Timer2 Interrupts

Background:

The execution time for routines sometimes needs to be set. This chapter loops at several ways to set the sampling rate.

Example: Write a routine which increments an 8-bit counter every 10 ms and sends this to PORTC

Assembler Solution	C Solution

`org 0x800`
`goto Start`
 	
`;*********************************`
`; 10ms wait routine`
`;*********************************`
 	
`Wait movlw 60`
`movwf CNT2`
 	
`Loop1 movlw 138`
`movwf CNT3`
 	
`Loop2 nop`
`nop`
`nop`
`decfsz CNT3`
`goto Loop2`
 	
`decfsz CNT2`
`goto Loop1`
`return`
 	
`;*********************************`
`; Main Routine`
`;*********************************`
 	
`Start bcf STATUS,RP1`
`bsf STATUS,RP0`
`clrf TRISC`
`bcf STATUS,RP0`
`clrf COUTNER`
 	
`Main incf COUNTER,F`
`movf COUNTER,W`
`movwf PORTC`
`call Wait`
`goto Main`
	`// Subroutine Definitions`
	
	`void Wait(void)`
	`{`
	`unsigned int X;`
	`for (X=0; X<5000; X++) {}`
	`}`
	
	`// Main Routine`
	
	`void Main(void)`
	`{`
	`unsigned char COUNTER;`
	
	`COUNTER = 0;`
	`TRISC = 0;`
	
	`do {`
	`COUNTER += 1;`
	`PORTC = COUNTER;`
	`Wait();`
	`} while (1>0);`
	`}`

In assembler, you can compute the exact time the wait routine takes:

$$\#clocks = ((6 * 138 + 5) * 60 + 5) = 49,985$$

$$time = (49,985 \text{ clocks})\left(\frac{100ns}{clock}\right) = 4.9985ms$$

In C, you aren't sure exactly how long the routine takes since you don't know exactly how the code compiles. Trial and error can be used to tweek the number 5,000

Note that

- The timing is slightly off. This clock will be off by 26 seconds per day
- The routine is rather inefficient. 99.95% of the time is spent in a wait loop. Only 0.05% of the time is spent in the main routine.
- This is much less accurate in C since you aren't sure exactly how your C code compiles.

TIMER Interrupts

One way to improve the efficiency of this program is to use interrupts. Interrupts are similar to subroutines except that

- Subroutines are routines called by software (such as the 10ms wait loop from before)
- Interrupts are routines called by hardware (such as a certain time elapses)

Timers are useful, so four are available on the PIC18F4626:

- TIMER0: Interrupt after N events (or N clocks). N = 1 to 2^{24} (1.67 seconds)
- TIMER1: Interrupt after N events (or N clocks). N = 1 to 2^{19} (52 milliseconds)
- TIMER2: Interrupt every N clocks. N = 1 to 2^{16} (6.5 millisecond)
- TIMER3: Interrupt after N events (or N clocks). N = 1 to 2^{19} (52 milliseconds)

Default for the PIC is to disable interrupts. You must set up the interrupt (enable and conditions for the interrupt) if you want to use them.

If an interrupt occurs,

- The present instruction is completed
- The processor inserts a *call 0x08* into the program

At address 0x08, the interrupt service routine must be placed (or a *goto InteruptService* needs to be placed. This routine must

- Save the W and STATUS register. Since you don't know where in the program the interrupt will be called, W and STATUS may be important.
- Clear TMR2IF. This tells the PIC that the present interrupt has been serviced. If you don't, the interrupt will be called immediately upon return, essentially halting the processor.
- (optional) Do something
- Restore the W and STATUS registers, and
- Terminate with *retfie* for a return from interrupt.

TIMER2 INITIALIZATION

Suppose you'd like to keep track of time. To do this, set up the interrupt so that it's called every 1 millisecond (10,000 clocks: 10000 * 100ns = 1ms).

For every interrupt you want to use, you need to initialize them by:

- Enable the interrupt
- Set up the conditions for the interrupt (10,000 clocks)

In the interrupt service routine (which is called every 10,000 clocks in this case), you need to

- Do something (such as increment a counter, which is how the main program keeps track of time),
- Set up the next interrupt (10,000 clocks from now), and
- Acknowledge the interrupt (clear the interrupt flag)

The hoops you have to jump through for TIMER0 to TIMER3 are summarized in the following table:

	Clock Source	N	Enable Bits	Flag
TIMER2	N = 1..65,535 200ns to 13.1ms N = A*B*C A = 1..16 B = 1..256 C = 1, 4, 16	N = A*B*C PR2 = B-1 T2CON = xaaaa1cc aaaa = 0000: A=1 aaaa = 0001: A=2 ::: aaaa = 1110: A=15 aaaa = 1111: A=16 cc = 00: C = 1 cc = 01: C = 4 cc = 10: C = 16 cc = 11: C = 16	TMR2ON = 1 TMR2IE = 1 TMR2IP = 1 PEIE = 1	TMR2IF

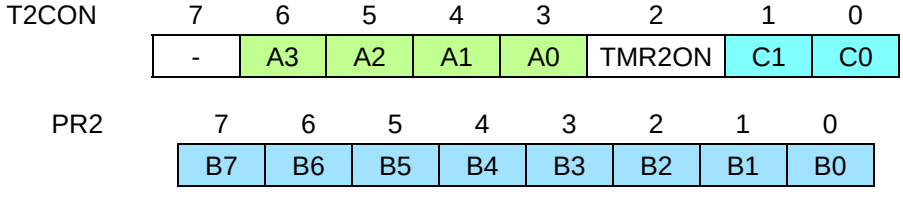
To enable a Timer2 interupt, you need to turn it on four times

- TMR2ON = 1; 0 turns off Timer2 and interrupts won't happen
- TMR2IE = 1; 0 disables interrupts
- PEIE = 1; 0 disables several interrupts, including TIMER2
- GIE = 1; 0 disables all interrupts
- TMR2IP = 1; Timer2 is a high priority interrupt

The rate at which the TIMER2 interrupts happen is

$$Time = (A \cdot B \cdot C) \cdot 100ns$$

A, B, and C are defined by registers T2CON and PR2:



The scalar values are

PostScalar A		Main Scalar B		Prescalar C	
A3:A2:A1:A0	A	B7:B0	B	C1:C0	C
0000	1	0000 0000	1	00	1
0001	2	0000 0001	2	01	4
				10	16
1110	15	1111 1110	255	11	16
1111	16	1111 1111	256		

The maximum time you can set for a Timer2 interrupt is

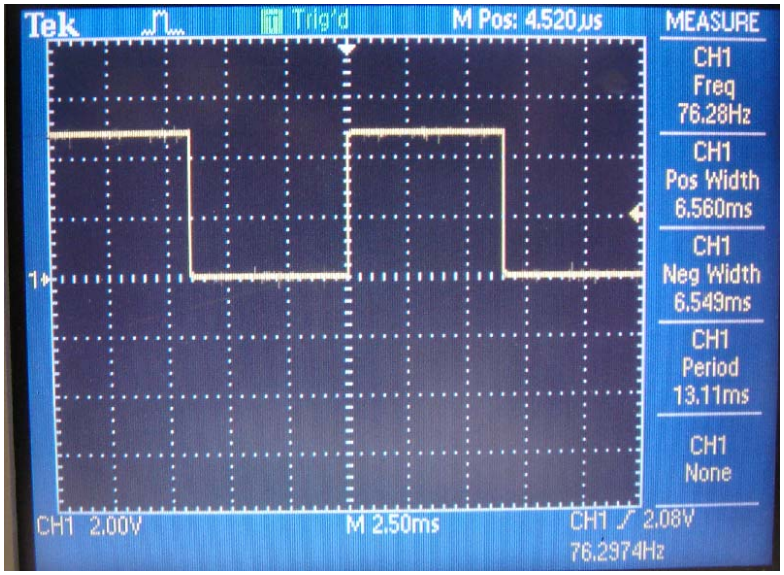
A x B x C = (16) x (256) x (16) = 65,536 clocks

= 6.5536 ms

Example: Toggle RC0 every 6.5536 ms (65,536 clocks)

```
#include <pic18.h>

// Global Variables

// Subroutine Declarations

void interrupt timer2(void)
{
   RC0 = !RC0;
   TMR2IF  = 0;
   }


void main(void)
{
   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   TRISD = 0;
   ADCON1 = 15;

// initialize Timer2

   T2CON   = 0xFF;
   PR2     = 255;
   TMR2IE  = 1;
   PEIE    = 1;
   TMR2ON  = 1;
   TMR2IP  = 1;

// Turn on all interrupts

   GIE = 1;

   while(1) {
      PORTB = PORTB + 1;      // Count for no reason other than to count
      }
}
```

The resulting signal on RC0 is as follows:



RC0 Toggles Every Timer2 Interrupt: Timer2 Set Up for N = 65,536 Clocks (6.5536 ms)

For a 1ms interrupt rate,

$$A \cdot B \cdot C = (1ms) \cdot (10,000,000 \text{ clock/second})$$

A x B x C = 10,000

One combination which works is

- C = 01 (x 4)
- B = 249 (x 250)
- A = 9 (x 10)

or

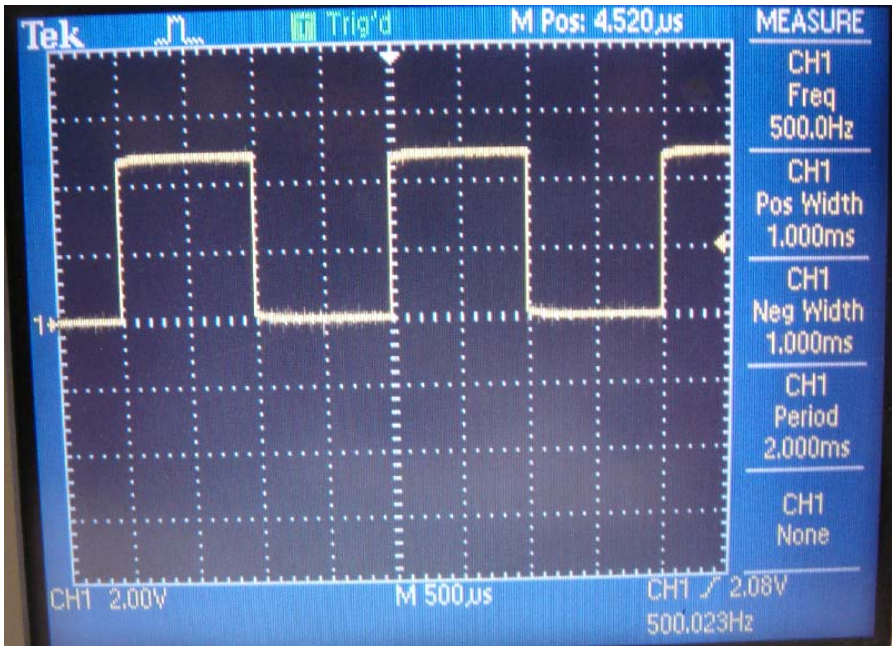
- PR2 = 249
- T2CON = 0x4D

T2CON	7	6	5	4	3	2	1	0
	-	A3	A2	A1	A0	T2E	C1	C0
(A=9, C=1)	0	1	0	0	1	1	0	1

Example: Toggle RC0 every 1.00ms

```
#include <pic18.h>

// Global Variables

// Subroutine Declarations

void interrupt timer2(void)
{
   RC0 = !RC0;
   TMR2IF  = 0;
```

```
    }


void main(void)
{
   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   TRISD = 0;
   ADCON1 = 15;

// initialize Timer2

   T2CON   = 0x4D;
   PR2     = 249;
   TMR2IE  = 1;
   PEIE    = 1;
   TMR2ON  = 1;
   TMR2IP  = 1;

// Turn on all interrupts

   GIE = 1;

   while(1) {
      PORTB = PORTB + 1;      // Count for no reason other than to count
      }
}
```

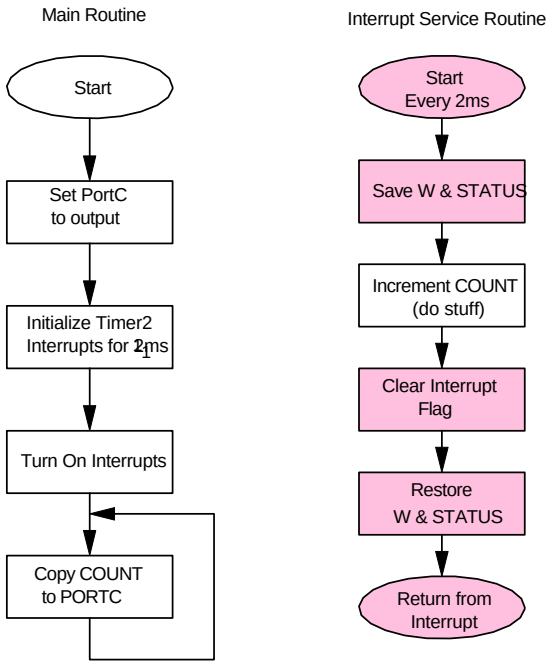


RC0 Toggles Every Timer2 Interrupt: Timer2 Set Up for N = 10,000 Clocks (1.000 ms)

The flow chart for this program is a little difficult to draw since two routines are running in parallel:

- The main routine which sends COUNTER to PORTC and repeats
- The Timer2 routine which increments COUNTER every 1ms.

Two parallel flow charts may be the best way to represent this:



Note that

- The main routine simply watches COUNTER and sends it to PORTC.
- The Interrupt routine is responsible for changing COUNTER every 1ms

Moroover:

- The shaded parts of the interupt routine are common to any interrupt service routine.
- Only the conditions under which this routine is called (set up in the main routine) and what you do when it is called change.

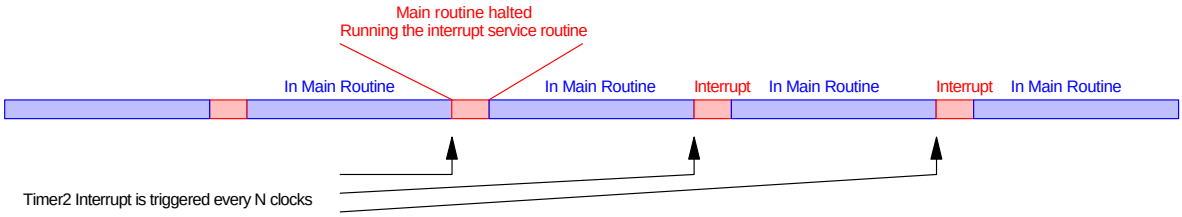
Interrupt Constraints

Background:

Timer2 interrupts are a way to keep track of time.

- The PIC is running at 10 million instructions / second (10MHz)
- Every N clocks, a Timer2 interrupt is triggered
  - N = A * B * C
  - A, C are from T2CON
  - B is from PR2
- When the interrupt is triggered, the main routine stops and you run the interrupt service routine.

If you plot time on the X axis, the processor is then running as follows:



Once set up, the main routine has no control over the timing: hardware triggers the interrupt every N clocks. Note however, that the interrupt service routine is stealing clocks from the main routine. You can't steal more than 100%. You probably don't want to steal more than 50%.

With interrupts turned on, you essentially have 2 (or more) programs running in parallel:

- The main routine which supposedly does stuff,
- The interrupt routines which handle administration every time an event occurs.

This creates several possible constraints:

Timing Constraints:

This interrupt service routine takes about 50 clocks to execute. This is much more efficient than the wait loop which wastes 10,000 clocks to wait 1ms. Instead, the interrupt 'steals' 50 clocks from the main routine every 10,000 clocks.

If you call the interrupt more often (because you want a timer with a better resolution than 1ms) this slows down the processor. If you go too fast, the main routine shuts down and all the time is spent in the interrupt service routine:

Interrupt Time (clock resolution)	Clocks / Interrupt (N)	# Clocks Spent in the Interrupt	# Clocks Left for the Main Routine	Processor 'Speed'
1ms	10,000	50	9,950	99.5%
100 us	1,000	50	950	95%
10 us	100	50	50	50%
1 us	10	50	-50	0%

The faster you interrupt, the more accurate your clock. The faster you interrupt, however, the slower the main routine appears to run. Moreover, if you interrupt too frequently, you spend all of the time servicing the interrupt and never get to the main routine.

This results in the following rule:

- **Keep interrupt service routines short. The interrupt service routine should take much less time to execute than the rate at which it is called.**
- **With Timer2 interrupts, you can keep track of time to 100us (slowing down the main routine by 50%)**.

There are ways to measure time to 100ns - but this comes up later with Timer1 interrupts.