Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze

Prof. Lothar Thiele

Embedded Systems - HS 2020

Sample solution to Lab 4

Date : 18.11.2020

Interactive FreeRTOS Sensor Application

Goals of this Lab

In this last lab session, you will combine your knowledge from previous Embedded Systems labs in an interactive sensor application. At the end of this session, you will be able to:

- write your own FreeRTOS tasks from scratch.
- combine individual tasks and data queues to build a FreeRTOS embedded application.
- understand how interrupts are handled in the context of FreeRTOS.

Introduction

In this lab, you will program an interactive FreeRTOS sensing application to review and apply the concepts learned in previous labs. While we provide you with an implementation of the functionality for most of the application's tasks with the project template, you are going to implement one of them on your own and combine all of them to build the final application.

Sensor Hardware The MSP432 LaunchPad itself does not include any sensors. To build an interactive application, you therefore get a sensor extension board (also called BoosterPack[1]) for this lab. More specifically, you are going to use the SENSOR-BOOSTERXL board that includes two motion and three ambient sensors. The focus in this lab will be on using the `BMI160` accelerometer. All relevant documentation of the sensor booster pack and the datasheets of the individual sensors can be found in the `boosterpack_sensors` directory of the *Supplementary Material* (`lab_documents.zip`) for the labs on the course website. Make sure to connect the sensor extension board as described in its user guide before starting with the lab.

Application Functionality and Structure The FreeRTOS application you are going to implement in this lab consists of four tasks, three queues as well as one interrupt service routine that handles two different interrupts. After system and task initialization, the application reads new accelerometer sensor values using a *Sensor* task and stores them in the sensor data queue. The *Processing* task reads this data and calculates the LaunchPad's attitude from it. In this lab, the LaunchPad's attitude is its rotation with respect to an inertial reference frame (i.e. the desk's surface). Mathematically, the 2-dimensional attitude is defined as the tuple $\{\theta_x, \theta_y\}$. Angle θ_x refers to the rotation between the inertial reference plane $(\vec{x}, \vec{y})$ and LaunchPad's $\vec{x_s}$-axes, while angle θ_y refers to the rotation between corresponding $\vec{y_s}$-axes. The definitions of these angles are also illustrated in Figure 2. This will be useful to implement the *Processing* task functionality in Task 2.3. If the resulting attitude changes more than 5° in any direction,

[1] More information about the BoosterPack ecosystem can be found at http://www.ti.com/boosterpacks.

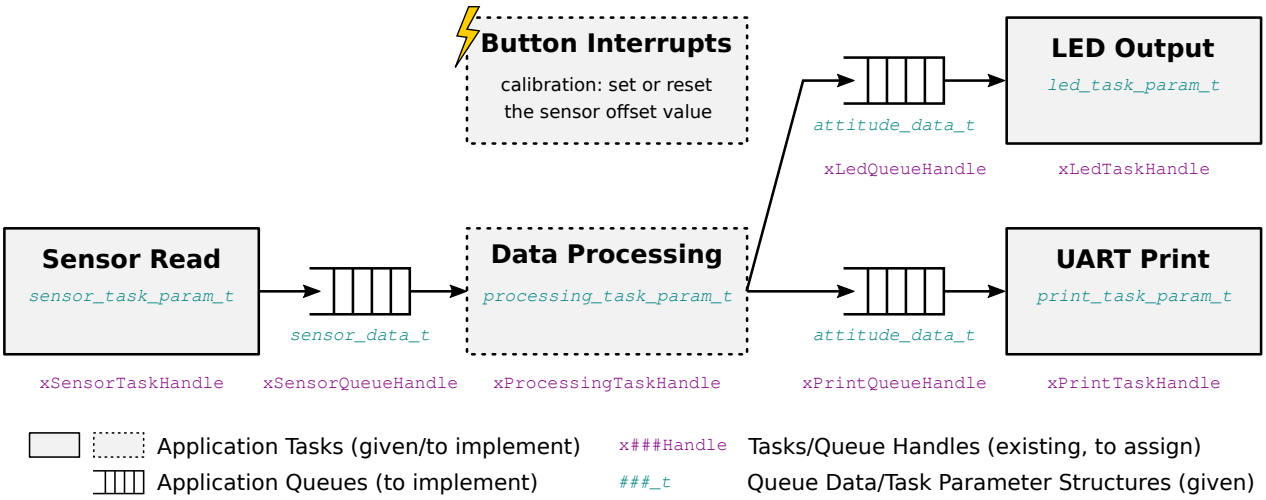

Figure 1: High-level data flow of the application you are going to implement in this lab. Components provided to you in the project template are highlighted. The data types in the task boxes or below the queue symbols indicate which data structures you are going to use for task arguments or queues. Below each task and queue, the variable name for the respective task and queue handles are given.

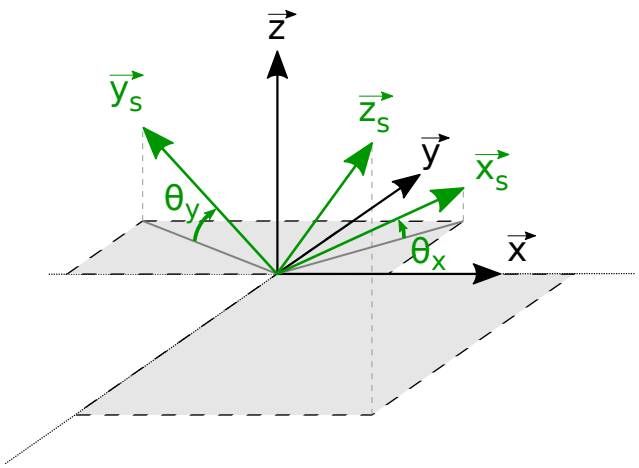

Figure 2: The LaunchPad's 2-dimensional attitude is defined by the angles θ_x and θ_y, referring to the rotation between the inertial reference plane $(\vec{x}, \vec{y})$ and the LaunchPad's $\vec{x_s}$- and $\vec{y_s}$-axes, respectively. For the calculation of these angles later in this lab, we assume that no other force than earth's gravitational pull is applied to the LaunchPad.

the new attitude is added to two output queues, the *Print* and *LED* queues. The *Print* task prints the new attitude to the UART interface, while the *LED* task switches the on-board LED2 from green to red if the LaunchPad was tilted beyond a critical angle of 30°. The last component is an interrupt handler, which stores or resets the accelerometer's sensor offset values when the buttons S1 or S2 are pressed, respectively. The interaction of the tasks and queues, as well as their arguments, data types, and already declared handler variables are shown in Figure 1. The *Sensor*, *Print* and *LED* tasks and all required data structures and defines are already provided in the project template. You are going to implement the remaining parts of the application step-by-step in this lab.

Project Structure For this lab, we provide you with a template project named `lab4` as a Zip file (`lab4.zip`). Import this project directly into your workspace using *Code Composer Studio*. While the project contains numerous files to separate different hardware modules, you will only edit the `app.c`, `gpio.c`, `main_freertos.c` and `interrupt.c` files and look up some predefined data structures and

Table 1: Project source files and folders of the `lab4` project. System startup code and linker files are omitted. The last column shows whether a file will be modified during this lab.

File/Folder Name	Description	Modified in
`FreeRTOS`	FreeRTOS source and header files	-
`libraries`	MSP432 DriverLib and Bosch `BMI160` sensor library code	-
`app.c`	Application tasks are implemented in this file	Task 2
`app.h`	Definitions and data structures for all tasks	-
`FreeRTOSConfig.h`	FreeRTOS configuration header file	-
`gpio.c`	GPIO initialization and interrupt setup	Task 3
`gpio.h`	GPIO definitions	-
`interrupt.c`	Interrupt service routines	Task 3
`interrupt.h`	Definitions for interrupts	-
`main_freertos.c`	Main program initializing the FreeRTOS tasks and queues	Task 1, 2
`sensor.c/h`	I2C interface code for the sensor library	-
`uart.c/h`	UART interface initialization and communication	-

Table 2: Task functionality and parameter overview.

Task	Task Handle	Parameters (Queue Handle)	Function Name
Sensor	`xSensorTaskHandle`	output (`xSensorQueueHandle`)	`prvSensorTask`
Processing	`xProcessingTaskHandle`	input (`xSensorQueueHandle`) output 1 (`xPrintQueueHandle`) output 2 (`xLedQueueHandle`)	`prvProcessingTask`
Print	`xPrintTaskHandle`	input (`xPrintQueueHandle`)	`prvPrintTask`
LED	`xLedTaskHandle`	input (`xLedQueueHandle`)	`prvLedTask`

constants in the corresponding header files. A more detailed summary of the functionality implemented by the individual files is given in Table 1. For reference, when implementing the tasks and queues, the task handle names, required parameters, and task function names are listed in Table 2. For more details on functions, data structures and definitions, we refer you to the documentation in the corresponding source and header files.

General Remarks If you reset the application while it accesses the accelerometer, the program locks up during sensor initialization upon restart because the sensor is unresponsive (this can occasionally happen). As a consequence, both LEDs remain dark and you will likely find the processor executing an infinite loop inside a DriverLib function called `I2C_isBusBusy`. A power cycle of the LaunchPad typically solves this issue: simply unplug the USB connection, wait some seconds for buffer capacitances to discharge, and replug it again.

Task 1: Combining Tasks to Form a FreeRTOS Application

To get started with today's lab, download the `lab4.zip` project template from the course webpage and import it into your workspace using *Code Composer Studio*.

Open `main_freertos.c` and scroll to the `main` function. It only contains some startup code to initialize the GPIO and UART peripherals, followed by two long comments summarizing the tasks to implement in this file and a UART print function call. When running this application (the template is a valid C program that compiles and runs), nothing happens beyond printing that initialization is complete. In contrary to the printed message, none of the tasks and queues described above are initialized or executed.

In this first task, you will extend the `main` function to create the queues and the tasks that are already provided by the template.

Task 1.1: Task Parameters and Priorities

Before starting with any implementation, you should have a closer look at the `app.h` header file. All task and queue related data structures and constants are already defined there.

(a) Locate the definitions for the task priorities, named `app###_TASK_PRIORITY`, where `###` is the name of the task. In which order do you expect FreeRTOS to start the tasks? Assume that the tasks block after initialization (e.g. on a queue read).

(b) How many data elements is the print queue going to store?

Solution for Task 1.1:

(a) The task priorities are increasing in the following order: *Sensor*, *Processing*, *LED*, *Print*. FreeRTOS will first run the *Print* task and then continue with the next lower priority task as soon as it encounters a blocking queue read.

(b) The `appPRINT_QUEUE_LENGTH` macro defines a queue length of a single element.

Task 1.2: Creating Data Queues

In the `main` function, add the necessary code to create the three queues shown in Figure 1. All required constants and data structures have been defined in `app.h` and are already included. Store the returned queue handles in the corresponding variables `x###QueueHandle` declared at the beginning of the file and check each queue handler for errors during creation by verifying that it is a valid handler.

Recap 1: Creating Queues in FreeRTOS

For creating a new FreeRTOS queue, use the `xQueueCreate(...)` function:

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,  // Max number of queue items
                            UBaseType_t uxItemSize      // Size of an item in bytes
                          );
```

This function returns a valid queue handle if the queue was created. If there is not enough memory available to allocate the queue, a `NULL` value is returned. The function `sizeof()` can be used to determine the size of an arbitrary data structure.

Solution for Task 1.2:
A sample implementation for Task 1.2 is shown in Snippet 1. Notice that we compare each handle against `NULL` to verify that it corresponds to a valid task.

```
BaseType_t result = pdPASS;

// Create Sensor data queue
xSensorQueueHandle = xQueueCreate(appSENSOR_QUEUE_LENGTH, sizeof(sensor_data_t));
if (xSensorQueueHandle == NULL)
{
  uart_println("Error creating Sensor data queue.");
  return 0;
}

// Create Print data queue
xPrintQueueHandle = xQueueCreate(appPRINT_QUEUE_LENGTH, sizeof(attitude_data_t));
if (xPrintQueueHandle == NULL)
{
  uart_println("Error creating Print data queue.");
  return 0;
}

// Create LED data queue
xLedQueueHandle = xQueueCreate(appLED_QUEUE_LENGTH, sizeof(attitude_data_t));
if (xPrintQueueHandle == NULL)
{
  uart_println("Error creating LED data queue.");
  return 0;
}

```

Snippet 1: Sample solution code for Task 1.2.

Task 1.3: Create Existing Tasks

Now that the queues and their handles have been defined, you can create the tasks that make use of these queues. Create the *Sensor*, *LED* and *Print* tasks for which we already provide you with the necessary functions that implement the logic in `app.c` (`prv###Task`). The *Processing* task is skipped for now and is going to be created as part of Task 2.
For each task, first create the required task argument variables (of type `###_task_param_t`) before you create the actual task. All required data structures and constants are already available from the `app.h` header; check their detailed documentation in the source code. Once again, do not forget to check for errors after the creation of each task.

Recap 2: Creating Tasks in FreeRTOS

For creating a FreeRTOS task, use the `xTaskCreate(...)` function:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,  // Pointer to the task function
                        const char * const pcName,  // Task name
                        uint16_t usStackDepth,      // Stack size to allocate
                        void *pvParameters,         // Task specific parameters
                        UBaseType_t uxPriority,     // Priority of the task
                        TaskHandle_t *pxCreatedTask // Handle to the task
                      );
```

This function returns `pdPASS` if the task was successfully created and stores the task handle at the memory location specified by `pxCreatedTask`. In case of insufficient memory to create a task, `errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY` is returned.

Solution for Task 1.3:
A sample implementation for Task 1.3 is shown in Snippet 2.

```
// Create the task used to read the motion sensors
sensor_task_param_t sensor_task_param = {
  .outDataQueue = xSensorQueueHandle,
};
result = xTaskCreate(prvSensorTask,
                     "Sensor",
                     appDEFAULT_TASK_STACK_SIZE,
                     &sensor_task_param,
                     appSENSOR_TASK_PRIORITY,
                     &xSensorTaskHandle);
if (result != pdPASS)
{
  uart_println("Error creating Sensor task.");
  return 0;
}

// Create the task used to send data via UART
print_task_param_t print_task_param = {
  .inDataQueue = xPrintQueueHandle,
};
result = xTaskCreate(prvPrintTask,
                     "Print",
                     appPRINT_TASK_STACK_SIZE,
                     &print_task_param,
                     appPRINT_TASK_PRIORITY,
                     &xPrintTaskHandle);
if (result != pdPASS)
{
  uart_println("Error creating Print task.");
  return 0;
}

// Create the task used to indicate the current angle using the LED
led_task_param_t led_task_param = {
  .inDataQueue = xLedQueueHandle,
};
result = xTaskCreate(prvLedTask,
                     "LED",
                     appDEFAULT_TASK_STACK_SIZE,
                     &led_task_param,
                     appLED_TASK_PRIORITY,
                     &xLedTaskHandle);
if (result != pdPASS)
{
  uart_println("Error creating LED task.");
  return 0;
}

```

Snippet 2: Sample solution code for Task 1.3.

Task 1.4: Testing Start Sequence of Tasks

Compile and run your application to test your implementation of the previous two tasks. Before resuming the program execution at `main`, open a serial port terminal to display the output written to the UART. Use a baudrate of 115200 and the default settings of 8 data bits, one stop bit and no parity or flow control.

Resume the task execution and observe the UART output. Each task writes a message to the UART when it is started for the first time. In which order are the tasks being started? Does this match your expectations from Task 1.1?

Solution for Task 1.4:
The tasks are starting in the expected order of decreasing priority: *Print*, *LED*, *Sensor*. There is no *Processing* task output, as it is not created at this point.

Task 2: Adding a new FreeRTOS Task

In this task, you are going to implement the missing *Processing* task step-by-step and integrate it into the application. This will lead to the (almost) feature-complete application described in the introduction, missing only the button functionality.

Task 2.1: Create a new task function

As you have learned in the last lab, the logic of FreeRTOS tasks is implemented in a function that takes a single `void` pointer argument and has no return value. To start with the new FreeRTOS task, create an empty function named `prvProcessingTask` following exactly this prototype. Implement this function in the beginning of the file `app.c`, right below the comment summarizing this and subsequent steps of this task.

Solution for Task 2.1:
The complete implementation of the `prvProcessingTask` function is given in Snippet 3 at the end of this task.

Task 2.2: Processing task arguments

Next, you will read the necessary task parameters passed as arguments.

(a) What information needs to be passed as argument to the *Processing* task so that it can implement the described functionality? *Hint:* Check the application description in the introduction again as well as the overview in Figure 1.

(b) The necessary data structure that holds the processing task arguments is already defined in the `app.h` header. Add the necessary code at the beginning of the `prvProcessingTask` function to copy the parameters to local variables. Recall the pointer type conversion from Lab 3.

Solution for Task 2.2:

(a) The task needs to know the input *Sensor* queue to read the sensor data from and the two output *Print* and *LED* queues to write the processed data to. The handles for these queues are passed as an argument using the `processing_task_param_t` structure defined in the `app.h` header file.

(b) The complete implementation of the `prvProcessingTask` function is given in Snippet 3 at the end of this task.

Task 2.3: Implementing the Task Logic

Now you are going to add the actual task logic described in the introduction:

(a) After reading the task argument and before the task's main loop, print the message *"Processing task starting..."* over the UART to indicate that the task is starting.

(b) Create an infinite main loop to continuously wait for new sensor data, process it to calculate the LaunchPad's attitude, and forward the result to the *Print* and *LED* queues if it has changed significantly. To do so, implement the following procedure inside the loop:

(i) Perform a blocking read on the *Sensor* queue to wait for new sensor data.

(ii) Compensate for offsets in the acceleration sensor data by subtracting the calibration values stored in the `calibration_acc` variable. This global variable is already defined and initialized at the beginning of `app.c`. You do not need to worry about updating the calibration value at this point (this is part of Task 3).

(iii) Allocate two variables for the tilt angles θ_x and θ_y that define the LaunchPad's attitude. For now, initialize them with a constant value, for example $\theta_x = -5.0$ and $\theta_y = +15.0$ degrees. Later in this task, you will derive the formula to calculate these values from the sensor data.

(iv) Store the calculated θ_x, θ_y and the timestamp of the original sensor readings in a combined attitude structure of type `attitude_data_t`.

(v) Forward the attitude to the *Print* and *LED* tasks if and only if it has changed significantly from the last forwarded value. A change is significant if at least one of the angles has changed more than the threshold defined as `appANGLE_SIGNIFICANT_CHANGE` macro (value in degrees).

At this point, the application can be tested in Task 2.5 without considering actual sensor data. However, the constant attitude values are printed at most once, since they never change. Switching back to this constant initialization of θ_x and θ_y might help in case you need to debug your application in Task 2.5.

(c) Lastly, replace the constant initialization of the LaunchPad's attitude, i.e. the tilt angles θ_x and θ_y, with the actual calculation based on the sensor data and store θ_x and θ_y in degrees. Recall that:

- The definition of θ_x and θ_y are illustrated in Figure 2.
- The sensor outputs the *resulting* acceleration vector, expressed in the sensor axes $(\vec{x_s}, \vec{y_s}, \vec{z_s})$. In other words, when the Launchpad is stationary, the sensor measures $-\vec{g}$, where $\vec{g}$ is the acceleration of the gravity (we assume the accelerometer measures only earth' gravitational pull).

Hints:

- The `math.h`[2] header is included and a `PI` constant defined – you can make use of trigonometric functions right away.
- How do you compute the norm of $\vec{g}$ from the sensor data?
- How do you compute the angle between a vector and a plane?

Solution for Task 2.3:
A sample implementation of the full *Processing* task is given in Snippet 3.

Task 2.4: Create and start the new task

Go back to the `main_freertos.c` file and create the *Processing* task you just implemented at the end of the `main` function. The necessary data structures and constants are defined in the `app.h` header.

Solution for Task 2.4:
The code to create the *Processing* task is shown in Snippet 4.

[2] Details about functions declared in the `math.h` header are found at http://www.cplusplus.com/reference/cmath/.

```c
void prvProcessingTask(void* pvParameters)
{
  // Data structures
  sensor_data_t sensor_data;
  attitude_data_t attitude_data;
  processing_task_param_t param;

  // Get the task parameters
  param = *((processing_task_param_t*)pvParameters);

  // Low level init
  // --- not required ---

  // Print task start
  uart_println("Processing task starting...");

  // Comparison variables (invalid angle value to update on first run)
  attitude_data_t attitude_last = {
    .theta_x = -1000, .theta_y = -1000, .time = 0,
  };

  // Enter main sensing loop
  while (1)
  {
    // Wait for new sensor data to be sent
    xQueueReceive(param.inDataQueue, (void*)&sensor_data, portMAX_DELAY);

    // Offset compensation
    float acc_x = (float)(sensor_data.acc_x - calibration_acc.x);
    float acc_y = (float)(sensor_data.acc_y - calibration_acc.y);
    float acc_z = (float)(sensor_data.acc_z - calibration_acc.z);

    // Basic angle calculation for gravity vector
    float norm = sqrtf(acc_x*acc_x + acc_y*acc_y + acc_z*acc_z);
    float theta_x = asinf(acc_x/norm); // in radian
    float theta_y = asinf(acc_y/norm); // in radian

    attitude_data.theta_x = 180 * theta_x / PI; // in degree
    attitude_data.theta_y = 180 * theta_y / PI; // in degree
    attitude_data.time = sensor_data.time;

    // Check if angles changed more than specified value
    if (abs(attitude_data.theta_x - attitude_last.theta_x) >= appANGLE_SIGNIFICANT_CHANGE ||
        abs(attitude_data.theta_y - attitude_last.theta_y) >= appANGLE_SIGNIFICANT_CHANGE)
    {
      // Send updated value
      xQueueSendToBack(param.outLedQueue, (void*)&attitude_data, 0);
      xQueueSendToBack(param.outPrintQueue, (void*)&attitude_data, 0);
      attitude_last = attitude_data;
    }
  }

  // Delete task before reaching end
  vTaskDelete(NULL);
}
```

Snippet 3: Sample implementation of the *Processing* task logic.

Task 2.5: Testing the Full Application

Compile and run your code to test your application. Again, use the terminal to check the UART output: you should now see all tasks starting in decreasing order of priority, now also including the *Processing* task. Following the task initialization, new attitude data is displayed whenever the board's attitude changed more than 5°. The resulting output should be similar to the sample shown in Snippet 5. If you tilt the board beyond 30° in either direction, the LED2 color should change from green to red.

What is the reason that all tasks, including the lowest priority sensor task, are executed before the first attitude value is printed to UART?

```
// Create the task used to process sensor data
processing_task_param_t processing_task_param = {
  .inDataQueue = xSensorQueueHandle,
  .outLedQueue = xLedQueueHandle,
  .outPrintQueue = xPrintQueueHandle,
};
result = xTaskCreate(prvProcessingTask,
                     "Processing",
                     appDEFAULT_TASK_STACK_SIZE,
                     &processing_task_param,
                     appPROCESSING_TASK_PRIORITY,
                     &xProcessingTaskHandle);
if (result != pdPASS)
{
  uart_println("Error creating Processing task.");
  return 0;
}

```

Snippet 4: Sample solution code for Task 2.4.

```
Queues and tasks initialized.
Print task starting...
LED task starting...
Processing task starting...
Sensor task starting...
time:     3546  --  theta_x:   2002   theta_y:    492
time:    18674  --  theta_x:  -3400   theta_y:   1465
```

Snippet 5: Expected UART output of the application. The time value is the accelerometer internal timestamp of when the sensor values are read, the angles θ_x and θ_y are displayed in millidegrees ($1 \times 10^{-3\circ}$).

Solution for Task 2.5:
The tasks are started in decreasing priority order. Because all queues are empty at the beginning, they block after initialization, waiting for new data to read from their input queue. At this point FreeRTOS switches to the next lower priority tasks, which again blocks after initialization, waiting for data to arrive in the incoming data queue. Only the lowest priority *Sensor* does not block and starts reading the first sensor data. As soon as it writes a new sensor value to the sensor data queue, it is pre-empted by the *Processing* task that was in a blocked state and was not able to resume before, and similarly with the *LED* and *Print* tasks as soon as the *Processing* task adds new results to the corresponding queues.

Task 3: Interrupt Handling in FreeRTOS (Optional)

Finally, you will integrate two button interrupts to add on-demand sensor calibration. For this, you can use interrupts to a) store the current sensor readings as offset values for subsequent calculations (see Task 2.3) using the left button (S1) and b) reset the offset values to zero with the right button (S2).

Interrupts in the context of FreeRTOS are configured and handled exactly the same way as in a bare-metal implementation (covered in Lab 2). Some restrictions apply to FreeRTOS API calls from within the interrupt service routine, but in today's lab you will not make use of them inside the interrupt service routine. For more details, we refer interested readers to Chapter 6 of the FreeRTOS tutorial book[3].

Task 3.1: Configure and Enable Button Interrupts

Open `gpio.c`, where you find the GPIO initialization function. Add the necessary code at the end of the

[3]The FreeRTOS tutorial book is available at http://www.freertos.org/Documentation/RTOS_book.html.

gpio_init function to setup and enable the interrupts for the buttons S1 and S2. All required constants are already defined in the gpio.h header.

Recap 3: Interrupt Configuration

Remember to clear the corresponding interrupt flag before enabling an interrupt. You must not only enable the interrupt in the GPIO peripheral module, but also enable interrupts for the specific module using the Interrupt_enableInterrupt(...) DriverLib function. In addition, there are dedicated processor instructions for enabling and disabling interrupts globally, called inside the Interrupt_enableMaster() and Interrupt_disableMaster() DriverLib functions.

Solution for Task 3.1:
The code to setup the interrupts is shown in Snippet 6.

```
// Interrupt setup
GPIO_interruptEdgeSelect(BUTTON_S1_PORT, BUTTON_S1_PIN, GPIO_HIGH_TO_LOW_TRANSITION);
GPIO_clearInterruptFlag(BUTTON_S1_PORT, BUTTON_S1_PIN);
GPIO_enableInterrupt(BUTTON_S1_PORT, BUTTON_S1_PIN);

GPIO_interruptEdgeSelect(BUTTON_S2_PORT, BUTTON_S2_PIN, GPIO_HIGH_TO_LOW_TRANSITION);
GPIO_clearInterruptFlag(BUTTON_S2_PORT, BUTTON_S2_PIN);
GPIO_enableInterrupt(BUTTON_S2_PORT, BUTTON_S2_PIN);

Interrupt_enableInterrupt(INT_PORT1);

```

Snippet 6: Sample solution code for Task 3.1.

Task 3.2: Handling the Button Interrupts

The interrupt service routine for GPIO port 1 already exists in the interrupt.c source file. Add the necessary code to check for the button S1 and S2 interrupts and handle them. You should store new calibration offsets when button S1 is pressed, and reset the offsets on a press to button S2. Use the vAppStoreCalibration() and vAppResetCalibration() inline functions implemented in app.h. All required functions and constants are defined in the app.h and gpio.h headers; make sure to include them at the beginning of the interrupt.c file.

Recap 4: Interrupt Service Routines

Typically, interrupt service routines are shared with multiple interrupt sources (in this task GPIO pins of the same port). Check the interrupt flags of the corresponding peripheral to find out which source (e.g. GPIO pin) triggered the interrupt. Do not forget to clear the corresponding interrupt flag after handling an interrupt.

Solution for Task 3.2:
The code of the interrupt service routine (ISR) is shown in Snippet 7. Notice that evaluating both cases unconditionally (no else statement) ensures that both flags are cleared even if both buttons are pressed simultaneously.

```c
// MSP432F401R Launchpad button S1 (left)
if (status & BUTTON_S1_PIN)
{
  vAppStoreCalibration();
  // Clear flag of processed interrupt
  GPIO_clearInterruptFlag(BUTTON_S1_PORT, BUTTON_S1_PIN);
}
// MSP432F401R Launchpad button S2 (right)
if (status & BUTTON_S2_PIN)
{
  vAppResetCalibration();
  // Clear flag of processed interrupt
  GPIO_clearInterruptFlag(BUTTON_S2_PORT, BUTTON_S2_PIN);
}

```

Snippet 7: Sample solution code for Task 3.2.

Task 3.3: Test Offset Calibration

Compile and run the application to test your interrupts. Follow these steps to test the calibration functionality:

(a) Tilt the board more than 30° and hold the board in that position. The LED2 should have switched to red at this point.

(b) Press button S1 to store the current sensor readings as a new calibration offset. The LED2 should turn to green and the UART output jump close to zero, as the current accelerometer readings are used as reference.

(c) Now move the board back to the default position by putting it back on the table. With the new calibration, the LED should now have switched to red again and the UART output values should differ significantly from zero.

(d) Press button S2 to reset the calibration offsets to zero. LED2 should turn back to green and UART output jump to a value reasonably close to zero.

If buttons are used to calibrate the offset when the board is in a leveled position, the accuracy of the calculated angle can actually be improved.

With this task, you have successfully completed the labs of the Embedded Systems lecture. Now you know all the basics to realize your own application idea using the MSP432 LaunchPad and FreeRTOS. If you want to dig deeper into the topic, we recommend to have a look at the code of the *Sensor* task used in today's lab, and check the extensive collection of demo projects for the MSP432 LaunchPad in the TI Resource Explorer inside *Code Composer Studio*. *Happy Coding!*

Solution for Task 3.3:
There is no specific solution for this task. Make sure that the application behavior matches the description in the task and debug the interrupt setup and service routines in case you suspect that there are some errors. A fully working version of the project is provided in `lab4_solution.zip`.