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 PIC24HJ32GP202/28 pin package has:

PORTA – bits RA4 through RA0
PORTB – bits RB15 through RB0

These are generically referred to as PORT*x*.

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

The LAT*x* register holds the last value written to PORT*x*.

PORTB Example

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

```
TRISB = 0x00FF;
```

Drive RB15, RB13 high;
others low:

```
PORTB = 0xA000;
```

Wait until input RB0 is high:

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

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

Wait until input RB3 is low:

Test returns true while RB3=1
so loop exited when RB3=0

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

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:

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

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

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

Wait until input RB3 is low:

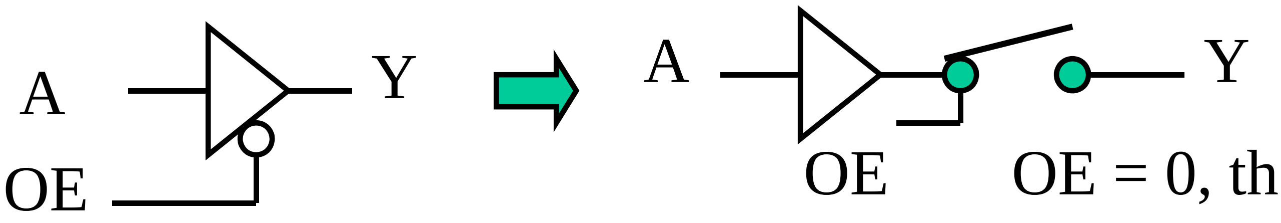
```
while (_RB3 == 1) ;
```

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

```
while (_RB3);
```

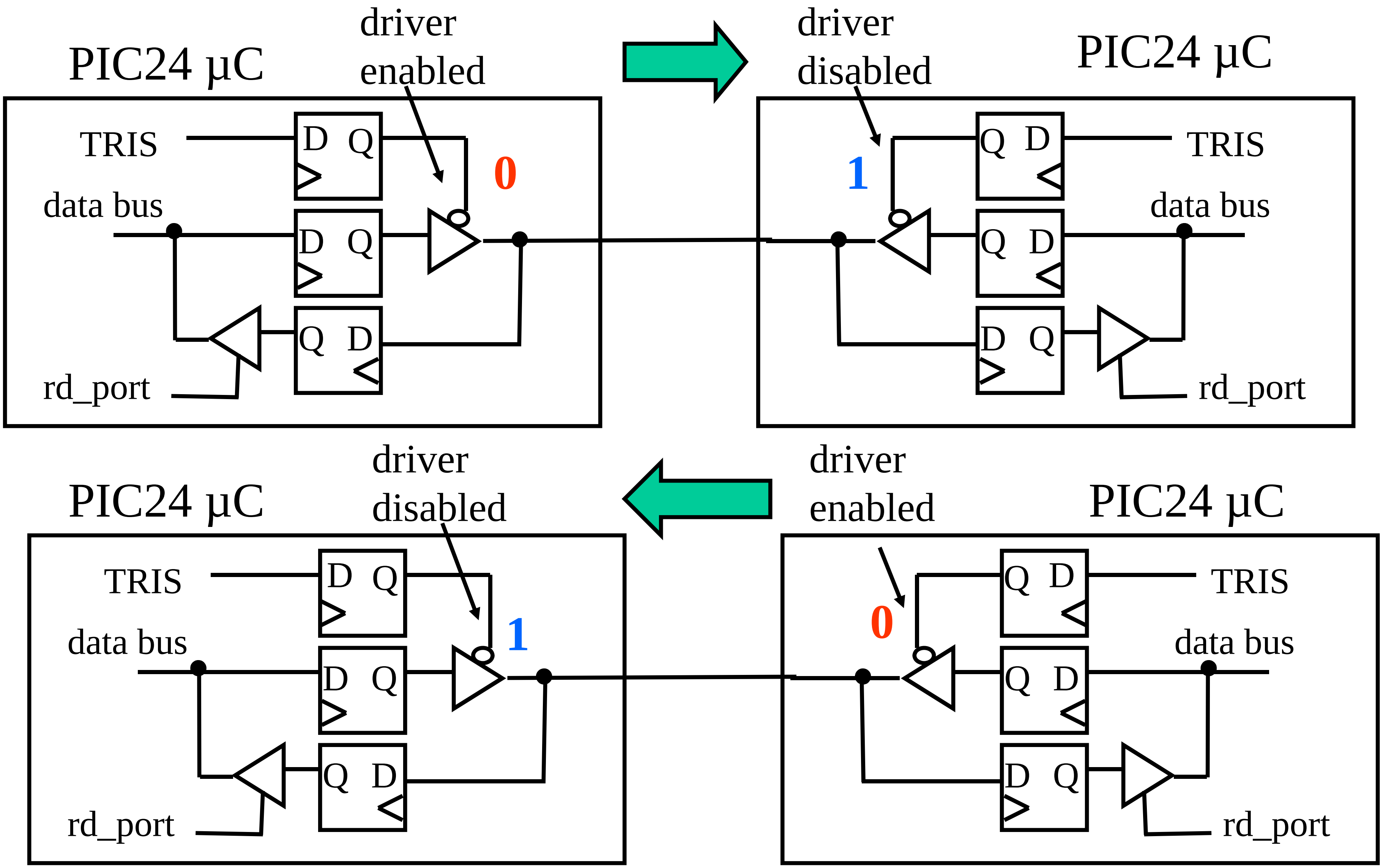
Aside: Tri-State Buffer (TSB) Review

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

Bi-directional, Half-duplex Communication

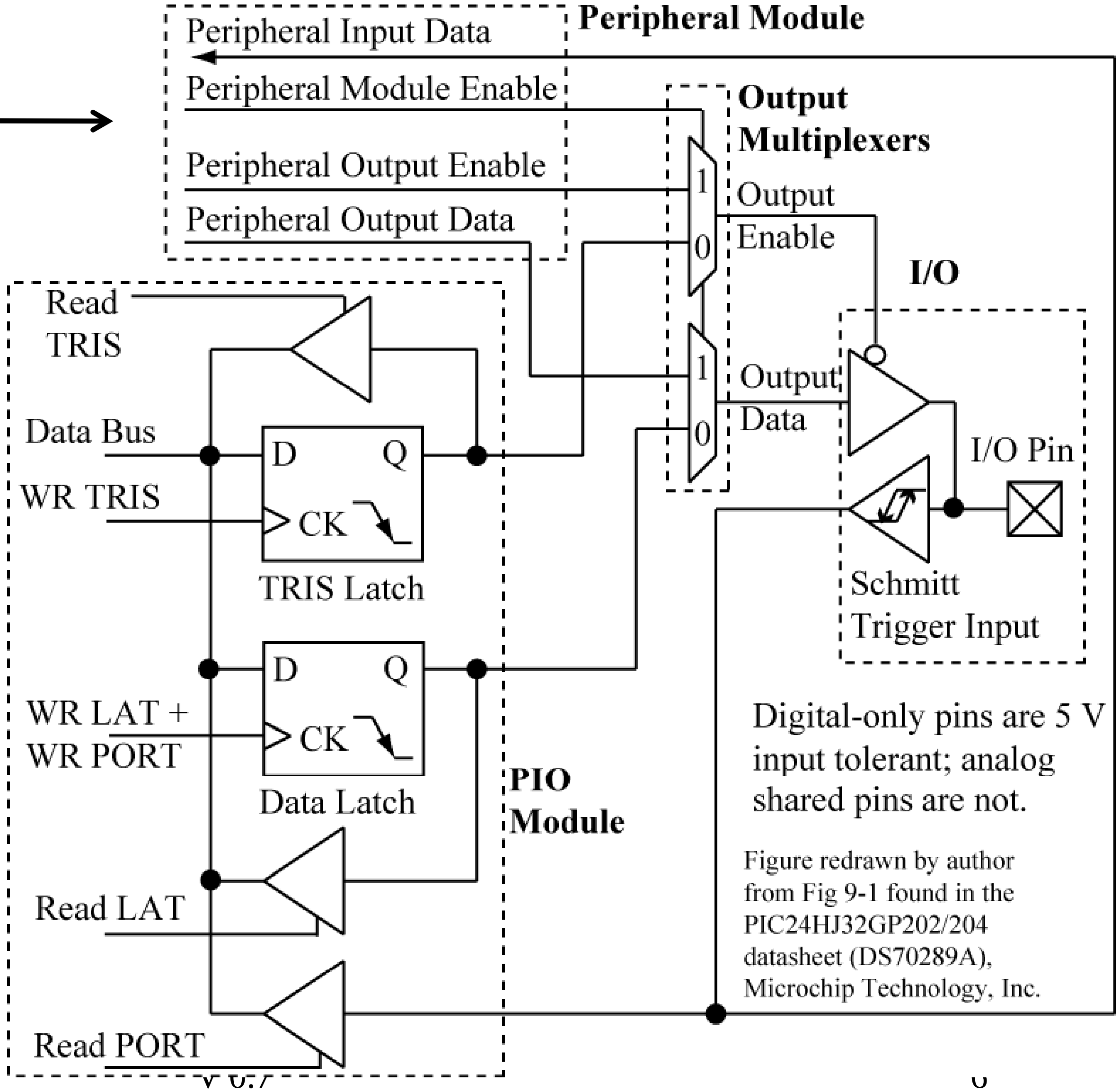


PORTx Pin Diagram

External pin shared with other on-chip modules →

TRIS bit controls tristate control on output driver →

Reading LAT*x* reads last value written; reading PORT*x* reads the actual pin



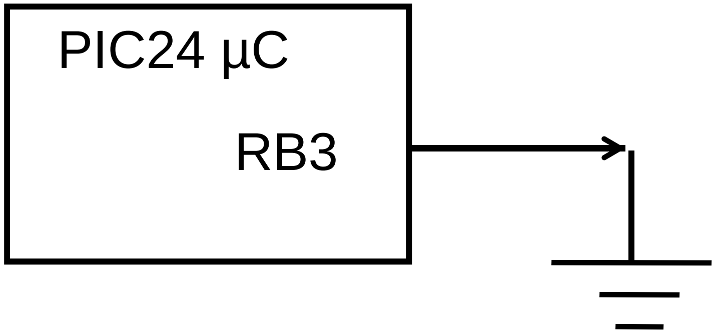
Digital-only pins are 5 V input tolerant; analog shared pins are not.

Figure redrawn by author from Fig 9-1 found in the PIC24HJ32GP202/204 datasheet (DS70289A), Microchip Technology, Inc.

LATx versus PORTx

Writing LAT*x* is the same as writing PORT*x*, both writes go to the latch.

Reading LAT*x* reads the latch output (last value written), while reading PORT*x* reads the actual pin value.



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)

```
_LATB0 = 1;
_LATB0 = 0;
```

Compiler →

```
bset LATB,#0
bclr LATB,#0
```

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

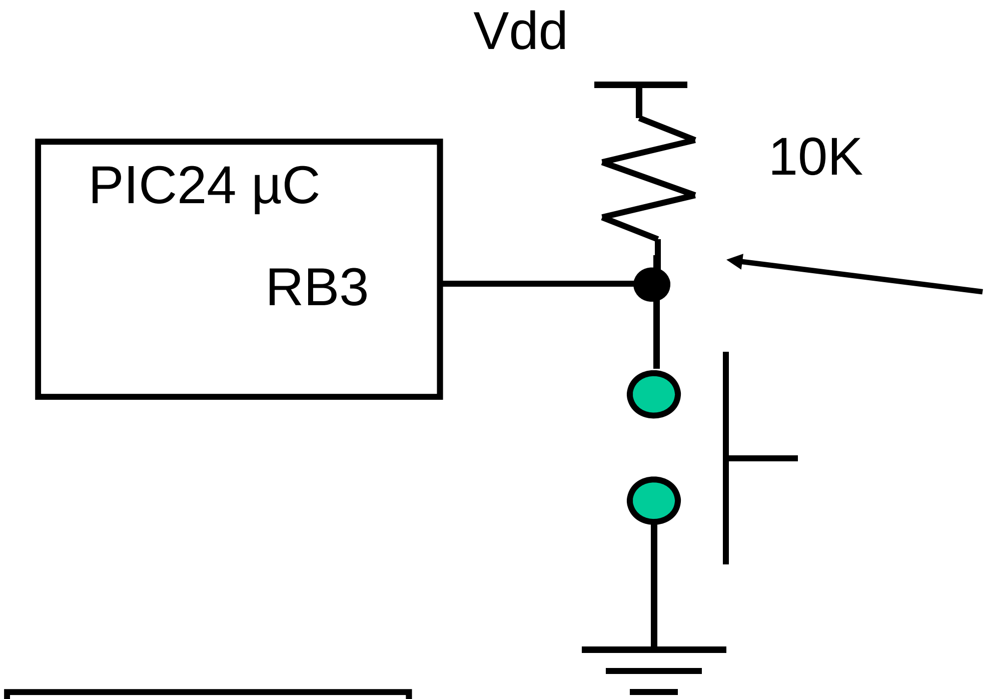
```
_RB0 = 0;
_RB0 = 1;
```

Compiler →

```
bclr PORTB,#0
bset PORTB,#0
```

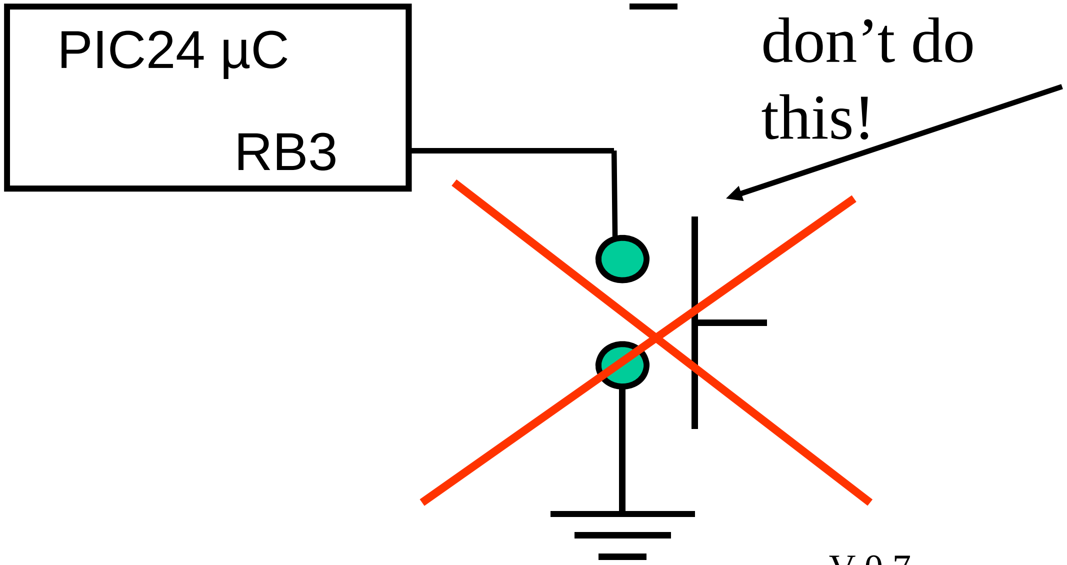
bset/bclr instructions are read/modify/write – in this case, read PORTB, modify its contents, then write PORTB. Because of pin loading and fast internal clock speeds, the second bset may not work correctly! (see datasheet explanation). For this reason, our examples use LATx when writing to a pin.

Switch Input



External pullup

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

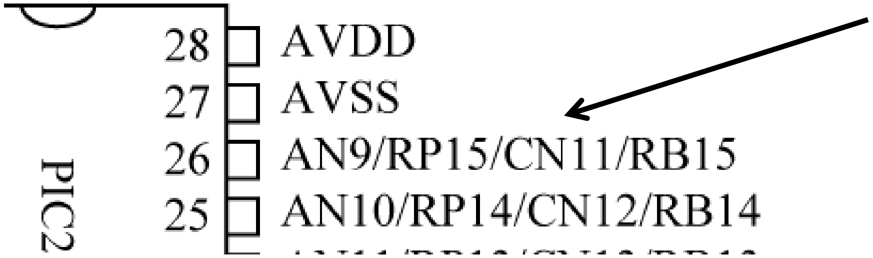


don’t do this!

If pullup not present, then input would float when switch is not pressed, and input value may read as ‘0’ or ‘1’ because of system noise.

Internal Weak Pullups

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

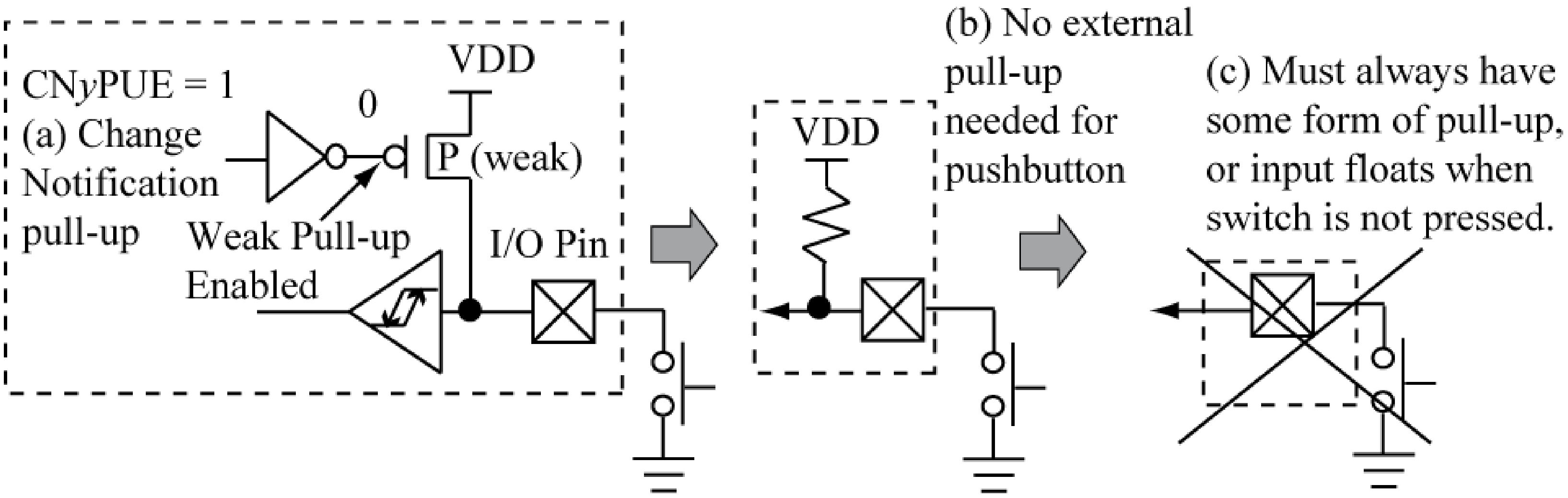


Change notification input; to enable pullup:

CN11PUE = 1;

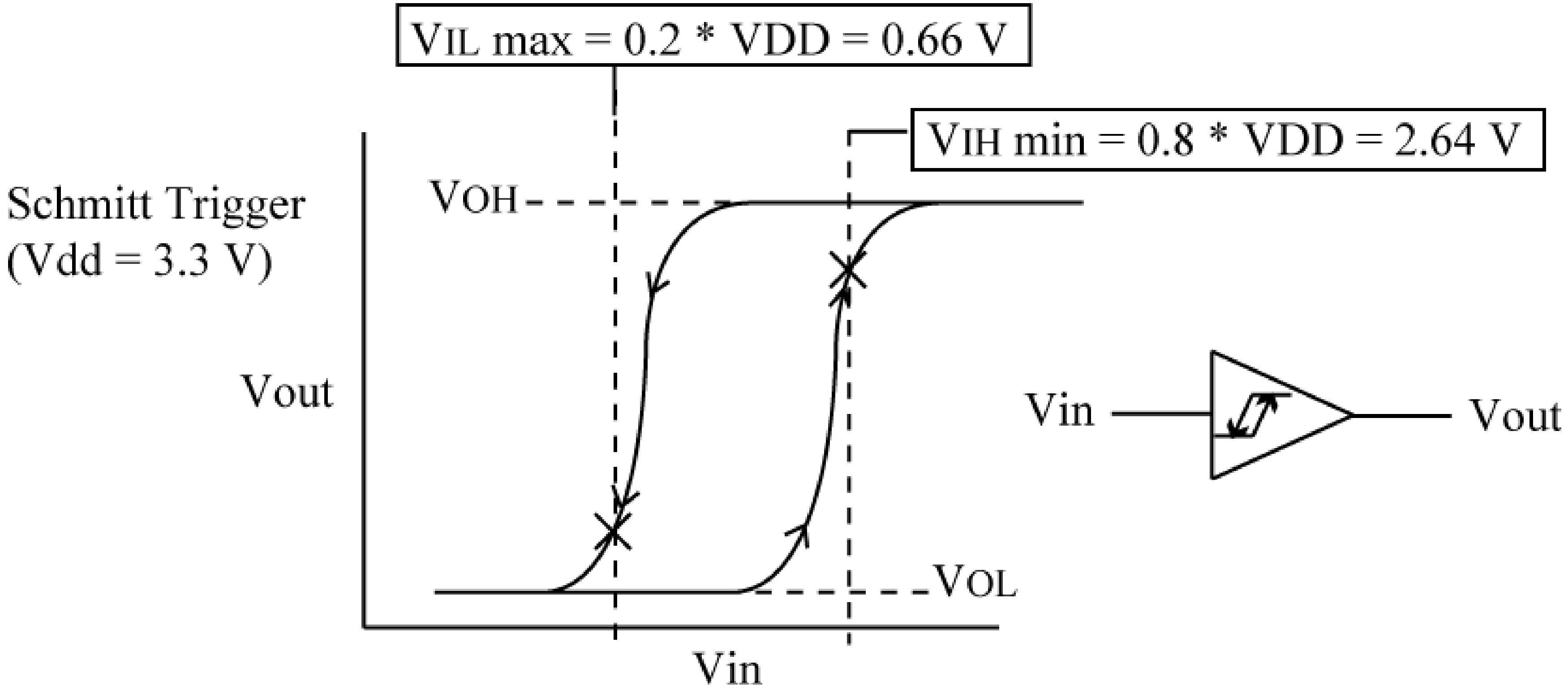
To disable pullup:

CN11PUE = 0;


CNyPUE = 1
(a) Change Notification pull-up
0
VDD
P (weak)
Weak Pull-up Enabled
I/O Pin
VDD
(b) No external pull-up needed for pushbutton
(c) Must always have some form of pull-up, or input floats when switch is not pressed.

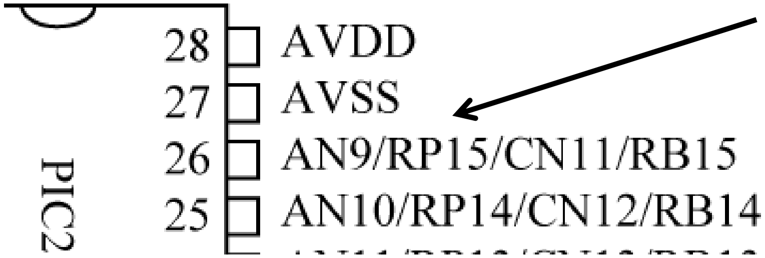
Schmitt Trigger Input Buffer

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



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 PORT*x* 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 Vdd + 0.3 V, so 3.6 V

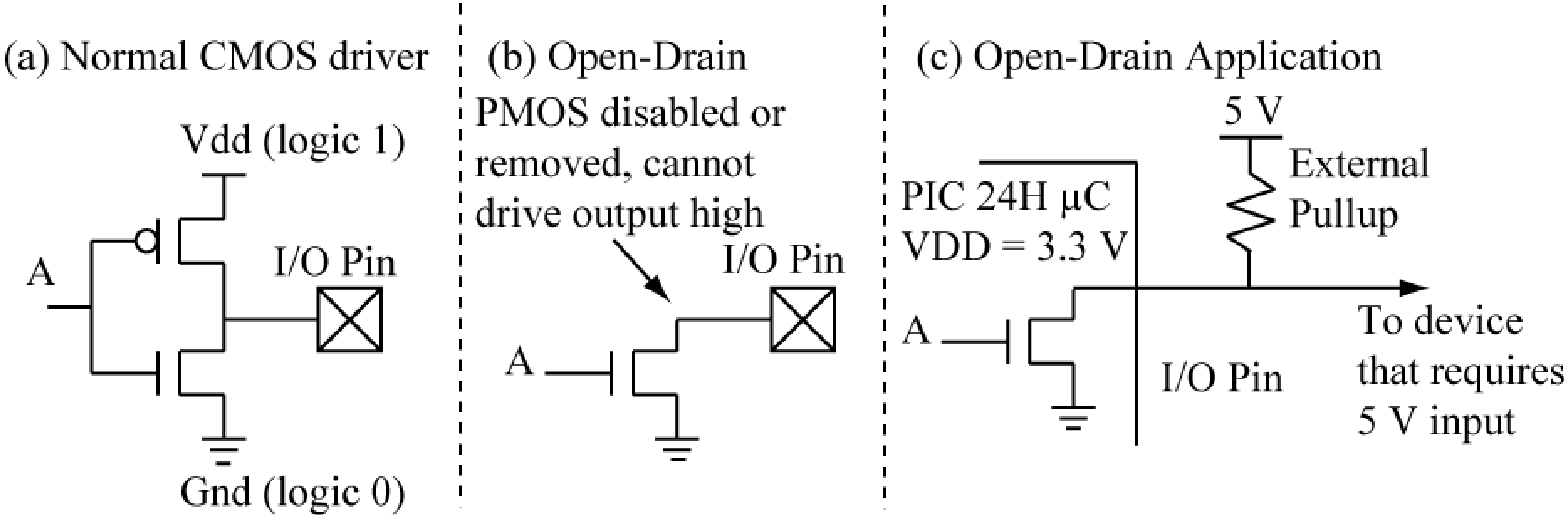
Pins with no analog functions (“digital-only” pins) are 5 V tolerant, their maximum input voltage is 5.6 V.

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.

Open Drain Outputs

Some PIO pins can be configured as an *open drain* output, which means the pullup transistor is disabled.

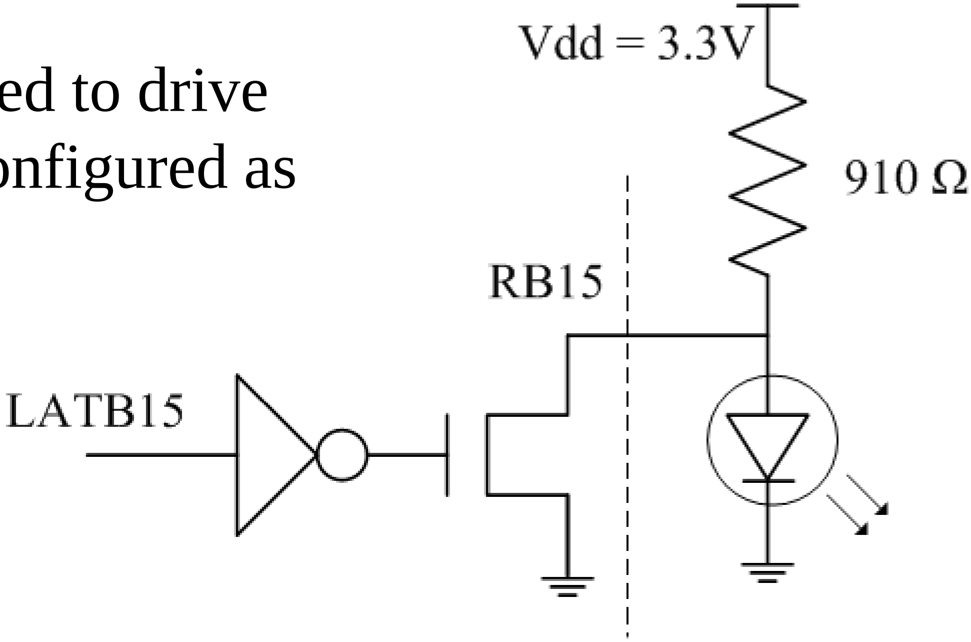


_ODC*xy* = 1 enables open drain, _ODC*xy* = 0 disables open drain

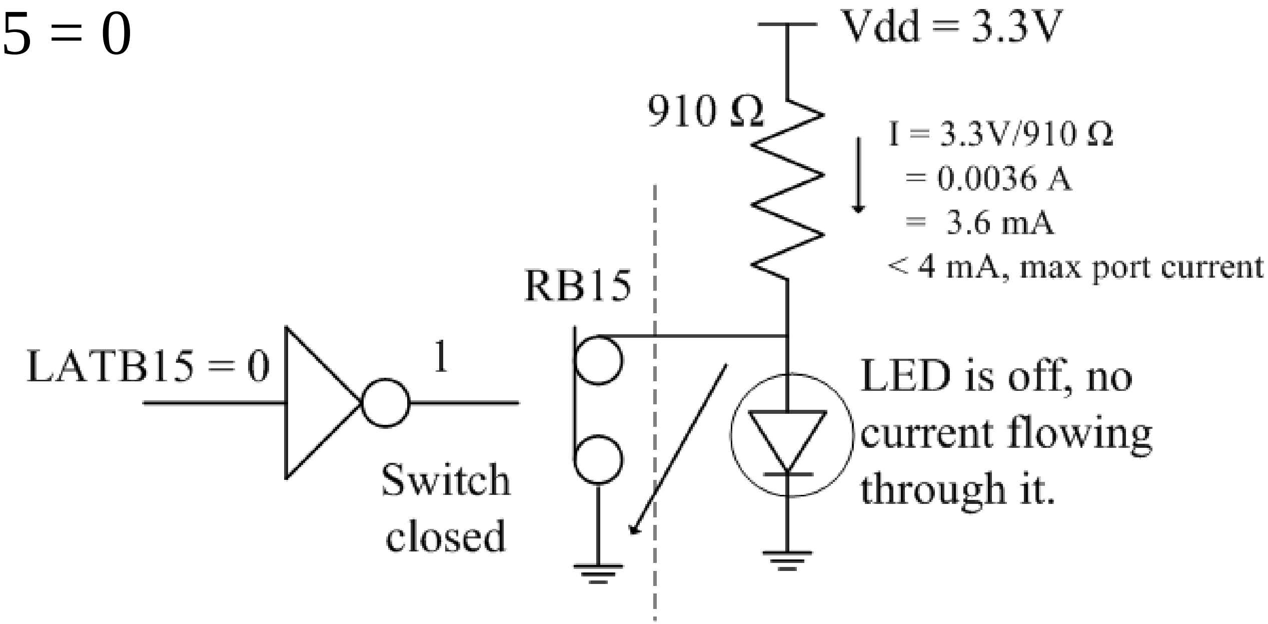
```
_ODCB15 = 1;
```

← Enables open drain on RB15

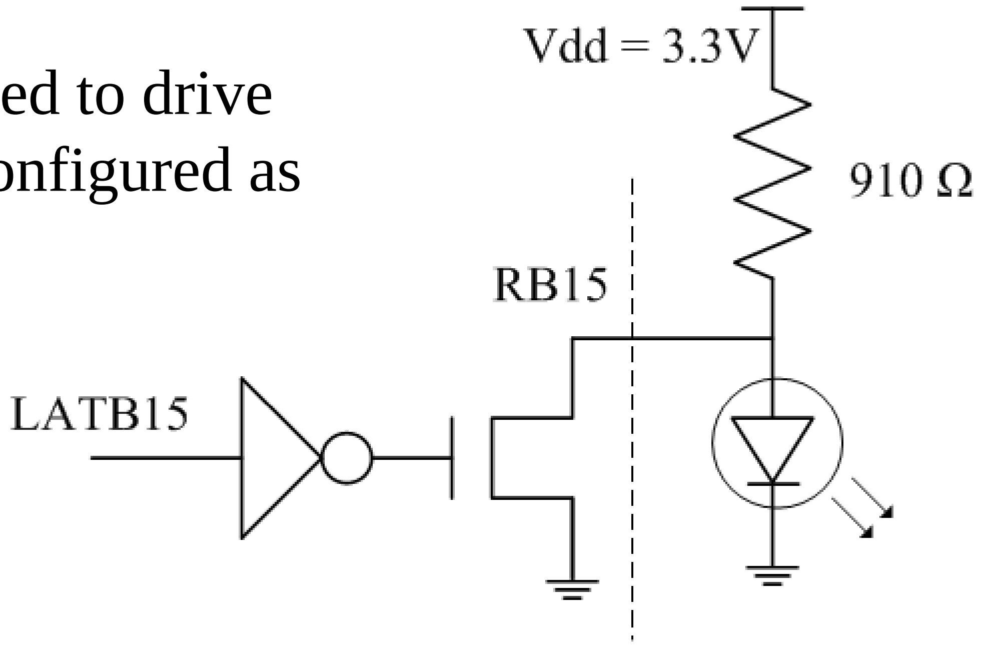
Recall that RB15 port used to drive heartbeat/Power LED, configured as open drain



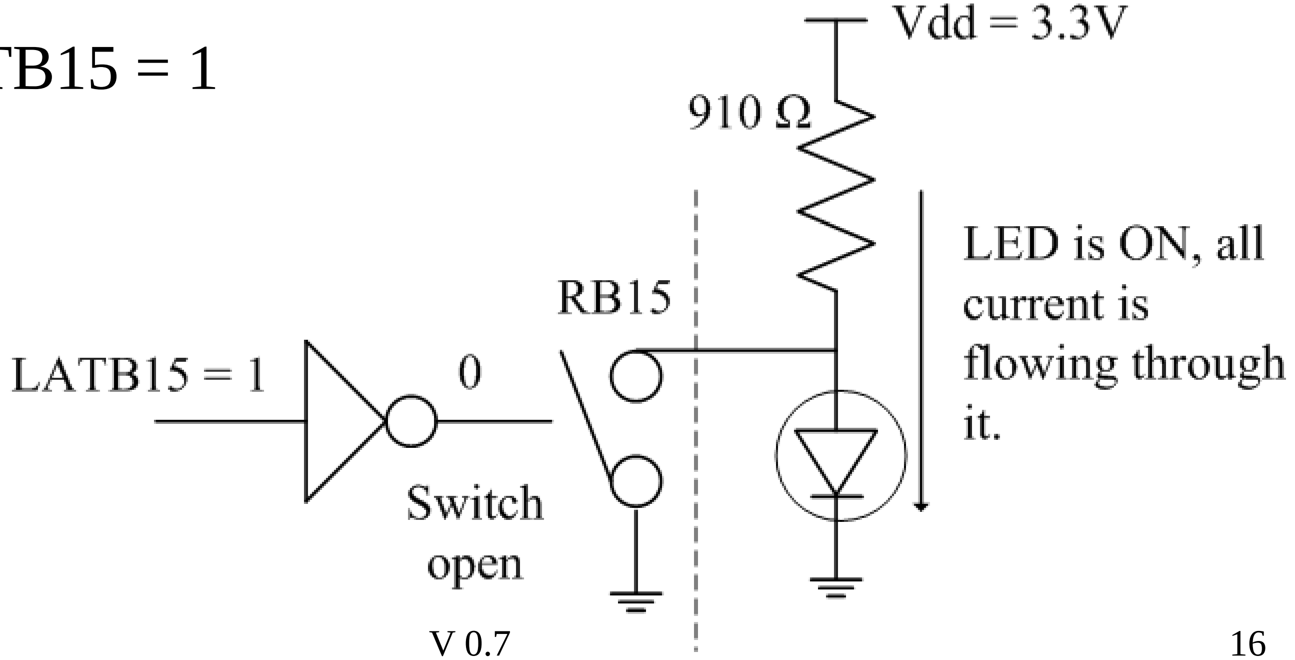
LED is off if LATB15 = 0



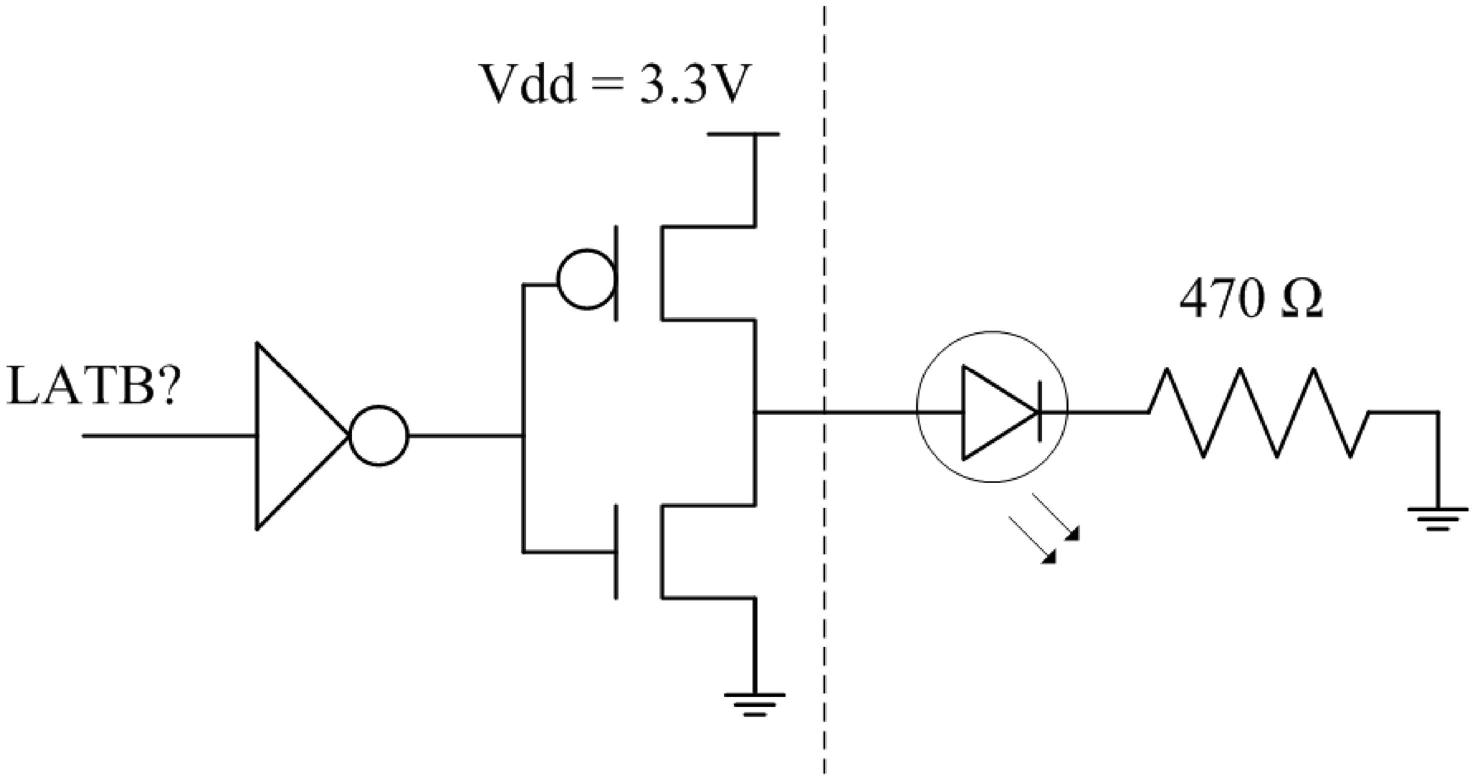
Recall that RB15 port used to drive heartbeat/Power LED, configured as open drain



LED is on if LATB15 = 1



Driving LEDs : port configured as a normal CMOS driver



RB15/open drain configuration for heartbeat LED is a special case.

Port Configuration Macros

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

```
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:

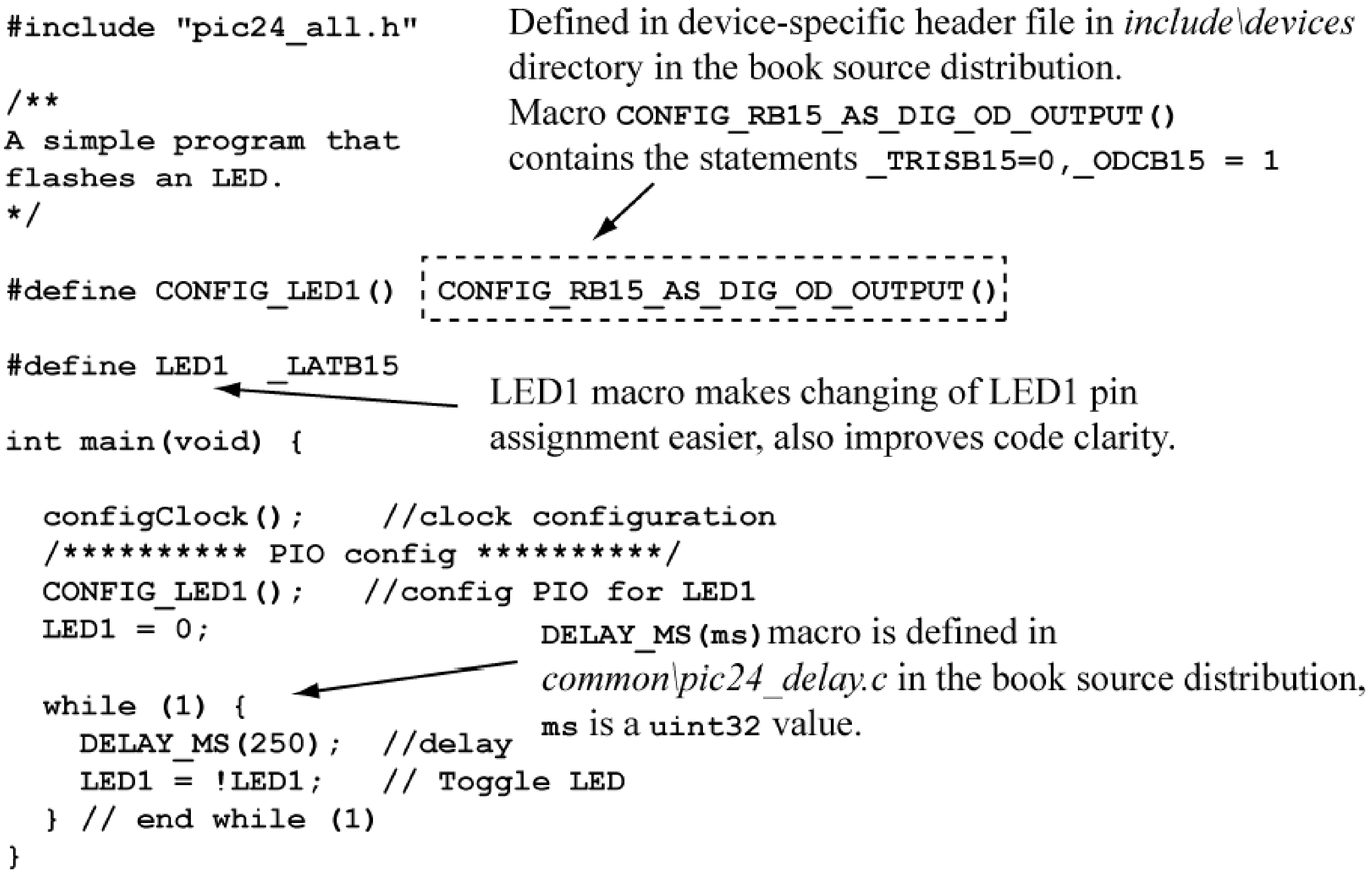
```
ENABLE_RB15_PULLUP();
DISABLE_RB15_PULLUP();
ENABLE_RB13_OPENDRAIN();
DISABLE_RB13_OPENDRAIN();
CONFIG_RB8_AS_DIG_OD_OUTPUT();
```

Output + Open drain config in one macro

General forms are ENABLE_R*xy*_PULLUP(), DISABLE_R*xy*_PULLUP(), ENABLE_R*xy*_OPENDRAIN(), DISABLE_R*xy*_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.

ledflash.c Revisited



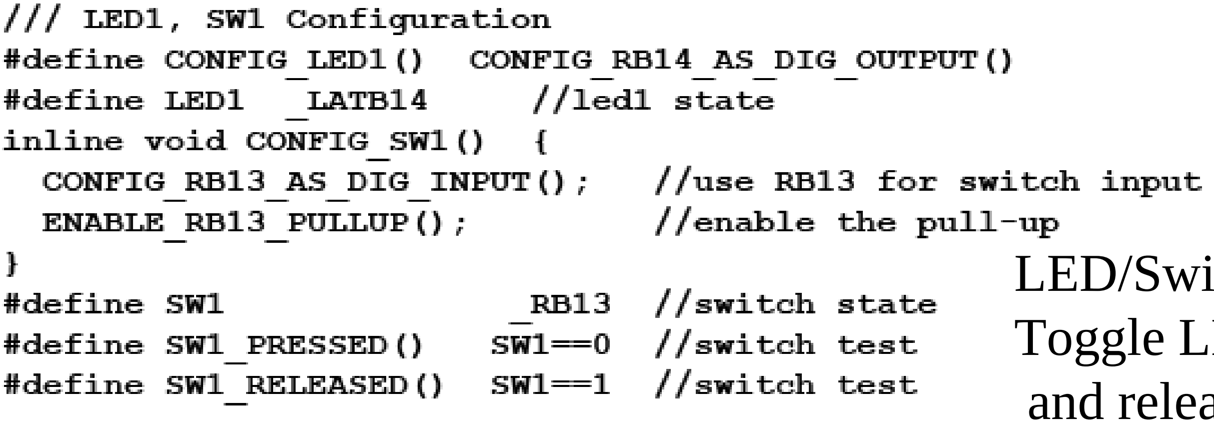
```
#include "pic24_all.h"

/**
A simple program that
flashes an LED.
*/

#define CONFIG_LED1() CONFIG_RB15_AS_DIG_OD_OUTPUT()

#define LED1   _LATB15

int main(void) {

  configClock();    //clock configuration
  /********** PIO config **********/
  CONFIG_LED1();   //config PIO for LED1
  LED1 = 0;

  while (1) {
    DELAY_MS(250);  //delay
    LED1 = !LED1;   // Toggle LED
  } // end while (1)
}
```

Defined in device-specific header file in *include\devices* directory in the book source distribution.
Macro `CONFIG_RB15_AS_DIG_OD_OUTPUT()` contains the statements `_TRISB15=0,_ODCB15 = 1`

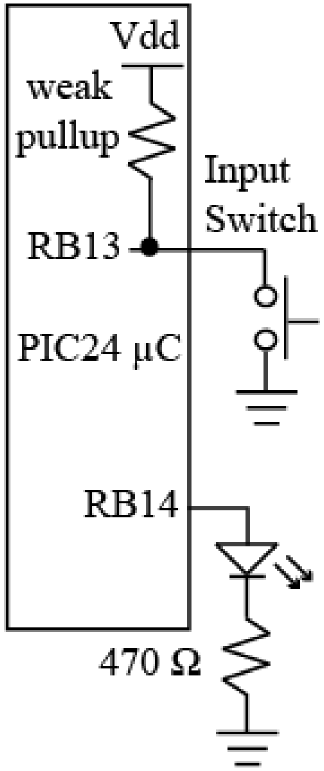
LED1 macro makes changing of LED1 pin assignment easier, also improves code clarity.

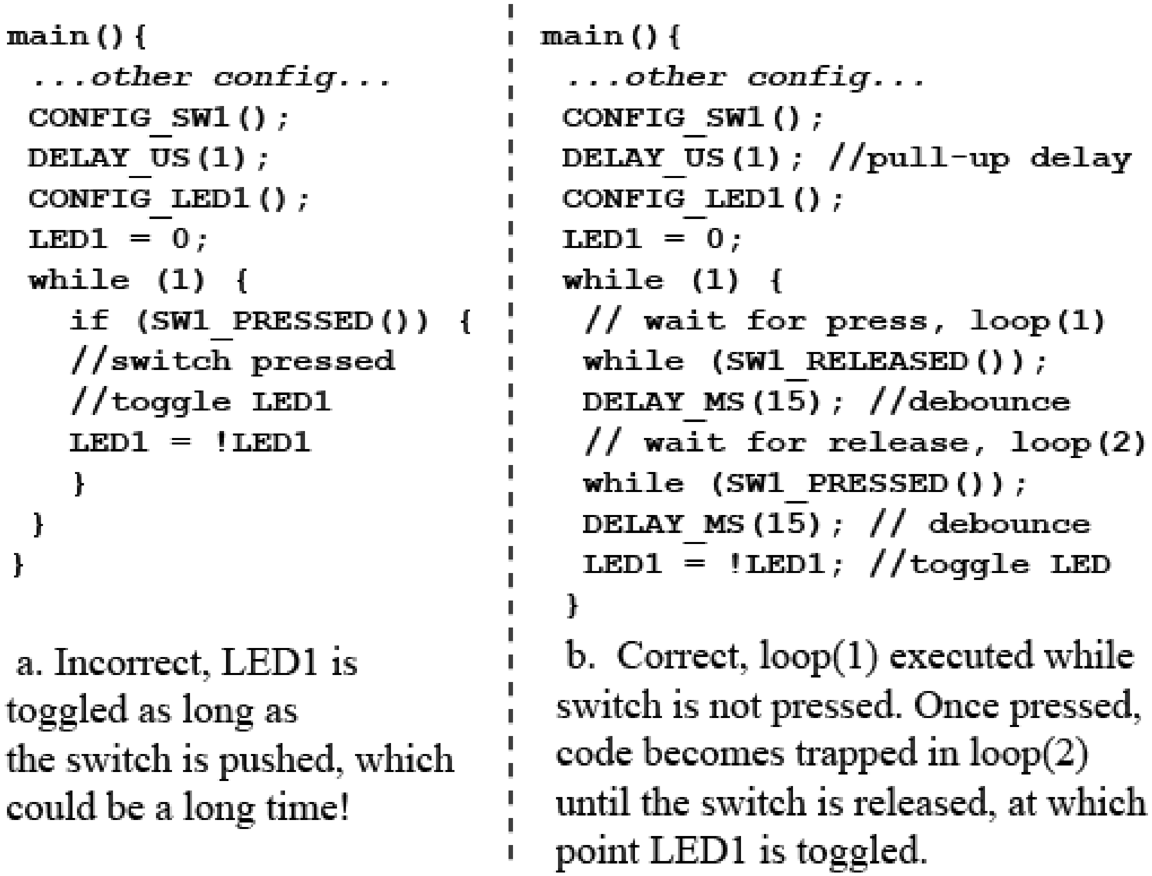
`DELAY_MS(ms)` macro is defined in *common\pic24_delay.c* in the book source distribution, `ms` is a `uint32` value.

LED/Switch IO: Toggle LED on each press and release



```
/// LED1, SW1 Configuration
#define CONFIG_LED1()  CONFIG_RB14_AS_DIG_OUTPUT()
#define LED1   _LATB14      //led1 state
inline void CONFIG_SW1()  {
  CONFIG_RB13_AS_DIG_INPUT();   //use RB13 for switch input
  ENABLE_RB13_PULLUP();         //enable the pull-up
}
#define SW1                _RB13  //switch state
#define SW1_PRESSED()    SW1==0  //switch test
#define SW1_RELEASED()   SW1==1  //switch test
```





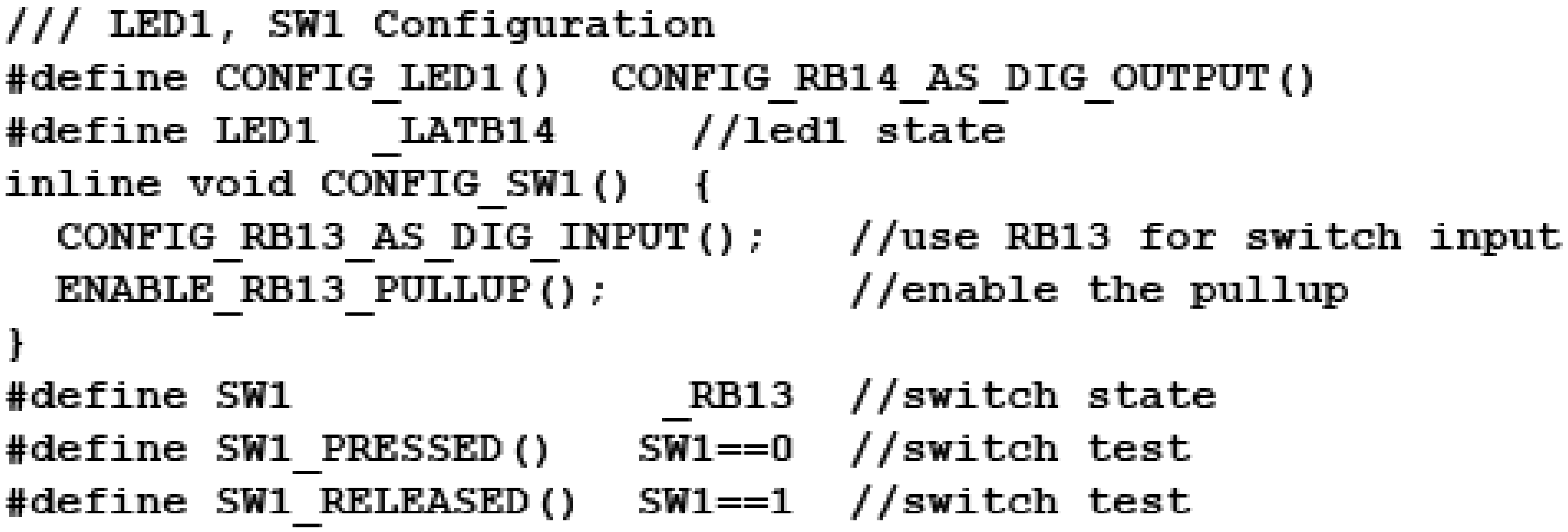
```
main(){
 ...other config...
 CONFIG_SW1();
 DELAY_US(1);
 CONFIG_LED1();
 LED1 = 0;
 while (1) {
   if (SW1_PRESSED()) {
   //switch pressed
   //toggle LED1
   LED1 = !LED1
   }
 }
}
```

a. Incorrect, LED1 is toggled as long as the switch is pushed, which could be a long time!

```
main(){
 ...other config...
 CONFIG_SW1();
 DELAY_US(1); //pull-up delay
 CONFIG_LED1();
 LED1 = 0;
 while (1) {
  // wait for press, loop(1)
  while (SW1_RELEASED());
  DELAY_MS(15); //debounce
  // wait for release, loop(2)
  while (SW1_PRESSED());
  DELAY_MS(15); // debounce
  LED1 = !LED1; //toggle LED
 }
```

b. Correct, loop(1) executed while switch is not pressed. Once pressed, code becomes trapped in loop(2) until the switch is released, at which point LED1 is toggled.

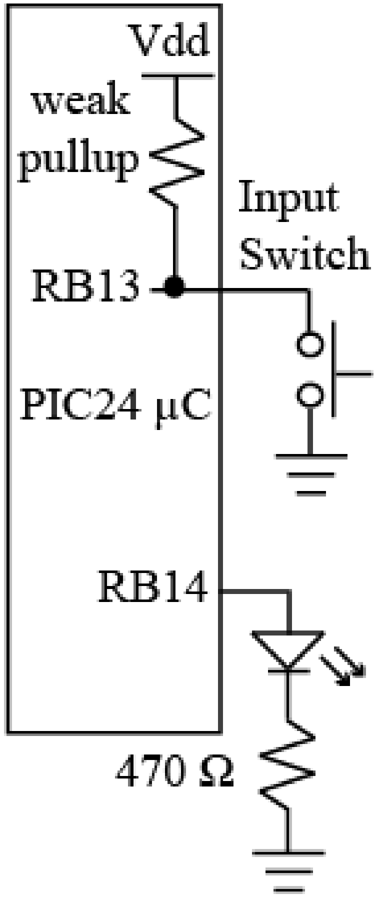
I/O Configuration



```
/// LED1, SW1 Configuration
#define CONFIG_LED1()  CONFIG_RB14_AS_DIG_OUTPUT()
#define LED1  _LATB14     //led1 state
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
```

Use macros to isolate pin assignments for physical devices so that it is easy to change code if (WHEN!) the pin assignments change!

Toggling for each press/release



```
main(){
 ...other config...
 CONFIG_SW1();
 DELAY_US(1);
 CONFIG_LED1();
 LED1 = 0;
 while (1) {
   if (SW1_PRESSED()) {
   //switch pressed
   //toggle LED1
   LED1 = !LED1
   }
 }
}
```

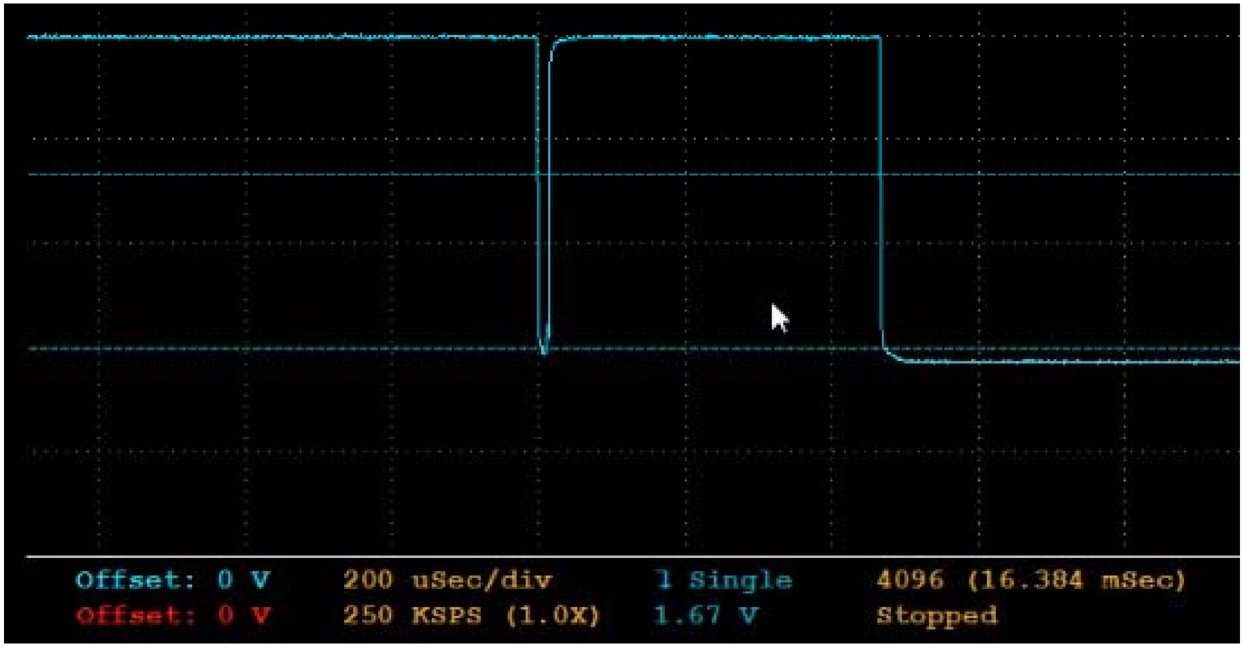
a. Incorrect, LED1 is toggled as long as the switch is pushed, which could be a long time!

```
main(){
 ...other config...
 CONFIG_SW1();
 DELAY_US(1); //pull-up delay
 CONFIG_LED1();
 LED1 = 0;
 while (1) {
  // wait for press, loop(1)
  while (SW1_RELEASED());
  DELAY_MS(15); //debounce
  // wait for release, loop(2)
  while (SW1_PRESSED());
  DELAY_MS(15); // debounce
  LED1 = !LED1; //toggle LED
 }
```

b. Correct, loop(1) executed while switch is not pressed. Once pressed, code becomes trapped in loop(2) until the switch is released, at which point LED1 is toggled.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Mechanical Switch Bounce

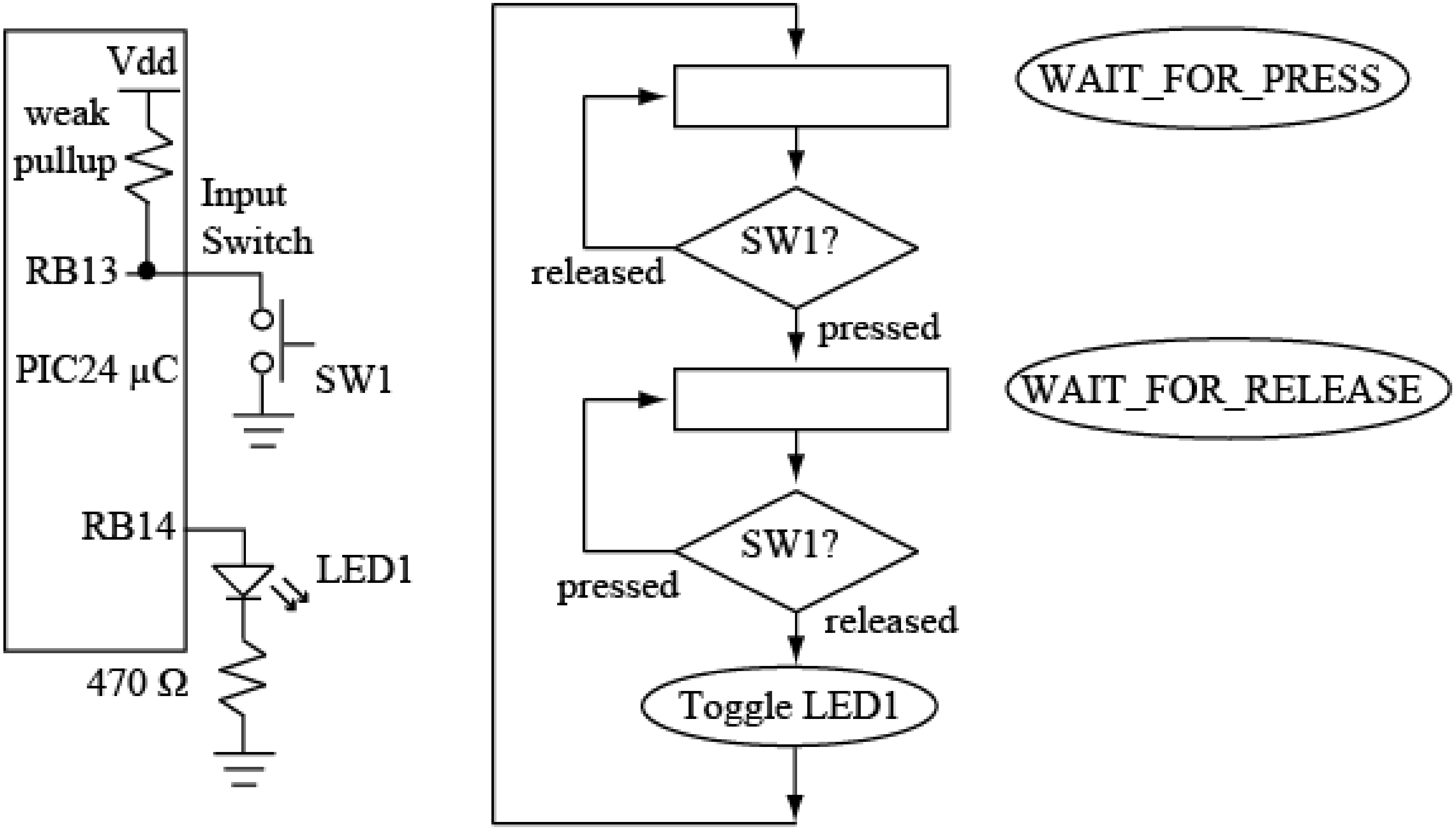


```
Offset: 0 V    200 uSec/div      1 Single   4096 (16.384 mSec)
Offset: 0 V    250 KSPS (1.0X)   1.67 V     Stopped
```

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

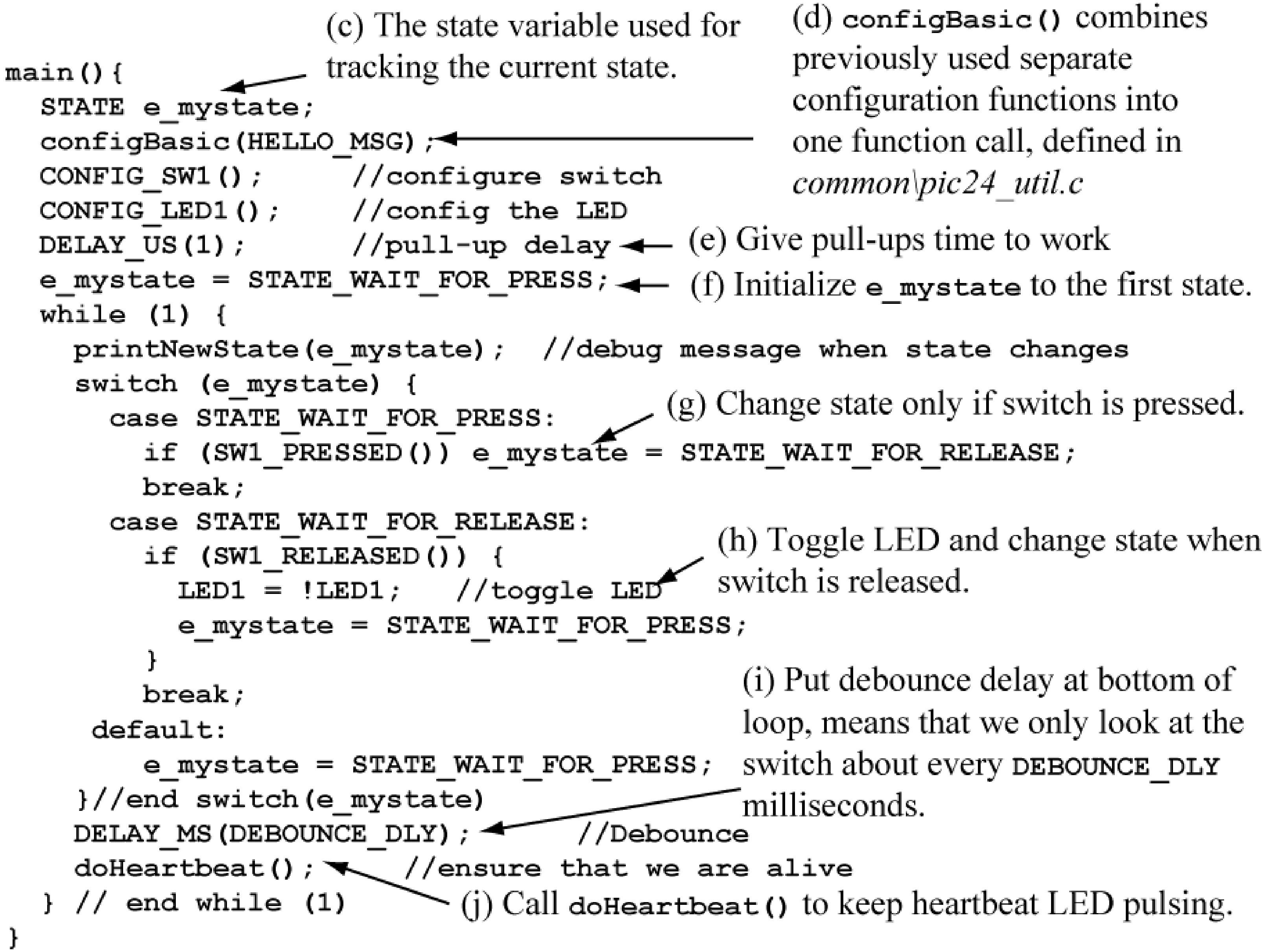
Scope shot of switch bounce; in this case, only bounced once, and settled in about ~500 microseconds.   After detecting a switch state change, do not want to sample again until switch bounce has settled. Our default value of 15 milliseconds is plenty of time. Do not want to wait too long; a human switch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

State Machine I/O



From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

C Code Solution



```
main(){
  STATE e_mystate;
  configBasic(HELLO_MSG);
  CONFIG_SW1();       //configure switch
  CONFIG_LED1();      //config the LED
  DELAY_US(1);        //pull-up delay
  e_mystate = STATE_WAIT_FOR_PRESS;
  while (1) {
    printNewState(e_mystate);  //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 that we are alive
  } // end while (1)
}
```

(c) The state variable used for tracking the current state.

(d) `configBasic()` combines previously used separate configuration functions into one function call, defined in *common\pic24_util.c*

(e) Give pull-ups time to work

(f) Initialize `e_mystate` to the first state.

(g) Change state only if switch is pressed.

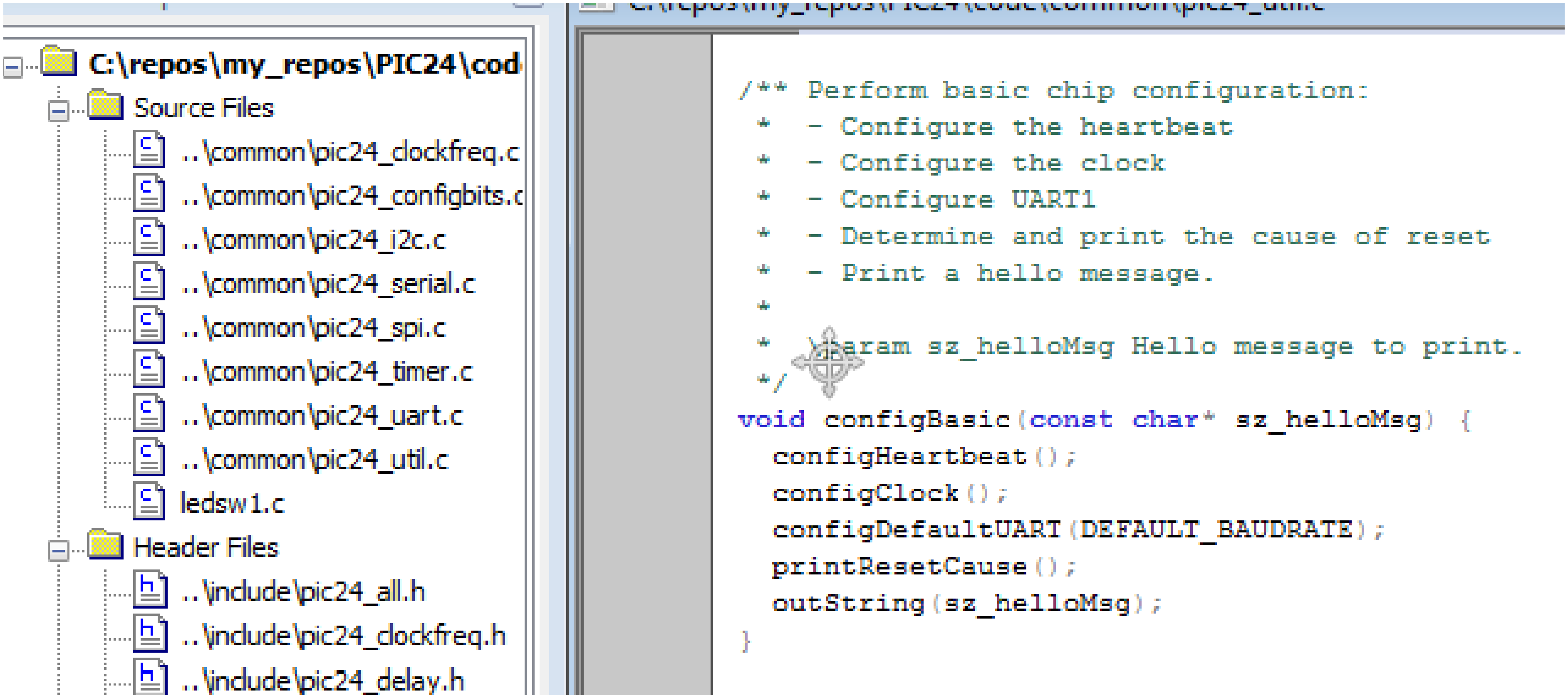
(h) Toggle LED and change state when switch is released.

(i) Put debounce delay at bottom of loop, means that we only look at the switch about every `DEBOUNCE_DLY` milliseconds.

(j) Call `doHeartbeat()` to keep heartbeat LED pulsing.

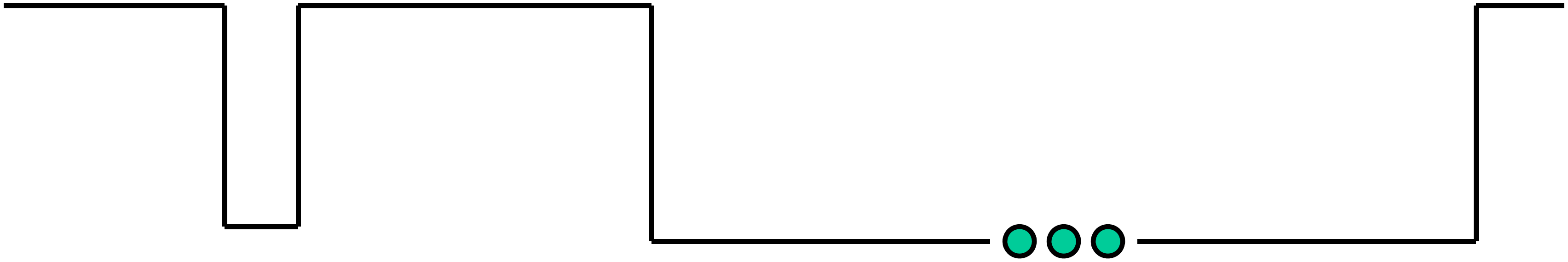
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

configBasic() function



```
/** 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();
  configClock();
  configDefaultUART(DEFAULT_BAUDRATE);
  printResetCause();
  outString(sz_helloMsg);
}
```

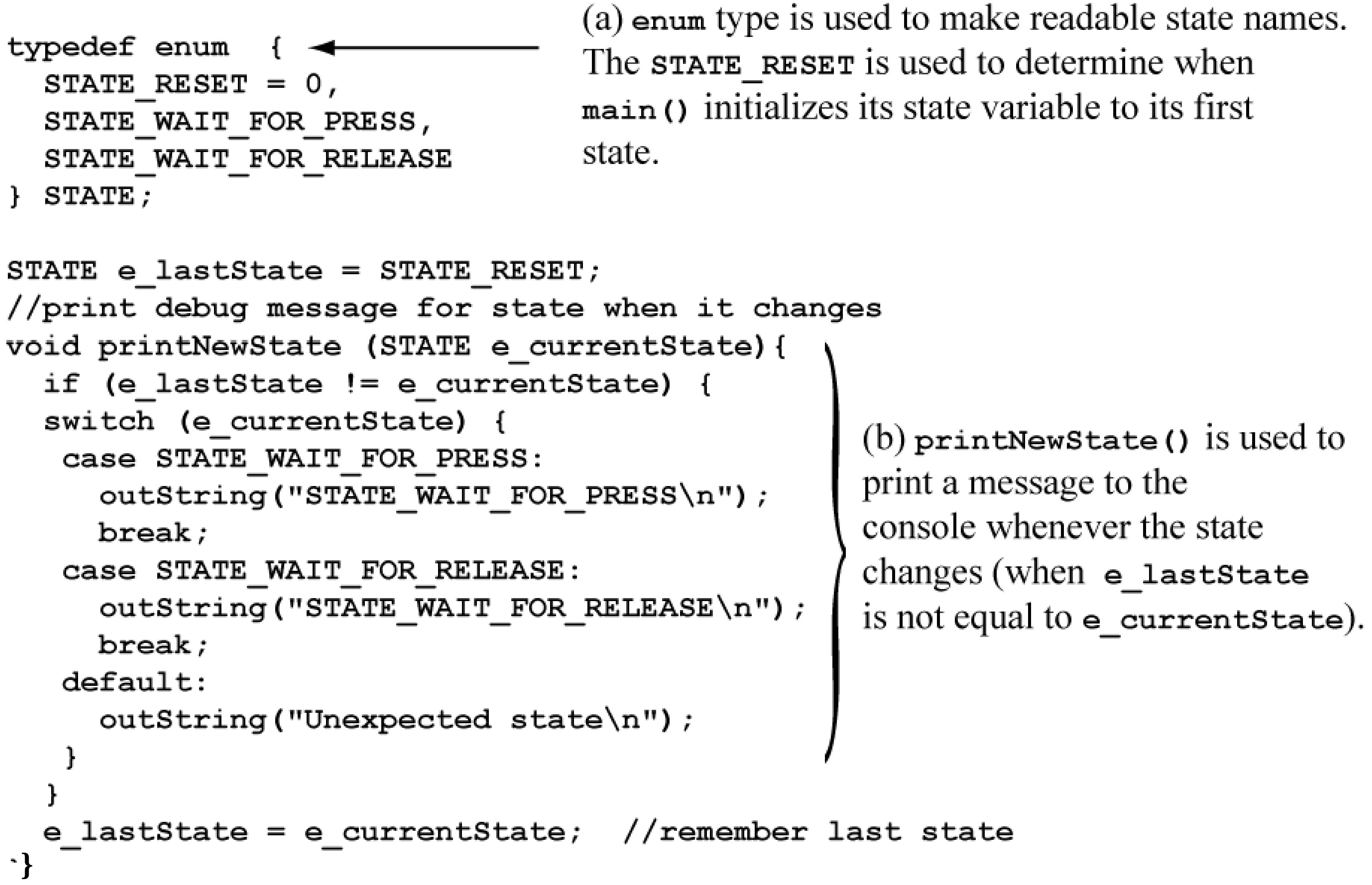
Switch Sampling and Debounce



Our new approach is **periodically sampling** the switch every ~15 ms in our while(1) loop. In the first solution, we were reading the switch as fast as the cpu 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).

C Code Solution (cont).



```
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");
   }
  }
  e_lastState = e_currentState;  //remember last state
}
```

(a) `enum` type is used to make readable state names. The `STATE_RESET` is used to determine when `main()` initializes its state variable to its first state.

(b) `printNewState()` is used to print a message to the console whenever the state changes (when `e_lastState` is not equal to `e_currentState`).

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

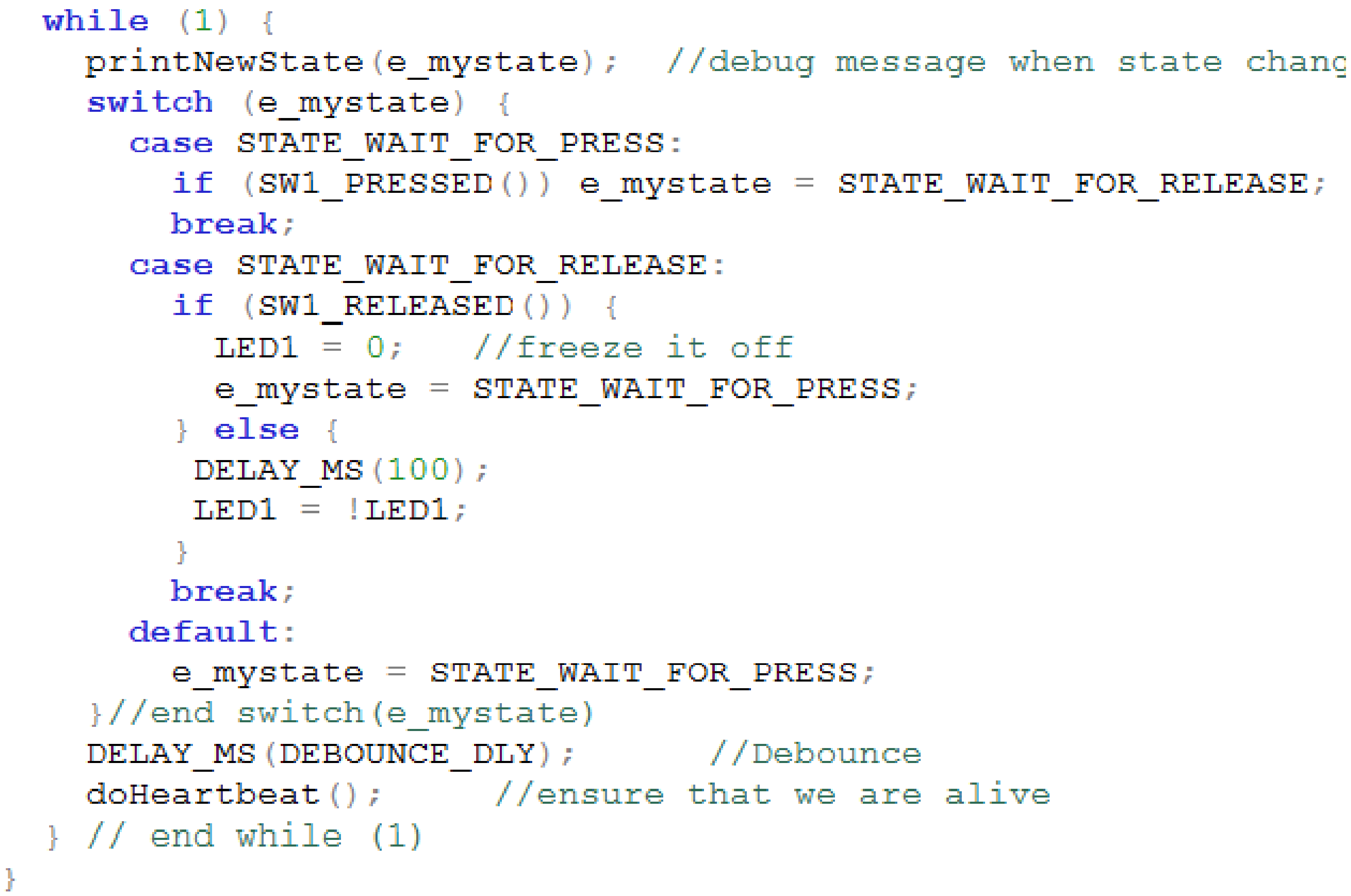
LED Toggle Variations

Two variations of LED Toggle (recall that LED was toggled when the switch was pressed and released)

- Variation 1: Blink the LED when the switch is pressed; when released, freeze it OFF
- Variation 2: Blink the LED a maximum of 4 times when switch is pressed; when released, freeze it OFF

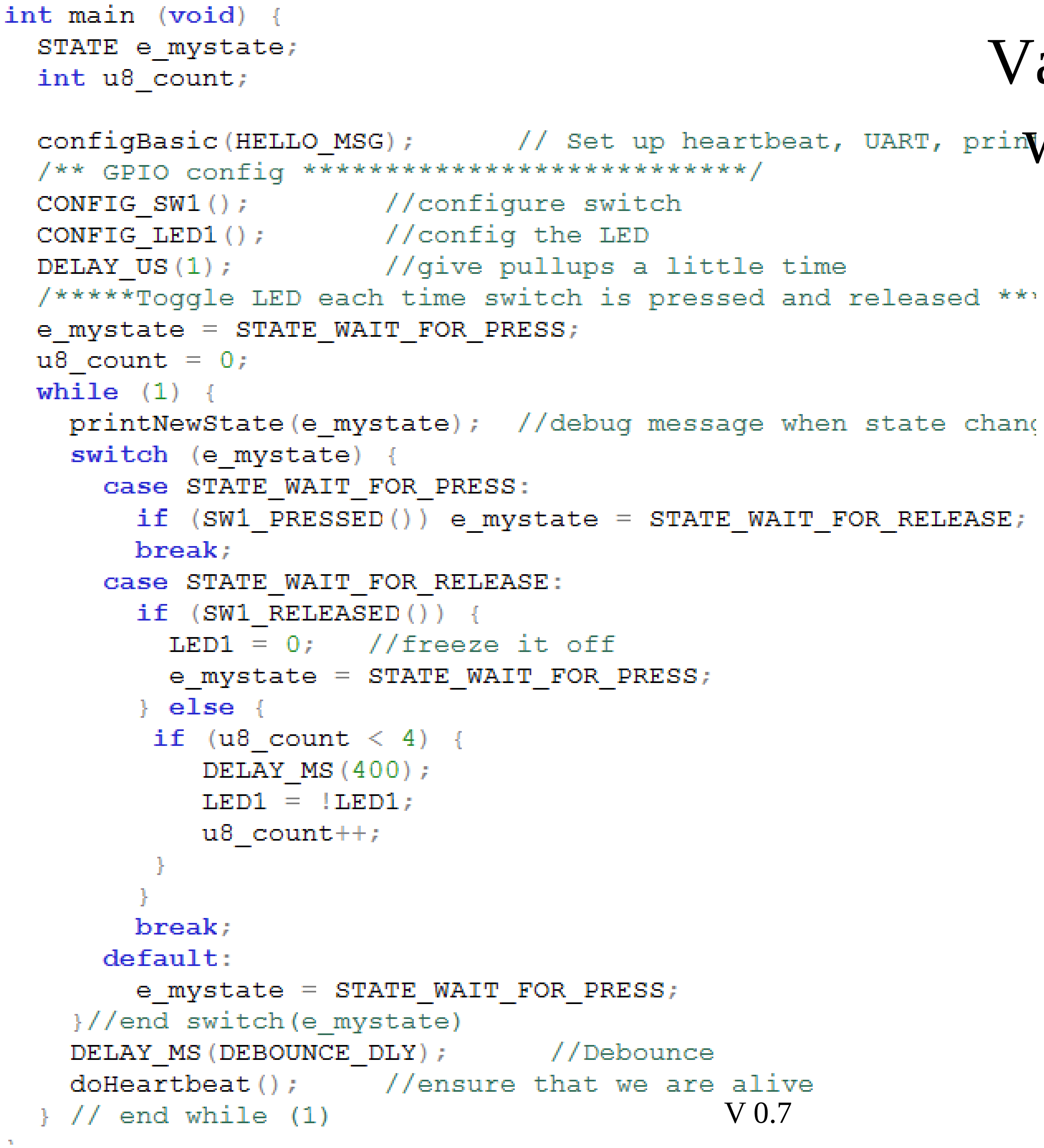
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Variation 1: Blink while pressed



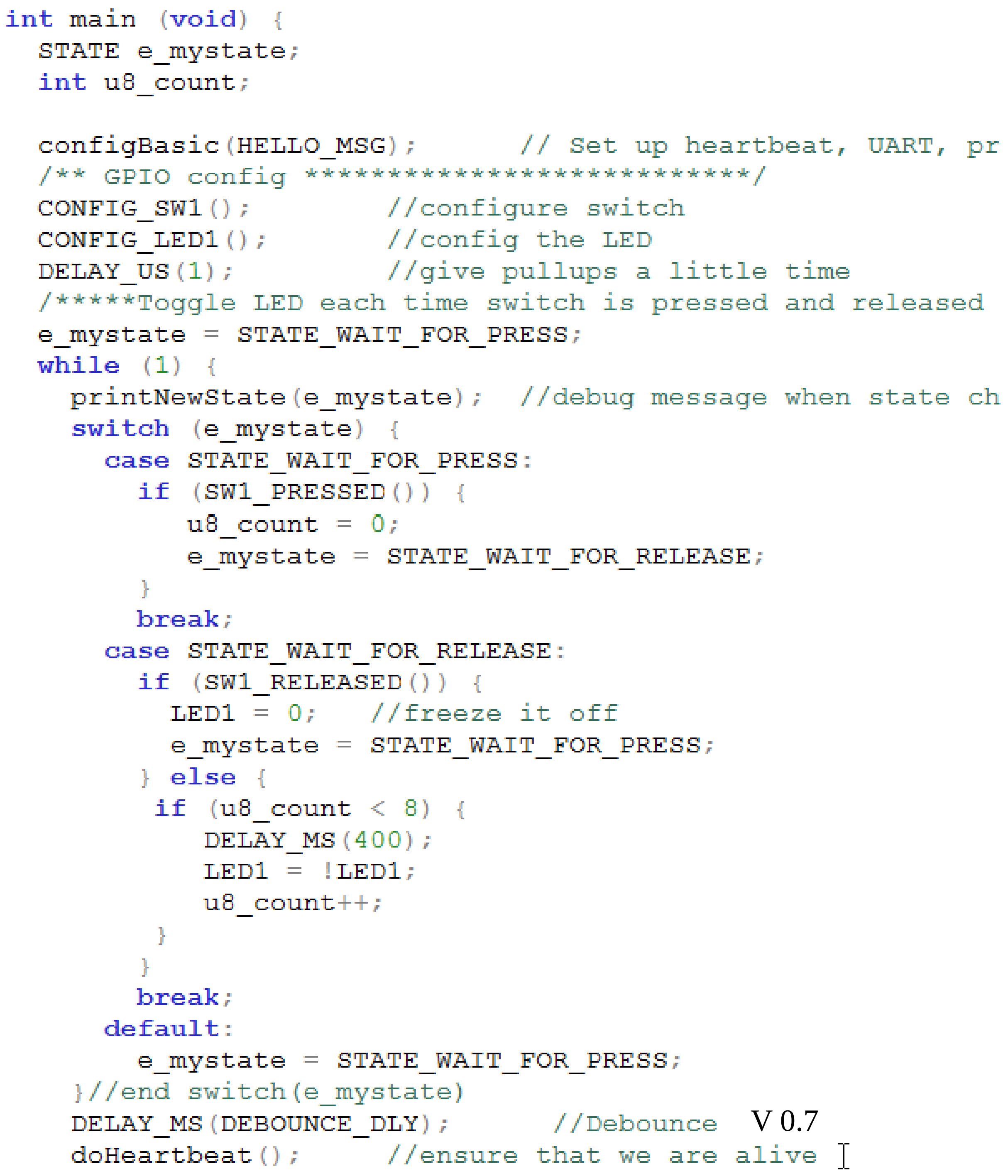
```
  while (1) {
    printNewState(e_mystate);  //debug message when state chang
    switch (e_mystate) {
      case STATE_WAIT_FOR_PRESS:
        if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE;
        break;
      case STATE_WAIT_FOR_RELEASE:
        if (SW1_RELEASED()) {
          LED1 = 0;   //freeze it off
          e_mystate = STATE_WAIT_FOR_PRESS;
        } else {
         DELAY_MS(100);
         LED1 = !LED1;
        }
        break;
      default:
        e_mystate = STATE_WAIT_FOR_PRESS;
    }//end switch(e_mystate)
    DELAY_MS(DEBOUNCE_DLY);      //Debounce
    doHeartbeat();     //ensure that we are alive
  } // end while (1)
}
```

Variation 2: Blink while pressed 4 times, try #1



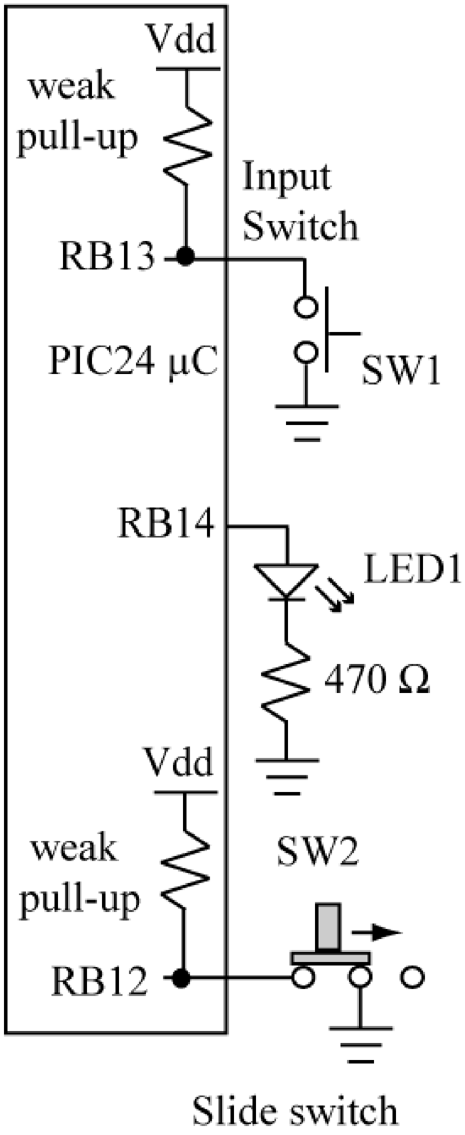
```
int main (void) {
  STATE e_mystate;
  int u8_count;

  configBasic(HELLO_MSG);      // Set up heartbeat, UART, prin
  /** GPIO config ***************************/
  CONFIG_SW1();         //configure switch
  CONFIG_LED1();        //config the LED
  DELAY_US(1);          //give pullups a little time
  /*****Toggle LED each time switch is pressed and released **
  e_mystate = STATE_WAIT_FOR_PRESS;
  u8_count = 0;
  while (1) {
    printNewState(e_mystate);  //debug message when state chang
    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 = 0;   //freeze it off
          e_mystate = STATE_WAIT_FOR_PRESS;
        } else {
         if (u8_count < 4) {
            DELAY_MS(400);
            LED1 = !LED1;
            u8_count++;
         }
        }
        break;
      default:
        e_mystate = STATE_WAIT_FOR_PRESS;
    }//end switch(e_mystate)
    DELAY_MS(DEBOUNCE_DLY);      //Debounce
    doHeartbeat();     //ensure that we are alive
  } // end while (1)
```

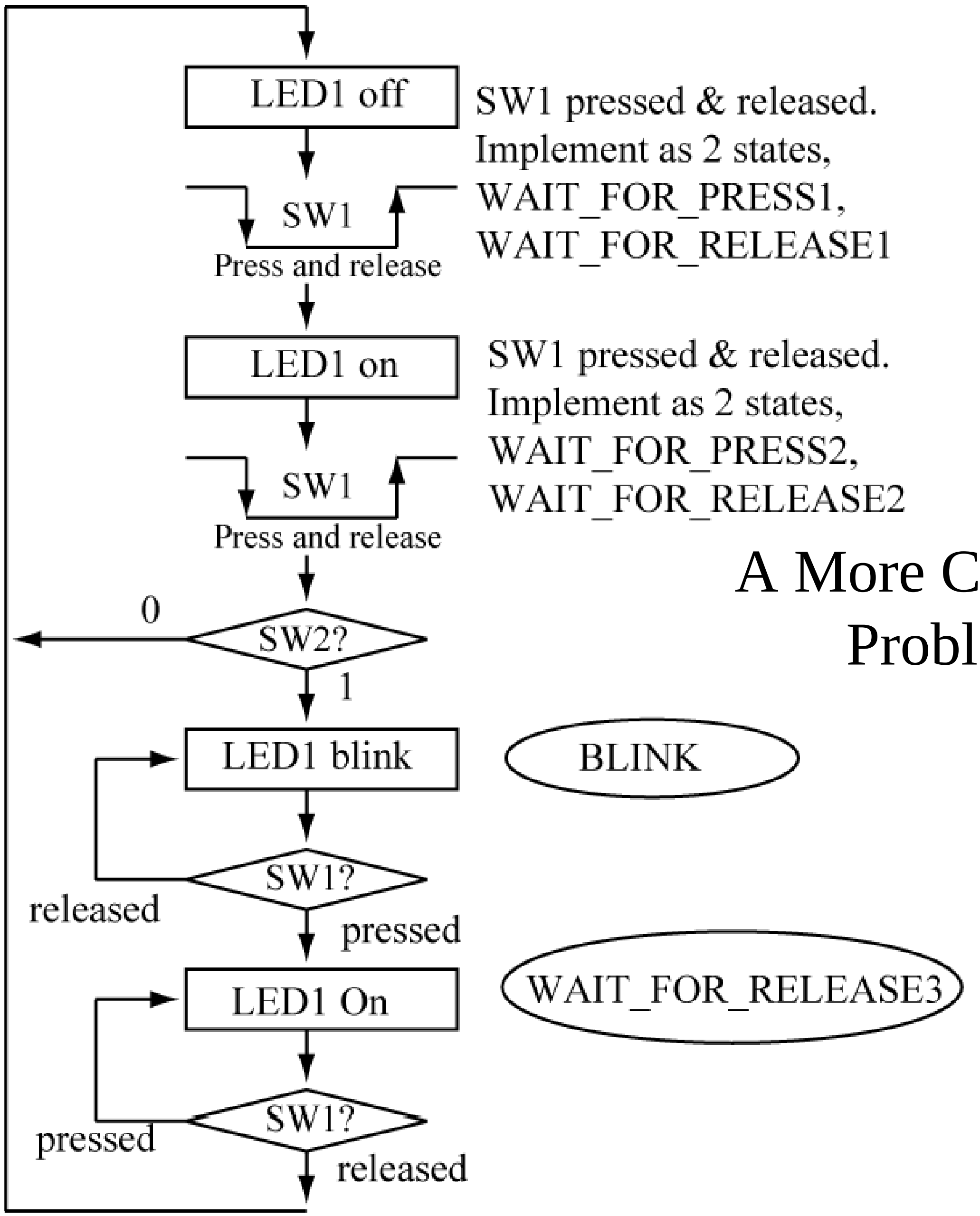
Variation 2: Blink while pressed 4 times, try #2



```
int main (void) {
  STATE e_mystate;
  int u8_count;

  configBasic(HELLO_MSG);      // Set up heartbeat, UART, pr
  /** GPIO config ***************************/
  CONFIG_SW1();        //configure switch
  CONFIG_LED1();       //config the LED
  DELAY_US(1);         //give pullups a little time
  /*****Toggle LED each time switch is pressed and released
  e_mystate = STATE_WAIT_FOR_PRESS;
  while (1) {
    printNewState(e_mystate);  //debug message when state ch
    switch (e_mystate) {
      case STATE_WAIT_FOR_PRESS:
        if (SW1_PRESSED()) {
           u8_count = 0;
           e_mystate = STATE_WAIT_FOR_RELEASE;
        }
        break;
      case STATE_WAIT_FOR_RELEASE:
        if (SW1_RELEASED()) {
          LED1 = 0;   //freeze it off
          e_mystate = STATE_WAIT_FOR_PRESS;
        } else {
         if (u8_count < 8) {
            DELAY_MS(400);
            LED1 = !LED1;
            u8_count++;
         }
        }
        break;
      default:
        e_mystate = STATE_WAIT_FOR_PRESS;
    }//end switch(e_mystate)
    DELAY_MS(DEBOUNCE_DLY);      //Debounce
    doHeartbeat();     //ensure that we are alive
```

A More Complex Problem



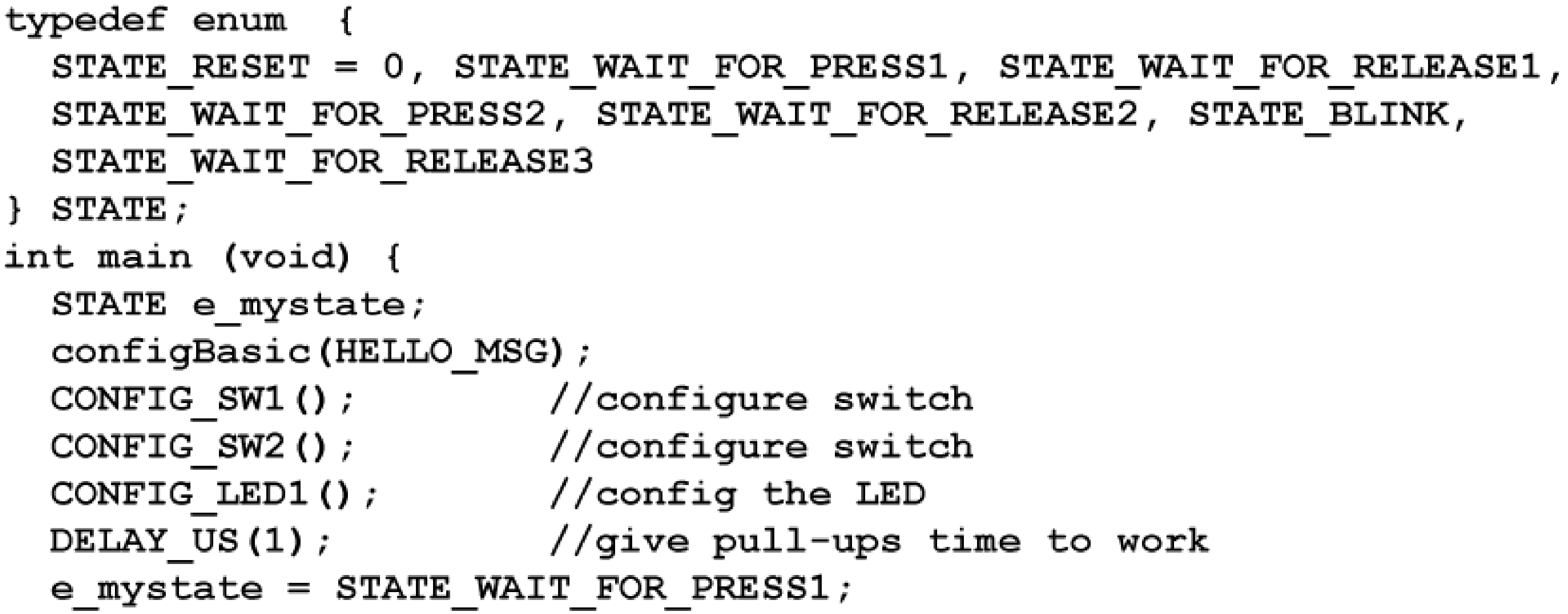


SW1 pressed & released.
Implement as 2 states,
WAIT_FOR_PRESS1,
WAIT_FOR_RELEASE1

SW1 pressed & released.
Implement as 2 states,
WAIT_FOR_PRESS2,
WAIT_FOR_RELEASE2

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

Solution, Part 1



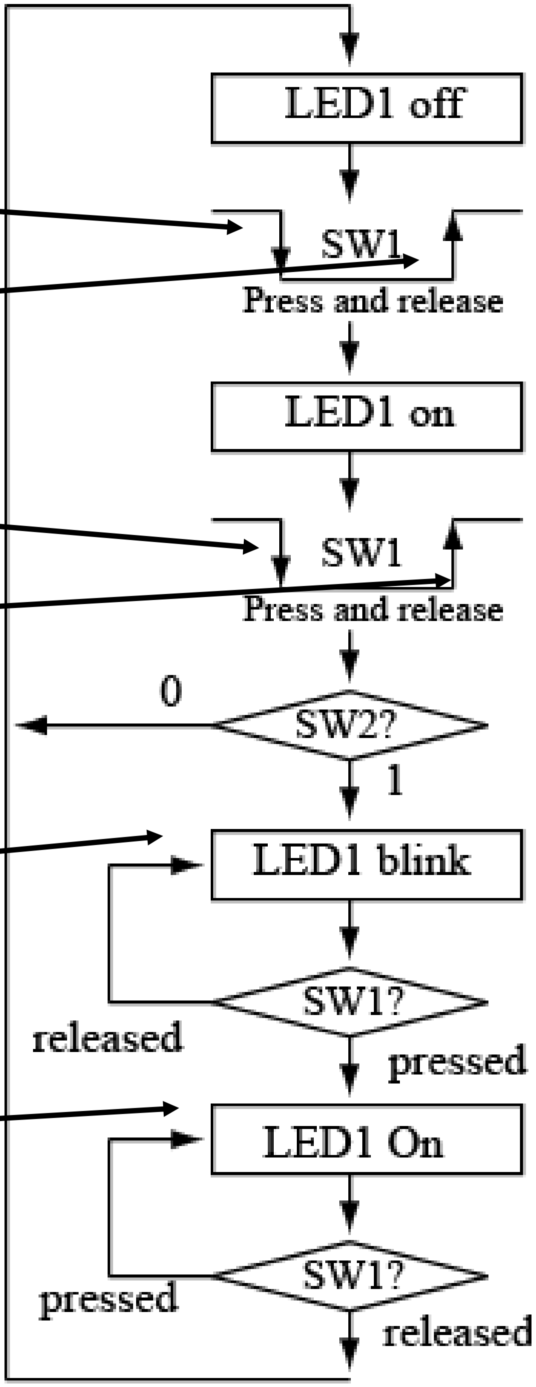
```
typedef enum  {
  STATE_RESET = 0, STATE_WAIT_FOR_PRESS1, STATE_WAIT_FOR_RELEASE1,
  STATE_WAIT_FOR_PRESS2, STATE_WAIT_FOR_RELEASE2, STATE_BLINK,
  STATE_WAIT_FOR_RELEASE3
} STATE;
int main (void) {
  STATE e_mystate;
  configBasic(HELLO_MSG);
  CONFIG_SW1();        //configure switch
  CONFIG_SW2();        //configure switch
  CONFIG_LED1();       //config the LED
  DELAY_US(1);         //give pull-ups time to work
  e_mystate = STATE_WAIT_FOR_PRESS1;
```

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

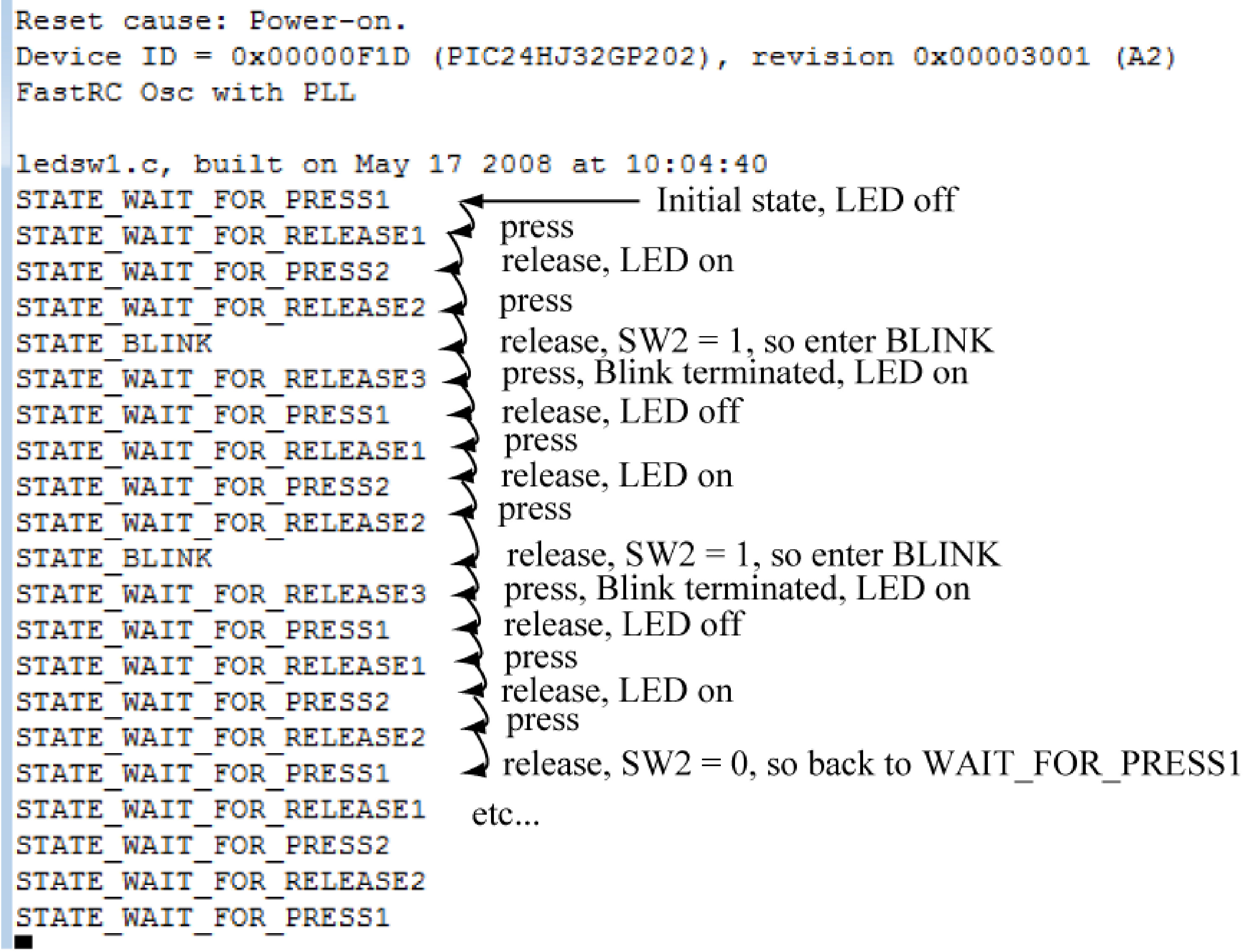
```
while (1) {
  printNewState(e_mystate);  //debug message when state changes
  switch (e_mystate) {
    case STATE_WAIT_FOR_PRESS1:
      LED1 = 0; //turn off the LED
      if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE1;
      break;
    case STATE_WAIT_FOR_RELEASE1:
      if (SW1_RELEASED()) 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;
    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;
```

(a) Test SW2 to determine next state.

(b) Need delay so that LED blink is visible.



Console Output for LED/SW Problem



```
Reset cause: Power-on.
Device ID = 0x00000F1D (PIC24HJ32GP202), revision 0x00003001 (A2)
FastRC Osc with PLL

ledsw1.c, built on May 17 2008 at 10:04:40
STATE_WAIT_FOR_PRESS1
STATE_WAIT_FOR_RELEASE1
STATE_WAIT_FOR_PRESS2
STATE_WAIT_FOR_RELEASE2
STATE_BLINK
STATE_WAIT_FOR_RELEASE3
STATE_WAIT_FOR_PRESS1
STATE_WAIT_FOR_RELEASE1
STATE_WAIT_FOR_PRESS2
STATE_WAIT_FOR_RELEASE2
STATE_BLINK
STATE_WAIT_FOR_RELEASE3
STATE_WAIT_FOR_PRESS1
STATE_WAIT_FOR_RELEASE1
STATE_WAIT_FOR_PRESS2
STATE_WAIT_FOR_RELEASE2
STATE_WAIT_FOR_PRESS1
STATE_WAIT_FOR_RELEASE1
STATE_WAIT_FOR_PRESS2
STATE_WAIT_FOR_RELEASE2
STATE_WAIT_FOR_PRESS1
```

Initial state, LED off

press

release, LED on

press

release, SW2 = 1, so enter BLINK

press, Blink terminated, LED on

release, LED off

press

release, LED on

press

release, SW2 = 1, so enter BLINK

press, Blink terminated, LED on

release, LED off

press

release, LED on

press

release, SW2 = 0, so back to WAIT_FOR_PRESS1

etc...

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

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.