Timers

- Recall that a **timer** on a μC is simply a counter

Basic equations that we have used:

General:

Time = Ticks * Clock period of counter
Ticks = Time / Clock period of counter

PIC24 specific:

Time = Ticks * timer_prescale / FCY
Ticks = Time * FCY / timer_prescale

Period Register

Recall that the timer **period register** controls the amount of time for setting the TxIF flag (controls the Timer roll-over time):

TxIF period = (PRx + 1) * Prescale / FCY

To generate a periodic interrupt of *Y* milliseconds, we have done:

```
PRx  =  msToU16Ticks(Y, getTimerPrescale(TxCONbits)) – 1;
```

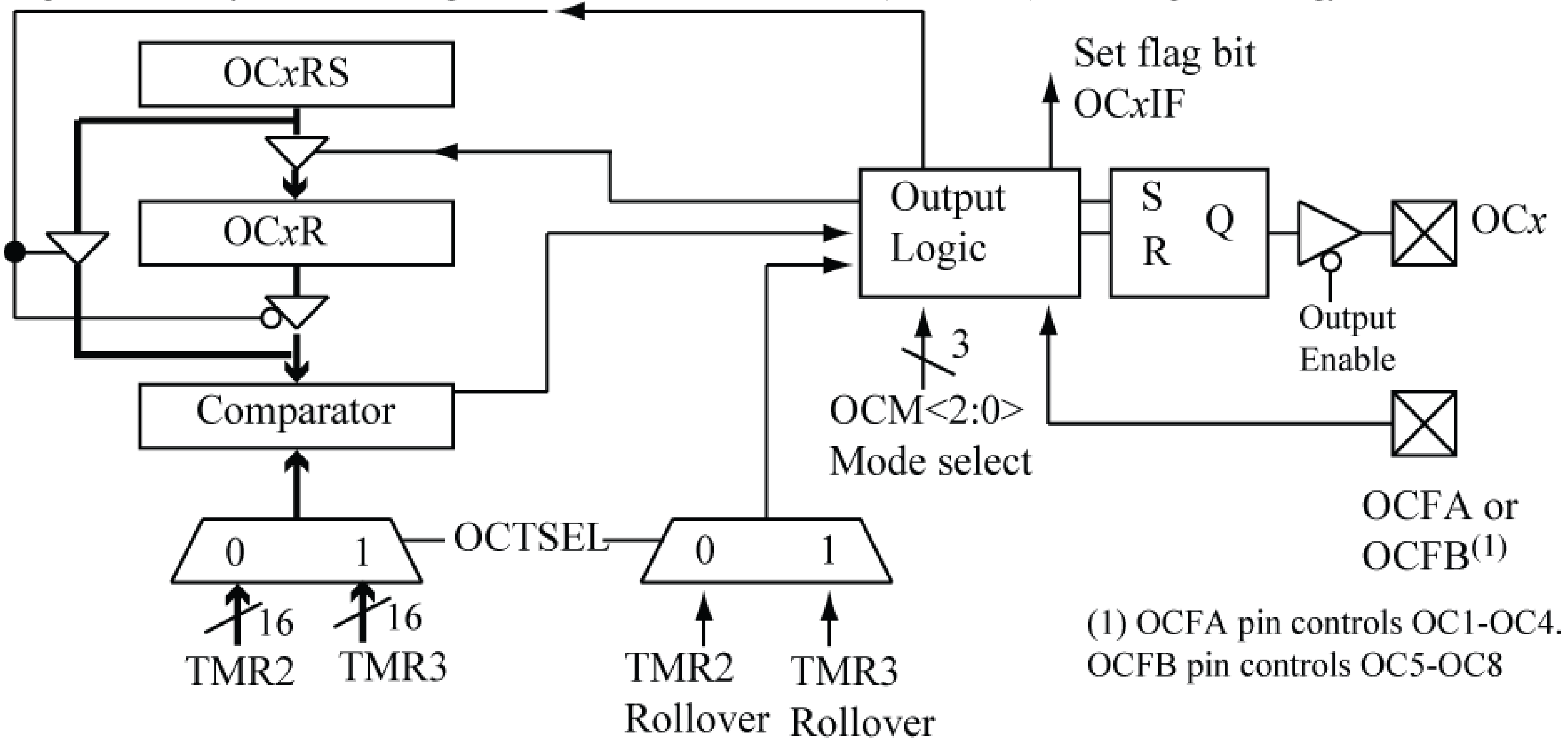
The `msToU16Ticks` function converts *Y* milliseconds to Timer ticks; the decrement by 1 is needed because rollover time is PR*x* + 1.

Input Capture, Output Compare Modules

- The Input Capture and Output Compare modules are peripherals that use Timer2 or Timer3 for time-related functions
- The Output Compare module can generate pulses of a specified pulse width and period
- The Input Capture module is used for measuring pulse width and period (elapsed time)

Output Compare Module

Figure redrawn by author from Fig 13-1 1 found in the PIC24 FRM (DS70247A), Microchip Technology Inc.



(1) OCFA pin controls OC1-OC4.
OCFB pin controls OC5-OC8

Pulses are generated on the OCx pin. The PIC24HJ32GP202 has two Output Compare modules (OC1, OC2). The OC*x*RS, OC*x*R registers control when the output pin is affected by comparing against either Timer2 or Timer3 values.

Output Compare Modes

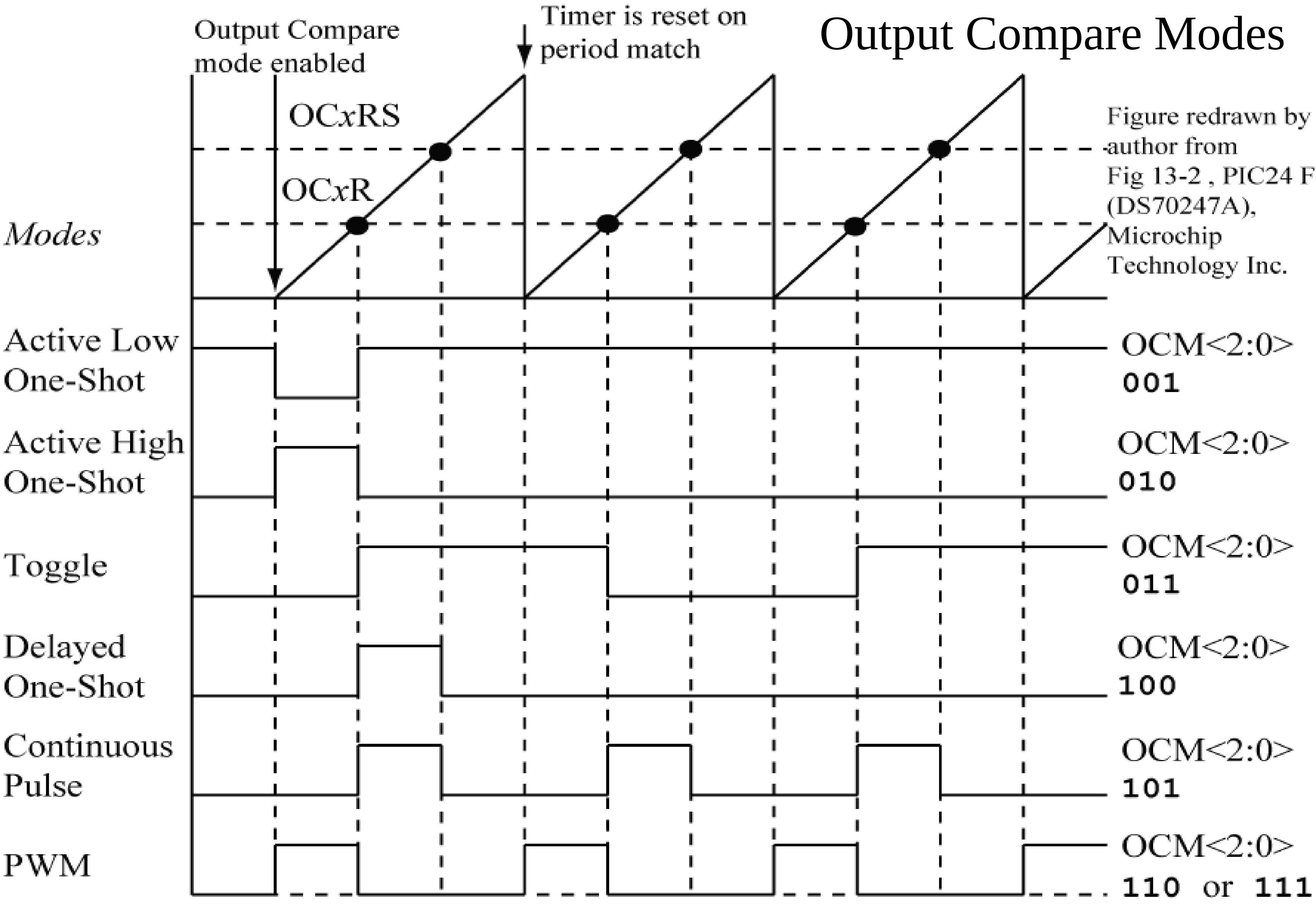


Figure redrawn by author from Fig 13-2 , PIC24 FRM (DS70247A), Microchip Technology Inc.

Generating a Square Wave using Toggle Mode

Steps (assume using Timer2, and OC1)

a. Configure Timer2 for a period that is greater than ½ period of the square wave.

b. Init OC1R register

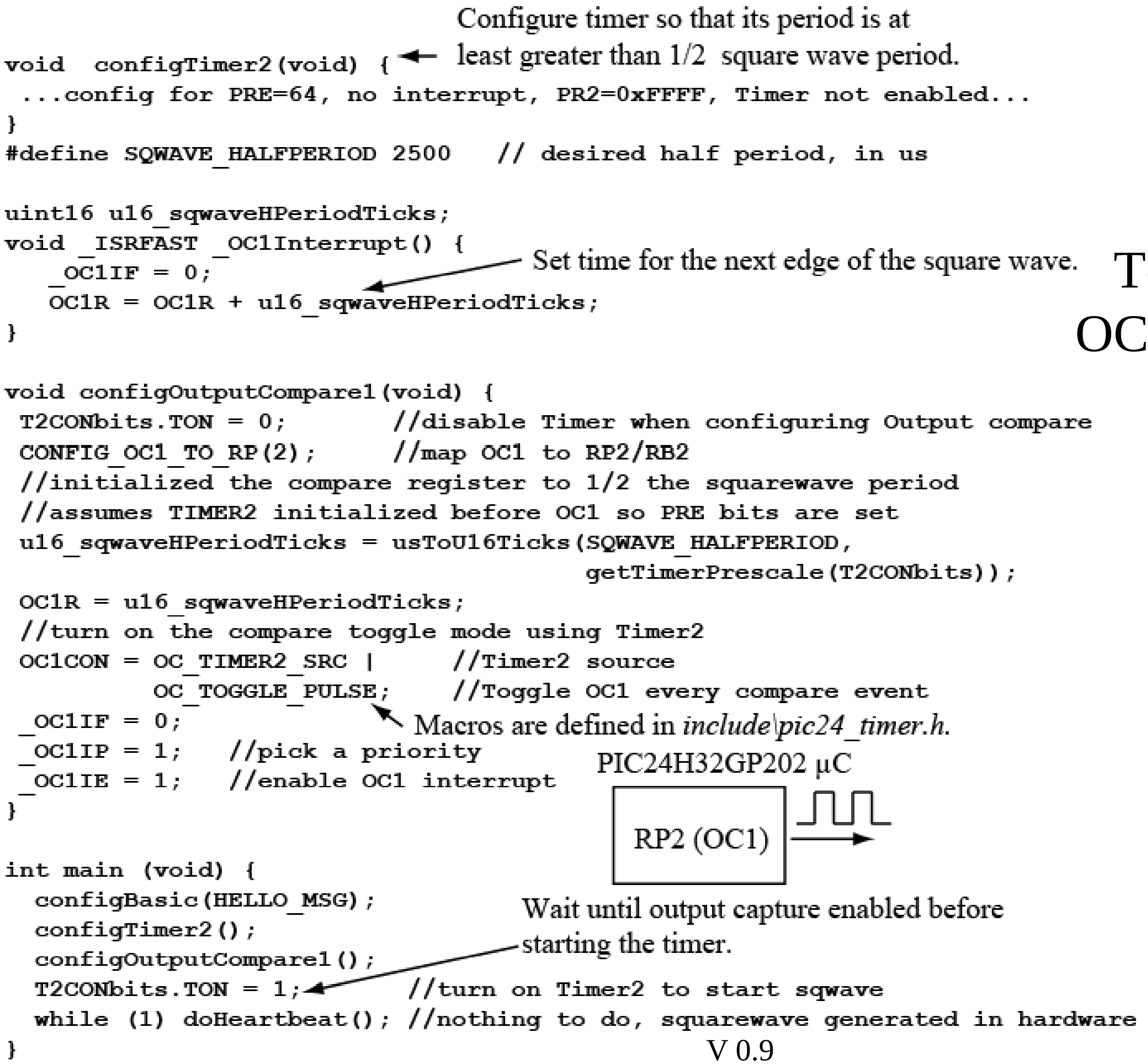
OC1R = TimerTicks_onehalfSquareWavePeriod

c. Each match of OC1R register generates an OC1 interrupt, toggles the OC1 pin.

d. In OC1 ISR, assign new OC1R register value as:

OC1R = OC1R + TimerTicks_onehalfSquareWavePeriod

Toggle Mode, OC1 square wave



Configure timer so that its period is at least greater than 1/2 square wave period.

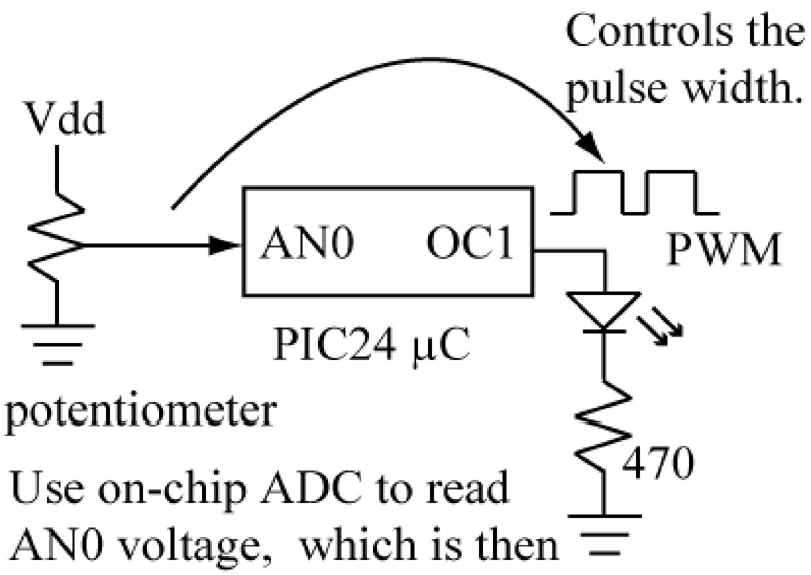
```
void  configTimer2(void) {
 ...config for PRE=64, no interrupt, PR2=0xFFFF, Timer not enabled...
}
#define SQWAVE_HALFPERIOD 2500   // desired half period, in us

uint16 u16_sqwaveHPeriodTicks;
void _ISRFAST _OC1Interrupt() {
   _OC1IF = 0;
   OC1R = OC1R + u16_sqwaveHPeriodTicks;
}

void configOutputCompare1(void) {
 T2CONbits.TON = 0;       //disable Timer when configuring Output compare
 CONFIG_OC1_TO_RP(2);     //map OC1 to RP2/RB2
 //initialized the compare register to 1/2 the squarewave period
 //assumes TIMER2 initialized before OC1 so PRE bits are set
 u16_sqwaveHPeriodTicks = usToU16Ticks(SQWAVE_HALFPERIOD,
                                       getTimerPrescale(T2CONbits));
 OC1R = u16_sqwaveHPeriodTicks;
 //turn on the compare toggle mode using Timer2
 OC1CON = OC_TIMER2_SRC |     //Timer2 source
          OC_TOGGLE_PULSE;    //Toggle OC1 every compare event
 _OC1IF = 0;
 _OC1IP = 1;   //pick a priority
 _OC1IE = 1;   //enable OC1 interrupt
}

int main (void) {
  configBasic(HELLO_MSG);
  configTimer2();
  configOutputCompare1();
  T2CONbits.TON = 1;        //turn on Timer2 to start sqwave
  while (1) doHeartbeat(); //nothing to do, squarewave generated in hardware
}
```

Set time for the next edge of the square wave.

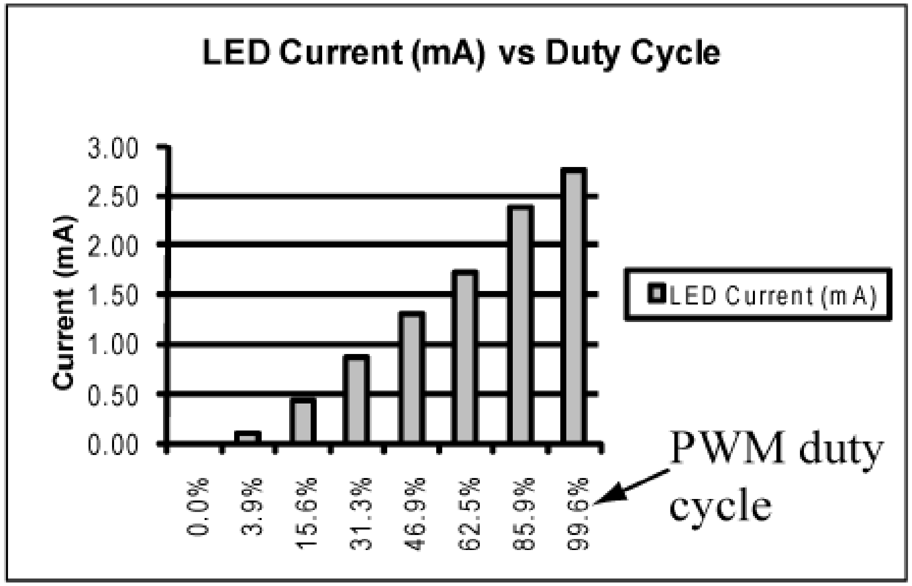
Macros are defined in *include\pic24_timer.h*.

Wait until output capture enabled before starting the timer.

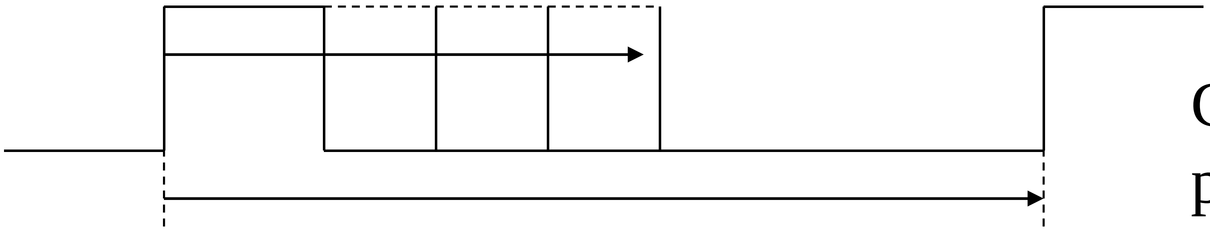
Pulse Width Modulation (PWM)



Use on-chip ADC to read AN0 voltage, which is then used to set the PWM pulse width.



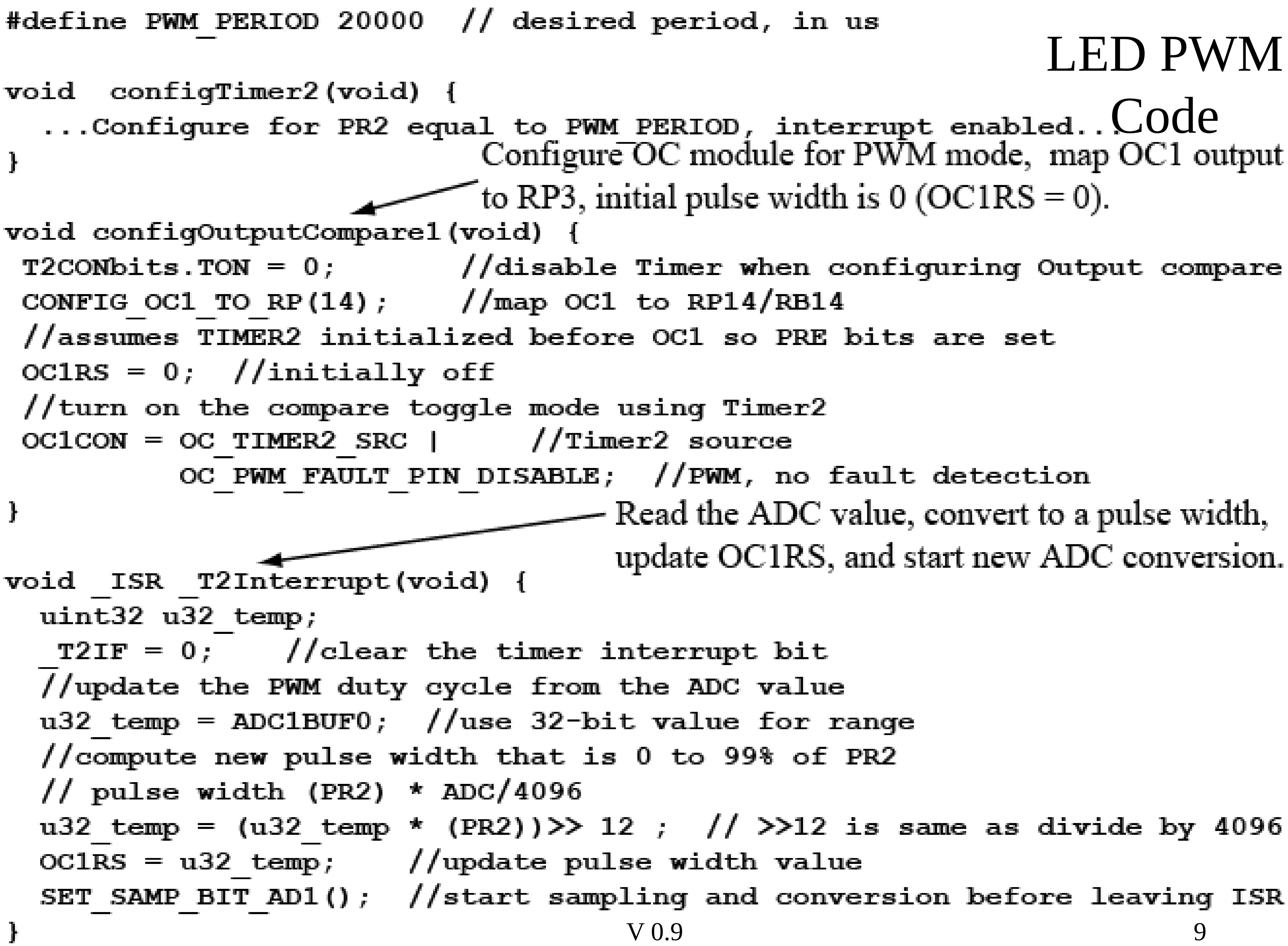
High pulse width varies (i.e., duty cycle varies)



period is fixed

Current to LED is proportional to high pulse width

LED PWM Code

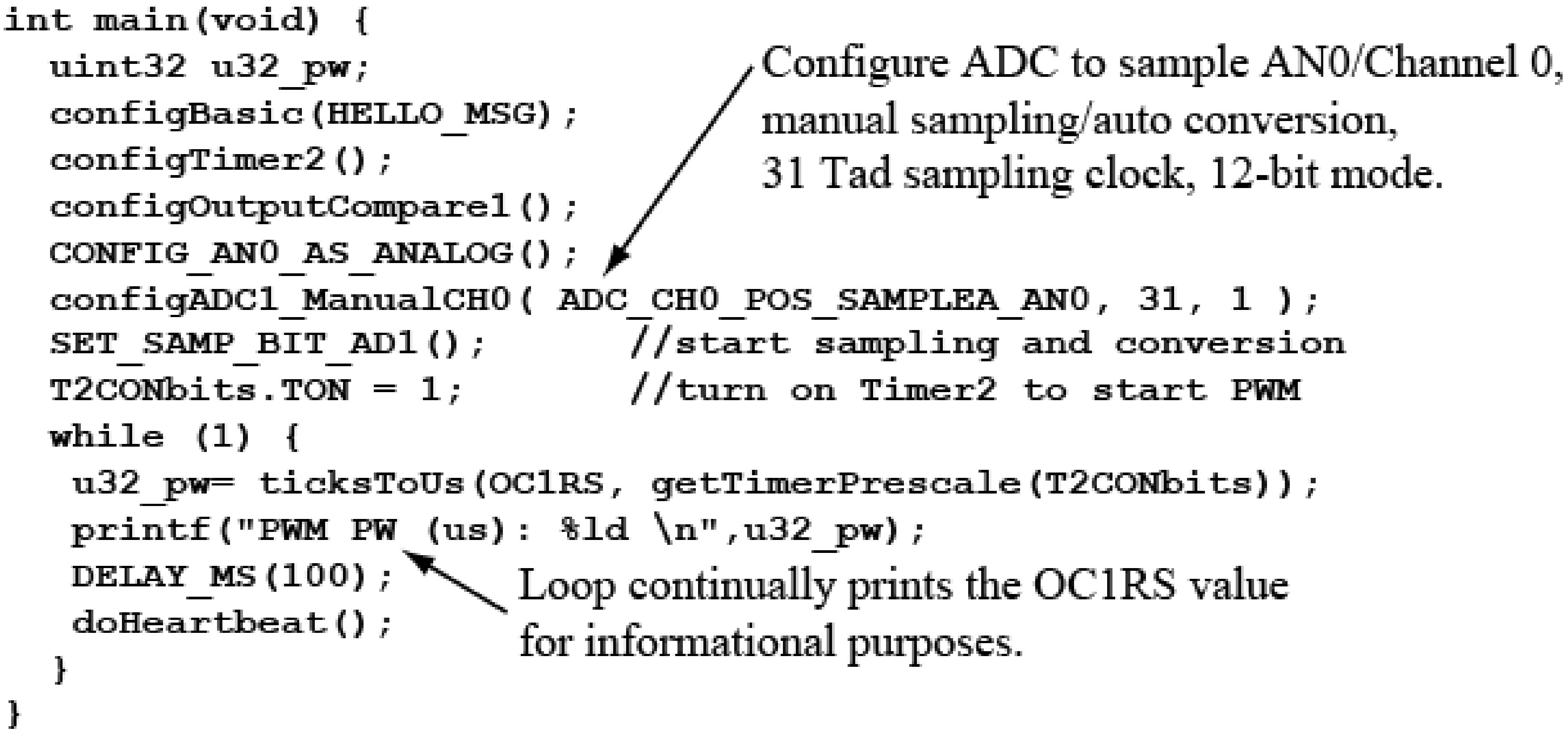


```
#define PWM_PERIOD 20000  // desired period, in us

void  configTimer2(void) {
  ...Configure for PR2 equal to PWM_PERIOD, interrupt enabled...
}

void configOutputCompare1(void) {
 T2CONbits.TON = 0;       //disable Timer when configuring Output compare
 CONFIG_OC1_TO_RP(14);    //map OC1 to RP14/RB14
 //assumes TIMER2 initialized before OC1 so PRE bits are set
 OC1RS = 0;  //initially off
 //turn on the compare toggle mode using Timer2
 OC1CON = OC_TIMER2_SRC |      //Timer2 source
          OC_PWM_FAULT_PIN_DISABLE;  //PWM, no fault detection
}

void _ISR _T2Interrupt(void) {
  uint32 u32_temp;
  _T2IF = 0;    //clear the timer interrupt bit
  //update the PWM duty cycle from the ADC value
  u32_temp = ADC1BUF0;  //use 32-bit value for range
  //compute new pulse width that is 0 to 99% of PR2
  // pulse width (PR2) * ADC/4096
  u32_temp = (u32_temp * (PR2))>> 12 ;  // >>12 is same as divide by 4096
  OC1RS = u32_temp;    //update pulse width value
  SET_SAMP_BIT_AD1();  //start sampling and conversion before leaving ISR
}
```

Configure OC module for PWM mode, map OC1 output to RP3, initial pulse width is 0 (OC1RS = 0).

Read the ADC value, convert to a pulse width, update OC1RS, and start new ADC conversion.

LED PWM Code main()



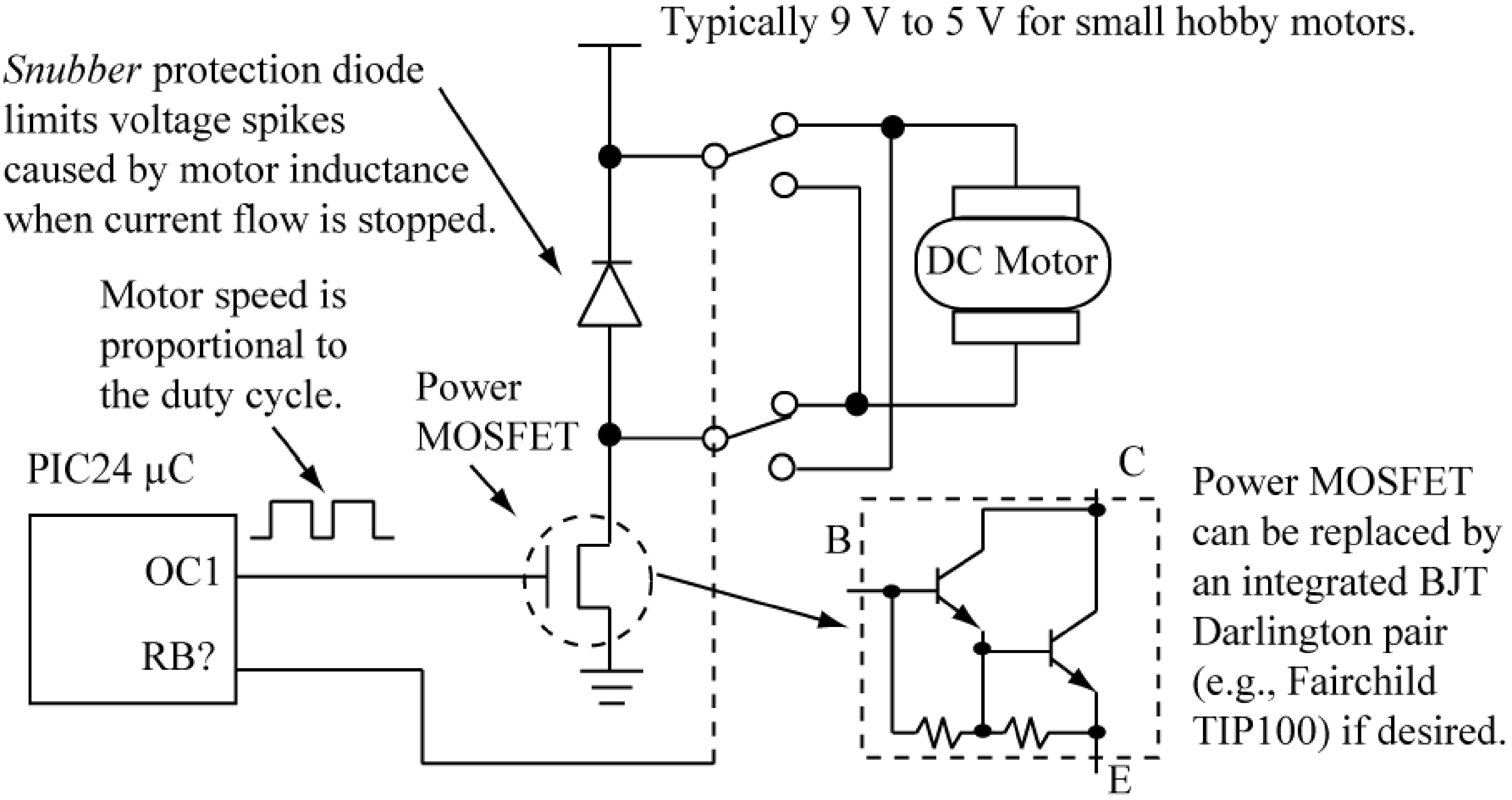
```
int main(void) {
  uint32 u32_pw;
  configBasic(HELLO_MSG);
  configTimer2();
  configOutputCompare1();
  CONFIG_AN0_AS_ANALOG();
  configADC1_ManualCH0( ADC_CH0_POS_SAMPLEA_AN0, 31, 1 );
  SET_SAMP_BIT_AD1();      //start sampling and conversion
  T2CONbits.TON = 1;       //turn on Timer2 to start PWM
  while (1) {
   u32_pw= ticksToUs(OC1RS, getTimerPrescale(T2CONbits));
   printf("PWM PW (us): %ld \n",u32_pw);
   DELAY_MS(100);
   doHeartbeat();
  }
}
```

Configure ADC to sample AN0/Channel 0,
manual sampling/auto conversion,
31 Tad sampling clock, 12-bit mode.

Loop continually prints the OC1RS value
for informational purposes.

While(1) loop just prints debugging information, work is actually done by Timer2 ISR that updates pulse width from ADC converted value.

DC Motor Speed Control



Typically 9 V to 5 V for small hobby motors.

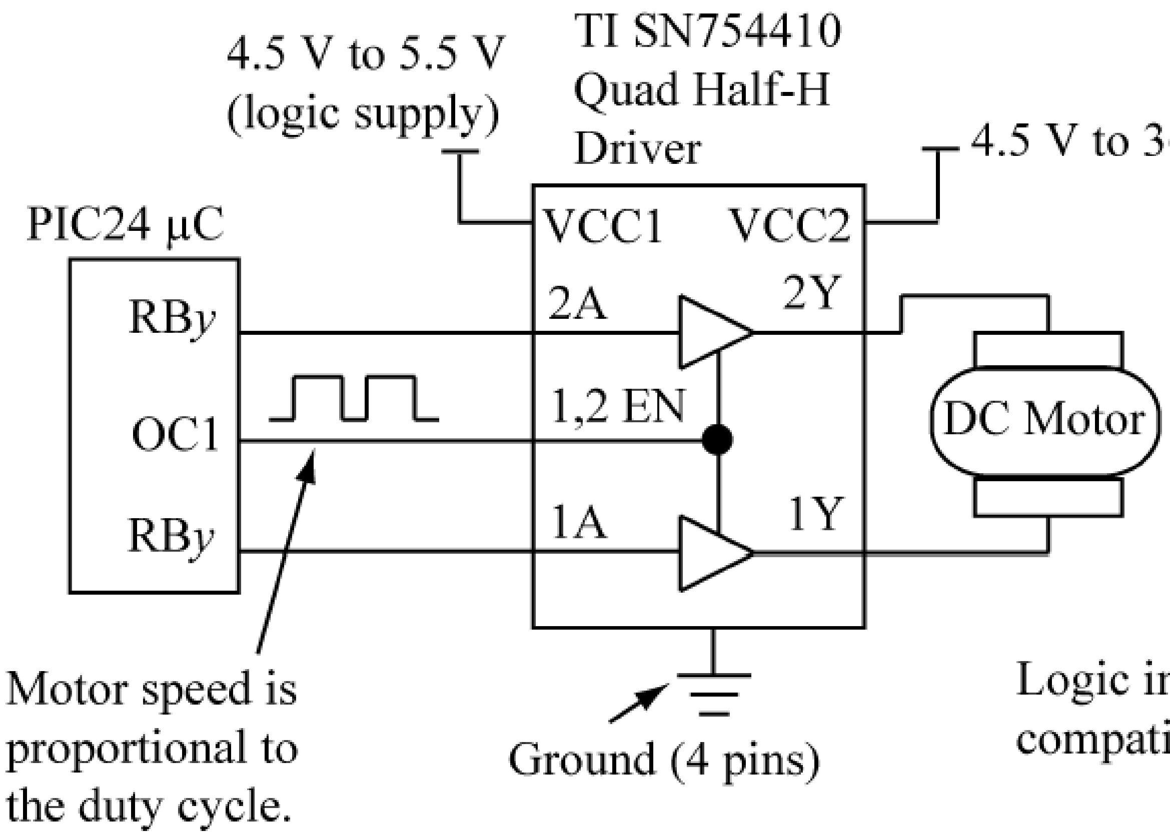
Snubber protection diode limits voltage spikes caused by motor inductance when current flow is stopped.

Motor speed is proportional to the duty cycle.

Power MOSFET

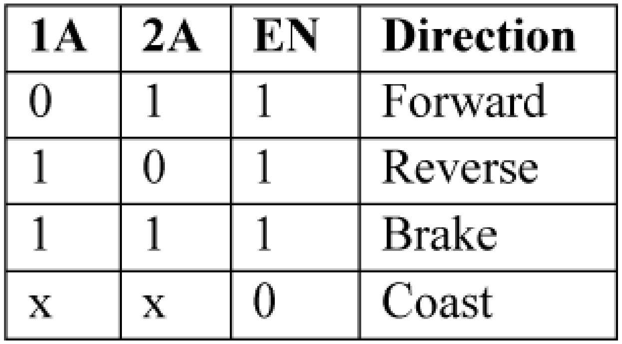
Power MOSFET can be replaced by an integrated BJT Darlington pair (e.g., Fairchild TIP100) if desired.

Half-bridge Driver



Motor speed is proportional to the duty cycle.

Ground (4 pins)



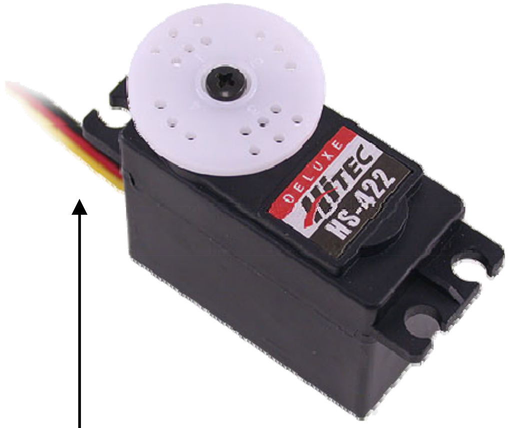
1A	2A	EN	Direction
0	1	1	Forward
1	0	1	Reverse
1	1	1	Brake
x	x	0	Coast

Logic inputs to the SN754410 are TTL compatible (VIH minimum is 2 V).

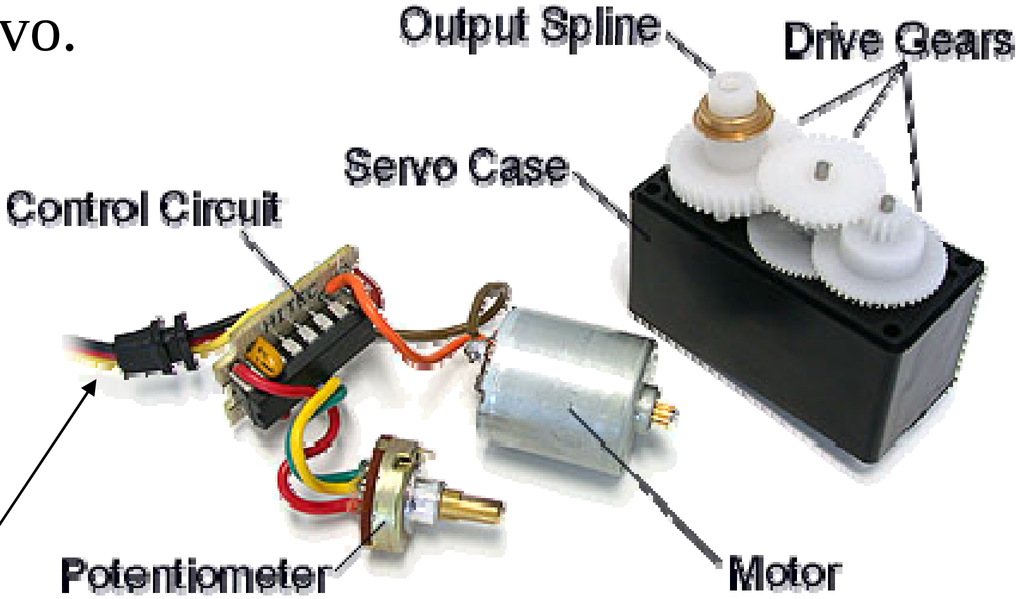
Integrates MOSFET/BJT drivers, protection diodes, switches.

Servos

Servos, widely used to steer model cars, airplanes, and boats, consist of a motor with gearing to reduce the output speed and increase output torque and a control circuit which spins the motor until the motor’s position measured by the potentiometer matches the desired position specified by a pulse supplied to the servo.



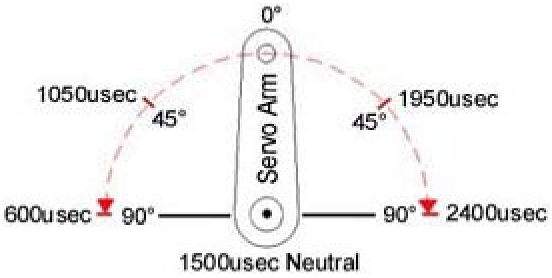
=



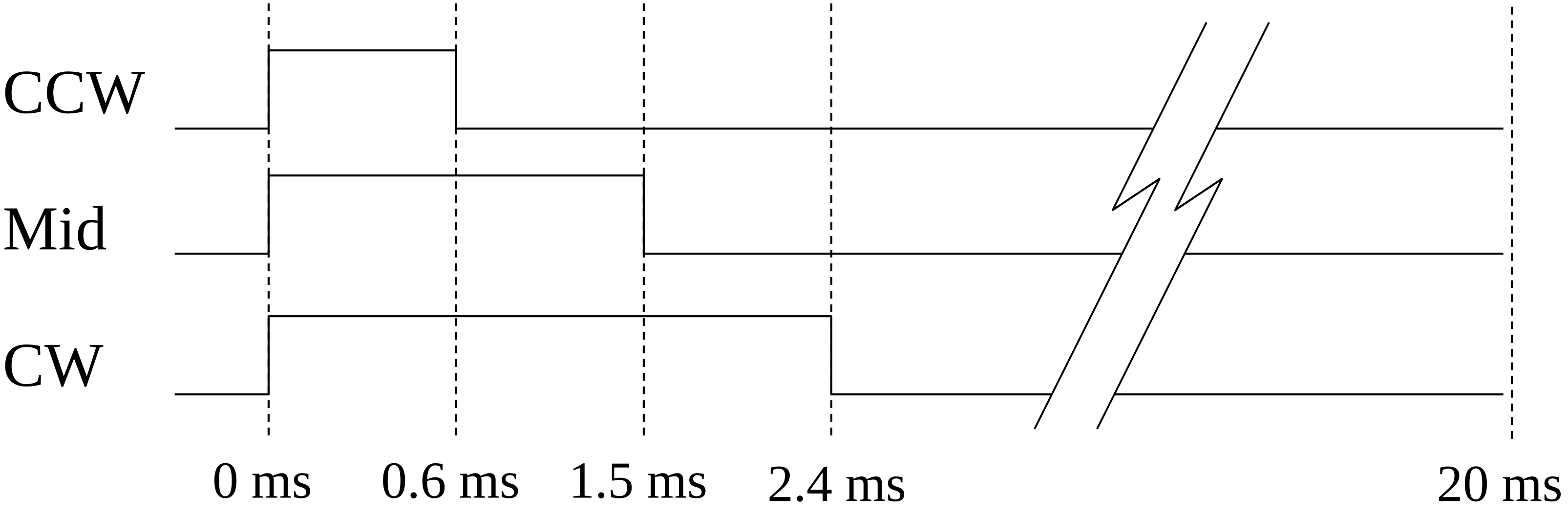
Cable: red = power (4.8 V – 6.0 V),
black = ground,
yellow = desired position

Controlling servos (HS-311)

The high time of a pulse gives the desired position of the servos. The pulse width must be 20 ms. A high time of 1.5 ms moves the servo to its center position; smaller or larger values moves it clockwise or counter-clockwise from the center position.



From www.servocity.com



Not all servos can cover this entire range!

Changes to LED PWM Code to convert Potentiometer input to control servo

```
#define PWM_PERIOD 20000  // desired period, in us
#define MIN_PW 600        //minimum pulse width, in us
#define MAX_PW 2400       //maximum pulse width, in us

uint16 u16_minPWTicks, u16_maxPWTicks;
void configOutputCapture1(void) {
 u16_minPWTicks = usToU16Ticks(MIN_PW, getTimerPrescale(T2CONbits));
 u16_maxPWTicks = usToU16Ticks(MAX_PW, getTimerPrescale(T2CONbits));
 ...rest of the function is the same...

}

void _ISR _T2Interrupt(void) {
  uint32 u32_temp;
  _T2IF = 0;    //clear the timer interrupt bit
  //update the PWM duty cycle from the ADC value
  u32_temp = ADC1BUF0;  //use 32-bit value for range
  //compute new pulse width using ADC value
  // ((max - min) * ADC)/4096 + min
  u32_temp = ((u32_temp * (u16_maxPWTicks-u16_minPWTicks))>> 12) +
              u16_minPWTicks;  // >>12 is same as divide/4096
  OC1RS = u32_temp;  //update pulse width value
  AD1CON1bits.SAMP = 1; //start next ADC conversion for next interrupt
}
```

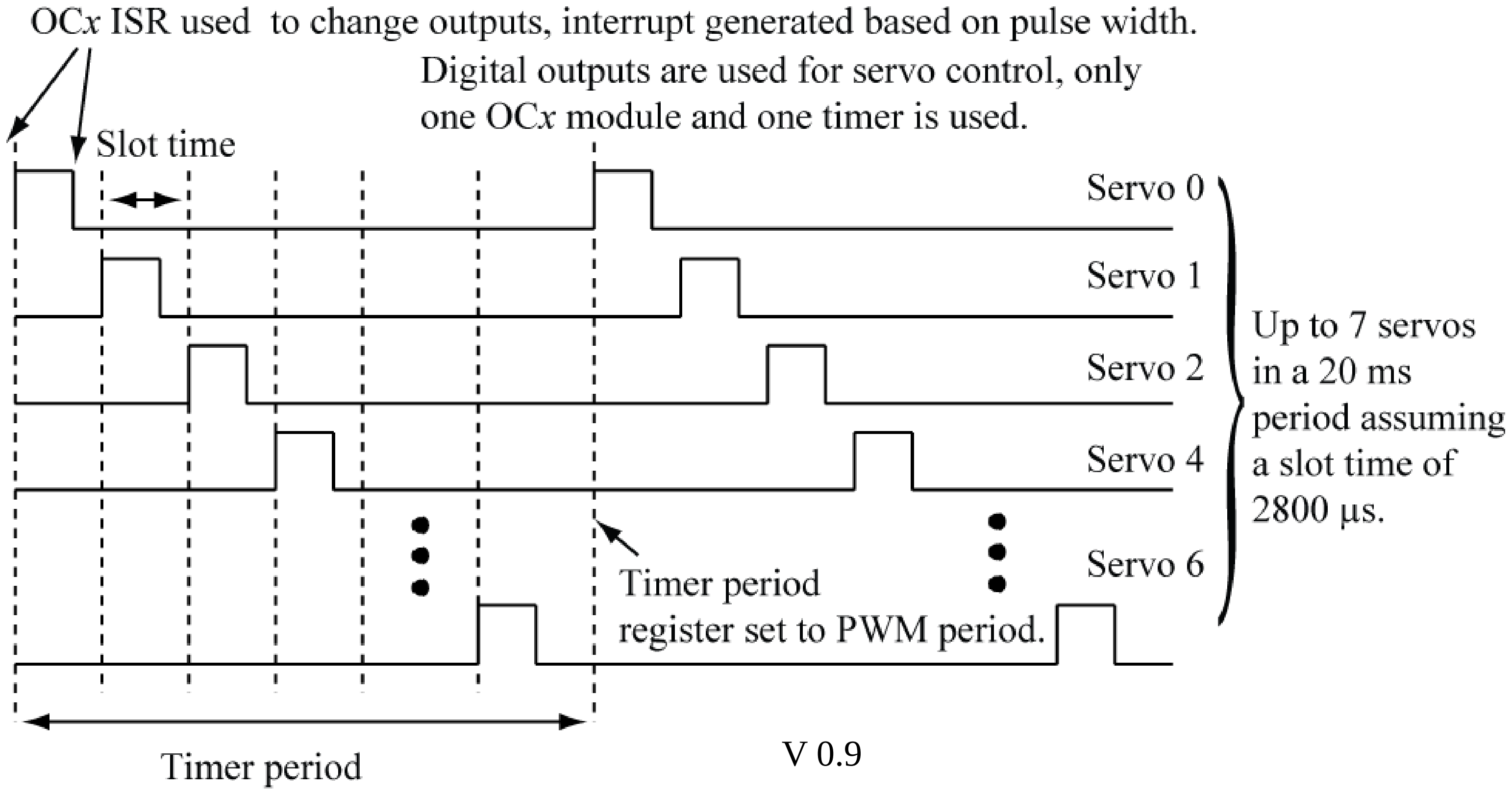
Map the potentiometer voltage to equivalent pulse-width range of the servo.

Controlling Multiple Servos

The PIC24HJ32GP202 has two output compare modules. How do you control more than two servos?

Solution:

Do not dedicate an OCx output per servo, use just one Output Compare module. Use RBx pins to control the servos, and use the OCx ISR to update the RBx pins.



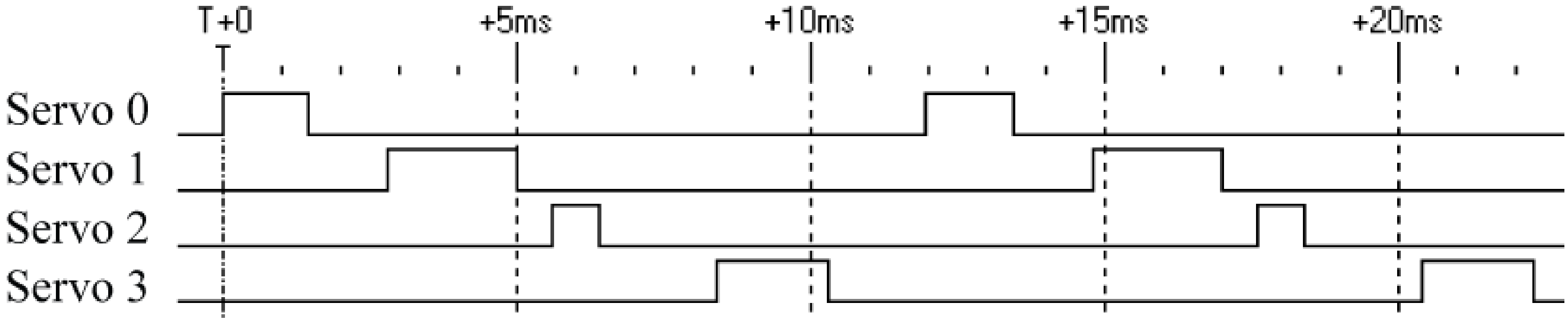
Multiple Servo Control Code

```
#define PWM_PERIOD 20000   //in microseconds
#define NUM_SERVOS 4
#define SERVO0  _LATB2
#define SERVO1  _LATB3
#define SERVO2  _LATB13
#define SERVO3  _LATB14
#define MIN_PW  600            //minimum pulse width, in us
#define MAX_PW 2400            //minimum pulse width, in us
#define SLOT_WIDTH 2800        //slot width, in us
volatile uint16 au16_servoPWidths[NUM_SERVOS];
volatile uint8 u8_currentServo =0;
volatile uint8 u8_servoEdge = 1; //1 = RISING, 0 = FALLING
volatile uint16 u16_slotWidthTicks = 0;
void initServos(void) {
  uint8 u8_i;
  uint16 u16_initPW;
  CONFIG_RB2_AS_DIG_OUTPUT(); CONFIG_RB3_AS_DIG_OUTPUT();
  CONFIG_RB13_AS_DIG_OUTPUT();  CONFIG_RB14_AS_DIG_OUTPUT();
  u16_initPW = usToU16Ticks(MIN_PW + (MAX_PW-MIN_PW)/2,
                              getTimerPrescale(T2CONbits));
//config all servos for half maximum pulse width
  for (u8_i=0; u8_i<NUM_SERVOS; u8_i++) au16_servoPWidths[u8_i]=u16_initPW;
  SERVO0 = 0; //all servo outputs low initially
  SERVO1 = 0; SERVO2 = 0; SERVO3 = 0;  //outputs initially low
  u16_slotWidthTicks = usToU16Ticks(SLOT_WIDTH, getTimerPrescale(T2CONbits));
}
```

Multiple Servo Control Code (cont.)

```
void  configTimer2(void) {
  T2CON = T2_OFF | T2_IDLE_CON | T2_GATE_OFF
          | T2_32BIT_MODE_OFF
          | T2_SOURCE_INT
          | T2_PS_1_256 ;  //1 tick = 1.6 us at FCY=40 MHz
  PR2 = usToU16Ticks(PWM_PERIOD,
getTimerPrescale(T2CONbits)) - 1;
  TMR2 = 0;       //clear timer2 value
}
void configOutputCapture1(void) {
 T2CONbits.TON = 0;       //disable Timer when configuring
Output compare
 OC1R =  0;  //initialize to 0, first match will be a first
timer rollover.
 //turn on the compare toggle mode using Timer2
 OC1CON = OC_TIMER2_SRC |     //Timer2 source
          OC_TOGGLE_PULSE;    //use toggle mode, just care
about compare event
 _OC1IF = 0; _OC1IP = 1; _OC1IE = 1;    //enable the OC1
interrupt
}
```

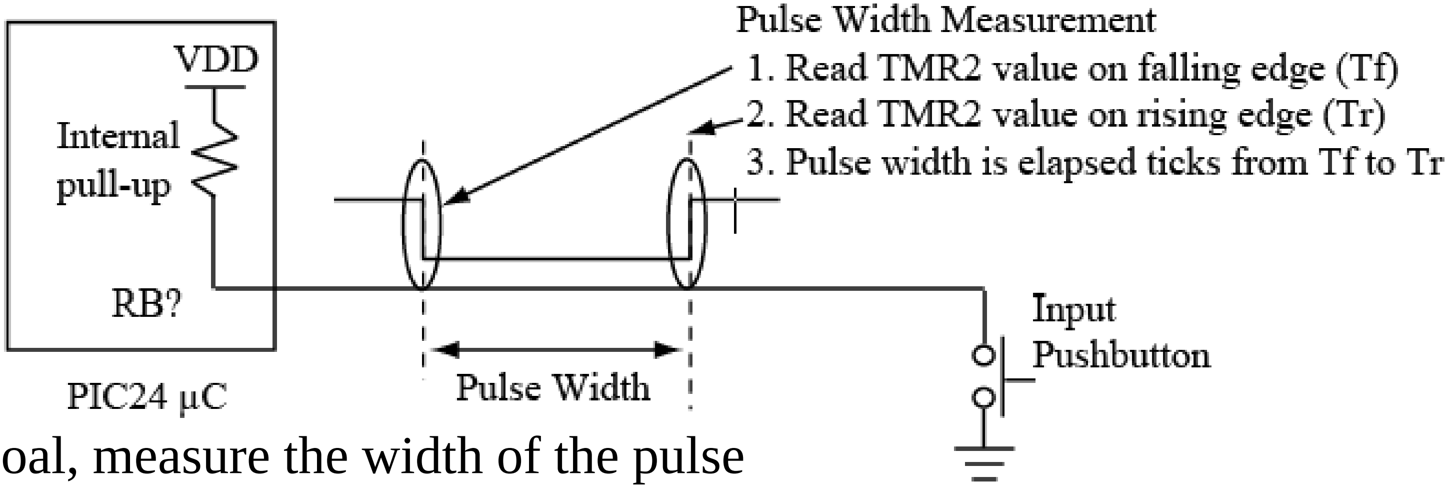
Screenshot of four servo control



Period set to 12 ms to emphasize pulse width differences.

Period set to 10 ms instead of 20 ms to emphasize pulse-width differences.

Time Measurement



A simple goal, measure the width of the pulse

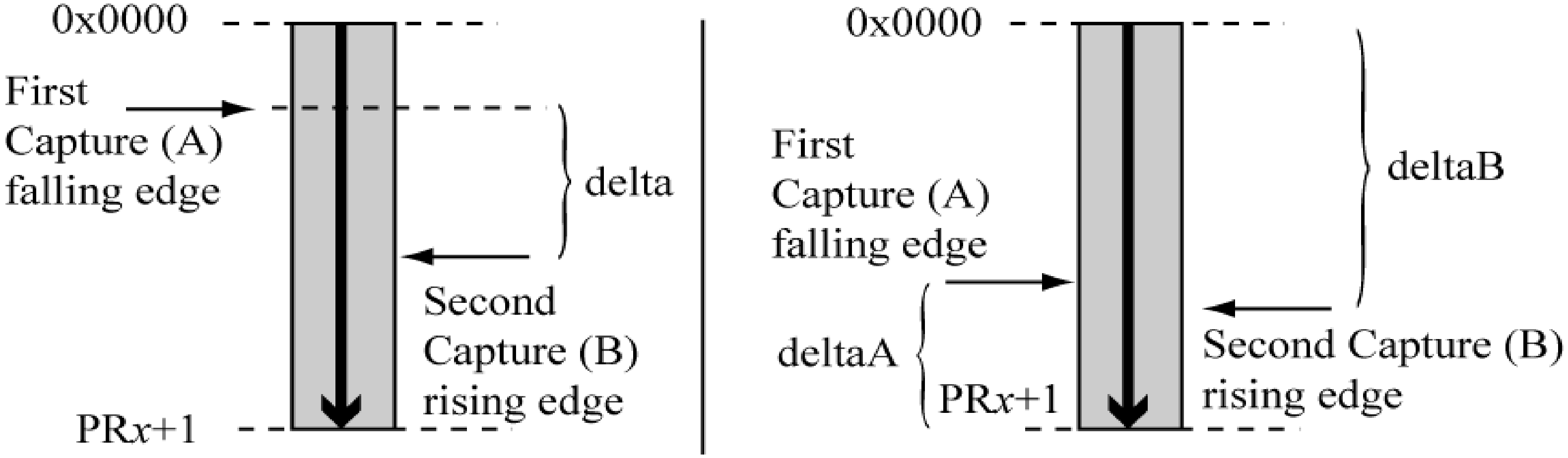
If Timer2 PR2 = 0xFFFF, then pulse width is:

PW = TMR2_rising – TMR2 _falling; (total pulse width must be less than 2^{16} ticks (timer period)!! Works even if one timer rollover occurs.)

If PR2 != 0xFFFF then: (assumes that at most only one rollover occurs, total pulse width is less than PR2+1 ticks, i.e., timer period)

```
if ((TMR2_rising > TMR2 _falling) && no timer rollover)
        PW = TMR2_rising – TMR2 _falling
else
      PW = TMR2_rising +  (PR2+1) – TMR2_falling
```

Delta Computation



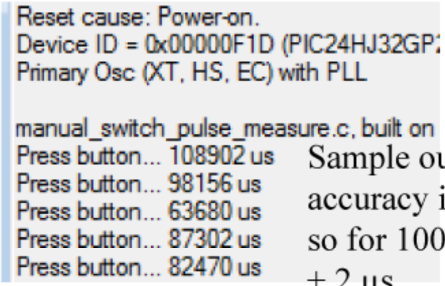
if PR2 != 0xFFFF then: (assumes that at most only one rollover occurs, total pulse width is less than PR2+1 ticks)

```
if ((TMR2_rising > TMR2 _falling)  && no timer rollover)
        PW = TMR2_rising – TMR2 _falling
else
      PW = TMR2_rising +  (PR2+1) – TMR2_falling
```

Code

```
void  configTimer2(void) {
  T2CON = T2_OFF | T2_IDLE_CON | T2_GATE_OFF
          | T2_32BIT_MODE_OFF
          | T2_SOURCE_INT
          | T2_PS_1_256 ;   //@40 MHz, ~420 ms period, 1 tick = 6.4 us
  PR2 = 0xFFFF;             //maximum period
  TMR2  = 0;                //clear timer2 value
  _T2IF = 0;                //clear interrupt flag
  T2CONbits.TON = 1;        //turn on the timer
}

/// 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

int main (void) {
  uint16 u16_start, u16_delta;
  uint32 u32_pulseWidth;
  configBasic(HELLO_MSG);
  CONFIG_SW1();    //use RB13
  configTimer2();
  while (1) {
    outString("Press button...");
    while(SW1_RELEASED())doHeartbeat();
    u16_start = TMR2;
    while(SW1_PRESSED())doHeartbeat();
    u16_delta = TMR2 - u16_start;  //works because using maximum PR2 value
    u32_pulseWidth = ticksToUs((uint32) u16_delta,
                         getTimerPrescale(T2CONbits));
    printf(" %ld us\n",u32_pulseWidth);
  }
}
```

Compute PW as
Timer2(rising edge)-Timer2(falling edge)
Assumes PR2= 0xFFFF



```
Reset cause: Power-on.
Device ID = 0x00000F1D (PIC24HJ32GP2
Primary Osc (XT, HS, EC) with PLL

manual_switch_pulse_measure.c, built on
Press button... 108902 us
Press button... 98156 us
Press button... 63680 us
Press button... 87302 us
Press button... 82470 us
```

Sample output; Crystal accuracy is ± 20 ppm, so for 100,000 μs this is ± 2 μs.

← Read TMR2 at falling edge.

Compute delta ticks between falling and rising edge.

← Convert to microseconds.

Total pulse width < 2^{16} ticks. Inaccurate because of instruction delay, also ISRs could delay capturing of TMR2 value.

Input Capture Module

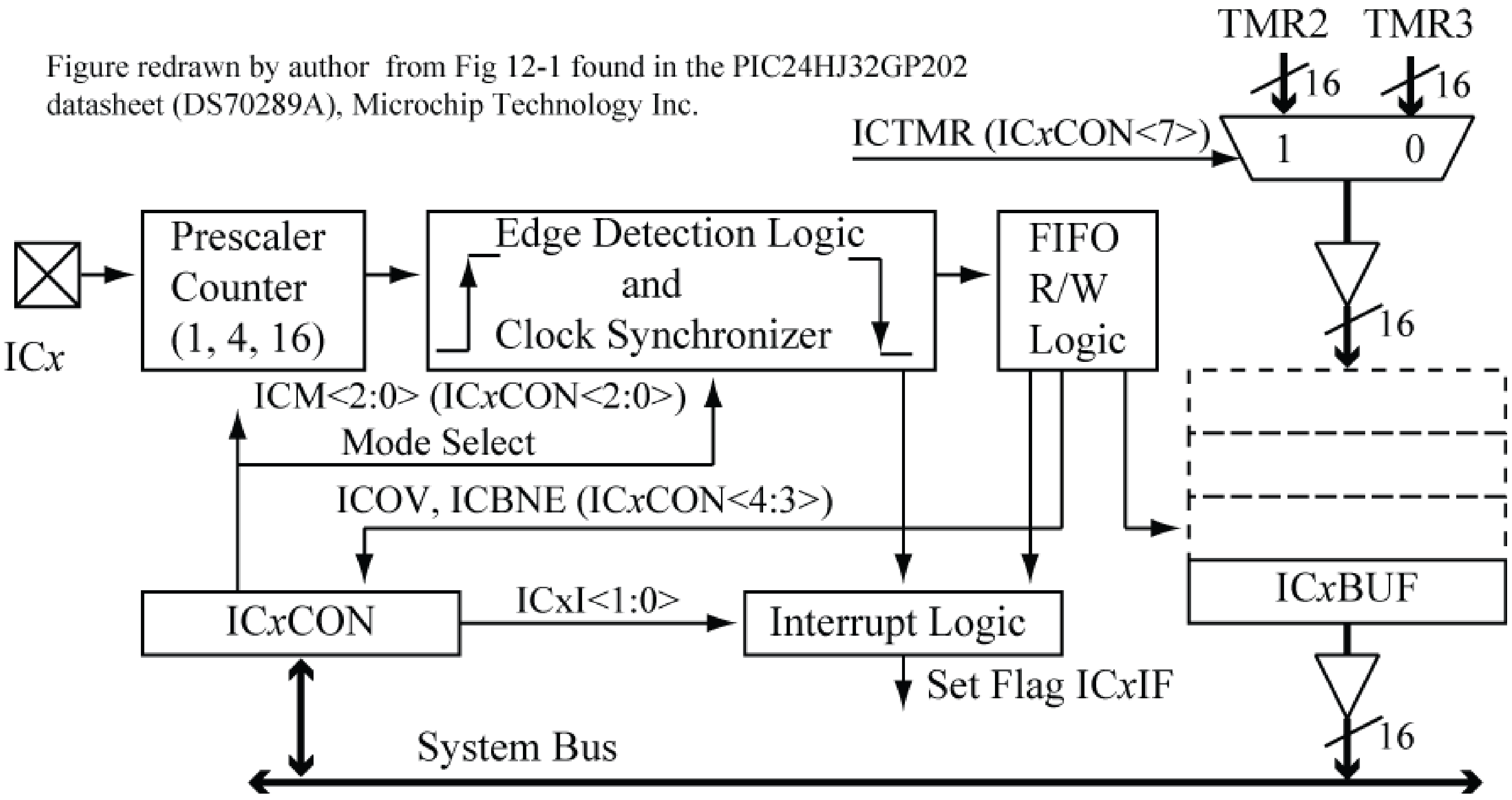
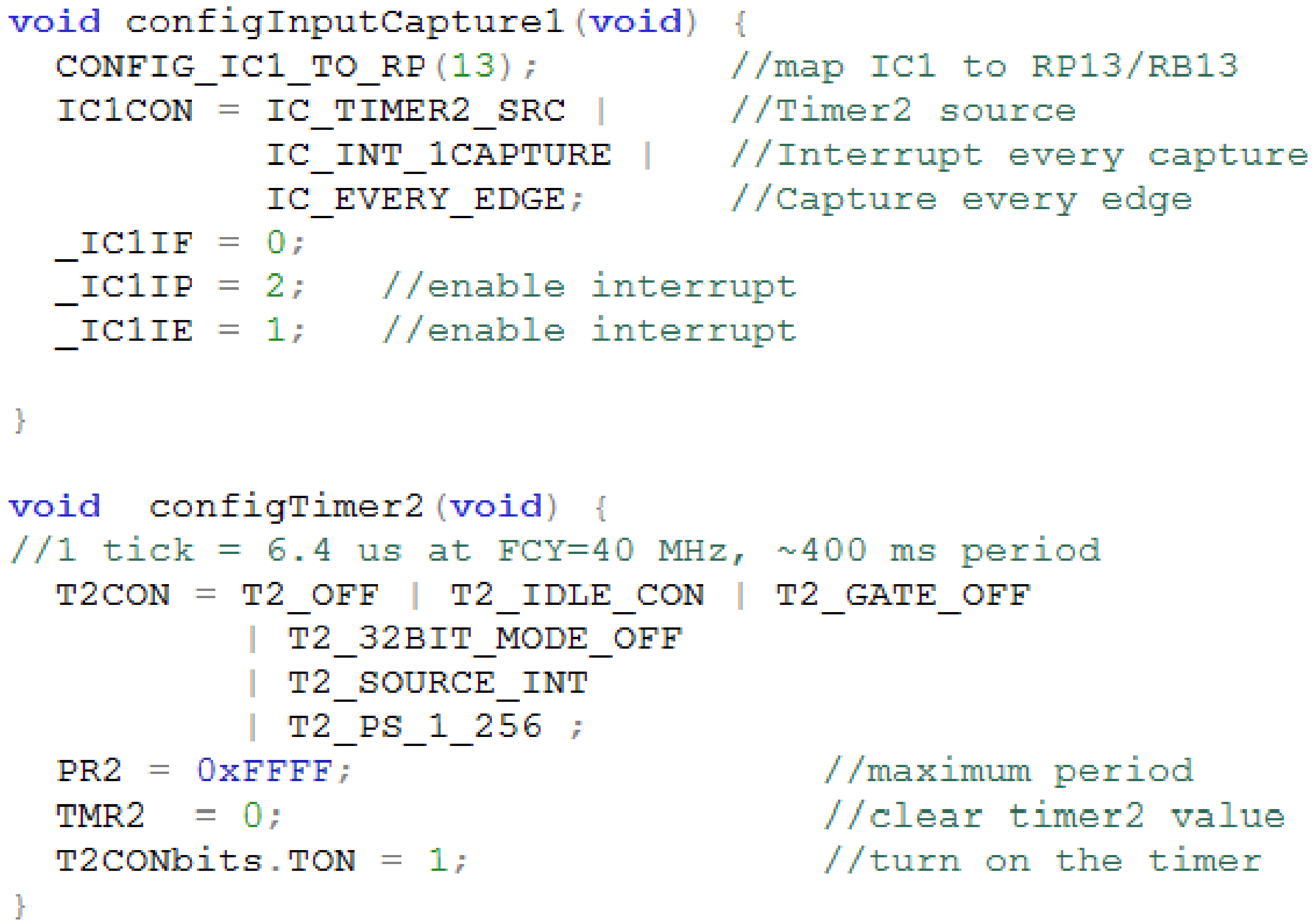


Figure redrawn by author from Fig 12-1 found in the PIC24HJ32GP202 datasheet (DS70289A), Microchip Technology Inc.

Timer value is transferred by hardware to input capture register (IC*x*) when an event occurs. The IC*x* register is really a 4-element FIFO. Use IC*x* pin for input capture.

Simple Use of Input Capture

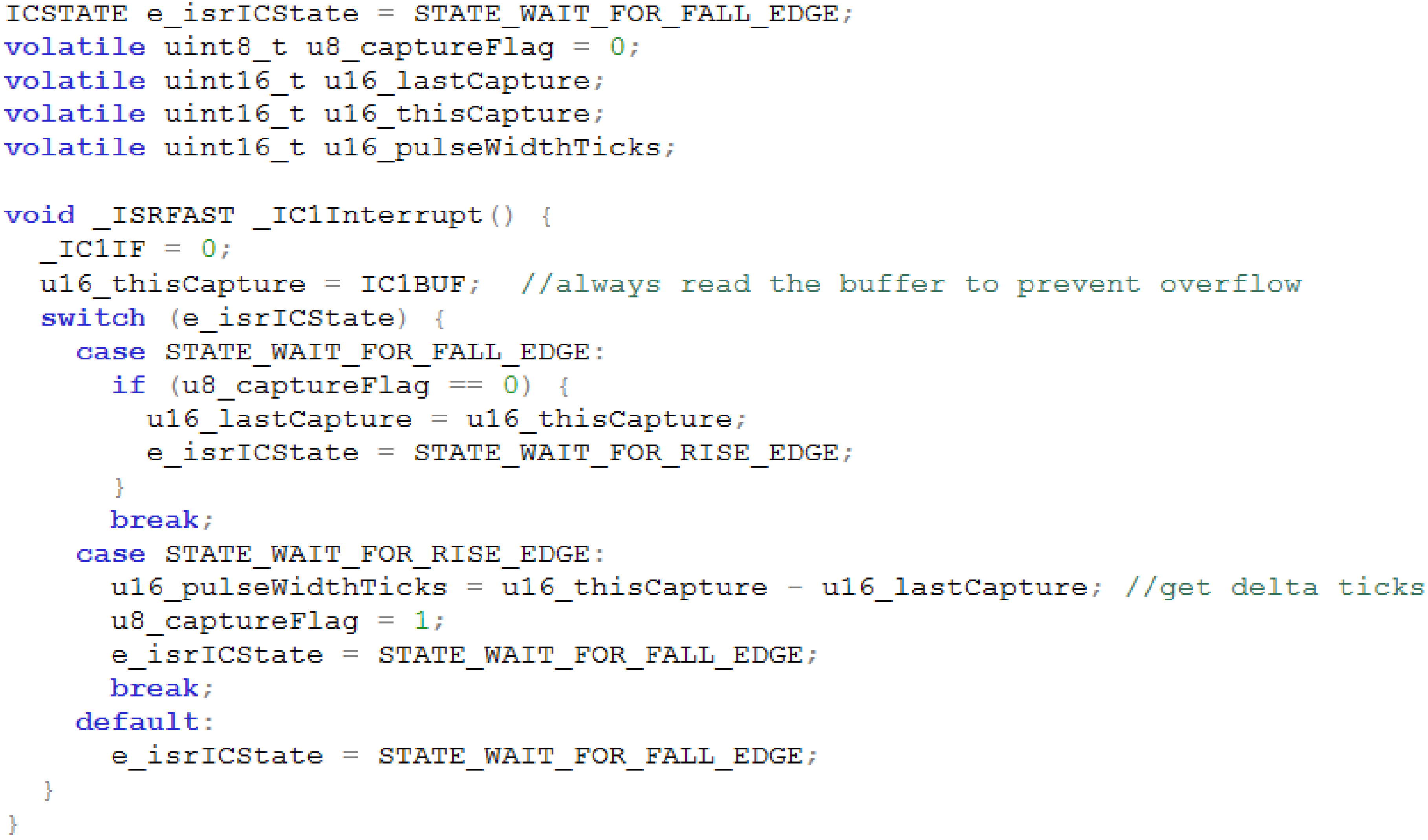
Use only one 16-bit timer, assume pulse width does not exceed timer period.



```
void configInputCapture1(void) {
  CONFIG_IC1_TO_RP(13);           //map IC1 to RP13/RB13
  IC1CON = IC_TIMER2_SRC |        //Timer2 source
           IC_INT_1CAPTURE |      //Interrupt every capture
           IC_EVERY_EDGE;         //Capture every edge
  _IC1IF = 0;
  _IC1IP = 2;   //enable interrupt
  _IC1IE = 1;   //enable interrupt

}

void  configTimer2(void) {
//1 tick = 6.4 us at FCY=40 MHz, ~400 ms period
  T2CON = T2_OFF | T2_IDLE_CON | T2_GATE_OFF
          | T2_32BIT_MODE_OFF
          | T2_SOURCE_INT
          | T2_PS_1_256 ;
  PR2 = 0xFFFF;                       //maximum period
  TMR2  = 0;                          //clear timer2 value
  T2CONbits.TON = 1;                  //turn on the timer
}
```

Simple Use of Input Capture (continued)

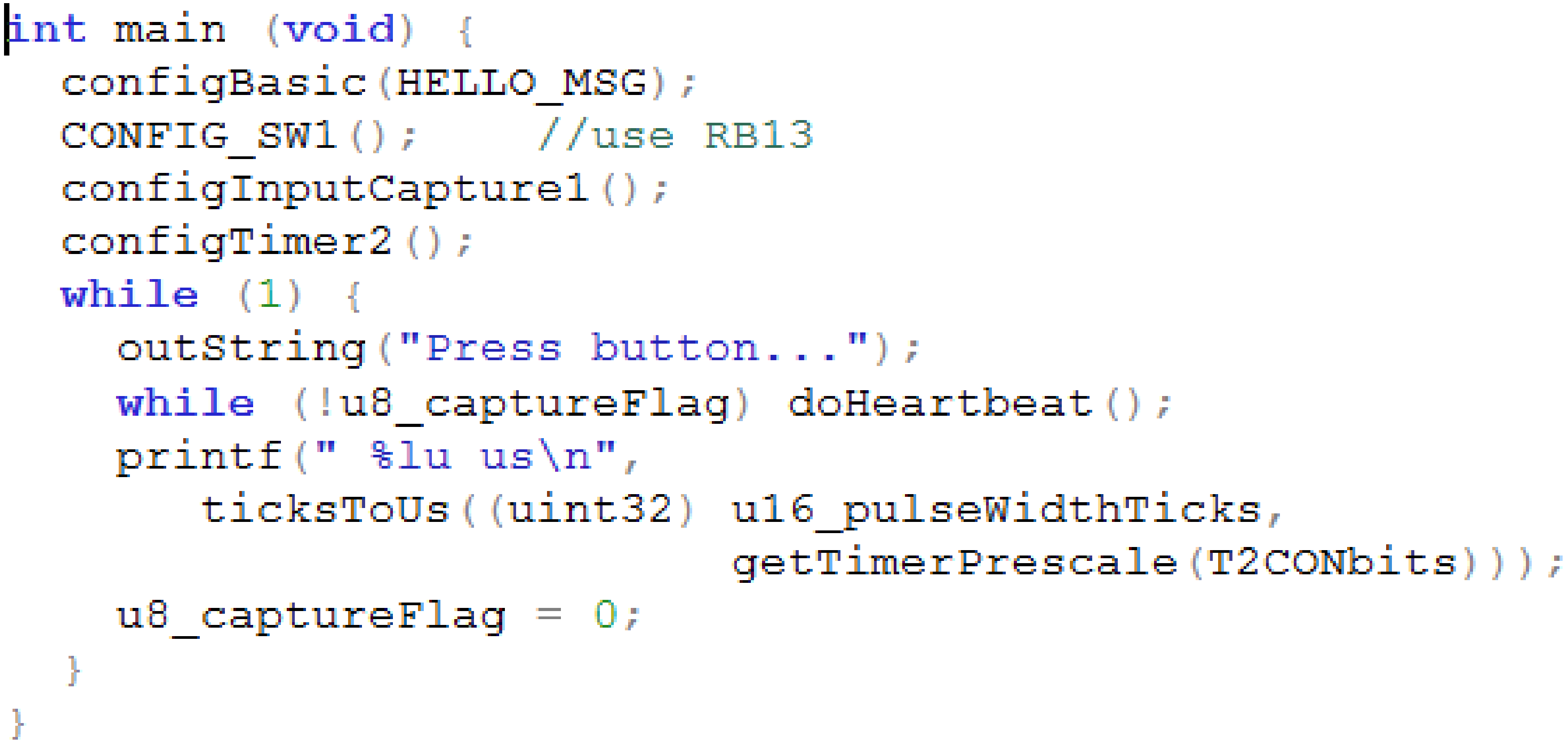
ISR computes delta ticks between fall/rise edges.



```
ICSTATE e_isrICState = STATE_WAIT_FOR_FALL_EDGE;
volatile uint8_t u8_captureFlag = 0;
volatile uint16_t u16_lastCapture;
volatile uint16_t u16_thisCapture;
volatile uint16_t u16_pulseWidthTicks;

void _ISRFAST _IC1Interrupt() {
  _IC1IF = 0;
  u16_thisCapture = IC1BUF;  //always read the buffer to prevent overflow
  switch (e_isrICState) {
    case STATE_WAIT_FOR_FALL_EDGE:
      if (u8_captureFlag == 0) {
        u16_lastCapture = u16_thisCapture;
        e_isrICState = STATE_WAIT_FOR_RISE_EDGE;
      }
      break;
    case STATE_WAIT_FOR_RISE_EDGE:
      u16_pulseWidthTicks = u16_thisCapture - u16_lastCapture; //get delta ticks
      u8_captureFlag = 1;
      e_isrICState = STATE_WAIT_FOR_FALL_EDGE;
      break;
    default:
      e_isrICState = STATE_WAIT_FOR_FALL_EDGE;
  }
}
```

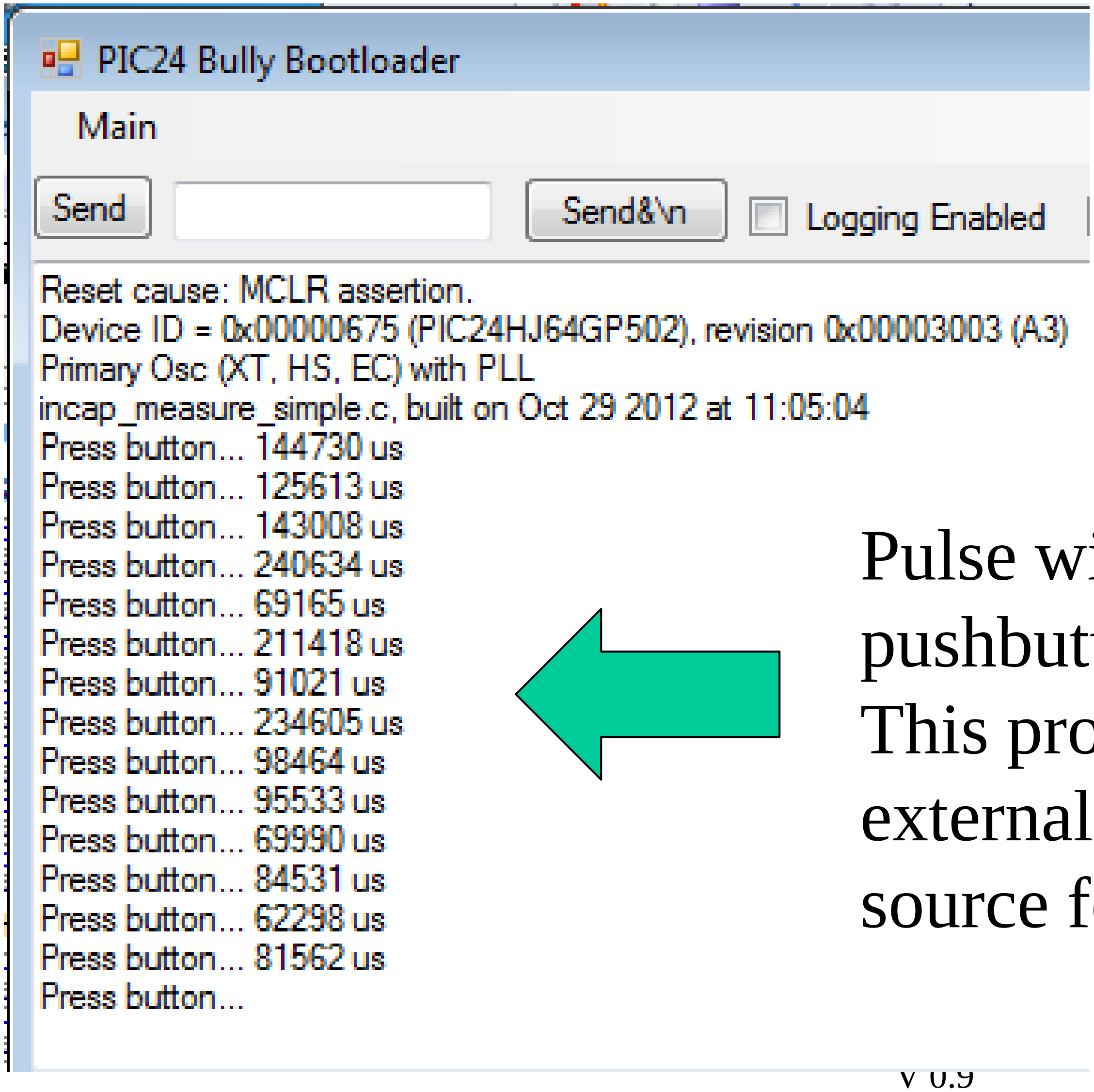
Simple Use of Input Capture (continued)

`main()` prints out result



```
int main (void) {
  configBasic(HELLO_MSG);
  CONFIG_SW1();     //use RB13
  configInputCapture1();
  configTimer2();
  while (1) {
    outString("Press button...");
    while (!u8_captureFlag) doHeartbeat();
    printf(" %lu us\n",
       ticksToUs((uint32) u16_pulseWidthTicks,
                          getTimerPrescale(T2CONbits)));
    u8_captureFlag = 0;
  }
}
```

Simple Use of Input Capture (continued)



```
Reset cause: MCLR assertion.
Device ID = 0x00000675 (PIC24HJ64GP502), revision 0x00003003 (A3)
Primary Osc (XT, HS, EC) with PLL
incap_measure_simple.c, built on Oct 29 2012 at 11:05:04
Press button... 144730 us
Press button... 125613 us
Press button... 143008 us
Press button... 240634 us
Press button... 69165 us
Press button... 211418 us
Press button... 91021 us
Press button... 234605 us
Press button... 98464 us
Press button... 95533 us
Press button... 69990 us
Press button... 84531 us
Press button... 62298 us
Press button... 81562 us
Press button...
```

Pulse width printed each time pushbutton is pressed.
This program is using an external crystal as the clock source for timing accuracy.

More on Time Measurement

What if trying to measure an interval that is longer than what one 16-bit timer can handle?

1. Use a 32-bit timer (combines Timer2/Timer3). Easy from a coding perspective, but consumes resources. See book example (chap12/timer32bit_switch_pulse_measure.c)

2. Track timer overflows and include in the equation. Results in more complex code. See book example (chap12/incap_switch_pulse_measure.c)

32-bit Timer (Timer2/3)

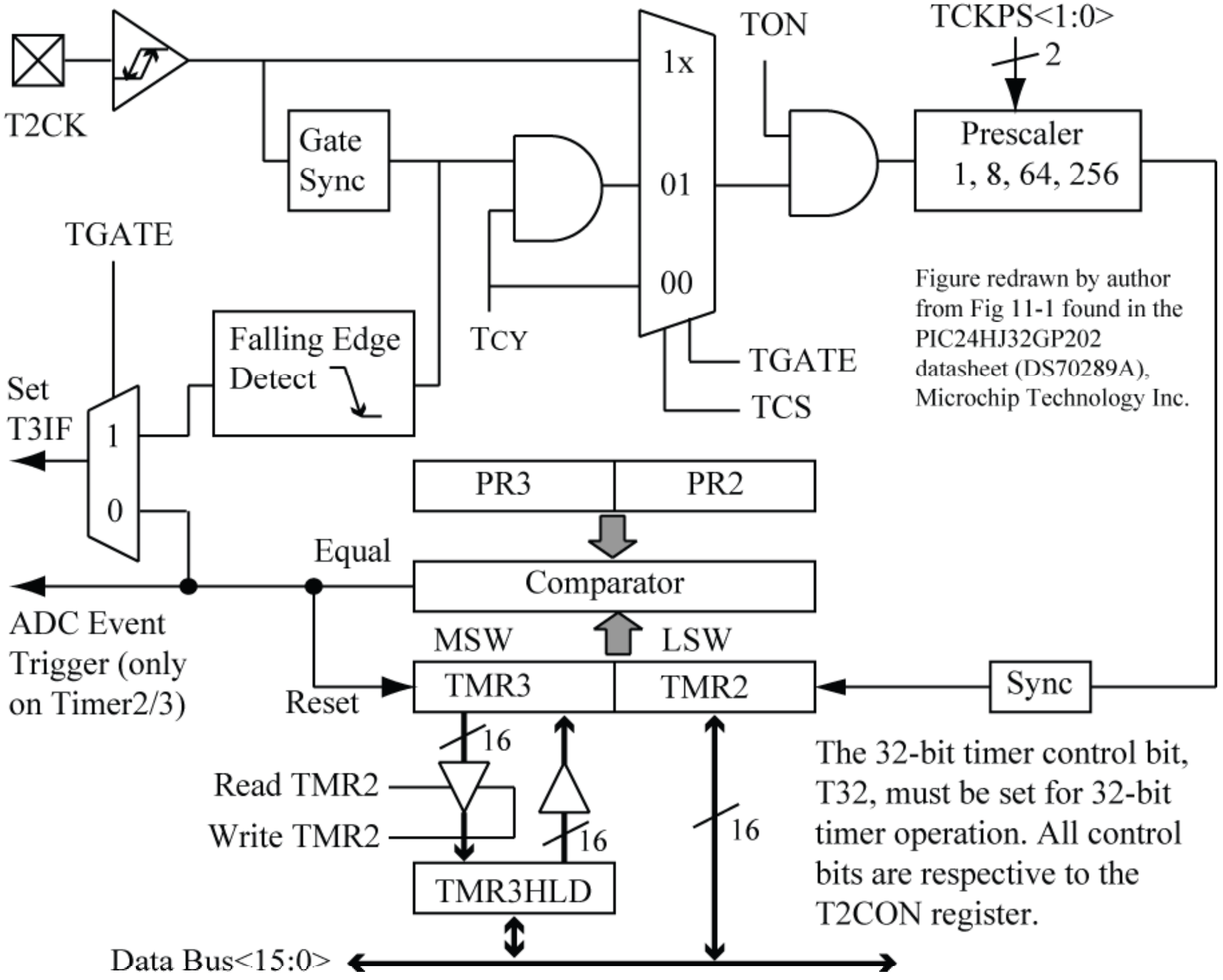


Figure redrawn by author from Fig 11-1 found in the PIC24HJ32GP202 datasheet (DS70289A), Microchip Technology Inc.

The 32-bit timer control bit, T32, must be set for 32-bit timer operation. All control bits are respective to the T2CON register.

Combines Timers2, 3 into single 32-bit timer.
Timer2 is LSW,
Timer3 is MSW.

Timer control bits are in T2CON

T3IF used for rollover.

Period Register is PR3:PR2

Read/Write of Timer2/3

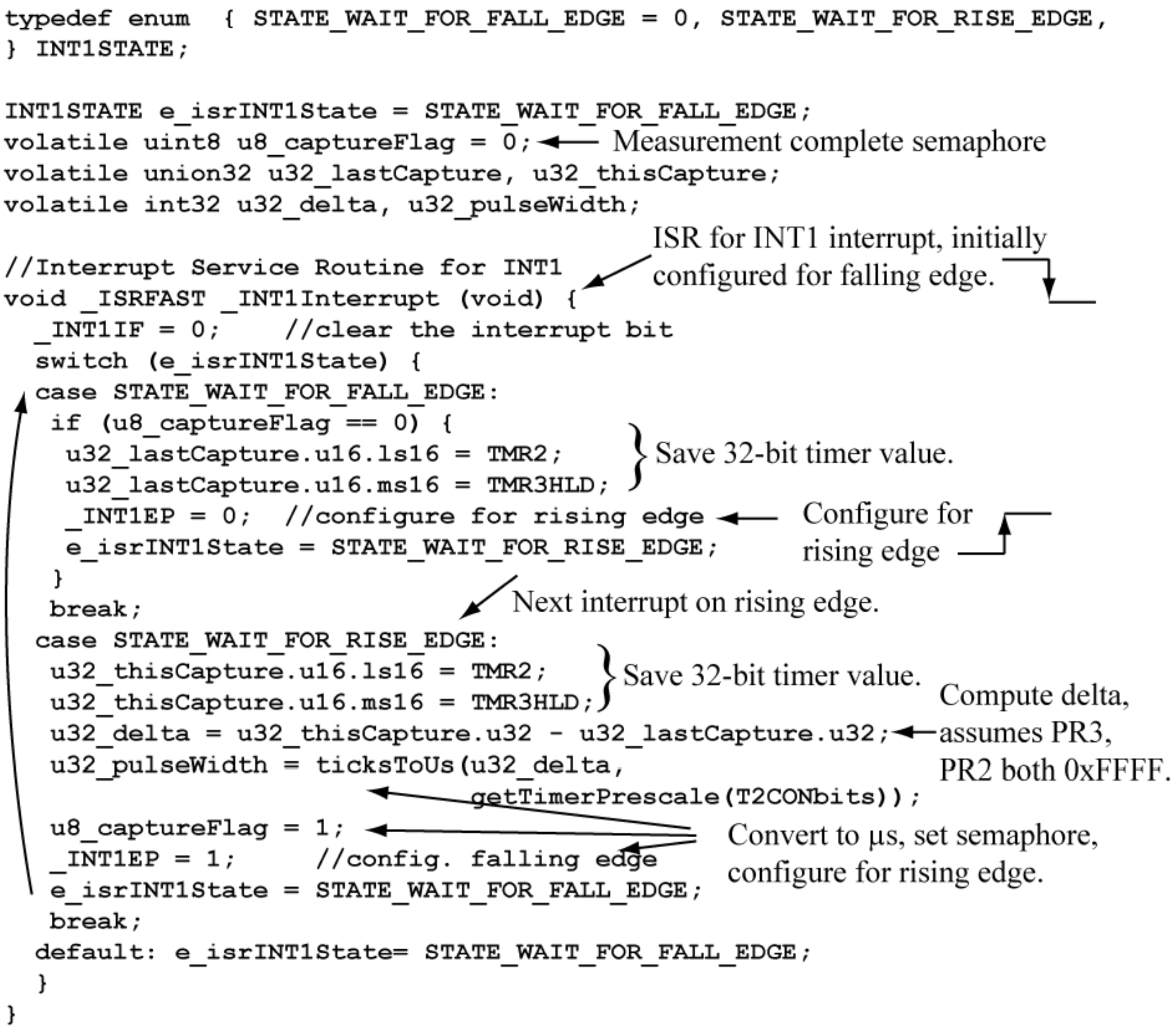
```
typedef union _union32 {
  uint32 u32;
  struct {
    uint16 ls16;
    uint16 ms16;
  } u16;
  uint8 u8[4];
} union32;

union32 write_value;
union32 read_value;
```

Write: Must write MSW first, write to TMR3HLD, which is auto transferred to TMR3 when TMR2 is written (a 32-bit update).

Read: Read LSW (TMR2) first, auto transfers TMR3 into TMR3HLD, which then can be read.

```
write_value.u32 = 0x12345678;
TMR3HLD = write_value.u16.ms16;  //write the MSW first
TMR2 = write_value.u16.ls16;     //then write the LSW
...
//read the timer
read_value.u16.ls16 = TMR2;      //read the LSW first
read_value.u16.ms16 = TMR3HLD;   //then read the MSW
```

Use 32-bit mode, Interrupt driven



```
typedef enum  { STATE_WAIT_FOR_FALL_EDGE = 0, STATE_WAIT_FOR_RISE_EDGE,
} INT1STATE;

INT1STATE e_isrINT1State = STATE_WAIT_FOR_FALL_EDGE;
volatile uint8 u8_captureFlag = 0;
volatile union32 u32_lastCapture, u32_thisCapture;
volatile int32 u32_delta, u32_pulseWidth;

//Interrupt Service Routine for INT1
void _ISRFAST _INT1Interrupt (void) {
  _INT1IF = 0;    //clear the interrupt bit
  switch (e_isrINT1State) {
  case STATE_WAIT_FOR_FALL_EDGE:
   if (u8_captureFlag == 0) {
    u32_lastCapture.u16.ls16 = TMR2;
    u32_lastCapture.u16.ms16 = TMR3HLD;
    _INT1EP = 0;  //configure for rising edge
    e_isrINT1State = STATE_WAIT_FOR_RISE_EDGE;
   }
   break;
  case STATE_WAIT_FOR_RISE_EDGE:
   u32_thisCapture.u16.ls16 = TMR2;
   u32_thisCapture.u16.ms16 = TMR3HLD;
   u32_delta = u32_thisCapture.u32 - u32_lastCapture.u32;
   u32_pulseWidth = ticksToUs(u32_delta,
                        getTimerPrescale(T2CONbits));

   u8_captureFlag = 1;
   _INT1EP = 1;      //config. falling edge
   e_isrINT1State = STATE_WAIT_FOR_FALL_EDGE;
   break;
  default: e_isrINT1State= STATE_WAIT_FOR_FALL_EDGE;
  }
}
```

Measurement complete semaphore

ISR for INT1 interrupt, initially configured for falling edge.

Save 32-bit timer value.

Configure for rising edge

Next interrupt on rising edge.

Save 32-bit timer value.

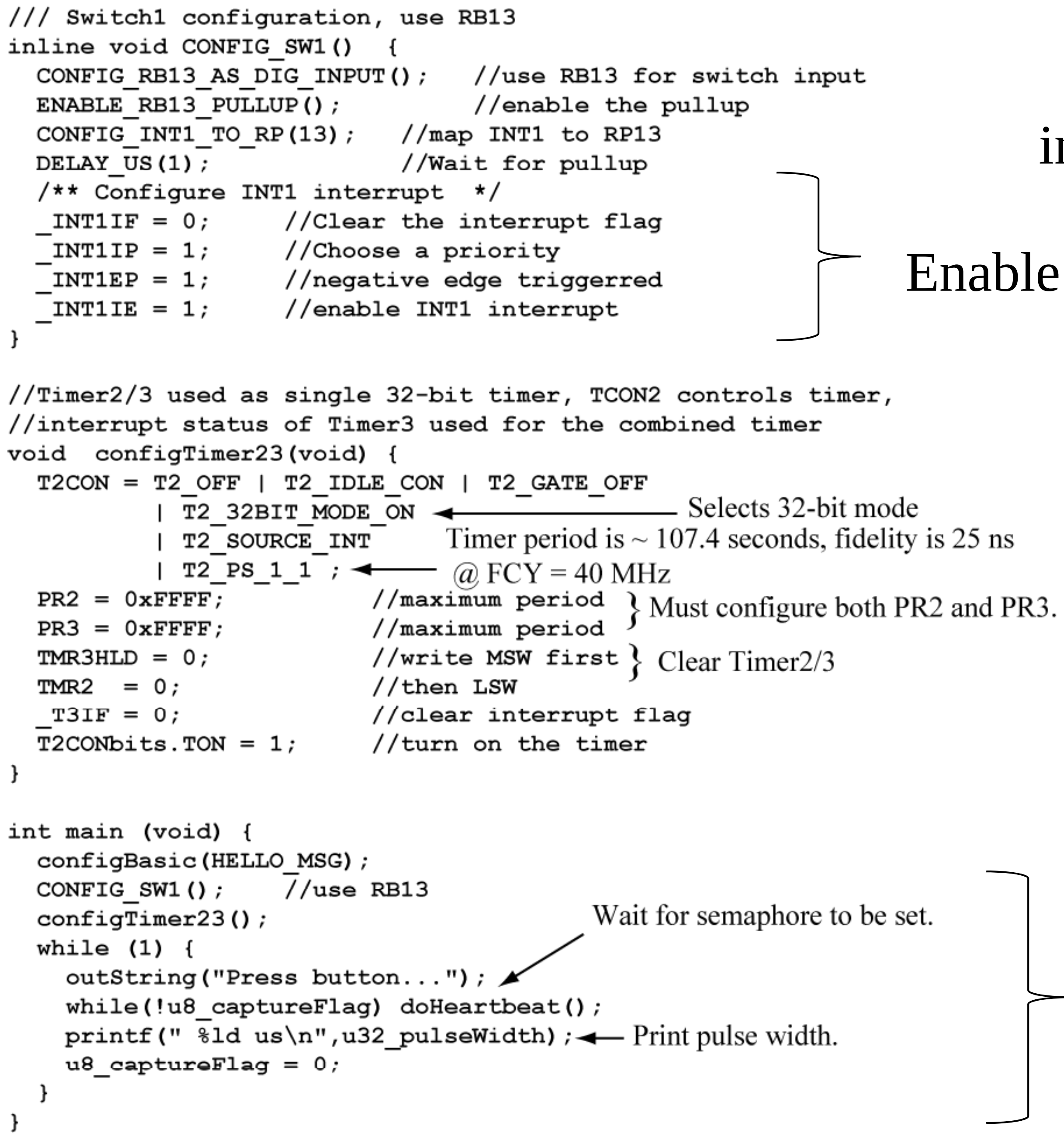
Compute delta, assumes PR3, PR2 both 0xFFFF.

Convert to µs, set semaphore, configure for rising edge.

Use INT1 to detect falling, rising edges to measure pulse width.

Compute pulse width inside the ISR, set a semaphore.

32-bit PW measure, interrupt driven (cont).



```
/// Switch1 configuration, use RB13
inline void CONFIG_SW1()  {
  CONFIG_RB13_AS_DIG_INPUT();   //use RB13 for switch input
  ENABLE_RB13_PULLUP();         //enable the pullup
  CONFIG_INT1_TO_RP(13);   //map INT1 to RP13
  DELAY_US(1);             //Wait for pullup
  /** Configure INT1 interrupt  */
  _INT1IF = 0;     //Clear the interrupt flag
  _INT1IP = 1;     //Choose a priority
  _INT1EP = 1;     //negative edge triggerred
  _INT1IE = 1;     //enable INT1 interrupt
}
```

Enable INT1

```
//Timer2/3 used as single 32-bit timer, TCON2 controls timer,
//interrupt status of Timer3 used for the combined timer
void  configTimer23(void) {
  T2CON = T2_OFF | T2_IDLE_CON | T2_GATE_OFF
          | T2_32BIT_MODE_ON
          | T2_SOURCE_INT
          | T2_PS_1_1 ;
  PR2 = 0xFFFF;           //maximum period
  PR3 = 0xFFFF;           //maximum period
  TMR3HLD = 0;            //write MSW first
  TMR2  = 0;              //then LSW
  _T3IF = 0;              //clear interrupt flag
  T2CONbits.TON = 1;      //turn on the timer
}
```

Selects 32-bit mode

Timer period is ~ 107.4 seconds, fidelity is 25 ns @ FCY = 40 MHz

Must configure both PR2 and PR3.

Clear Timer2/3

Configure 32-bit timer

```
int main (void) {
  configBasic(HELLO_MSG);
  CONFIG_SW1();     //use RB13
  configTimer23();
  while (1) {
    outString("Press button...");
    while(!u8_captureFlag) doHeartbeat();
    printf(" %ld us\n",u32_pulseWidth);
    u8_captureFlag = 0;
  }
}
```

Wait for semaphore to be set.

Print pulse width.

Wait for semaphore, set the flag.

Problems with 32-bit approach

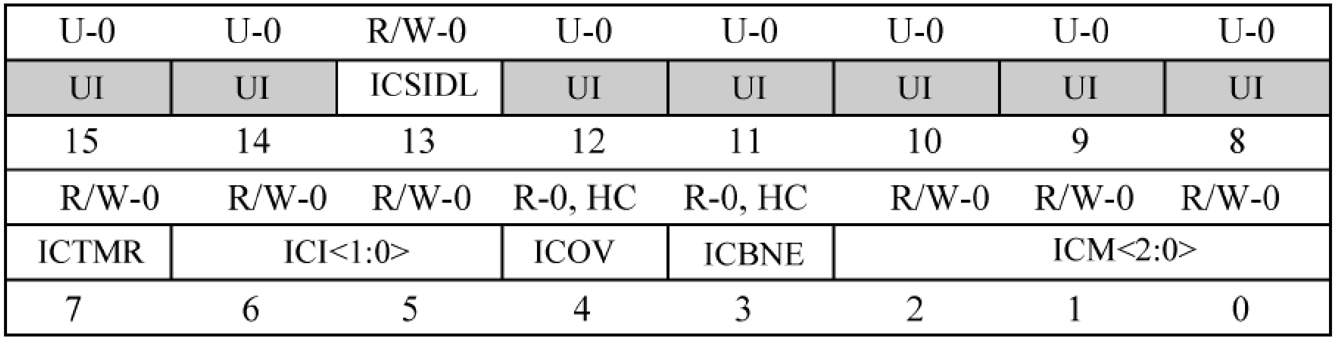
Hooray, can measure pulses that are 107 seconds long, with a timer fidelity of 25 ns @ FCY = 40 MHz!!!!

Overkill – wasteful of Timer resources to use two timers, should be better way where we do not have to use both timers.

Not really this accurate – INT0 ISR uncertain – if higher priority interrupt occurring we will read the timer value late, also uncertain as to where in the instruction cycle the interrupt is recognized.

Also, if want accuracy, use a Crystal. External crystal accuracy is approx. ±20 ppm (parts per million), i.e, 20 μs in 1 second.
For a 100 ms (100,000 μs) push button pulse, is ±2 μs .
If using internal oscillator, this varies by ± 2%!!!

Input Capture modes



U-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
UI	UI	ICSIDL	UI	UI	UI	UI	UI
15	14	13	12	11	10	9	8
R/W-0	R/W-0	R/W-0	R-0, HC	R-0, HC	R/W-0	R/W-0	R/W-0
ICTMR	ICI<1:0>		ICOV	ICBNE	ICM<2:0>		
7	6	5	4	3	2	1	0

Figure redrawn by author from Reg. 12-1 found in the PIC24HJ32GP202 datasheet (DS70289A), Microchip Technology Inc.

Bit 13 : ICSIDL: Input Capture Module Stop in Idle Control Bit
1 = Input Capture will halt in CPU Idle mode
0 = Input Capture will continue to operate in CPU Idle Mode
Bit 7: ICTMR: Input Capture Timer Select Bits
1 = TMR2 contents are captured on capture event
0 = TMR3 contents are captured on capture event
Bit 6-5: ICI<1:0>: Select Number of Captures per Interrupt bits
11 = Interrupt on every fourth capture event
10 = Interrupt on every third capture event
01 = Interrupt on every second capture event
00 = Interrupt on every capture event

← Note: Disable module (ICM<2:0> = 000) before changing the ICI bits.

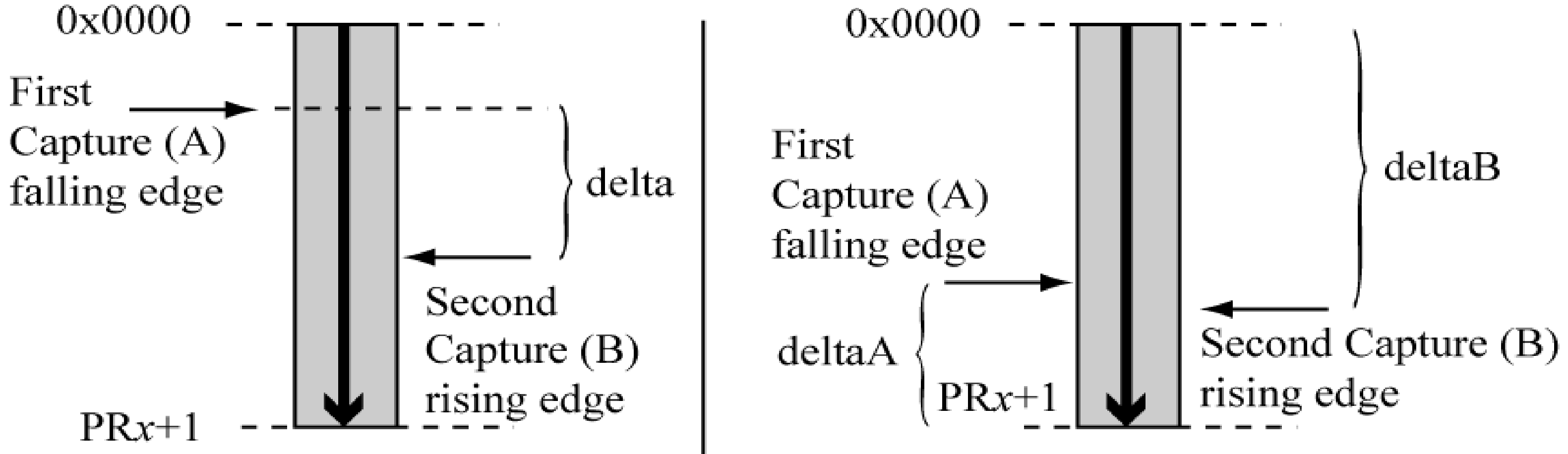
Interrupt selection

Bit 4: ICOV: Input Capture Overflow Status Flag bit (read-only)
1 = Input capture overflow occurred; 0 = No input capture overflow occurred
Bit 3: ICBNE: Input Capture Buffer Empty Status bit (read-only)
1 = Input capture buffer is not empty; 0 = Input capture buffer is empty
Bit 2-0: ICM<2:0>: Input Capture Mode Select Bits
111 = Input capture functions as interrupt pin only when device is in Sleep or Idle mode (Rising edge detect only, all other control bits are not applicable.)
110 = Unused (module disabled)
101 = Capture mode, every 16th rising edge
100 = Capture mode, every 4th rising edge
011 = Capture mode, every rising edge
010 = Capture mode, every falling edge
001 = Capture mode, every edge (rising and falling) (ICI<1:0> bits do not control interrupt generation for this mode)
000 = Input capture module turned off

} Edge selection

Input Capture Approach

Use only one 16-bit timer.



(a) No overflow case
TDelta = B – A

(b) Overflow case
TDelta = (#oflows – 1)* ticks_per_oflow + deltaA + deltaB
= (#oflows – 1) *(PRx+1) + (PRx+1 – A) + B

Track number of Timer overflows using Timer2 ISR and include this in the computation to compute long pulse widths.

Delta Time Code

```
uint32 computeDeltaTicksLong(uint16 u16_start, uint16 u16_end,
uint16 u16_tmrPR, uint16 u16_oflows) {
  uint32 u32_deltaTicks;
  if (u16_oflows == 0) u32_deltaTicks = u16_end - u16_start;
  else {
    //compute ticks from start to timer overflow
    u32_deltaTicks = (u16_tmrPR + 1) - u16_start;
    //add ticks due to
    //overflows =  (overflows -1) * ticks_per_overflow
    u32_deltaTicks += ((((uint32) u16_oflows)- 1) *
(((uint32)u16_tmrPR) + 1)) ;
    //now add in the delta due to the last capture
    u32_deltaTicks += u16_end;
  }
  return (u32_deltaTicks);
}
```

Input Capture Code, IC1Interrupt ISR

```
typedef enum  { STATE_WAIT_FOR_FALL_EDGE = 0, STATE_WAIT_FOR_RISE_EDGE,
} ICSTATE;

volatile uint16 u16_oflowCount = 0;
void _ISRFAST _T2Interrupt (void) {
  u16_oflowCount++;  //count number of TMR2 overflows
  _T2IF = 0;         //clear the timer interrupt bit
}

ICSTATE e_isrICState = STATE_WAIT_FOR_FALL_EDGE;
volatile uint8 u8_captureFlag = 0;
volatile uint16 u16_lastCapture, volatile uint16 u16_thisCapture;
volatile uint32 u32_pulseWidth;

void _ISRFAST _IC1Interrupt() {
 _IC1IF = 0;
 u16_thisCapture = IC1BUF;
 switch (e_isrICState) {
  case STATE_WAIT_FOR_FALL_EDGE:
   if (u16_thisCapture == 0 && _T2IF)
          u16_oflowCount = 0 - 1; //simultaneous timer with capture
   else u16_oflowCount = 0;
    u16_lastCapture = u16_thisCapture;
    e_isrICState = STATE_WAIT_FOR_RISE_EDGE;
   }
   break;
  case STATE_WAIT_FOR_RISE_EDGE:
   if (u16_thisCapture == 0 && _T2IF) u16_oflowCount++; //simult. interpt
   u32_pulseWidth = computeDeltaTicksLong(u16_lastCapture,
                                     u16_thisCapture,
                                     PR2, u16_oflowCount);
   u32_pulseWidth = ticksToUs(u32_pulseWidth,
                        getTimerPrescale(T2CONbits));
   u8_captureFlag = 1;
   e_isrICState = STATE_WAIT_FOR_FALL_EDGE;
   break;
  default: e_isrICState = STATE_WAIT_FOR_FALL_EDGE;
 }
}
```

Track number of TMR2 overflows so that we can measure long pulse widths.

Measurement complete semaphore

Configured for interrupt on every edge, has higher priority than TMR2 interrupt.

Always read capture buffer to prevent overflow.

Simultaneous IC1 with TMR2, so init `oflowCount` as –1 so that ISR makes it 0.

Clear overflow count.

Save capture value.

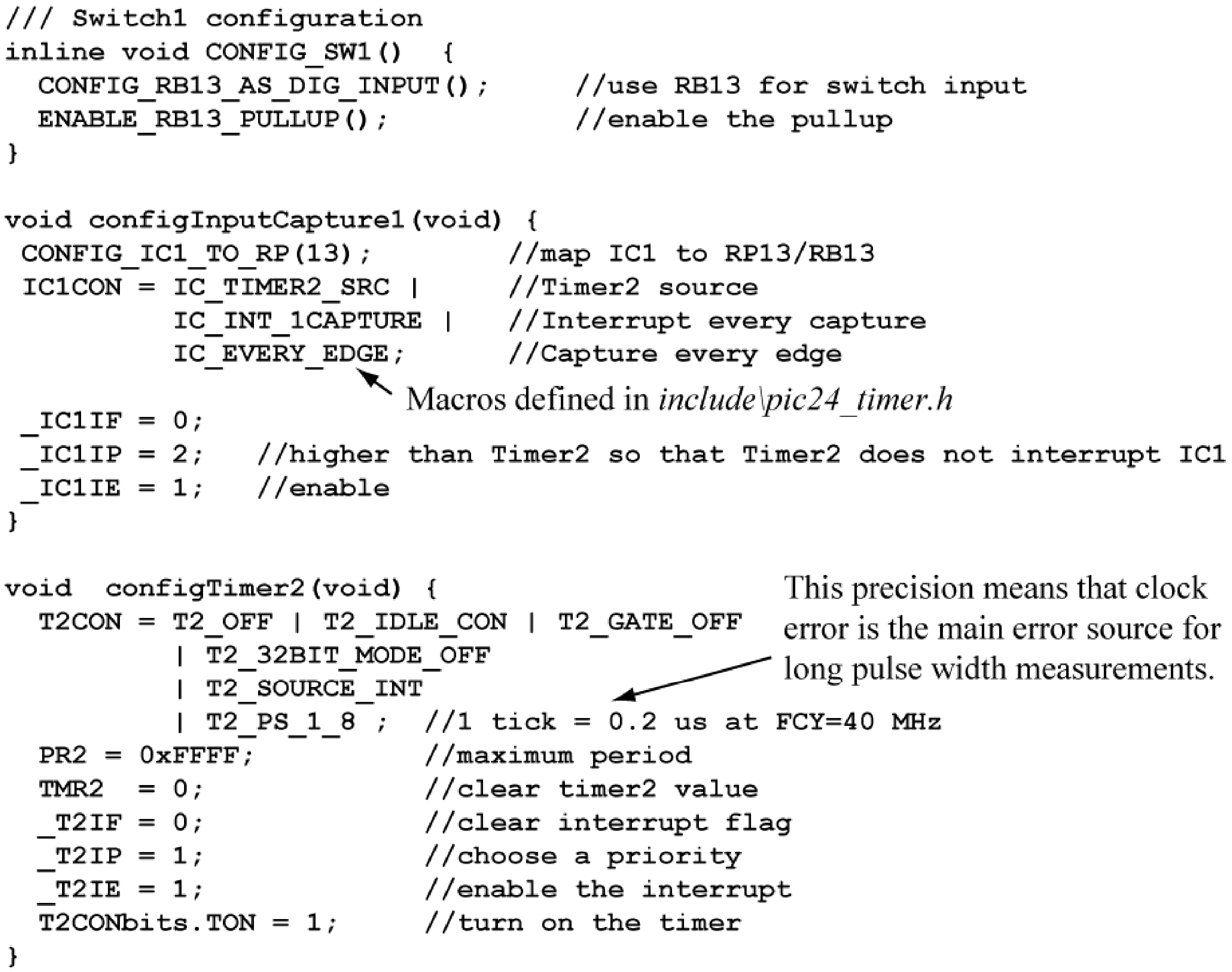
Next edge

Simultaneous IC1 with TMR2, so increment `oflowCount` here.

Compute delta ticks.

Convert to µs.

Input Capture Mode Code (config)



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

void configInputCapture1(void) {
 CONFIG_IC1_TO_RP(13);        //map IC1 to RP13/RB13
 IC1CON = IC_TIMER2_SRC |     //Timer2 source
          IC_INT_1CAPTURE |   //Interrupt every capture
          IC_EVERY_EDGE;      //Capture every edge

 _IC1IF = 0;
 _IC1IP = 2;   //higher than Timer2 so that Timer2 does not interrupt IC1
 _IC1IE = 1;   //enable
}

void  configTimer2(void) {
  T2CON = T2_OFF | T2_IDLE_CON | T2_GATE_OFF
          | T2_32BIT_MODE_OFF
          | T2_SOURCE_INT
          | T2_PS_1_8 ;  //1 tick = 0.2 us at FCY=40 MHz
  PR2 = 0xFFFF;          //maximum period
  TMR2  = 0;             //clear timer2 value
  _T2IF = 0;             //clear interrupt flag
  _T2IP = 1;             //choose a priority
  _T2IE = 1;             //enable the interrupt
  T2CONbits.TON = 1;     //turn on the timer
}
```

Macros defined in *include\pic24_timer.h*

This precision means that clock error is the main error source for long pulse width measurements.

Reducing Error for Period Measurement

The previous time measurement examples measured the pulse width (falling edge to rising edge) of a single pulse.

If you have a repeating square wave of a fixed frequency, and want to measure period, then measure either rising-to-rising or falling-to-falling edges.

For more accuracy, use the input capture modes that captures these edges either every 4 edges or every 16 edges (this reduces error by a factor of 4 and factor of 16, respectively). See book example (chap10/incap_freqmeasure.c).

Input Capture modes

U-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
UI	UI	ICSIDL	UI	UI	UI	UI	UI
15	14	13	12	11	10	9	8
R/W-0	R/W-0	R/W-0	R-0, HC	R-0, HC	R/W-0	R/W-0	R/W-0
ICTMR	ICI<1:0>		ICOV	ICBNE	ICM<2:0>		
7	6	5	4	3	2	1	0

Figure redrawn by author from Reg. 12-1 found in the PIC24HJ32GP202 datasheet (DS70289A), Microchip Technology Inc.

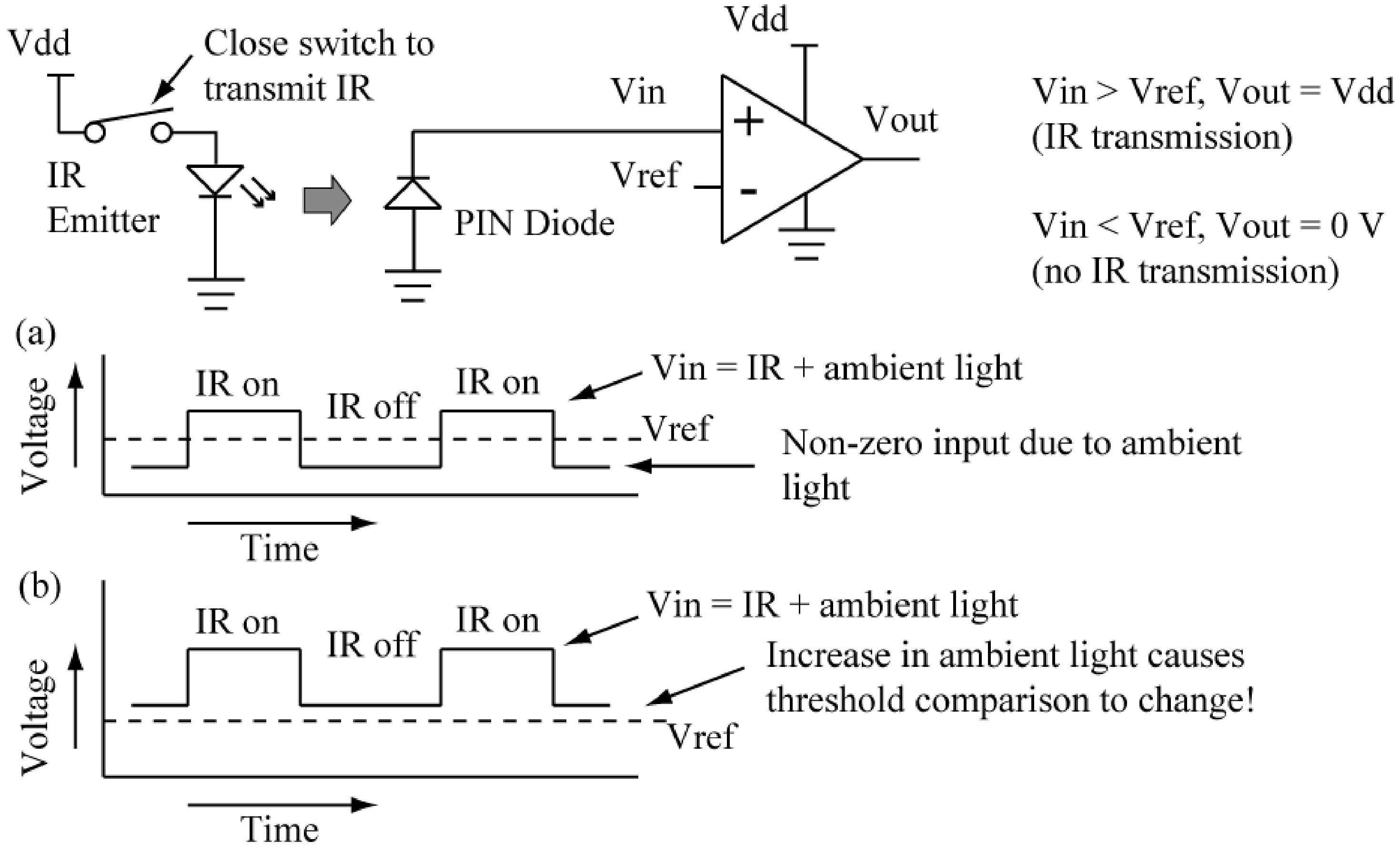
Bit 13 : ICSIDL: Input Capture Module Stop in Idle Control Bit
1 = Input Capture will halt in CPU Idle mode
0 = Input Capture will continue to operate in CPU Idle Mode
Bit 7: ICTMR: Input Capture Timer Select Bits
1 = TMR2 contents are captured on capture event
0 = TMR3 contents are captured on capture event
Bit 6-5: ICI<1:0>: Select Number of Captures per Interrupt bits
11 = Interrupt on every fourth capture event
10 = Interrupt on every third capture event
01 = Interrupt on every second capture event
00 = Interrupt on every capture event

Note: Disable module (ICM<2:0> = 000) before changing the ICI bits.

Bit 4: ICOV: Input Capture Overflow Status Flag bit (read-only)
1 = Input capture overflow occurred; 0 = No input capture overflow occurred
Bit 3: ICBNE: Input Capture Buffer Empty Status bit (read-only)
1 = Input capture buffer is not empty; 0 = Input capture buffer is empty
Bit 2-0: ICM<2:0>: Input Capture Mode Select Bits
111 = Input capture functions as interrupt pin only when device is in Sleep or Idle mode
(Rising edge detect only, all other control bits are not applicable.)
110 = Unused (module disabled)
101 = Capture mode, every 16th rising edge
100 = Capture mode, every 4th rising edge
011 = Capture mode, every rising edge
010 = Capture mode, every falling edge
001 = Capture mode, every edge
(rising and falling) (ICI<1:0> bits do not control interrupt generation for this mode)
000 = Input capture module turned off

These modes useful for measuring period of square waves

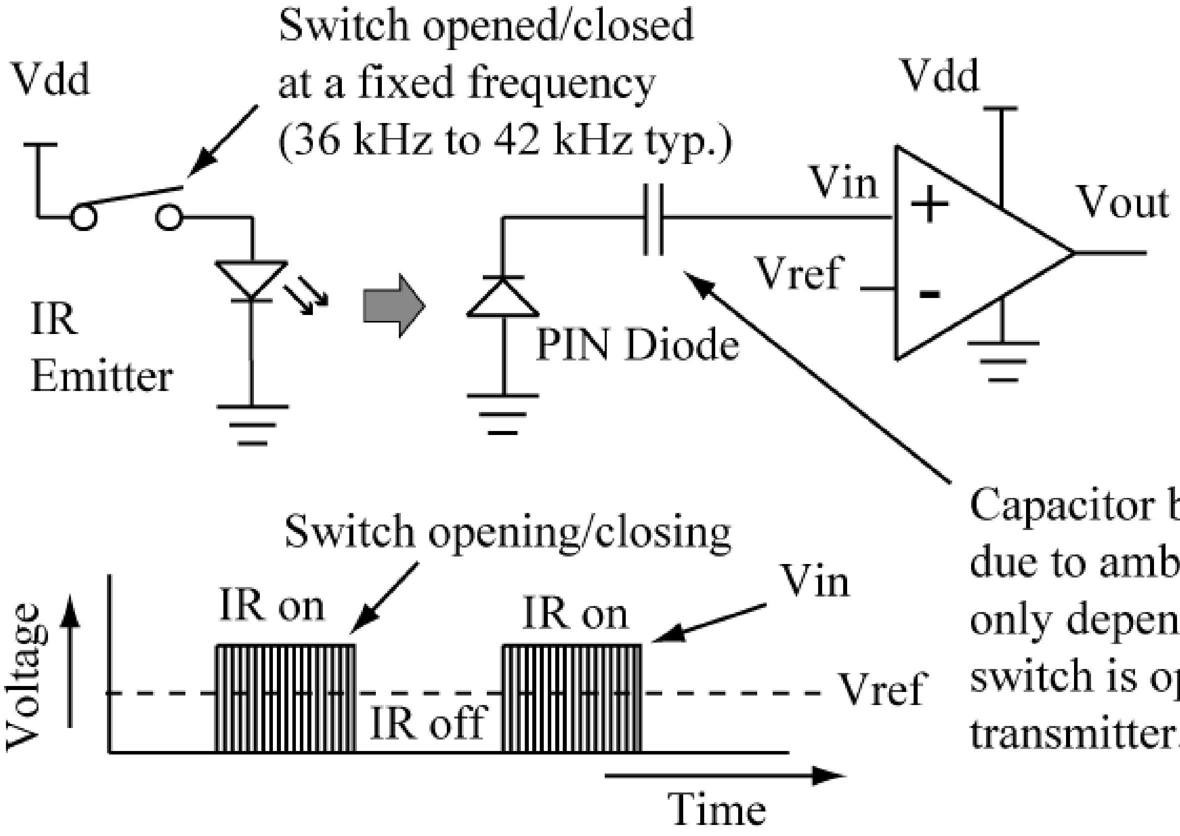
PW measure application: IR Decoding



IR transmission is common wireless control method.

IR transmission is not as simple as turning on or off an IR source

IR Modulation to remove Ambient Light

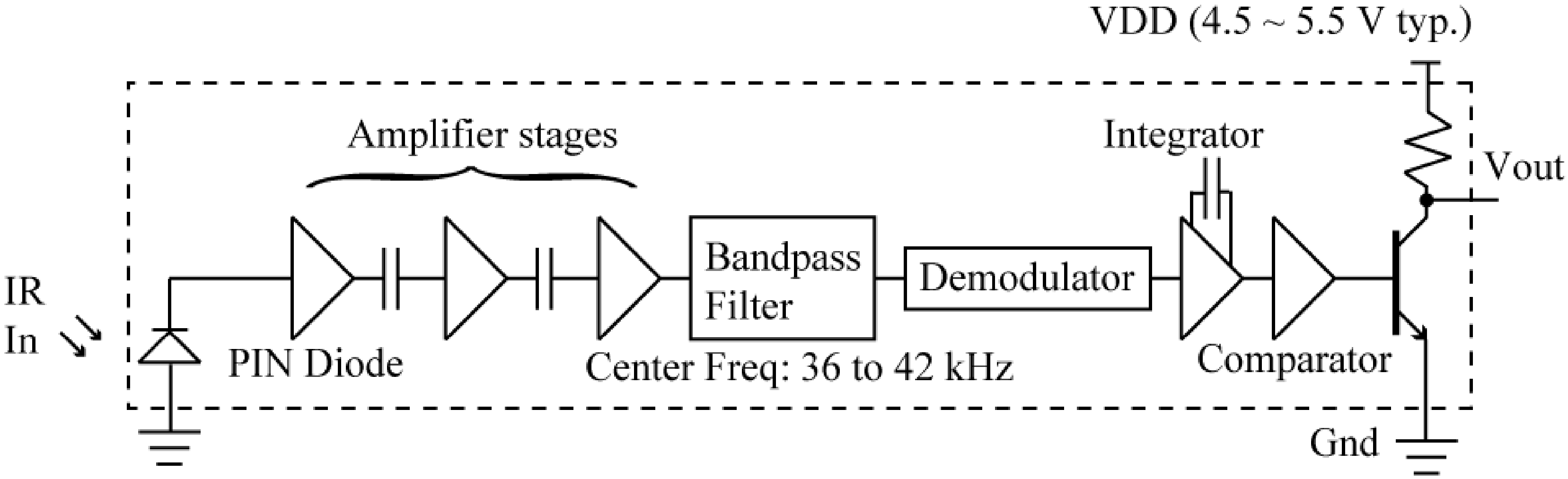


Vin > Vref, Vout = Vdd
(IR present)

Vin < Vref, Vout = 0 V
(IR absent)

Capacitor blocks DC voltage due to ambient light. Vin voltage only depends on how fast the switch is open/closed at the transmitter.

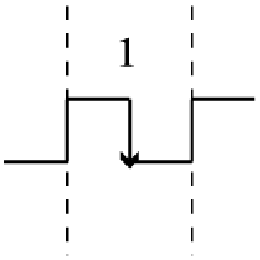
Integrated IR Receiver



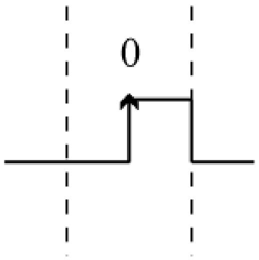
Can be use to receive signals
from a universal remote control

Space Width Serial Encoding

(a) Biphase encoding, 1 and 0 are distinguished by a transition in the center of the bit period.



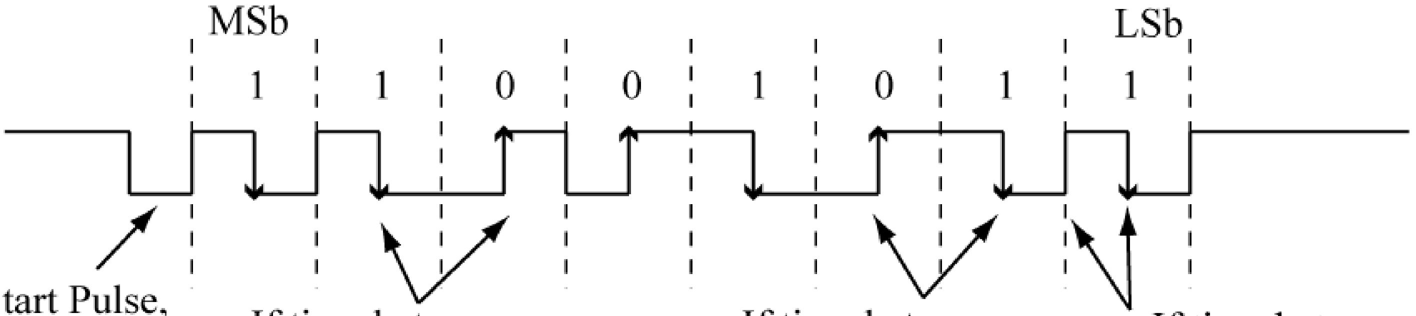
High to Low
transition in middle
of bit period is a 1.



Low to High
transition in middle
of bit period is a 0.

Bit period varies by manufacturer, Philips uses ~1800 µs.

(b) Biphase encoding example, value is 0xCB. Can send multiple bytes in one transmission.



Start Pulse,
1/2 bit period

If time between
edges is 1 period,
the bit value has
changed!

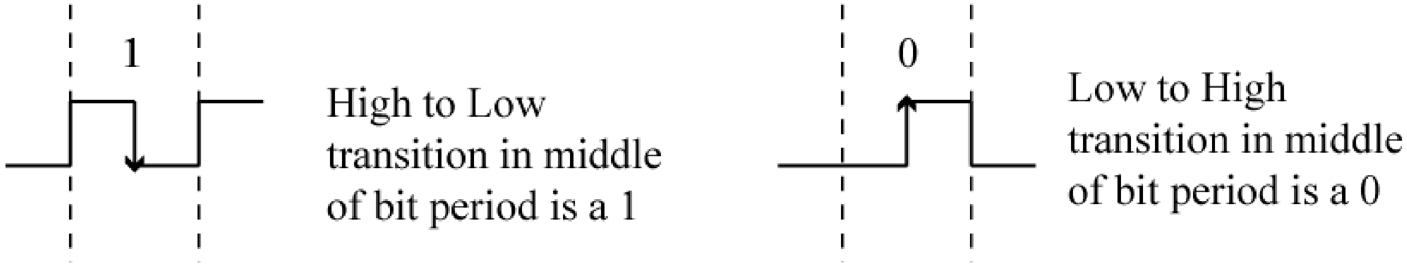
If time between
edges is 1 period,
the bit value has
changed!

If time between
edges is 1/2 period,
the bit value remains
the same.

Must detect each edge transition to decode waveform.

Biphase Serial Encoding

(a) Biphase encoding, 1 and 0 distinguished by transition in middle of bit period

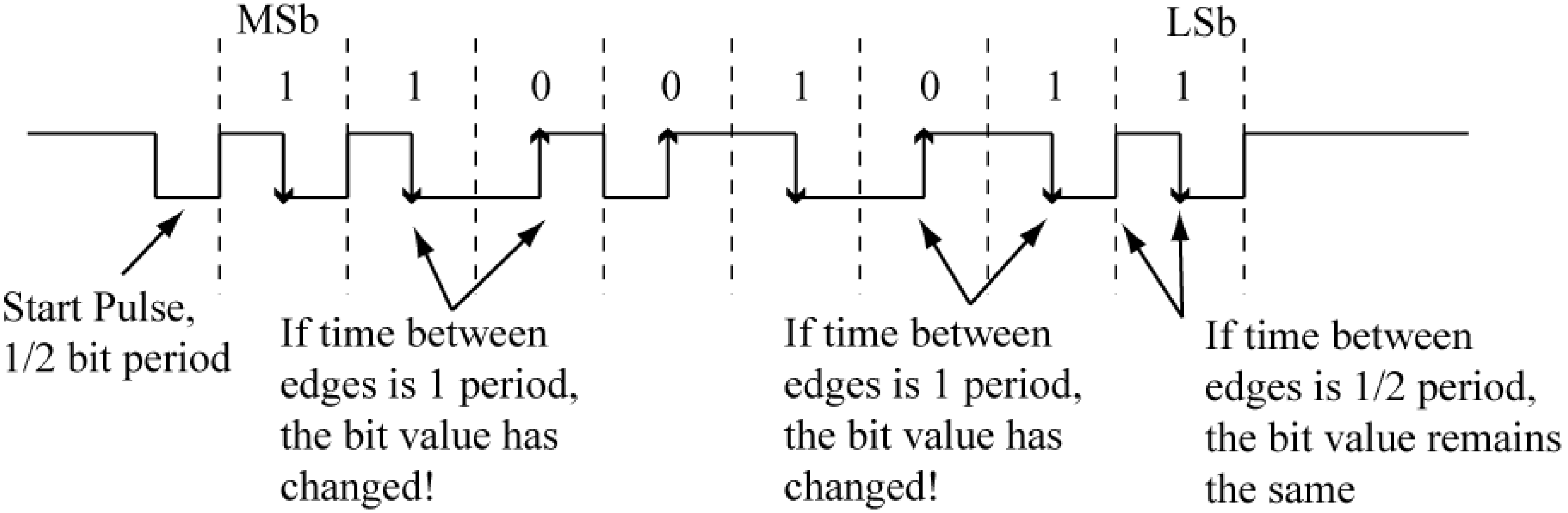


High to Low transition in middle of bit period is a 1

Low to High transition in middle of bit period is a 0

Bit period varies by manufacturer, Philips uses ~1800 µs.

(b) Biphase encoding example, value is 0xCB. Can send multiple bytes in one transmission.



Start Pulse, 1/2 bit period

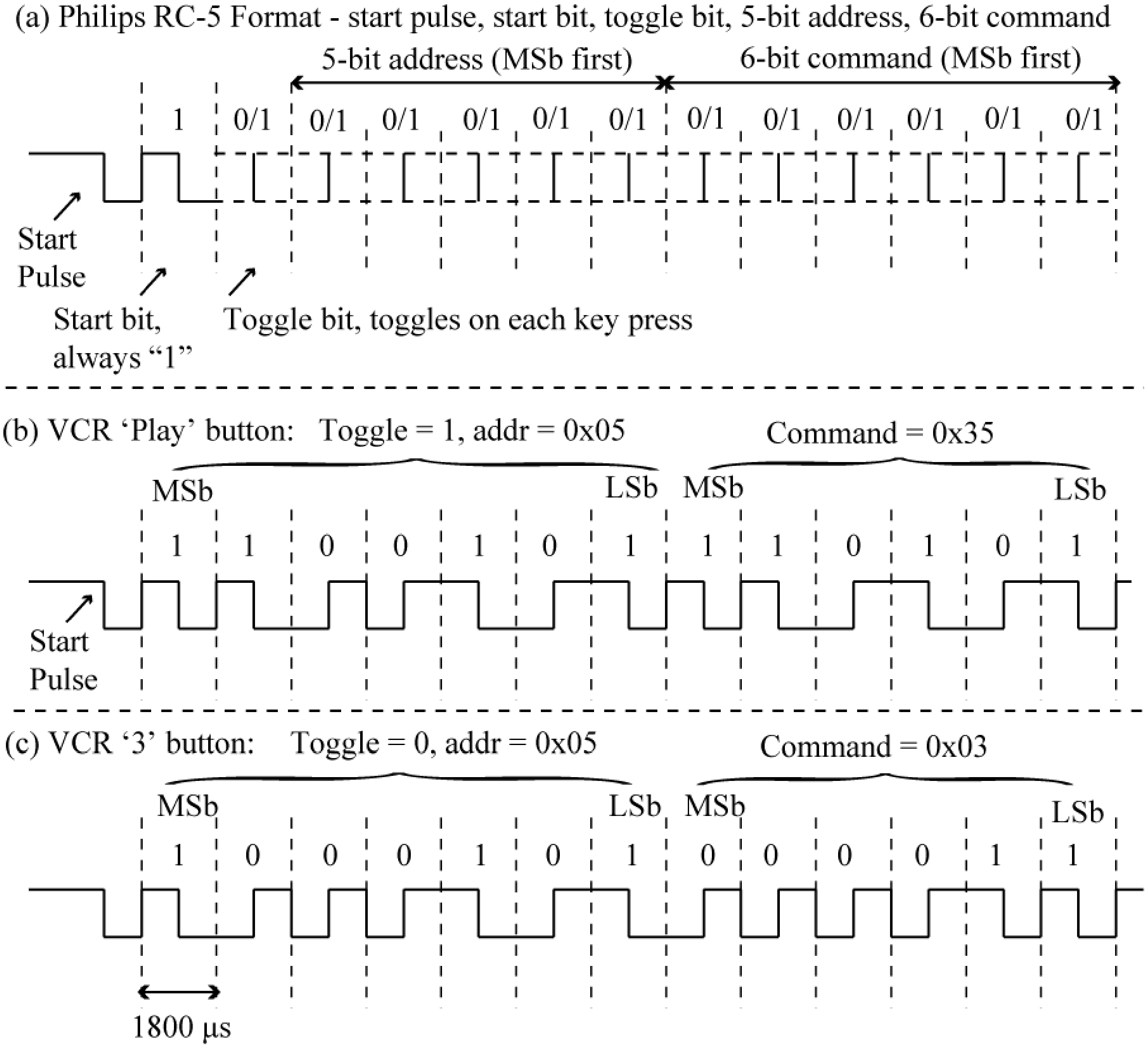
If time between edges is 1 period, the bit value has changed!

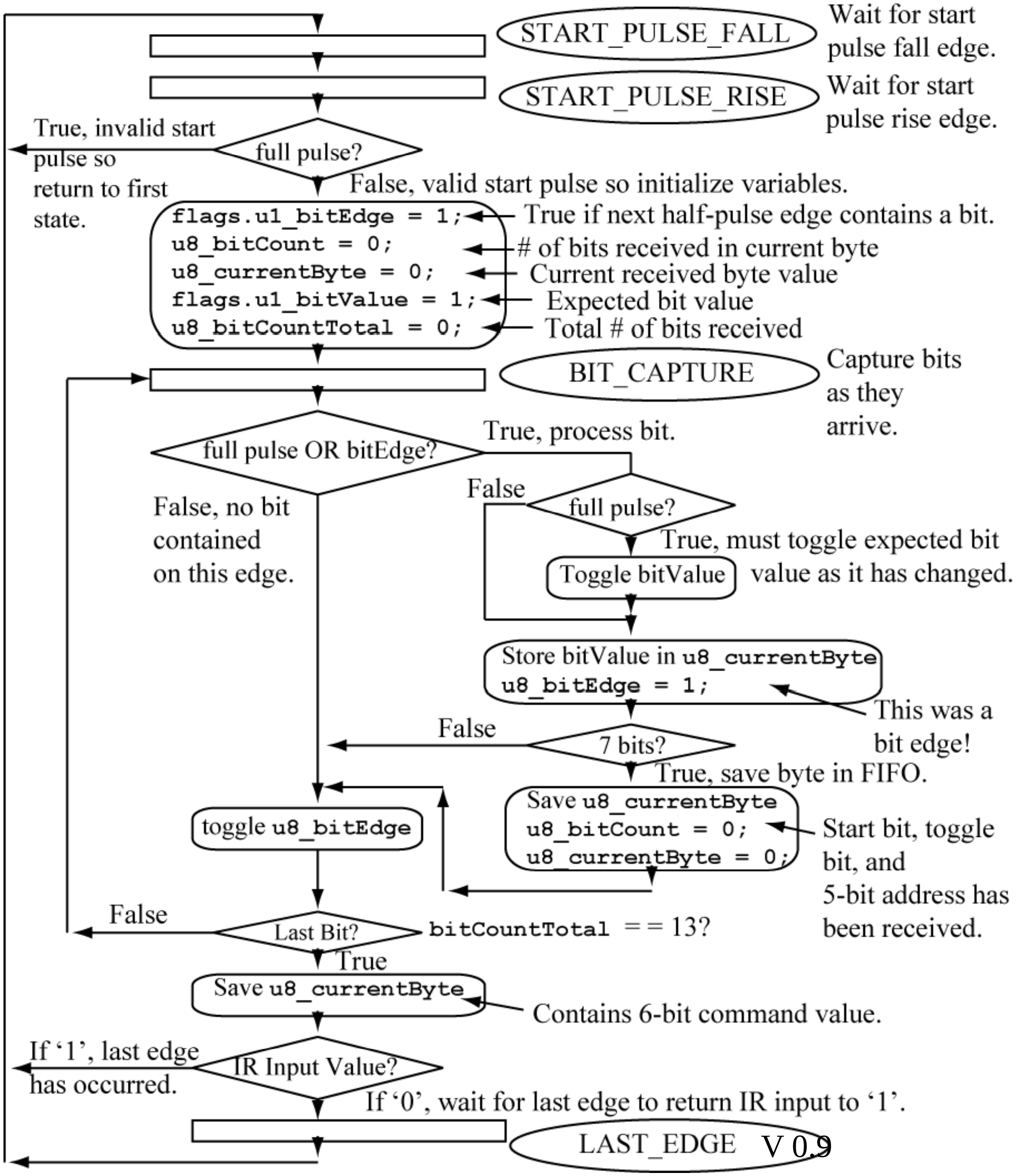
If time between edges is 1 period, the bit value has changed!

If time between edges is 1/2 period, the bit value remains the same

Must detect each edge transition to decode waveform

Phillips VCR RC-5 Code



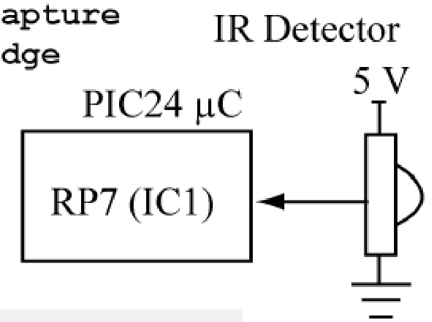


IC1 ISR Flowchart for Decoding

Book has full code.

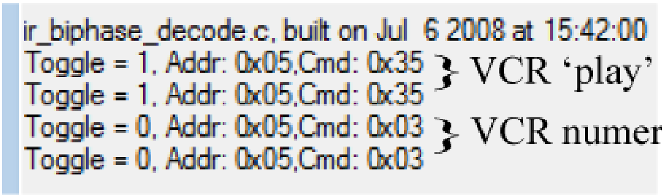
Use IC1 pulse width measurement determine if a half-pulse or full-pulse has been received, decode bits, place bytes in software FIFO.

```
//configure input capture.
void configInputCapture1(void) {
 CONFIG_RB7_AS_DIG_INPUT();   //use RB7 for IR Input
 CONFIG_IC1_TO_RP(7);         //map IC1 to RP7/R7
 e_isrICState = STATE_START_PULSE_FALL;
 u16_irFifoHead = 0;
 u16_irFifoTail = 0;
 u16_twoThirdsPeriodTicks = usToU16Ticks(TWOTHIRDS_PERIOD_US,
                                        getTimerPrescale(T2CONbits));
 IC1CON = IC_TIMER2_SRC |     //Timer2 source
          IC_INT_1CAPTURE |   //Interrupt every capture
          IC_EVERY_EDGE;      //Interrupt every edge
 _IC1IF = 0;
 _IC1IP = 1;
 _IC1IE = 1;   //enable IC1 interrupt
}
```



```
int main (void) {
  uint8 u8_x, u8_y;
  configBasic(HELLO_MSG);
  configTimer2();
  configInputCapture1();
  while (1) {
    u8_x = irFifoRead(); //read addr
    u8_y = irFifoRead(); //read cmd
    if (u8_x & 0x20) outString("Toggle = 1, ");
    else outString("Toggle = 0, ");
    outString("Addr: "); outUint8(u8_x & 0x1F);
    outString(",Cmd: "); outUint8(u8_y);
    outString("\n");
  }
}
```

Sample Output



```
ir_biphase_decode.c, built on Jul  6 2008 at 15:42:00
Toggle = 1, Addr: 0x05,Cmd: 0x35
Toggle = 1, Addr: 0x05,Cmd: 0x35
Toggle = 0, Addr: 0x05,Cmd: 0x03
Toggle = 0, Addr: 0x05,Cmd: 0x03
```

} VCR ‘play’ button

} VCR numeral ‘3’ button

Code Output

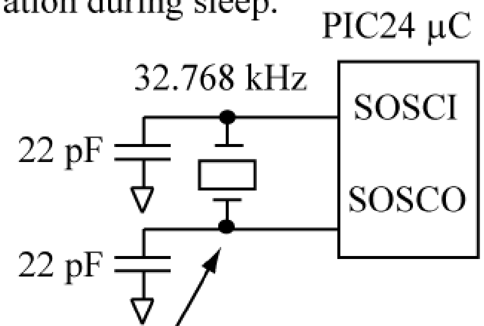
Real-Time Timekeeping

```
volatile uint16 u16_seconds = 0;

void _ISRFAST _T1Interrupt (void) {
  u16_seconds++;
  _T1IF = 0;  //clear interrupt flag
}
`
```

Occurs once a second, increment the `u16_seconds` variable.

Config Timer1 to use the external clock, prescale of 1, continue operation during sleep.

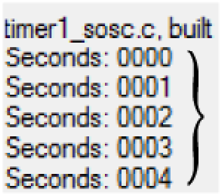
```
void  configTimer1(void) {
  T1CON = T1_OFF | T1_IDLE_CON | T1_GATE_OFF
          | T1_SYNC_EXT_OFF
          | T1_SOURCE_EXT .//ext clock
          | T1_PS_1_1  ;  // prescaler of 1
  PR1 = 0x7FFF;           //period is 1 second
  _T1IF = 0;              //clear interrupt flag
  _T1IP = 1;              //choose a priority
  _T1IE = 1;              //enable the interrupt
  T1CONbits.TON = 1;      //turn on the timer
}
```



Sets the OSCCON.SOSCEN bit which enables the secondary oscillator amplifier; without this, the crystal will not oscillate.

```
int main(void) {
   __builtin_write_OSCCONL(OSCCON | 0x02);    //OSCCON.SOSCEN=1;
  configBasic(HELLO_MSG);   //say Hello!
  configTimer1();
  while (1) {
    outString("Seconds: ");
    outUint16Decimal(u16_seconds);
    outString("\n");
    while(!IS_TRANSMIT_COMPLETE_UART1());
    SLEEP();
  }
}
```

Timer1 interrupt can wake from sleep since it has been configured to keep operating during sleep mode.



```
timer1_sosc.c, built
Seconds: 0000
Seconds: 0001
Seconds: 0002
Seconds: 0003
Seconds: 0004
```

Example output

Timer1 is special, can use the secondary oscillator.

A 32.768 kHz watch crystal and a PR1 = 0x7FFF means timer period is 0x8000 ticks, or 32768 ticks, or 1 second for a 32.768 kHz crystal!

Use ISR to increment a 'seconds' variable!

Real-Time Clock/Calendar Module

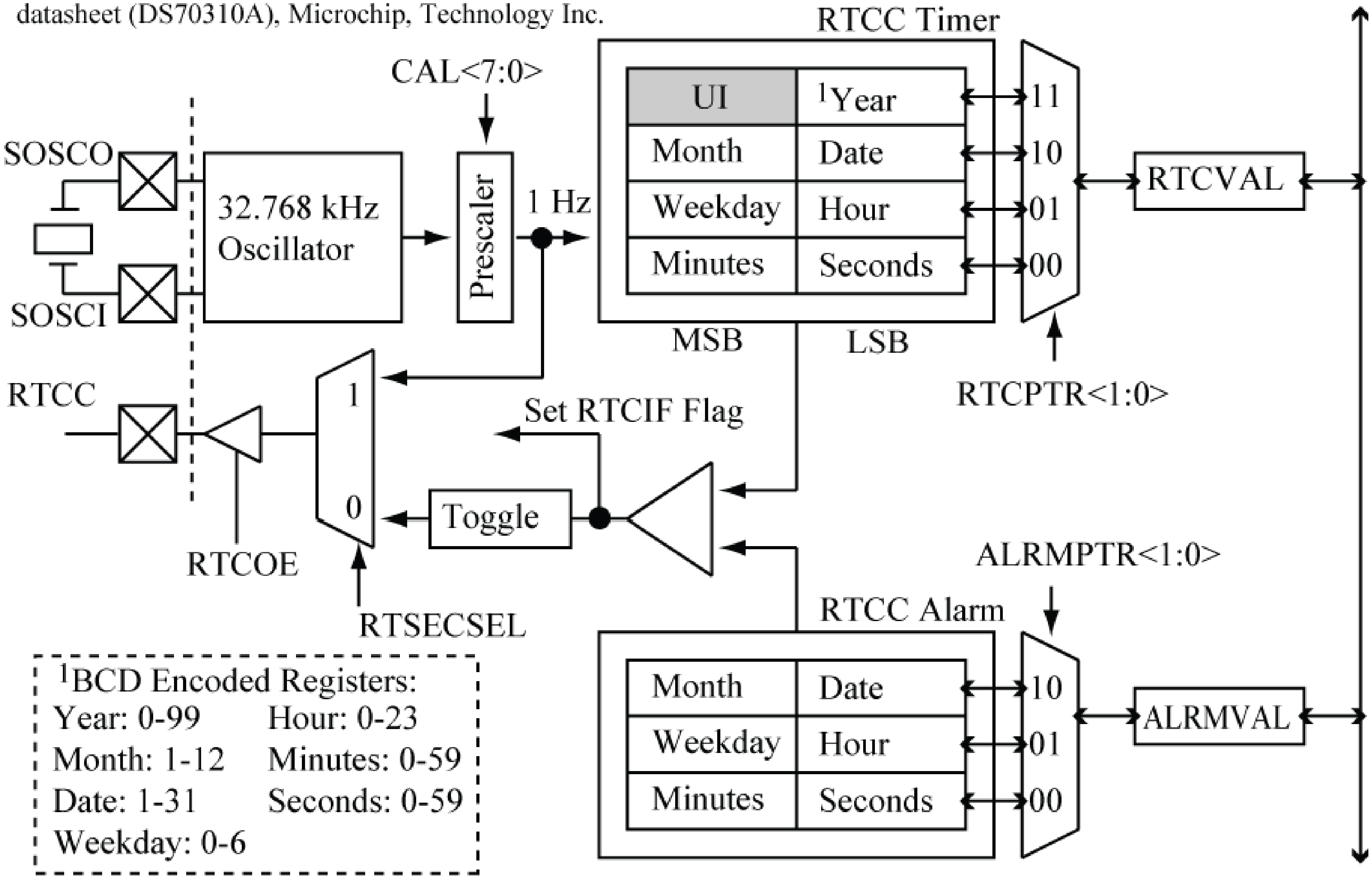


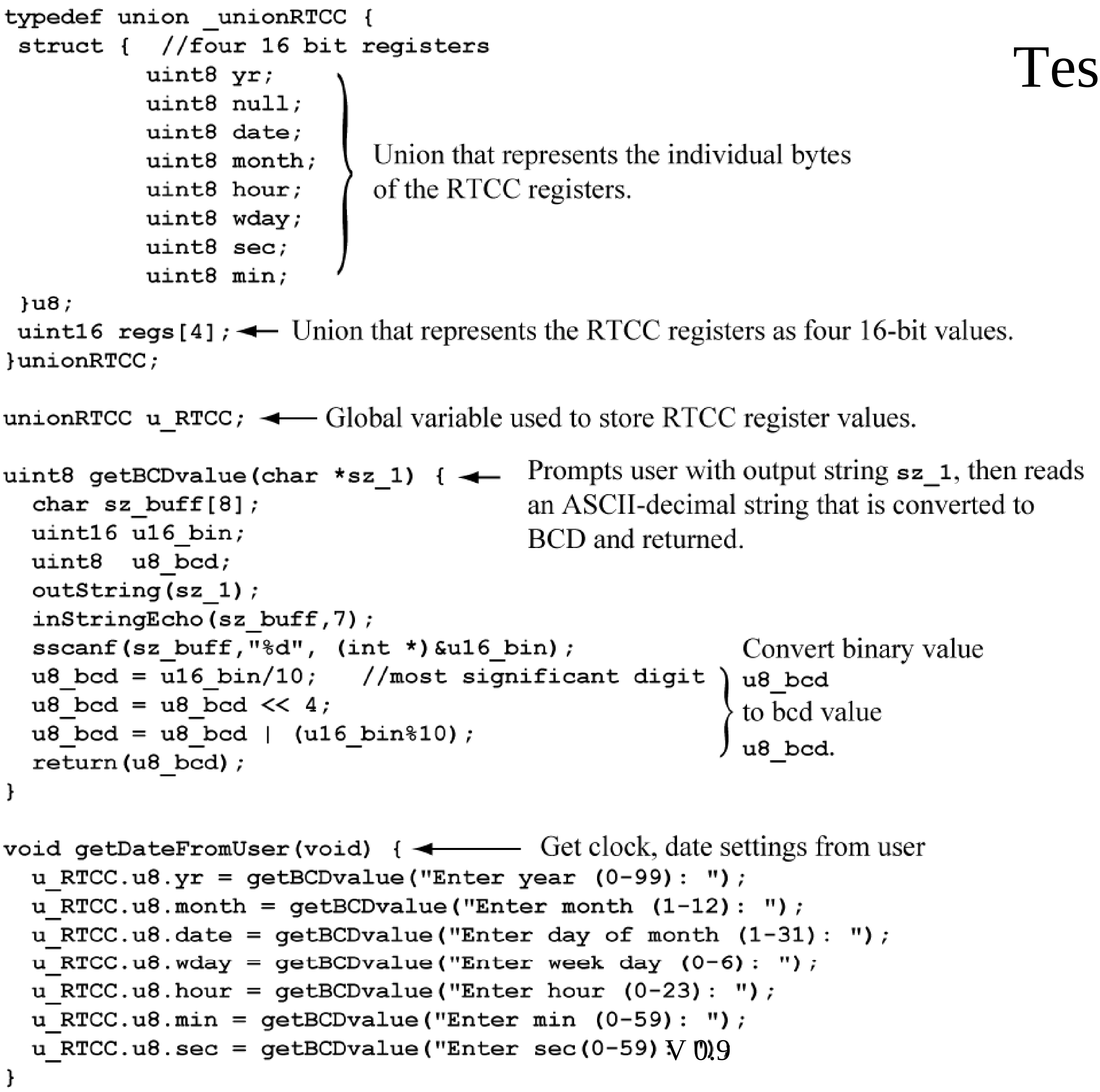
Figure redrawn by author from Fig 37-1 found in the PIC24 FRM datasheet (DS70310A), Microchip, Technology Inc.

[1]BCD Encoded Registers:

Year: 0-99	Hour: 0-23
Month: 1-12	Minutes: 0-59
Date: 1-31	Seconds: 0-59
Weekday: 0-6	

Found on some PIC24 µCs; see chap10/rtcc.c

Test Code



```
typedef union _unionRTCC {
 struct {  //four 16 bit registers
          uint8 yr;
          uint8 null;
          uint8 date;
          uint8 month;
          uint8 hour;
          uint8 wday;
          uint8 sec;
          uint8 min;
 }u8;
 uint16 regs[4];
}unionRTCC;
```

Union that represents the individual bytes of the RTCC registers.

Union that represents the RTCC registers as four 16-bit values.

```
unionRTCC u_RTCC;
```

Global variable used to store RTCC register values.

```
uint8 getBCDvalue(char *sz_1) {
  char sz_buff[8];
  uint16 u16_bin;
  uint8  u8_bcd;
  outString(sz_1);
  inStringEcho(sz_buff,7);
  sscanf(sz_buff,"%d", (int *)&u16_bin);
  u8_bcd = u16_bin/10;   //most significant digit
  u8_bcd = u8_bcd << 4;
  u8_bcd = u8_bcd | (u16_bin%10);
  return(u8_bcd);
}
```

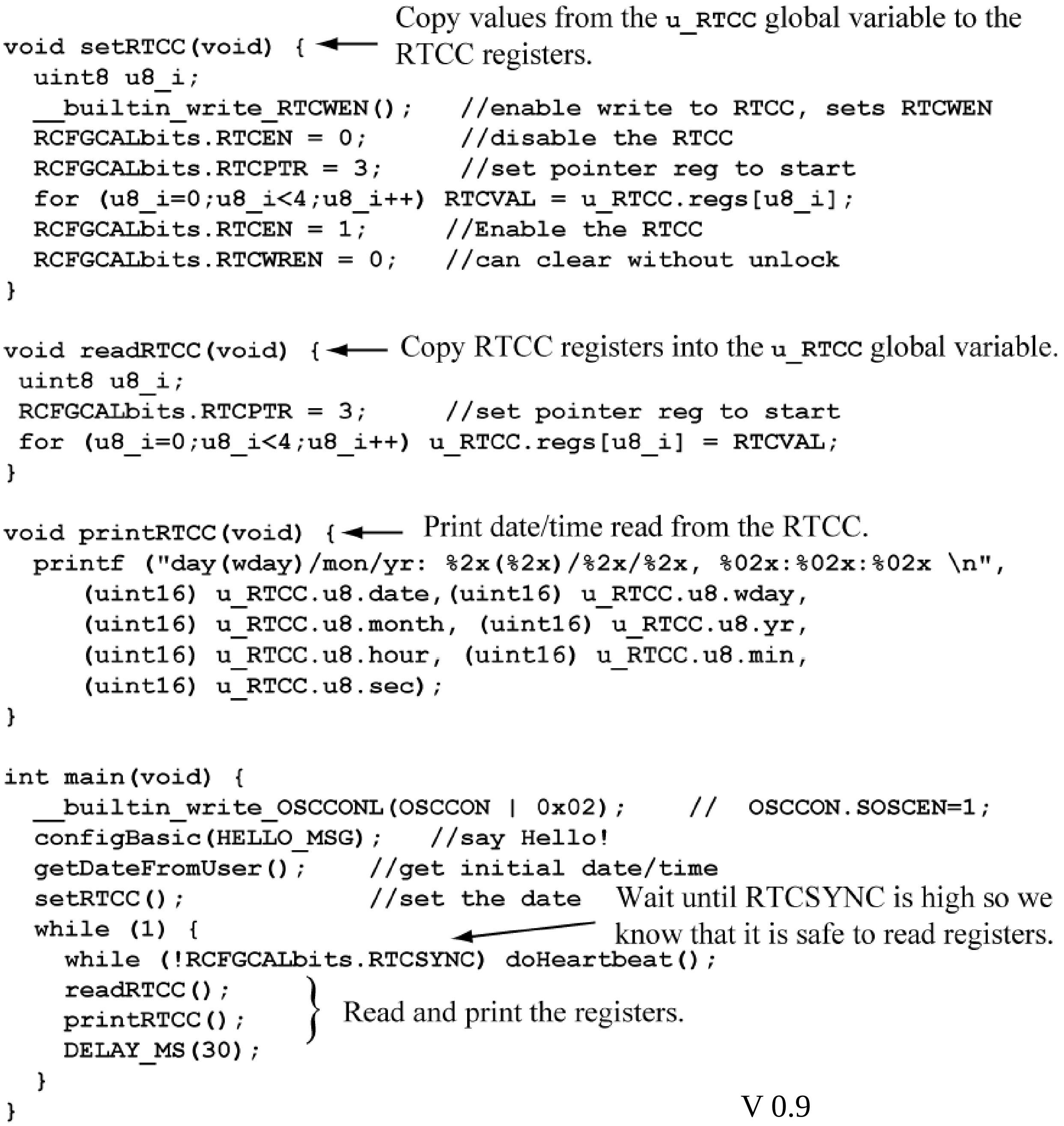
Prompts user with output string `sz_1`, then reads an ASCII-decimal string that is converted to BCD and returned.

Convert binary value `u8_bcd` to bcd value `u8_bcd`.

```
void getDateFromUser(void) {
  u_RTCC.u8.yr = getBCDvalue("Enter year (0-99): ");
  u_RTCC.u8.month = getBCDvalue("Enter month (1-12): ");
  u_RTCC.u8.date = getBCDvalue("Enter day of month (1-31): ");
  u_RTCC.u8.wday = getBCDvalue("Enter week day (0-6): ");
  u_RTCC.u8.hour = getBCDvalue("Enter hour (0-23): ");
  u_RTCC.u8.min = getBCDvalue("Enter min (0-59): ");
  u_RTCC.u8.sec = getBCDvalue("Enter sec(0-59): ");
}
```

Get clock, date settings from user

Test Code (cont)



Copy values from the `u_RTCC` global variable to the RTCC registers.

```
void setRTCC(void) {
  uint8 u8_i;
  __builtin_write_RTCWEN();   //enable write to RTCC, sets RTCWEN
  RCFGCALbits.RTCEN = 0;      //disable the RTCC
  RCFGCALbits.RTCPTR = 3;     //set pointer reg to start
  for (u8_i=0;u8_i<4;u8_i++) RTCVAL = u_RTCC.regs[u8_i];
  RCFGCALbits.RTCEN = 1;     //Enable the RTCC
  RCFGCALbits.RTCWREN = 0;   //can clear without unlock
}
```

Copy RTCC registers into the `u_RTCC` global variable.

```
void readRTCC(void) {
 uint8 u8_i;
 RCFGCALbits.RTCPTR = 3;     //set pointer reg to start
 for (u8_i=0;u8_i<4;u8_i++) u_RTCC.regs[u8_i] = RTCVAL;
}
```

Print date/time read from the RTCC.

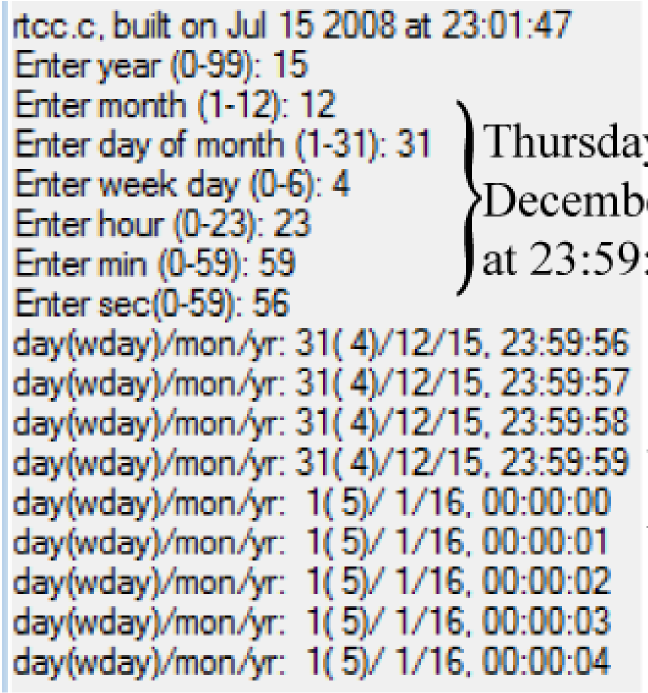
```
void printRTCC(void) {
  printf ("day(wday)/mon/yr: %2x(%2x)/%2x/%2x, %02x:%02x:%02x \n",
     (uint16) u_RTCC.u8.date,(uint16) u_RTCC.u8.wday,
     (uint16) u_RTCC.u8.month, (uint16) u_RTCC.u8.yr,
     (uint16) u_RTCC.u8.hour, (uint16) u_RTCC.u8.min,
     (uint16) u_RTCC.u8.sec);
}
```

```
int main(void) {
  __builtin_write_OSCCONL(OSCCON | 0x02);    //  OSCCON.SOSCEN=1;
  configBasic(HELLO_MSG);   //say Hello!
  getDateFromUser();    //get initial date/time
  setRTCC();            //set the date
  while (1) {
    while (!RCFGCALbits.RTCSYNC) doHeartbeat();
    readRTCC();
    printRTCC();
    DELAY_MS(30);
  }
}
```

Wait until RTCSYNC is high so we know that it is safe to read registers.

Read and print the registers.

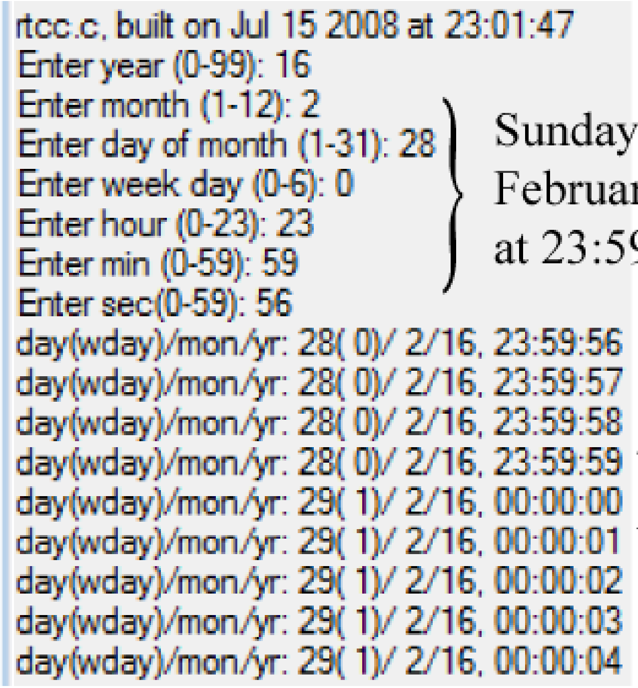
Test Code Output



```
rtcc.c, built on Jul 15 2008 at 23:01:47
Enter year (0-99): 15
Enter month (1-12): 12
Enter day of month (1-31): 31
Enter week day (0-6): 4
Enter hour (0-23): 23
Enter min (0-59): 59
Enter sec(0-59): 56
day(wday)/mon/yr: 31( 4)/12/15, 23:59:56
day(wday)/mon/yr: 31( 4)/12/15, 23:59:57
day(wday)/mon/yr: 31( 4)/12/15, 23:59:58
day(wday)/mon/yr: 31( 4)/12/15, 23:59:59
day(wday)/mon/yr:  1( 5)/ 1/16, 00:00:00
day(wday)/mon/yr:  1( 5)/ 1/16, 00:00:01
day(wday)/mon/yr:  1( 5)/ 1/16, 00:00:02
day(wday)/mon/yr:  1( 5)/ 1/16, 00:00:03
day(wday)/mon/yr:  1( 5)/ 1/16, 00:00:04
```

Thursday,
December 31, 2015,
at 23:59:56

Year,
month,
day
rollover



```
rtcc.c, built on Jul 15 2008 at 23:01:47
Enter year (0-99): 16
Enter month (1-12): 2
Enter day of month (1-31): 28
Enter week day (0-6): 0
Enter hour (0-23): 23
Enter min (0-59): 59
Enter sec(0-59): 56
day(wday)/mon/yr: 28( 0)/ 2/16, 23:59:56
day(wday)/mon/yr: 28( 0)/ 2/16, 23:59:57
day(wday)/mon/yr: 28( 0)/ 2/16, 23:59:58
day(wday)/mon/yr: 28( 0)/ 2/16, 23:59:59
day(wday)/mon/yr: 29( 1)/ 2/16, 00:00:00
day(wday)/mon/yr: 29( 1)/ 2/16, 00:00:01
day(wday)/mon/yr: 29( 1)/ 2/16, 00:00:02
day(wday)/mon/yr: 29( 1)/ 2/16, 00:00:03
day(wday)/mon/yr: 29( 1)/ 2/16, 00:00:04
```

Sunday,
February 28, 2016,
at 23:59:56

Leap year
rollover,
Feb. 29th

What do you have to know?

Output Compare:

In general, how the Output Compare module works.

How PWM works for motor, servo control

Input Capture:

In general, how the Input Capture module works.

How to compute delta Time given two timer captures, number of timer overflows, and PR value. (guaranteed question!)

Definitions of space-width, biphase decoding.

Real-Time Timekeeping – purpose of a 32.768 kHz crystal