

First Experiments Part2

1. "Using Timer"
2. "Using Interrupts "





Contents

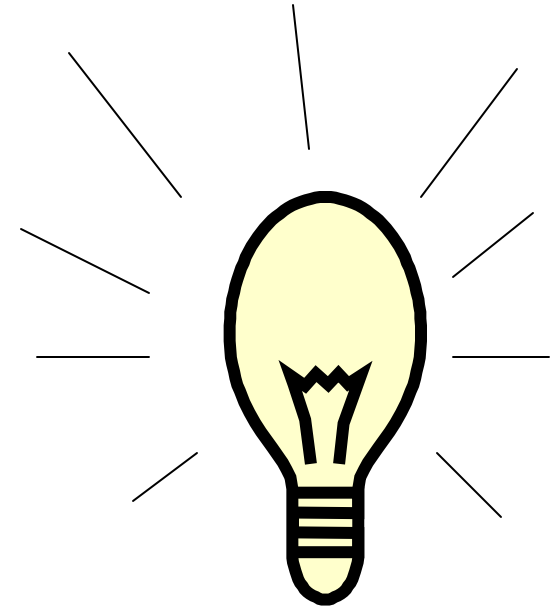
- Directions and descriptions for beginner 16 bit Experimenter starting experiments
 - Each BLINK experiment builds on another
- Recommend review of “Quick Start Guide” (available on www.kibacorp.com) to better understand tools installation
- Recommend review of “First Experiments Part 1” (available on www.kibacorp.com)





Experiment #2 Using Timer

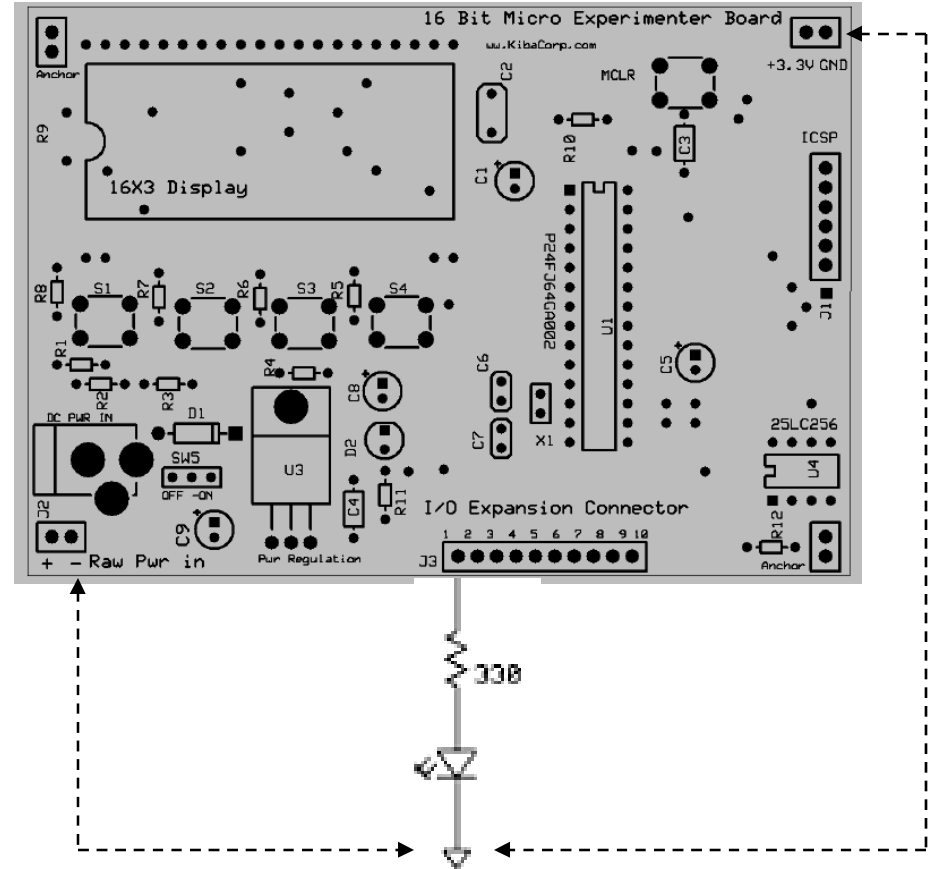
- This experiment will highlight timer capabilities of the PIC24F
- It use an internal timer peripheral to create delay necessary for toggle led on/off.
- Hardware configuration same an Blink, and Hello World covered in Part 1





“Using Timer” Hardware

- Same as used in part 1
 - Connect an LED with 330 ohm resistor to I/O Expansion Pin 1
 - Any I/O pin can sink/source up to 25 ma --the resistor is used to limited current
 - The Expansion I/O pin 1 Port B pin 2 or RB2 is configured for output.



Alternative Ground Connections



Why Timers?

- The Timer is a hardware counter to expedite counting and delay operation that would be time consuming and inefficient if done in software.
- The Timer once set up performs its function with little to no oversight by the CPU, and only alerts the CPU once an overflow event has occurred.
- Timers in the PIC can be set up to count external signals or source with internal clock. Rates and initial count settings are all configurable



PIC24F Timer features

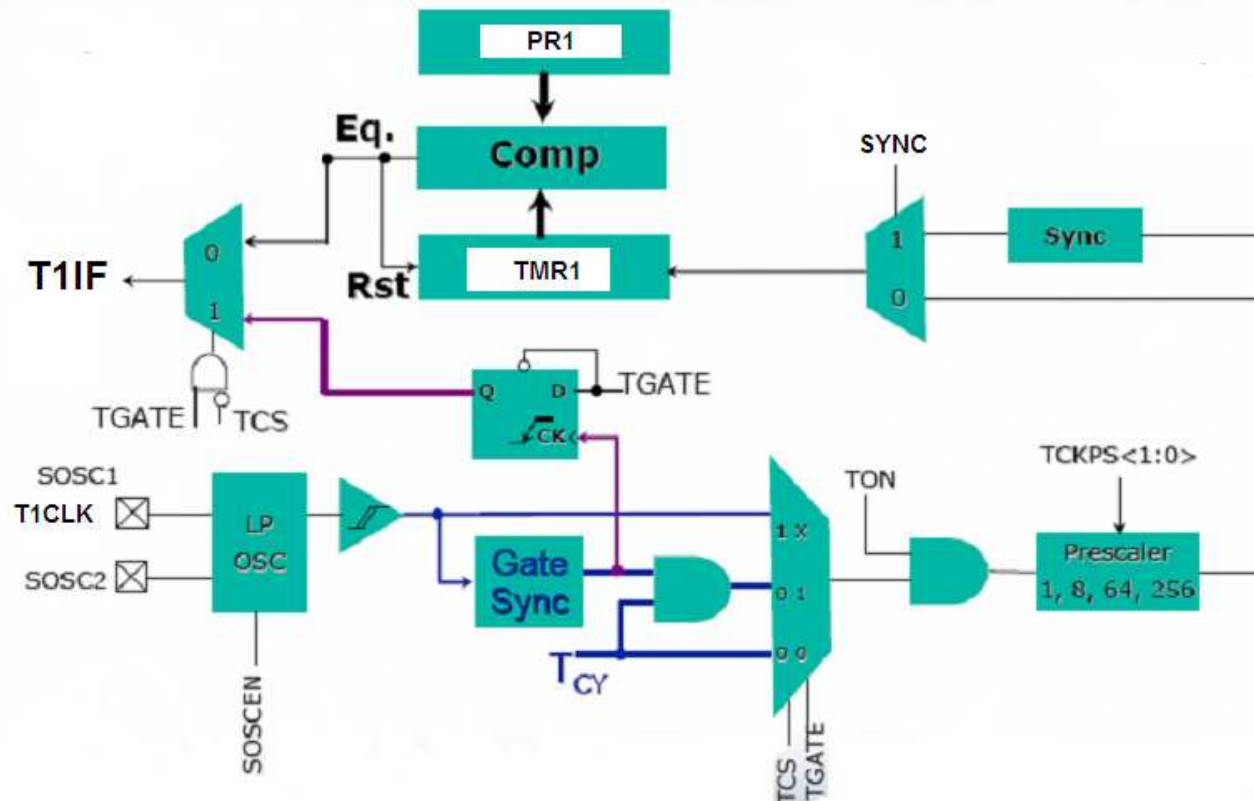
- **Five 16-bit General Purpose Timers/Counters**
 - Similar functionality between all 5 timers
 - Asynchronous counter feature only in Timer1
- **Period Registers for Each**
 - Interrupt generation on match
 - Reset on match
- **Gated Timer Operation on Each**
 - Interrupt on falling edge of gate
- **Four of these timers (Timer 2+3 and 4+5) can make two 32-bit timers/counters**



KibaCorp

Timer 1

- Special Function Register (SFR) TMR1 contains the 16 bit value
- SFR T1CON controls activation and operating mode of timer
 - TGATE, TCS, TCPS0, TCPS1, TSYNC, TON
- SFR PR1 which can be used to produce a periodic reset of timer (not used here)
- When Timer1 rolls over from FFFF to 0000 the T1IF flag is set
- TON bit in T1CON activated Timer





Timer1 (look at page 150 of Microchip PIC24F datasheet)

REGISTER 10-1: T1CON: TIMER1 CONTROL REGISTER⁽¹⁾

R/W-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
TON	—	TSIDL	—	—	—	—	—
bit 15							bit 8

U-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	U-0
—	TGATE	TCKPS1	TCKPS0	—	TSYNC	TCS	—
bit 7							bit 0

Legend:							
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'					
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown				

T1CON = 0xb1000000001100

Or

T1CON = 0x8030

This sets

Timer1 activated TCON = 1

Main MCU Clock is source TCS=0 Fosc/2 or 16MHz

Prescaler TCPS = 11 for 1:256

Gating and synchronization not required TGATE = 0, TSYNC = 0

Do not worry about behavior in idle mode TSIDL = 0

- bit 15 **TON:** Timer1 On bit
1 = Starts 16-bit Timer1
0 = Stops 16-bit Timer1
- bit 14 **Unimplemented:** Read as '0'
- bit 13 **TSIDL:** Stop in Idle Mode bit
1 = Discontinue module operation when device enters Idle mode
0 = Continue module operation in Idle mode
- bit 12-7 **Unimplemented:** Read as '0'
- bit 6 **TGATE:** Timer1 Gated Time Accumulation Enable bit
When TCS = 1:
This bit is ignored.
When TCS = 0:
1 = Gated time accumulation enabled
0 = Gated time accumulation disabled
- bit 5-4 **TCKPS1:TCKPS0:** Timer1 Input Clock Prescale Select bits
11 = 1:256
10 = 1:64
01 = 1:8
00 = 1:1
- bit 3 **Unimplemented:** Read as '0'
- bit 2 **TSYNC:** Timer1 External Clock Input Synchronization Select bit
When TCS = 1:
1 = Synchronize external clock input
0 = Do not synchronize external clock input
When TCS = 0:
This bit is ignored.
- bit 1 **TCS:** Timer1 Clock Source Select bit
1 = External clock from T1CK pin (on the rising edge)
0 = Internal clock (FOSC/2)
- bit 0 **Unimplemented:** Read as '0'



Examining “Using Timer” program

- Open timer.mcp
- Examine Source code timer.c
- The initialization , which includes both the device peripherals initialization and variables initialization, executed only one at the beginning
- Timer1 runs continuously in background using system internal clock.
- A software loop blinks the led checking status timer1 against constant delay value for timing interval.



Using Timer Code

```
timer.c*
1 // First Experimenter part 2 using Timer
2 // A Timer based delay Loop in the pattern
3 //
4 //modified thk 3/26/09
5
6 #include "p24FJ64ga002.h"
7
8
9 //config fues settings
10 _CONFIG2(0xF9C7); //demo
11 //default primary osc disabled, IOLOCK may be changed via unlock sequence, OSC pin has digital I/O function, clock switching and monitor disabled
12 //Fast Internal Oscillator enabled with PLL
13 _CONFIG1(0x3F5F); //demo default
14 //Watchdog disabled, ICSP cahnnel PGCq1/PGD1, code protect and JTAG idsabled.
15
16 #define DELAY 16000
17
18 main()
19 {
20     TRISBbits.TRISB2 = 0; // PORTB RB2 bit as output
21     T1CON = 0x8030; // TMR1 on, prescale 1:256 Tclk/2
22
23     while( 1)
24     {
25
26         PORTEbits.RB2 =1;
27         TMR1 = 0; //clear timer1
28         while ( TMR1 < DELAY)
29         {
30         }
31
32         PORTEbits.RB2 =0;
33         TMR1 = 0; //clear Timer1
34         while ( TMR1 < DELAY)
35         {
36         }
37     } // main loop
38 } // main
```



Where is the Timer Delay?

```
#define DELAY 16000
```

```
TMR1 = 0;
```

```
//clear timer1
```

```
while ( TMR1 < DELAY)
```

```
{ // wait on timer reaching  
    delay
```

```
}
```



Using Timer Experiment

1. Wire led/resistor as shown to pin 1 of experimenter
2. Hook up experimenter to PICKIT2
3. Open project Timer.mcp
4. Build code/download to PICKIT2 as programmer
5. Verify LED is blinking.
6. Examine source code
7. Reconfigure MPLAB for debugger
Hookup and select PICKIT2 rebuild/program –watch, break, single step
8. Additional challenge –configure MPLAB Debugger for simulator –use simulator stop watch (with breakpoints) to measure timer1 interval



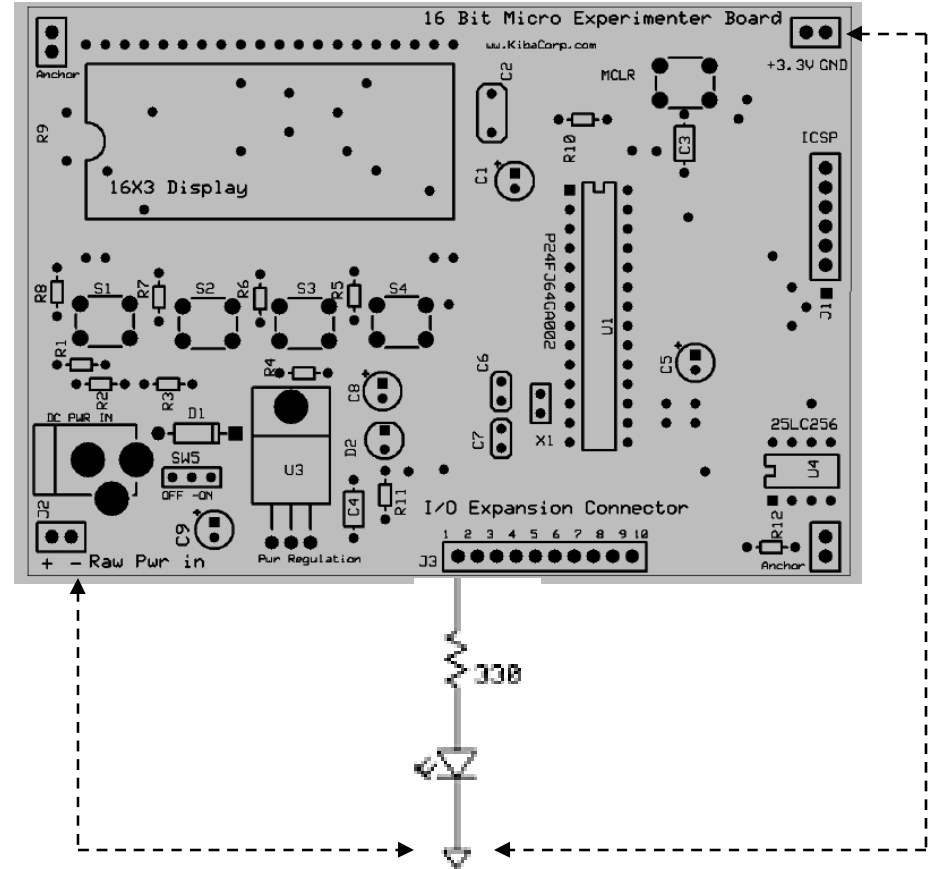
Using Interrupts

Rework Blink with Timer1
Interrupts



“Using Interrupt” Hardware

- Same as used in part 1
 - Connect an LED with 330 ohm resistor to I/O Expansion Pin 1
 - Any I/O pin can sink/source up to 25 ma --the resistor is used to limited current
 - The Expansion I/O pin 1 Port B pin 2 or RB2 is configured for output.



Alternative Ground Connections



In general

- It is up to the user to decide when to use interrupts.
- For reasons of efficiency, size, and ultimately cost, in the embedded-control world the smallest applications, which happen to be implemented in the highest volumes, most often cannot afford the “luxury” of a multitasking operating system and use the interrupt mechanism instead to “divide their attention” on the many tasks at hand.
- By deciding not to use interrupts the user may run the risk of missing response to critical outside events.
 - I.e. Air-bag, Anti-lock breaking, speed control
- Interrupt increase the response time of the processor to more “real-time” versus the alternative of “polling” peripherals.
- In interrupt the peripheral tells the processor when it needs service immediately



Interrupt Overview

- An interrupt is an internal or external event that requires quick attention from the CPU
- PIC interrupts have guaranteed latency time for response
- The PIC24 architecture provides a rich interrupt system and can manage as many as 118 distinct sources of interrupt
- Each PIC24F interrupt source has a unique piece of code called the Interrupt Service Routine (ISR) directly associated with via a pointer or “vector” to provide the required response action
- The PIC24F minimizes interrupt latency (the time the CPU requires to respond to interrupt-defined as the time between the triggering event and the execution of the first instruction of the ISR.
 - Only three instructions for internal interrupts and four for external interrupts



Interrupts

Interrupts are events that cause your program to stop what it is doing in order to run an Interrupt Service Routine which will handle the event by taking whatever action is required before finally returning control to your main program.

- PIC24 interrupts are vectored
- Interrupts require special functions to service the events that cause them:
 - ISRs must not have any parameters
 - ISRs must not be called by the main code
 - ISRs should not call other functions



MPLAB C Compiler

- Helps manage the complexity of the interrupt system by providing a few language extension
- The PIC24 keeps all the interrupts vectors in one large Interrupt Vector Table (IVT)
- Compiler automatically associates interrupt vector with 'special' user defined C function as long as a few limitation are kept in consideration
 - Do not use any return types (use type void)
 - No parameter can be passed to the function (use parameter void)
 - They cannot be called directly by other functions
 - Ideally they should not call any other functions



PIC24F Interrupt External Sources

- 5x External Pins with level trigger detection
- 22x External pins connected to Change Notification Module
- 5x Capture Modules
- 5x Output Compare Modules
- 2x serial port interfaces (UARTs)
- 4x Synchronous serial ports (SPI and I2C)
- Parallel Master Port



PIC24F Interrupt Internal Sources

- 5x 16 bit Timers
- 1 x Analog-to Digital Converter
- 1x Analog Comparators Module
- 1x real-time Clock and Calendar
- 1x CRC Generator



PIC24F Traps

- Eight additional vectors occupy the first locations on top of the IVT table. They are used to capture special error conditions such as failure of the selected CPU oscillator, incorrect address, stack underflow, divide by zero
 - Oscillator Failure Trap (level 14)
 - Address Error Trap (level 13)
 - Instruction fetch from illegal program space
 - Data fetch from unimplemented data space
 - Unaligned word access from data space
 - Stack Error Trap (level 12)
 - Stack overflow or underflow
 - Math Error Trap (level 11)
 - Divide by Zero
 - Unsaturated Accumulator Overflow (A or B)
 - Catastrophic Accumulator Overflow (either)
 - Accumulator Shift Overflow



PIC24F Interrupt Functions

Partial List of Interrupt Function Names

<u>IRQ #</u>	<u>Primary Name</u>	<u>IRQ #</u>	<u>Primary Name</u>
N/A	_ReservedTrap0	4	_IC2Interrupt
N/A	_OscillatorFail	5	_OC2Interrupt
N/A	_AddressError	6	_T2Interrupt
N/A	_StackError	7	_T3Interrupt
N/A	_MathError	8	_SPI1Interrupt
N/A	_ReservedTrap5	9	_U1RXInterrupt
N/A	_ReservedTrap6	10	_U1TXInterrupt
N/A	_ReservedTrap7	11	_ADCInterrupt
0	_INT0Interrupt	12	_NVMInterrupt
1	_IC1Interrupt	13	_SI2CInterrupt
2	_OC1Interrupt	14	_MI2CInterrupt
3	_T1Interrupt	15	_CNInterrupt



Interrupts

Interrupt Function Names

- Interrupt Function names may be found in:
 - Device's Linker Script (e.g. p24f64ga002.gld)
 - MPLAB[®] C30 User's Guide (Section 7.4)
 - MPLAB[®] C30 Online Help
- Used by LINK30 to associate interrupt function with the appropriate location in the interrupt vector table
- Linker puts the address of the interrupt function in the appropriate location in the interrupt vector table



Interrupts

How to Declare an Interrupt Service Routine

```
void __attribute__((interrupt , no_auto_psv)) _ISRName(void)
{ ...           Function Code Here           ... }
```

- No parameters and `void` return type (required)
- Use pre-defined name (required)
- Do NOT call from main line code (required)
- Do not call other functions (*recommended*)
- **Example given for external interrupt INT0**

```
void __attribute__((interrupt , no_auto_psv)) _INT0Interrupt(void)
{
    //Ordinary C code goes here to handle interrupt
    _INT0IF =0;
}
```



Timer1 Interrupt

- Timer1 interrupt service routine
- Invoked when Timer overflows from all FFFF to 0000.
- Sets T1IF flag –must be cleared by user if other Timer1 interrupts are to occur.

```
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void)

{
    // user code here
    // clear the interrupt flag
    _T1IF = 0;
} //T1Interrupt
```



Considerations in writing ISR

- You must clear the interrupt flag manually
- You must save any registers/variables you access in the ISR's code if they are not handled automatically by the compiler
 - Save them manually in your ISR code
- Variables modified by an interrupt should be tagged with the `volatile` keyword



FYI--Interrupt Priority

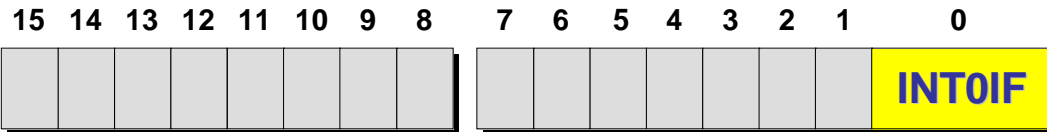
- CPU has 16 priority levels
 - Level 0 is the default CPU level (**main**)
 - Level 1 - 7 for user interrupts
 - Level 8 -15 reserved for traps (**NMI**)
- An interrupt source must have a user-assigned priority level greater than current CPU priority level (IPL<3:0>) to initiate an exception process. Level 4 is the default for all sources
- Natural order priority (IVT order) is used to resolve conflicts between simultaneous, pending interrupt sources within a given user-assigned priority level
- ***We use the defaults!***



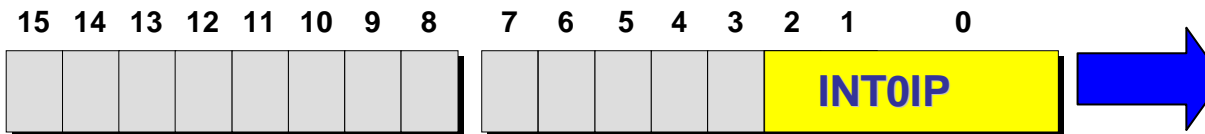
SFR to Configure Interrupts

example INTO

IFS0: Interrupt Flag Control Register 0 : Clear Interrupt in the ISR

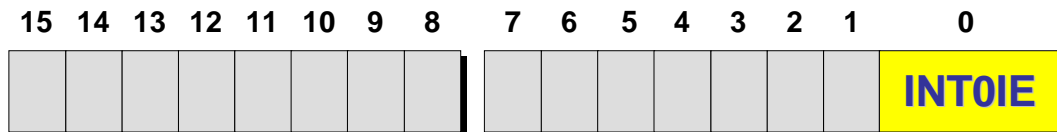


IPC0: Interrupt Priority Control Register 0 : Assign Priority



- 000: Disabled
- 001: Priority 1 Interrupt
- 010: Priority 2 Interrupt
- 011: Priority 3 Interrupt
- 100: Priority 4 Interrupt
- 101: Priority 5 Interrupt
- 110: Priority 6 Interrupt
- 111: Priority 7 Interrupt

IEC0: Interrupt Enable Control Register 0 : Enable Interrupt





Examining “Using Interrupt” program

- Open interrupt.mcp
- Examine Source code interrupt.c
- The initialization , which includes both the device peripherals initialization and variables initialization, executed only one at the beginning
- Timer1 interrupts are enabled
 - Verify SFR setup operations
- Timer1 ISR runs continuously in background using system internal clock and blinks the led clearing T1IF flag for next timing interval.
- Main code still in a do nothing loop –awaiting interrupts



Using Interrupt Source

```
Interrupts.c
14
15     volatile int flag =0;
16     // 1. Timer1 interrupt service routine
17     void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void)
18     {
19     {
20
21     if ( flag ==0 )
22     {LATBbits.LATB2 =1;
23
24     }
25     else
26     {
27     LATBbits.LATB2=0;
28
29     }
30     flag =flag ^ 1;          // Toggle flag
31
32     // 1.2 clear the interrupt flag
33     _T1IF = 0;
34
35     } //T1Interrupt
36
37     main()
38     {
39     // 2. init Timer 1, T1ON, 1:1 prescaler, internal clock source
40     // this is the default value anyway
41     TMR1 = 0;          // clear the timer
42     PR1 = 0x7fff;    // set the period register
43     TRISBbits.TRISB2 = 0; // PORTB bit 2 RB2 as output 4
44     PORTBbits.RB2 =0;
45     // 2.1 configure Timer1 module
46     T1CON = 0x8030; // TMR1 on, prescale 1:256 Tclk/2
47
48     // 2.2 init the Timer 1 Interrupt, clear the flag, enable the source
49     _T1IF = 0;
50     _T1IE = 1;
51
52
53
54     // 3. main loop
55     while( 1)
56     {
57     } // main loop
58
59     } // main
60
```



Using Interrupt Experiment

1. Wire led/resistor as shown to pin 1 of experimenter
2. Hook up experimenter to PICKIT2
3. Open project interrupt.mcp
4. Build code/download to PICKIT2 as programmer
5. Verify LED is blinking.
6. Examine source code
7. Reconfigure MPLAB for debugger
Hookup and select PICKIT2 rebuild/program –watch,
break, single step