Lab 08 Report: FreeRTOS Software Timer

Peter Mowen
peter.mowen@temple.edu

Summary
This lab explores software timers in FreeRTOS. A software timer will be used to toggle the LEDs. The buttons will be used to restart the timer, change the timer period, stop the timer, and start the timer.

Introduction
FreeRTOS provides software timers as an optional functionality. To be able to use software timers, one has to include the FreeRTOS "timers.h" file. This file gives the user access to the FreeRTOS software timer API.

There are two main types of software timers in FreeRTOS: one-shot and auto-reload. A one-shot timer runs once and then stops until it is started again. An auto-reload timer automatically restarts once it has expired. When a timer expires, its associated timer callback function runs. These should short functions that do not contain any statements that might but the callback function into a blocked state.

Timer commands are processed using a command queue. When a timer API function is called, it generates a command that is placed on a queue. A FreeRTOS daemon then processes the commands. In this lab, the daemon will be able to remove commands from the queue and process them immediately.

Timers can be in one of two states: running or dormant. In the dormant state, the timer is not running, and the callback function is never called. In the running state, the timer is running, and the callback function gets called when a timer expires. If a timer is a one-shot timer, then it is put into the dormant state at the end of the first time its callback function finishes. If a timer is an auto-reload timer, then when its callback function finishes, the timer is restarted and stays in the running state.

In this lab, one software timer will be created with an initial period of 5 seconds. When the timer expires, it is to toggle the LEDs. The LEDs start with the pattern 1001. When the callback function runs the first time, it should cause the LEDs to change to 0110. The next time the timer callback function runs, the LEDs should toggle to 1001. This pattern continues unless it is changed by the buttons.

The buttons will be used in this lab in the following manner:

1. When button 0 is pressed, the timer is restarted
2. When button 1 is pressed, the timer's period is set to 2.5 seconds
3. When button 2 is pressed, the timer stops, and the LEDs are set to 0000.
4. When button 3 is pressed, the timer's period is set to 5 seconds, and the LEDs are set to 1001.

Discussion

Hardware Setup in Vivado HLx

Below is a screenshot of the block diagram from Vivado HLx:

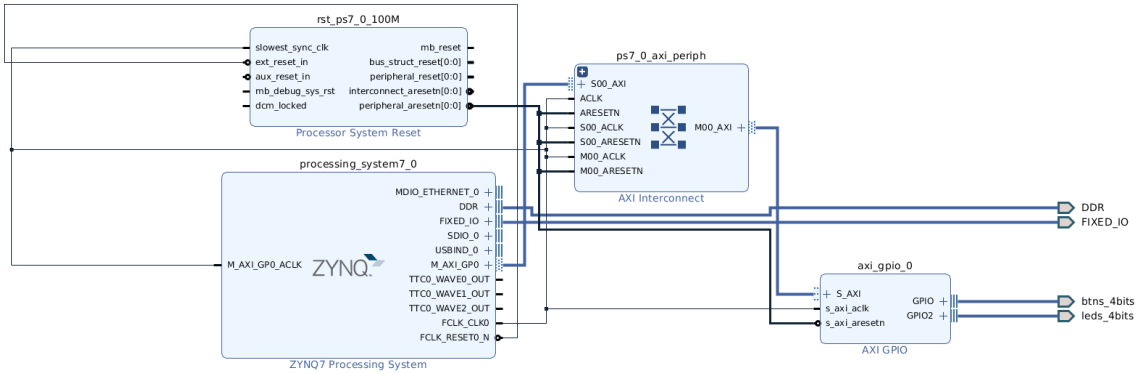

Figure 1: Vivado HLx block diagram

Note that one GPIO block is used for both the buttons and the LEDs. The buttons are on channel 1 and the LEDs are on channel 2.

Include Files

Below is a screenshot of the FreeRTOS-related include statements in this program:

```
1  /* FreeRTOS includes. */
2  #include "FreeRTOS.h"
3  #include "task.h"
4  #include "timers.h"
```

Figure 2: FreeRTOS-related include statements

To gain access to FreeRTOS timers, the “timers.h” file was included in this program. This file includes all of the FreeRTOS data types and API functions needed to create and control software timers including the `TimerHandle_t` data type, `xTimerCreate()`, `xTimerChangePeriod()`, `xTimerStart()`, `xTimerStop()`, etc.

The following files were included for Xilinx support:

```
6  /* Xilinx includes. */
7  #include "xparameters.h"
8  #include "xgpio.h"
9  #include "xstatus.h"
10 #include "xil_printf.h"
11
12 #define printf xil_printf
```

Figure 3: Xilinx include files

These files have been discussed in detail in previous lab reports and will not be covered again here.

Global Variables, Function Prototypes, and Preprocessor Definitions

The buttons and LEDs were setup on the same GPIO IP block in Vivado HLx as can be seen in Figure 1. The following object and preprocessor definitions were created to access them:

```
14 // GPIO object and constants
15 XGpio GpioInst;                              // GPIO Device driver instance
16 #define GPIO_ID   XPAR_AXI_GPIO_0_DEVICE_ID // GPIO device id for Switches
17 #define BTN_CHANNEL 1                       // GPIO port for Switches
18 #define LEDS_CHANNEL 2                      // GPIO port for LEDs
```

Figure 4: GPIO instance and constants

One `XGpio` device driver was instantiated to be used for both the buttons and the LEDs and channel numbers were defined each.

The following screenshot shows the prototype functions used in this project as well as the handles for the FreeRTOS tasks and timer.

```
20 // Helper function declarations
21 static BaseType_t BoardInit();
22
23 // FreeRTOS function declarations
24 static void vTaskReadButtons( void *pvParameters );
25 static void vTaskToggleLEDs( void *pvParameters );
26 static void pvTimerCallback( TimerHandle_t xTimer );
27
28 // FreeRTOS Task Handles
29 static TaskHandle_t xReadButtonsTask;
30 static TaskHandle_t xToggleLEDsTask;
31
32 // FreeRTOS Timer Handle
33 static TimerHandle_t xTimer;
```

Figure 5: Function prototypes and FreeRTOS handles

The `BoardInit()` function will be used to initialize the GPIO devices. Two FreeRTOS tasks are declared and one FreeRTOS timer callback function is declared. One task is for reading the buttons and the other is for toggling the LEDs. The timer callback function will be used to tell the toggle LEDs function whether to toggle the LEDs. Handles were created for the tasks and timer callback function.

Finally, here is a screenshot of preprocessor definitions and global variables used throughout the lab:

```
35 // Timer, button, and LED constants
36 #define DEFAULT_TIMER_PERIOD pdMS_TO_TICKS( 5000 )
37 #define BTN0 0b0001
38 #define BTN1 0b0010
39 #define BTN2 0b0100
40 #define BTN3 0b1000
41 #define DEFAULT_LED_STATE 0b1001
42 UBaseType_t toggleLEDs = 0;
```

Figure 6: Preprocessor definitions and global variables

The default timer period is set to 5 seconds. Constants representing the binary value that each button takes are defined. A default LED state is defined. Finally, a variable that will be used to control whether the LEDs toggle is defined and set to zero.

Board Initialization, Task and Timer Creation

The following is a screenshot shows the function which initializes the buttons and LEDs:

```
113 /*-----------------------------------------------------------*/
114 static BaseType_t BoardInit()
115 {
116     // Create variable to hold Status
117     BaseType_t xStatus;
118
119     // Initialize GPIO and check status
120     xStatus = XGpio_Initialize(&GpioInst, GPIO_ID);
121     if (xStatus != XST_SUCCESS) { return XST_FAILURE; }
122
123     // Set the direction for the switches to input
124     XGpio_SetDataDirection(&GpioInst, BTN_CHANNEL, 0xF);
125
126     // Set the direction for the LEDs to output
127     XGpio_SetDataDirection(&GpioInst, LEDS_CHANNEL, 0x0);
128
129     return XST_SUCCESS;
130 }
```

Figure 7: Board initialization function

This function is called from main and returns a status. Line 120 initializes the GPIO instance and stores the result. Line 121 checks the result and returns if it is not successful. Assuming the GPIO instance is successfully initialized, line 124 sets the data direction of the buttons to inputs and line 127 sets the data direction of the LEDs to outputs.

The following screenshot shows the task creation:

```
60      printf("Creating tasks...");
61      xTaskCreate( vTaskReadButtons,          // Pointer to task function
62                   ( const char * ) "BTNS",   // Descriptive task name. Used for debugging
63                   configMINIMAL_STACK_SIZE,  // Stack size of task.
64                   NULL,                      // Parameters to pass to task
65                   tskIDLE_PRIORITY+1,        // Priority level
66                   &xReadButtonsTask );       // Handle for task
67
68      xTaskCreate( vTaskToggleLEDs,           // Pointer to task function
69                   ( const char * ) "LEDS",   // Descriptive task name. Used for debugging
70                   configMINIMAL_STACK_SIZE,  // Stack size of task.
71                   NULL,                      // Parameters to pass to task
72                   tskIDLE_PRIORITY+1,        // Priority level
73                   &xToggleLEDsTask );        // Handle for task
74      printf("Tasks created!\n");
```

Figure 8: Task creation code

Note that the tasks are created with the same priority. This means the will swapping back and forth every tick period.

The following screenshot shows the timer creation and verification:

```
76      printf("Creating timer...");
77      xTimer = xTimerCreate( "Timer",                 // Timer name used for debugging
78                             DEFAULT_TIMER_PERIOD,    // Timer period
79                             pdTRUE,                  // Autoreload timer?
80                             0,                       // Timer ID
81                             pvTimerCallback);        // Handle to callback function
82
83      // Check that the timer was created properly
84      if (xTimer != NULL)
85      {
86          printf("Timer created!\n");
87
88          printf("Starting timer...");
89          xTimerStarted = xTimerStart(xTimer, 0);
90          if (xTimerStarted == pdPASS)
91          {
92              printf("Timer started!\n");
93
94              printf("Starting task scheduler...\n");
95              // Starts tasks
96              vTaskStartScheduler();
97          }
98          else
99          {
100             printf("timer failed to start!\n");
101         }
102     }
103     else
104     {
105         printf("Failed to create timer!\n");
106     }
```

Figure 9: Timer creation and verification code

The first argument in the `xTimerCreate()` API function is a name given to the timer that can be used for debugging. The second argument specifies the timer period; here it is set to the default period, which was defined as 5 seconds. The third argument tells the designates whether the timer should auto-reload where `pdTRUE` indicates that it should. The fourth argument is a timer ID variable. It can be used when multiple timers call the

same callback function to identify which timer called the callback function. The final argument is the handle to the timer's callback function.

Line 84 checks to see if the timer handle is not NULL. If it is null, then the timer was not created successfully. If it is not NULL, then the timer is started and the status of is stored in a variable. If this status indicates the timer started successfully, then the task scheduler is called, and the tasks start.

Timer Callback Function

The following is a screenshot of the timer callback function:

```
186 /*-----------------------------------------------------------*/
187 static void pvTimerCallback( TimerHandle_t xTimer )
188 {
189     //printf("Timer Callback!\n");
190     toggleLEDs = 1;
191 }
```

Figure 10: Timer callback function

The only thing this callback function does is set the toggleLEDs variable to one. This will cause the toggle LED task to toggle the LEDs. Since timer callback functions are like interrupts, they should be short and have not chance to enter the blocked state. This will happen ever time the timer's callback function is called.

Tasks

Below is a screenshot of the task which reads the buttons:

```
/*-----------------------------------------------------------*/
static void vTaskReadButtons( void *pvParameters )
{
    uint8_t buttons;
    while(1)
    {
        buttons = XGpio_DiscreteRead(&GpioInst, BTN_CHANNEL) & 0x0F;
        switch (buttons)
        {
            case BTN0:
                // Reset timer while button 0 is pressed.
                //  While button is held, leds will not toggle bc timer is continuously reset
                xTimerReset(xTimer, 0);
                break;
            case BTN1:
                // Change period to half of default period while button 1 is pressed
                //  While button is held, leds will not toggle bc period is continuously reset
                xTimerChangePeriod(xTimer, DEFAULT_TIMER_PERIOD/2, 0 );
                break;
            case BTN2:
                // Stop time and turn-off LEDs if button 2 is pressed
                xTimerStop(xTimer, 0);
                XGpio_DiscreteWrite(&GpioInst, LEDS_CHANNEL, 0x00);
                break;
            case BTN3:
                // Change the leds and the timer period back to default
                //  Timer won't really start until button is released
                xTimerChangePeriod(xTimer, DEFAULT_TIMER_PERIOD, 0 );
                XGpio_DiscreteWrite(&GpioInst, LEDS_CHANNEL, DEFAULT_LED_STATE);
                break;
            default:
                break;
        }
    }
}
```

Figure 11: Read buttons task code

This task reads the four least significant bits of the buttons, the decides what to do based on which buttons are pressed. Note that if more than one button is pressed, one of these actions will be taken because there is not delay before reading the buttons. Since it is not possible to press or release the buttons at the exact same time, one of them will always be read.

If button zero is pressed, then the `xTimerReset()` function is called. This function restarts the timer. So, if a timer is about to expire and this function is called, the timer is reset. The effect is that the callback function will not be called until the new time expires. If a timer is in the dormant state, this will put it back into the running state. The first argument is the handle for the timer. The second argument is the number of ticks to wait to put the timer command on the timer command queue. It is set to zero because the timer command queue should always be empty in this project. Each time an item is put on the queue, the timer daemon will process it immediately.

If button 1 is pressed, then the `xTimerChangePeriod()` function is called. This function changes the time it takes for a timer to expire. This will change the frequency of the callback function. Again, the first argument is the timer handle. The second argument is the new period. Since the default period is 5 seconds, dividing it by 2 will produce a period of 2.5 seconds. Again, the final argument is the number of ticks to wait to put the timer command on the timer command queue.

If button 2 is pressed, then 0x00 is written to the LEDs and the `xTimerStop()` function is called. This function stops a timer and puts it into the dormant state. The first argument is the timer handle and the second argument is the ticks to wait to put the timer command on the queue.

If button 3 is pressed, then the default LED state is written to the LEDs and the timer period is changed back to the default of 5 seconds.

Below is a screenshot of the task which toggles the LEDs:

```
168 /*-----------------------------------------------------------*/
169 static void vTaskToggleLEDs( void *pvParameters )
170 {
171     uint8_t leds;
172     leds = DEFAULT_LED_STATE;
173     XGpio_DiscreteWrite(&GpioInst, LEDS_CHANNEL, leds);
174
175     while(1)
176     {
177         if (toggleLEDs == 1)
178         {
179             leds = ~leds;
180             XGpio_DiscreteWrite(&GpioInst, LEDS_CHANNEL, leds);
181             toggleLEDs = 0;
182         }
183     }
184 }
```

Figure 12: Toggle LED task code

The first time this task is run, it writes the default LED state to the LEDs. From then on, it checks the `toggleLEDs` variable and toggles the LEDs if it is set to 1. The only other task that writes to the `toggleLEDs` variable is the timer callback function. Since the toggle leds task only writes to the `toggleLEDs` variable after the timer expires, there is no chance that both the callback function and this task will try to write to the `toggleLEDs` variable at the same time, so not precautions were taken.

Results

The video at the following link demonstrates this lab:

https://youtu.be/GIyBKwXb-Rs

Conclusions

This lab successfully introduced software timers in FreeRTOS. A timer was created, and its callback function was used to toggle the Zybo LEDs. Furthermore, various timer related API function calls were made to change the period of the timer, restart the timer, and stop the timer. Finally, two physical objects, the buttons and the LEDs, were controlled using one GPIO object.

Appendix

C Code

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

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

#define printf xil_printf

// GPIO object and constants
XGpio GpioInst;                                 // GPIO Device driver instance
#define GPIO_ID   XPAR_AXI_GPIO_0_DEVICE_ID     // GPIO device id for Switches
#define BTN_CHANNEL 1                           // GPIO port for Switches
#define LEDS_CHANNEL 2                          // GPIO port for LEDs

// Helper function declarations
static BaseType_t BoardInit();

// FreeRTOS function declarations
static void vTaskReadButtons( void *pvParameters );
static void vTaskToggleLEDs( void *pvParameters );
static void pvTimerCallback( TimerHandle_t xTimer );

// FreeRTOS Task Handles
static TaskHandle_t xReadButtonsTask;
static TaskHandle_t xToggleLEDsTask;

// FreeRTOS Timer Handle
static TimerHandle_t xTimer;

// Timer, button, and LED constants
#define DEFAULT_TIMER_PERIOD pdMS_TO_TICKS( 5000 )
#define BTN0 0b0001
#define BTN1 0b0010
#define BTN2 0b0100
#define BTN3 0b1000
#define DEFAULT_LED_STATE 0b1001
UBaseType_t toggleLEDs = 0;

/*-----------------------------------------------------------*/
int main( void )
{
        // Create variable to hold Status
        BaseType_t xStatus, xTimerStarted;
        printf("Initializing board...");
        xStatus = BoardInit();

        if (xStatus != XST_SUCCESS)
```

```
{
	printf("Failed to initialize board!\n");
	return XST_FAILURE;
}

printf("Board initialized!\n");

printf("Creating tasks...");
xTaskCreate( vTaskReadButtons,		// Pointer to task function
		( const char * ) "BTNS",	// Descriptive task name. Used for debugging
		configMINIMAL_STACK_SIZE,// Stack size of task.
		 NULL,			// Parameters to pass to task
		 tskIDLE_PRIORITY+1,	// Priority level
		 &xReadButtonsTask );	// Handle for task

xTaskCreate( vTaskToggleLEDs,		// Pointer to task function
		( const char * ) "LEDS",	// Descriptive task name. Used for debugging
		configMINIMAL_STACK_SIZE,// Stack size of task.
		NULL,			// Parameters to pass to task
		tskIDLE_PRIORITY+1,	// Priority level
		&xToggleLEDsTask );	// Handle for task
printf("Tasks created!\n");

printf("Creating timer...");
xTimer = xTimerCreate( "Timer",			// Timer name used for debugging
			DEFAULT_TIMER_PERIOD,// Timer period
			pdTRUE,			// Autoreload timer?
			0,			// Timer ID
			pvTimerCallback);	// Handle to callback function

// Check that the timer was created properly
if (xTimer != NULL)
{
	printf("Timer created!\n");

	printf("Starting timer...");
	xTimerStarted = xTimerStart(xTimer, 0);
	if (xTimerStarted == pdPASS)
	{
		printf("Timer started!\n");

		printf("Starting task scheduler...\n");
		// Starts tasks
		vTaskStartScheduler();
	}
	else
	{
		printf("timer failed to start!\n");
	}
}
else
{
	printf("Failed to create timer!\n");
}
```

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

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

	// Initialize GPIO and check status
	xStatus = XGpio_Initialize(&GpioInst, GPIO_ID);
	if (xStatus != XST_SUCCESS) { return XST_FAILURE; }

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

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

	return XST_SUCCESS;
}

/*-----------------------------------------------------------*/
static void vTaskReadButtons( void *pvParameters )
{
	uint8_t buttons;
	while(1)
	{
		buttons = XGpio_DiscreteRead(&GpioInst, BTN_CHANNEL) & 0x0F;
		switch (buttons)
		{
			case BTN0:
				// Reset timer while button 0 is pressed.
				//	While button is held, leds will not toggle bc timer is
continuously reset
				xTimerReset(xTimer, 0);
				break;
			case BTN1:
				// Change period to half of default period while button 1 is pressed
				//	While button is held, leds will not toggle bc period is
continuously reset
				xTimerChangePeriod(xTimer, DEFAULT_TIMER_PERIOD/2, 0 );
				break;
			case BTN2:
				// Stop time and turn-off LEDs if button 2 is pressed
				xTimerStop(xTimer, 0);
				XGpio_DiscreteWrite(&GpioInst, LEDS_CHANNEL, 0x00);
				break;
			case BTN3:
				// Change the leds and the timer period back to default
				//	Timer won't really start until button is released
				xTimerChangePeriod(xTimer, DEFAULT_TIMER_PERIOD, 0 );
```

```
                                        XGpio_DiscreteWrite(&GpioInst, LEDS_CHANNEL,
DEFAULT_LED_STATE);
                                        break;
                                default:
                                        break;
                        }
                }
}

/*-----------------------------------------------------------*/
static void vTaskToggleLEDs( void *pvParameters )
{
        uint8_t leds;
        leds = DEFAULT_LED_STATE;
        XGpio_DiscreteWrite(&GpioInst, LEDS_CHANNEL, leds);

        while(1)
        {
                if (toggleLEDs == 1)
                {
                        leds = ~leds;
                        XGpio_DiscreteWrite(&GpioInst, LEDS_CHANNEL, leds);
                        toggleLEDs = 0;
                }
        }
}

/*-----------------------------------------------------------*/
static void pvTimerCallback( TimerHandle_t xTimer )
{
        //printf("Timer Callback!\n");
        toggleLEDs = 1;
}
```