ECSE444 lab4

Simon Li
dept. electrical eng
McGill University
Montreal, Canada
xi.yang.li@mail.mcgill.ca

Emily Li
dept. electrical eng
McGill University
Montreal, Canada
yue.l.li@mail.mcgill.ca

I. Abstract

This report presents an in-depth analysis and implementation of embedded Real-Time Operating Systems (RTOS) using UART (Universal Asynchronous Receiver/Transmitter) and I2C (Inter-Integrated Circuit) peripherals on a B-L4S5I-IOT01A board. The primary objective of this lab is to gain familiarity with essential communication interfaces in embedded systems and demonstrate the advantages of using an RTOS for efficient task management, preemptive scheduling, and resource allocation. The lab utilizes various sensors, including the HTS221 humidity sensor, LIS3MDL magnetometer, LSM6DSL gyroscope, and LPS22HB pressure sensor, to facilitate hands-on experience interfacing with these sensors and managing their output. A Board Support Package (BSP) is employed to simplify the integration and coordination of hardware components within the embedded system. The lab demonstrates the implementation of various tasks to read sensor data, monitor button status, and display sensor values using the FreeRTOS environment, offering a practical understanding of user interaction and control in embedded systems.

II. Introduction

This report focuses on embedded Real-Time Operating Systems (RTOS), which are specifically designed to handle time-sensitive tasks and applications in embedded systems. An RTOS offers several advantages, such as efficient task management, preemptive scheduling, and resource allocation. These systems enable developers to control how often different parts of a program execute, ensuring that critical tasks are completed in a timely manner while less critical tasks are preempted when necessary. Consequently, an RTOS significantly simplifies the development process for complex and time-sensitive embedded systems.

The primary objective of this lab is to familiarize ourselves with UART (Universal Asynchronous Receiver/Transmitter) and I2C (Inter-Integrated Circuit) peripherals, which are essential communication interfaces in embedded systems. The lab aims to demonstrate how to coordinate OS tasks to read I2C sensor values from various sensors and manage their output. Additionally, we will learn how to control the display of specific sensor values using a push-button, offering a practical approach to user interaction and control in embedded systems.

III. Setup and Configuration

The lab setup consists of the B-L4S5I-IOT01A board, which serves as the foundation for the project. The board is equipped with a variety of sensors, including the HTS221 humidity sensor, LIS3MDL magnetometer, LSM6DSL gyroscope, and LPS22HB pressure sensor. To streamline the process of interfacing with these sensors and managing their respective functionalities, a Board Support Package (BSP) is utilized. The BSP provides an abstraction layer for the hardware, simplifying the integration and coordination of various components within the embedded system.

Furthermore, we configured two GPIO pins to enhance user interaction and deliver real-time feedback. GPIO pin PA5 was established as a push-pull output connected to an LED, acting as a visual representation of the program's current state. In contrast, GPIO pin PC13 was set up as an interrupt input for the initial phase and as a conventional GPIO input for the subsequent phase, permitting users to manipulate the program's state dynamically.

Referring to the user manual, we ascertained that the I2C2 interface was utilized for the internal sensors under examination. We enabled this interface, maintaining all default parameters. Simultaneously, we consulted the same user manual to configure USART1, preserving the default settings while manually assigning PB7 and PB6 as RX and TX, respectively.

IV. Lab Procedure

In this section, we present the lab procedure for configuring UART and I2C peripherals and setting up CMSIS RTOS and FreeRTOS. The STM32L4S5I-IOT01 board is utilized, and various sensors are interfaced, including HTS221 for humidity, LIS3MDL for magnetometer, LSM6DSL for gyroscope, and LPS22HB for pressure.

The necessary header files are included for the sensors and other peripherals. The TIM_HandleTypeDef and UART_HandleTypeDef structures are declared to configure the TIM2 timer and UART1 communication, respectively.

The PUTCHAR_PROTOTYPE macro is defined to facilitate the transmission of characters through UART1. The HAL_UART_Transmit function is utilized to transmit the characters, with a maximum delay set for the transmission.

Private user code is defined, including a string array 'str' of length 100, a volatile integer 'currentSensor', and several other

```
/* Private includes ----------
/* USER CODE BEGIN Includes */
#include "stm32l4s5i_iot01_tsensor.h"
#include "stm32l4s5i_iot01.h"
#include "stm32l4s5i_iot01_accelero.h"
#include "stm32l4s5i_iot01_gyro.h"
#include "stm32l4s5i_iot01_hsensor.h"
#include "stm32l4s5i_iot01_magneto.h"
#include "stm32l4s5i_iot01_psensor.h"
#include <stdio.h>
```

Fig. 1. Include Statements

```
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
static void MX_USART1_UART_Init(void);
void StartDefaultTask(void const * argument);
void StartTask02(void const * argument);
void StartTask03(void const * argument);
void StartTask04(void const * argument);

/* USER CODE BEGIN PFP */
#ifdef __GNUC__
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif

PUTCHAR_PROTOTYPE
{
  HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, HAL_MAX_DELAY);
  return ch;
}
```

Fig. 2. Function Prototypes

variables for storing sensor readings. The ReadSensorValues function is implemented to read the values from different sensors, and the PrintSensorValues function is used to display the obtained sensor values.

The main function initializes the MCU and configures the system clock. The GPIO, TIM2 timer, and UART1 communication are initialized. The sensors are initialized using BSP_HSENSOR_Init(), BSP_MAGNETO_Init(), BSP_GYRO_Init(), and BSP_PSENSOR_Init() functions. The default task, along with three additional tasks, are defined and created using the osThreadDef and osThreadCreate functions, followed by starting the FreeRTOS kernel.

A. Part 1

The HAL_GPIO_EXTI_Callback function is a callback function that is triggered whenever an external interrupt occurs. The function checks if the interrupt occurred due to the button press by comparing the GPIO_Pin argument with mybutton_Pin. If the condition is met, the LED is toggled using the HAL_GPIO_TogglePin function. This function takes two arguments: the GPIO port and the GPIO pin. Here, myled_GPIO_Port and myled_Pin are used to specify the LED. After toggling the LED, the currentSensor variable is incremented, which is used to switch between the different sensors. To cycle through the four available sensors, the currentSensor value is reset to 0 when it reaches 4 using the modulo operation.

```
//for part 1
void HAL_GPIO_EXTI_Callback (uint16_t GPIO_Pin){
    if(GPIO_Pin == mybutton_Pin){
        HAL_GPIO_TogglePin(myled_GPIO_Port, myled_Pin); // Toggle LED
        currentSensor++;
    }
    if(currentSensor%4==0) currentSensor = 0;
}
```

Fig. 3. Button Interrupts

The ReadSensorValues function reads the sensor data depending on the input sensorIndex. It uses a switch statement to handle different cases corresponding to each sensor:

Case 0: HTS221 humidity sensor. The function BSP_HSENSOR_ReadHumidity is called to read the humidity value, which is then stored in the humidity variable. The humidity value is typecast to an integer and stored in the variable h.

Case 1: LIS3MDL magnetometer. The function BSP_MAGNETO_GetXYZ is called to get the magnetometer readings for the X, Y, and Z axes. The readings are stored in the magnetometer array.

Case 2: LSM6DSL gyroscope. The function BSP_GYRO_GetXYZ is called to get the gyroscope readings for the X, Y, and Z axes. The readings are stored in the gyroscope array.

Case 3: LPS22HB pressure sensor. The function BSP_PSENSOR_ReadPressure is called to read the pressure value, which is then stored in the pressure variable.

```
void ReadSensorValues(int sensorIndex)
{
//  printf("hello");
  switch (sensorIndex)
  {
    case 0: // Display humidity from HTS221
      humidity = BSP_HSENSOR_ReadHumidity();
      h = (int) humidity;
      break;

    case 1: // Display magnetic field (X-axis) from LIS3MDL
      BSP_MAGNETO_GetXYZ(magnetometer);
      break;

    case 2: // Display gyroscope (X-axis) from LSM6DSL
      BSP_GYRO_GetXYZ(gyroscope);
      break;

    case 3: // Display pressure from LPS22HB
      pressure = BSP_PSENSOR_ReadPressure();
      break;

    default:
      break;
  }
}
```

Fig. 4. Sensor Reading Function

The PrintSensorValues function displays the sensor readings based on the input sensorIndex. Like the ReadSensorValues function, it uses a switch statement to handle different cases for each sensor. The printf function is used to display the values on the terminal.

In the infinite loop of the main function, the sensor values are read and displayed at a 10 Hz sampling rate using a 100 ms delay between the reads.

```
132 void PrintSensorValues(int sensorIndex)
133 {
134   switch (sensorIndex)
135   {
136     case 0: // Display humidity from HTS221
137
138       printf("Humidity: %d percent\r\n", h);
139       break;
140
141     case 1: // Display magnetic field (X-axis) from LIS3MDL
142       printf("Magnetometer X: %d mG\r\n", (int)magnetometer[0]);
143       break;
144
145     case 2: // Display gyroscope (X-axis) from LSM6DSL
146       printf("Gyroscope X: %d dps\r\n", (int)gyroscope[0]);
147       break;
148
149     case 3: // Display pressure from LPS22HB
150       printf("Pressure: %d hPa\r\n", (int)pressure);
151       break;
152
153     default:
154       break;
155   }
156 }
```

Fig. 5. Function to Display Sensor Values

```
266   while (1)
267   {
268     /* USER CODE END WHILE */
269
270     /* USER CODE BEGIN 3 */
271
272       // for part 1
273       ReadSensorValues(currentSensor);
274       PrintSensorValues(currentSensor);
275       HAL_Delay(100); // 10 Hz sampling rate (100 ms delay between reads)
276   }
277   /* USER CODE END 3 */
278 }
```

Fig. 6. Infinite Loop in Main

B. Part 2

The key advantage of an embedded operating system is that it makes it easy to more carefully control when different parts of our program execute. For instance, perhaps we want to sample one sensor at 1 Hz, another at 10 Hz, and another at 100 Hz. Maybe we only want to check that a button has been pressed every 500 ms. And perhaps we want to log data (to display or otherwise take action) any time a new sample is taken. Implementing this with a single main(), even with timers and interrupts, may make it difficult to meet performance requirements. Therefore, in this part, we implement the same functionalities as in part one using the FreeRTOS environment.

For this part, four task handles are created to manage different tasks in the FreeRTOS environment. To do this, we use the CMSIS-RTOS API, an abstraction layer for the underlying FreeRTOS, to define and create threads.

Default Task Thread: The default task thread is defined using the osThreadDef macro. It takes four arguments: the thread name (defaultTask), the function that the thread will execute (StartDefaultTask), the thread's priority (osPriorityNormal), and the stack size (256 words). Following the definition, the thread is created using the osThreadCreate function, which takes two arguments: the thread definition and a pointer to arguments passed to the thread function. In this case, no arguments are provided (NULL).

Task 02 Thread: Similar to the default task, the myTask02 thread is defined and created with the corresponding function (StartTask02) and priority level (osPriorityIdle). A lower priority level indicates that this thread will only execute when higher priority threads are not active.

Task 03 Thread: The myTask03 thread follows the same structure, with its function (StartTask03) and priority level (osPriorityIdle) specified during definition and creation.

Once all the threads are defined and created, the RTOS scheduler is started using the osKernelStart function. This function initializes and begins the scheduling of the threads based on their priorities and availability. The scheduler ensures that threads are executed concurrently, allowing for efficient multitasking within the embedded system. Note that we have increased the stack size for all the threads to 256 words. This is to prevent the hard fault triggers when attempting to display float values on the terminal.

```
241   osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 256);
242   defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);
243
244   /* definition and creation of myTask02 */
245   osThreadDef(myTask02, StartTask02, osPriorityIdle, 0, 256);
246   myTask02Handle = osThreadCreate(osThread(myTask02), NULL);
247
248   /* definition and creation of myTask03 */
249   osThreadDef(myTask03, StartTask03, osPriorityIdle, 0, 256);
250   myTask03Handle = osThreadCreate(osThread(myTask03), NULL);
251
252   /* definition and creation of myTask04 */
253 //  osThreadDef(myTask04, StartTask04, osPriorityNormal, 0, 256);
254 //  myTask04Handle = osThreadCreate(osThread(myTask04), NULL);
255
256   /* USER CODE BEGIN RTOS_THREADS */
257 //  /* add threads, ... */
258   /* USER CODE END RTOS_THREADS */
259
260   /* Start scheduler */
261   osKernelStart();
```

Fig. 7. Threads Creation

The StartDefaultTask function is implemented to read sensor values in an infinite loop, with a delay of 100 ms. The

```
469 void StartDefaultTask(void const * argument)
470 {
471   /* USER CODE BEGIN 5 */
472   /* Infinite loop */
473   for(;;)
474   {
475     osDelay(100);
476     ReadSensorValues(currentSensor);
477     PrintSensorValues(currentSensor);
478   }
479   /* USER CODE END 5 */
480 }
```

Fig. 8. Default Task

StartTask02 function is implemented as an infinite loop to monitor the button status. When the button is pressed, the LED is toggled, and the currentSensor variable is updated to switch between sensors. Finally, the StartTask03 function is implemented as an infinite loop to display sensor values with a delay of 100 ms.

V. Conclusion

In conclusion, this lab provided a comprehensive understanding of the essential communication interfaces

```
489 void StartTask02(void const * argument)
490 {
491   /* USER CODE BEGIN StartTask02 */
492   /* Infinite loop */
493   for(;;)
494   {
495     osDelay(2);
496     status = HAL_GPIO_ReadPin(mybutton_GPIO_Port, mybutton_Pin);
497         if(status != prev){ // button was pressed or released
498             if(status == GPIO_PIN_RESET){ // button was pressed SET=1, RESET=0
499                 HAL_GPIO_TogglePin(myled_GPIO_Port, myled_Pin); // Toggle LED
500                 currentSensor = (currentSensor+1)%4;
501             }
502             prev = status; // update previous status
503         }
504   }
505   /* USER CODE END StartTask02 */
506 }
```

Fig. 9. Task 2

```
515 void StartTask03(void const * argument)
516 {
517   /* USER CODE BEGIN StartTask03 */
518   /* Infinite loop */
519   for(;;)
520   {
521     osDelay(100);
522     PrintSensorValues(currentSensor);
523   }
524   /* USER CODE END StartTask03 */
525 }
```

Fig. 10. Task 3

in embedded systems, namely UART and I2C peripherals. The B-L4S5I-IOT01A board, equipped with various sensors, enabled us to acquire hands-on experience interfacing with these sensors and managing their output. We familiarized ourselves with the Board Support Package (BSP), which simplified the integration and coordination of hardware components in the embedded system.

Furthermore, this lab demonstrated the advantages of using an RTOS, such as efficient task management, preemptive scheduling, and resource allocation. We learned how to control the execution of different parts of a program using FreeRTOS, ensuring that critical tasks were completed in a timely manner while less critical tasks were preempted when necessary. By implementing various tasks to read sensor data, monitor button status, and display sensor values, we developed a practical understanding of user interaction and control in embedded systems.

All in all, the knowledge and skills acquired in this lab will prove invaluable for the development of complex and time-sensitive embedded systems in the future.

REFERENCES

[1] “Lab 3: Lab 4: I2 C Peripherals and OS” Winter 2023.
[2] ”Discovery kit for IoT node, multi-channel communication with STM32L4+ Series” STM32 Board User Manual
[3] B-L4S5I-IOT01 BSP User Manual
[4] https://forum.digikey.com/t/easily-use-printf-on-stm32/20157. Accessed 2023