Microcomputers

General Purpose Input/Output and PIC24 Parallel I/O (PIO) Ports

General-purpose I/O

- The simplest type of I/O via the PIC24 µC external pins are parallel I/O (PIO) ports.

- A PIC24 µC can have multiple PIO ports named PORTA, PORTB, PORTC, PORTD, etc.
  - Each is 16-bits, and the number of PIO pins depends on the particular PIC24 µC and package.
  - The PIC24HJ128GP502/28 pin package has:
    - PORTA – bits RA4 through RA0
    - PORTB – bits RB15 through RB0
    - These are generically referred to as PORTx.

- Each pin on these ports can either be an input or output – the data direction is controlled by the corresponding bit in the TRISx registers (‘1’ = input, ‘0’ = output).

- The LATx register holds the last value written to PORTx.
Lecture 4-3 Electrical & Computer Engineering – Microcomputers

**PIC24HJ128GP502 Pin Diagram**

- 5 volt tolerant pin

---

Lecture 4-4 Electrical & Computer Engineering – Microcomputers

**PORTB Example**

Set the upper 8 bits of PORTB to outputs, lower 8 bits to be inputs:

```c
TRISB = 0x00FF;
```

Drive RB15, RB13 high; others low:

```c
PORTB = 0xA000;
```

Wait until input RB0 is high:

```c
while ((PORTB & 0x0001) == 0);
```

Wait until input RB3 is low:

```c
while ((PORTB & 0x0008) == 1);
```

Test returns true while RB0=0 so loop exits when RB0=1

Test returns true while RB3=1 so loop exits when RB3=0
**PORTB Example (cont.)**

Individual PORT bits are named as _RB0, _RB1, .._RA0, etc. so this can be used in C code.

Wait until input RB2 is high:

```c
while (_RB2 == 0);
```

Test returns true while RB2=0 so loop exits when RB2=1. Can also be written as:

```c
while (!_RB2);
```

Wait until input RB3 is low:

```c
while (_RB3 == 1);
```

Test returns true while RB3=1 so loop exits when RB3=0. Can also be written as:

```c
while (_RB3);
```

---

**Switch Input**

External pullup resistor

When switch is pressed RB3 reads as ‘0’, else reads as ‘1’.

If pullup is not present, then input would float (i.e. unconnected) when switch is not pressed, and input value may read as ‘0’ or ‘1’ because of system noise.
External pin shared with other on-chip modules
TRIS bit controls tristate control on output driver
Reading LATx reads last value written; reading PORTx reads the actual pin

TRIS bit set to enable output
Write to LAT/PORT
PORTx Pin Diagram (read from LAT)

Read from LAT

PORTx Pin Diagram (read from PORT)

Read from PORT
**LATx versus PORTx**

Writing LATx is the same as writing PORTx, both writes go to the latch.

Reading LATx reads the latch output (last value written), while reading PORTx reads the actual pin value.

**PIC24 µC**

Configure RB3 as an open-drain output, then write a ‘1’ to it.

The physical pin is tied to ground, so it can never go high.

Reading _RB3 returns a ‘0’, but reading _LATB3 returns a ‘1’ (the last value written).

**LATx versus PORTx (cont)**

```
_MACRO_ = 1;
_MACRO_ = 1;
```

Compiler

```
bset LATB,#0
bset LATB,#1
```

bitset/bitclr instructions are read/modify/write, in this case, read LATB, modify contents, write LATB. This works as expected.

```
_MACRO_ = 1;
_MACRO_ = 1;
```

Compiler

```
bset PORTB,#0
bset PORTB,#1
```

bitset/bclr instructions are read/modify/write – in this case, read PORTB, modify its contents, then write PORTB. Because of pin loading and fast internal clock speeds, the second bset may not work correctly! (see datasheet explanation). For this reason, our examples use LATx when writing to a pin.
Aside: Tri-State Buffer (TSB) Review

A tri-state buffer (TSB) has input, output, and output-enable (OE) pins. Output can either be ‘1’, ‘0’ or ‘Z’ (high impedance).

OE = 0, then switch closed
OE = 1, then switch open

Schmitt Trigger Input Buffer

Each PIO input has a Schmitt trigger input buffer; this transforms slowly rising/falling input transitions into sharp rising/falling transitions internally.

Schmitt Trigger (Vdd = 3.3 V)
PORTx Shared Pin Functions

External pins are shared with other on-chip modules. Just setting _TRISx = 1 may be not be enough to configure a PORTx pin as an input, depending on what other modules share the pin:

- RB15 shared with AN9, which is an analog input to the on-chip Analog-to-Digital Converter (ADC). Must disable analog functionality.

```
_PCFG9 = 1;  // Disables analog function
_TRISB15 = 1;  // Configure as input
```

```
_PCFG9 = 1;  // Disables analog function
_TRISB15 = 0;  // Configure as output
```

Analog/Digital Pin versus Digital-only Pin

- Pins with shared analog/digital functions have a maximum input voltage of $V_{dd} + 0.3$ V, so 3.6 V
- Pins with no analog functions (“digital-only” pins) are 5 V tolerant, their maximum input voltage is 5.6 V.
  - This is handy for receiving digital inputs from 5V parts.
- Most PIO pins can only source or sink a maximum 4 mA.
  - You may damage the output pin if you tie a load that tries to sink/source more than this current.
### Internal Weak Pullups

External pins with a CNy pin function have a weak internal pullup that can be enabled or disabled.

- Change notification input; to enable pullup:
  - CN11PUE = 1;
- To disable pullup:
  - CN11PUE = 0;

### Open Drain Outputs

Each PIO pin can be configured as an open drain output, which means the pullup transistor is disabled.

- _ODCxy = 1 enables open drain, _ODCxy = 0 disables open drain
- _ODCB15 = 1; Enables open drain on RB15
Port Configuration Macros

• For convenience, we supply macros/inline functions that hide pin configuration details:

```c
CONFIG_RB15_AS_DIG_OUTPUT();
CONFIG_RB15_AS_DIG_INPUT();
```

• These macros are supplied for each port pin.

• Because these functions change depending on the particular PIC24 µC, the `include/devices` directory has a include file for each PIC24 µC, and the correct file is included by the `include/pic24_ports.h` file.

Other Port Configuration Macros

• Other macros are provided for pull-up and open drain configuration:

```c
ENABLE_RB15_PULLUP();
DISABLE_RB15_PULLUP();
ENABLE_RB13_OPENDRAIN();
DISABLE_RB13_OPENDRAIN();
CONFIG_RB8_AS_DIG_OD_OUTPUT();
```

• General forms are:

```c
ENABLE_Rxy_PULLUP()
DISABLE_Rxy_PULLUP(),
ENABLE_Rxy_OPENDRAIN(),
DISABLE_Rxy_OPENDRAIN()
CONFIG_Rxy_AS_DIG_OD_OUTPUT()
```

• A port may not have a pull-up if it does not share the pin with a change notification input, in this case, the macro does not exist and you will get an error message when you try to compile the code.

Output + Open drain config in one macro
A red LED with a 470Ω resistor is connected to pin RA0.

What is the purpose of the resistor?

#include "pic24_all.h"

// A simple program
// that flashes an LED

#define CONFIG_LED1()  CONFIG_RA0_AS_DIG_OUTPUT()
#define LED1  _LATA0
//_LATA0 is port register for RA0

int main(void) {
  configClock(); //Configure clock source for processor
  /********** GPIO config **********/
  CONFIG_LED1();
  LED1 = 0;
  while (1) {
    DELAY_MS(250); //delay long enough to see LED blink
    LED1 = !LED1;  // Toggle LED
  } // end while (1)
}

Original file c:\microchip\chap8\ledflash.c
LED/Switch IO: Count number of press/releases

• **GOAL:** Design a program to count the number of press/release operations on a pushbutton attached to an input pin.
  
  - Toggle an LED attached to an output pin whenever a press/release operation is detected.
  - Assume the following hardware is used.

```c
#include "pic24_all.h"

/// LED
#define LED1
#define CONFIG_LED1() CONFIG_RB14_AS_DIG_OUTPUT()
#define LED1 _LATB14 //led1 state

/// Switch1 configuration
inline void CONFIG_SW1() {
    CONFIG_RB13_AS_DIG_INPUT(); //use RB13 for switch input
    ENABLE_RB13_PULLUP(); //enable the pullup
}
#define SW1 _RB13 //switch state
#define SW1_PRESSED() (SW1==0) //switch test
#define SW1_RELEASED() (SW1==1) //switch test
```

LED/Switch IO: Count number of press/releases

• I/O MACRO definitions
  
  - Use macros to isolate pin assignments for physical devices so that it is easy to change code if (WHEN!) the pin assignments change
LED/Switch IO: Count number of press/releases (naive, incorrect solution)

• I/O configuration and main() definitions

```c
int main (void) {
    configBasic(HELLO_MSG);
    /** GPIO config ************************************/
    CONFIG_SW1();    // configure switch
    CONFIG_LED1();   // configure LED
    DELAY_US(1);     // give pullups a little time
    LED1 = 0;        // LED off initially
    while (1) {
        if (SW1_PRESSED()) {
            // switch pressed
            // toggle LED1
            LED1 = !LED1; // toggle the LED
        }
    }
}
```

Incorrect, LED1 is toggled as long as pushbutton is pressed.

Approximately how many times would the LED toggle if you pressed the switch as fast as you could?

LED/Switch IO: Count number of press/releases (correct solution)

• I/O configuration and main() definitions

```c
int main (void) {
    configBasic(HELLO_MSG);
    /** GPIO config ************************************/
    CONFIG_SW1();    // configure switch
    CONFIG_LED1();   // configure LED
    DELAY_US(1);     // give pullups a little time
    LED1 = 0;        // LED off initially
    while (1) {
        // wait for press
        while (SW1_RELEASED()); // loop (1)
        DELAY_MS(15);          // debounce
        // wait for release
        while (SW1_PRESSED()); // loop (2)
        DELAY_MS(15);          // debounce
        LED1 = !LED1;          // toggle the LED
    }
}
```

Original file c:\microchip\chap8\ledtoggle_nofsm.c
**configBasic(HELLO_MSG);**

- Defined in c:\microchip\lib\common\pic24_util.c
  - Modified for use with Microstick II and putty terminal emulation program

```c
/** Perform basic chip configuration:
 * - Configure the heartbeat
 * - Configure the clock
 * - Configure UART1
 * - Determine and print the cause of reset
 * - Print a hello message.
 * 
 * \param sz_helloMsg Hello message to print.
 */
void configBasic(const char* sz_helloMsg) {
    configHeartbeat(); // Not really needed for the Microstick II
    configClock();
    configDefaultUART(9600); // Configure communication with putty on PC
    printResetCause(); // Output reset cause to UART
    outString(sz_helloMsg); // Output default hello message to UART
}
```

---

**Mechanical Switch Bounce**

Mechanical switches can ‘bounce’ (cause multiple transitions) when pressed.

- Scope shot of switch bounce
  - In this case, only bounced once, and settled in about ~500 microseconds
  - After detecting a switch state change, do not want to sample again until switch bounce has settled
  - A default value of 15 milliseconds is plenty of time
  - Do not want to wait too long; a human switch press is generally always > 50 ms in duration
Switch Sampling and Debounce

- Our new approach is **periodically sampling** the switch every ~15 ms in the while(1) loop.
- In the first solution, we were reading the switch as fast as the PIC24 could loop.
- We want this **sampling period** to be longer than any switch bounce settling time, and we want it to be short enough that we do not miss a switch press entirely.
  - A human switch press is at least greater than 50 ms, so 15 ms is short enough.

Software-Based Finite State Machines

- The previous problem solution was very specific to a given problem.
  - Would be hard to expand to more difficult problems using this design approach.
- One, more formal, approach to designing software is to use **finite state machines**. This approach:
  - Is applicable to a large variety of software projects.
  - Provides a fairly rigorous strategy that yields a simple and consistent code structure.
  - Can incorporate an effective debugging tool with minimal effort.
- Basic approach: Model actions in hardware/software as states.
  - Similar to the concept of hardware finite state machines.
Revisit the State Machine I/O Problem (with corresponding state diagram)

C Code Solution

```
#include "pic24_all.h"

/*
A program that uses a finite state machine approach for toggling an LED whenever
a pushbutton switch is pressed and released. Demonstrates the use of debounce
delays when polling a switch input.
*/

#include <pic24_all.h>

// LED1
#define CONFIG_LED1() CONFIG_RB14_AS_DIG_OUTPUT()
#define LED1 _LATB14

// Switch1 configuration
inline void CONFIG_SW1() {
    CONFIG_RB13_AS_DIG_INPUT();      // use RB13 for switch input
    ENABLE_RB13_PULLUP();            // enable the pullup
}

#define SW1              _RB13
#define SW1_PRESSED()   (SW1==0)
#define SW1_RELEASED()  (SW1==1)
```

Original file c:\microchip\chap8\ledtoggle.c
C Code Solution (continued)

typedef enum {
    STATE_RESET = 0,
    STATE_WAIT_FOR_PRESS,
    STATE_WAIT_FOR_RELEASE
} STATE;

STATE e_lastState = STATE_RESET; // print debug message for state when it changes
void printNewState (STATE e_currentState) {
    if (e_lastState != e_currentState) {
        switch (e_currentState) {
            case STATE_WAIT_FOR_PRESS:
                outString("STATE_WAIT_FOR_PRESS\n");
                break;
            case STATE_WAIT_FOR_RELEASE:
                outString("STATE_WAIT_FOR_RELEASE\n");
                break;
            default:
                outString("Unexpected state\n");
                break;
        }
        e_lastState = e_currentState; // remember last state
    }
}

C Code Solution (continued)

int main (void) {
    STATE e_mystate;
    // Set up heartbeat, UART, print hello message and diags
    configBasic(HELLO_MSG);

    /** GPIO config **************************/
    CONFIG_SW1(); //configure switch
    CONFIG_LED1(); //config the LED
    DELAY US(); //give pullups a little time

    /** Toggle LED each time switch is pressed and released ******************/
    e_mystate = STATE_WAIT_FOR_PRESS;
    State variable used for tracking the current state
    Initialize state variable to the first state
C Code Solution (continued)

while (1) {
    printNewState(e_mystate);  // print debug message when state changes
    switch (e_mystate) {
        case STATE_WAIT_FOR_PRESS:
            if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE;
            break;
        case STATE_WAIT_FOR_RELEASE:
            if (SW1_RELEASED()) {
                LED1 = !LED1;  // toggle LED
                e_mystate = STATE_WAIT_FOR_PRESS;
            }
            break;
        default:
            e_mystate = STATE_WAIT_FOR_PRESS;
    }  // end switch(e_mystate)
    DELAY_MS(DEBOUNCE_DLY);  // Debounce
    doHeartbeat();  // ensure we are alive, not used in Microstick II
}  // end while (1)

Sample output to putty (across serial port)

Finite state machine implemented with a switch statement within an infinite while() loop
A More Complex Problem

C Solution (configuration)

```c
#include "pic24_all.h"

/// LED1
#define CONFIG_LED1() CONFIG_RB14_AS_DIG_OUTPUT()
#define LED1 _LATB14 // led1 state

/// Switch1 configuration
inline void CONFIG_SW1() {
    CONFIG_RB13_AS_DIG_INPUT();       // use RB13 for switch input
    ENABLE_RB13_PULLUP();              // enable the pullup
}
#define SW1 _RB13 // switch state
#define SW1_PRESSED() (SW1==0) // switch test
#define SW1_RELEASED() (SW1==1) // switch test

/// Switch2 configuration
inline void CONFIG_SW2() {
    CONFIG_RB12_AS_DIG_INPUT();       // use RB12 for switch input
    ENABLE_RB12_PULLUP();              // enable the pullup
}
#define SW2 _RB12 // switch state
```

Original file c:\microchip\chap8\ledsw1.c
C Solution (Part 1)

typedef enum {  
    STATE_RESET = 0,  
    STATE_WAIT_FOR_PRESS1,  
    STATE_WAIT_FOR_RELEASE1,  
    STATE_WAIT_FOR_PRESS2,  
    STATE_WAIT_FOR_RELEASE2,  
    STATE_BLINK,  
    STATE_WAIT_FOR_RELEASE3  
} STATE;

int main (void) {  
    STATE e_mystate;

    configBasic(HELLO_MSG);

    /** GPIO config *******************/  
    CONFIG_SW1();  // configure switch
    CONFIG_SW2();  // configure switch
    CONFIG_LED1();  // configure the LED
    DELAY_US(1);  // give pullups a little time

    /*** Toggle LED each time switch is pressed and released *************/
    e_mystate = STATE_WAIT_FOR_PRESS1;

    while (1) {  
        printNewState(e_mystate);  // debug message when state changes
        switch (e_mystate) {  
            case STATE_WAIT_FOR_PRESS1:  
                LED1 = 0;  // turn off the LED
                if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE1;
                break;
            case STATE_WAIT_FOR_RELEASE1:  
                if (SW1_RELEASED()) e_mystate = STATE_WAIT_FOR_PRESS2;
                break;
            case STATE_WAIT_FOR_PRESS2:  
                LED1 = 1;  // turn on the LED
                if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE2;
                break;
            case STATE_WAIT_FOR_RELEASE2:  
                if (SW1_RELEASED()) {  
                    // decide where to go
                    if (SW2) e_mystate = STATE_BLINK;
                    else e_mystate = STATE_WAIT_FOR_PRESS1;
                }
                break;
        }
    }
}

C Solution (Part 2)
## C Solution (Part 3)

```
C Solution (Part 3)

```case STATE_BLINK:
    LED1 = !LED1;  // Blink while not pressed
    DELAY_MS(100);  // Blink delay
    if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE3;
    break;
  case STATE_WAIT_FOR_RELEASE3:
    LED1 = 1;     // Freeze LED1 at 1
    if (SW1_RELEASED()) e_mystate = STATE_WAIT_FOR_PRESS1;
    break;
default:
    e_mystate = STATE_WAIT_FOR_PRESS1;
    }  // end switch(e_mystate)
  DELAY_MS(15); // Debounce
}  // end while (1)
```

---

## Console Output for LED/SW Problem

```
Console Output for LED/SW Problem

SW2 on for the first test
SW2 off for the second test
```

---
What do you have to know?

- GPIO port usage of PORTA, PORTB
- How to use the weak pullups of PORTB
- Definition of Schmitt Trigger
- How a Tri-state buffer works
- How an open-drain output works and what it is useful for
- How to write C code for finite state machine description of LED/Switch IO