T2CON: Timer2 Control Register

7	6	5	4	3	2	1	0
-- u --	TOUTPS3	TOUTPS2	TOUTPS3	TOUTPS3	TMR2ON	T2CKPS1	T2CKPS0

-- u -- : unimplemented

TOUTPS3:TOUTPS2 Postscale Select
0000 = 1:1 Postscale
0001 = 1:2 Postscale
0010 = 1:3 Postscale
....
1110 = 1:15 Postscale
1111 = 1:16 Postscale

TMR2ON: Timer2 On Bit (1 is on, 0 is off)

T2CKPS1: T2CKPS0: Timer2 Clock Prescale
00 = Prescaler is 1
01 = Prescaler is 4
1x = Prescaler is 16

FIGURE 10.22 Timer2 configuration register (T2CON).

Sample Question: Assuming FOSC = 30 MHz, what Timer2 configuration will generate a periodic interrupt every 5 ms?

Answer: Using Equation 10.2 and letting POST = 16, we find:

PR2 = [5 ms / [(1/30 MHz)*4*PRE*16]] = [0.005/[3.33e-8 * 4 * PRE*16]

This results in PR2 = 2343 for PRE = 1, PR2 = 585 for PRE = 4, and PR2 = 145 for PRE = 16. Thus, the only valid choice for POST = 16 is PRE = 16, PR2 = 145, as this is the only configuration that gives a PR2 < 255.

10.9 SWITCH DEBOUNCING USING A TIMER

The LED/switch examples of Section 10.7 use a 30 ms software delay in the ISR for switch debouncing. This is not the best method to use, as the ISR is stealing time from the foreground code via wasted cycles in the software delay loop. A better method is to use a timer for switch debouncing as shown in Figure 10.23. The goal is to create a semaphore that signals a press and release of the momentary switch in the presence of switch bounce. Timer2 is configured to generate periodic interrupts and the INTx input is configured as a falling edge interrupt.

The first falling edge from a switch activation triggers the ISR, which sets a semaphore and then disables the interrupt so that successive falling edges due to switch bounce do not generate interrupts. On each Timer2 interrupt thereafter, a counter is kept that tracks the number of successive Timer2 interrupts that the INT2 input remains high. If the INT2 input remains high long enough, it is considered idle and the INT2 interrupt is re-enabled.

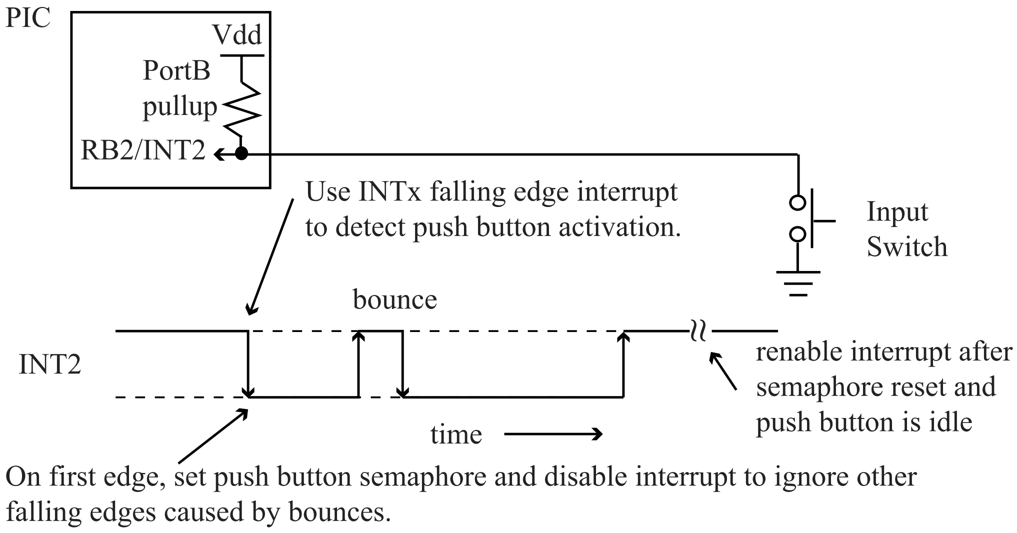

PIC
Vdd
PortB
pullup
RB2/INT2
Use INTx falling edge interrupt to detect push button activation.
Input Switch
bounce
INT2
renable interrupt after semaphore reset and push button is idle
time
On first edge, set push button semaphore and disable interrupt to ignore other falling edges caused by bounces.

FIGURE 10.23 Using a timer to debounce an interrupt-driven switch input.

Figure 10.24 shows the *C* code implementation of this debounce scheme. The `button` variable is the semaphore that is set by the ISR when a falling edge occurs on the INT2 input indicating that switch activation has occurred. Observe that once the INT2 interrupt is recognized, it is disabled via `INT2IE = 0` and the `button_debounce` count value is cleared. The INT2IF flag is not cleared at this time because any switch bounce that is present sets the flag; the INT2IF flag cannot be reliably cleared until the switch bounce has settled. For each Timer2 interrupt, if the INT2 interrupt is disabled, this means that the last switch activation is being debounced. If the RB2 input is high, the debounce counter, `button_debounce`, is incremented. If the RB2 input is low, `button_debounce` is cleared. Once RB2 has tested high for `DEBOUNCE` consecutive Timer2 interrupts it is considered idle. If RB2 is idle and the previous semaphore has been acknowledged (i.e., `button` has been cleared by the foreground code), the INT2 interrupt is re-enabled via `INT2IE = 1` for triggering by a switch activation.

The `main()` code of Figure 10.24 initializes the INT2 input for falling edge triggering and Timer2 for periodic interrupt generation. The values POST = 11, PRE = 16, and PR2 = 250 for a FOSC = 29.4952 MHz (PIC reference board) give a Timer2 interrupt period of approximately 6 ms. With `DEBOUNCE = 5`, this means that the RB2 input remaining high for approximately 24 to 30 ms is considered idle. The variation in the debounce time is because the Timer2 value is unknown when the switch activation occurs. Thus, the first Timer2 interrupt may occur anywhere in the 5 ms interrupt interval. The `while(1){}` loop is free-running with respect to the button semaphore; when the `button` semaphore is set, a message is printed and the semaphore is acknowledged by clearing it.

```
#define DEBOUNCE  5
volatile unsigned char button, button_debounce;

void interrupt pic_isr(void){
 if (INT2IF && INT2IE) {
  // pushbutton detected
  INT2IE = 0;  button = 1;  button_debounce = 0;
 }
 if (TMR2IF) { // debouncing timer
  TMR2IF = 0;
  if (!INT2IE) {
   if (RB2) {
    if (button_debounce != DEBOUNCE)
      button_debounce++;
    }
   else button_debounce=0;
   if (button_debounce == DEBOUNCE && !button){
    //button idle high ,re-enable interrupt
    INT2IF=0; INT2IE=1;
   }
  }
 }
}
main(void){
 serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
 // configure INT2 for falling edge interrupt input
 TRISB2 = 1;  INT2IF = 0; INTEDG2 = 0;  INT2IE = 1;
 RBPU = 0; // enable weak pullup on port B
 // configure timer 2
 // post scale of 11, prescale 16, PR=250, FOSC=29.4912 MHz
 // gives interrupt interval of ~ 6 ms
 TOUTPS3 = 1; TOUTPS2 = 0; TOUTPS1 = 1; TOUTPS0 = 0;
 T2CKPS1 = 1;  PR2 = 250;
 // enable TMR2 interrupt
 IPEN = 0;  TMR2IF = 0; TMR2IE = 1;   PEIE = 1;  GIE = 1;
 TMR2ON = 1 ;
 pcrlf();  printf("Pushbutton with Timer2 Debounce");  pcrlf();
 while(1) {
  if (button) {
   button=0; // acknowledge this semaphore
   printf("Push Button activated!"); pcrlf();
  }
 }// end while
}//end main
```

5 * 6 ms = 24 to 30 ms debounce time

Falling edge interrupt, set the semaphore and disable interrupt

If the interrupt is disabled, then debounce the switch by waiting for it to become idle high.
After the switch is debounced and the semaphore is acknowledged, then reenable the interrupt.

Configure INT2 for falling edge interrupt

Configure Timer2 for ~6 ms interrupt period

If pushbutton is activated, then print message and reset the semaphore

ON THE CD

FIGURE 10.24 Switch debounce implementation.

This approach sets the `button` semaphore on each press and release of the push button switch. If the interface requires that a press and hold be detected, a similar approach that waits for the input to be idle low can be used. The next section discusses a second example in which a periodic interrupt is used to sample noisy inputs to reject momentary pulses or glitches that may be present.

10.10 A ROTARY ENCODER INTERFACE

A rotary encoder is used to encode the direction and distance of mechanical shaft rotation. There are different ways to accomplish this; Figure 10.25 shows a 2-bit

Gray code rotary encoder. Clockwise rotation of the shaft produces the sequence 00, 10, 11, 01, and counterclockwise rotation produces 00, 01, 11, and 10. In a Gray code, adjacent encodings differ by only one bit position. Rotation direction is determined by comparing the current 2-bit value with the last value. For example, if the current value is 11 and the last value is 10, the shaft is rotating in a clockwise direction. One common use for a rotary encoder is as an input device on a control panel where clockwise rotation increments a selected parameter setting, while counter-clockwise rotation decrements the parameter. The rotary encoder of Figure 10.25 is an *incremental* encoder as the absolute position of the shaft is indeterminate; only relative motion is encoded. Some rotary encoders include more bits that provide absolute shaft position, in BCD or binary encoding. An *n*-position encoder outputs *n*-codes for each complete shaft rotation. Common values of *n* for 2-bit incremental rotary encoders are 16 and 32.

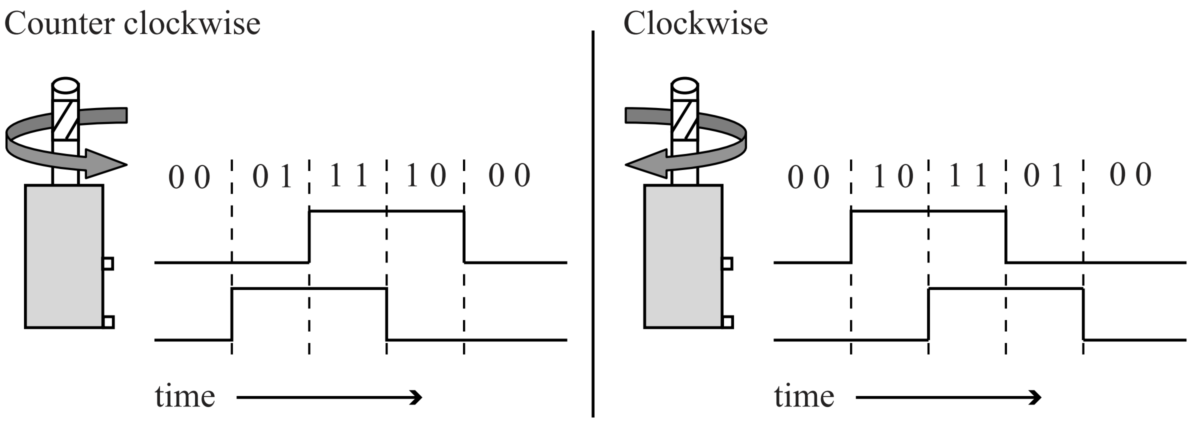


FIGURE 10.25 Two-bit Gray code rotary encoder.

Rotary encoders use mechanical, optical, or magnetic means of detecting shaft rotation, with mechanical encoders being the least expensive and magnetic the most expensive. A key specification for optical and mechanical encoders is rotational life with optical ~ 1 million and mechanical ~ 100,000 rotations due to mechanical wear. Magnetic encoders are meant for high-speed rotational applications with encoder lifetime measured in thousands of hours for a fixed rotational speed in revolutions per minute (RPMs).

Figure 10.26 shows ISR code that uses INT0/INT1 edge triggered interrupts for a rotary encoder interface. The ISR triggers on the occurrence of an active edge on either INT0 or INT1. The ISR checks the flag bits INT0IF/INT1IF, determines which interrupt occurred, and toggles the appropriate edge bit INTEDG0/INTEDG1. The `update_state()` function then reads the INT0/INT1 inputs and compares them against the previous state to determine clockwise or counterclockwise rotation of the encoder. If a valid state transition is found, the *count* variable is either incremented or decremented appropriately. Observe that an invalid state transition indicates that an illegal transition has occurred, perhaps caused by noise, and the state variable is reset to the current value of the INT0/INT1 inputs.

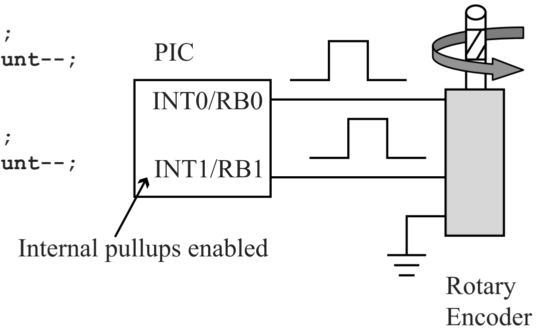
```
#define S0 0
#define S1 1
#define S2 2
#define S3 3

volatile unsigned char state, last_state;
volatile unsigned char count, last_count;

update_state(){
  state = PORTB & 0x3;
  switch(state) {
  case S0:
    if (last_state == S1) count++;
    else if (last_state == S2) count--;
    break;
  case S1:
    if (last_state == S3) count++;
    else if (last_state == S0) count--;
    break;
  case S2:
    if (last_state == S0) count++;
    else if (last_state == S3) count--;
    break;
  case S3:
    if (last_state == S2) count++;
    else if (last_state == S1) count--;
    break;
  }
  if (last_count != count) {
    // valid pulse
    last_state = state;
    last_count = count;
  } else {
    // invalid transistion, reset last state
    last_state = state;
  }
}
void interrupt pic_isr(void){
  if (INT0IF || INT1IF) {
    if (INT0IF) {
      INT0IF = 0;
      // toggle active edge
      if (INTEDG0) INTEDG0 = 0; else INTEDG0 = 1;
    }
    if (INT1IF) {
      INT1IF = 0;
      // toggle active edge
      if (INTEDG1) INTEDG1 = 0; else INTEDG1 = 1;
    }
    update_state();
  }
}
```

Update the `state` and `count` variables based upon the INT0/INT input values.

Check previous state and determine if rotating clockwise or counter-clockwise



Should not get here unless illegal transition occurred, attempt a recovery

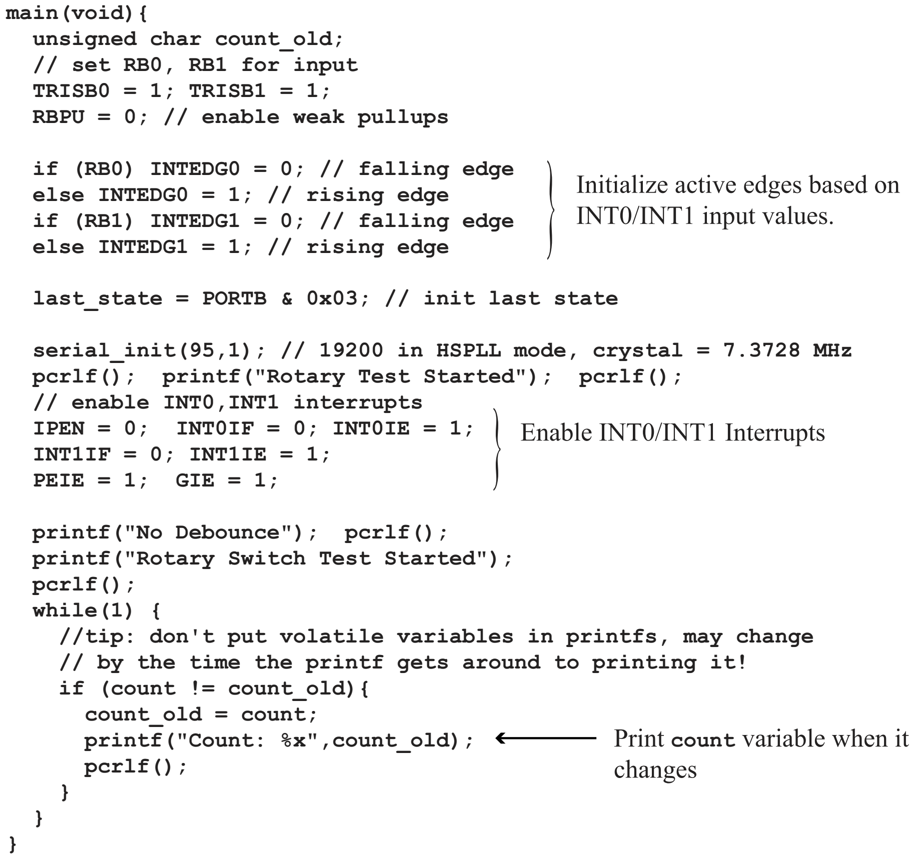
INT0 Active Edge occurred

INT1 Active Edge occurred

ON THE CD

FIGURE 10.26 Two-bit rotary encoder interface using INT0/INT1 interrupts.

The main() code shown in Figure 10.27 initializes the INT0/INT1 active interrupt edges (INTEDG0/INTEDG1) by reading the current value of the INT0/INT1 inputs; if the input is high, the falling edge is chosen, else the rising edge is selected. This is necessary because the initial values of the rotary encoder outputs depend upon the current shaft position, which is unknown at main() startup. The state variable used by the ISR of Figure 10.26 to track the position of the rotary encoder is also initialized by main() based upon the INT0/INT1 inputs. The while(1){} loop in the main() code waits for a change on the count variable and prints its value once a change is detected.



```
main(void){
  unsigned char count_old;
  // set RB0, RB1 for input
  TRISB0 = 1; TRISB1 = 1;
  RBPU = 0; // enable weak pullups

  if (RB0) INTEDG0 = 0; // falling edge
  else INTEDG0 = 1; // rising edge
  if (RB1) INTEDG1 = 0; // falling edge
  else INTEDG1 = 1; // rising edge

  last_state = PORTB & 0x03; // init last state

  serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
  pcrlf(); printf("Rotary Test Started"); pcrlf();
  // enable INT0,INT1 interrupts
  IPEN = 0;  INT0IF = 0; INT0IE = 1;
  INT1IF = 0; INT1IE = 1;
  PEIE = 1;  GIE = 1;

  printf("No Debounce"); pcrlf();
  printf("Rotary Switch Test Started");
  pcrlf();
  while(1) {
    //tip: don't put volatile variables in printfs, may change
    // by the time the printf gets around to printing it!
    if (count != count_old){
      count_old = count;
      printf("Count: %x",count_old);
      pcrlf();
    }
  }
}
```

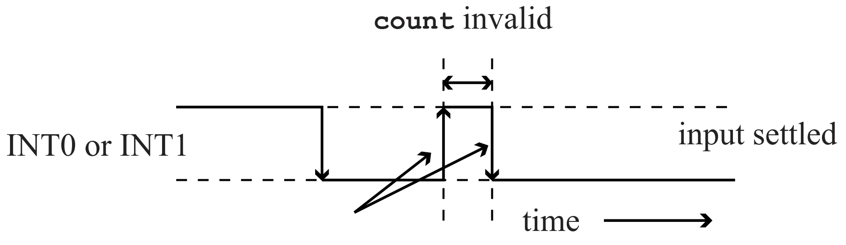
Initialize active edges based on INT0/INT1 input values.

Enable INT0/INT1 Interrupts

Print count variable when it changes

FIGURE 10.27 main() for initializing INT0/INT1 interrupts, state variable.

The code of Figures 10.26 and 10.27 works well as long as the signal transitions are noise free and clean of contact bounce, which is generally true of the signals produced by optical and magnetic encoders. However, mechanical encoders have contact bounce that will cause the count variable to change multiple times for a single shaft movement, potentially creating errors in code that samples the count value. Figure 10.28 illustrates this problem, as the ISR is triggered on each edge of the contact bounce, causing count to be modified each time. There is a possibility that `count` can be sampled by the normal program flow when it contains an incorrect value, causing unreliable behavior.



Contact bounce edges, ISR triggered on each edge, modifies `count` each time
as each state transition is valid since it is returning to the previous state on the bounce.

FIGURE 10.28 Switch bounce in mechanical encoders.

As was done for the LED/switch IO example of the previous section, Timer2 is used as a periodic interrupt for debouncing the rotary encoder inputs. Figure 10.29 shows an ISR triggered by Timer2 that samples the INT0/RB0, INT1/RB1 inputs on each interrupt. The `int0_last`, `int1_last` variables contain the last stable values of the INT0, INT1 inputs. If an input is different from its last stable value and remains that way for `DEBOUNCE` consecutive interrupt periods, the input has reached a new stable value and the `update_state()` function is called to update the `state`, `count` variables. The `int0_cnt`, `int1_cnt` variables are used for tracking the number of consecutive interrupts that an input remains changed from its previous value. The count variable for an input is reset to zero if the input returns to its previous value before `DEBOUNCE` consecutive interrupt periods occurs. This approach uses Timer2 as the only interrupt source, the RB0/INT0 and RB1/INT1 interrupts are not enabled. The `update_state()` function is not shown in Figure 10.29, as it is the same function from Figure 10.26.

Figure 10.30 shows the `main()` code for configuring Timer2 as a periodic interrupt source. Timer2 is configured in the same manner as the switch debouncing example in which values of FOSC = 29.4952 MHz, POST = 11, PRE = 16, and PR2 = 250 give an interrupt period of approximately 6 ms. With `DEBOUNCE = 5`, this means that any pulses of width less than 30 ms are rejected as switch bounce or noise. The pulse width rejection should be chosen based on worst-case datasheet values for contact bounce. The sampling period should be chosen to guarantee several samples between valid input changes, with the time between input changes dependent upon the maximum expected shaft rotation speed and the number of positions for the encoder.

```
volatile unsigned char state,last_state;
volatile unsigned char count,last_count;
volatile unsigned char int0_cnt,int0_last;
volatile unsigned char int1_cnt,int1_last;
volatile unsigned char update_flag;

#define DEBOUNCE  5

void interrupt pic_isr(void){
  if (TMR2IF) {
    // debouncing rotary inputs
    TMR2IF = 0;
    if (RB0 != int0_last) {
      int0_cnt++;
      if (int0_cnt == DEBOUNCE) {
        update_flag = 1;
        int0_cnt = 0;int0_last = RB0;
      }
    }
    // reset cnt, if pulse width
    // not long enough
    else if (int0_cnt)  int0_cnt = 0;

    if (RB1 != int1_last) {
      int1_cnt++;
      if (int1_cnt == DEBOUNCE) {
        update_flag = 1;
        int1_cnt = 0; int1_last = RB1;
      }
    }
    // reset cnt, if pulse width
    // not long enough
    else if (int1_cnt)   int1_cnt = 0;

    if (update_flag) {
      // can read the rotary inputs
      update_state();
      update_flag = 0;
    }
  }
}
```

Variables for tracking stability of rotary encoder inputs

Rotary encoder input must be stable for this many consecutive Timer2 interrupts to be classified as a valid input

Has RB0 input changed value?

Yes, track number of times it remains stable

Stable for **`DEBOUNCE`** interrupt periods, set update flag, record input value

Reset counter if not stable for long enough

Check stability of RB1 input

Update **`state`** and **`count`** variables; **`update_state()`** function not shown as it is unchanged from previous example.

ON THE CD

FIGURE 10.29 Using Timer2 to sample the rotary encoder outputs.

```
main(void){
  unsigned char count_old;
  // set RB0, RB1 for input
  TRISB0 = 1; TRISB1 = 1;
  RBPU = 0; // enable weak pullups
  int0_last = RB0;
  int1_last = RB1;
  last_state = PORTB & 0x03; // init last state
  serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz

  // configure timer 2
  // post scale of 11
  TOUTPS3 = 1; TOUTPS2 = 0;
  TOUTPS1 = 1; TOUTPS0 = 0;
  // pre scale of 16 */
  T2CKPS1 = 1;
  TMR2ON = 1 ;
  PR2 = 250;

  // enable TMR2 interrupt
  IPEN = 0;  TMR2IF = 0; TMR2IE = 1;
  PEIE = 1;  GIE = 1;
  printf("With Timer2 Debounce");  pcrlf();
  printf("Rotary Switch Test Started");
  pcrlf();
  while(1) {
    //tip: don't put volatile variables in printfs, may change
    // by the time the printf gets around to printing it!
    if (count != count_old){
      count_old = count;
      printf("Count: %x",count_old);
      pcrlf();
    }
  }
}
```

Timer2 Interrupt period = POST*PRE*4*TOSC*PR2
= 11*16*4*(1/29491200)*250 = 6 ms (approx)

Postscale bits = 1010 for postscaler of 1:11

Prescale = 16

Turn on Timer2

Set period register

Enable Timer2 interrupt

ON THE CD

FIGURE 10.30 Configuring Timer2 for sampling the rotary encoder inputs (see CD-ROM file ./code/chap10/F_10_29_rotint_debounced.c).

10.11 A NUMERIC KEYPAD INTERFACE

A numeric keypad is a common element in a microcontroller system, as it provides an inexpensive method of input. A numeric keypad is simply a matrix of switches arranged in rows and columns and has no active electronics; a keypress connects a row and column pin together as shown in Figure 10.31.

The 4x3 numeric keypad of Figure 10.31 is shown connected to the PIC in Figure 10.32. The RB[3:1] port pins are configured as outputs driving low and connected to the row pins, while RB[7:4] are configured as inputs with the weak pullup enabled and connected to the column pins.