Lab 05 Report: Time Slice Scheduling in FreeRTOS on Zybo v2

Peter Mowen
peter.mowen@temple.edu

Summary
Four FreeRTOS tasks are created, each running for a set amount of time, and each lighting one of the LEDs on the Zybo Board while it is running. Nine task management API functions from FreeRTOS are used to create a repeating pattern on the LEDs, representing a periodic task flow.

Introduction
The main objective for this lab is to explore the various API functions used for task management in FreeRTOS. The student is required to create four FreeRTOS tasks that use the following functions from the FreeRTOS API:

1. xTaskCreate()
2. xTaskDelete()
3. xTaskGetHandle()
4. xTaskDelay()
5. xTaskDelayUntil()
6. xTaskAbortDelay()
7. vTaskSuspend()
8. vTaskResume()
9. vTaskPrioritySet()
10. uxTaskPriorityGet()
11. taskYIELD()

However, numbers 3 and 6 were removed from this list and do not need to be used. Each task should run for at least 5 seconds and should light-up one of the LEDs. The tasks will be mapped to the LEDs as follows:

- ILDE Task – NO LEDs
- Task 1 – LED0
- Task 2 – LED1
- Task 3 – LED2
- Task 4 – LED3

This report discusses the hardware setup, then the control logic for this lab, followed by a brief discussion of the results, and a conclusion.

Discussion

Hardware Setup In Vivado HLx

The following is a screenshot of the hardware design used for this lab:

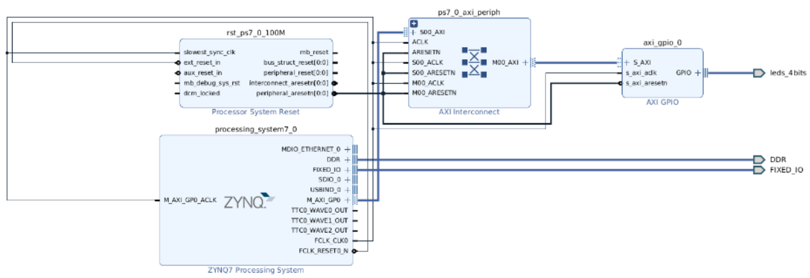

Figure 1: Vivado HLx Hardware Design

The hardware uses the Zynq Processing System IP and the AXI GPIO IP. The GPIO is connected to the LEDs. These will be used to display which FreeRTOS Task is running. No other GPIOs, timers, or interrupts were used in this lab.

Control Logic

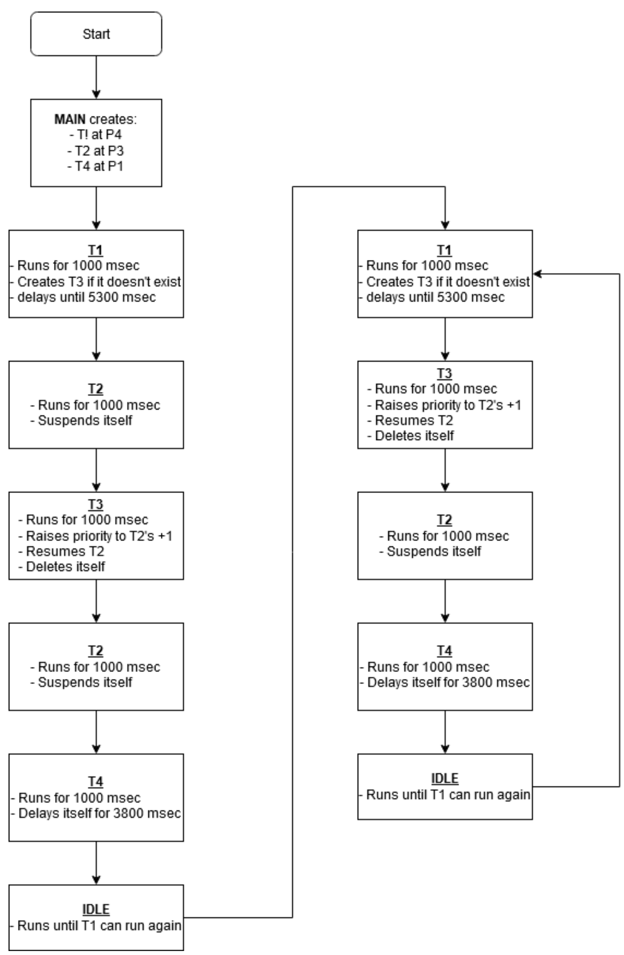

Figure 2: Flow Chart for Tasks

The following a screenshot of the `main` function:

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

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

    xTaskCreate( Task1,             // Pointer to task function
                 ( const char * ) "T3", // Descriptive task name.
                 configMINIMAL_STACK_SIZE,  // Stack size of task.
                 NULL,          // Parameters to pass to task
                 tskIDLE_PRIORITY+4,    // Priority level
                 &xTask1 );         // Handle for task


    xTaskCreate( Task2, ( const char * ) "T2",
                 configMINIMAL_STACK_SIZE, NULL,
                 tskIDLE_PRIORITY+3, &xTask2 );

    xTaskCreate( Task4, ( const char * ) "T4",
                 configMINIMAL_STACK_SIZE, NULL,
                     tskIDLE_PRIORITY+1, &xTask4 );

    // Starts tasks
    vTaskStartScheduler();

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

Figure 3: Screenshot of `main` function

The first thing main does is initialize the board. This configures the LEDs as in previous labs. Next, tasks 1, 2, and 4 are created as in Lab 2. This satisfies the use of API function 1, `xTaskCreate()`. Finally, the scheduler is called to start the tasks. Since Task 1 has the highest priority, it runs first.

The following is a screenshot of the code for Task 1:

```
122 static void Task1( void *pvParameters )
123 {
124     // Setup for delay until
125     TickType_t xLastWakeTime;
126     xLastWakeTime = xTaskGetTickCount();
127     const TickType_t xPeriod = pdMS_TO_TICKS( 5300 );
128
129     while(1)
130     {
131         // Print top of loop message and update LED
132         printf("Top of Task 1. time  = %d ms\n", xTaskGetTickCount()*10);
133         XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_1_LED);
134
135         // Burn Time
136         BurnTime("Task 1", TASK_1_LED);
137
138         // Check to see if task 3 exists. Create it if it doesn't
139         if (xTask3 == NULL)
140         {
141             printf("Task 3 does not exist!\n");
142             printf("Creating Task 3...\n");
143             xTaskCreate( Task3, ( const char * ) "T3",
144                          configMINIMAL_STACK_SIZE, NULL,
145                          tskIDLE_PRIORITY+2, &xTask3 );
146             printf("Task 3 created...\n");
147         }
148         else printf("Task 3 already exists!");
149
150         // Use taskYIELD bc it's required
151         XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED);   // T1 is not running
152         printf("Task 1 is Yielding\n");
153         taskYIELD();    // will come right back bc no other task shares T1's priority
154         printf("Task 1 done Yielding\n");
155         XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_1_LED); // running so LED should be lit
156
157         // Turn-off LED...
158         XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED);
159         // ... and delay yourself.
160         printf("Bottom of Task 1. time  = %d ms\n", xTaskGetTickCount()*10);
161         printf("Task 1 is using DelayUntil...\n");
162         vTaskDelayUntil( &xLastWakeTime, xPeriod );
163         printf("Task 1 is done waiting!\n");
164     }
165
166     vTaskDelete(NULL);  // should never reach here
167 }
```

Figure 4: Screenshot of code for Task 1

Lines 125-127 setup the variables to be used in `vTaskDelayUntil()`. The first variable, `xLastWakeTime`, holds the tick count returned by `xTaskGetTickCount()` while the variable `xPeriod` holds the number of ticks required to delay 5300 msec.

Once inside the main while loop, Task 1 prints that it is at the top of Task 1 and outputs the current millisecond based on the tick count. Next it turns on the LED corresponding to task 1. After the LED is on, it runs the function `BurnTime()`. This function sits in a `for` loop that writes the LED value for a corresponding task to the LEDs every time through the loop. This loop runs for about 1 second.

After burning time, Task 1 checks Task 3's handle to see if has anything attached to it. If it does not, Task 1 creates Task 3 with a priority of P2. This ensures that T3 will run after T2 but before T4 on the first time through the tasks.

Once T3 is created, T1 yields the rest of its tick to another task with the same priority using `taskYIELD()`. This satisfies the use of API function 11, `TaskYIELD()`. Since there are no other tasks at the same priority as T1, it just jumps back in. Next, T1 prints that it has reached the bottom of its loop with the time in msec based on the current tick count. Finally, T1 runs `vTaskDelayUntil()` and will run periodically with a period of 5300 msec. This satisfies the use of API function 5, `vTaskDelayUntil()`. This puts T1 in the blocked state. T2 has the next highest priority, so it takes over.

Below is a screenshot of the code for T2:

```
static void Task2( void *pvParameters )
{
    while(1)
    {
        // Print top of loop message and update LED
        printf("Top of Task 2. time  = %d ms\n", xTaskGetTickCount()*10);
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_2_LED);

        // Burn Time
        BurnTime("Task 2", TASK_2_LED);

        // Turn-off LED...
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED);

        // ... and suspend yourself
        printf("Bottom of Task 2. time  = %d ms\n", xTaskGetTickCount()*10);
        printf("Task 2 is suspending itself\n");
        vTaskSuspend(NULL);
    }
    vTaskDelete(NULL);  // should never reach here
}
```

Figure 5: Screenshot of code for Task 2

Task 2 is fairly simple. It prints that it is at the top of its `while` loop along with the time in msec based on the tick count. Next, it turns on its LED. Then it burns time just like T1. Once it is done burning time, it turns-off its LED, prints the time based on the current tick count, then suspends itself. This satisfies the use of API function 7, `vTaskSuspend()`. Since T2 is suspended, T3 runs, as it has the next highest priority and T1 is still blocked.

Below is a screenshot of the code for Task 3:

```c
static void Task3( void *pvParameters )
{
    // Create variables to hold T2 and T3 priorities.
    UBaseType_t uxTask2Priority, uxTask3Priority;
    while(1)
    {
        // Print top of loop message and update LED
        printf("Top of Task 3. time  = %d ms\n", xTaskGetTickCount()*10);
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_3_LED);

        // Burn Time
        BurnTime("Task 3", TASK_3_LED);

        // Get T2's priority...
        printf("Task 3 is checking Task 2's priority...\n");
        uxTask2Priority = uxTaskPriorityGet(xTask2);
        printf("Task 3 reports Task 2's priority is: %d\n", uxTask2Priority);

        // ... and raise T3's priority to one more than  T2's...
        uxTask3Priority = uxTask2Priority + 1;
        printf("Task 3 is raising its priority to %d\n", uxTask3Priority);
        vTaskPrioritySet(NULL, uxTask2Priority + 1);
        printf("Task 3 is checking its own priority...\n");
        uxTask3Priority = uxTaskPriorityGet(NULL);
        printf("Task 3's priority is: %d\n", uxTask3Priority);

        // ... so that it can resume T2 without it preempting T3...
        printf("Task 3 is resuming task 2\n");
        vTaskResume(xTask2);

        printf("Bottom of Task 3. time  = %d ms\n", xTaskGetTickCount()*10);

        // ... so T3 can kill itself in peace
        printf("Task 3 is killing itself!\n");
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED);   // T3 is not running so LED is off
        xTask3 = NULL;      // Detach task from handle...
        vTaskDelete(NULL);  // ... and delete T3.
    }
    vTaskDelete(NULL);  // should never reach here
}
```

Figure 6: Screenshot of code for Task 3

Task 3 starts by creating variables to hold the priority of T2 and T3. Once inside the main while loop, it prints that it is at the top of its main loop as well as the time based on the current tick count, then turns on its LED. Next, it burns time for about 1 second.

Once it is done burning time, it gets T2's priority using `uxTaskPriorityGet()`. This satisfies the use of API function 10, `uxTaskPriorityGet()`. T3 then raises its priority to one more than T2's priority using `vTaskPrioritySet()`. This satisfies the use of API function 9, `vTaskPrioritySet()`. It does this so it can resume T2 without T2 preempting T3 and taking over.

After raising its own priority, T3 resumes T2 using `vTaskResume()`. This satisfies the use of API function 8, `vTaskResume()`. Finally, T3 prints that it is at the bottom of its loop along with the time in msec based on the current tick count, turns-off its LED, disconnects itself from its handle, and deletes itself using `vTaskDelete()`. This satisfies the use of API function 2, `vTaskDelete()`.

On the first time through the tasks, T2 will run again. Once T2 suspends itself again, T4 will enter the running state since it has the next highest priority and T1 is still blocked.

Below is a screenshot of the code for Task 4:

```c
static void Task4( void *pvParameters )
{
    while(1)
    {
        // Print top of loop message and update LED
        printf("Top of Task 4. time  = %d ms\n", xTaskGetTickCount()*10);
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_4_LED);

        // Burn Time
        BurnTime("Task 4", TASK_4_LED);

        //Turn-off LED...
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED);

        // ... and Delay yourself
        printf("Bottom of Task 4. time  = %d ms\n", xTaskGetTickCount()*10);
        printf("Task 4 is delaying itself 3800 ms\n");
        vTaskDelay( pdMS_TO_TICKS( 3800 ) );
    }
    vTaskDelete(NULL);  // should never reach here
}
```

Figure 7: Screenshot of code for Task 4

Like T2, T4 is fairly simple. It starts by printing that it is at the top of the loop along with he current time in msec based on the current tick count, turns on its LED, then calls `BurnTime()` to kill about one second. Once the program returns from `BurnTime()`, T4 turns-off its LED, prints that it has reached the bottom of its task along with the time based on the current tick count, and delays itself for 3800 msec using `vTaskDelay()`. This satisfies the use of API function 4, `vTaskDelay()`.

At the end of each task, there is a call to `vTaskDelete()`. While these should never actually be reached, it is good practice to include it at the end of a task so that a task cannot just exit.

Also, there is space in this sequence for the Idle task to run. The Idle task takes care of cleaning deleted tasks from memory. Since `vTaskDelete()` was used and new tasks are created as the program runs, it is necessary for the Idle task to run.

<u>Results</u>
Below are photos of each LED being lit on the Zybo as a task is being run:

Figure 8: T1 LED

Figure 9: T2 LED

Figure 10: T3 LED

Figure 11: T4 LED

Figure 12: Idle Task LED

Although the specification says to have each task run for *at least 5 seconds*, this student misread it as *at most 5 seconds*, so each task only runs for one second. This made debugging and observing the repeating pattern easier.

The pattern laid-out in Figure 2 runs on the LEDs without trouble. After the initial pass through T1-T2-T3-T2-T4-IDLE, it successfully repeats the pattern T1-T3-T2-T4-IDLE, running each task, including the IDLE task, for about one second. The SDK Terminal output with messages from the first 30 seconds of the program running can be found in the Appendix. The described pattern and timing can be seen there. Although the timing is not exactly one second, it is usually within one tick, or 10 msec.

Conclusions

Overall, this lab successfully introduced the concept of task management in FreeRTOS. The main task management API function calls were used to create a periodic set of tasks, which generated a periodic pattern on the LEDs.

The biggest mistake was human error in reading the lab specification. However, this mistake could be fixed by multiplying by five all the times throughout the program. This would create a fairly boring pattern given the current tasks. A more interesting pattern could have been created but would have required completing rewriting of the lab code without gaining much experience with the API. The only concept not explored by this student was having two tasks running at the same priority. However, this concept was explored in Lab 2.

Appendix

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 printf xil_printf                                   // smaller, optimized printf

// GPIO definitions
#define LED_DEVICE_ID  XPAR_AXI_GPIO_0_DEVICE_ID            // GPIO device id for LEDs
#define LED_CHANNEL 1                                       // GPIO port for LEDs

XGpio SWInst, LEDInst;    // GPIO Device driver instance

// LED outputs
#define TASK_1_LED 0x01
#define TASK_2_LED 0x02
#define TASK_3_LED 0x04
#define TASK_4_LED 0x08
#define IDLE_LED 0x00

static char led_count;      // Number displayed on LEDs

// Function prototypes
static void Task1( void *pvParameters );                               // FreeRTOS Task
static void Task2( void *pvParameters );                               // FreeRTOS Task
static void Task3( void *pvParameters );                               // FreeRTOS Task
static void Task4( void *pvParameters );                               // FreeRTOS Task
static uint32_t BoardInit();                                           // Setup LEDs
static void BurnTime(char* task_name, unsigned char led);         // Kill processor time

// FreeRTOS Task Handles
static TaskHandle_t xTask1;
static TaskHandle_t xTask2;
static TaskHandle_t xTask3;
static TaskHandle_t xTask4;

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

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

	xTaskCreate( Task1,                             // Pointer to task function
				( const char * ) "T1",      // Descriptive task name.
				configMINIMAL_STACK_SIZE,       // Stack size of task.
```

```
                                        NULL,                   // Parameters to pass to task
                                        tskIDLE_PRIORITY+4,     // Priority level
                                        &xTask1 );                      // Handle for task


        xTaskCreate( Task2, ( const char * ) "T2",
                                        configMINIMAL_STACK_SIZE, NULL,
                                        tskIDLE_PRIORITY+3, &xTask2 );

        xTaskCreate( Task4, ( const char * ) "T4",
                                        configMINIMAL_STACK_SIZE, NULL,
                                                tskIDLE_PRIORITY+1, &xTask4 );

        // Starts tasks
        vTaskStartScheduler();

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

/*
 * Board Initialization function.
 *      Setups the LEDs
 */
static uint32_t BoardInit()
{
        // Create variable to hold Status
        uint32_t Status;

        // Initialize LEDs
        led_count = 0x00;

        // 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;
}

/*
 * Function "burn time." Each task is supposed to run long enough
 *      so that the LED can be read by a human. This function will
 *      run for about 1 second, assuming it is not preempted.
 *
 *      INPUTS - task_name: name of task burning time
 *                      led: value to write to LEDs
 *      OUTPUTS - none
 */
static void BurnTime(char* task_name, unsigned char led)
{
        TickType_t xStartTick;
        TickType_t xEndTick;
```

```
        uint32_t millis;
        xStartTick = xTaskGetTickCount();
        printf("%s burning time...\n", task_name);
        for(uint32_t n = 0; n< 3955500; n++)
        {
                XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, led);
        }
        xEndTick = xTaskGetTickCount();
        millis = (xEndTick - xStartTick)*10;
        printf("%s is done burning time.\n", task_name);
        printf("%s killed %d ms \n", task_name, millis);
}

/*---------------------------FreeRTOS Tasks------------------------------*/
static void Task1( void *pvParameters )
{
        // Setup for delay until
        TickType_t xLastWakeTime;
        xLastWakeTime = xTaskGetTickCount();
        const TickType_t xPeriod = pdMS_TO_TICKS( 5300 );

        while(1)
        {
                // Print top of loop message and update LED
                printf("Top of Task 1. time  = %d ms\n", xTaskGetTickCount()*10);
                XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_1_LED);

                // Burn Time
                BurnTime("Task 1", TASK_1_LED);

                // Check to see if task 3 exists. Create it if it doesn't
                if (xTask3 == NULL)
                {
                        printf("Task 3 does not exist!\n");
                        printf("Creating Task 3...\n");
                        xTaskCreate( Task3,     ( const char * ) "T3",
                                                configMINIMAL_STACK_SIZE, NULL,
                                                tskIDLE_PRIORITY+2, &xTask3 );
                        printf("Task 3 created...\n");
                }
                else printf("Task 3 already exists!");

                // Use taskYIELD bc it's required
                XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED)// T1 is not running
                printf("Task 1 is Yielding\n");
                taskYIELD();     // will come right back bc no other task shares T1's priority
                printf("Task 1 done Yielding\n");
                XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_1_LED);// running so LED
should be lit

                // Turn-off LED...
                XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED);
                // ... and delay yourself.
                printf("Bottom of Task 1. time  = %d ms\n", xTaskGetTickCount()*10);
                printf("Task 1 is using DelayUntil...\n");
```

```
        vTaskDelayUntil( &xLastWakeTime, xPeriod );
        printf("Task 1 is done waiting!\n");
    }

    vTaskDelete(NULL);      // should never reach here
}

static void Task2( void *pvParameters )
{
    while(1)
    {
        // Print top of loop message and update LED
        printf("Top of Task 2. time  = %d ms\n", xTaskGetTickCount()*10);
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_2_LED);

        // Burn Time
        BurnTime("Task 2", TASK_2_LED);

        // Turn-off LED...
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED);

        // ... and suspend yourself
        printf("Bottom of Task 2. time  = %d ms\n", xTaskGetTickCount()*10);
        printf("Task 2 is suspending itself\n");
        vTaskSuspend(NULL);
    }
    vTaskDelete(NULL);      // should never reach here
}

static void Task3( void *pvParameters )
{
    // Create variables to hold T2 and T3 priorities.
    UBaseType_t uxTask2Priority, uxTask3Priority;
    while(1)
    {
        // Print top of loop message and update LED
        printf("Top of Task 3. time  = %d ms\n", xTaskGetTickCount()*10);
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_3_LED);

        // Burn Time
        BurnTime("Task 3", TASK_3_LED);

        // Get T2's priority...
        printf("Task 3 is checking Task 2's priority...\n");
        uxTask2Priority = uxTaskPriorityGet(xTask2);
        printf("Task 3 reports Task 2's priority is: %d\n", uxTask2Priority);

        // ... and raise T3's priority to one more than  T2's...
        uxTask3Priority = uxTask2Priority + 1;
        printf("Task 3 is raising its priority to %d\n", uxTask3Priority);
        vTaskPrioritySet(NULL, uxTask2Priority + 1);
        printf("Task 3 is checking its own priority...\n");
        uxTask3Priority = uxTaskPriorityGet(NULL);
        printf("Task 3's priority is: %d\n", uxTask3Priority);
```

```
        // ... so that it can resume T2 without it preempting T3...
        printf("Task 3 is resuming task 2\n");
        vTaskResume(xTask2);

        printf("Bottom of Task 3. time  = %d ms\n", xTaskGetTickCount()*10);

        // ... so T3 can kill itself in peace
        printf("Task 3 is killing itself!\n");
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED);// T3 is not running so
LED is off
        xTask3 = NULL;          // Detach task from handle...
        vTaskDelete(NULL);      // ... and delete T3.
    }
    vTaskDelete(NULL);      // should never reach here
}

static void Task4( void *pvParameters )
{
    while(1)
    {
        // Print top of loop message and update LED
        printf("Top of Task 4. time  = %d ms\n", xTaskGetTickCount()*10);
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, TASK_4_LED);

        // Burn Time
        BurnTime("Task 4", TASK_4_LED);

        //Turn-off LED...
        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, IDLE_LED);

        // ... and Delay yourself
        printf("Bottom of Task 4. time  = %d ms\n", xTaskGetTickCount()*10);
        printf("Task 4 is delaying itself 3800 ms\n");
        vTaskDelay( pdMS_TO_TICKS( 3800 ) );
    }
    vTaskDelete(NULL);      // should never reach here
}
```

SDK Terminal Output

```
Top of Task 1. time  = 0 ms
Task 1 burning time...
Task 1 is done burning time.
Task 1 killed 980 ms
Task 3 does not exist!
Creating Task 3...
Task 3 created...
Task 1 is Yielding
Task 1 done Yielding
Bottom of Task 1. time  = 990 ms
Task 1 is using DelayUntil...
Top of Task 2. time  = 1000 ms
```

```
Task 2 burning time...
Task 2 is done burning time.
Task 2 killed 990 ms
Bottom of Task 2. time  = 1990 ms
Task 2 is suspending itself
Top of Task 3. time  = 2000 ms
Task 3 burning time...
Task 3 is done burning time.
Task 3 killed 990 ms
Task 3 is checking Task 2's priority...
Task 3 reports Task 2's priority is: 3
Task 3 is raising its priority to 4
Task 3 is checking its own priority...
Task 3's priority is: 4
Task 3 is resuming task 2
Bottom of Task 3. time  = 3010 ms
Task 3 is killing itself!
Top of Task 2. time  = 3010 ms
Task 2 burning time...
Task 2 is done burning time.
Task 2 killed 990 ms
Bottom of Task 2. time  = 4010 ms
Task 2 is suspending itself
Top of Task 4. time  = 4010 ms
Task 4 burning time...
Task 4 is done burning time.
Task 4 killed 1000 ms
Bottom of Task 4. time  = 5010 ms
Task 4 is delaying itself 3800 ms
Task 1 is done waiting!
Top of Task 1. time  = 5300 ms
Task 1 burning time...
Task 1 is done burning time.
Task 1 killed 990 ms
Task 3 does not exist!
Creating Task 3...
Task 3 created...
Task 1 is Yielding
Task 1 done Yielding
Bottom of Task 1. time  = 6290 ms
Task 1 is using DelayUntil...
Top of Task 3. time  = 6300 ms
Task 3 burning time...
Task 3 is done burning time.
Task 3 killed 990 ms
Task 3 is checking Task 2's priority...
Task 3 reports Task 2's priority is: 3
```

```
Task 3 is raising its priority to 4
Task 3 is checking its own priority...
Task 3's priority is: 4
Task 3 is resuming task 2
Bottom of Task 3. time  = 7310 ms
Task 3 is killing itself!
Top of Task 2. time  = 7310 ms
Task 2 burning time...
Task 2 is done burning time.
Task 2 killed 990 ms
Bottom of Task 2. time  = 8310 ms
Task 2 is suspending itself
Top of Task 4. time  = 8810 ms
Task 4 burning time...
Task 4 is done burning time.
Task 4 killed 980 ms
Bottom of Task 4. time  = 9790 ms
Task 4 is delaying itself 3800 ms
Task 1 is done waiting!
Top of Task 1. time  = 10600 ms
Task 1 burning time...
Task 1 is done burning time.
Task 1 killed 990 ms
Task 3 does not exist!
Creating Task 3...
Task 3 created...
Task 1 is Yielding
Task 1 done Yielding
Bottom of Task 1. time  = 11590 ms
Task 1 is using DelayUntil...
Top of Task 3. time  = 11600 ms
Task 3 burning time...
Task 3 is done burning time.
Task 3 killed 990 ms
Task 3 is checking Task 2's priority...
Task 3 reports Task 2's priority is: 3
Task 3 is raising its priority to 4
Task 3 is checking its own priority...
Task 3's priority is: 4
Task 3 is resuming task 2
Bottom of Task 3. time  = 12610 ms
Task 3 is killing itself!
Top of Task 2. time  = 12620 ms
Task 2 burning time...
Task 2 is done burning time.
Task 2 killed 990 ms
Bottom of Task 2. time  = 13610 ms
```

```
Task 2 is suspending itself
Top of Task 4. time  = 13610 ms
Task 4 burning time...
Task 4 is done burning time.
Task 4 killed 990 ms
Bottom of Task 4. time  = 14610 ms
Task 4 is delaying itself 3800 ms
Task 1 is done waiting!
Top of Task 1. time  = 15900 ms
Task 1 burning time...
Task 1 is done burning time.
Task 1 killed 990 ms
Task 3 does not exist!
Creating Task 3...
Task 3 created...
Task 1 is Yielding
Task 1 done Yielding
Bottom of Task 1. time  = 16890 ms
Task 1 is using DelayUntil...
Top of Task 3. time  = 16900 ms
Task 3 burning time...
Task 3 is done burning time.
Task 3 killed 990 ms
Task 3 is checking Task 2's priority...
Task 3 reports Task 2's priority is: 3
Task 3 is raising its priority to 4
Task 3 is checking its own priority...
Task 3's priority is: 4
Task 3 is resuming task 2
Bottom of Task 3. time  = 17910 ms
Task 3 is killing itself!
Top of Task 2. time  = 17910 ms
Task 2 burning time...
Task 2 is done burning time.
Task 2 killed 990 ms
Bottom of Task 2. time  = 18910 ms
Task 2 is suspending itself
Top of Task 4. time  = 18910 ms
Task 4 burning time...
Task 4 is done burning time.
Task 4 killed 990 ms
Bottom of Task 4. time  = 19910 ms
Task 4 is delaying itself 3800 ms
Task 1 is done waiting!
Top of Task 1. time  = 21200 ms
Task 1 burning time...
Task 1 is done burning time.
```

```
Task 1 killed 990 ms
Task 3 does not exist!
Creating Task 3...
Task 3 created...
Task 1 is Yielding
Task 1 done Yielding
Bottom of Task 1. time  = 22190 ms
Task 1 is using DelayUntil...
Top of Task 3. time  = 22200 ms
Task 3 burning time...
Task 3 is done burning time.
Task 3 killed 990 ms
Task 3 is checking Task 2's priority...
Task 3 reports Task 2's priority is: 3
Task 3 is raising its priority to 4
Task 3 is checking its own priority...
Task 3's priority is: 4
Task 3 is resuming task 2
Bottom of Task 3. time  = 23210 ms
Task 3 is killing itself!
Top of Task 2. time  = 23220 ms
Task 2 burning time...
Task 2 is done burning time.
Task 2 killed 990 ms
Bottom of Task 2. time  = 24210 ms
Task 2 is suspending itself
Top of Task 4. time  = 24210 ms
Task 4 burning time...
Task 4 is done burning time.
Task 4 killed 990 ms
Bottom of Task 4. time  = 25210 ms
Task 4 is delaying itself 3800 ms
Task 1 is done waiting!
Top of Task 1. time  = 26500 ms
Task 1 burning time...
Task 1 is done burning time.
Task 1 killed 990 ms
Task 3 does not exist!
Creating Task 3...
Task 3 created...
Task 1 is Yielding
Task 1 done Yielding
Bottom of Task 1. time  = 27490 ms
Task 1 is using DelayUntil...
Top of Task 3. time  = 27500 ms
Task 3 burning time...
Task 3 is done burning time.
```

```
Task 3 killed 990 ms
Task 3 is checking Task 2's priority...
Task 3 reports Task 2's priority is: 3
Task 3 is raising its priority to 4
Task 3 is checking its own priority...
Task 3's priority is: 4
Task 3 is resuming task 2
Bottom of Task 3. time  = 28510 ms
Task 3 is killing itself!
Top of Task 2. time  = 28520 ms
Task 2 burning time...
Task 2 is done burning time.
Task 2 killed 990 ms
Bottom of Task 2. time  = 29510 ms
Task 2 is suspending itself
Top of Task 4. time  = 29510 ms
Task 4 burning time...
Task 4 is done burning time.
Task 4 killed 990 ms
Bottom of Task 4. time  = 30510 ms
Task 4 is delaying itself 3800 ms
Task 1 is done waiting!
Top of Task 1. time  = 31800 ms
Task 1 burning time...
Task 1 is done burning time.
Task 1 killed 990 ms
Task 3 does not exist!
Creating Task 3...
Task 3 created...
Task 1 is Yielding
Task 1 done Yielding
Bottom of Task 1. time  = 32790 ms
Task 1 is using DelayUntil...
Top of Task 3. time  = 32800 ms
Task 3 burning time...
```