ELECTRICAL AND ELECTRONICS ENGINEERING DEPARTMENT

Parallel Input/Output and Keyboard Interface

Experiment 2 - Parallel Input/Output and Keyboard Interface

Objectives

Every microcomputer system, as our MCU TM4C123GH6PM, constitutes of 3 main parts; processor, memory, and input/output facilities. Input/Output facilities are simply needed both for entering system new data and for getting processed output. The TM4C123GH6PM performs I/O using 6 ports to be explained in the following section. In this experiment, general purpose I/O (GPIO) with TM4C123GXL board will be discussed.

Keyboards are one of the fundamental input devices to the MCUs. They are typically constructed as a wired grid structure. That is, individual key switches are placed on the intersection of row and column wires. With this structure, fewer I/O pins are needed than the actual number of switches. Yet, sophisticated software is needed to provide a keyboard interface that can determine whether a key is pressed and if so, which one. In this experiment, you will design and implement that interface.

Moreover, when you close a practical switch, the two metals contact and they may generate a signal of high frequency as if an ideal switch is opened and closed many times. This tendency is called *bouncing*. Therefore, de-bouncing is required to make sure that the MCU sees only one action upon for a single opening or closing. De-bouncing is performed either via hardware device or software. In this experiment, you will try to handle that by software.

To conclude, when you complete this laboratory session, you should be able to:

- Perform simple parallel input and output transfers using GPIO ports,
- De-bounce switch using software,
- Use polling to do data transfers,
- Interface matrix structured keypad with the MCU.

1 Background Information

There are different input/output methods such as standard I/O and memory mapped I/O. In standard I/O, the I/O ports are assigned to a separate address space, whereas memory mapped I/O uses memory locations to make input and output operations. That is, the microcontroller uses its address, data, and control buses to control I/O hardware as if it were memory. In that scheme, each port has an address. Upon configuration which is explained soon, a store instruction to the port's data address enables output data through that port by sending data to the port. Similarly, to input data from a port, a load instruction is used. All the ports are accessed through 8-bit registers, however not all ports have all 8 bits available for use. The TM4C123G has four 8-bit I/O ports (A-D), one 6-bit I/O port (E), and one 5-bit I/O port (F). You may check the user's guides for your Texas Instruments TM4C123GXL board [1] to see the ports and connections available for you to use.

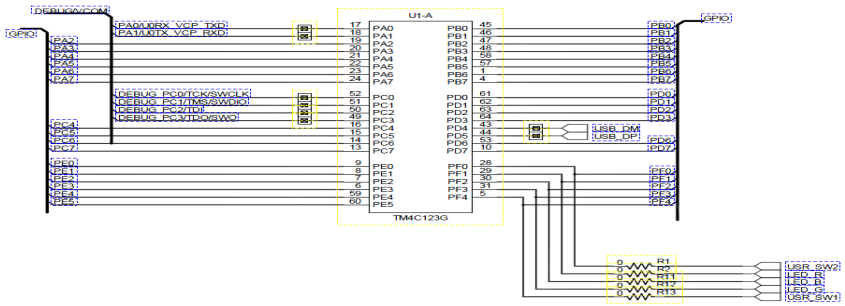


Figure 1: TM4C123GXL I/O Pin-out Specification

The pin connections for all I/O ports are provided in the user guide [1] and they are also illustrated in Figure 1. On the board, the pin names are labeled next to pins. In this experiment, ports B and E are exploited for GPIO. 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 GPIODIR, and is addressed like memory at $data\ address + 0x0400$. In other words, the data address of a port can be considered to be its base address. By giving offset to the base address, different registers can be reached to configure the functionality of the port. The base addresses of the ports on the TM4C123G are given by Figure 2-(a) and the offset values of the commonly used port registers are given by Figure 2-(b). For instance, one should write to address location 0x4002.4510 ($base\ address + pullup\ offset$) to enable/disable pull-up resistors of port E pins. It should be noted that the offset of the configuration registers are the same for all ports. The base addresses are different and that is what distinguishes the control registers of different ports. For the sake of clarity, port B at address 0x4000.5000 has a GPIODIR register at 0x4000.5400 (0x0400 past the base address). Port E at address 0x4002.4000 also has a GPIODIR register; however, it is at address 0x4002.4400. This is also 0x0400 past the base address, but at a different location in memory. Notice also that the GPIODATA register has the offset of 0x0000, that is, the base address is the GPIODATA register where you should read/write for input/output. Nearly all features of the TM4C123G 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 TM4C123GH6PM datasheet on page 660 [2].

Though the registers are of 4 Byte (the word length of the MCU registers), the least significant byte is considered for configuration. General structure of the configuration registers is given by Figure 3.

The GPIODATA register is used to transfer data to output pins or to get data from input pins. The content written to that register is transferred to the output pins and the content read from that register is the state of the inputs. Recall from the lectures that the bits 9:2 of the address bus are used as a mask to access particular pins so that some pins can be configured without affecting the others. Thus, if you want to access all the pins at once, you should provide an offset of 0x03FC while reading or writing.

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

(a)

Offset	Name	Type	Reset	Description
0x000	GPIODATA	RW	0x0000.0000	GPIO Data
0x400	GPIODIR	RW	0x0000.0000	GPIO Direction
0x410	GPIOIM	RW	0x0000.0000	GPIO Interrupt Mask
0x414	GPIORIS	RO	0x0000.0000	GPIO Raw Interrupt Status
0x420	GPIOAFSEL	RW	-	GPIO Alternate Function Select
0x510	GPIOPUR	RW	-	GPIO Pull-Up Select
0x514	GPIOPDR	RW	0x0000.0000	GPIO Pull-Down Select
0x51C	GPIODEN	RW	-	GPIO Digital Enable
0x520	GPIOLOCK	RW	0x0000.0001	GPIO Lock
0x528	GPIOAMSEL	RW	0x0000.0000	GPIO Analog Mode Select
0x52C	GPIOPCTL	RW	-	GPIO Port Control

(b)

Figure 2: GPIO base (DATA) addresses (a) and commonly used GPIO configuration address offsets (b)

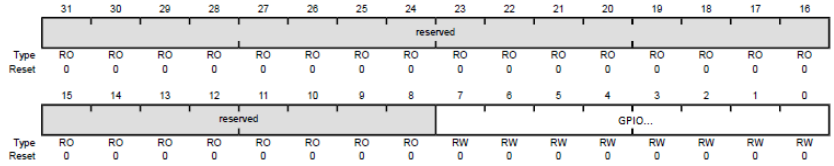


Figure 3: General structure of the configuration registers of the GPIO ports

The GPIODIR register controls the configuration of the pins as inputs or outputs. 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 **A**lternate **F**unction **SEL**ect. Each pin on each I/O port can be configured to be a port for a different peripheral located on the TM4C123G. More on this is to be explained later. For now, to use the port as a simple on/off switch controlled by the DATA register, AFSEL needs to be disabled.

GPIODEN is the **D**igital **EN**able register. The TM4C123G has the capability to output an analog signal, however, in this experiment only digital inputs/outputs are considered. Therefore, these bits should be enabled by setting them.

GPIOPUR register is the pull up control register. When a bit is set, a weak pull up resistor on the corresponding GPIO signal is enabled. Setting a bit in GPIOPUR automatically clears the corresponding bit in the GPIO Pull Down Select (GPIOPDR) register. Similarly, GPIOPDR register is the pull down control register. When a bit is set, a weak pull down resistor on the corresponding GPIO signal is enabled. Setting a bit in GPIOPDR automatically clears the corresponding bit in the GPIO Pull Up Select (GPIOPUR). As other registers, both GPIOPUR and GPIOPDR registers use only the least significant 8 bits of the register. Each of these bits corresponds to the same numbered pin of the same GPIO register. A 1 in a given position means the resistor is in the circuit and a 0 in a given position means no resistor in the circuit. "The resistor is in the circuit" means that the corresponding pin is either pulled up to the VCC or pulled down to the ground by the resistor. Whichever register (PUR or PDR) has been set most recently determines what resistor is in that GPIO's circuits.

Finally, to use any GPIO port, that peripheral have to be turned on. By default the TM4C123G turns off all of its features for the sake of lower power consumption. Prior to configuration of the ports, the ports should be powered up by an initialization routine in the software. To "power up" a GPIO port, the clock that controls the GPIO port to be used should be started. To do so, the System Control portion of the TM4C123G is to be used. Specifically, the Run Clock Gate Control register for GPIO (RCGCGPIO) at address 0x400F.E608 should be configured. Bits 5:0 are associated with each of the ports available

(F:A). That is, bit 0 is for port A and bit 5 is for port F. Setting a bit to 1 starts the clock which is associated to the corresponding port. Notice that the base address for System Control is 0x400F.E000, and the RCGCGPIO register is offset by 0x608. The same convention is used.

As an illustrative example, the following code presents the equate directives to be used to configure port E for digital input, and bits 7:4 of port B for inputs, bits 3:0 of port B as outputs.

```
1  GPIO_PORTB_DATA      EQU     0x400053FC ;data address to all pins
2  GPIO_PORTB_DIR       EQU     0x40005400
3  GPIO_PORTB_AFSEL     EQU     0x40005420
4  GPIO_PORTB_DEN       EQU     0x4000551C
5  IOB                  EQU     0x0F
6  GPIO_PORTE_DATA      EQU     0x400243FC ;data address to all pins
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
13                      AREA |.text|, READONLY, CODE, ALIGN=2
14                      THUMB
15                      EXPORT  Start
16
17 Start                LDR     R1, =SYSCTL_RCGCGPIO
18                      LDR     R0, [R1]
19                      ORR     R0, R0, #0x12
20                      STR     R0, [R1]
21                      NOP
22                      NOP
23                      NOP                             ; let GPIO clock stabilize
24
25                      LDR     R1, =GPIO_PORTB_DIR     ; config. of port B starts
26                      LDR     R0, [R1]
27                      BIC     R0, #0xFF
28                      ORR     R0, #IOB
29                      STR     R0, [R1]
30                      LDR     R1, =GPIO_PORTB_AFSEL
31                      LDR     R0, [R1]
32                      BIC     R0, #0xFF
33                      STR     R0, [R1]
34                      LDR     R1, =GPIO_PORTB_DEN
35                      LDR     R0, [R1]
36                      ORR     R0, #0xFF
37                      STR     R0, [R1]                ; config. of port B ends
38
39                      LDR     R1, =GPIO_PORTE_DIR     ; config. of port E starts
40                      LDR     R0, [R1]
41                      ORR     R0, #IOE
42                      STR     R0, [R1]
43                      LDR     R1, =GPIO_PORTE_AFSEL
44                      LDR     R0, [R1]
45                      BIC     R0, #0xFF
46                      STR     R0, [R1]
47                      LDR     R1, =GPIO_PORTE_DEN
48                      LDR     R0, [R1]
49                      ORR     R0, #0xFF
50                      STR     R0, [R1]                ; config. of port E ends
```

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, 0x0F 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.

2 External Hardware to be Used in the Experiment

In this experiment, switches and LEDs of the keypad module are used to test functionality of the simple parallel I/O and the keypad part is used to practice interfacing of array structured keypad with the MCU. The switches are push buttons that give logic high/low (Figure 6) and the LEDs are simple output monitor units where the LEDs are connected to the VCC through a resistor (Figure 7). That is, when logic low is applied to the input terminal of a LED, it emits light.

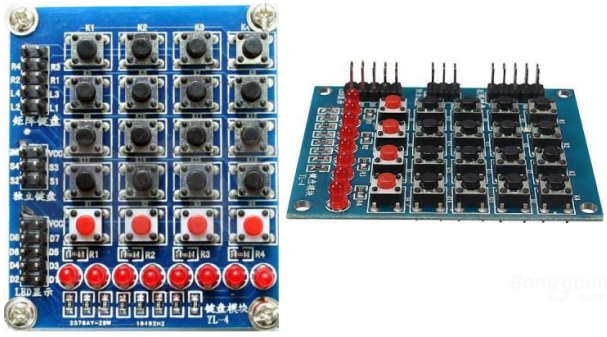


Figure 4: Keypad module to be used in the experiment

The keypad module, Figure 4, is composed of three parts: 4x4 Push Button Keypad, 4 Push Buttons and 8 LEDs. The explicit circuit diagram of the sub-modules are provided in Figures 5 - 7. In this experiment, 4x4 Keypad will be used to experience grid structured keypad interfacing.

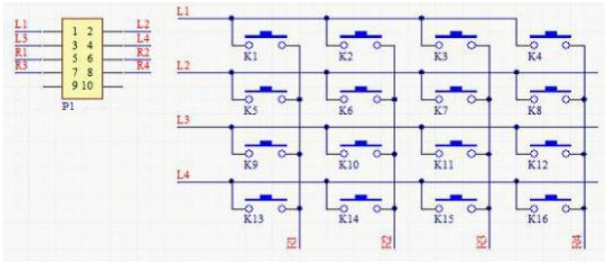


Figure 5: Circuit diagram of 4x4 Keypad

Detecting which button is pressed or which switch is closed is straightforward for simple button and switch structures that have one to one correspondences to the pins. Yet, for matrix structured keypads, a software interface should be implemented to detect which button is pressed. General framework is to divide the row and column lines as output and input lines. The lines that are to be read by the MCU are pulled up to the VCC by resistors. The rest of the lines are left to be fed by the MCU, that is, those lines are the output lines of the MCU. In Figure 8, an illustrative diagram is provided. In that design, the row lines are fed by the MCU and the column lines are inputs to the MCU. Thanks to pull up resistor capability of TM4C123GH6PM's pins, no external connections are required to pull the lines up to the VCC. The code to enable weak pull up resistors on inputs B_7-B_4 is simple as long as the aforementioned background information in Section 1 is considered:

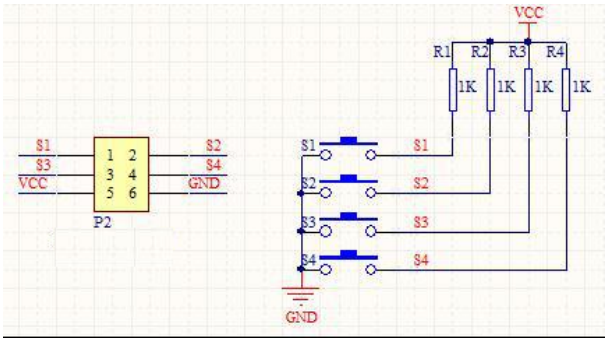


Figure 6: Circuit diagram of 4 Push Buttons

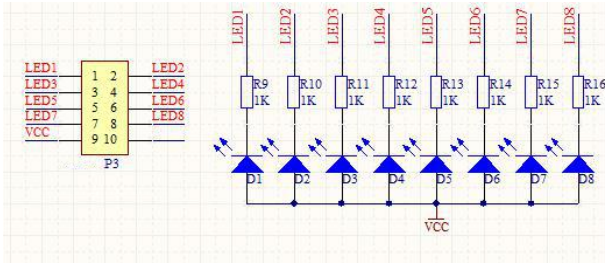


Figure 7: Circuit diagram of 8 LEDs

```
1   GPIO_PORTB_DATA      EQU     0x400053FC  ;data address to all pins
2                                            ;PUR Offset0x510
3   GPIO_PORTB_PUR       EQU     0x40005510  ;PUR actual address
4   PUB                  EQU     0xF0        ; or #2_11110000
...
25                                           ; assume configs.'ve done
26                       LDR     R0, =GPIO_PORTB_PUR
27                       MOV     R1, #PUB
28                       STR     R1, [R0]
...
```

Note that it is assumed that outputs B_3-B_0 do not require pull up resistors, since writing 0xF0 removes them.

Up to this point, the hardware part of the general framework for interfacing matrix structured keypads with the MCU has been presented. The rest will be in your preliminary work for you to figure it out.

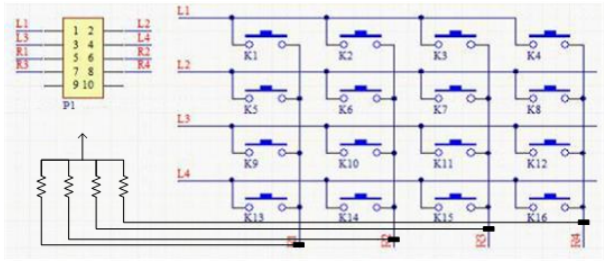
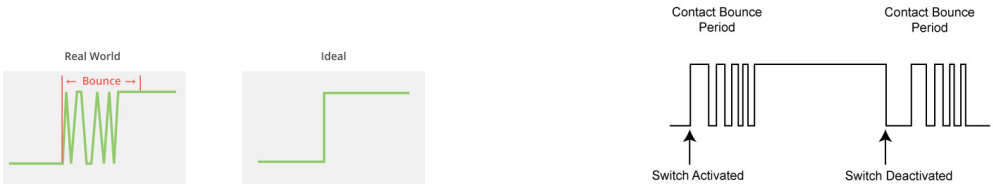


Figure 8: Circuit diagram of 4x4 Keypad with pull-up resistors

Debouncing

It should be noted that the keypad module has no debouncing hardware. Thus, bouncing problem should be handled with software. Practical push buttons of the keypad module cause high frequency signals upon contact and separation of the two metals, which is illustrated in Figure 9-(a). That signal is recognized by a digital input port as a binary pattern that reflects multiple push and release actions (9-(b)). Therefore, for a proper interpretation of a switch action, debouncing is required.



(a) (b)

Figure 9: (a) Practical vs Ideal Push Button Signal (b) Generated digital binary signal upon contact and separation

Simplest software debouncing can be just checking the consistency of the readings of two different time instances. The intuition is that the bouncing occurs during the transition from two logic levels. Thus, waiting for a period of time after the first reading and then comparing the current reading with the first reading implicitly makes your readings as if they came from an ideal push button.

3 Preliminary Work (70% Total Work)

In order to be prepared for the experimental work, "Parallel Input/Output and Keyboard Interface", you have to complete the following work:

1. (10%) Write a subroutine, *DELAY100*, that causes approximately 100 msec delay upon calling.
2. (20%) Write a program for a simple data transfer scheme. You are required to take inputs from push buttons and reflect the status of the buttons to the LEDs that are connected to the output port for approximately every 5 seconds. Namely, an input should be read for every 5 seconds and the status of that reading should remain at the output until the next reading. The status of a pressed button is 1 and the status of a released button is 0. You should use low nibble of Port B (B_3 to B_0) for inputs, and high nibble of Port B (B_7 to B_4) for outputs.
3. (70%) Consider the interface of the 4x4 keypad introduced in Section 2. You are required to write a program that continuously detects which key is pressed and outputs the ID of the key through Termite Window after the key is released. The IDs can be sequential numbers. For instance, the ID of the key at row 1 and column 1 can be 0 and the key at row 4 and column 4 can be F (0:F total 16 IDs). You may assume that only one key is to be pressed at a time and no other key can be pressed before releasing a key. Your program should be robust to possible bouncing effect during both pressing and releasing. Please attempt this problem by answering the following items:
   a. How can you detect whether **any** key is pressed?
   b. How can you detect whether a pressed key is **released**?
   c. Assuming that you have detected that a key is pressed. Explain your algorithm to determine which one is pressed.
   d. Discuss what can happen due to bouncing. How can you avoid bouncing effects?
   e. Now, develop your overall end-to-end algorithm that outputs ID of the pressed key to the terminal window and draw its flow chart.
   f. Implement the developed algorithm in part-e by using assembly language.

4 Experimental Work (30% Total Work)

1. Simple Data Transfer: (30%) Implement the simple data transfer part you have designed in 2^{nd} step (Section 3.2) of the preliminary work. Demonstrate your results in your report with illustrations.
2. 4x4 Keypad Interface: (70%) Implement the keypad interface part you have designed in 3^{rd} step (Section 3.3) of the preliminary work. Demonstrate your results in your report with illustrations.

5 Parts List

```
TM4C123G Board
1 x 4x4 Keypad Module (or an equivalent setup)
```

References

[1] TI, "Tiva^{TM} c series tm4c123g launchpad evaluation board user's guide." `http://www.ti.com/lit/ug/spmu296/spmu296.pdf`.

[2] TI, "Tiva^{TM} tm4c123gh6pm microcontroller data sheet." `http://www.ti.com/lit/ds/spms376e/spms376e.pdf`.