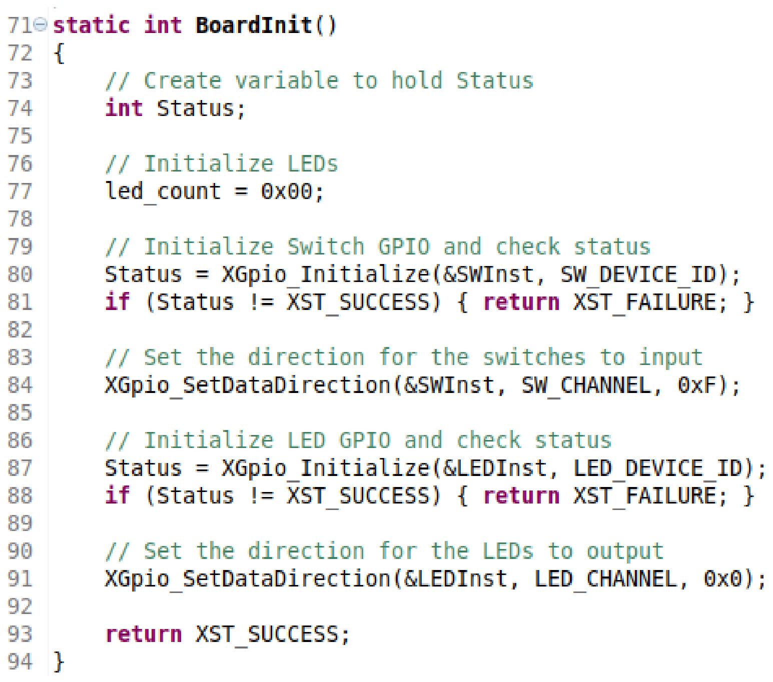
1. Create a FreeRTOS task TaskLED that has the LED count and display start at 0 (0000) and increase as a somewhat random sequence as listed then repeats the sequence starting at 0 (with a delay of approximately 1 second and at the Idle task priority +1 (tskIDLE_PRIORITY+1).
    a. The pattern is: 0000, 1001, 0110, 1010, 0001, 0101, 1000, 0100, 1100, 1110, 0010, 1011, 1101, 0011, 0111, 1111
2. Create a FreeRTOS task TaskSW that reads the SWs at the Idle task priority+ 1. Within the FreeRTOS task TaskSW if SW0 is ON and no other SWs are ON then TaskLED is suspended. If SW1 is ON and no other SWs are ON then TaskLED is resumed.

The hardware design used in Lab 01 to setup the switches and LEDs is reused in this lab since the same hardware is being used.

Discussion

Hardware Setup SDK

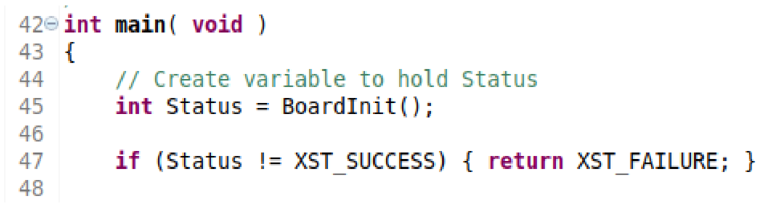
SDK was launched from the same Vivado Hardware design using the "Launch SDK" option in the "File" menu of Vivado HLx. The same include files and preprocessor defines were used and will not be repeated in this report. However, in this program, the following function was made to initialize the hardware:



```c
static int BoardInit()
{
    // Create variable to hold Status
    int Status;

    // Initialize LEDs
    led_count = 0x00;

    // Initialize Switch GPIO and check status
    Status = XGpio_Initialize(&SWInst, SW_DEVICE_ID);
    if (Status != XST_SUCCESS) { return XST_FAILURE; }

    // Set the direction for the switches to input
    XGpio_SetDataDirection(&SWInst, SW_CHANNEL, 0xF);

    // Initialize LED GPIO and check status
    Status = XGpio_Initialize(&LEDInst, LED_DEVICE_ID);
    if (Status != XST_SUCCESS) { return XST_FAILURE; }

    // Set the direction for the LEDs to output
    XGpio_SetDataDirection(&LEDInst, LED_CHANNEL, 0x0);

    return XST_SUCCESS;
}
```

Figure 1: BoardInit() function

This function was called from main before creating the FreeRTOS tasks. The following shows the function call in main:



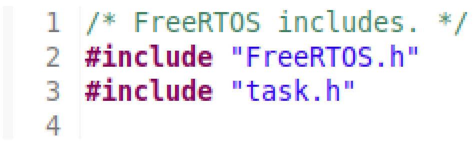
```c
int main( void )
{
    // Create variable to hold Status
    int Status = BoardInit();

    if (Status != XST_SUCCESS) { return XST_FAILURE; }

```

Figure 2: BoardInit() in main function

If the board is not initialized successfully, main will gracefully return the failure.

<u>FreeRTOS Tasks</u>

The following screenshot shows the required include files for FreeRTOS:

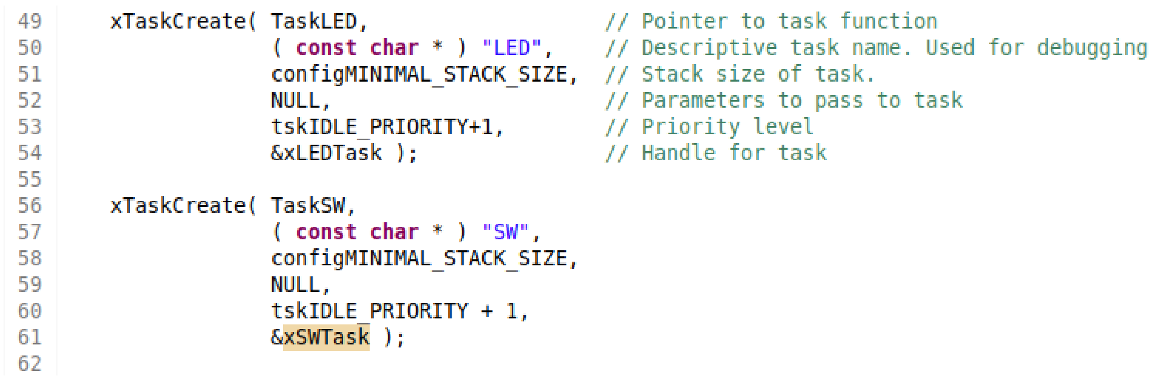


```c
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"

```

Figure 3: Included FreeRTOS files

The first file is the header file for the FreeRTOS kernel. This file is essentially the operating system itself. The second file is the header file for the tasks. This header file imports the functions for creating, suspending, and resuming tasks. It also imports the function for starting the scheduler.

The following screenshot shows the task creation in main:

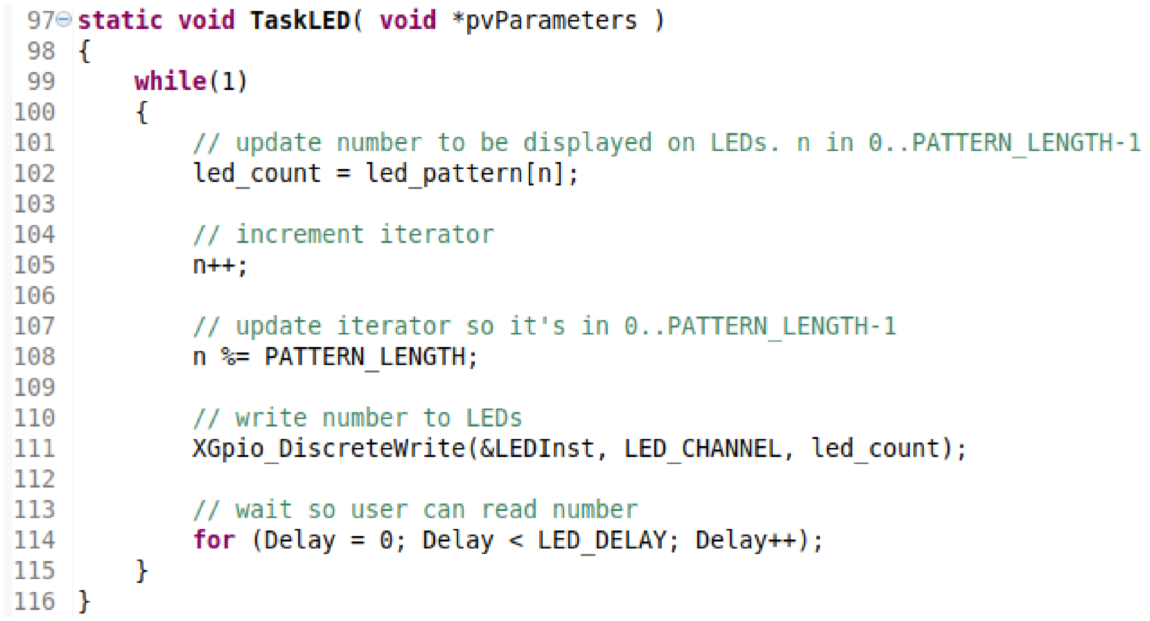


```
    xTaskCreate( TaskLED,                  // Pointer to task function
                 ( const char * ) "LED",    // Descriptive task name. Used for debugging
                 configMINIMAL_STACK_SIZE,  // Stack size of task.
                 NULL,                      // Parameters to pass to task
                 tskIDLE_PRIORITY+1,        // Priority level
                 &xLEDTask );               // Handle for task

    xTaskCreate( TaskSW,
                 ( const char * ) "SW",
                 configMINIMAL_STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY + 1,
                 &xSWTask );

```

Figure 4: FreeRTOS Task creation in main

The `xTaskCreate` function is used to create FreeRTOS tasks. The first parameter is a pointer to the function that is the task. The second parameter is a descriptive name of the task and is only used for debugging. The third parameter is the stack size for the given task. Each task has its own stack size. In this program, it was set to the minimum recommended size. The fourth parameter is for passing parameters into the task. In this program, neither task has any parameters passed into it. The fifth parameter is the task priority. In this program, both are set to one level above the idle priority. Since both tasks have the same priority, the scheduler will alternate between them at every tick. The last parameter is the memory address to the task handle. It is used by the API to make calls to the task. This will be clear after discussing the switch task.

The following is a screenshot of the LED task:

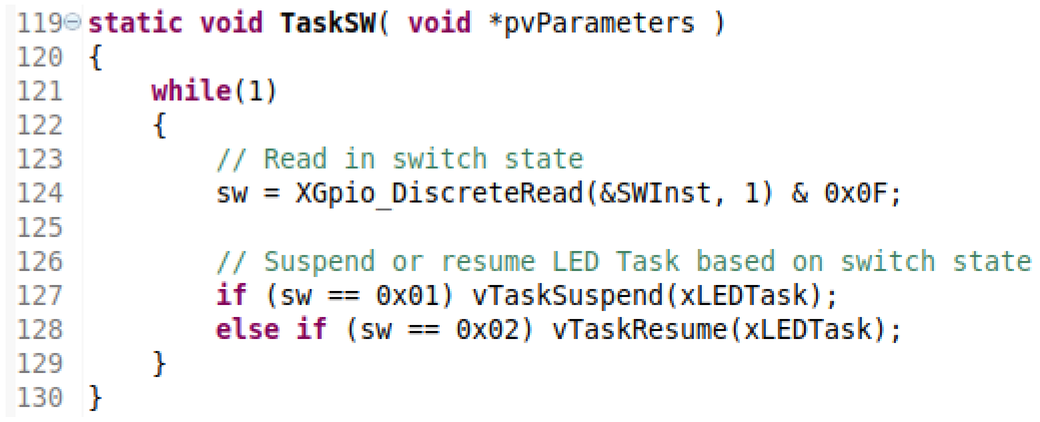


```
static void TaskLED( void *pvParameters )
{
    while(1)
    {
        // update number to be displayed on LEDs. n in 0..PATTERN_LENGTH-1
        led_count = led_pattern[n];

        // increment iterator
        n++;

        // update iterator so it's in 0..PATTERN_LENGTH-1
        n %= PATTERN_LENGTH;

        // write number to LEDs
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, led_count);

        // wait so user can read number
        for (Delay = 0; Delay < LED_DELAY; Delay++);
    }
}
```

Figure 5: TaskLED code

This task iterates through the pattern setup in the `led_pattern` variable using `n` as an iterator. It increments `n`, then takes the modulus of `n` to keep it in the range from 0 to the length of the pattern minus one. This way, `n` never exceeds the `led_pattern` array length. The pattern is defined as a global variable at the beginning of the code. See the appendix for full code. After updating `led_count` with the next element in the pattern and incrementing the iterator, `led_count` is written to the LEDs using the same function from Lab 01. Finally, a `for` loop is used to wait so the user can read the LEDs. With the time added from task switching, this wait time is about one second. All of this is put in a `while(1)` loop so it keeps repeating forever, and the `TaskLED` never dies.

The following is a screenshot of the switch task:

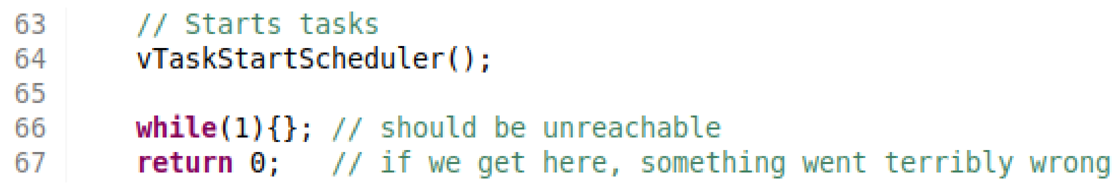


```
static void TaskSW( void *pvParameters )
{
    while(1)
    {
        // Read in switch state
        sw = XGpio_DiscreteRead(&SWInst, 1) & 0x0F;

        // Suspend or resume LED Task based on switch state
        if (sw == 0x01) vTaskSuspend(xLEDTask);
        else if (sw == 0x02) vTaskResume(xLEDTask);
    }
}
```

Figure 6: TaskSW code

In this task, `sw` is assigned the value of the bottom four bits of the switch memory location, just like in Lab 01. The variable `sw` is defined at the beginning of the code. See appendix for full code. Depending on the value of `sw`, the LED task is suspended or resumed. If only the first switch is on, then the LED task is suspended. This is done using the `vTaskSuspend` function and passing it the handle that was assigned to the LED function by `xTaskCreate`. Once the LED task is marked as suspended, the task scheduler will not run it. If only the second switch is on, then the LED task is resumed. Like `vTaskSuspend`, `vTaskResume` takes the handle for the LED task as a parameter. Once the LED task is marked as resumed, the task scheduler will run it again and it picks up wherever it left-off. Note that the whole task is in a `while(1)` loop, it will repeat forever and never let the task die.

The following is a screenshot of the main function calling the scheduler:



```
    // Starts tasks
    vTaskStartScheduler();

    while(1){}; // should be unreachable
    return 0;   // if we get here, something went terribly wrong
```

Figure 7: Start Task Scheduler code

The task scheduler is the part of FreeRTOS that controls which task is currently running. It chooses which task is running based on their priority level. Since both `TaskLED` and `TaskSW` have the same priority level, the scheduler switches between the two tasks every tick. The default tick time is set to 10ms. In the LED task, the `for` loop takes longer than 10ms to run. However, when the scheduler pauses `TaskLED` so `TaskSW` can run, it saves the current state of `TaskLED`. When `TaskLED` resumes, it picks up right where it left-off.

Results

The following are two pictures of lit LEDs from part of the `led_pattern`.

ANALOG DEVICES
LD3(D18) LD2(G14) LD1(M15) LD0(M14)
SW3(T16) SW2(W13) SW1(P15) SW0(G15)

ANALOG DEVICES
LD3(D18) LD2(G14) LD1(M15) LD0(M14)
SW3(T16) SW2(W13) SW1(P15) SW0(G15)

A video demonstrating the solution can be found here: https://youtu.be/Smf4gRIG-gQ

Conclusions

This lab successfully introduces task management in FreeRTOS on the Zybo Board. The hardware concepts from lab 01 were reinforced, including how to setup the hardware drivers, read from hardware, and write to hardware. The new FreeRTOS concepts were successfully grasped. This lab covered how to create tasks, how to suspend one task from another task, how to resume one task from another task, and how the scheduler orchestrates all of it.

Appendix

References

Dennis Silage, P. (2020). Task Management with FreeRTOS on Zybo.

Wikipedia. (2020, January 24). *Real-time operating system*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Real-time_operating_system

C Code

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"

/* Xilinx includes. */
#include "xparameters.h"
#include "xgpio.h"
#include "xstatus.h"
#include "xil_printf.h"

#define SW_DEVICE_ID   XPAR_AXI_GPIO_0_DEVICE_ID  // GPIO device id for Switches
#define LED_DEVICE_ID  XPAR_AXI_GPIO_1_DEVICE_ID  // GPIO device id for LEDs
#define SW_CHANNEL 1                             // GPIO port for Switches
```

```
#define LED_CHANNEL 1                          // GPIO port for LEDs
#define LED_DELAY 23000000                     // Software delay length
#define printf xil_printf                      // smaller, optimized printf

XGpio SWInst, LEDInst;     // GPIO Device driver instance

// LED and Switch related variables
static char led_count;     // Number displayed on LEDs
static char sw;     // Switch state
static char led_pattern[] = {     0b0000, 0b1001, 0b0110, 0b1010, 0b0001,
                                  0b0101, 0b1000, 0b0100, 0b1100, 0b1110,
                                  0b0010, 0b1011, 0b1101, 0b0011, 0b0111,
                                  0b1111};     // Pattern to run on LEDs
#define PATTERN_LENGTH sizeof(led_pattern)/sizeof(led_pattern[0])
unsigned char n = 0;       // pattern iterator

// Function declarations
static void TaskLED( void *pvParameters );     // FreeRTOS Task
static void TaskSW( void *pvParameters );      // FreeRTOS Task
static int BoardInit();                        // Setup LEDs and switches

// FreeRTOS Task Handles
static TaskHandle_t xLEDTask;
static TaskHandle_t xSWTask;

volatile int Delay; // Create variable to hold delay count

/*-----------------------------------------------------------*/
int main( void )
{
	// Create variable to hold Status
	int Status = BoardInit();

	if (Status != XST_SUCCESS) { return XST_FAILURE; }

	xTaskCreate( TaskLED,                          // Pointer to task function
	                    ( const char * ) "LED",    // Descriptive task name.
	                    configMINIMAL_STACK_SIZE, // Stack size of task.
	                    NULL,               // Parameters to pass to task
	                    tskIDLE_PRIORITY+1,        // Priority level
	                    &xLEDTask );               // Handle for task

	xTaskCreate( TaskSW,
	                    ( const char * ) "SW",
	                    configMINIMAL_STACK_SIZE,
	                    NULL,
	                    tskIDLE_PRIORITY + 1,
	                    &xSWTask );

	// Starts tasks
	vTaskStartScheduler();

	while(1){};  // should be unreachable
	return 0;    // if we get here, something went terribly wrong
}

/*-----------------------------------------------------------*/
static int BoardInit()
{
	// Create variable to hold Status
	int Status;

	// Initialize LEDs
	led_count = 0x00;

	// Initialize Switch GPIO and check status
	Status = XGpio_Initialize(&SWInst, SW_DEVICE_ID);
	if (Status != XST_SUCCESS) { return XST_FAILURE; }

	// Set the direction for the switches to input
	XGpio_SetDataDirection(&SWInst, SW_CHANNEL, 0xF);

	// Initialize LED GPIO and check status
	Status = XGpio_Initialize(&LEDInst, LED_DEVICE_ID);
	if (Status != XST_SUCCESS) { return XST_FAILURE; }

	// Set the direction for the LEDs to output
	XGpio_SetDataDirection(&LEDInst, LED_CHANNEL, 0x0);

	return XST_SUCCESS;
}

/*-----------------------------------------------------------*/
static void TaskLED( void *pvParameters )
{
	while(1)
	{
		// update number to be displayed on LEDs. n in 0..PATTERN_LENGTH-1
		led_count = led_pattern[n];

		// increment iterator
		n++;

		// update iterator so it's in 0..PATTERN_LENGTH-1
		n %= PATTERN_LENGTH;

		// write number to LEDs
		XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, led_count);

		// wait so user can read number
		for (Delay = 0; Delay < LED_DELAY; Delay++);
	}
}

/*-----------------------------------------------------------*/
static void TaskSW( void *pvParameters )
{
	while(1)
	{
		// Read in switch state
		sw = XGpio_DiscreteRead(&SWInst, 1) & 0x0F;

		// Suspend or resume LED Task based on switch state
		if (sw == 0x01) vTaskSuspend(xLEDTask);
		else if (sw == 0x02) vTaskResume(xLEDTask);
	}
}
```