7

Parallel I/O and Keyboard Scanning

7.1 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.

7.2 Related material to read:

- **Text, Chapter 14**.
- TM4C123GH6PM Microcontroller Data Sheet, Chapter 10 (subset), pages 649-685.
- TM4C123G LaunchPad User Guide, Chapter 2, pages 7-12.

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 **`STRB`** instruction to its data address, sends data to the port. To input data from a port, you do the same thing but use a **`LDRB`** instruction. One caveat: although all the ports are accessed through 8-bit registers, not all ports have all 8 bits available for use. 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 7.6 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. For example, Port E is a general-purpose 6-bit input / output port and Port B is a general-purpose 8-bit input / output port. 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*.

Each port has a set of configuration registers that specify how the port should function. Each of these configuration registers are placed at a fixed offset after their port's base address. Figure 7.1a shows the base address for each of the ports on the Tiva. For example, the register that specifies the direction for each pin on a particular port is the GPIODIR register which has an offset of 0x400. Port B's GPIODIR register will be at address **0x4000.5**400 (**0x4000.5000** + 0x400 = Port B base address + GPIO offset). Port E's GPIODIR register will be at address **0x4002.4**400. Figure 7.1b shows the offsets and addresses of the most commonly used configuration registers. Nearly all features of the 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 microcontroller 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 7.1a: GPIO Base Addresses

			Port A	Port B	Port C	Port D	Port E	Port F
Register	**Offset**	**Base->**	0x4000.4000	0x4000.5000	0x4000.6000	0x4000.7000	0x4002.4000	0x4002.5000
GPIODATA	0x3FC		0x4000.43FC	0x4000.53FC	0x4000.63FC	0x4000.73FC	0x4002.43FC	0x4002.53FC
GPIODIR	0x400		0x4000.4400	0x4000.5400	0x4000.6400	0x4000.7400	0x4002.4400	0x4002.5400
GPIOIM	0x410		0x4000.4410	0x4000.5410	0x4000.6410	0x4000.7410	0x4002.4410	0x4002.5410
GPIORIS	0x414		0x4000.4414	0x4000.5414	0x4000.6414	0x4000.7414	0x4002.4414	0x4002.5414
GPIOAFSEL	0x420		0x4000.4420	0x4000.5420	0x4000.6420	0x4000.7420	0x4002.4420	0x4002.5420
GPIOPUR	0x510		0x4000.4510	0x4000.5510	0x4000.6510	0x4000.7510	0x4002.4510	0x4002.5510
GPIOPDR	0x514		0x4000.4514	0x4000.5514	0x4000.6514	0x4000.7514	0x4002.4514	0x4002.5514
GPIODEN	0x51C		0x4000.451C	0x4000.551C	0x4000.651C	0x4000.751C	0x4002.451C	0x4002.551C
GPIOLOCK	0x520		0x4000.4520	0x4000.5520	0x4000.6520	0x4000.7520	0x4002.4520	0x4002.5520
GPIOAMSEL	0x528		0x4000.4528	0x4000.5528	0x4000.6528	0x4000.7528	0x4002.4528	0x4002.5528
GPIOPCTL	0x52C		0x4000.452C	0x4000.552C	0x4000.652C	0x4000.752C	0x4002.452C	0x4002.552C

Figure 7.1b: Commonly used GPIO configuration addresses and offsets.

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 (e.g., the GPIODIR register at 0x4000.5400 controls the direction of the pins on Port B, bit 0 corresponds with pin 0). When a bit in the GPIODIR register 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.

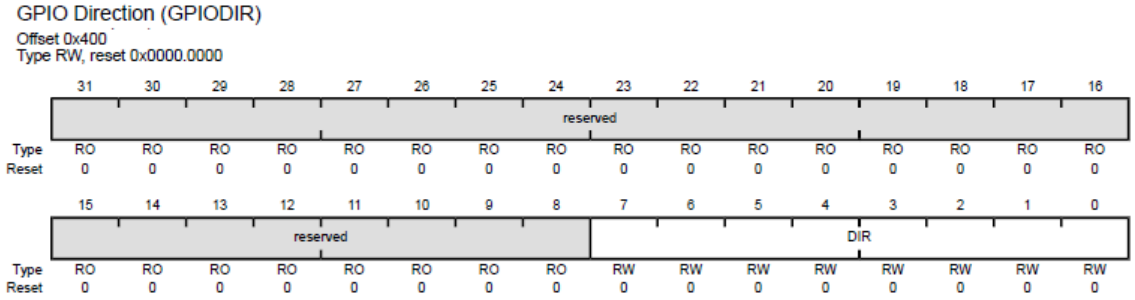


Figure 7.2: *General-Purpose Input / Output Direction (GPIODIR)*

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 set to zero.

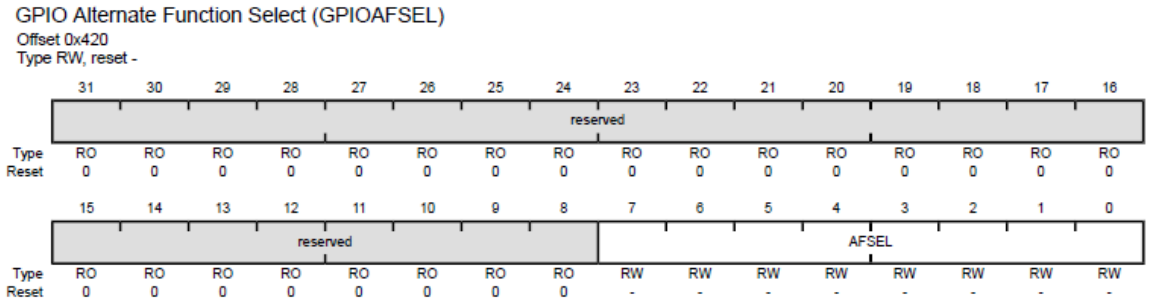


Figure 7.3: *General-Purpose Input / Output Alternate Function Select (GPIOAFSEL)*

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 set these bits to '1'.

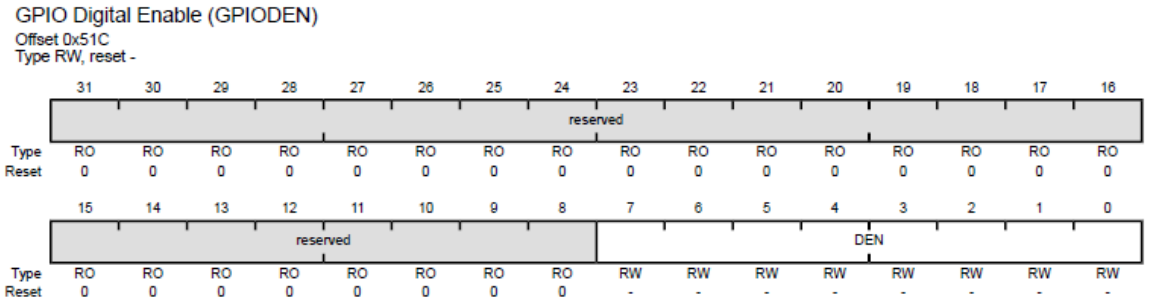


Figure 7.4: *General-Purpose Input / Output Digital Enable (GPIODEN)*

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 turn them on 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 (F:A). Setting a bit to '1' starts the clock (turns on) that associated port. For example, setting bit 0 to '1' starts the clock to Port A, setting bit 2 to'1' starts the clock to Port C. Note that the base address for System Control is 0x400F.E000, and the RCGCGPIO register is offset by 0x608.

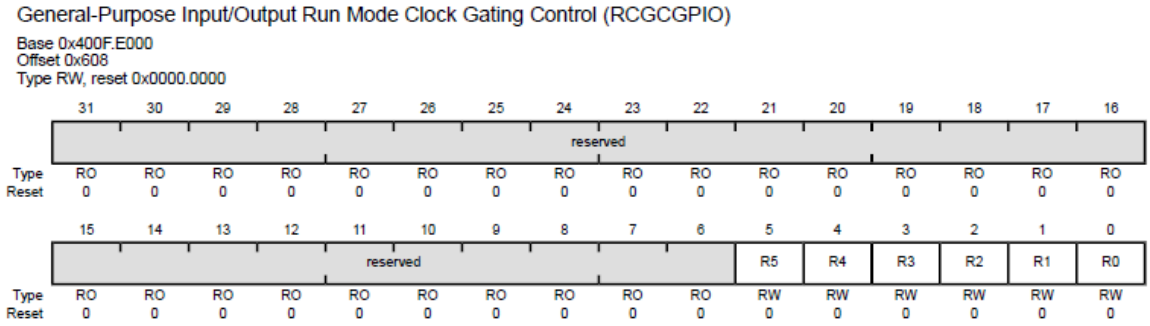


Figure 7.5: *General-Purpose Input/Output Run Mode Clock Gating Control (RCGCGPIO)*

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. Note that this code will not work as is for 7.3 part B below – it will need to be modified.

```
1       .equ GPIO_PORTB_BASE,   0x40005000      // Port B base address
2       .equ GPIO_PORTE_BASE,   0x40024000      // Port E base address
3
4       .equ GPIO_DATA,         0x3FC   // Data R/W (all bits)
5       .equ GPIO_DIR,          0x400   // Direction
```

```
6       .equ GPIO_AFSEL,        0x420   // Alternate function
7       .equ GPIO_DEN,          0x51C   // Digital Enable
8
9       .equ IOB,       0x07
10      .equ IOE,       0x00
11
12      .equ SYSCTL_RCGCGPIO,   0x400FE608
13
14
15      .section .text
16      .align 2
17      .thumb
18      .global main
19 main:LDR     R1, =SYSCTL_RCGCGPIO
20      LDR     R0, [R1]
21      ORR     R0, R0, #0x12
22      STR     R0, [R1]
23      NOP
24      NOP
25      NOP
26
27      LDR     R1, =GPIO_PORTB_BASE
28      LDR     R0, [R1, GPIO_DIR]
29      BIC     R0, #0xFF
30      ORR     R0, #IOB
31      STR     R0, [R1, GPIO_DIR]
32      LDR     R0, [R1, GPIO_AFSEL]
33      BIC     R0, #0xFF
34      STR     R0, [R1, GPIO_AFSEL]
35      LDR     R0, [R1, GPIO_DEN]
36      ORR     R0, #0xFF
37      STR     R0, [R1, GPIO_DEN]
38
39      LDR     R1, =GPIO_PORTE_BASE
40      LDR     R0, [R1, GPIO_DIR]
41      BIC     R0, #0xFF
42      ORR     R0, #IOE
43      STR     R0, [R1, GPIO_DIR]
44      LDR     R0, [R1, GPIO_AFSEL]
45      BIC     R0, #0xFF
46      STR     R0, [R1, GPIO_AFSEL]
47      LDR     R0, [R1, GPIO_DEN]
48      ORR     R0, #0xFF
49      STR     R0, [R1, GPIO_DEN]
```

Lines 1 – 7 just define the port and configuration register addresses constants to a name that is easier to remember than the particular address. Line 9 defines the Input vs. Output pattern for port B to IOB (0x07 = first three pins are output the rest are input). Similarly, Line 10 defines that pattern for port E to IOE (0x00 = all inputs). Making these constant definitions at the beginning of the program makes it easier to change the I/O pattern later. Lines 19 – 22 start the clock for both ports B, E. Lines 23 - 25 do nothing except allow the GPIO clock to stabilize. **(This must be done before modifying Ports B and E)** Lines 27 – 37 configure port B, and lines 39 – 49 configure port E.

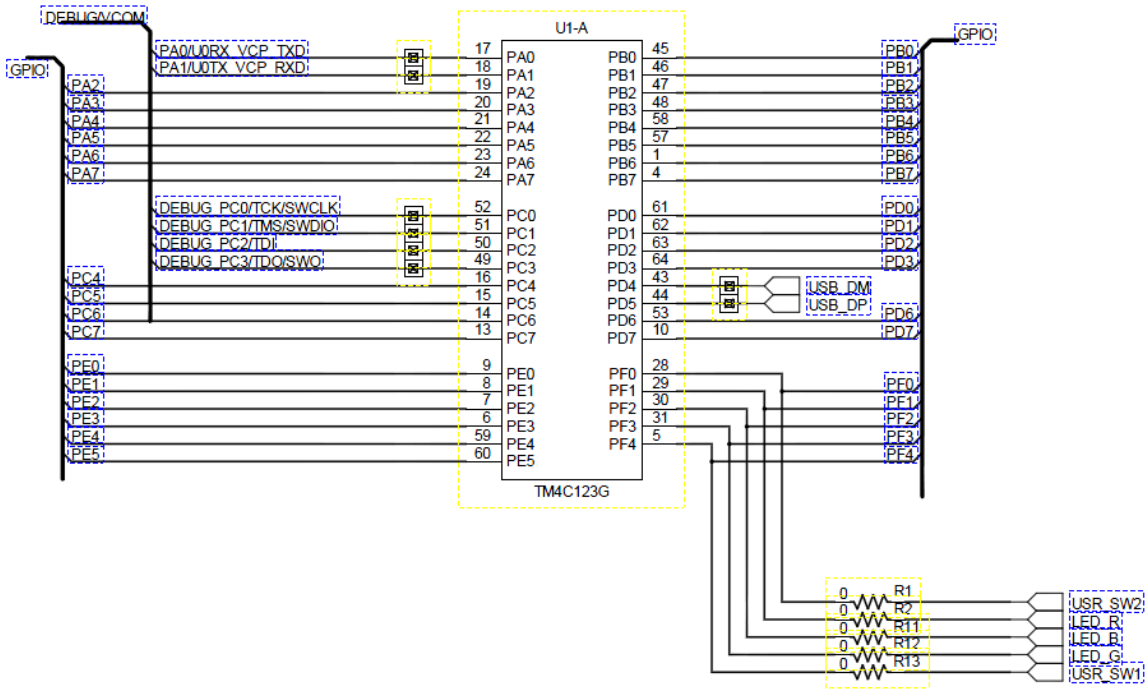


Figure 7.6: TM4C123GXL I/O Pin-out Specification

Special Case for Port D and F:

Bit 7 of Port D and bit 0 of Port F need special treatment. Those bits are "locked" and must be "unlocked" before programming. Unfortunately, since Port F is where the on-board LED and buttons are connected, this is something that you will have to deal with to use those things.

The following code will unlock Port F:

```
.equ GPIO_PORTF_BASE, 0x40025000 // Port F Base Addr
.equ GPIO_LOCK,       0x520      // Lock Register
.equ GPIO_CR,         0x524      // Lock Register
.equ Lock_Key,        0x4C4F434B // Unlock code

// Unlock Port F
LDR R1, =GPIO_PORTF_BASE   // load R1 with PortF base
LDR R0, =Lock_Key          // load R0 with lock key
STR R0, [R1, GPIO_LOCK]    // store key in PORTF_LOCK_R
MOV R0, #0xFF              // 1 means allow access
STR R0, [R1, GPIO_CR]      // enable commit for Port F
```

Place this code after you enable the clocks for Port F and before you configure it.

External hardware interface with Tiva:

Because of the 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.

GPIO Pin	Pin Function	User Device
PF4	GPIO	SW1
PF0	GPIO	SW2
PF1	GPIO	**Red** LED
PF2	GPIO	**Blue** LED
PF3	GPIO	**Green** LED

Table 7.1: Ports for TIVA board switches and LEDs

Table 7.1 and Figure 7.7 show how the on-board LEDs are connected to the board. If you trace the signals to the schematic in Figure 7.6, you can what ports are connected to each component. For example, SW1 is connected to "USR_SW1" which is connected to PF4. When either of these switches are pressed, a logic '0' is provided to the corresponding input port pin. When the switches are not pressed, you must put a pull-up resistor on the line in order to guarantee that the signal on the input pin is defined (will be logic '1' if the switch is not pressed). For the input pins, the pull-up resistor can be programmed using PUR register or it can be connected externally. ***In 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. After this short delay, the switch should be finished bouncing. This is simple to do by writing a short delay subroutine (which we did in Lab 6) that you call after seeing a switched input state change.***

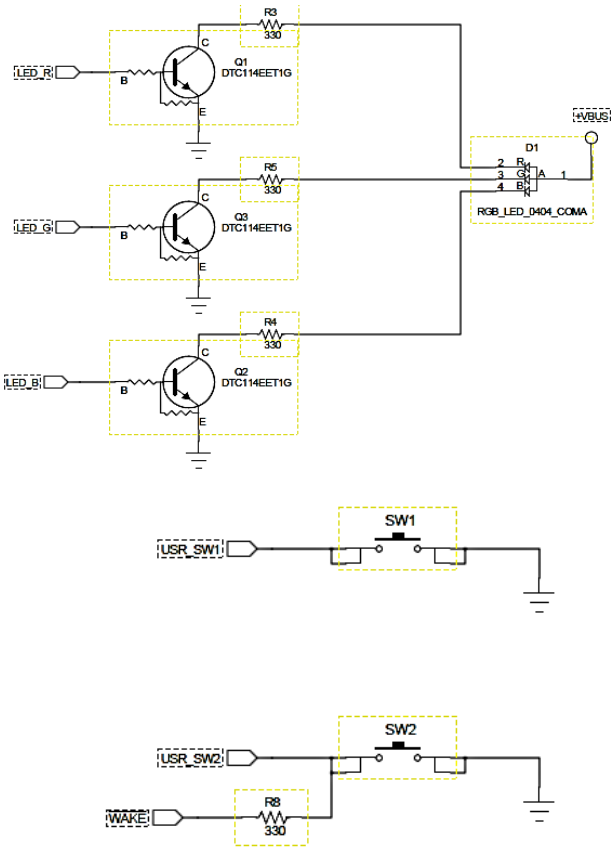


Figure 7.7: Connection of ports to on-board LEDS (top) and switches (bottom).

Keypad Interface:

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.

The keypad circuit is shown in Figure 7.8. The rows are connected to the outputs and the columns are connected to the inputs of the microprocessor. You will use Ports A and B for the rows (you will have to configure it as an output port) and Ports B and E for the columns (you will have to configure it as an input port). Internal pull-up resistors on Ports B and E are added by setting the appropriate bits in the PUR register (offset 0x0510). 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 7.10. 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 of 100 msec in the subroutine gives the switch time to stop bouncing.

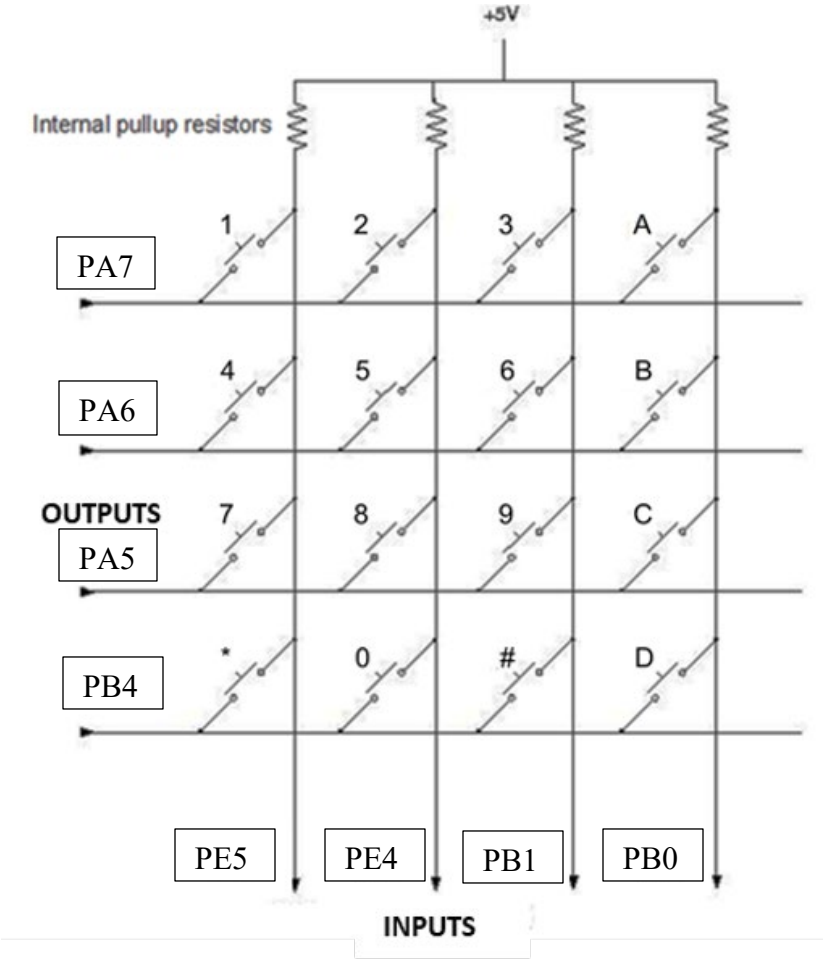


Figure 7.8: Connection of keypad switches

Figure 7.9 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 7.8 above:

Pin 1 = Port PB0 **Pin 5 = Port PB4**

Pin 2 = Port PB1 **Pin 6 = Port PA5**

Pin 3 = Port PE4 **Pin 7 = Port PA6**

Pin 4 = Port PE5 **Pin 8 = Port PA7**

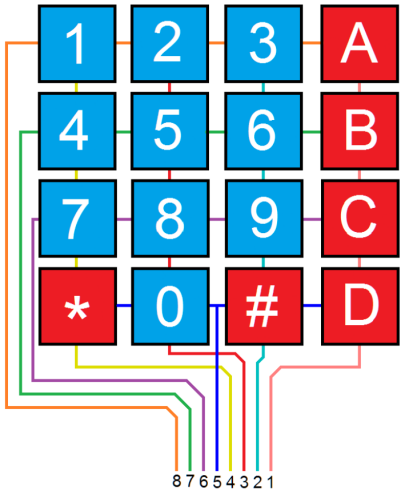


Figure 7.9: 4X4 SimplyTronics Flexible Keypad Connections

7.3 Procedure:
Include in your project folder the Startup.s file located on Canvas.

A. The goal for this section is to continuously read the on-board switches and output the status to the LEDs. Create a new project, "Lab7a". Copy the Lab7b.s, setup.s and platformio.ini files from the provided zip file.

   1. Configure the Port F pin 1-3 as outputs and Port F pin 0 and 4 as input. Port F bits 0 and 4 should have their pull-up resistors enabled. Since Port F pin 0 is a special signal, you will need to unlock the Port (see the "Special Case" paragraph above).
   2. Write a program that continuously reads the switches and outputs the status to LEDs. Don't forget to debounce the switches (~100ms delay is sufficient)! You should read the status of the switches constantly (poll). Once one of the switches has been pressed, you should delay for the debounce period and then read the switch status again.

      i) SW1 only pressed = Blue LED on

      ii) SW2 only pressed = Red LED on

      iii) SW1 and SW2 pressed = Green LED on

      iv) No switches pressed = all LEDs off

   Show the TA when this works.

B. The goal for this section is to scan a membrane keypad and output the detected key pressed to the terminal. Create a new project "Lab7b". Copy the Lab7b.s, setup.s, ECE251_asm_utils.s and platformio.ini files from the provided zip file.

   1. Connect the SimplyTronics membrane keypad to your development board as shown in Figure 7.9. You should be able to plug the keypad directly into the board using the outer row of headers next to SW1, with pin 1 towards the top.

2. Write a subroutine SCAN to scan this keypad (as described in the previous section). Also see Figure 7.10.
3. Write a subroutine NEXTKEY 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. See Figure 7.11.
4. Write the main program that calls the subroutines SCAN and NEXTKEY. See Figure 7.11. Download the program on the board and display the key pressed on the terminal window.

Show the TA when this works.

7.4 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

7.5 Lab report:

For the lab write up, include

1. A copy of your working *.s* files.
2. A brief discussion of the objectives of the lab and procedures performed in the lab.
3. Answers to any questions in the discussion, procedure, or question sections of the lab.

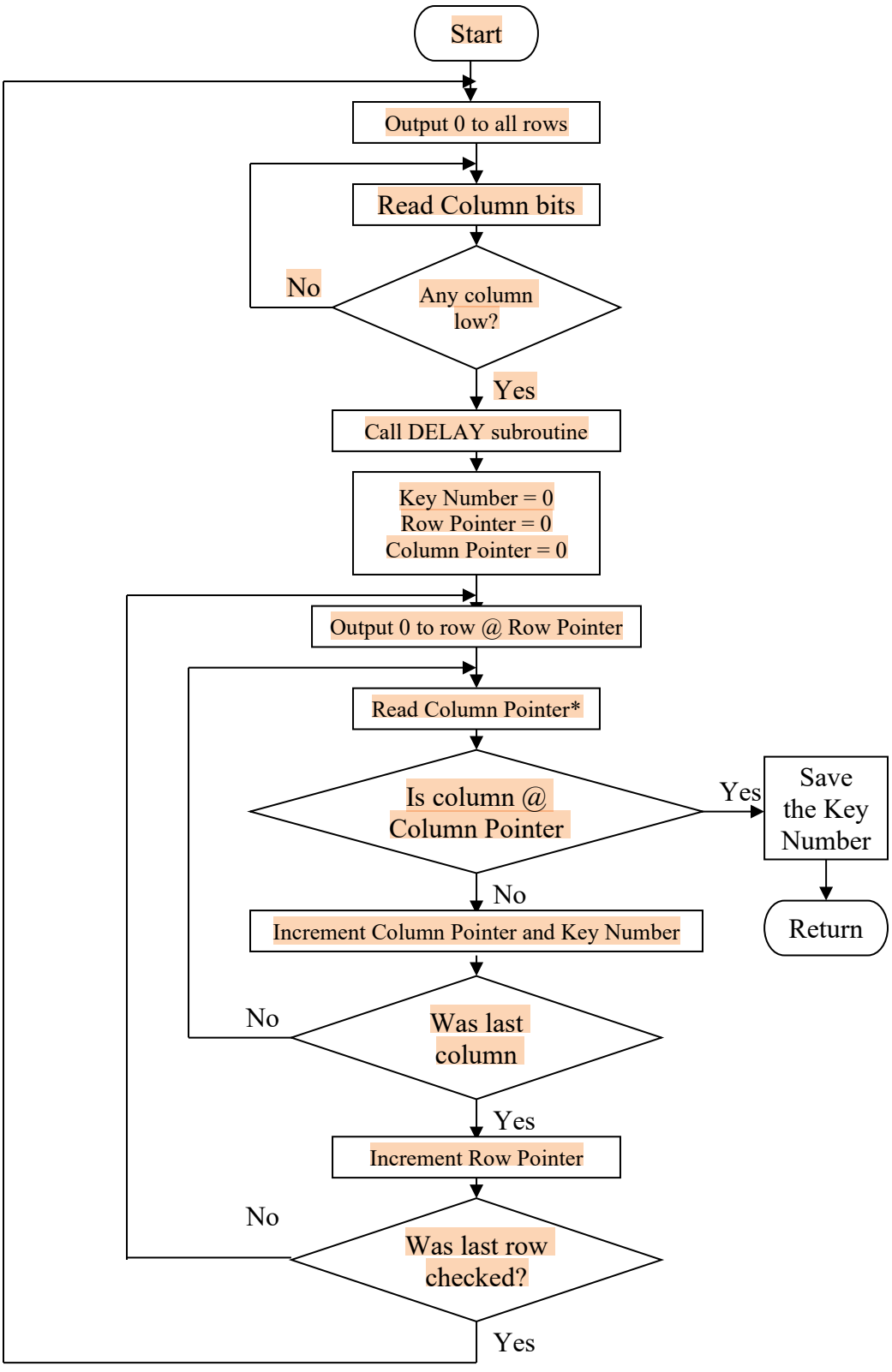
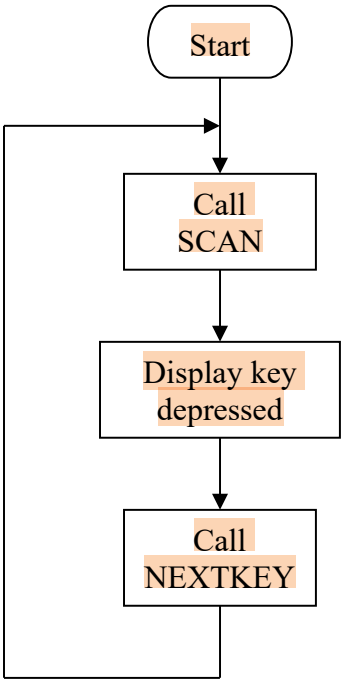
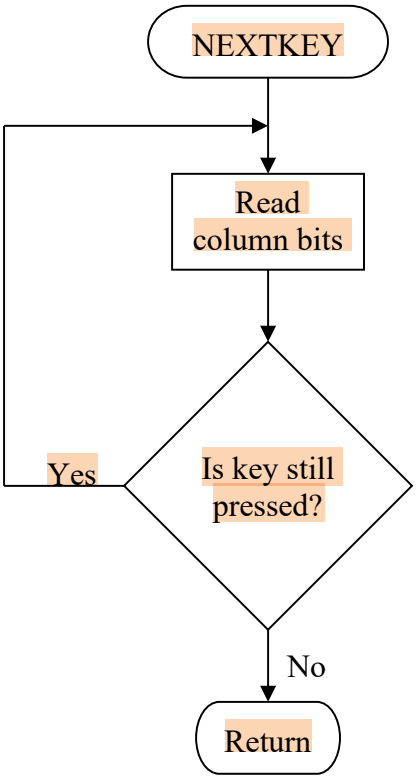


Figure 7.10: Flow chart for keypad SCAN subroutine.



Main program



NEXTKEY subroutine

7.11: Main program and NEXTKEY subroutine for Keypad scanning.