Polled IO versus Interrupt Driven IO

- Polled Input/Output (IO) – processor continually checks IO device to see if it is ready for data transfer
  - Inefficient, processor wastes time checking for ready condition
  - Either checks too often or not often enough

- Interrupt Driven IO – IO device interrupts processor when it is ready for data transfer
  - Processor can be doing other tasks while waiting for last data transfer to complete – very efficient.
  - All IO in modern computers is interrupt driven.
PIC24 μC Interrupt Operation

The normal program flow (main) is referred to as the foreground code. The **interrupt service routine** (ISR) is referred to as the background code.

---

**Normal Program Flow**

```c
main() {
    instr1
    instr2
    instr3
    ...
    instrN
    Interrupt occurs at instrN (which completes)
    ...
    instrN+1
    instrN+2
    ...
    ...
}

ISR called by interrupt generation logic; main() code *does not* call ISR explicitly.
```

**Interrupt Service Routine (ISR)**

1. Status (lower byte), CPU priority level, and return address saved on stack.
2. CPU priority level set to priority of pending interrupt, thus masking interrupts of same or lower priority.
3. PC set to interrupt vector.

**ISR responsibilities:**
(a) save processor context
(b) service interrupt
(c) restore processor context

Return from Interrupt instruction

---

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
### Vector Table

#### Interrupt Vector Table (IVT)

This contains the starting address of the ISR for each interrupt source.

#### Alternate Interrupt Vector Table (AIVT)

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000</td>
<td>Reset - goto Instruction</td>
</tr>
<tr>
<td>0x000002</td>
<td>Reset - goto Address</td>
</tr>
<tr>
<td>0x000004</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x000006</td>
<td>Oscillator Fail Trap Vector</td>
</tr>
<tr>
<td>0x000008</td>
<td>Address Error Trap Vector</td>
</tr>
<tr>
<td>0x00000A</td>
<td>Stack Error Trap Vector</td>
</tr>
<tr>
<td>0x00000C</td>
<td>Math Error Trap Vector</td>
</tr>
<tr>
<td>0x00000E</td>
<td>DMAC Error Trap Vector</td>
</tr>
<tr>
<td>0x000014</td>
<td>Interrupt Vector 0</td>
</tr>
<tr>
<td>0x000016</td>
<td>Interrupt Vector 1</td>
</tr>
<tr>
<td>0x0000FC</td>
<td>Interrupt Vector 116</td>
</tr>
<tr>
<td>0x0000FE</td>
<td>Interrupt Vector 117</td>
</tr>
<tr>
<td>0x000100</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x000102</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x000104</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x000106</td>
<td>Oscillator Fail Trap Vector</td>
</tr>
<tr>
<td>0x000108</td>
<td>Address Error Trap Vector</td>
</tr>
<tr>
<td>0x00010A</td>
<td>Stack Error Trap Vector</td>
</tr>
<tr>
<td>0x00010C</td>
<td>Math Error Trap Vector</td>
</tr>
<tr>
<td>0x00010E</td>
<td>DMAC Error Trap Vector</td>
</tr>
<tr>
<td>0x000114</td>
<td>Interrupt Vector 0</td>
</tr>
<tr>
<td>0x000116</td>
<td>Interrupt Vector 1</td>
</tr>
<tr>
<td>0x0001FC</td>
<td>Interrupt Vector 116</td>
</tr>
<tr>
<td>0x0001FE</td>
<td>Interrupt Vector 117</td>
</tr>
<tr>
<td>0x000200</td>
<td>Start of Code</td>
</tr>
</tbody>
</table>

---

Figure redrawn by author from Figure 6-1 of the PIC24 FRM datasheet (DS70224B), Microchip Technology, Inc.

V 0.9

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
<table>
<thead>
<tr>
<th>IVT Address</th>
<th>Vector Num</th>
<th>PIC24 Compiler Name</th>
<th>Vector Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000006</td>
<td>1</td>
<td>_OscillatorFail</td>
<td>Oscillator Failure</td>
</tr>
<tr>
<td>0x000008</td>
<td>2</td>
<td>_AddressError</td>
<td>Address Error</td>
</tr>
<tr>
<td>0x00000A</td>
<td>3</td>
<td>_StackSize</td>
<td>Stack Error</td>
</tr>
<tr>
<td>0x00000C</td>
<td>4</td>
<td>_MathError</td>
<td>Math Error</td>
</tr>
<tr>
<td>0x000014</td>
<td>8</td>
<td>_INT0Interrupt</td>
<td>INT0 – External Interrupt</td>
</tr>
<tr>
<td>0x000016</td>
<td>9</td>
<td>_IC1Interrupt</td>
<td>IC1 – Input Capture 1</td>
</tr>
<tr>
<td>0x000018</td>
<td>10</td>
<td>_OC1Interrupt</td>
<td>OC1 – Output Compare 1</td>
</tr>
<tr>
<td>0x00001A</td>
<td>11</td>
<td>_T1Interrupt</td>
<td>T1 – Timer1 Expired</td>
</tr>
<tr>
<td>0x00001E</td>
<td>13</td>
<td>_IC2Interrupt</td>
<td>IC2 – Input Capture 2</td>
</tr>
<tr>
<td>0x000020</td>
<td>14</td>
<td>_OC2Interrupt</td>
<td>OC2 – Output Compare 2</td>
</tr>
<tr>
<td>0x000022</td>
<td>15</td>
<td>_T2Interrupt</td>
<td>T2 – Timer2 Expired</td>
</tr>
<tr>
<td>0x000024</td>
<td>16</td>
<td>_T3Interrupt</td>
<td>T3 – Timer3 Expired</td>
</tr>
<tr>
<td>0x000026</td>
<td>17</td>
<td>_SPI1ErrInterrupt</td>
<td>SPI1E – SPI1 Error</td>
</tr>
<tr>
<td>0x000028</td>
<td>18</td>
<td>_SPI1Interrupt</td>
<td>SPI1 – SPI1 transfer done</td>
</tr>
<tr>
<td>0x00002A</td>
<td>19</td>
<td>_U1RXInterrupt</td>
<td>U1RX – UART1 Receiver</td>
</tr>
<tr>
<td>0x00002C</td>
<td>20</td>
<td>_U1TXInterrupt</td>
<td>U1TX – UART1 Transmitter</td>
</tr>
<tr>
<td>0x00002E</td>
<td>21</td>
<td>_ADC1Interrupt</td>
<td>ADC1 – ADC 1 convert done</td>
</tr>
<tr>
<td>0x000034</td>
<td>24</td>
<td>_SI2C1Interrupt</td>
<td>SI2C1 – I2C1 Slave Events</td>
</tr>
<tr>
<td>0x000036</td>
<td>25</td>
<td>_MI2C1Interrupt</td>
<td>MI2C1 – I2C1 Master Events</td>
</tr>
<tr>
<td>0x00003A</td>
<td>27</td>
<td>_CNInterrupt</td>
<td>Change Notification Interrupt</td>
</tr>
<tr>
<td>0x00003C</td>
<td>28</td>
<td>_INT1Interrupt</td>
<td>INT1 – External Interrupt</td>
</tr>
<tr>
<td>0x000040</td>
<td>30</td>
<td>_IC7Interrupt</td>
<td>IC7 – Input Capture 7</td>
</tr>
<tr>
<td>0x000042</td>
<td>31</td>
<td>_IC8Interrupt</td>
<td>IC8 – Input Capture 8</td>
</tr>
<tr>
<td>0x00004E</td>
<td>37</td>
<td>_INT2Interrupt</td>
<td>INT2 – External Interrupt</td>
</tr>
<tr>
<td>0x000096</td>
<td>73</td>
<td>_U1EErrInterrupt</td>
<td>U1E – UART1 Error</td>
</tr>
</tbody>
</table>

Interrupt Sources

Serial data has arrived

CNx Pin has changed state

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

4
Interrupt Priorities

An interrupt can be assigned a priority from 0 to 7.

Normal instruction execution is priority 0.

An interrupt MUST have a higher priority than 0 to interrupt normal execution. Assigning a priority of 0 to an interrupt masks (disables) than interrupt.

An interrupt with a higher priority can interrupt a currently executing ISR with a lower priority.

If simultaneous interrupts of the SAME priority occur, then the interrupt with the LOWER VECTOR NUMBER (is first in the interrupt vector table) has the higher natural priority. For example, the INT0 interrupt has a higher natural priority than INT1.
Enabling an Interrupt

Each interrupt source generally has FLAG bit, PRIORITY bits, and an ENBLE bit.

The flag bit is set whenever the flag condition is true, which varies by the interrupt.

The priority bits set the interrupt priority.

The enable bit must be ‘1’ for the ISR to be executed. (NOTE: the enable bit does not have to be a ‘1’ for the flag bit to be set!!!!!).

One of the things that must be done by the ISR is to clear the flag bit, or else the ISR will get stuck in an infinite loop.

By default, all priority bits and enable bits are ‘0’, so interrupt ISRs are disabled from execution.
Traps vs. Interrupts

A Trap is a special type of interrupt, is non-maskable, has higher priority than normal interrupts. **Traps are always enabled!**

Hard trap: CPU stops after instruction at which trap occurs

Soft trap: CPU continues executing instructions as trap is sampled and acknowledged

<table>
<thead>
<tr>
<th>Trap</th>
<th>Category</th>
<th>Priority</th>
<th>Flag(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oscillator Failure</td>
<td>Hard</td>
<td>14</td>
<td>_OSCFAIL (oscillator fail, INTCON1&lt;1&gt;), _CF (clock fail, OSSCON&lt;3&gt;)</td>
</tr>
<tr>
<td>Address Error</td>
<td>Hard</td>
<td>13</td>
<td>_ADDRERR (address error, INTCON1&lt;3&gt;)</td>
</tr>
<tr>
<td>Stack Error</td>
<td>Soft</td>
<td>12</td>
<td>_STKERR (stack error, INTCON1&lt;2&gt;)</td>
</tr>
<tr>
<td>Math Error</td>
<td>Soft</td>
<td>11</td>
<td>_MATHERR (math error, INTCON1&lt;4&gt;)</td>
</tr>
<tr>
<td>DMAC Error</td>
<td>Soft</td>
<td>10</td>
<td>_DMACERR (DMA conflict write, INTCON1&lt;5&gt;)</td>
</tr>
</tbody>
</table>
Interrupt Latency

(a) Latency on Interrupt entry

<table>
<thead>
<tr>
<th>PC</th>
<th>INST_A (PC-2)</th>
<th>INST_B (PC)</th>
<th>FNOP</th>
<th>Fetch Vector</th>
<th>PC+2</th>
<th>PC+2</th>
<th>2000 (ISR)</th>
<th>2002</th>
<th>2004</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Peripheral interrupt event occurs at or before midpoint of this cycle

Save PC in temporary buffer

Push SRL and high 8 bits of PC (from temporary buffer)

Push Low 16 bits of PC (from temporary buffer)

(b) Return from Interrupt timing

<table>
<thead>
<tr>
<th>PC</th>
<th>ISR</th>
<th>ISR+2</th>
<th>PC</th>
<th>PC+2</th>
<th>PC+4</th>
<th>PC+6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

ISR Entry:
Number of cycles from interrupt until 1st instruction of ISR is executed.

ISR Exit:
From RETFIE to program resumed.
ISR Overhead

- **Ientry**: Number of instruction cycles for ISR entry (four on the PIC24 µC).

- **Ibody**: Number of instruction cycles for the ISR body (not including `retfie`).

- **Iexit**: Number of instruction cycles for ISR exit (three on the PIC24 µC).

- **Fisr**: Frequency (number of times per second) at which the ISR is triggered.

- **Tisr**: The ISR triggering period, which is 1/Fisr. For example, if an ISR is executed at 1 KHz, Tisr is 1 ms.
ISR Overhead (cont)

Percentage of CPU time taken up by one ISR:

$$\text{ISR\%} = [(\text{Ientry} + \text{Ibody} + \text{Iexit}) \times F_{\text{isr}}]/F_{\text{cy}} \times 100$$

ISR CPU Percentage for FCY = 40 MHz, IBODY = 50 instr. cycles

<table>
<thead>
<tr>
<th>T_{\text{isr}}</th>
<th>10 ms</th>
<th>1 ms</th>
<th>100 μs</th>
<th>10 μs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISR%</td>
<td>0.01%</td>
<td>0.14%</td>
<td>1.43%</td>
<td>14.3%</td>
</tr>
</tbody>
</table>

**GOLDEN RULE:** An ISR should do its work as quickly as possible. When an ISR is executing, it is keeping other ISRs of equal priority and lower from executing, as well as the main code!
The compiler uses the _DefaultInterrupt function as the default ISR. If an interrupt is triggered, and the ISR is the _DefaultInterrupt, then the user did not expect the interrupt to occur. This means the interrupt is ‘unhandled’. We have written our own _DefaultInterrupt that prints diagnostic information since this is an unexpected occurrence.
Our DefaultInterrupt ISR

Used for all interrupts when you do not provide an ISR.

Our version saves the interrupt source, does a software reset, then interrupt source is printed.
Output from the _DefaultInterrupt ISR

(a) Simplified test code (\texttt{trap_test.c}) to generate a Math Error Trap

```c
int main (void) {
  volatile uint8 u8_zero;
  configBasic(HELLO_MSG);
  while (1) {
    outString("Hit a key to start divide by zero test...");
    inChar();
    outString("OK. Now dividing by zero.\n");
    u8_zero = 0;
    u8_zero = 1/u8_zero; \textcolor{red}{\begin{tabular}{l}
      Generates divide-by-zero
      (Math Error) trap
    \end{tabular}}
    doHeartbeat();
  } // end while (1)
}
```

(b) Console Output

Reset cause: Power-on.
Device ID = 0x00000F1D (PIC24HJ32GP202), revision 0x00003001 (A2)
Fast RC Osc with PLL

\texttt{trap_test.c}, built on Jun 6 2008 at 10:17:57
Hit a key to start divide by zero test...OK. Now dividing by zero.
Reset cause: Software Reset.
Error trapped: Unhandled interrupt, Priority: 0x0B, Vector number: 0x04\textcolor{red}{\begin{tabular}{l}
  _DefaultInterrupt() ISR saves error message and interrupt information
  from INTTREG, then causes the software reset.
  _printResetCause() then prints out the saved error message, interrupt information.
\end{tabular}}
These ISRs just clear the _MATHERR interrupt flag and return. If the interrupt flag is not cleared, get stuck in an infinite interrupt loop.
Change Notification Interrupts

When enabled, triggers an interrupt when a change occurs on a pin.

Figure redrawn by author from Fig 10-4 found in the PIC24 FRM datasheet (DS70230B), Microchip Technology, Inc.
Use Change Notification
to wake from Sleep

//Interrupt Service Routine for Change Notification
void _ISRFAST _CNInterrupt (void) {
    _CNIF = 0; //clear the change notification interrupt bit
}

Clear the interrupt flag before exiting!

/// Switch1 configuration
inline void CONFIG_SW1() {
    CONFIG_RB13_AS_DIG_INPUT(); //use RB13 for switch input
    ENABLE_RB13_PULLUP(); //enable the pull-up
    ENABLE_RB13_CN_INTERRUPT(); //CN13IE = 1
    DELAY_US(1); //Wait for pull-up
}

Macro to set CNxIE bit associated with RB13 port.

int main (void) {
    configBasic(HELLO_MSG);
    //** Configure the switch **********/
    CONFIG_SW1(); //enables individual CN interrupt also
    //** Configure Change Notification general interrupt */
    _CNIF = 0; //Clear the interrupt flag
    _CNIP = 2; //Choose a priority
    _CNIE = 1; //enable the Change Notification general interrupt

    while(1) {
        outString("Entering Sleep mode, press button to wake.\n");
        // Finish sending characters before sleeping
        WAIT_UNTIL_TRANSMIT_COMPLETE_UART1();
        SLEEP(); //macro for asm("pwrsv #0")
    }

    Pushing the switch here generates CN interrupt, causing
    wakeup and execution of the _CNInterrupt ISR, which then
    returns here and loop continues.

    An interrupt flag (_CNIF) should be cleared before the interrupt is enabled (_CNIE=1).
    The priority (_CNIP = 2) chosen here was arbitrary, but it must be greater than 0 for
    the ISR to be executed.
Remappable Pins

Some inputs/outputs for internal modules must be mapped to RPx pins (remappable pins) if they are to be used.

<table>
<thead>
<tr>
<th>Input Name</th>
<th>Function Name</th>
<th>Example Assignment mapping inputs to RPn</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Interrupt 1</td>
<td>INT1</td>
<td>_INT1R = n;</td>
</tr>
<tr>
<td>External Interrupt 2</td>
<td>INT2</td>
<td>_INT2R = n;</td>
</tr>
<tr>
<td>Timer2 Ext. Clock</td>
<td>T2CK</td>
<td>_T2CKR = n;</td>
</tr>
<tr>
<td>Timer3 Ext. Clock</td>
<td>T3CK</td>
<td>_T3CKR = n;</td>
</tr>
<tr>
<td>Input Capture 1</td>
<td>IC1</td>
<td>_IC1R = n;</td>
</tr>
<tr>
<td>Input Capture 2</td>
<td>IC2</td>
<td>_IC2R = n;</td>
</tr>
<tr>
<td>UART1 Receive</td>
<td>U1RX</td>
<td>_U1RXR = n;</td>
</tr>
<tr>
<td>UART1 Clr To Send</td>
<td>U1CTS</td>
<td>_U1CTSR = n;</td>
</tr>
<tr>
<td>SPI1 Data Input</td>
<td>SDI1</td>
<td>_SDI1R = n;</td>
</tr>
<tr>
<td>SPI1 Clock Input</td>
<td>SCK1</td>
<td>_SCK1R = n;</td>
</tr>
<tr>
<td>SPI1 Slave Sel. Input</td>
<td>SS1</td>
<td>_SS1R = n;</td>
</tr>
</tbody>
</table>
## Remappable Pins (cont.)

<table>
<thead>
<tr>
<th>Output Name</th>
<th>Function</th>
<th>RPnR&lt;4:0&gt;</th>
<th>Example Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Port Pin</td>
<td>NULL</td>
<td>0</td>
<td>_RPnR = 0;</td>
</tr>
<tr>
<td>UART1 Transmit</td>
<td>U1TX</td>
<td>3</td>
<td>_RPnR = 3;</td>
</tr>
<tr>
<td>UART1 Rdy. To Send U1RTS</td>
<td></td>
<td>4</td>
<td>_RPnR = 4;</td>
</tr>
<tr>
<td>SPI1 Data Output</td>
<td>SDO1</td>
<td>7</td>
<td>_RPnR = 7;</td>
</tr>
<tr>
<td>SPI1 Clock Output</td>
<td>SCK1OUT</td>
<td>8</td>
<td>_RPnR = 8;</td>
</tr>
<tr>
<td>SPI1 Slave Sel. Out.</td>
<td>SS1OUT</td>
<td>9</td>
<td>_RPnR = 9;</td>
</tr>
<tr>
<td>Output Compare 1</td>
<td>OC1</td>
<td>18</td>
<td>_RPnR = 18;</td>
</tr>
<tr>
<td>Output Compare 2</td>
<td>OC2</td>
<td>19</td>
<td>_RPnR = 19;</td>
</tr>
</tbody>
</table>

Mapping outputs to RPx pins.
Remapping Macros

Contained in pic24_ports.h:

CONFIG_U1RX_TO_RP(pin)
CONFIG_U1TX_TO_RP(pin)

etc..

Example Usage:

CONFIG_U1RX_TO_RP(10);  //UART1 RX to RP10
CONFIG_U1TX_TO_RP(11);  //UART1 TX to RP11
INT2, INT1, INT0 Interrupts

These are input interrupt sources (INTx) that can be configured to be rising edge triggered or falling-edge triggered by using an associated INTxE bit (‘1’ is falling edge, ‘0’ is rising edge).

On the PIC24HJ32GP202, INT1 and INT2 must be brought out to remappable pins (RPx); INT0 is assigned a fixed pin location.
//Interrupt Service Routine for INT1
void _ISRFAST _INT1Interrupt (void) {
  _INT1IF = 0;    //clear the interrupt bit
}

/// Switch1 configuration, use RB13
inline void CONFIG_SW1() {
  CONFIG_RB13_AS_DIG_INPUT();   //use RB13 for switch input
  ENABLE_RB13_PULLUP();         //enable the pullup
  DELAY_US(1);                  // Wait for pull-up
}

int main (void) {
  configBasic(HELLO_MSG);
  /** Configure the switch ***********/
  CONFIG_SW1();
  CONFIG_INT1_TO_RP(13);      //map INT1 to RP13
  /** Configure INT1 interrupt */
  _INT1IF = 0;     //Clear the interrupt flag
  _INT1IP = 2;     //Choose a priority
  _INT1EP = 1;     //negative edge triggered
  _INT1IE = 1;     //enable INT1 interrupt
  while(1) {
    outString("Entering Sleep mode, press button to wake.\n");
    //finish sending characters before sleeping
    WAIT_UNTIL_TRANSMIT_COMPLETE_UART1();
    SLEEP();       //macro for asm("pwrsv #0")
  }
}

Use INT1 to wake from Sleep mode
Timers

Recall that a Timer is just a counter. Time can be converted from elapsed Timer Ticks ($Ticks$) by multiplying by the clock period ($Ttmr$) of the timer:

$$\text{Time} = \text{Ticks} \times \text{Ttmr}$$

If a timer is a 16-bit timer, and it is clocked at the FCY = 40 MHz, then it will count from 0x0000 to 0xFFFF (65536 ticks) in:

$$\text{Time} = 65536 \times \frac{1}{40 \text{ MHz}}$$

$$= 65536 \times 25 \text{ ns} = 1638400 \text{ ns} = 1638.4 \text{ us} = 1.6384 \text{ ms}$$
The Timer 3 block diagram is the same, with TMR3, PR3 used for these registers and T3IF for the interrupt flag.
T2IF Period

The T2IF flag is set at the following period \((T_{t2if})\):

\[ T_{t2if} = (PR2+1) \times PRE \times T_{cy} = (PR2+1) \times PRE/F_{cy} \]

Observe that because Timer2 is a 16-bit timer, if PR2 is its maximum value of 0xFFFF (65535), and the prescaler is ‘1’, this is just:

\[ T_{t2if} = 65536 \times 1/F_{cy} \]

We typically want to solve for \(T_{t2if}\), given a PRE value:

\[ PR2 = (T_{t2if} \times F_{cy} /PRE ) – 1 \]
Example T2IF Periods

PR2/PRE Values for $T_{t2if} = 15$ ms, $Fcy = 40$ MHz

<table>
<thead>
<tr>
<th>PRE=1</th>
<th>PRE=8</th>
<th>PRE=64</th>
<th>PRE=256</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR2</td>
<td>600000</td>
<td>75000</td>
<td>9375</td>
</tr>
</tbody>
</table>

(invalid)  (invalid)

The PR2 for PRE=1, PRE=8 are invalid because they are greater than 65535 (PR2 is a 16-bit register).

Configuring Timer2 to interrupt every $T_{t2if}$ period is called a **PERIODIC INTERRUPT**.
Utility Functions

// convert milliseconds to Timer Ticks
uint16 msToU16Ticks(uint16 u16_ms, uint16 u16_pre) {
    float f_ticks = Fcy;
    uint16 u16_ticks;
    f_ticks = (f_ticks*u16_ms)/u16_pre/1000L;
    ASSERT(f_ticks < 65535.5);
    u16_ticks = roundFloatToUint16(f_ticks); // back to integer
    return u16_ticks;
}

// convert microseconds to Timer Ticks
uint16 usToU16Ticks(uint16 u16_us, uint16 u16_pre) {
    float f_ticks = Fcy;
    uint16 u16_ticks;
    f_ticks = (f_ticks*u16_us)/u16_pre/1000000L;
    ASSERT(f_ticks < 65535.5);
    u16_ticks = roundFloatToUint16(f_ticks); // back to integer
    return u16_ticks;
}

// return the timer prescale based on the TxCNBits SFR
#define getTimerPrescale(TxCNBits) getTimerPrescaleBits(TxCNBits.TCKPS)

// return the timer prescale based on the TCKPS bitfield in TxCNBits
uint16 getTimerPrescaleBits(uint8 u8_TCKPS) {
    uint16 au16_prescaleValue[] = { 1, 8, 64, 256 };
    ASSERT(u8_TCKPS <= 3);
    return au16_prescaleValue[u8_TCKPS];
}
# Timer2 Control Register

<table>
<thead>
<tr>
<th>R/W-0</th>
<th>U-0</th>
<th>R/W-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TON</td>
<td>UI</td>
<td>TSIDL</td>
<td>UI</td>
<td>UI</td>
<td>UI</td>
<td>UI</td>
<td>UI</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>U-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>U-0</td>
<td>R/W-0</td>
<td>U-0</td>
</tr>
<tr>
<td>UI</td>
<td>TGATE</td>
<td>TCKPS&lt;1:0&gt;</td>
<td>T32</td>
<td>UI</td>
<td>TCS</td>
<td>UI</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Bit 15: TON: Timer2 On Bit

When T32 = 1:

- 1 = Starts 32-bit Timer2/3
- 0 = Stops 32-bit Timer2/3

When T32 = 0:

- 1 = Starts 16-bit Timer2
- 0 = Stops 16-bit Timer2

Bit 13: TSIDL: Stop in Idle Mode Bit

- 1 = Discontinue module operation device enters Idle mode
- 0 = Continue module operation in Idle mode

Bit 6: TGATE: Timer2 Gated Time Accumulation Enable

When TCS = 1:

- This bit is ignored.
- 1 = Gated time accumulation enabled
- 0 = Gated time accumulation disabled

When TCS = 0:

- 1 = Gated time accumulation enabled
- 0 = Gated time accumulation disabled

Bit 5-4: TCKPS<1:0>: Timer2 Input Clock Prescale Select Bits

- 11 = 1:256, 10 = 1:64, 01 = 1:8, 00 = 1:1

Bit 3: T32: 32-bit Timer Mode Select bit

- 1 = Timer2 and Timer3 form a single 32-bit timer
- 0 = Timer2 and Timer3 act as two 16-bit timers

Bit 1: TCS: Timer2 Clock Source Select bit

- 1 = External clock from pin T2CK (on the rising edge)
- 0 = Internal clock (FCY)

---

**Legend:**

- R = Readable bit
- -n = Value at POR
- U = Unimplemented bit, read as ‘0’
- W = Writeable bit
- ‘1’ = bit is set
- ‘0’ = bit is cleared
- ‘x’ = bit is unknown

---

**Note 1:** In 32-bit mode, T3CON bits do not affect 32-bit operation.
Square Wave Generation

Timer2 configuration sets T2CON, PR2; enables the Timer2 interrupt; turns on the timer.

The msTo16Ticks() value is decremented by 1 before PR2 assignment because timer period is PR2+1 to generate an interrupt every 15 ms. An output pin is toggled in the ISR, so square wave has period of 30 ms.

After configuration, the ISR does the work of generating the square wave.
Switch Input: Polling

```c
#define CONFIG_LED1()   CONFIG_RB14_AS_DIG_OUTPUT()
#define LED1 _LATB14     // led1 state
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) {
    CONFIG_SW1();       // configure switch
    CONFIG_LED1();      // configure LED
    DELAY_US(1);        // give pullups a little time
    LED1 = 0;           // LED off initially
    while (1) {
        while (SW1_RELEASED()) doHeartbeat(); // loop (1)
        DELAY_MS(15); // debounce
        while (SW1_PRESSED()) doHeartbeat(); // loop (2)
        DELAY_MS(15); // debounce
        LED1 = !LED1;  // toggle the LED
    }
}  
```

ledtoggle_nofsm.c from Ch 8 examples
volatile uint8_t bcnt;

void _ISRFAST _CNInterrupt (void) {
  _CNIF = 0;    //clear the interrupt bit
  bcnt++;       //increment the bounce count
}

#define SW1             _RB8
#define SW1_PRESSED()   (SW1==0)
#define SW1RELEASED()   (SW1==1)

inline void CONFIG_SW1()  {
  CONFIG_RB8_AS_DIG_INPUT();
  ENABLE_RB8_PULLUP();
  ENABLE_RB8_CN_INTERRUPT();
}

int main (void) {
  configBasic(HELLO_MSG);
  CONFIG_SW1();
  _CNIF = 0;         //Clear the interrupt flag
  _CNIP = 2;         //Choose a priority
  _CNIE = 1;         //enable the interrupt
  while (1) {
    bcnt = 0;
    outString("Press & release switch... ");
    while (SW1RELEASED());
    DELAY_MS(DEBOUNCE_DLY);
    while (SW1_PRESSED());
    DELAY_MS(DEBOUNCE_DLY);
    if (bcnt != 2) outString("..bounced: ");
    else outString("..no bounce: ");
    outUint8(bcnt);
    outString("\n");
  }
}
Ignore Switch Bounce?

ToggleLED.c

```c
#define SW1 _RB8
#define SW1_PRESSED() (SW1==0)
#define SW1_RELEASED() (SW1==1)
#define LED _LATB14
#define CONFIG_LED()
    CONFIG_RB14_AS_DIG_OUTPUT()

void _ISRFAST _CNInterrupt (void) {
    _CNIF = 0;
    if (SW1_RELEASED())
        LED = !LED;
}
inline void CONFIG_SW1() {
    CONFIG_RB8_AS_DIG_INPUT();
    ENABLE_RB8_PULLUP();
    ENABLE_RB8_CN_INTERRUPT();
}

int main (void) {
    configBasic(HELLO_MSG);
    CONFIG_SW1();
    CONFIG_LED();
    DELAY_US(1);  //delay for pullup
    _CNIF = 0;    //Clear interrupt flag
    _CNIP = 2;    //Choose a priority
    _CNIE = 1;    //enable interrupt
    LED = 1;
    while (1) {
        doHeartbeat();
    }
}
```
Eliminate Switch Bounce

**ToggleLEDWithTimer.c**

```c
#define CONFIG_LED()
    CONFIG_RB14_AS_DIG_OUTPUT()
#define LED  _LATB14
#define SW1   _RB8
inline void CONFIG_SW1()  {
    CONFIG_RB8_AS_DIG_INPUT
    ENABLE_RB8_PULLUP
}

volatile uint8_t SWvalue = 0;
volatile uint8_t oldSWvalue = 0;

void _ISRFAST _T3Interrupt (void) {
    _T3IF = 0; //clear the timer interrupt bit
    SWvalue = SW1;
    if(SWvalue == 1 && oldSWvalue == 0) {
        LED = !LED;
    }
    oldSWvalue = SWvalue;
}

#define ISR_PERIOD     15      // in ms
void  configTimer3(void) {
    T2CONbits.T32 = 0;   // 32-bit mode off
    T3CON = T3_OFF |T3_IDLE_CON
     | T3_GATE_OFF | T3_SOURCE_INT
     | T3_PS_1_64 ;
    PR3 = msToU16Ticks (ISR_PERIOD,
               \ getTimerPrescale(T3CONbits)) - 1;
    TMR3  = 0;                      //clear timer3 value
    _T3IF = 0;                       //clear interrupt flag
    _T3IP = 1;                       //choose a priority
    _T3IE = 1;                       //enable the interrupt
    T3CONbits.TON = 1;       //turn on the timer
}

int main (void) {
    configBasic(HELLO_MSG);
    CONFIG_SW1();
    CONFIG_LED();
    configTimer3();
    configBasic(HELLO_MSG);
    while (1) {
        doHeartbeat();
    }
}
```