Laboratory 4: I2C Peripherals and OS

Karim Elgammal
Computer Engineering, McGill University
260920556

Rafid Saif
Electrical Engineering, McGill University
260903802

***Abstract*—This laboratory serves as an introduction to the use of I2C Sensors available on the STM32L4S5 Discovery Kit and relaying the observed data through UART. The lab introduces pin configuration based on PCB schematics of the board and subsequently using the Board Support Packages for the various sensor peripherals. The final application developed is able to toggle between various sensor readings, and this is implemented both with and without FreeRTOS.**

***Index Terms*—I2C, UART, sensors, FreeRTOS, interrupts, STM32L4S5, STMCubeMX, STMCubeIDE**

I. Introduction

One of the core features of embedded systems is reading and transmitting sensor data. The STM32L4S5 board used in this laboratory is equipped with a number of sensors such as temperature, acceleration, magnetic field and pressure. This is explored in the first part of the lab. Another important aspect is scheduling and multitasking, which is particularly important when strict timing requirements must be met. This is implemented on the board using FreeRTOS to create multiple tasks in the final part of the lab.

II. Methodology and Analysis

A. Part 1: UART and I2C Peripherals

The first part of the lab involves setting up various sensors connected on the I2C bus of the development kit as well as the UART connected to the ST-Link in order to send messages through a virtual COM port. The final application toggles between different sensor displays when the pushbutton is pressed.

To start, we set up the I2C2 peripheral on CubeMX, making sure it matches the pins on the schematic, PB10 for SCL and PB11 for SDA. Following code generation, we added in the Board Support Packages provided for all the sensors. From the source files, we found the corresponding functions to initialize and read values from the sensors, such as BSP_TSENSOR_Init() and BSP_TSENSOR_ReadTemp() for the temperature sensor.

Next, we set up the USART connected to the ST-Link. As the USART1 TX and RX pins have a few possible pin configurations, we once again consulted the schematic to find that the required pins were PB6 and PB7. After enabling these pins for USART and generating code, we simply initialized a char array and used the printf() function to set the string using a sensor value. We then used the library function HAL_UART_Transmit() to transmit the string on the serial port. This is seen in figure 1.

```
while (1)
{
  if (sensor_state == TEMP_STATE){
    char temp_str[20];
    temp = BSP_TSENSOR_ReadTemp();
    sprintf(temp_str, "\r\nTemperature: %.2f", temp);
    HAL_UART_Transmit(&huart1,temp_str,sizeof(temp_str),10);// Sending in normal mode
  } else if (sensor_state == ACCEL_STATE){
    char accel_str[40];
    BSP_ACCELERO_AccGetXYZ(accel);
    sprintf(accel_str, "\r\nAcceleration x:%.4d, y:%.4d, z:%.4d", accel[0], accel[1], accel[2]);
    HAL_UART_Transmit(&huart1,accel_str,sizeof(accel_str),10);// Sending in normal mode
  }else if(sensor_state == MAGNETO_STATE){
    char magneto_str[40];
    BSP_MAGNETO_GetXYZ(magnet);
    sprintf(magneto_str, "\r\nMagnometer x:%.4d, y:%.4d, z:%.4d", magnet[0], magnet[1], magnet[2]);
    HAL_UART_Transmit(&huart1,magneto_str,sizeof(magneto_str),10);// Sending in normal mode
  }else if(sensor_state == PRESS_STATE){
    char press_str[20];
    pressure = BSP_PSENSOR_ReadPressure();
    sprintf(press_str, "\r\nPressure: %.2f", pressure);
    HAL_UART_Transmit(&huart1,press_str,sizeof(press_str),10);// Sending in normal mode

  }

  HAL_Delay(100);
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
```

Fig. 1. Updating Sensor Readings

Finally, as in the previous lab, we enabled pushbutton interrupts in order to toggle between sensors. For this, we defined the enum sensor_states for readability. Upon pressing the pushbutton, the button simply changes the sensor state to the next, allowing us to read this and adjust the string printed based on which sensor state we are in.

```
enum sensor_states{TEMP_STATE, ACCEL_STATE, MAGNETO_STATE, PRESS_STATE};
enum sensor_states sensor_state = TEMP_STATE;
const uint8_t total_states = 4;
```

Fig. 2. Sensor State Enumerated

B. Part 2: CMSIS RTOS and FreeRTOS

The second part of the lab is concerned with running the same functional application discussed in part 1; however, while utilizing the FreeRTOS OS to split the application into three threads or 'tasks'. The three tasks can be described as follows; a task that determines when the button has been pressed, then changes the sensor that is being read. output data from the next sensor in the sequence, a task that reads sensor data, and finally, a task that transmits this data to the terminal using the virtual COM port. And so, in order to achieve this, we configured the FreeRTOS component within the middleware section of the STM32Cube application and set up its interface mode to CMSIS_V1. While all other settings were kept as default, we added the three tasks and a binary semaphore object to the configuration, to allow STM32Cube to generate their instances in our code automatically. Further, it is noted that FreeRTOS and HAL_Delay both use SysTick for timing,

causing priority conflicts. To resolve this, the system core's time base source was switched to TIM6.

Once we generated the code, we set up the Callback function of the pushbutton that we used in the preceding part as seen in figure 3. However, this time around, we implemented the callback with one line of code after the verification of the source of the interrupt; osSemaphoreRelease(myBinarySem01Handle);. This call releases the binary semaphore (synchronization primitive) which was created earlier in main(). Once it is released, the highest priority task containing a osSemaphoreWait() function will take this semaphore and perform the task's specified function. And so, in our example in figure 3; we can see that the ISR of the pushbutton interrupt releases this semaphore, and in figure 4, our first task StartReactButton() contains the line osSemaphoreWait(myBinarySem01Handle, osWaitForever). This line places the task into a blocked state, until it receives the specified semaphore; this is equivalent to using the osDelay() function; as osDelay() places tasks in a blocked state until a specific duration has passed. In the case of osSephamoreWait() however, the task is placed in a blocked state for the duration that the semaphore has not been released. Once the semaphore has been received this task simply increments the state attribute discussed in part 1.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{

    if (GPIO_Pin == pButton_Pin)
    {
        osSemaphoreRelease(myBinarySem01Handle);
    }
}
```

Fig. 3. Push-Button Callback Function, ISR

```
void StartReactButton(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    osSemaphoreWait(myBinarySem01Handle, osWaitForever);
    ++sensor_state;
    sensor_state %= total_states;

  }
  /* USER CODE END 5 */
}
```

Fig. 4. StartReactButton(), Task 1

Our second task is then reading values from the specified sensor, based on the enum type we defined previously, sensor_state. As this is the most fundamental and crucial step of our program; we set its priority as the highest between the three tasks. This is especially in order to make sure that once the processor enters this task, it does not abandon it for another, or if another task is contested with this one, then this one is chosen. As seen in figure 5, the task simply checks which state the application is in, then commences to read the sensor value using the respective BSP function. Then, once the value is read and the variable is updated, the Boolean new_sample is set to true to signify that the current reading being carried is indeed a new sample. Further, we use the osDelay(100) function in order to place this task in a blocked state for around 100 ticks, allowing us to sample the sensors at a rate of 10 Hz, with our 120 MHz system.

```
void StartReadSensor(void const * argument)
{
  /* USER CODE BEGIN StartReadSensor */
  /* Infinite loop */
  for(;;)
  {
      osDelay(100);
        if (sensor_state == TEMP_STATE){
            temp = BSP_TSENSOR_ReadTemp();
        }else if (sensor_state == ACCEL_STATE){
            BSP_ACCELERO_AccGetXYZ(accel);
        }else if(sensor_state == MAGNETO_STATE){
            BSP_MAGNETO_GetXYZ(magnet);
        }else if(sensor_state == PRESS_STATE){
            pressure = BSP_PSENSOR_ReadPressure();
        }
        new_sample = true;
  }
  /* USER CODE END StartReadSensor */
}
```

Fig. 5. StartReadSensor(), Task 2

Finally, our third task, found in figure 6 pertains to printing the reading obtained from the relevant sensor through the same USART setup discussed in part 1. As seen from the figure, this task is very similar to the program structure in part 1; it checks which sensor reading the program is currently in and formats the relevant readings into a string using sprintf. Then, once the string has been formatted it is sent over to the USART by way of the HAL_UART_Transmit() function. Once the string has been printed, we set the Boolean we discussed previously to false, signifying that the reading that was taken in this cycle has been printed in this cycle. Lastly, we use the same osDelay() function as before, but this time with a delay of 50 ticks; allowing us to synchronize the printing and the reading respectively, with a half cycle being printing and a cycle being reading the sensor.

```
void StartPrintVal(void const * argument)
{
  /* USER CODE BEGIN StartPrintVal */
  /* Infinite loop */
  for(;;)
  {
      osDelay(50);
      if(new_sample){
        if (sensor_state == TEMP_STATE){
            char temp_str[20];
            sprintf(temp_str, "\r\nTemperature: %.2f", temp);
            HAL_UART_Transmit(&huart1,temp_str,sizeof(temp_str),10);// Sending in normal mode
        } else if (sensor_state == ACCEL_STATE){
            char accel_str[40];
            sprintf(accel_str, "\r\nAcceleration x:%.4d, y:%.4d, z:%.4d", accel[0], accel[1], accel[2]);
            HAL_UART_Transmit(&huart1,accel_str,sizeof(accel_str),10);// Sending in normal mode
        }else if(sensor_state == MAGNETO_STATE){
            char magneto_str[40];
            sprintf(magneto_str, "\r\nMagnometer x:%.4d, y:%.4d, z:%.4d", magnet[0], magnet[1], magnet[2]);
            HAL_UART_Transmit(&huart1,magneto_str,sizeof(magneto_str),10);// Sending in normal mode
        }else if(sensor_state == PRESS_STATE){
            char press_str[20];
            sprintf(press_str, "\r\nPressure: %.2f", pressure);
            HAL_UART_Transmit(&huart1,press_str,sizeof(press_str),10);// Sending in normal mode
        }
        new_sample = false;
      }

  }
  /* USER CODE END StartPrintVal */
}
```

Fig. 6. StartPrintVal(), Task 3

III. Conclusion

In conclusion, this laboratory focused on the integration of I2C peripherals and the FreeRTOS operating system in a microcontroller-based application. The first part of the lab required setting up various sensors on the I2C bus and UART on the development kit for communication. These functionalities were used to realize an application that toggled between sensor readings upon pushbutton presses. The second part of the lab involved the implementation of the same application but with the use of the FreeRTOS OS, splitting the application into three tasks: pushbutton press detection, sensor data reading, and data transmission to the terminal. The use of FreeRTOS allowed for efficient task management, ensuring that the most crucial task, sensor data reading, had the highest priority. Through this lab we have been familiarized with the functionalities of threading using an OS, a valuable tool that can reduce computation time and complexity in terms of making the processor more efficient; having it only perform code when it is necessary.