COMPSYS723 Lab Manual

Application Programming with FreeRTOS on DE2-115

Wei-Tsun Sun, Isuru Pathirana
Updated in 2017 by Johnny Yeh and Benjamin Tan

Table of Contents

Introduction

Welcome to COMPSYS 723 2017! This lab exercise is designed to introduce you to embedded software design with the Nios II Software Build Tools (SBT). We will be using a system-on-programmable-chip based on Nios II, which is a soft-core processor. Soft-core processors are designed in a hardware description language (HDL) such as VHDL or Verilog. Designers can combine the processor with other soft-core components to create a full embedded system that can be programmed onto an FPGA.

Required resources:

(Please ensure you have all the equipment necessary. If not, talk to your TAs)

- Quartus II version 13.0 or above
- Nios II Software Build Tools (Eclipse)
- DE2-115 development board and cables (USB and power)
- PS/2 Keyboard
- VGA cable
- Lab Files from Canvas

Learning outcomes:

- Learn how to create a new software project in Nios II SBT
- Understand the components of the software project
- Understand how software can be written to control peripherals in your System-on-Chip using the Hardware Abstraction Layer
- Program your FPGA development board with an SoC
- Run your program on the board
- Use API functions provided by a real-time operating system

Background Information

The system was designed using the Qsys tool which is a part of the Quartus II design package. Qsys is similar to SOPC builder - a tool that was bundled with earlier versions of Quartus which you may have encountered previously. This tool provides a graphical interface for creating systems-on-programmable-chips.
These systems typically contain components that allow you to control various things, such as the switches, LEDs and buttons on the DE2-115 board, or the off-chip SRAM/SDRAM.
The hardware components of the Nios II system can be categorized into several groups, according to how they are used in the practice.

1. Memory Components – these can be read-only (ROM), random-access (RAM) or other types according to their characteristics.
2. Input/Output (I/O) peripherals – An embedded system interacts with its environment through input/output peripherals. You can read from/write to these peripherals either by accessing memory-mapped registers directly or by calling available APIs (application programming interfaces) in the board support package.
3. Integrated peripherals – These are peripherals that are integrated into the Nios II processor. An example of this is the timer peripheral, which can be used after setting up its parameters. A hardware timer is often required by system software (such as a real-time operating system).

We will make use of these peripherals in the following sections.

Part 1 – Setting up a new workspace and project

In this part of the lab, we will introduce you to the Nios II SBT (which might also be referred to as the Nios II IDE).

YOU WILL FIRST NEED TO CONFIGURE THE FPGA BOARD USING THE QUARTUS II PROGRAMMER, WITH THE PROVIDED .SOF FILE. REFER TO THE EXTRA-INFO DOCUMENT.

Instructions:

1. Download the lab folder from Canvas and unzip it to a sensible location.
2. Create a new folder which will serve as the location for your project work and copy freq_relay_controller.sof and nios2.sopcinfo into the folder. Choose a directory using the following guidelines:

   - There should be **no spaces** in the name of the directory. For instance, paths such as "c:\compsys 723 lab\" or "c:\compsys723\lab test 1" are not acceptable.
   - The NIOS II Eclipse software uses absolute paths in project files. As a result, your project should always be in the same path. If the path of the project directory has changed, then you will need to import the project into your workspace again.
   - You can use H: drive to avoid access issues that may otherwise occur, however, this may increase your login/logout times as H: drive is synced with the university network. You can also use a USB flash drive. You can also use your own computer.
   - **"c:\compsys723\lab" is used in this document**

3. Start the Nios II IDE from the start menu.
4. A dialog window will appear, asking you to specify a **workspace**. Browse to the folder you created in step 2.

   A workspace is used to store the current state of the IDE. It stores the currently opened projects and their associated configurations. You can also change the current workspace to an existing workspace or a new workspace using the File menu (File->Switch Workspace).

5. To create an application, select "**File->New->Nios II Application and BSP from Template**" from the main menu and perform the following steps:

   A. Choose nios2.sopcinfo for the SOPC Information File name.
   B. Choose **nios2** for the **CPU name**.
   C. Input hello_lab for the Project name.
   D. You should see "c:\compsys723\lab\software\hello_lab" appear as the default project location. (or something similar)
   E. Choose "Hello World" for the Project template.
   F. Click "Next".

6. Compile and run the first Nios II program in this lab

   To build and run the Nios II program, perform the following:

   A. Right-click on the **hello_lab** project and select "**Build Project**" to start compiling. You don't need to do this for the BSP project as it is a dependency of the hello_lab project, so the BSP project will be built automatically if necessary (for instance when the BSP settings are modified).
   B. To run the application, click "**Run**" from the menu bar and select "**Run Configurations**". **Right-click** on "**Nios II Hardware**" and select "**New**".

C. Make sure the **Project** tab is selected and use the name “hello_lab” for this configuration. Make sure the **project name** is **hello_lab** by selecting it from the dropdown box.
D. Now change to the “**Target Connection**” tab. Select “**USB-Blaster**” for the **Processor** and **Byte Stream Device**. If **USB-Blaster** is not displayed, make sure the board is connected to the computer with the provided SOF file correctly downloaded onto it and click **Refresh**. Now click **“Run”** on the lower right corner. If code in the project has changed, the project may be re-built before being loaded onto the processor.
E. You should soon be able to see the progress of the building and downloading process in the Console tab of the IDE. *“Hello from Nios II”* will be displayed in the “Nios II Console”, once the application has been downloaded and executed successfully. Click  to terminate the execution.
F. You may ignore mismatched system ID and timestamp with the target platform when running your program

What’s going on?

When we created the project with a BSP, two folders can be seen in the project navigator window. The first folder is your software, in this case, the hello world program. Any c source code you write goes into this folder.

The second folder is the BSP, or Board Support Package. When we specified the sopcinfo file, the SBT automatically generated this folder and all the things inside it. This BSP contains **Hardware abstraction layer (HAL)** that is specific to our system. When we write application software, we utilize the HAL to manipulate the different hardware components. It is very difficult to write software for hardware directly, so the BSP includes low level routines to control hardware and exposes APIs to allow easier control of hardware in a user application. The BSP is **platform dependent**. The HAL is also responsible for initializing the system and devices before the user application is run.

The **software** (the combination of the BSP and application logic) is stored in memory. When the processor is **reset**, it begins executing from a specific location in the memory

Qsys generates a **.sopcinfo** file which includes information about the Nios II system. This information includes the processor type, available hardware and internal interconnections.

Why don’t you have a look at it in a text editor?

Part 2 – Using some of the peripherals

In this section, we will introduce to you the use of the peripherals in your system, such as the LEDs, SWITCHES and LCD. We'll also have a look at system.h, which is an integral part of the board support package.

Accessing components in the Nios II system

As mentioned previously, nios2.sopcinfo contains information about the components in the Nios II system. Each component is described with a set of fields including:

1. The address of the component: this is the base address which the component is mapped to in the processor's address space.
2. The IRQ of the component: indicates the interrupt number of the component. This is only applicable for components that generate interrupts.
3. Other information.

However, even though it is stored in clear text, nios2.sopcinfo is not programmer friendly. The **system.h** file, which is produced during the BSP build, presents the same information in a more readable manner. You can find system.h in the **hello_lab_bsp.**

The system.h file consists of entries in the form of symbolic constants which are defined with the **#define** macro. For instance, the base address of the timer1ms component (0x43040) is assigned with a symbolic name of "**TIMER1MS_BASE**". You can use **TIMER1MS_BASE** in your application code instead of a hard-coded base address value. This increases the portability of the program from one configuration to another (in cases where the names of the modules are the same in both configurations but the base address values may differ).

The base address can be considered a hardware component's identifier. A component is accessed by using its base address as an argument to function calls provided by the BSP.

In order to use the symbolic constants in the system.h file, add **#include <system.h>** to the beginning of your application code (in this case, the **hello_world.c** file within the **hello_lab** project).

Accessing PIO in the Nios II system

Functions for accessing PIO (Parallel Input/Output) registers are provided in the **altera_avalon_pio_regs.h** file, therefore this file must also be included.

The peripherals on the DE2-115 board can be divided into three groups:

1. **Output only**- includes the red and green LEDs on the board and the seven-segment displays. Values are written to the peripheral using the **IOWR_ALTERA_AVALON_PIO_DATA(BASE_ADDRESS, value)** function.
2. **Input only**– includes the 18 switches and 3 push buttons on the board. Values are read from these peripherals using the **IORD_ALTERA_AVALON_PIO_DATA(BASE_ADDRESS)** function.
3. **Character-based**– includes the character LCD and the UART peripherals. These peripherals are accessed using file descriptors. They are initialized by creating a file descriptor with the **fopen** function. The **fprintf** function can then be used to output characters from the peripherals; more details will be introduced later in the lab.

The **example_lcd_led_switches_buttons** directory of the lab's resource package illustrates an example of programming the Nios II to interact with peripherals on the DE2-115 board.

You can copy the entire contents of the **example_lcd_led_switches_buttons** directory to your hello_lab

project directory (e.g c:\compsys723\lab\software\hello_lab). After copying the files into your project directory, right-click the project in the Project Explorer in the Nios II IDE and select **"Refresh"**. Re-build the project and run it on the Nios II processor. Refer to previous sections for instructions on building and running a Nios II application.

Writing an interrupt service routine (ISR)

Preparation

The interrupt service routine (ISR) is a specialized function which is associated with an IRQ of a corresponding hardware component. The registered ISR will be activated or called when the interrupt occurs.

To implement an ISR, these requirements must be met:

1. The hardware peripheral in question must be capable of generating an interrupt. Output-only peripherals usually do not meet this criterion. An IRQ number is assigned (not -1) to each peripheral in system.h which is able to generate interrupts.
2. You have to register the ISR with the associated interrupt. This configures the ISR to run when the interrupt is triggered.
3. Within the ISR, you may need to do some house-keeping work such as clearing the flags which caused the interrupt, or re-enabling the interrupt if required (sometimes the interrupt is disabled once the ISR is entered).

Looking at the definitions for PUSH_BUTTON in system.h, we can see that **PUSH_BUTTON_IRQ** is defined. We are also going to use **PUSH_BUTTON_BASE** instead of hard coding the address of the buttons.

PIO and ISR relevant functions

IORD_ALTERA_AVALON_PIO_IRQ_MASK(BASE_ADDR): This will return an "interrupt mask" value that indicates which input ports of a component have interrupts enabled. For instance, when interrupts are enabled for the first 3 buttons, a value of 7 (**111** in binary, which "masks" the first 3 bits) will be returned.

IORD_ALTERA_AVALON_PIO_IRQ_MASK(BASE_ADDR, value): This function is used to set the interrupt mask which is explained above.

IORD_ALTERA_AVALON_PIO_EDGE_CAP(BASE_ADDR): This function returns the value of the edge capture register of a PIO peripheral. During system generation, a PIO core can be configured to capture different types of edges, such as low-to-high transitions and high-to-low transitions. When an edge from an input port is captured, the corresponding bit is set in the edge capture peripheral. In the system provided in this lab, the buttons PIO peripheral has been configured to trigger an interrupt when any of the bits in the edge-capture register have been set. The value returned by this function can be read inside an ISR in order to determine which input port (a button in this case) caused the interrupt.

IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BASE_ADDR, value): This function writes a value to the edge capture register which was explained above.

Register the PIO ISR

The ISR implementation has to have a function prototype of this form:

```
void name_of_the_isr_function (void* context, alt_u32 id)
```

The first argument is a void pointer. This means that you can pass any kind of pointer to the ISR function. The second argument is the id of the interrupt, this is the IRQ number which the ISR is associated with. For

the buttons peripheral, it is **PUSH_BUTTONS_IRQ** as mentioned previously. Both the context pointer and the interrupt id must be passed to the ISR register function as well.

The ISR registering function has the following signature:

int alt_irq_register (alt_u32 id, void* context, void (*isr)(void* , alt_u32))

The first argument is the IRQ number of the interrupt. The second argument is a pointer to anything which the programmer may wish to pass to the interrupt (if you are working with global variables, then you can just pass NULL to it). The third argument is a **function pointer**, which points to the ISR function. To use these functions you need to include "**sys/alt_irq.h**"

The code **example_button_isr** shows an example of a button ISR which changes the value of the context passed to it. The value which is stored, indicates which button has been pressed. This value is then output to the green LEDs.

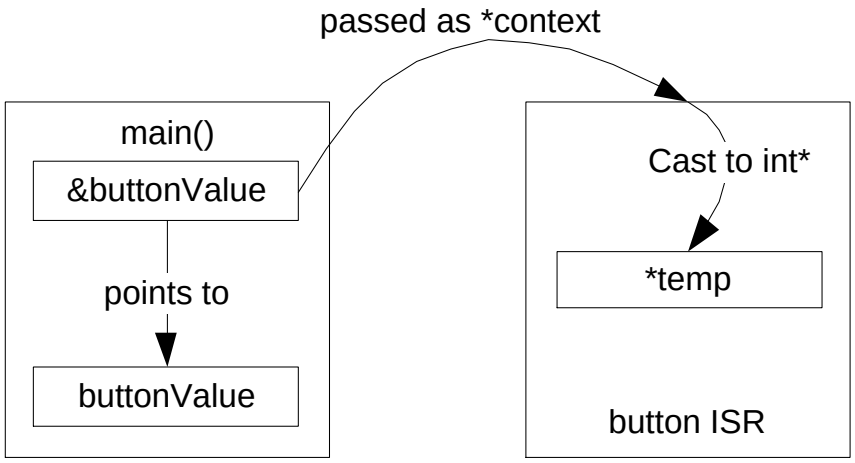

Figure 1: How the context pointer works in ISR

Figure 1 illustrates how the *buttonValue* variable is changed by the ISR. In the main function, the address of *buttonValue* (obtained using *&buttonValue*) is passed as the context to the button ISR when registering the ISR in line 27. When an interrupt is triggered, the context is passed to the button ISR, where it is cast back to an integer pointer (**temp*). As a result, the ISR can modify the value of the *buttonValue* variable using the *temp* pointer.

Using Timer ISRs

A timer ISR is a bit different to the PIO ISRs. A timer ISR must have a function prototype of this form:

alt_u32 timer_isr_function(void* context)

The return value of the timer ISR is the timeout value of the next timer iteration (which will be described in further detail below). If it is 0 then the timer is stopped once control returns from the timer ISR function. The timer ISR does not need to be registered like a general ISR. Instead, the timer must be started using the following function:

```
int alt_alarm_start(      alt_alarm* timer_pointer, alt_u32 time_to_run,
                          alt_u32 (*function_pointer_to_the_timer_isr) (void* context),
                         void* context);
```

The first argument is a pointer to an *alt_alarm* structure which is used by the HAL to manage the timer. The second argument is the time period after which the first timeout is generated. The third argument is a

function pointer to the ISR function for the timer. The last argument is the context pointer that is passed to the timer ISR. The function will return a negative value if the timer fails to start. To use these functions, you need to include "**sys/alt_alarm.h**"

Using a PS/2 peripheral

The PS/2 peripheral is an Avalon Slave which communicates with PS/2 devices such as a keyboard. The peripheral can be used in application code by either polling or by using an interrupt. Software routines are provided for using a keyboard that is connected to the PS/2 port. In order to use these routines, you need to include the **altera_up_avalon_ps2.h** and **altera_up_ps2_keyboard.h** header files. An example of using a PS/2 keyboard as an input device can be found in the ***example_ps2*** folder of the lab's resource package.

In the previous example, the **read_make_code** function which decoded the key event and key code, blocks the main loop of the program. To avoid this, we can use an interrupt service routine as seen in the ***example_ps2_irq*** example.

What is the difference? Which is better?

Using the VGA display controller

You can find example codes (char_buffer.c and pixel buffer.c) on the use of VGA controller in '*Additional example codes* '. Go through the example code for clues as to how you might display information on the screen.

Using the Frequency analyser

You can find details about this component in the assignment hand-out and example codes in '*Additional example codes* '.

Part 3 – Intro to FreeRTOS

In this section we will introduce you to FreeRTOS, the operating system that will be used as the basis for your assignment. We will explore the use of FreeRTOS on the Nios II processor. A real-time operating system (RTOS) can be utilized to run software with multiple tasks or threads on the provided single processor hardware platform.

Important Resources

- http://www.freertos.org/ - FreeRTOS main page
- http://www.freertos.org/a00106.html - API Reference
- Your lecture slides
- FreeRTOS code (provided on Canvas)

Create a simple software application with FreeRTOS

In order to use FreeRTOS in our application, we need to include a number of essential files.

Create a new Nios II application project (called **freertos_test**) in addition to an associated BSP (called **freertos_test_bsp)**. Make sure you remove any C files that may be generated for you in the *application* project directory to prevent errors due to an existing definition of a main function.
Copy the ***freertos*** directory from the lab's resource package to the *folder of the application* software, for instance C:\compsys723\lab\software\freertos_test. Also copy the example application's source code (***freertos_test.c*** from the *freertos_test* directory in the resource package) to the project directory as shown in Figure 2.

You should see the **freertos** folder and **freertos_test.c** in the project explorer of the Nios II IDE after right-clicking the freertos_test project and clicking on "**Refresh**".

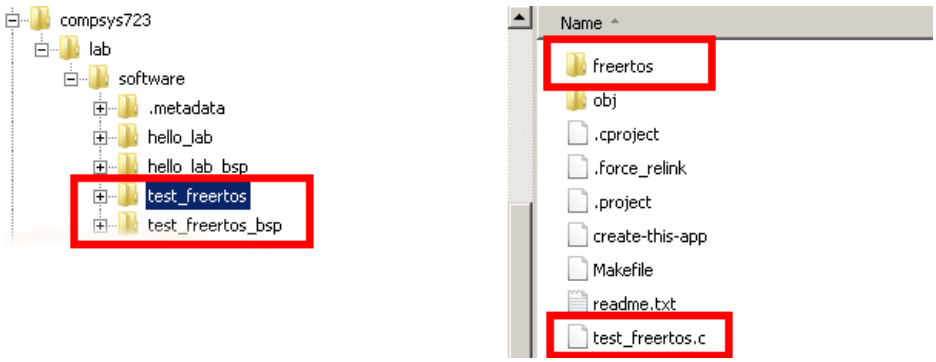

Figure 2: Folder view of the application directory

In order for the provided FreeRTOS files to work, you may need to modify the **system.h** file in the BSP project (**freertos_test_bsp**).

Make sure the following entry is present:
#define ALT_LEGACY_INTERRUPT_API_PRESENT
(as opposed to ALT_ENHANCED_INTERRUPT_API_PRESENT)

The system.h file is re-generated whenever the BSP settings are modified. As a result, this change must be re-applied any time the BSP settings are updated.

In this example, two tasks are created with the same priority. Each task prints out an identifier string (e.g. Task 1 or Task 2), and waits for 1 second by calling the **vTaskDelay** function. To run the example, return to the IDE to refresh and build the **freertos_test** project. Run the project using the procedure described in the

previous section. **What do you see? Why?**

Look into more details, communication and synchronization between threads

You may now proceed to the next example: **freertos_semaphore**, which can also be found in the lab resource package. Replace *freertos_test.c* with *freertos_semaphore.c*

Tasks of a FreeRTOS application have the following signature:

void task_function_name(void *pvParameters) { ... }.

A task function usually contains an infinite loop. Within this loop there can be one or more RTOS API calls, such as time delay functions or functions which acquire semaphores. The task may release processor control explicitly using these mechanisms or it may be pre-empted by a task with a higher priority.

Each task has its own stack with a stack-size that is specified when creating the task. In this example, the **TASK_STACKSIZE** constant is used to define the size of the stack of each task. Semaphores and queues are represented using the **xSemaphoreHandle** and **xQueueHandle** types. They are usually declared globally, for instance, **shared_resource_sem** in this example is a global semaphore which can be used by any task.

The application consists of the following tasks:

1. **print_status_task**: prints out the status of the system through the JTAG UART port. This output is visible in the Nios II terminal program.
2. **getsem_task1**: one of two tasks which share a single semaphore. This task attempts to take the **shared_resource_sem** semaphore. If the other task has obtained the semaphore, this task will be blocked until the other task releases it. Once the semaphore has been obtained, a counter variable which is printed in the **print_status_task** is incremented. The semaphore is then released.
3. **getsem_task2**: functions in a similar manner to **getsem_task1** except this task increments a different counter variable once it obtains the semaphore.
4. **send_task**: this task sends messages to a message queue named **msgqueue**. Messages are only sent when the queue is not full. If the queue is full, the task is delayed for 1000 clock ticks before it attempts to send to the queue again.
5. **receive_task1**: one of two tasks which receive messages from the **msgqueue**. After receiving a message from the queue, the task increments a message received counter variable which the print status task will display. The task is then delayed for 333 ticks.
6. **receive_task2**: functions in a similar manner to **receive_task1** except after receiving a message from the queue, it increments a different counter variable. Also, it is delayed for 1000 ticks instead of 333 ticks.

The flow of the application is as follows:

1. The main() function of the program calls the ***initOSDataStructs*** function which creates a semaphore and a message queue. The 6 tasks in the application are then created with the ***initCreateTasks*** call.
2. Once the tasks have been created, the **vTaskStartScheduler** function is called in the main function in order to command FreeRTOS to begin scheduling the tasks. The RTOS schedules the ready task with the highest priority to run.

Run the program and observe the outputs. What is happening? Why?

Do it yourself: Create a Counting Task

Now, try and modify the example program so that your system increments a counter and displays it on the

seven-segment displays of the DE2-115.

Hints:

- You will need a function to access the PIO peripheral. Add the header for accessing PIO peripherals.
- You will need to define a new task priority with the lowest priority
- Create the task body, and make the task wait for 1 second before incrementing the count

```
void counter_task(void *pvParameters){
	int counter = 0;
	while(1)
	{
		counter++;
		// write a value to the seven-segment display
		// make the task delay for 1 second
	}
}
```

- Add a function call to create the task in the ***initCreateTasks(void)*** function:

```
xTaskCreate(
counter_task,    // the function used to be task
"counter_task", // the name of the task, for debugging purpose
TASK_STACKSIZE, // the size of the stack
NULL,           // the parameter passed to the task, in this case, none.
counter_task_priority, // the priority of the task
NULL          // task handler, which can be used to delete the task
);
```

Now build the application and run it to see if it works. You can use the **freertos_counter.c** file of the **freertos_counter** directory in the resource package to verify your solution.

Do it yourself: Modify the application to make two tasks share a single LCD display

In this exercise, we will add two tasks to the application which both write to the LCD display. The tasks will display the messages “This is LCD task 1” and “This is LCD task 2” respectively. As mentioned previously, the LCD device can be accessed via a file descriptor. The **LCD_NAME** symbolic constant, which resolves to the filename of the LCD, is provided by system.h. The operation of writing characters to the LCD with **fprintf**, is **non-atomic** (i.e it can be interrupted). Therefore, if the execution of the two tasks are interleaved, the output on the LCD may also be interleaved. For instance, “**ThiThs iis LC LCDD task task12**” may be printed. A mechanism is required to make sure that each task finishes writing a complete message to the screen before it is pre-empted by another task. Furthermore, the message should be displayed long enough for a human to read; therefore, the message should be displayed for at least 1 second before a new message can be displayed.

A few design decisions are made for the two new tasks:

1. The new tasks will be assigned lower priorities than the existing ones.
2. The size of the stack of each new task will be the same as the other tasks in the application.
3. A semaphore will be used as the mechanism which ensures that the character LCD is only used by a single task at a time.
4. Task 1 will have a delay of 5 seconds, Task 2 will have a delay of 3 seconds.
5. The character LCD must be initialized before the two new tasks are started.

Some code segments are provided in order to create a design based on the decisions outlined above. You need to insert them into suitable locations in your application code:

Skeletons of the two LCD tasks:

```
void lcd_task1(void* pvParameters)
{
  while(1)
  {
    // insert some code here to request semaphore
```

```
    // clear the LCD
    fprintf(lcd, "LCD task 1\n");
    // insert some code here to release semaphore
    // insert the time delay
  }
}

void lcd_task2(void* pvParameters)
{
  while(1)
  {
    // insert some code here to request semaphore
    // clear the LCD
    fprintf(lcd, "\nLCD task 2\n");
    // insert some code here to release semaphore
    // insert the time delay
  }
}
```

Define the priorities of the tasks:

```
#define lcd_task1_priority 13
#define lcd_task2_priority 14
```

Create the tasks:

```
  xTaskCreate(
lcd_task1,
"lcd_task1",
TASK_STACKSIZE,
NULL,
lcd_task1_priority,
NULL
);

  xTaskCreate (
lcd_task2,
"lcd_task2",
TASK_STACKSIZE,
NULL,
lcd_task2_priority,
NULL
);
```

Declare the semaphore:

```
xSemaphoreHandle shared_lcd_sem;
```

Create the semaphore (Question: why is 9999 used for the first argument?):

```
shared_lcd_sem = xSemaphoreCreateCounting( 9999, 1 );
```

Request the semaphore to obtain access to write to the LCD:

```
xSemaphoreTake(shared_lcd_sem, portMAX_DELAY);
```

Release the semaphore to allow the other task to access the LCD:

```
xSemaphoreGive(shared_lcd_sem);
```

Delay 5 seconds to make sure the message lasts long enough to be seen:

```
vTaskDelay(5000);
```

Delay 3 seconds to make sure the message lasts long enough to be seen:

```
vTaskDelay(3000);
```

Create the file descriptor for accessing the LCD:

```
FILE *lcd;
```

Open the LCD in write mode:

```
lcd = fopen(CHARACTER_LCD_NAME, "w");
```

Code to clear the LCD:

```
#define ESC 27
#define CLEAR_LCD_STRING "[2J"
fprintf(lcd, "%c%s", ESC, CLEAR_LCD_STRING);
```

Close the LCD file descriptor:

```
fclose(lcd);
```

Show the TAs how you are going to place these segments into the application template to achieve the requirements. This lab is not marked, however the concept of sharing peripherals and data will be frequently used in the assignment, and would greatly affect the correctness of your solution.