Timer1 Compare Mode:

Drive a pin high or low at a precisely controlled time

Interrupt	Description	Input	Output	Conditions	Enable	Flag
Timer 1	Trigger after N events N = 1 .. 2^{19} 100ns to 0.52 sec	RC0 TMR1CS = 1 OSC/4 TMR1CS = 0	none	N = (PS)(Y) T1CON = 0x81: PS = 1 T1CON = 0x91: PS = 2 T1CON = 0xA1: PS = 4 T1CON = 0xB1: PS = 8 TMR1 = -Y	TMR1ON = 1 TMR1IE = 1 TMR1IP = 1 PEIE = 1	TMR1IF
Timer 1 Compare Mode 1	Drive a pin high or low at a precise time Interrupt when TMR1 = CCPR1	OSC/4	RC2	Interrupt when CCPR1 = TMR1 CCP1CON = 0x08: Set RC2 CCP1CON = 0x09: Clear RC2 CCP1CON = 0x0A: no change	CCP1IE = 1 TMR1ON = 1 PEIE = 1	CCP1IF
Timer 1 Compare Mode 2	Drive a pin high or low at a precise time Interrupt when TMR1 = CCPR2	OSC/4	RC1	Interrupt when CCPR2 = TMR1 CCP2CON = 0x08: Set RC1 CCP2CON = 0x09: Clear RC1 CCP2CON = 0x0A: no change	CCP12E = 1 TMR1ON = 1 PEIE = 1	CCP2IF

The PIC we use is able to measure time to 100ns. If you want to drive an output pin high or low at a precise time (accurate to 100ns), Timer1 compare interrupts are used.

There are several reasons you might want to do this:

- To output a precise frequency, the I/O pins need to be driven high/low at a precise time
- To generate a pulse with a precise duration
- To output a voltage between 0V and 5V, you can use vary the duty cycle of an I/O pin (termed pulse with modulation). Timer1 Compare allows you to adjust this duty cycle very precisely.

Output a Precise Frequency: Capture1.C

Problem: Output the note F4 (349.228Hz) on pin RC0

Solution: To generate this frequency, you need to toggle RC0 every 14317 clocks (rounded down)

$$N = \left(\frac{10{,}000{,}000}{2 \cdot 349.228 Hz}\right) = 14317.29$$

Assume Timer1 is set up with PS = 1 and TMR1 = 0. To trigger a Timer1 interrupt at time 14317, just set

CCPR1 = 14317

When TRM1 = CCPR1, the interrupt triggers and you toggle RC0. The next interrupt should be 1431 clocks later, so increment CCPR1 by 14317

CCPR1 += 14317

The next next interrupt should then be 14317 clocks after that. And so on and so on.

Essentially, each interrupt you

- Toggle RC0, and
- Set up the next interrupt 14317 clocks in the future, from the previous interrupt.

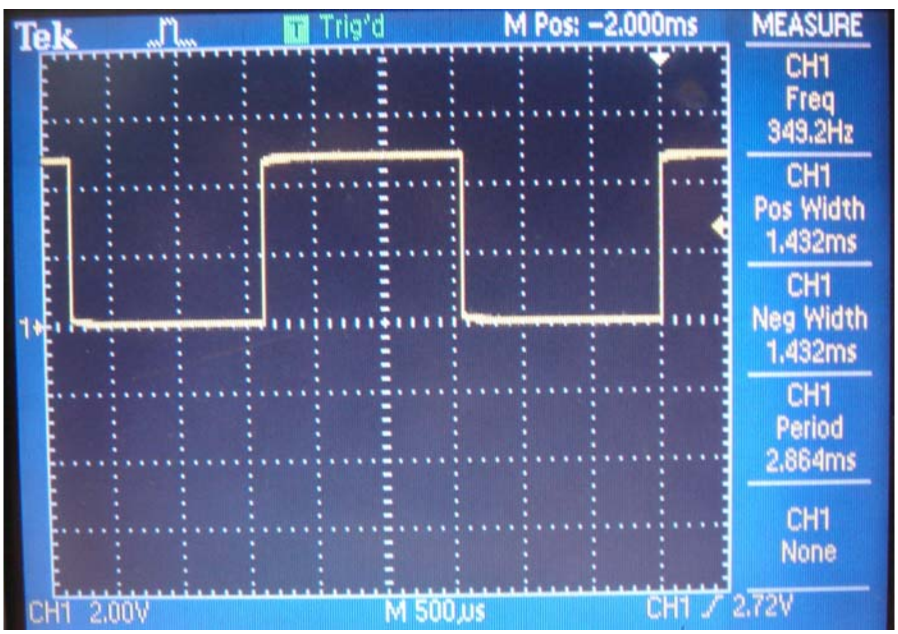
Code: Interrupt Service Routine

```
void interrupt IntServe(void)
{
   if (TMR1IF) {
      TIME = TIME + 0x10000;
      TMR1IF = 0;
      }
   if (CCP1IF) {
      RC0 = !RC0;
      CCPR1 += 14317;
      CCP1IF = 0;
      }
   }
```

Interrupt Set Up:

```
// set up Timer1 for PS = 1
   TMR1CS = 0;
   T1CON = 0x81;
   TMR1ON = 1;
   TMR1IE = 1;
   TMR1IP = 1;
   PEIE = 1;
// set up Compare for no change
   CCP1CON = 0x0A;
   CCP1IE = 1;
   PEIE = 1;


// turn on all interrupts
   GIE = 1;
```

That's pretty much it: interrupts do all the work from here on.



Compare1.C: RC0 outputs a 349.228Hz using Timer1 Compare interrupts

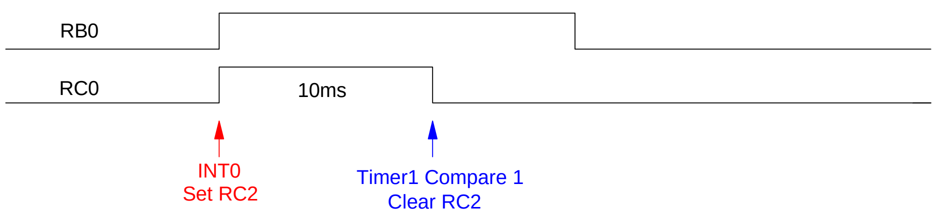
If you change the number 14317 to a variable and change it with the key pressed, you can build an 8-key piano where the notes are precise (within 1/2 clock).

Generating a Precise Pulse Width

Problem: Every time you press RB0, output a pulse that is precisely 10ms long.

Solution: Use two different interrupts:

- INT0 records the time that the button was pressed (time of rising edge on RB0). RC0 is set at that time.
- Compare1 kicks in 10ms later. At that time, RC0 is cleared.



Note that 10ms is equal to 100,000 clocks - more than TMR1 can count up to (max = 65,536: it's a 16-bit counter.) To bring this in range, use a pre-scalar of 8 for TMR1. With this

10ms = 100,000 / 8 = 12,500

Compare1 interrupt should kick in 12,500 counts on TMR1 after the INT interrupt:

Code: Interrupt Service Routine:

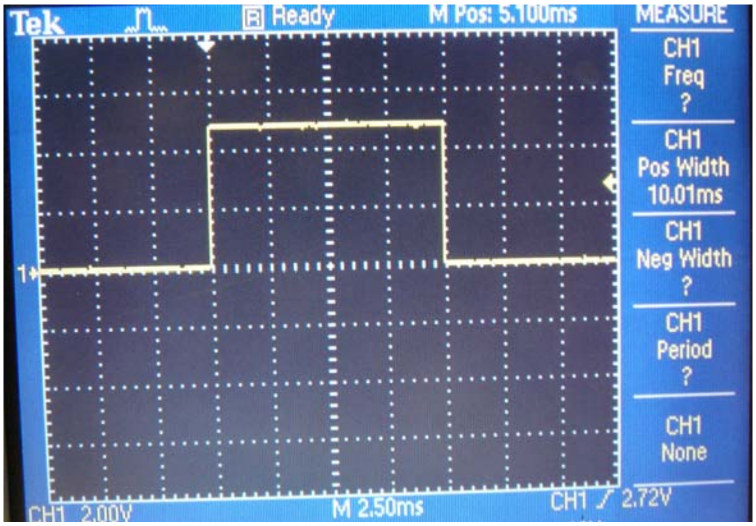
```
// Interrupt Service Routine

void interrupt IntServe(void)
{
   if (INT0IF) {
      RC0 = 1;
      CCPR1 = TMR1 + 12500;    // 10ms with PS = 8
      INT0IF = 0;
      }
   if (TMR1IF) {
      TIME = TIME + 0x10000;
      TMR1IF = 0;
      }
   if (CCP1IF) {
      RC0 = 0;
      CCP1IF = 0;
      }
   }
```

Interrupt Set Up:

```
// set up INT0 for rising edge
   INTEDG0 = 1;
   INT0IE = 1;
```

```
        TRISB0 = 1;
     // set up Timer1 for PS = 8
        TMR1CS = 0;
        T1CON = 0xB1;
        TMR1ON = 1;
        TMR1IE = 1;
        TMR1IP = 1;
        PEIE = 1;
     // set up Timer1 Compare for no change
        CCP1CON = 0x0A;
        CCP1IE = 1;
        PEIE = 1;
```

Resulting Signal on RC0:



Compare2.c: A 10ms pulse is generated every time you press RB0

Note that you could also use pin RC2, where the Timer1 Compare interrupt automatically clears RC2 when TMR1 = CCPR1.

Example 3: Pulse Width Modulation (PWM.C)

A third use of Timer1 Compare is to generate a pulse width modulation with a precise duty cycle. The idea is this:

- Set up Timer1 to run with a pre-scalar of 1.
- Every Timer1 interrupt you set RC0 (TMR1 = 0x0000)
- When Timer1 = CCPR1, clear RC0

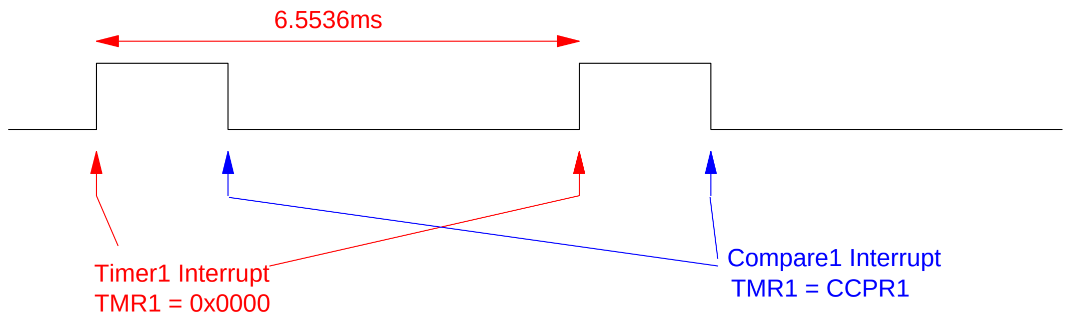
By adjusting CCPR1 from 0x0001 to 0xFFFF, you can change how long RC0 is on from

- 1 clock out of a period of 65,536 clocks (0.0015%), to
- 65535 clocks out of a period of 65,536 clocks (99.998%),
- With 65,536 steps between 0% and 100% on

This is called pulse width modulation.

Actually, you can't quite get to 0% or 100%. It takes about 50 clocks to call an interrupt. The min and max duty cycle is then

- min: 50 / 65,536 (0.076%)
- max: 65,486 / 65,536 (99.924%)



To generate a waveform which in 1.00V on averatge, we want a duty cycle of 20%

$$PWM = \left(\frac{1V}{5V}\right) = 20\%$$

CCPR1 should then b 25% of its maximum value:

$$CCPR1 = 0.25 \cdot 65,536 = 13,107$$

Code: PWM.C

Interrupt Service Routine:

```
// Global Variables
unsigned long int TIME;

// Interrupt Service Routine

void interrupt IntServe(void)
{
   if (TMR1IF) {
      RC0 = 1;
      TIME = TIME + 0x10000;
      TMR1IF = 0;
      }
   if (CCP1IF) {
      RC0 = 0;
      CCP1IF = 0;
      }
   }
```

Interrupt Set-Up:

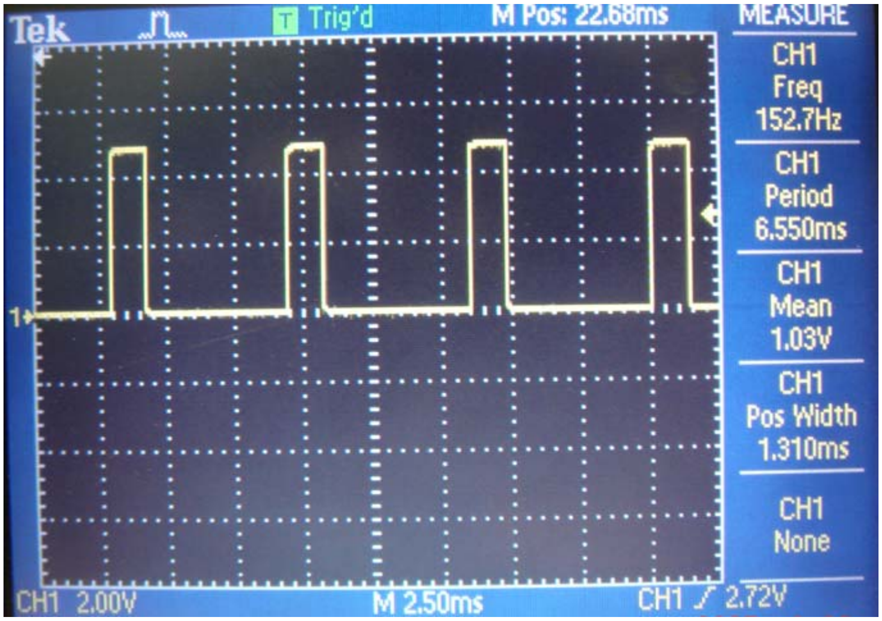
```
// set up Timer1 for PS = 1
   TMR1CS = 0;
   T1CON = 0x81;
   TMR1ON = 1;
   TMR1IE = 1;
   TMR1IP = 1;
   PEIE = 1;
// set up Timer1 Compare for no change
   CCP1CON = 0x0A;
   CCP1IE = 1;
   PEIE = 1;
```

Main Routine:

```
CCPR1 = 13107;    // 20% duty cycle

while(1) {
   LCD_Move(0,0);  LCD_Out(TIME + TMR1, 7);
   }
```

If you change CCPR1, it changes the duty cycle as

$$\%ON = \left(\frac{CCPR1}{65{,}536}\right) \cdot 100\%$$



PWM.C: A 20% duty cycle square wave is output on RC0, resulting in its average voltage being 1.000 (20% of 5.00V)