Lab 2 – Task Management in FreeRTOS

Leomar Durán
https://github.com/lduran2

Summary

This project get the student acquainted with FreeRTOS which has facilities for controlling tasks. The lab works with the FreeRTOS environment on the Zybo board, and builds on both exercise 2B and Lab 1, taking the approache of splitting up its objective into 3 tasks, which involve output to the LED and control of tasks by the buttons and switches respectively.

The motivation for this analysis is multitasking which is an expected feature of all computers. It is a basic function offered by most modern operating systems, and a must have for any new operating systems being designed.

The project was completed in 5 iterations, which built on the original task design presented in the FreeRTOS hello world project and adding each of the specified features. Assumptions were made as to delay constants and some combinatorial behavior not discussed in the specification. Some compromises were made to keep the program as close as possible to the specification as possible without complicating it. However, by using button combinations, the user will get as close to full behavior as can be had while keeping the design simple.

Introduction

The objective in this project is that the student will be able to manage tasks using the FreeRTOS environment. FreeRTOS is a real-time operating system, which is capable of multitasking.

This lab builds on the design of exercise 2B before interrupts by adding a second channel to the GPIO used for input.

It also builds on the first lab by splitting the polling and displaying loop into 3 separate, albeit somewhat different, tasks.

The first task keeps a counter which is displayed in the LEDs.

The second task is controlled by the buttons. If buttons 1 and 0 are pressed together, the LED counter will pause. If button 2 is pressed, the LED counter will continue without regards as to the state of buttons 1 and 2.

The third task is controlled by the switches. If switch 3 is thrown, it will also pause the LED counter, and if it is retracted, the LED counter will continue.

Additionally, the two inputs may suspend each other. If button 3 is pressed, it suspends the operation of the switches, and once it's released, they will resume function. Also, if

switches 1 and 0 are thrown together, it suspends the operation of the buttons, but once they are both retracted, the buttons will operate normally.

The buttons require debounce.

The assumptions made were a $250\,\mathrm{ms}$ delay for debounce, and a $500\,\mathrm{ms}$ delay for visualization of the LEDs. It was assumed that the switches require no debounce.

In order for the effect of the buttons on the LEDs to be noticeable, rather than assuming the LEDs resume whenever switch 3 is in its retracted state, it is assumed that the LEDs resume whenever switch 3 is retracted from its thrown state. All other logic for the switches has their state checked continuously.

Although the customary behavior of a switch is an effect from when it's thrown until it is retracted in either direction unless it is cancelled out by another switch or button, this affects the retracted state. So I believe it is justified. Alternatively, the desired effect can be reached by suspending the switches with button 3, but this would leave it so the buttons always have to suspend the switches to reach any noticeable effect.

Beyond this, the attempt is made to keep the logic as simple as possible with the tasks having limited knowledge of each other's internal state.

As a result, after throwing switch 3 to pause the LEDs, pressing button 2 does not appear to have an effect because the LED counter is suspended as soon as it is resumed, and the debounce stops another attempt to resume until the stage of the buttons changes.

It is also assumed that any combination of switches or buttons is valid.

This project can alternatively be performed using polling in a single function or by using interrupts.

Discussion

Hardware design

This experiment builds on the design 2B that has a Zybo board and 2 GPIOs, one for buttons and one for the LEDs. However, it adds a second channel to the former GPIO to include switch input as well.

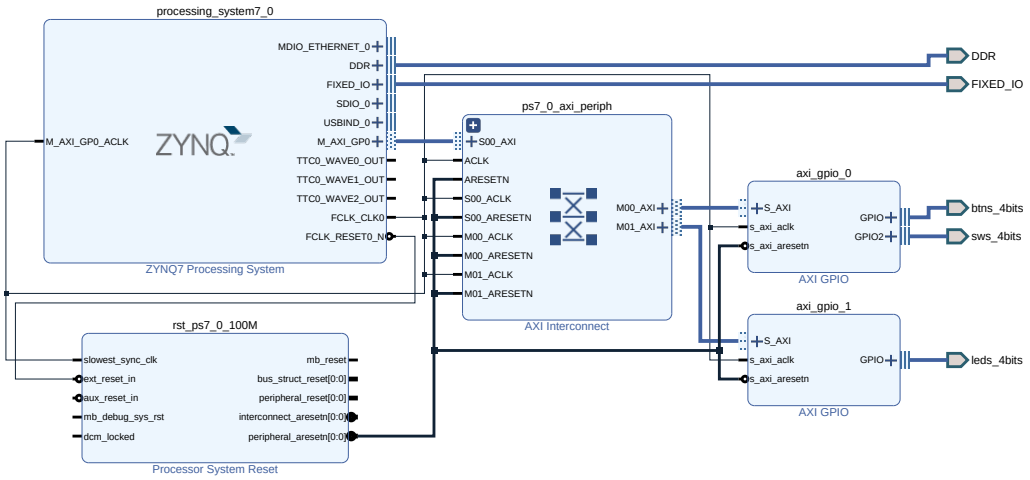

Figure 1: Hardware design used in this analysis, based on 2B, but without interrupts and two channels on `axi_gpio_0` *for input.*

In order to rename the board, first the Block Design was imported and saved as the Tcl script "zynq_dualchannel_system.tcl". In line 53 of the original Tcl script,

```
set design_name zynq_interrupt_system
```

`zynq_interrupt_system` was changed to `zynq_dualchannel_system`. Then the Tcl script was ran in the Tcl console with the command `source /Zynq_book/rtos_task_management/zynq_interrupt_system.tcl`

The second channel was added by first double clicking `axi_gpio_0`. Then on the right side of the dialog under the "Board" tag, in the table, the `sws_4bits` option was selected from the dropdown list under the "GPIO2" and "**Board interface**" headers. Then okay was selected. The design was saved.

"Run Connection Automation" was selected. It was reviewed that this was for GPIO2, and that the "Select Board Part Interface" option was set to "`sws_4bits ( sws_4bits )`". The design was validated and saved again.

Then an HDL Wrapper was created under the Block Design's source.

Finally, the Bitstream was generated, the Hardware was exported to SDK, and SDK was launched.

The C program

This program differed from Lab 1 in that, when the New Application was created, the OS Platform was chosen as freertos10_xilinx, rather standalone.

The C program was made in 5 iterations.

In the first iteration, the work was based on the FreeRTOS Hello World application. Specif-

ically, the application was used as a template to set up the FreeRTOS environment and the tasks. The timer and queue in the original program were not needed and therefore discarded. The LED counter was set up based on Lab 1. Unfortunately, I do not have a link to this code because I made a mistake including the file and it was overwritten.

In the second iteration, the TaskBTN feature was added, but this version could only control the LED counter, and its constants were defined. In this iteration, each button logic got its own debounce. This version can be found here rtos_task_management.c v1.2.

In the third iteration, the TaskSW feature was stubbed, and constants to enable and disable tasks were added. Some GPIO code was refactored that was written when `axi_gpio_0` only controlled buttons. I left it there for this version because I was having trouble understanding it. The button debounce was optimized to a single debounce for all button combinations, and the code was made more readable by using more constants. This version can be found here rtos_task_management.c v1.3.

In the fourth iteration, the TaskSW feature is completed. The trick to this code is that rather than polling whether switch 3 is retracted to resume the LED counter, switch 3 must be checked to be retracted immediately after checking if it was thrown and suspending the LED counter. I accidentally left in some debug code in TaskBTN. This version can be found here rtos_task_management.c v1.4.

In the final iteration, the TaskBTN feature was completed with the ability to control the switches as well. The code was cleaned by removing debugging code, adding logs and making these clearer. Of the switch states, only switch 3 off after switch 3 on is logged, because unlike buttons, the switches fire their events even if there is no change in stage. So this logging would be constant. This version can be found here rtos_task_management.c v1.5.

The results

The resulting analysis can be seen here `https://youtu.be/ucZpgsqakyc`.

Conclusions

This project teaches the student how to handle multiple tasks using FreeRTOS, which provides functions to create, delay, suspend and resume tasks, as well as a task scheduler. I believe I got a good understanding of the basics of handling tasks. The analysis itself seems complete except for the issue that switch 3 effectively renders the buttons for controlling the LED unusable.

This can be fixed by using a flag that is activated when button 2 is depressed, and deactivated when buttons 1 and 0 are depressed, or when switch 3 is retracted, then thrown. This flag would also signal to switch 3 to stop suspending the LED counter task.

Alternatively, from the user controlling the switches standpoint, the user can hold button 3, which suspends polling of the switches, then press button 2 to resume the LED counter until they released button 3 again.

Appendices

The following pages include the complete source code, the latest version of which is available at https://github.com/lduran2/ece3623-lab2-rtos_task_management/blob/master/rtos_task_management.sdk/rtos_task_management/src/rtos_task_management.c.

```
/*
    Copyright (C) 2017 Amazon.com, Inc. or its affiliates.  All Rights
Reserved.
    Copyright (C) 2012 - 2018 Xilinx, Inc. All Rights Reserved.

    Permission is hereby granted, free of charge, to any person
obtaining a copy of
    this software and associated documentation files (the "Software"),
to deal in
    the Software without restriction, including without limitation the
rights to
    use, copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of
    the Software, and to permit persons to whom the Software is
furnished to do so,
    subject to the following conditions:

    The above copyright notice and this permission notice shall be
included in all
    copies or substantial portions of the Software. If you wish to use
our Amazon
    FreeRTOS name, please do so in a fair use way that does not cause
confusion.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR
    IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS
    FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR
    COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER
    IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF
OR IN
    CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

    http://www.FreeRTOS.org
    http://aws.amazon.com/freertos


    1 tab == 4 spaces!
*/
/*
 * rtos_task_management.c
 *
 * Created on:  15 September 2020 (based on FreeRTOS_Hello_World.c)
 *     Author:  Leomar Duran
 *    Version:  1.5
 */

/*************************************************************************
**********************
* VERSION HISTORY
*************************************************************************
********************
*   v1.5 - 16 September 2020
```

```
*       Added TaskBTN feature that controls TaskSW.
*
*   v1.4 - 16 September 2020
*       Added TaskSW  feature that controls TaskLED and TaskBTN.
*
*   v1.3 - 16 September 2020
*       Stubbed TaskSW.  Optimized TaskBTN.
*
*   v1.2 - 16 September 2020
*       Added TaskBTN feature that controls TaskLED.
*
*   v1.1 - 15 September 2020
*       Set up LED counter.
*
*   v1.0 - 2017
*       Started with FreeRTOS_Hello_World.c
*
******************************************************************************
*******************/

/******************************************************************************
**********************
* TASK DESCRIPTION
******************************************************************************
*********************
* TaskLED := a counter from 0 to 15, then looping back to 0 that is
displayed in the LEDs.
*
* TaskBTN := reads the buttons to control the other tasks
*
* TaskSW  := reads the switches to control the other tasks
*
******************************************************************************
*******************/

/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
/* Xilinx includes. */
#include "xil_printf.h"
#define printf  xil_printf
#include "xparameters.h"
#include "xgpio.h"
#include "xstatus.h"

/* task definitions */
#define DO_TASK_LED 1                               /* whether to do
TaskLED */
#define DO_TASK_BTN 1                               /* whether to do
TaskBTN */
#define DO_TASK_SW  1                               /* whether to do
TaskSW */

/* GPIO definitions */
```

```
#define  IN_DEVICE_ID   XPAR_AXI_GPIO_0_DEVICE_ID   /* GPIO device for
input */
#define OUT_DEVICE_ID   XPAR_AXI_GPIO_1_DEVICE_ID   /* GPIO device that
LEDs are connected to */
#define LED 0b0000                                  /* Initial LED
value - 0000 */
#define BTN_DELAY   250UL                           /* button delay
length for debounce */
#define LED_DELAY   500UL                           /* LED delay length
for visualization */
#define BTN_CHANNEL 1                               /* GPIO port for
buttons */
#define  SW_CHANNEL 2                               /* GPIO port for
switches */
#define LED_CHANNEL 1                               /* GPIO port for
LEDs */

/* GPIO instances */
XGpio  InInst;                                  /* GPIO Device driver
instance for input */
XGpio OutInst;                                  /* GPIO Device driver
instance for output */

/* bit masks for on */
#define       ON4   0b1111                      /* 4 bit on  */
#define BTN10_ON    0b1100
#define  BTN2_ON    0b1011
#define  BTN3_ON    0b0111
#define  SW10_ON    0b1100
#define   SW2_ON    0b1011
#define   SW3_ON    0b0111

/* bit masks for off */
#define       OFF4  0b0000                      /* 4 bit off */
#define BTN10_OFF   0b0011
#define  BTN2_OFF   0b0100
#define  BTN3_OFF   0b1000
#define  SW10_OFF   0b0011
#define   SW2_OFF   0b0100
#define   SW3_OFF   0b1000
/*-----------------------------------------------------------*/

/* The tasks as described at the top of this file. */
static void prvTaskLED( void *pvParameters );
static void prvTaskBTN( void *pvParameters );
static void prvTaskSW ( void *pvParameters );
/*-----------------------------------------------------------*/

/* The task handles to control other tasks. */
static TaskHandle_t xTaskLED;
static TaskHandle_t xTaskBTN;
static TaskHandle_t xTaskSW;
/* The LED Counter. */
int ledCntr = LED;
/* The last value of button. */
int btn;
```

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

    if (DO_TASK_LED) {
        printf( "Starting TaskLED. . .\r\n" );
        /* Create TaskLED with priority 1. */
        xTaskCreate(    prvTaskLED,                 /* The function
that implements the task. */
                ( const char * ) "TaskLED",         /* Text name for
the task, provided to assist debugging only. */
                        configMINIMAL_STACK_SIZE,   /* The stack
allocated to the task. */
                        NULL,                       /* The task
parameter is not used, so set to NULL. */
                        ( UBaseType_t ) 1,          /* The next to
lowest priority. */
                        &xTaskLED );
        printf( "\tSuccessful\r\n" );
    }

    if (DO_TASK_BTN) {
        printf( "Starting TaskBTN. . .\r\n" );
        /* Create TaskBTN with priority 1. */
        xTaskCreate(
                    prvTaskBTN,                     /* The function
implementing the task. */
                ( const char * ) "TaskBTN",         /* Text name
provided for debugging. */
                    configMINIMAL_STACK_SIZE,       /* Not much need
for a stack. */
                    NULL,                           /* The task
parameter, not in use. */
                    ( UBaseType_t ) 1,              /* The next to
lowest priority. */
                    &xTaskBTN );
        printf( "\tSuccessful\r\n" );
    }

    if (DO_TASK_SW) {
        printf( "Starting TaskSW . . .\r\n" );
        /* Create TaskSW with priority 1. */
        xTaskCreate(
                    prvTaskSW,                      /* The function
implementing the task. */
                ( const char * ) "TaskSW",          /* Text name
provided for debugging. */
                    configMINIMAL_STACK_SIZE,       /* Not much need
for a stack. */
                    NULL,                           /* The task
parameter, not in use. */
                    ( UBaseType_t ) 1,              /* The next to
lowest priority. */
                    &xTaskSW );
        printf( "\tSuccessful\r\n" );
    }
```

```
    /* initialize the GPIO driver for the LEDs */
    Status = XGpio_Initialize(&OutInst, OUT_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        printf("GPIO output to the LEDs failed!\r\n");
        return 0;
    }
    /* set LEDs to output */
    XGpio_SetDataDirection(&OutInst, LED_CHANNEL, 0x00);

    /* initialize the GPIO driver for the buttons */
    Status = XGpio_Initialize( &InInst,  IN_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        printf("GPIO input from the buttons and switches failed!\r\n");
        return 0;
    }
    /* set buttons to input */
    XGpio_SetDataDirection( &InInst, BTN_CHANNEL, 0xFF);
    /* set switches to input */
    XGpio_SetDataDirection( &InInst,  SW_CHANNEL, 0xFF);

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

    /* If all is well, the scheduler will now be running, and the
following line
    will never be reached.  If the following line does execute, then
there was
    insufficient FreeRTOS heap memory available for the idle and/or
timer tasks
    to be created.  See the memory management section on the FreeRTOS
web site
    for more details. */
    for( ;; );
}


/*-----------------------------------------------------------*/
static void prvTaskLED( void *pvParameters )
{
const TickType_t LEDseconds = pdMS_TO_TICKS( LED_DELAY );

    for( ;; )
    {
        /* display the counter */
        XGpio_DiscreteWrite(&OutInst, LED_CHANNEL, ledCntr);
        printf("TaskLED: count := %d\r\n", ledCntr);

        /* Delay for visualization. */
        vTaskDelay( LEDseconds );

        /* update the counter */
        ++ledCntr;
    }
}


/*-----------------------------------------------------------*/
```

```
static void prvTaskBTN( void *pvParameters )
{
const TickType_t BTNseconds = pdMS_TO_TICKS( BTN_DELAY );
    int nextBtn;    /* Hold the new button value. */
    for( ;; )
    {
        /* Read input from the buttons. */
        nextBtn = XGpio_DiscreteRead(&InInst, BTN_CHANNEL);

        /* Debounce: */
        /* If the button has changed, */
        if ( btn != nextBtn ) {
            btn = nextBtn;  /* store the old value */
            vTaskDelay( BTNseconds );   /* delay until the end of the
bounce */
            nextBtn = XGpio_DiscreteRead( &InInst, BTN_CHANNEL );   /*
read again */
            /* if the button value is still the same, continue */
            if ( btn == nextBtn ) {
                printf("TaskBTN: Button changed to 0x%x:", btn);

                btn = nextBtn;  /* update btn */
                /* If BTN2 is depressed, regardless of the
                 * status of BTN0 and BTN1, then TaskLED is
                 * resumed.  So BTN2 gets priority. */
                if ( ( btn | BTN2_ON ) == ON4 ) {
                    vTaskResume(xTaskLED);
                    printf("TaskBTN: TaskLED is resumed.");
                }
                /* Otherwise if BTN0 and BTN1 are depressed at
                 * some point together then TaskLED is
                 * suspended */
                else if ( ( btn | BTN10_ON ) == ON4 ) {
                    vTaskSuspend(xTaskLED);
                    printf("TaskBTN: TaskLED is suspended.");
                }

                /* This logic below is independent from those above. */
                /* If BTN3 is depressed then TaskSW is suspended */
                if ( ( btn | BTN3_ON ) == ON4 ) {
                    vTaskSuspend(xTaskSW);
                    printf("TaskBTN: TaskSW  is suspended.");
                }
                /* Either BTN3 is depressed or it is released. */
                /* If BTN3 is released then TaskSW is resumed */
                else {
                    vTaskResume(xTaskSW);
                    printf("TaskBTN: TaskSW  is resumed.");
                }

            } /* end if ( btn == nextBtn ) check if button is
consistent */
        } /* end if ( btn != nextBtn ) check if button has changed
since last call */
    } /* end for( ;; ) */
}
```

```
/*-----------------------------------------------------------*/
static void prvTaskSW( void *pvParameters )
{
    int sw; /* Hold the current switch value. */
    for( ;; )
    {
        /* Read input from the switches. */
        sw = XGpio_DiscreteRead(&InInst,  SW_CHANNEL);

        /* If SW0 and SW1 are ON together at some point then
         * TaskBTN is suspended */
        if ( ( sw | SW10_ON ) == ON4 ) {
            vTaskSuspend(xTaskBTN);
        }
        /* Switches 0 and 1 cannot both be off if they're both on. */
        /* If SW0 and SW1 are OFF then TaskBTN is resumed. */
        else if ( (sw & SW10_OFF ) == OFF4 ) {
            vTaskResume(xTaskBTN);
        }

        /* This logic below is independent from those above. */
        /* If SW3 is ON then TaskLED is suspended. */
        if ( ( sw | SW3_ON ) == ON4 ) {
            vTaskSuspend(xTaskLED);
            /* If SW3 is then turned OFF, then resume TaskLED. */
            sw = XGpio_DiscreteRead(&InInst,  SW_CHANNEL);
            if ( ( sw & SW3_OFF ) == OFF4 ) {
                vTaskResume(xTaskLED);
                printf("TaskSW : TaskLED is resumed.");
            }
        }
    }
}
```