ECE 153B – Winter 2024
Sensor and Peripheral Interface Design
Project

Deadline: **March 21, 2024, 11:00 AM**

Objectives

1. Understand the importance of non-blocking operation.
2. Learn how to design non-blocking step motor implementation.
3. Learn how to design non-blocking terminal communication implementation.

Grading

Part	**Weight**
Part A	30 %
Part B	30 %
Part C	40 %

Submit your code and answers to the questions to the submission link on Gradescope by the specified deadline. In the week following the submission on Gradescope, you will demo/checkoff your lab to the TA.

Please note that we take the Honor Code very seriously, do not copy the code from others.

Contents

1 Necessary Supplies

- STM32L4 Nucleo Board
- Type A Male to Mini B USB Cable
- Breadboard & Jumper Wires
- 28BYJ-48 5v stepper motor + ULN2003 driver board
- HC-05 Bluetooth Module
- 1 kΩ Resistor (x2)
- TC-74 Temperature Sensor
- ADXL345 Accelerometer

2 Project Overview

This goal of this course project is to create an automated garage door system using what you have learned this quarter. The garage door should be controlled using the stepper motor. The garage can be controlled using either terminal commands or under specified conditions such as an increase/decrease in temperature. The accelerometer should be used to manage the position of the door to determine whether it has been closed or opened. A Bluetooth terminal should be implemented to read out the garage door behavior (open/close status, temperature status) and send back out manual commands for the door. Specifics such as temperature constraints and open/close values is left to you. To make a garage door, you may use cardboard, 3D printed materials, or whatever you see fit to build the door. To complete the project, two techniques will be required: non-blocking motor control and non-blocking UART. The description of these are as follows:

A. You will use Termite input to control the movement of step motor using non-blocking control of the step motor

B. You will display the I2C and SPI sensor reading on Termite every second while maintaining the previous features by implementing non-blocking UART communication to Termite.

3 Part A – Non-Blocking Step Motor

In this part of the lab, we are going to use Termite to control the step motor rotation direction. It will have 3 modes: rotate clockwise, rotate counterclockwise, and stop rotating.

In lab 2, we drive the step motor with a busy-waiting delay. However, this prevents the CPU from doing other tasks, such as receiving inputs from other devices. Thus, we would like to drive the step motor using SysTick instead. We are going to use half stepping for this lab. The steps are:

1. Adapt UART, SysTick, and Motor code from previous labs.
2. Prepare values for the GPIO to drive the motor in each of the 8 steps of the half stepping sequence. Since the motor pins are mapped to a single GPIO port, write each step as a single word to give to the `GPIO_ODR` and store the words in the array `HalfStep` in `motor.c`. Specify the `MASK` as well, which is to clear all the motor pin bits.
3. Find the minimum delay t_{min} in milliseconds required to run the step motor precisely.
4. Call the `rotate()` method from `motor.c` in the SysTick handler every t_{min} interrupts by using an counter that counts up to t_{min} then resets itself. (If your t_{min} is exactly 1, you don't need this step.)
5. Use another static counter in `motor.c` to keep track of the step in the sequence for the motor. It should count upward in clockwise rotation, and downward in counterclockwise rotation.
6. In `rotate()`, increment or decrement the counter based on current rotation direction to find the next value for GPIO.
7. Implement UART communication so that the direction can be set based on user input.

4 Part B - Non-Blocking UART

In this part of the lab, we are going to print readings from the I2C and SPI peripherals every second while maintaining the features from part A. Since I2C and SPI communication requires us to wait on ready flags, we cannot put them in the SysTick interrupt. Thus, we need to call them in the main function, and use delay function to time a 1 second period.

The implementation has the following steps:

1. Initialize UART and DMA, and setup corresponding interrupt handlers.
2. Implement `UART_print` that sends a string through UART. This function should be non-blocking. If the DMA is not sending data, it should set up the DMA and start it. If there is an ongoing transmission, it should store the data to send either in a backup array or in a queue.
3. Implement `on_complete_transfer` that checks and sends a pending string through UART. This function should be called when a UART TX operation is completed via the UART TC interrupt.
4. Implement `transfer_data` to accumulate user input. This function receives a character and appends it to a buffer. Then if the character is $'\backslash n'$, call `UART_onInput` and clear the buffer.
5. Setup the DMA or UART interrupts to call the function specified in step 4.
6. Implement `UART_onInput`: Based on the input, configure the motor direction, and then print a feedback message based on what is done.

In this part, you are free to decide how you are going to implement it. However, if you are unsure about it, you can still follow the template code.

The suggested implementation uses 2 buffers: `data_t_0` and `data_t_1`, and then we have 2 pointers: `active` and `pending`. `active` represents the buffer that is currently selected for DMA, and `pending` is the one that is not. When sending the pending data, instead of copying data from `pending` to `active`, we switch the buffers being used by reconfiguring DMA memory buffer to `pending` then exchanging `active` and `pending`.

In this part, since we are using a different way to send data, we can no longer use `printf`. Instead, we use `sprintf` that writes to a buffer, and then use `UART_print` to send the buffer to UART. This is demonstrated in `main.c`

If you would like to follow the template code, here are some steps to get you started:

The first step is to plan how to perform data transactions.

- In DMA.h, set up the DMA similar lab 6. Make sure the DMA is set up to transfer data from peripheral to memory and that the data size is 8-bits since we are sending *char*'s.
- In our `UART_Init` functions, set up the correct DMA channel to use `USARTx_TX`. Set up the *active* buffer as the memory location in the DMA CMAR register
- For transmitting data, the DMA will send data from memory to UART. This process starts when the DMA is enabled and stops when UART completes sending data through UART's Transfer Complete interrupt. We need to implement the following features:

  - A function to store strings into a memory buffer for DMA TX. We can use `sprintf` to store the string we want to print into a *char* array.
  - A function to prepare the DMA with our new string. We can write `UART_printf` in `UART.c` to take in our string and measure its length. Once we have measured its length, we can configure the DMA to read from our string.
  - When there is an active transaction going on and we have a new data to send, we need to delay printing the new data without blocking. It's recommended to store the new string in another static buffer and process it after the current TX DMA operation is complete.
  - In your USART interrupt handler, you will need to set a condition for when the transmission is complete so we can send any pending data out. You can keep track on whether there is a pending transmission by using a static variable.
  - We need a function to send any pending data out. Write `on_complete_transfer()` to check for any pending transmissions and if so, set up the DMA and start the new transmission.

- For receiving data, we use the UART interrupt found in `stm32l476xx.h` to keep track of new incoming data. We need to keep receiving data until we get newline character, $'\backslash n'$, from Termite indicating the end of a message.
  - We need to store incoming characters from the UART RX buffer. We can store these characters into a static *char* array in UART.c and write `transfer_data()` to perform this action. You should include a counter in your implementation to place characters correctly in your array.
  - Once a new line is received, we can check the new message to see if we need to perform any actions. Write `UART_onInput()` in `main.c` to check the message and determine if we need to change the motor behavior.
  - To receive the characters, we can use the RXNE interrupt from UART. You should call the functions you just wrote within the interrupt to prevent your code from blocking.

When configuring the DMA, please note the following:

- Check the DMA reqeust table in the MCU reference manual to find the right DMA channel to use.
- Be sure to disable Memory-to-Memory mode as we are communicating with UART which is a peripheral bus.
- Set the memory and peripheral data size appropriately. Since we are sending characters, choose 8-bits.
- Define the memory buffer in accordance with your data size.
- Peripheral increment mode should be disabled.
- Enable memory increment mode.
- Configure the transfer direction.
- Configure interrupts. Disable interrupts that you don't want, and write interrupt handlers for the rest. Be sure to clear the global DMA interrupt flag.

5 Part C - Project Implementation

In this part, you are going to build the garage door. You need to design the door, prepare materials, and build the door. You can use cardboard or 3D printing. The accelerometer should be on the door to measure the door orientation.

Requirements:

1. The default position for the door should be closed (down facing). You should use accelerometer to ensure that it's always facing down regardless of initial motor position.
2. When temperature crosses a threshold value (should be a few degrees higher than room temperature), the door should open to a horizontal position. Use the accelerometer to ensure this. Before and after this operation, print a line to the console describing the operation. (Something similar to `"Temperature too high. Door opening."` and `"Door opened"`)
3. When temperature drops below a threshold value (should be a bit lower than the previous threshold), close the door. Use the accelerometer to calibrate the position and log your operation accordingly.
4. You should allow console control of the door. When the door is opened or closed by console, it should stay in that state for at least 3 seconds before returning to normal behavior.
5. Print the temperature when it changes. Measure temperature and acceleration with an interval of 0.1 seconds.

Implementation details:

1. Temperature and acceleration measurement code should be in the main loop.
2. Use SysTick to count the 3-seconds delay. During the delay, the temperature and acceleration should still go on, but they should not trigger the stepper motor.
3. The temperature thresholds form a Schmitt trigger. Keep track of the current state and prevent measurement noise from triggering the door.
4. When attaching the accelerometer to the door, align the X axis to be parallel to the axis of rotation of the door. When the door is closed, the Y axis measurement should be close to 1 or -1. When the door is opened, the Z axis measurement should be close to 1 or -1.

Note: You may have pin conflicts between peripherals (ex. I2C and the Bluetooth module share PB6 and PB7). You will need to select different pins for your peripherals and set them up accordingly.

6 Checkoff Requirements

1. Part A
    a) Motor is controlled over UART
    b) Motor behavior is non-blocking
2. Part B
    a) UART is non-blocking via DMA
    b) I2C and SPI readings are implemented into main loop with 1s delay
    c) Motor control from Part A is included
3. Project Implementation
    a) Implement a physical garage door
    b) Bluetooth terminal to control garage door and readout status
    c) Incorporate the temperature sensor to create constraints on when to open and close the door (heat or cold)
    d) Incorporate the accelerometer to measure the position of the door to determine door position. Stop the door when the door reach target position.

7 References

[1] STM32L4x6 Advanced ARM-based 32-bit MCUs Reference Manual

[2] Yifeng Zhu, "Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C", ISBN: 0982692633