



# Reading/Writing Timer0 in 16-bit mode

```
unsigned int tmr0_tics;  
char *ptr;
```

```
(1) ptr = (char *) &tmr0_tics;    // ptr points to LSB of tmr0_tics  
(2) tmr0_tics = TMR0;             // this works for read  
(3) *ptr = TMR0L; *(ptr+1) = TMR0H; // also works for read  
(4) *(ptr+1) = TMR0H, *ptr = TMR0L; // wrong order for read  
(5) TMR0 = tmr0_tics;            // wrong order for write  
(6) TMR0H = *(ptr+1); TMR0L= *ptr; // correct order for write  
(7) TMR0H = (tmr0_tics)>>8; TMR0L= (tmr0_tics & 0xFF); //ok as well
```

When reading, read TMR0L first!

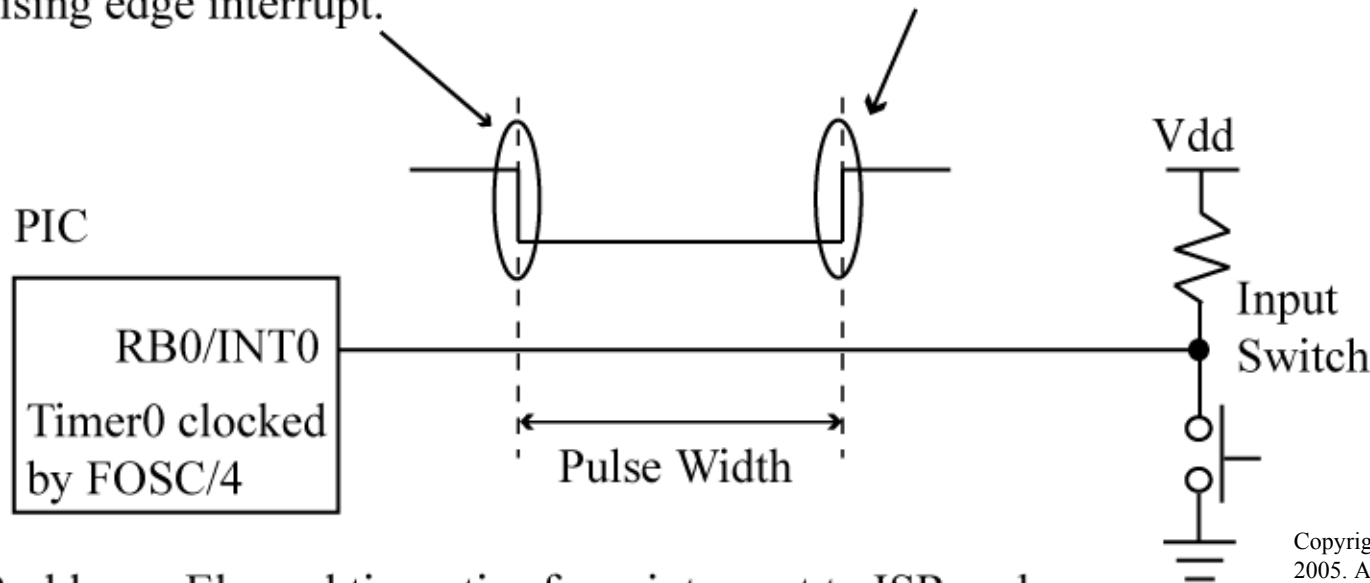
When writing, write TMR0L last!

The statement *tmr0\_tics = TMR0* works for read because TMR0L, TMR0H are stored in little endian order in the file registers, and TMR0L is read first, then TMR0H by the compiler generated code.

# Pulse Width Measurement: Timer0

1. Clear Timer0 to start, configure INT0 for falling edge interrupt.
2. On falling edge INT0 interrupt, turn on Timer0. Configure INT0 for rising edge interrupt.

3. On rising edge interrupt, turn off Timer0; convert the Timer0 value to the elapsed time from falling edge to rising edge.



Copyright Thomson/Delmar Learning  
2005. All Rights Reserved.

Problems: Elapsed timer tics from interrupt to ISR code are not counted, and overflow of Timer0 is not handled.

Inaccurate, as timer0 tics from falling edge interrupt to ISR code are not counted. Also, timer0 tics from rising edge interrupt to ISR code are incorrectly added to total. Also, Timer0 overflow not counted.

# Pulse Width Measurement: Timer0 main

```
pcrlf();printf("Ready for button mashing!");pcrlf();
while(1){
    capture_flag = 0;
    // clear timer0, write low byte last
    TMR0H = 0;
    TMR0L = 0;
    INTEDG0 = 0; // falling edge
    INTOIE = 1; //RB0 Interrupt
    while(!capture_flag); // wait for capture
    // compute time in microseconds
    pulse_width_float = TMR0TIC * tmr0_tics * 1.0e6;
    pulse_width = (long)pulse_width_float;
    printf ("Switch pressed, timer ticks: %d, pwidth: %ld (us)",
            tmr0_tics,pulse_width); pcrlf();
}
}
```

Wait for pulse width to be captured by ISR

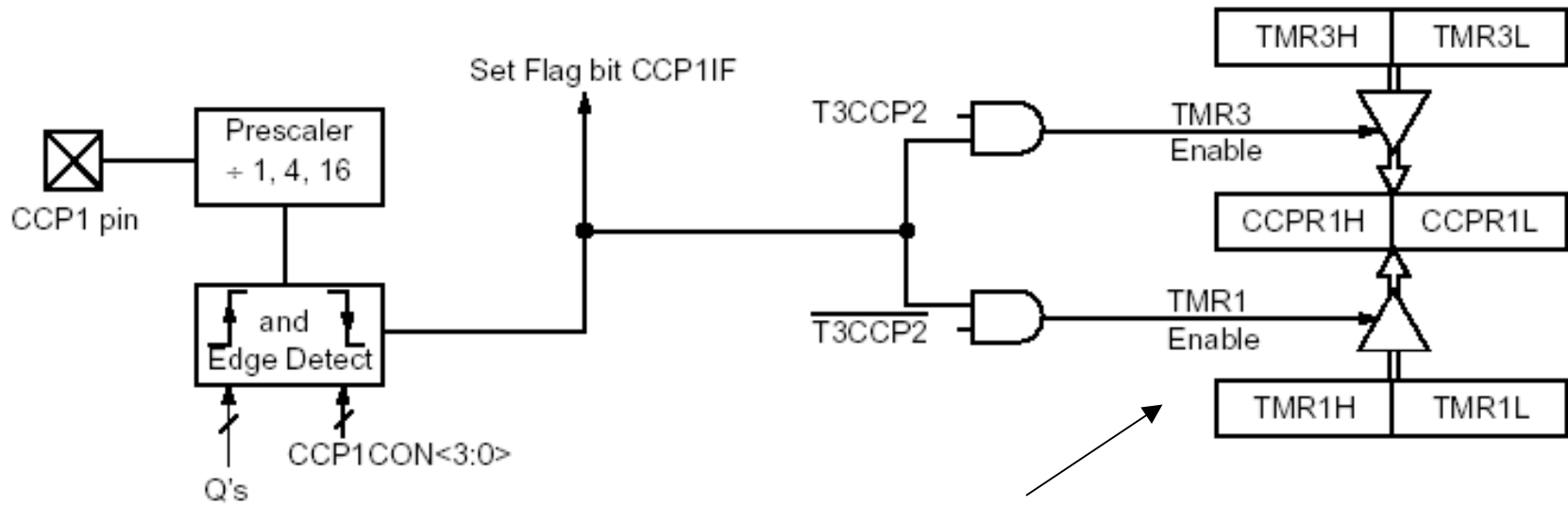
Convert Timer0 tics to microseconds

Configuration code before loop is not shown.

This works of for human activated pushbutton time measurement, but if more accurate measurements are needed, then use the **Capture** module.

# Capture Module Time Measurement

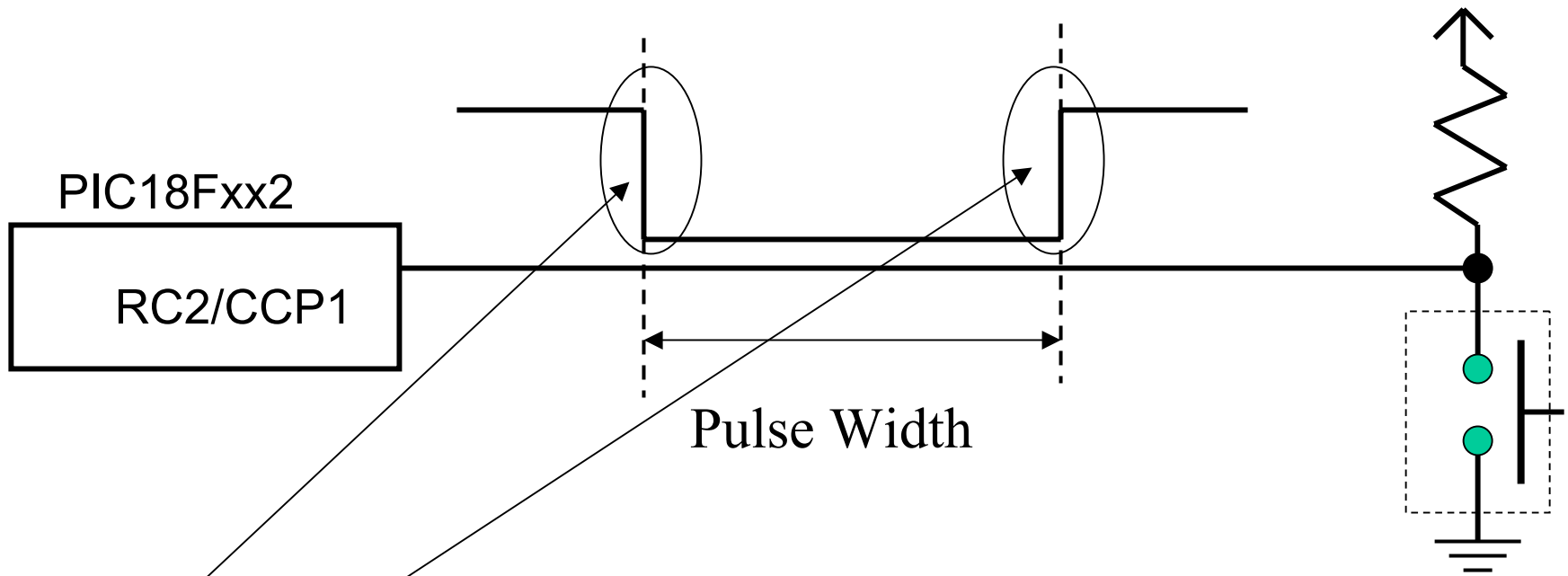
- **Capture Mode** of the Capture/Compare/PWM module is used for time measurement.



Rising or falling edge detect, with interrupt flag set.

TMR1 or TMR3 16-bit value transferred to 16-bit capture register on edge detect.

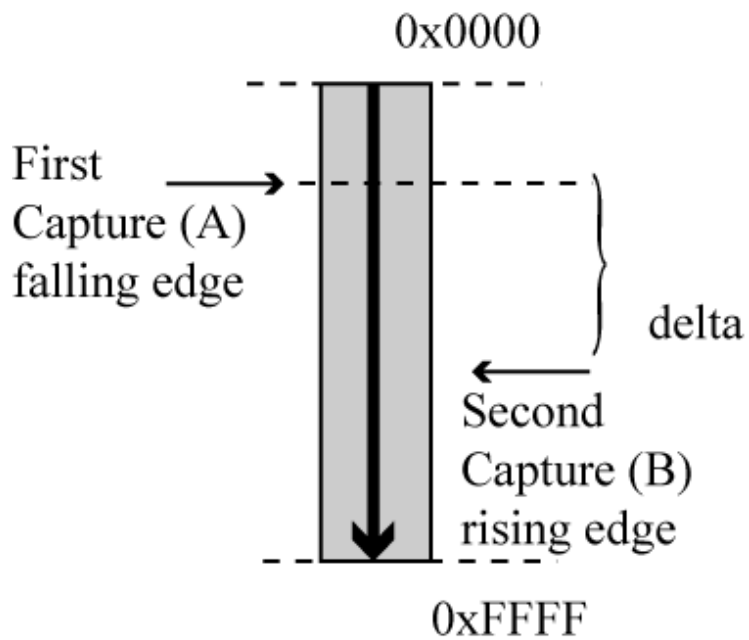
# Measuring pushbutton pulse width



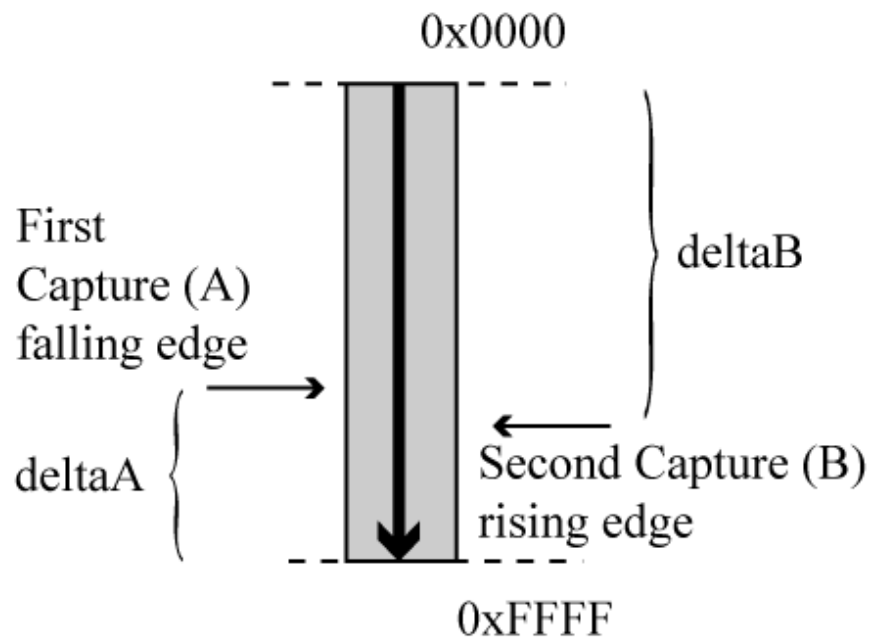
1. Capture TMR1 value on falling edge ( $T_f$ ) in CCPR1
2. Capture TMR1 value on rising edge ( $T_r$ ) in CCPR1
3. Pulse width =  $T_r - T_f$  (Elapsed Timer1 Tics)

Use interrupt to save timer values.

# Computing Pulse Width



(a) No overflow case  
 $\text{TimerDelta} = B - A$



(b) Overflow case  
 $\text{TimerDelta} = (\#\text{offlows}-1) * 65536 + \text{deltaA} + \text{deltaB}$   
 $= (\#\text{offlows} - 1) \ll 2^{16} + (0 - A) + B$

© Thomson/Delmar Learning

In overflow case, the value can be greater  $> 16$  bits so need to use a LONG type to hold TimerDelta value.

```

volatile unsigned int last_capture;
volatile unsigned int this_capture;
// this must be long
volatile unsigned long delta;
// timer 1 overflow cnt
volatile unsigned char tmr1_ov;
volatile unsigned char capture_flag;

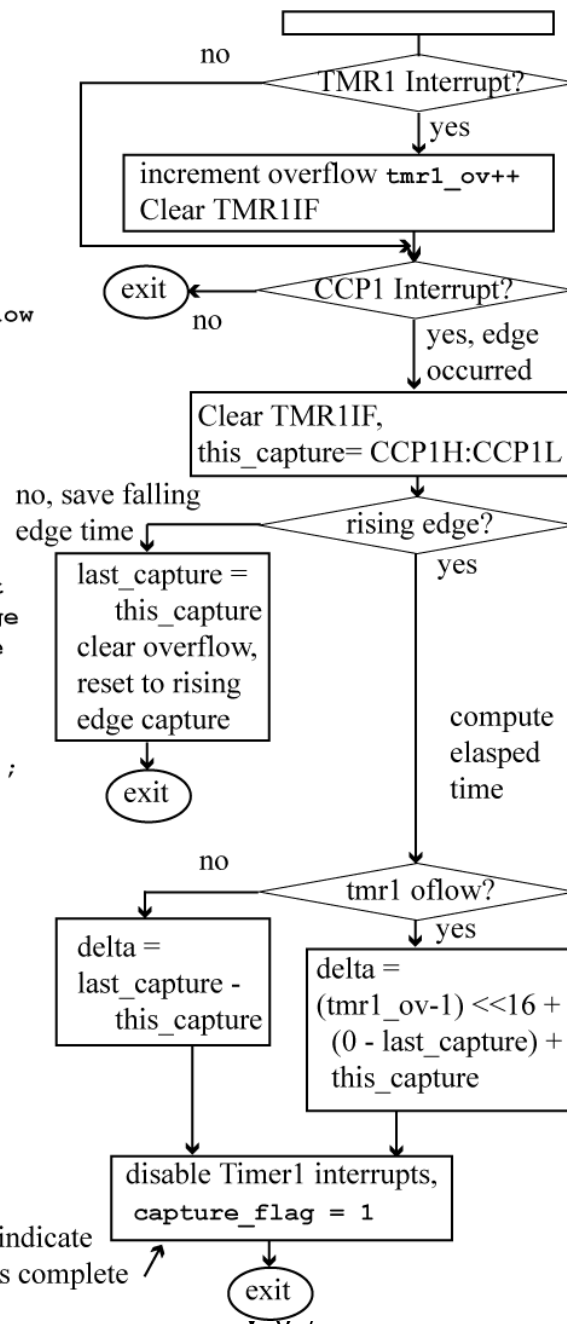
```

```

timer_isr(void){
  if (TMR1IF) {
    tmr1_ov++; // increment timer1 overflow
    TMR1IF = 0;
  }
  if (CCP1IF) {
    // read CCP1 as a 16-bit value
    this_capture = CCP1;
    if (!bittst(CCP1CON,0)) {
      //falling edge
      last_capture = this_capture;
      tmr1_ov = 0; // clear overflow count
      CCP1CON = 0x0; // turn off when change
      CCP1CON = 0x5; // capture rising edge
    } else {
      if (!tmr1_ov) {
        // no overflow at all
        delta = this_capture - last_capture ;
      }
      else {
        // compute delta time
        delta = tmr1_ov-1;
        delta = (delta << 16);
        last_capture = 0 - last_capture;
        delta = delta + last_capture;
        delta = delta + this_capture;
      }
      // disable timer1 interrupt
      TMR1ON = 0; TMR1IE = 0; TMR1IF = 0;
      capture_flag = 1;
    }
    //clear capture interrupt flag
    CCP1IF = 0;
  }
}

```

Semaphore to main() to indicate that pulse width capture is complete



ISR for capturing pulse width.

tmr1\_ov variable keeps track of timer1 overflows.

After falling edge, reconfigure for rising edge capture.

After rising edge, compute delta timer ticks

```

volatile unsigned int last_capture;
volatile unsigned int this_capture;
// this must be long
volatile unsigned long delta;
// timer 1 overflow cnt
volatile unsigned char tmr1_ov;
volatile unsigned char capture_flag;

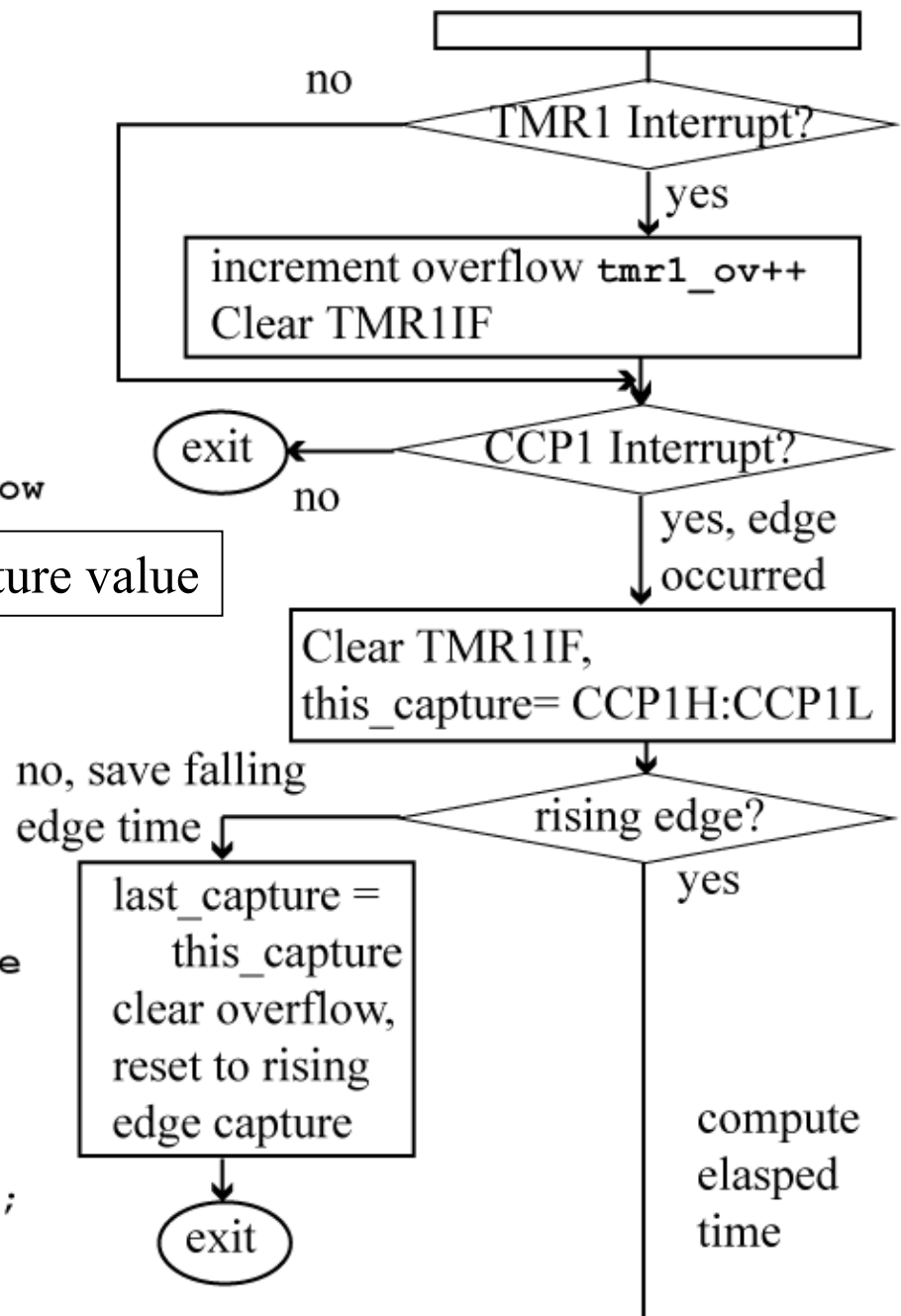
```

```

timer_isr(void) {
  if (TMR1IF) {
    tmr1_ov++; // increment timer1 overflow
    TMR1IF = 0;
  }
  if (CCP1IF) {
    // read CCP1 as a 16-bit value
    this_capture = CCP1H:CCP1L;
    if (!bittst(CCP1CON,0)) {
      //falling edge
      last_capture = this_capture;
      tmr1_ov = 0; // clear overflow count
      CCP1CON = 0x0; // turn off when change
      CCP1CON = 0x5; // capture rising edge
    } else {
      if (!tmr1_ov) {
        // no overflow at all
        delta = this_capture - last_capture ;
      }
    }
  }
}

```

Read 16-bit capture value

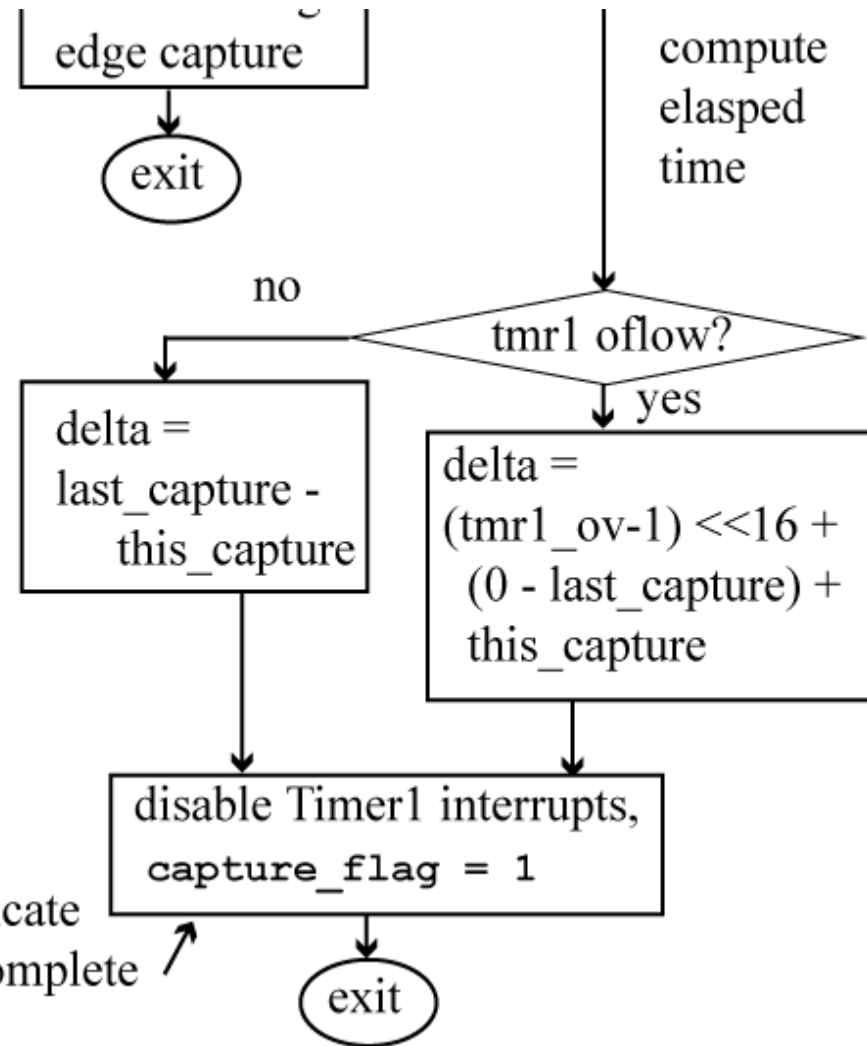


```

if (!tmr1_ov) {
    // no overflow at all
    delta = this_capture - last_capture ;
}
else {
    // compute delta time
    delta = tmr1_ov-1;
    delta = (delta << 16);
    last_capture = 0 - last_capture;
    delta = delta + last_capture;
    delta = delta + this_capture;
}
// disable timer1 interrupt
TMR1ON = 0; TMR1IE = 0; TMR1IF = 0;
capture_flag = 1;
}
//clear capture interrupt flag
CCP1IF = 0;
}
}

```

Semaphore to main() to indicate that pulse width capture is complete

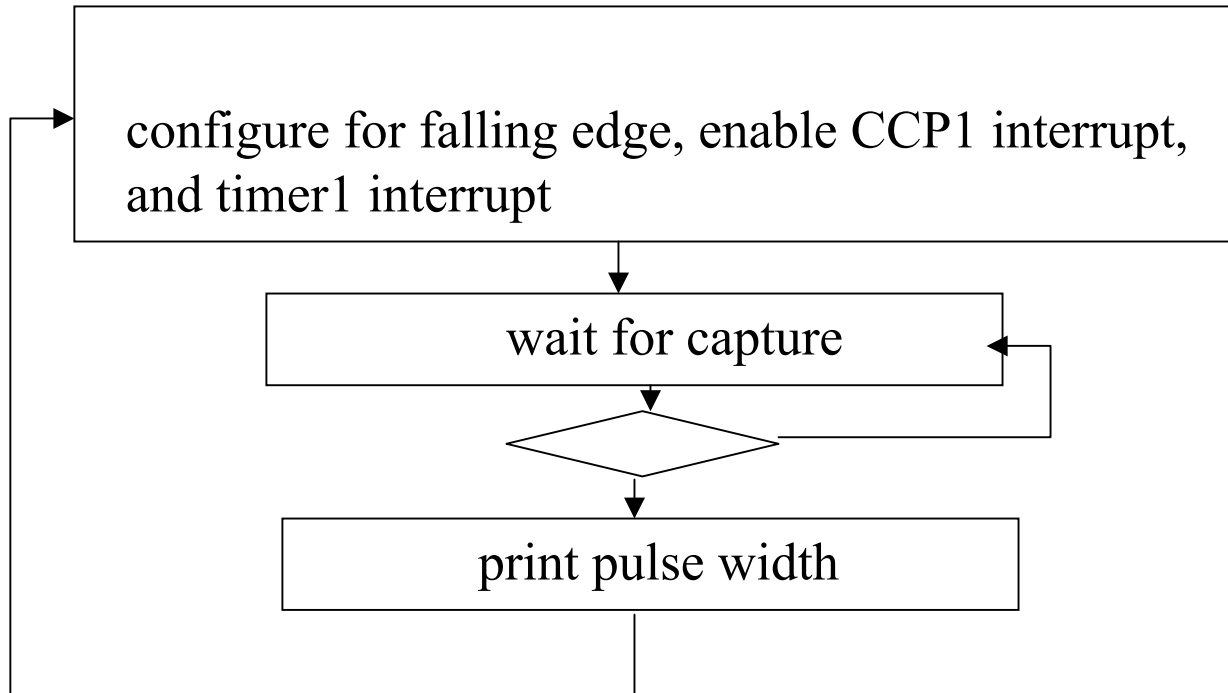


After pulse width is captured, the *capture\_flag* semaphore is set and the Timer0 interrupt is disabled as the pulse width has been measured.

# swdetov.c (configuration)

```
#define FOSCQ 29491200
#define PRESCALE 2.0
#define TMR1TIC 1.0/(FOSCQ/4.0)*PRESCALE
double pulse_width_float;
long pulse_width;
main(void) {
    serial_init(95,1); // 19200 in HSPLL mode
    // initialize timer 1
    // prescale by 2 ← Initialize Timer1
    T1CKPS1 = 0; T1CKPS0 = 1;
    T1OSCEN = 0; // disable the oscillator
    TMR1CS = 0; //use internal clock FOSC/4
    T1SYNC = 0;
    // set CCP1 as input
    bitset(TRISC,2); ← CCP1 used as input pin.
    // enable interrupts
    IPEN = 0; PEIE = 1; GIE = 1;
```

# swdet.c (main loop)



# swdetov.c (main loop)

```
printf("(Timer1 version) Ready for button mashing!");
pcrlf();
while(1) {
    // configure capture
    CCP1IE = 0;    // disable when changing modes
    CCP1CON = 0x0; // turnoff before changing
    CCP1CON = 0x4; // capture every falling edge
    CCP1IF = 0; // clear CCP1IF interupt flag
    CCP1IE = 1;    // enable capture interrupt
    TMR1IF = 0;    // clear timer 1 interrupt flag
    TMR1IE = 1;    // allow timer 1 interrupts
    TMR1ON = 1;    // enable timer 1
    capture_flag = 0;
    while(!capture_flag); //wait for capture
    pulse_width_float = (delta * TMR1TIC)*1.0e6;
    pulse_width = (long) pulse_width_float;
    printf ("Switch pressed, timer ticks: %ld, pwidth: %ld (us)",
    delta,pulse_width);
    pcrlf();
}
}
```

floating point computation to convert Timer tics to microseconds.

compile with `-lf` flag to print long variables



# Implementing a Clock/Calendar

Use 32.768KHz crystal on T1OS0/T1OS1 pins, this crystal frequency good for accurate time keeping

$$32.768\text{KHz} = 32768 \text{ Hz}$$

→ when 16-bit counter rolls over, then exactly 2 seconds

→ when counter = 0x8000, exactly 1 sec

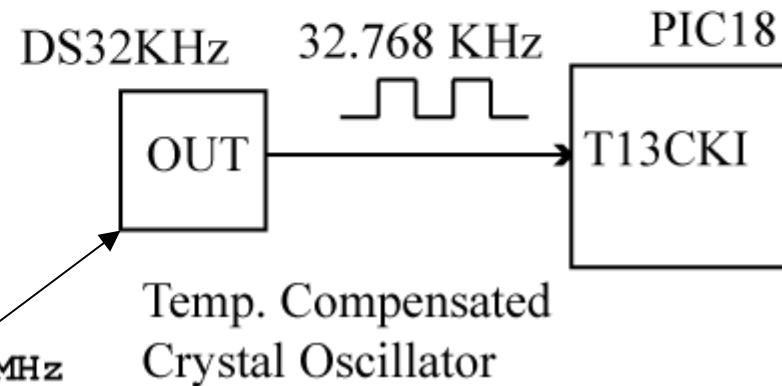
No error in time keeping, good for long term clock/calendar function.

# Clock/Calendar TMR1

```
// does simple timekeeping, assumes 32.768KHz ext. clk
volatile unsigned char hours, mins, secs,old_secs;
void interrupt timer_isr(void){
  if (TMR1IF) {
    TMR1IF = 0;
    secs = secs + 2; // seconds
    if (secs == 60) {
      mins++;
      secs = 0;
      if (mins == 60) {
        mins = 0;
        hours++;
        if (hours == 24) hours = 0;
      }
    }
  }
}
```

With 32.768 KHz ext. clock, interrupt occurs every 2 seconds on overflow of 16-bit timer1.

```
main(void) {
  // 19200 in HSPLL mode, crystal = 7.3728 MHz
```



External clock source for timer1

```

main(void) {
    // 19200 in HSPLL mode, crystal = 7.3728 MHz
    serial_init(95,1);
    // initialize timer 1
    // prescale by 1
    T1CKPS1 = 0; T1CKPS0 = 0;
    T1OSCEN = 0; // disable the oscillator
    TMR1CS = 1; //use external clock
    T1SYNC = 0; // sync extern clock
    // set T1CKI/RC0 as input
    bitset(TRISC,0);

```

Temp. Compensated  
Crystal Oscillator

Copyright Thomson/Delmar  
Learning 2005. All Rights  
Reserved.

} Initialize timer1

Clock/Calendar

TMR1 (cont)

```

pcrlf();
printf("(2 sec version) Enter hours, mins, secs: ");
scanf("%d %d %d", &hours, &mins, &secs);
TMR1IF = 0; // clear timer 1 interrupt flag
TMR1IE = 1; // allow timer 1 interrupts
TMR1ON = 1; // enable timer 1
IPEN = 0; // priorities disabled
PEIE = 1; GIE = 1;

```

} Input initial values  
for min, hour, sec

} Enable timer1  
interrupt

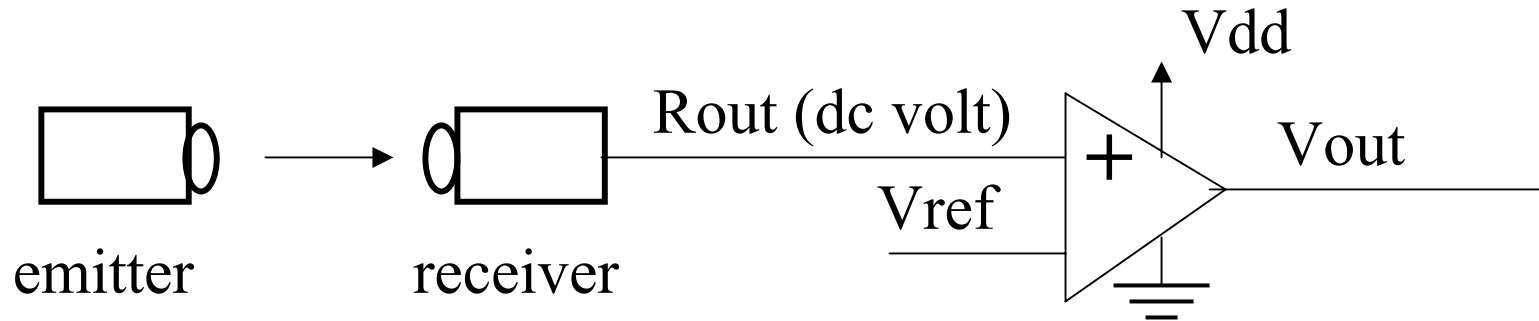
```

while(1) {
    while(secs == old_secs); ← Wait for time to change
    old_secs = secs;
    pcrlf(); printf(" %d:%d:%d", hours, mins, secs); } Print time
}

```

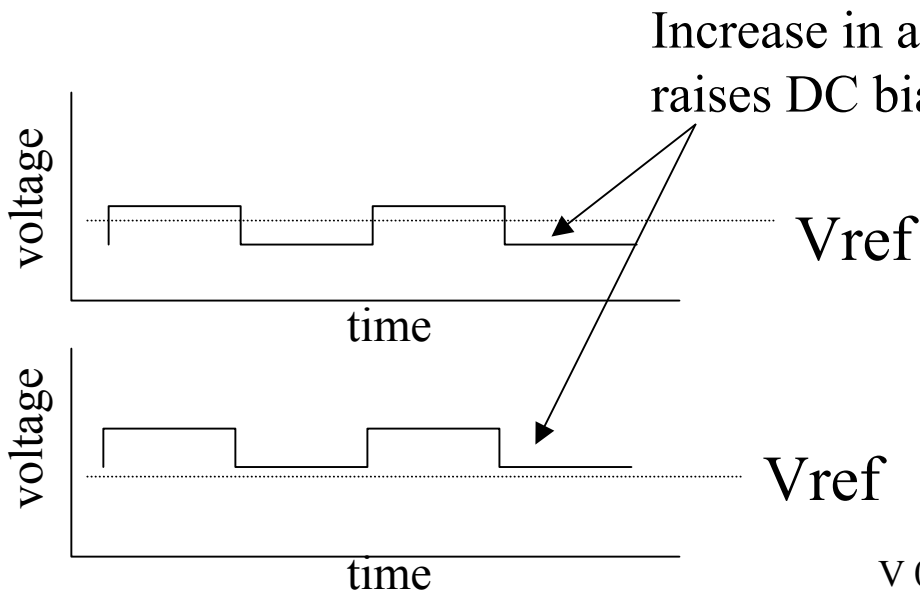
(2 sec version) Enter hours, mins, secs: 0 0 0

# Intensity Based Infrared



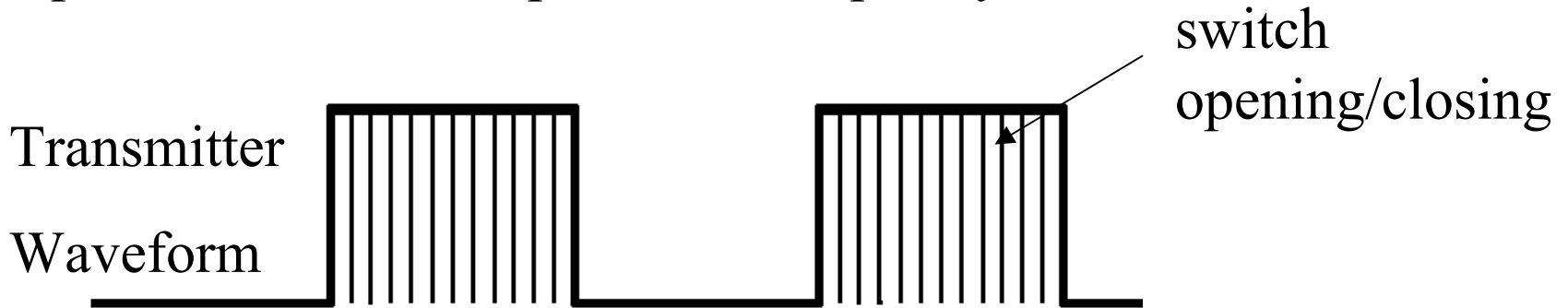
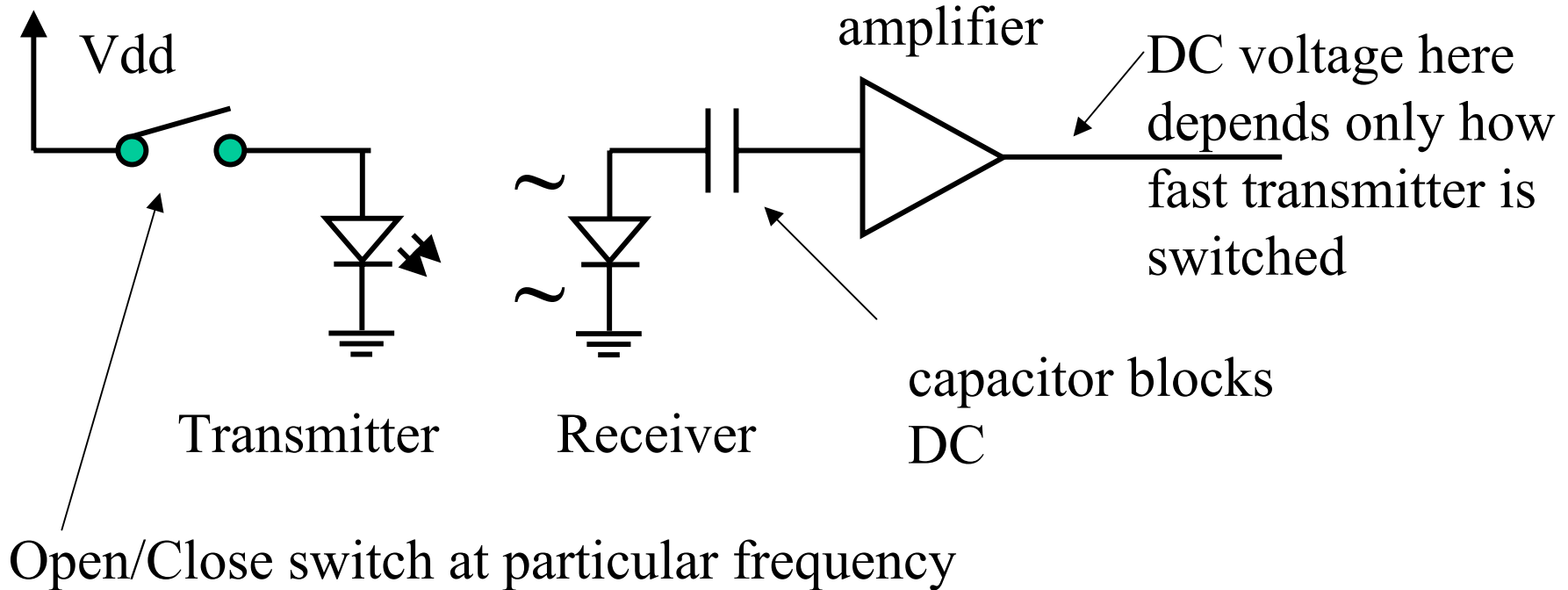
$V_{out} = V_{dd}$ , IR present ( $R_{out} > V_{ref}$ )

$V_{out} = 0v$ , IR absent ( $R_{out} < V_{ref}$ )



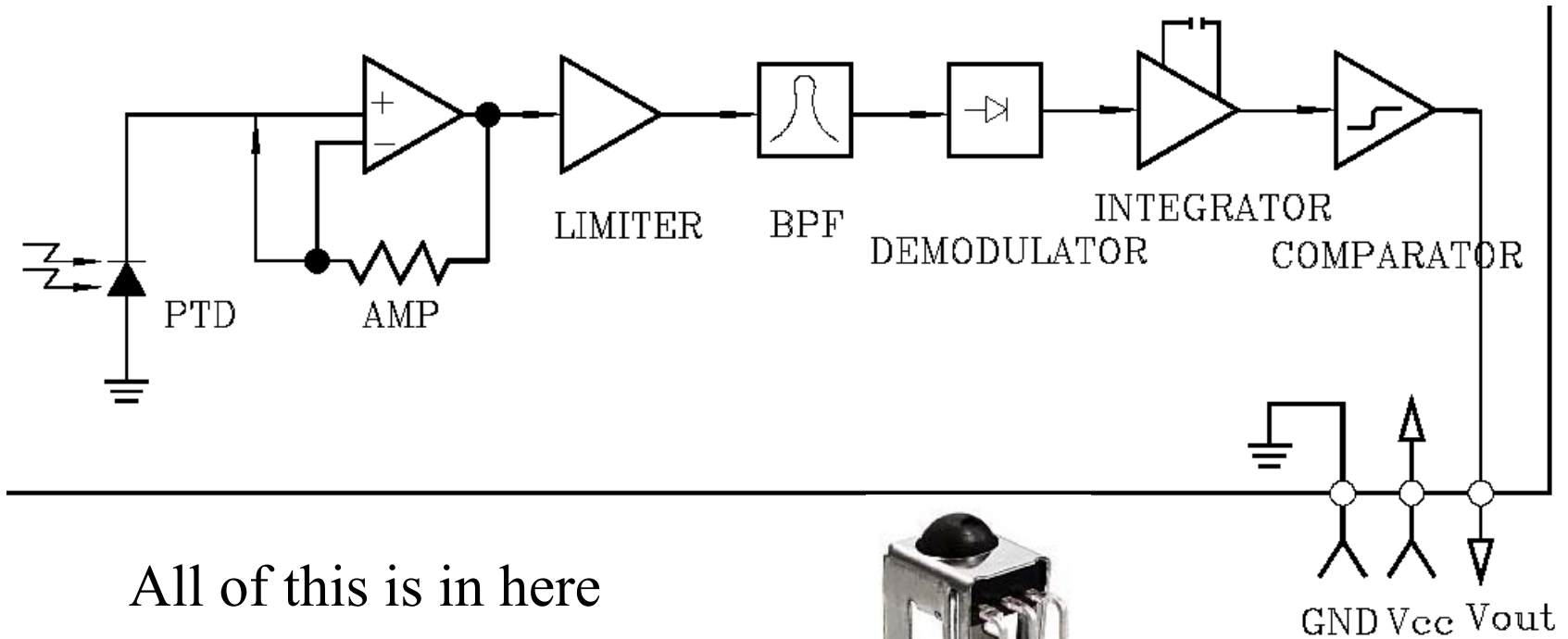
Problem: value for  $V_{ref}$  changes depending on ambient light!

# How to block Ambient Light?

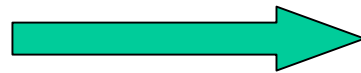


# Integrated IR Receiver

Actual IR receiver a bit more complicated



All of this is in here



V 0.7

Out 1

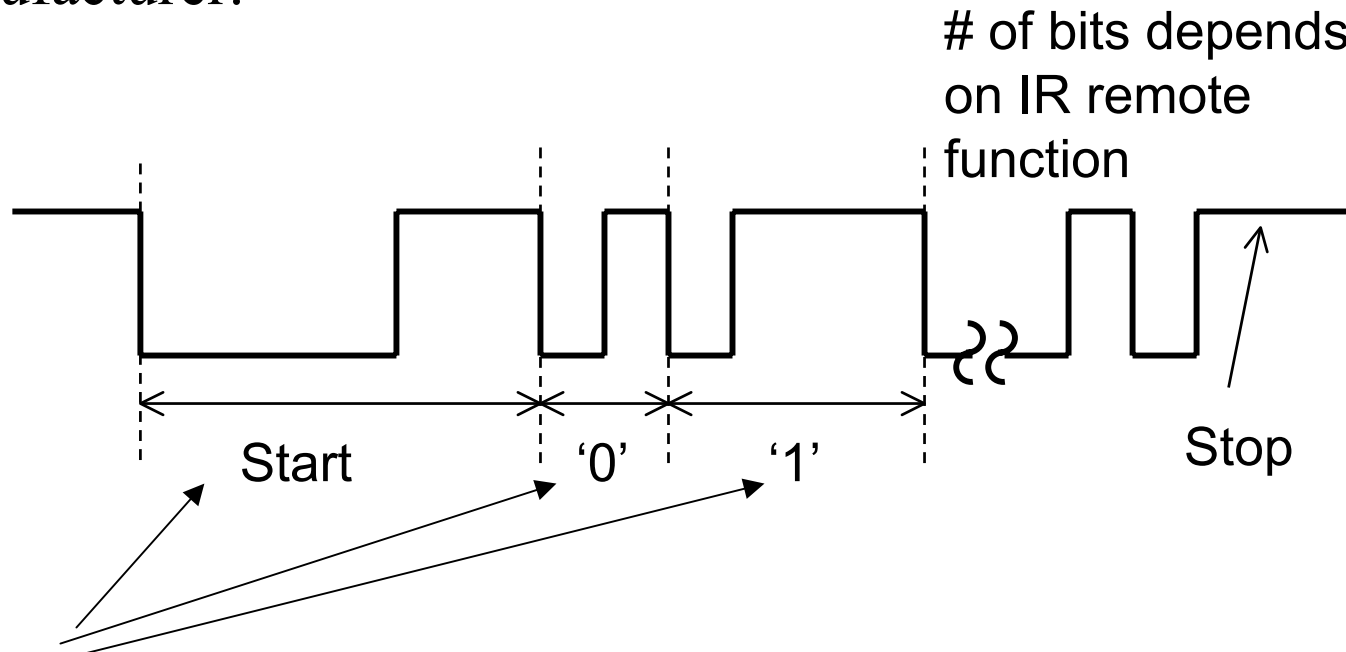
2 3

Gnd

VCC

# IR Waveform

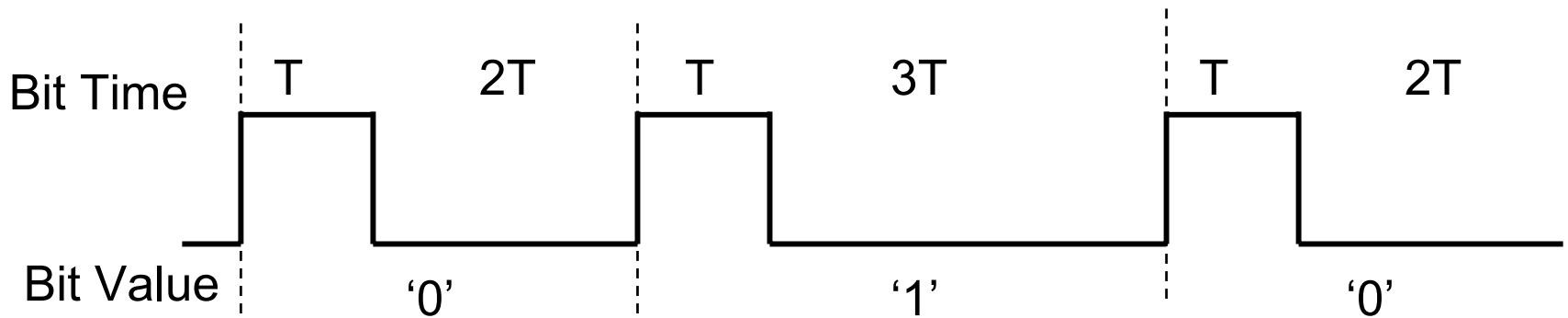
Waveform produced by receiver when stimulated by a universal remote control depends on function and manufacturer.



Length of start period, '0' period, '1' period will vary with function. '0', '1' waveforms may be inverted. Called *space-width* modulation.

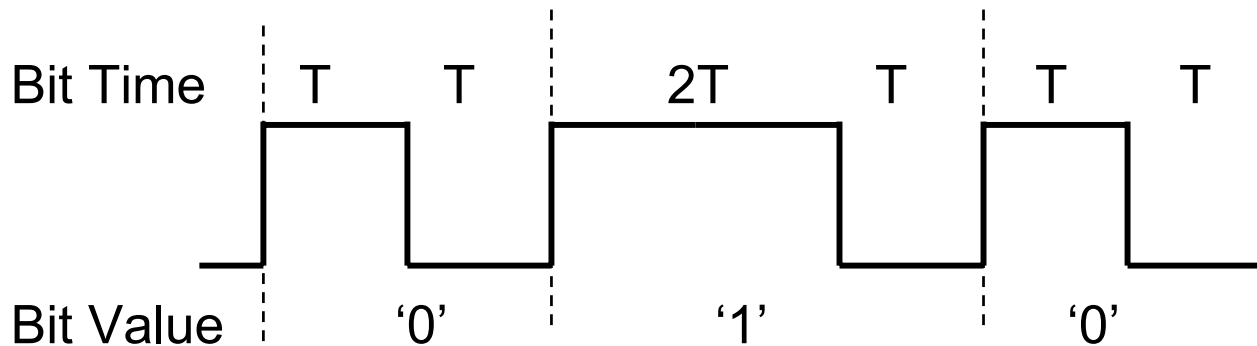
# Space-Width Encoding Examples

REC-80 code, Panasonic: '0' period =  $3T$ , '1' period =  $4T$



---

Sony code: '0' period =  $2T$ , '1' period =  $3T$



# Lab #12: Decoding IR Waveform

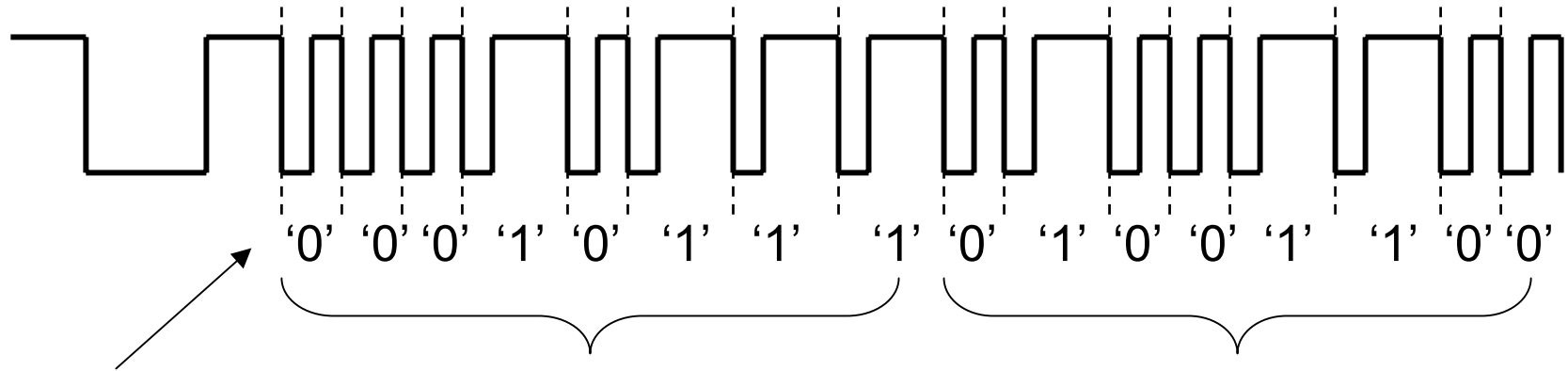
You will be assigned a function on the Universal Remote.

Use the scope and determine the periods of ‘start’, ‘0’, and ‘1’ (assume a ‘0’ the short period, a ‘1’ is the long period).

Using ‘swdetov.c’ as a starting point, decode the first two bytes of a frame and print these two bytes out to the screen via Hyperterm. Choose a particular prescale for Timer1 to give you enough fidelity on Timer1 to distinguish ‘1’, ‘0’ periods.

Your particular function may have LOTS of bits in a frame, just decode the first two bytes (16 bits). If your function does NOT have at least 16 bits, then get the TA to assign you a different IR remote function (or find one, and show this to the TA).

# Example Waveform



assume  
data sent  
MSB first

First Byte

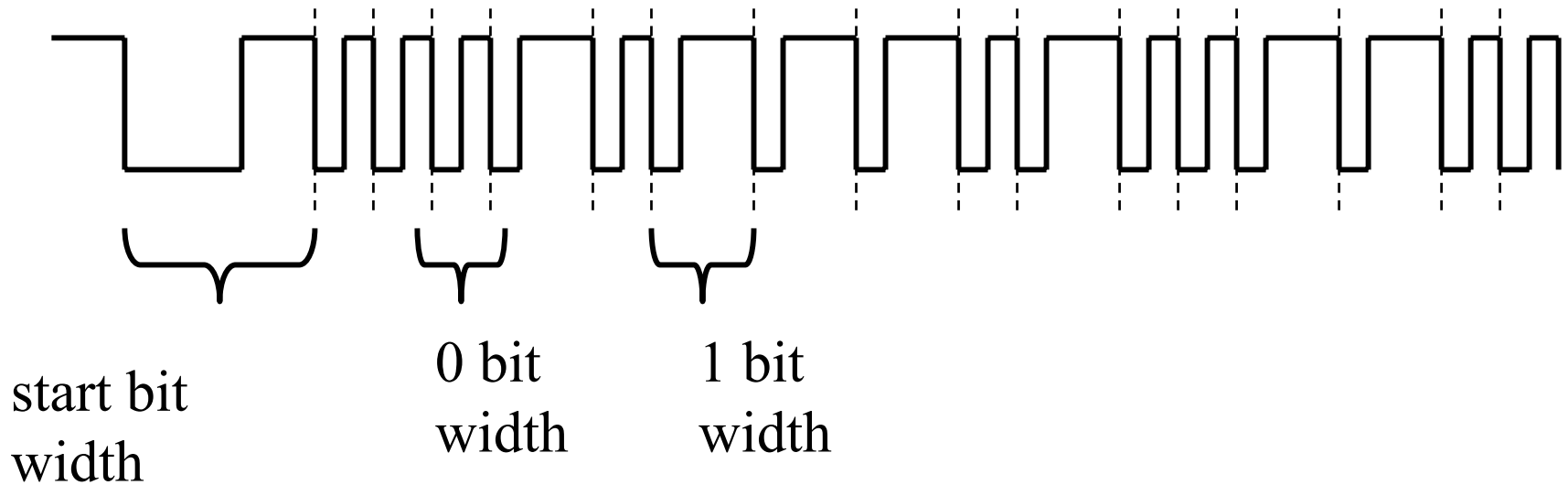
00010111 = 0x17

Second Byte

01001100 = 0x4C

Print first two bytes of frame to screen.

# Space-width Decoding Approach



Measure the time (timer ticks) between successive FALLING edges. This is one bit time.

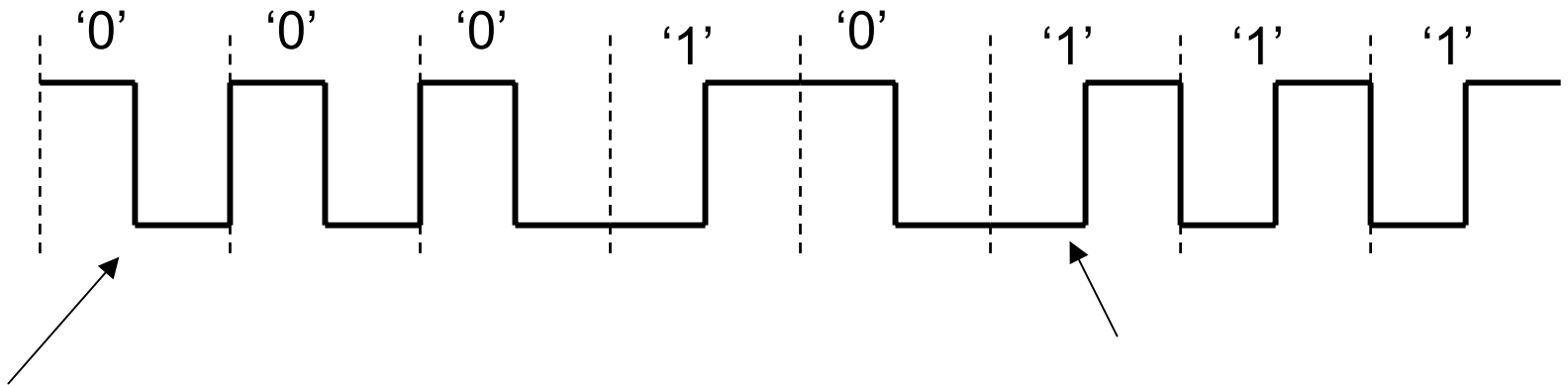
If very first period measurement, then this is start bit.

If period width is less than '1' bit width (give yourself some margin) then this is a '0', else it is a '1'.

# Biphase Encoding

In a previous example, '1' and '0' were distinguished by having different periods.

Some Remote function/manufacture use biphase encoding; '1' and '0' have same period, but use different transition in middle of the period (low-to-high or high-to-low).



transition high-to-low is a '0'

transition low-to-high is a '1'

# Experiment 12 Notes

Ensure that your assigned IR Remote function does NOT use biphasse encoding (must use *space-width* encoding).

Simply measure the time between falling edges – the first period will be the start period, the periods after that will be either ‘1’ or ‘0’.

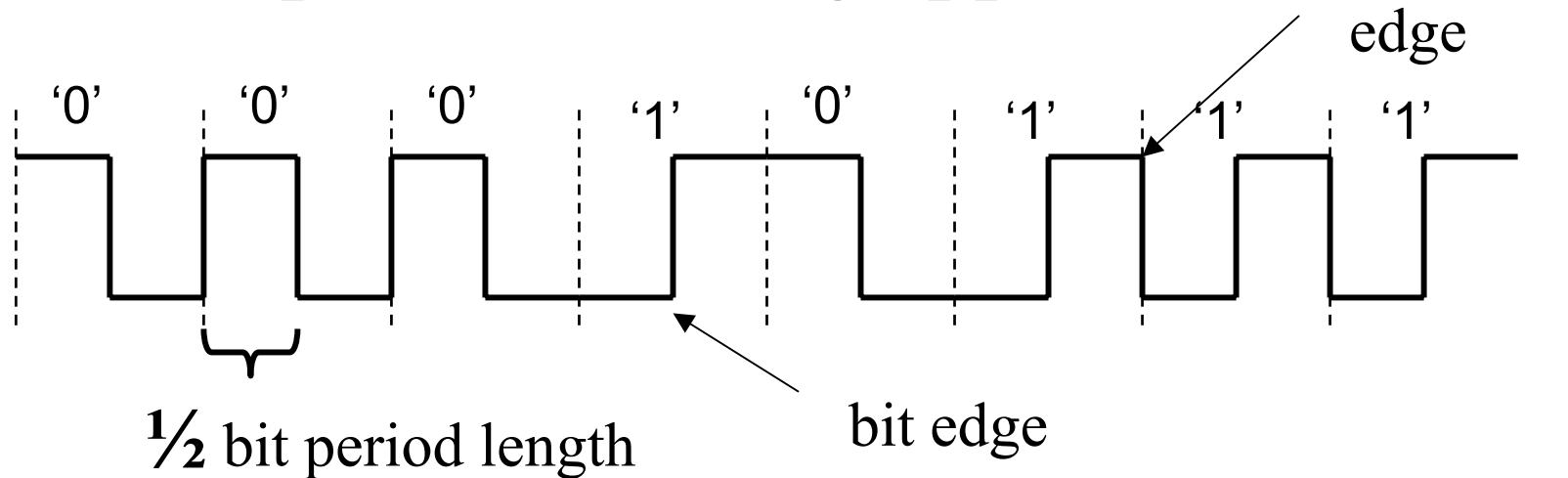
Compute the number of timer tics for a ‘1’ or ‘0’, and compare what is measured. Use some slack, if you compute 2000 timer ticks for a ‘0’, and 4000 for a ‘1’, then distinguish a ‘1’ as being greater than 3000 tics.

You must compute an appropriate prescale for timer1 so that your timer tics will have necessary fidelity to distinguish between ‘0’ and ‘1’.

# Writing the Space-width Decode Code

- Two examples are provided
  - `swdetov.c` – measures switch pulse width
  - `irdet_biphase.c` – decodes IR biphase data
- Either example can be used as a starting point for your code
  - `swdetov.c` is simpler, but requires a lot more code to be added. Basically just provides an ISR that measures time between edges.
  - `biphase.c` might be a better starting point but you must understand how it works. It is more complex than what you need; you need to simplify its operation in order to perform space-width decoding.

# biphase decoding approach



This explanation is provided to help you understand `irdet_biphase.c`

Measure time between EACH edge (this is different from space-width encoding).

If time is one-half bit period, then this bit is equal to last bit value.

If time is equal to a full bit period, and this edge is a bit edge, then the bit value has changed!

# Notes on `irdet_biphase.c`

ISR measures time between EVERY edge (falling and rising) on IR input.

ISR calls a function named `do_ircap()` that does the main job of decoding the IR waveform. It uses a state machine approach to decoding the waveform.

`do_ircap` is called on any timer1 overflow or any input capture (input capture triggers on both rising and falling edges).

If an input capture has occurred, the *delta* variable has the time measurement from the current edge to the last edge.

The textbook has a discussion on the `irdet_biphase.c` code.

## ISR, part 1.

```
volatile unsigned int last_capture, this_capture;
volatile unsigned long delta;
volatile unsigned char tmr1_ov; // timer 1 overflow cnt
```

```
#define MAXBYTES 8
```

```
volatile unsigned char cbuff[MAXBYTES];
volatile unsigned char bitcount, bytecount, bit_edge;
volatile unsigned char state, edge_capture, current_bit;
volatile unsigned char this_byte;
```

buffer used for to store  
IR received bytes.

```
#define IDLE_TIME 4
```

```
#define BITCHANGE 10000
```

```
#define IDLE 0
```

```
#define START_PULSE 1
```

```
#define BIT_CAPTURE 2
```

```
#define IO_FINISH 3
```

# of Timer1 tics to detect change from 0 to 1 or  
vice versa

} State definitions for do\_ircap() function

```
void interrupt timer_isr(void) {
```

```
    if (TMR1IF) {
        tmr1_ov++; // increment timer1 overflow
    }
```

```
    if (CCP1IF) {
        // read CCPR1 as 16-bit value
        this_capture = CCPR1;
        if (!tmr1_ov) {
```

Interrupt on timer1  
overflow or

CCP1 edge  
capture.

Compute number of

```

void interrupt timer_isr(void) {
    if (TMR1IF) {
        tmr1_ov++; // increment timer1 overflow
    }
    if (CCP1IF) {
        // read CCP1 as 16-bit value
        this_capture = CCP1;
        if (!tmr1_ov) {
            // no overflow at all
            delta = this_capture - last_capture ;
        } else {
            delta = tmr1_ov-1;
            delta = (delta << 16);
            last_capture = 0 - last_capture;
            delta = delta + last_capture;
            delta = delta + this_capture;
        }
        last_capture = this_capture;
        tmr1_ov = 0; // clear timer 1 overflow count
        if (bittst(CCP1CON,0)) {
            CCP1CON = 0x0; //reset first
            CCP1CON = 0x4; //falling edge
        }
        else {
            CCP1CON = 0x0; //reset first
            CCP1CON = 0x5; // rising edge
        }
        edge_capture = 1;
    }
    do_ircap(); // Call function that decodes captured edges
}

```

ISR keeps track of timer1 overflows, and measures time between EVERY edge.

Compute number of Timer1 tics between last active edge and current active edge.  
Store this value in variable delta.

delta variable keeps track of time between edges.

Reconfigure CCP1 to toggle active edge from falling edge to active edge or vice versa.  
Do this so can trigger on every edge.

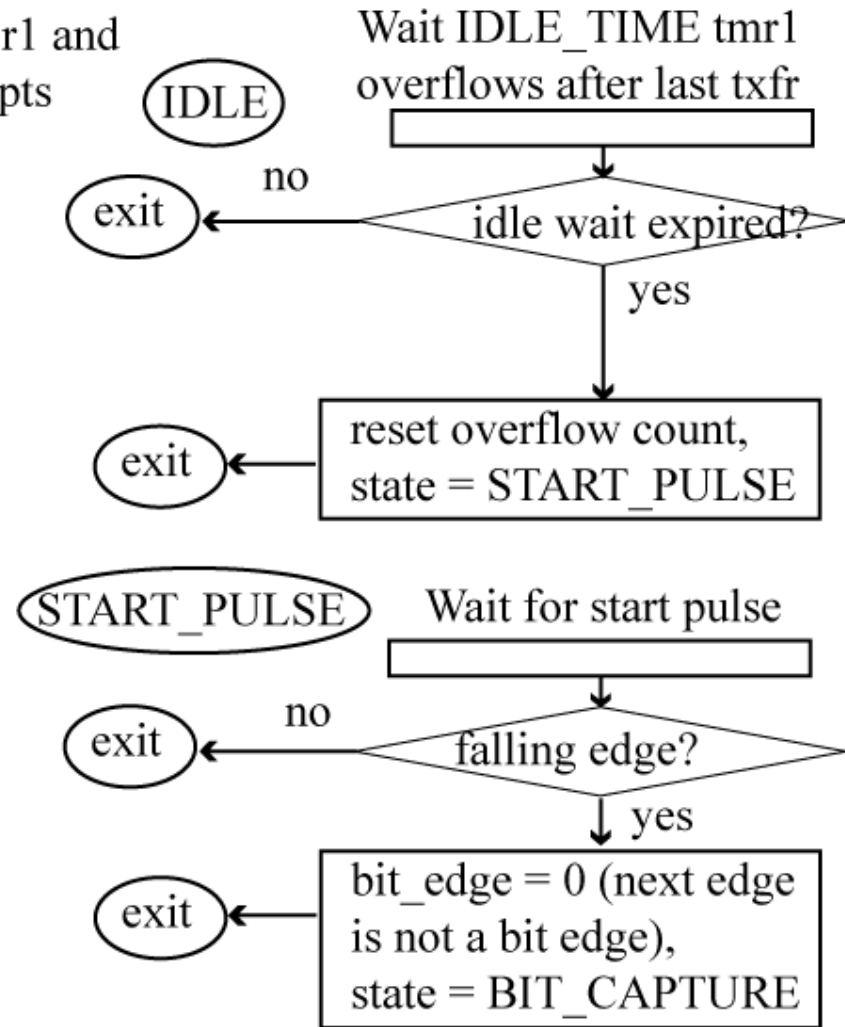
Does real work of decoding bits.

}

# do\_ircap()

```
// decode IR biphas  
void do_ircap() {  
    TMR1IF = 0; CCP1IF = 0;  
    switch (state) {  
        case IDLE:  
            // wait for line to become idle  
            if (tmr1_ov > IDLE_TIME) {  
                tmr1_ov = 0;  
                state = START_PULSE;  
                edge_capture = 0;  
            }  
            break;  
        case START_PULSE:  
            if (edge_capture) { // wait for edge  
                edge_capture = 0;  
                bit_edge = 0;  
                state = BIT_CAPTURE;  
            }  
            break;  
        case BIT_CAPTURE:  
            // wait for edge or idle condition  
            if (tmr1_ov > 1) { // finished
```

Called for both timer1 and  
edge capture interrupts



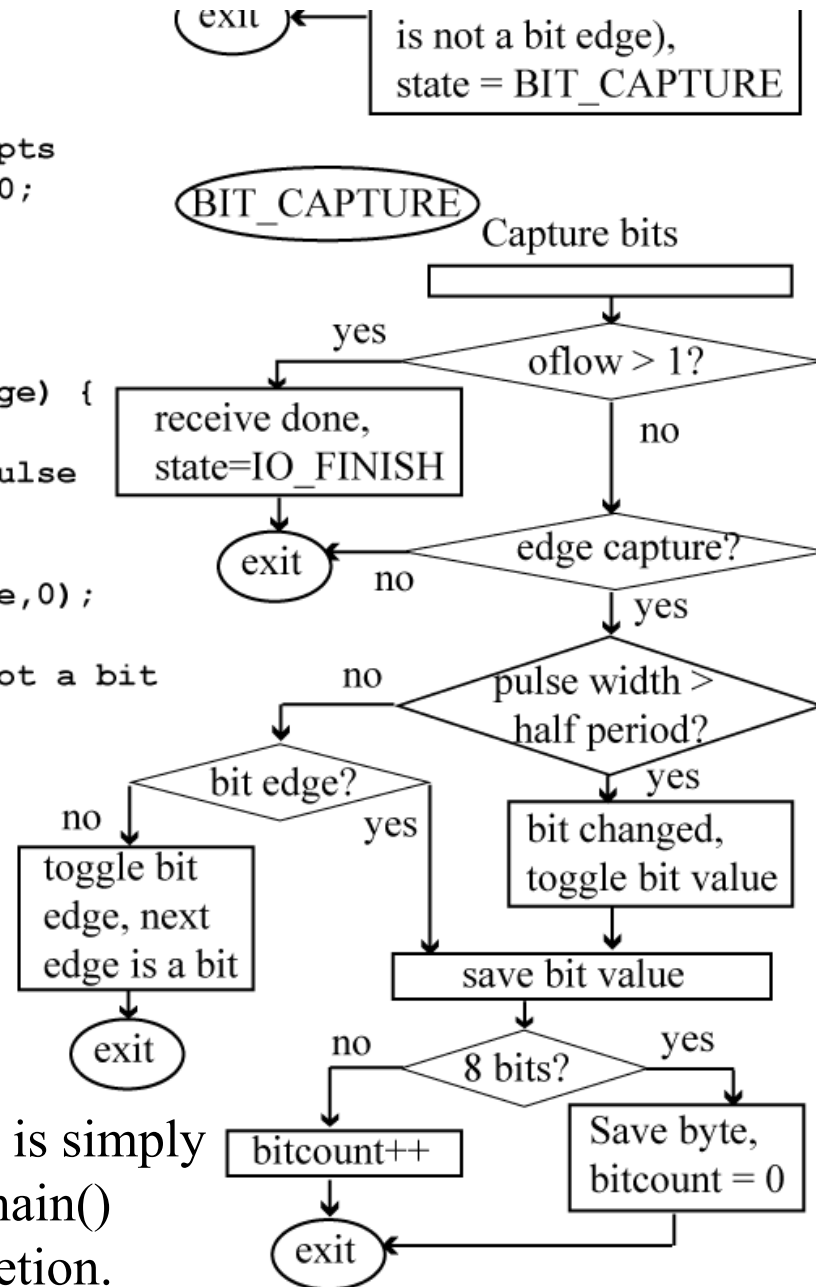
# do\_ircap() cont.

```

case BIT_CAPTURE:
  // wait for edge or idle condition
  if (tmr1_ov > 1) { // finished
    // disable capture, timer1 interrupts
    CCP1IE = 0; TMR1IE = 0; TMR1ON = 0;
    state = IO_FINISH;
  } else if (edge_capture) {
    edge_capture = 0;
    //accumulating bits, MSB to LSB
    if ((delta > BITCHANGE) || bit_edge) {
      if (delta > BITCHANGE) {
        // toggle current bit if wide pulse
        current_bit = ~current_bit;
      }
      if (current_bit) bitset(this_byte, 0);
      bitcount++;
      bit_edge = 0; // next edge is not a bit
      if (bitcount == 8) {
        bitcount = 0;
        cbuff[bytecount] = this_byte;
        bytecount++;
        this_byte = 0;
      } else {
        this_byte = this_byte << 1;
      }
    } else if (!bit_edge)
      bit_edge = 1;
  }
  break;
} //end switch
} // end do_ircap()

```

state IO\_FINISH is simply a semaphore to main() indicating completion.



Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

# main() while loop

```
pcrlf(); printf("Ready for capture\n"); pcrlf();
while(1) {
    // wait for IR data to arrive
    while (state != IO_FINISH);
    // copy interrupt data to safe place
    t_bytecount = bytecount;
    t_bitcount = bitcount;
    t_this_byte = this_byte;
    for (i = 0; i < t_bytecount; i++) {
        tbuff[i] = cbuff[i]; cbuff[i] = 0;
    }
    reset_ir();
    // print out last captured data
    if (t_bitcount != 0) {
        // adjust last byte assuming input bits are zero
        for (i=t_bitcount; i < 7; i++)
            t_this_byte = t_this_byte << 1;
        tbuff[t_bytecount] = t_this_byte;
        t_bytecount++;
    }
    printf("(%d): Received %d bytes, %d bits.",
        cnt, t_bytecount, t_bitcount); pcrlf();
    for (i = 0; i < t_bytecount; i++) {
        printf(" Byte RX: %x", tbuff[i]); pcrlf();
    }
    cnt++;
}
}
```

← Wait for ISR to capture transmission

} Copy variables used by do\_ircap() to safe place so that IR capture interrupt can be re-enabled

← Re-enable IR capture

} Adjust if partial byte received

} Print captured bytes