Parallel I/O and Keyboard Scanning

4.1 Objectives:

Microprocessors can monitor the outside world using input ports. They can also control it using output ports. The TM4C123G (Tiva) performs I/O using 6 ports.

Computer keyboards are typically wired as rows and columns of individual key switches. This switch matrix is then scanned to determine if a key is pressed, and if so, which one. The I/O capability of Tiva can be used to do this scanning. In this lab, you will learn,

- How to perform simple parallel input and output transfers using GPIO ports.
- How to use polling to do data transfers.
- How to interface the keypad with Tiva.
- How to scan the keypad and detect that a key has been pressed.
- How to de-bounce the keypad using software.

4.2 Related material to read:

- Text, Chapter 14, pages 293-312.
- TM4C123GH6PM Microcontroller Data Sheet, Chapter 10 (subset), pages 649-663.
Parallel I/O using Tiva:

Tiva has four 8-bit I/O ports (A-D), one 6-bit I/O port (E), and one 5-bit I/O port (F). The microcontroller uses its address, data, and control buses to control I/O hardware as if it were memory. Controlling I/O this way is called memory mapped I/O. Each port has an address. To output data through that port, after you configure the port, a store instruction to its data address, sends data to the port. To input data from a port, you do the same thing but use a load instruction. One caveat: although all the ports are accessed through 8-bit registers, not all ports have all 8 bits available for use. In addition, different chip packages of the TM4C123GH6PM have different bits available - the packages with fewer pins have fewer ports and fewer bits available within some ports. If you look at the user's guides for your Texas Instruments Tiva board, you will see the ports and connections available for you to use.

The pin connections for all I/O ports are labeled in Figure 4.6 below (which is also located in your user guide that came with your board). The pin names are also labeled on the board next to the pins. Port E is a multi-purpose 6-bit input / output port at address 0x4002.4000 and port B is a general-purpose 8-bit input / output port at address 0x4000.5000. These ports are programmable so that some of the bits can be used for parallel input and some for parallel output. This is done by storing a value in an 8-bit register called the direction register. The direction register for all ports is named GPIO_DIR, and is addressed like memory at address 0x0400 past (offset) the data address.

Each port has a set of configuration registers following the base address that specify how the port should function. (The GPIO_DIR register is one of them) Each of the configuration ports have the same general name, but refer only to the ports in which they follow. For instance, port B at address 0x4000.5000 has a GPIO_DIR register at 0x4000.5400 (0x0400 past the base address). Port E at address 0x4002.4000 also has a GPIO_DIR register; however, it is at address 0x4002.4400. This is also 0x0400 past the base address, but at a different location in memory. Figure 4.1a shows the base address for each of the ports on Tiva. Figure 4.1b shows the offsets and addresses of the most commonly used configuration registers. Can you see the pattern? Nearly all features of Tiva follow this format where there is a base address for the feature, and each configuration register is located at some offset from that address. This is very useful to understand. A complete list of the configuration registers can be found in the Tiva datasheet on page 660.

- GPIO Port A (APB): 0x4000.4000
- GPIO Port B (APB): 0x4000.5000
- GPIO Port C (APB): 0x4000.6000
- GPIO Port D (APB): 0x4000.7000
- GPIO Port E (APB): 0x4002.4000
- GPIO Port F (APB): 0x4002.5000

**Figure 4.1a:** GPIO base (DATA) addresses
The GPIODIR register can be read like memory, but writing to it connects internal hardware in a particular configuration. The bits in the GPIODIR correspond bit by bit with the pins of the associated port. When a bit in GPIODIR is 0, the corresponding pin in the port is programmed as an input pin. A '1' in the GPIODIR corresponds to programming that pin as an output in that port.

The GPIOAFSEL stands for Alternate Function SELect. Each pin on each I/O port can be configured to be a port for a different peripheral found on Tiva. We will see more of this later. To use the port as a simple on/off switch controlled by the DATA register, AFSEL needs to be disabled (set to zero).
GPIODEN is the Digital ENable register. Tiva has the capability to output an analog signal, however, most of our labs will be digital, so we will enable (set) these bits.

![GPIODEN Register](image)

**Figure 4.4:** General-Purpose Input / Output Digital Enable (GPIODEN)

Finally, to use any GPIO port, we must turn that peripheral on. By default Tiva turns off all of its features in order to save power. You must start them in software, usually through an initialization routine, before you configure them. To “power up” a GPIO port we start the clock that controls the GPIO port we want to use. To do this we must use the System Control portion of Tiva. Specifically, the Run Clock Gate Control register for GPIO (RCGCGPIO) at address 0x400F.E608. Bits 5:0 are associated with each of the ports available (A-F). Setting a bit to 1 starts the clock (turns on) that associated port. (Note that the base address for System Control is 0x400F.E000, and the RCGCGPIO register is offset by 0x608).

![RCGCGPIO Register](image)

**Figure 4.5:** General-Purpose Input / Output Run Mode Clock Gating Control (RCGCGPIO)

The following code shows the EQU directives you would use to configure port E for digital input, bits 7-3 of port B for inputs, and bits 2-0 of port B as outputs.

```
1 GPIO_PORTB_DATA  EQU 0x400053FC ;Port B data address
2 GPIO_PORTB_DIR   EQU 0x40005400
3 GPIO_PORTB_AFSEL EQU 0x40005420
4 GPIO_PORTB_DEN   EQU 0x4000551C
5 IOB               EQU 0x07
6 GPIO_PORTE_DATA  EQU 0x400243FC ;Port E data address
```
7  GPIO_PORTE_DIR    EQU 0x40024400
8  GPIO_PORTE_AFSEL EQU 0x40024420
9  GPIO_PORTE_DEN   EQU 0x4002451C
10  IOE         EQU 0x00
11  SYSCTL_RCGCGPIO EQU 0x400FE608

12  AREA    |.text|, READONLY, CODE, ALIGN=2
13  THUMB
14  EXPORT __main
15
16  __main      LDR    R1, =SYSCTL_RCGCGPIO
17  LDR    R0, [R1]
18  ORR   R0, R0, #0x12
19  STR   R0, [R1]
20  NOP
21  NOP
22  NOP
23
24  LDR    R1, =GPIO_PORTB_DIR
25  LDR    R0, [R1]
26  BIC   R0, #0xFF
27  ORR   R0, #IOB
28  STR   R0, [R1]
29  LDR    R1, =GPIO_PORTB_AFSEL
30  LDR    R0, [R1]
31  BIC   R0, #0xFF
32  STR   R0, [R1]
33  LDR    R1, =GPIO_PORTB_DEN
34  LDR    R0, [R1]
35  ORR   R0, #0xFF
36  STR   R0, [R1]
37
38  LDR    R1, =GPIO_PORTE_DIR
39  LDR    R0, [R1]
40  ORR   R0, #IOE
41  STR   R0, [R1]
42  LDR    R1, =GPIO_PORTE_AFSEL
43  LDR    R0, [R1]
44  BIC   R0, #0xFF
45  STR   R0, [R1]
46  LDR    R1, =GPIO_PORTE_DEN
47  LDR    R0, [R1]
48  ORR   R0, #0xFF
49  STR   R0, [R1]
50

Lines 1 – 11 just equate the port and configuration register addresses to a name that is easier to remember than an address. Line 5 equates the I/O pattern for port B, 0x07 to ‘IOB’ and line 10 equates the I/O pattern for port E, 0x00, to ‘IOE’. Making these constant definitions at the beginning of the program makes it easier to change the I/O pattern later. Lines 17 – 20 start the clock for both ports B, E. Lines 21 -23 do nothing except allow the GPIO clock to stabilize. Lines 25 – 37 configure port B, and lines 39 – 50 configure port E.
External hardware interface with Tiva:

Because of the parallel I/O capability of Tiva, it can control the outside world by connecting it to external hardware. Simple and common external hardware devices are push button switches, DIP switches, light emitting diodes (LEDs), keypads, seven segment displays, transducers, control valves, etc. In this lab, we will discuss the interface of DIP switches, LEDs and keypads. In the next lab, we will consider the interface of the seven-segment displays when we design a digital voltmeter.

Figure 4.6 shows one way to connect DIP switches to the Tiva input port. These DIP switches provide 8 individual switches. When any of these switches is in the open position, logic ‘1’ is provided to the corresponding input port pin by the corresponding pull up resistor. When the switch is in the close position, the input to the corresponding input port pin is grounded through the switch and thus provides logic ‘0’. The resistor limits the current flow when the switch is closed. For the input pins, the pull-up resistor can be programmed using PUR register (more details later in the) or it can be connected externally. In the actual switches, the switch has a tendency to bounce, i.e. the contact in the switch closes and opens the circuit quickly (milliseconds) before settling as closed. To de-bounce a switch using software, after the first switch contact is read, you need to introduce a software delay of 100 msec. During this short delay, switch bouncing is effectively locked out. This is relative simple to do by writing a short delay
subroutine (which we did in Lab 3) that you call after seeing the state change of a switched input.

Figure 4.7: Connection of DIP switches to input ports.

Figure 4.8 shows how to connect the light emitting diodes (LEDs) to the Tiva output ports. When logic low is presented on the output port pin, current flows from the power supply (3.3V) through the current limiting resistor R and the LED to the output pin. The LED then illuminates. However, when logic high is present on the output port pin, there is not a sufficient potential difference between the logic high and the supply voltage to satisfy the forward voltage drop requirement of the LED, and hence the LED does not illuminate. Your lab instructor may allow you to omit the inverters, as Tiva is capable of supplying enough current to light the LEDs by itself. However if you do omit the inverters, the “logic” will invert, and a logic low will light the LEDs.
CAUTION: Never, ever, omit the resistors!

Figure 4.8: Connection of LED’s to output ports.

Most keyboards use a number of individual switches arranged in a matrix of columns and rows. This reduces the number of I/O lines needed to interface with the keyboard. A keyboard of 64 keys could be arranged in an 8-by-8 matrix, so that only 16 I/O lines are required instead of 64. Software is used to scan the keyboard to detect a depressed switch and determine which key is pressed. In this lab, we will interface and write the software to scan a 4-by-4 keypad. Figure 14-8 in the text shows the direct connections for sixteen-key keypad, but you will design your own keypad to get a better understanding of its internal circuit.

The keypad circuit is shown in Figure 4.9. The rows are connected to the outputs and the columns are connected to the inputs of the microprocessor. You will use Port B for the rows (you will have to configure it as an output port) and Port E for the columns (you will have to configure it as an input port). Internal pull-up resistors on Port E are added by setting the appropriate bits in the PUR register (offset 0xFF10). The scanning program outputs 0’s to the rows and tests all four-column inputs in a continuous loop. From the circuit, you can see that these inputs will all be high until a switch is closed.

You will write a subroutine for scanning this keypad circuit. The flowchart for this subroutine is shown in Figure 4.11. The subroutine will output 0’s to all four-row bits, and then it will read the column bits in a continuous loop until one of them goes low.

When one input goes low, the subroutine will scan the keypad by outputting a zero to the first row and ones to the other rows. Each column will then be checked, from left to right, for a low signal. If a low signal is not detected, a zero is output to the next row, and ones to the other rows, the columns are checked again. This continues until the pressed key is found. A counter is used to keep track of which switch is being checked.
When one of these switches closes, again mechanical bouncing occurs, causing the switch to open and close rapidly several times. The scanning subroutine is so fast, that it can scan the whole keypad before the switch even gets done bouncing, missing the closed switch. A delay in the subroutine gives the switch time to stop bouncing. You can use the delay subroutine from Lab 5 to provide the delay time of 100 msec. Have your scan program call the 'DELAY' subroutine after a key is pressed.

Figure 4.9: Connection of keypad switches or DIP switches to simulate Keypad circuit
Figure 4.10 below shows the connections for the keypad you will use in the lab. It shows the relationship between the 16 keys in the diagram, with the 8 terminal pins on connector, labeled 1-8. This diagram provides sufficient information to create a relationship between the keypad numbers and the port labels in Figure 4.9 above:

Pin 1 = Port ___  Pin 5 = Port ___
Pin 2 = Port ___  Pin 6 = Port ___
Pin 3 = Port ___  Pin 7 = Port ___
Pin 4 = Port ___  Pin 8 = Port ___

Figure 4.10: 4X4 SimplyTronics Flexible Keypad Connections
Figure 4.11: Flow chart for keypad SCAN subroutine.
4.2 Procedure:

Before you come to the lab, write the flow charts for the following procedures and solve for the values of Vout and current through the resistor in the circuit in Figure 4.13 for closed and open switch. Use the Startup.s file located on the lab web page under Software Downloads, which has all of the symbol definitions for the ports defined. You can add this into your source folder of your project.

1. Configure the port E lines as all inputs and connect them to six DIP switches as shown in Figure 4.7. Configure the port B lines as all outputs and connect the lower six bits to six LEDs in the manner shown in Figure 4.8. Have the lab TA check your circuit before you turn the power ON. It would be a good idea to build this circuit before you come to the lab. Write a program to continuously read the switches connected to port E and output the status to the LED's connected to port B. Show the TA when this works.

2. Connect the SimplyTronics membrane keypad to your development board as shown in Figure 4.9. Write a program to scan this keypad (as described in the previous section). Include a subroutine to wait until the key is released before another key is read, as you do not want to read the key that is depressed more than once. The flowchart for this subroutine is included in Figure 4.11. Call the subroutines SCAN and NEXTKEY with the main program. Download the program on the board and display the key pressed on the terminal window. You don’t need to use the keypad until you have your program setup. Ask your T.A. for a keypad when you are ready to start testing.

4.3 Questions:

1. Is the polling method to input data efficient? Is that important here?
2. In the keypad circuit, what will happen if your de-bounce delay is too short? What will happen if it is too long?
3. What key is read if you depress switches 8 and C at the same time? What if you depress A and C at the same time? What about 8 and 6? Explain why this behavior is predictable

4.4 Lab report:

For the lab write up, include

1. Flowcharts and programs that you wrote before the lab.
2. A copy of your working .s files.
3. A brief discussion of the objectives of the lab and procedures performed in the lab.
4. Answers to any questions in the discussion, procedure, or question sections of the lab.
Figure 4.12: Main program and NEXTKEY subroutine for Keypad scanning.

Figure 4.13: Prelab circuit (Vin=3.3V).