LAB 6

COMMUNICATION USING THE UART, PART II, AND INTERRUPTS

INTRODUCTION

This week in lab you will get familiar with a key concept in embedded systems – interrupts. You will continue to use the UART, and instead of using polling as the method of input/output, you will use interrupts. You will remove the blocking function for receiving data, and instead trigger an interrupt when a byte is received in the UART. Doing so, your code will not have to busy-wait in a loop waiting for a key to be pressed in PuTTY, possibly ignoring other work to do, and instead your code will be notified when the character from the keystroke has been received. This functionality will be very useful for the lab project.

REFERENCE FILES

The following reference files will be used in this lab:

- lab6_template.c, contains a main function template to be used as needed with uart.h and uart.c
- uart.h, header file for uart.c
- uart.c, program file with partially implemented functions (use your uart.c file from Lab 5)
- lab6-interrupt_template.c, contains a main function template to use with uart-interrupt.h and uart-interrupt.c files
- uart-interrupt.h, header file for uart-interrupt.c
- uart-interrupt.c, program file with partially implemented functions (**you will implement the functions in this lab**)
- cyBot_Scan.h, header file for pre-compiled library for CyBot sensor scanning
- libcybotScan.lib: pre-compiled library for CyBot sensor scanning (note: must change extension of file from .txt to .lib after copying)
- lcd.c, program file containing various LCD functions
- lcd.h, header file for lcd.c
- timer.c, program file containing various wait commands
- timer.h, header file for timer.c
- TI Tiva TM4C123G Microcontroller Datasheet
- TI TM4C123G Register Definitions C header file: REF_tm4c123gh6pm.h
- GPIO and UART register lists and tables: GPIO-UART-registers-tables.pdf
- Reading guides for GPIO, UART, and interrupts, as needed, e.g., reading-guide-interrupts.pdf

The code files are available to download.

PRELAB

See the prelab assignment in Canvas and submit it prior to the start of lab.

STRUCTURED PAIRING

You are expected to continue to use structured pairing in this lab and in future labs. It was introduced in Lab 2.

PART 1: SENDING SCAN DATA FROM CYBOT TO PUTTY

The first part of this lab is similar to part 4 from Lab 5. Start with your UART code from Lab 5 that uses polling (uart.h and uart.c) (you can compare your uart.c code with the provided uart.c template file to double-check that your code is complete). You are not using interrupts in this first part of the lab, so do not use the files with "interrupt" in the filenames.

The purpose of part 4 of Lab 5 was for the CyBot to send messages to PuTTY using the uart_sendStr() function, while also having the CyBot receive and echo characters typed into PuTTY. In this lab, you will send sensor scan data to PuTTY instead of messages. The lab6_template.c file can be used to create your main program.

Sensor scan data are important in the lab project. As shown in Lab 4, the CyBot will collect data about the environment around it using a servo motor that scans side to side while collecting distance information using two sensors, an infrared (IR) sensor and a sonar (ultrasound or PING) sensor. In Lab 4, you used the cyBot_FindObjects() function to get information about objects found in the test field. This function identified objects using distance data from the IR and PING sensors; it activated and read the sensors and analyzed the sensor data for you. In this lab, instead of displaying object information, you will display the sensor data in PuTTY. You will learn more about the sensors and sensor data processing in Labs 7 and 8. To get the sensor data for this lab, you will use functions from a precompiled scan library.

Copy **libcybotScan.lib** into your project, and take a look at the **cyBot_Scan.h** header file.

```
typedef struct{
     float sound_dist;  // Distance in cm from PING sensor
     int IR_raw_val;    // Raw digital value from IR sensor
} cyBOT_Scan_t;

void cyBOT_init_Scan(void);

// Rotate sensors to specified angle, and get scan data
void cyBOT_Scan(int angle, cyBOT_Scan_t* getScan);
```

Notice the cyBOT_Scan function that rotates the servo motor to the given angle and collects sensor data in a struct. The function takes a reading from the PING sensor (distance in centimeters) and IR sensor (raw digital value from analog-to-digital conversion). For example, this function call returns the sensor data read at an angle of 45 degrees in your program's struct variable:

```
cyBOT_Scan(45, <address of your cyBOT_Scan_t variable>);
```

Your program should display the following information in PuTTY:

```
Angle       PING distance        IR raw value
0               ?                    ?
5               ?                    ?
10              ?                    ?
…
180             ?                    ?
```

Note: The function for scanning and collecting data has not been calibrated to your specific robot. So it is likely that 0 and 180 degrees are off from actual rotation of the motor (it could be off by a lot). Don't worry about it in this lab. In a later lab, you will be calibrating your own version of the servo motor functionality.

When the CyBot receives a 'g' from PuTTY (go command), have the servo motor scan from 0 degrees to 180 degrees. Take a sensor reading at every 5-degree increment. Send this information back to PuTTY as it is collected, and display it in the format above. Use a tab to separate each column.

Tip: In order to format your data in PuTTY, you can send '\r' to return, '\t' for tab, and '\n' for newline.

CHECKPOINT:

Send a 'g' command to the CyBot from PuTTY to start a scan, and display formatted sensor information in PuTTY.

MORE INFORMATION ABOUT THE SENSOR DATA IN PART 1

See this video for a demonstration of an older lab project that shows the CyBot navigating a test field, detecting, recognizing and avoiding objects, as appropriate, to reach its destination (white square): https://www.youtube.com/watch?v=ulidN0rs1NA
Notice the sensors scanning the region in front of the robot.

It might also be helpful to revisit the video shared at the beginning of the semester that demonstrates the virtual lab environment. For example, watch from about time 16:30-19:30. You will see a demo displaying sensor data in PuTTY (one difference is that your IR data from the scan function are raw digital values, not distance values as shown in the video).
Video link: https://drive.google.com/drive/folders/1EhUmbo9rfCEiSuDjeUOAIeq--IJ5wBC3
(this is in the same Google Drive folder where code files are found, and this video is under "Course Videos" as "virtual lab environment demos.mp4")

The datasheets for the IR and PING sensors are provided on the Lab Resources page in Canvas. These sensors will be examined in more detail in Labs 7 and 8.

IR Sensor
The infrared (IR) sensor is an electro-optical device that emits an infrared beam from an LED and has a position sensitive detector (PSD) that receives reflected IR light. A lens is positioned in front of the PSD in order to focus the reflection before it reaches the sensor. The PSD sensor is an array of IR light detectors, and the distance of an object can be determined through optical triangulation. The location of

the focused reflection on the PSD is translated to a voltage that corresponds with the measured distance. The IR distance sensor is designed to measure distances from 9 – 80 cm. However, the IR sensor can only moderately accurately display a distance value for an object 9 – 50 cm away. Objects must be placed far enough apart to provide a measurable gap between them. This gap may not be measurable by the PING sensor, but the tighter beam of the IR distance sensor will generate data consistent with a physical gap.
The IR sensor is read using the analog to digital converter (ADC) module on the microcontroller. The analog voltage from the sensor is converted to a digital value by the ADC. The relationship between the voltage value and the measured distance is shown in the sensor's datasheet. This is the focus of Lab 7.

PING Sensor
The range of the PING SONAR sensor is 2 cm to 3 m. SONAR (Sound Navigation and Ranging) uses an ultrasonic burst (well above human hearing range) to determine the presence and distance of objects. SONAR is a term that is valid for both air and water. The frequencies of the pulses emitted are different for these two mediums. The sensor emits 40 KHz pulses and receives a sound wave reflected from an object. The distance is calculated by determining the time between emitting a sound pulse and receiving an echo. The echo is translated into distance given the speed of sound in a particular medium. In our case, the medium is air. While air temperature does affect the speed of sound, for our purposes we will assume the speed of sound to be constant at 340 m/s or 1130 ft/sec. The PING sensor will only report one echo for any given pulse. This one echo is the first received to meet the minimum threshold used to discriminate between noise and a proper echo. The pulse does spread as it travels away, so the first reflection may come from an object that is not directly in front of the sensor. Clutter affects the usefulness of the sensor. The composition and shape of objects affects the amount of sound that gets reflected back to the sensor. A soft material like fabric will absorb more sound than a hard material like steel. It is possible to angle a box so that sound waves will reflect away from the sensor, so the box may "disappear" like the B2 or F117 US Air Force airplanes.

The PING sensor is read using a special mode of the timer module on the microcontroller. The timer module measures the time taken to receive the echo, from which distance is calculated. This is the focus of Lab 8.

Object detection is done by distinguishing between distance measurements that pertain to objects in contrast to background noise. Distance measurements are needed to detect the presence of an object, and from these measurements, the edges of the object can be estimated at particular angular positions.

PART 2: CONTROLLING THE SENSOR SCAN

In part 1, you pressed 'g' in PuTTY to send a go command to the CyBOT to initiate a sensor scan. Now, suppose you want to stop the scan.

Try modifying your code from part 1 to stop scanning when a stop command (e.g., character 's') is received from PuTTY.

Think about whether this is possible with a blocking receive function. In other words, what is the effect of using a blocking function on the rest of your program? What happens when your program waits for a character to be received from PuTTY?

Try writing a non-blocking receive function that returns the character received, or returns 0 if no character was received. Make a copy of the uart_receive() function and rename it as uart_receive_nonblocking(). Modify this new function so that it is nonblocking (i.e., does not busy-wait on the RX flag). Note that the function will still need to check the flag. Use this new function to help you implement a stop command.

CHECKPOINT:

Send an 's' command to the CyBot from PuTTY to stop a scan. Otherwise, the scan should behave as it did in part 1.

PART 3: UART USING INTERRUPTS

As shown in part 2, if the uart_receive() function is implemented as a blocking receive, that means the function is waiting in a loop until the UART receives a byte. This prevents a program from doing other tasks while waiting. A nonblocking receive solves that problem. However, even with a nonblocking receive, there are other issues to consider. For example, the behavior of your program depends on how frequently it checks for the stop command, other code your program is executing, and how quickly it reacts to stop the scan. Whether due to slow input devices or slow software design, your program could be unresponsive to other tasks. One way to make programs more efficient and responsive is to use interrupts.

Suppose that instead of having your program wait for a byte to be received, the bytes are automatically received and echoed, independent of your program. If needed by your program, the bytes could also be saved in a buffer, and your program would simply read bytes from the buffer when needed while doing other tasks.

This approach is shown below in Figure 12.2 from the VYES book (here "Fifo" means a character buffer).

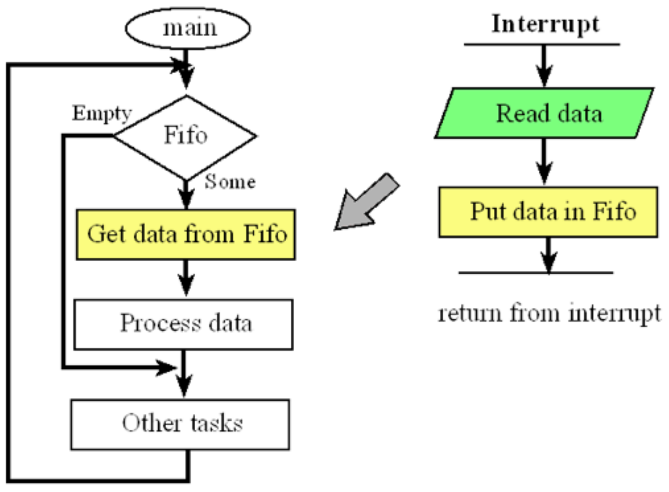

Figure 12.2. For an input device we can use a FIFO to pass data from the ISR to the main program.

In this part of the lab, you will re-write your code to use interrupts. Simply put, your program will no longer call any version of the uart_receive() function. A character will be received by another function called an interrupt handler or interrupt service routine (ISR). While you as a programmer write the ISR, you do not call it. It is executed automatically when a receive/RX interrupt occurs and is processed by the interrupt system.

Using interrupts requires the following steps:

1. Enable interrupts from UART1 RX (receiving a byte triggers a UART interrupt)
2. Enable the interrupt controller (NVIC) to process UART1 interrupts and tell the CPU which interrupt handler function (or ISR) to execute
3. Enable interrupts in the system (letting the CPU suspend executing your main program, and go execute the ISR)
4. Declare volatile global variables as needed for the ISR to share data with your main program (an ISR does not return a value, and does not have arguments, so global variables are used)
5. Write an interrupt handler function (ISR) to "service the interrupt"
6. Write the main program so that it uses global variables as needed to "talk" with the ISR

Most of these steps have been implemented for you in this part of the lab. For now, our goal is to introduce you to interrupts so you become aware of what they are, why they are important, and generally how they work. You should gain some awareness of interrupt processing features of the TM4C123 microcontroller, even if you don't understand all of the coding details. Your lab project will involve the use of several interrupts.

****Note**: In order to use interrupts, you must include driverlib/interrupt.h and stdbool.h. You should use the following header files.

```
#include <stdbool.h>
#include "driverlib/interrupt.h"
```

Tip: In order to view functions that may be useful for configuring interrupts, once the interrupt library is included as a header file, you can use Ctrl + Click on the name of the include file to open it. This is also useful for viewing the file containing the register macros.

Refer to the following "interrupt" code files for Lab 6 and browse through all of them:

- lab6-interrupt_template.c
- uart-interrupt.h
- uart-interrupt.c

First, update the code in uart-interrupt.c.

1. Copy code to this file as appropriate based on the initialization code previously written for the UART (i.e., from uart.c, where you replaced the "???" placeholders). Also copy code to implement the functions (i.e., the "TODO" placeholders).
2. Notice the new "?????" placeholders in this file. You need to write code for these. The comments in the code are intended to provide guidance.
3. Start with the code in the uart_interrupt_init() function to initialize UART1 to use receive (RX) interrupts. Note that you will only be using RX interrupts. You will not use interrupts for transmit (TX).
    a. You will need to look up specific registers in the Tiva datasheet to complete the code.
    b. The NVIC enable registers (NVIC_ENx_R) appear complicated in the datasheet. Instead, all you need to know is that there is <u>one enable bit in an NVIC EN register for each</u>

device that can interrupt. Devices that can interrupt are numbered starting with 0 (called the Interrupt Number, or IRQ Number, where IRQ stands for Interrupt ReQuest). The Interrupt/IRQ Number for UART1 is 6. This is shown in Table 2-9 in the datasheet. That means that bit 6 in the NVIC enable register is used to enable or disable UART1 interrupts.

4. Next see the UART1_Handler() function, which is the interrupt handler or interrupt service routine (ISR). You, as an embedded programmer, write the code for this function based on how you want to respond to the UART1 interrupt. However, your program does not call this function. The interrupt system of the microcontroller tells the CPU if and when to execute this function. Read those last three sentences again; this is a key principle of interrupt-driven I/O. ☺
   Some ISR code has been written for you.
   a. Start by writing code for the "?????" placeholders in the ISR. Simply replacing these placeholders should make the ISR fully functional to receive and echo characters from PuTTY. You don't need to modify/add any other code in the ISR for now.
   b. Test that receiving and echoing work using interrupts before adding any new functionality that uses the global shared variables.
   c. In other words, at this point, once you have replaced all placeholders (???, ?????, TODO), you can build your program with the main function in lab6-interrupt_template.c, and it should support communication between PuTTY and the CyBot.

Remember, previously written UART code to receive bytes from PuTTY used a polling method for reading data. In this part of the lab, you are using an interrupt method for reading data.

CHECKPOINT:

Verify that the receive/echo functionality works using interrupts. The CyBot should receive characters from PuTTY and send each character it receives back to PuTTY to be displayed. Your program must not call any version of the uart_receive() function.

PART 4: USING INTERRUPTS TO RECEIVE SENSOR SCAN COMMANDS

Recall the go ('g') and stop ('s') commands from parts 1 and 2. In this part of the lab, implement one or both of these commands using interrupts.

For example, you could implement the go command using the original blocking receive function, i.e., call uart_receive() in your program until 'g' is typed and then start a scan, just like you did in part 1. However, instead of checking for the stop command by calling a function, as you did in part 2, let the stop command be processed as an interrupt. The comments in the code provide guidance on how to do this.

Notice in uart-interrupt.c, there are global variables. You can use these, or declare your own.

```
// value for special character to be used as a command
// e.g., initialize here or assign in main program, such as 's'
volatile char command_byte = -1;
// flag to tell the main program a special command was received
volatile int command_flag = 0; //0 means command not received
```

Next look at the interrupt handler code.

```
    {
            //AS NEEDED
            //code to handle any other special characters
            //code to update global shared variables
            //DO NOT PUT TIME-CONSUMING CODE IN AN ISR

            if (byte_received == command_byte)
            {
              command_flag = 1;
            }
        }
```

If you are only processing one command using interrupts, e.g., stop command using character 's', the above code should suffice. If you want more commands to be interrupt-driven, then you will need additional comparisons with all possible commands.

Keep in mind that the global variables used here in the interrupt handler and in the main program provide a way for these code elements to "talk" to each other about a command that has been received.

Note the comment: do not put time-consuming code in an ISR. Interrupt handlers are supposed to get their work done as quickly as possible so that a) the CPU can resume executing your program, and b) the next interrupt is processed promptly. For example, do not call the scan function in the ISR (call it in your main program). Do not busy-wait on any other input in the ISR.

Finally, take a look at lab6-interrupt_template.c. You did not need to modify the main program for part 3, and now you will add code. Read through the "YOUR CODE HERE" comments and add new functionality.

Here is some functionality you may want in the main while(1) loop:

- Code from part 2 to start the scan when the go command is received
- Scanning and displaying sensor data in PuTTY
- Checking the command_flag to see if the stop command was received, and if so, stop the scan
- Reset the command_flag to 0

There are many ways to program this, and the code and comments suggest one approach.

CHECKPOINT:

Start and stop a scan using commands from PuTTY, with at least one command being interrupt-driven.

DEMONSTRATIONS:

1. **Functional demo of a lab milestone** – Specific milestone to demonstrate is Part 3.
2. **Debug demo using debugging tools to explain something about the internal workings of your system** – The TA will announce any specific debugging requirements at the start of lab; otherwise you will create your own debug demo based on your needs and interests in the lab.
3. **Q&A demo showing the ability to formulate and respond to questions** – This can be done in concert with the other demos.