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.
The normal program flow (main) is referred to as the foreground code. The interrupt service routine (ISR) is referred to as the background code.
<table>
<thead>
<tr>
<th>Vector Table</th>
<th>Interrupt Vector Table (IVT)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reset - goto Instruction</strong></td>
<td>0x000000</td>
</tr>
<tr>
<td><strong>Reset - goto Address</strong></td>
<td>0x000002</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>0x000004</td>
</tr>
<tr>
<td><strong>Oscillator Fail Trap Vector</strong></td>
<td>0x000006</td>
</tr>
<tr>
<td><strong>Address Error Trap Vector</strong></td>
<td>0x000008</td>
</tr>
<tr>
<td><strong>Stack Error Trap Vector</strong></td>
<td>0x00000A</td>
</tr>
<tr>
<td><strong>Math Error Trap Vector</strong></td>
<td>0x00000C</td>
</tr>
<tr>
<td><strong>DMAC Error Trap Vector</strong></td>
<td>0x00000E</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>0x000010</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>0x000012</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>0x000014</td>
</tr>
<tr>
<td><strong>Oscillator Fail Trap Vector</strong></td>
<td>0x000106</td>
</tr>
<tr>
<td><strong>Address Error Trap Vector</strong></td>
<td>0x000108</td>
</tr>
<tr>
<td><strong>Stack Error Trap Vector</strong></td>
<td>0x00010A</td>
</tr>
<tr>
<td><strong>Math Error Trap Vector</strong></td>
<td>0x00010C</td>
</tr>
<tr>
<td><strong>DMAC Error Trap Vector</strong></td>
<td>0x00010E</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>0x000110</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>0x000112</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>0x000114</td>
</tr>
<tr>
<td><strong>Interrupt Vector 0</strong></td>
<td>0x000116</td>
</tr>
<tr>
<td><strong>Interrupt Vector 1</strong></td>
<td>0x000118</td>
</tr>
<tr>
<td><strong>Interrupt Vector 116</strong></td>
<td>0x00011A</td>
</tr>
<tr>
<td><strong>Interrupt Vector 117</strong></td>
<td>0x00011C</td>
</tr>
<tr>
<td><strong>Start of Code</strong></td>
<td>0x000200</td>
</tr>
</tbody>
</table>

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

Figure redrawn by author from Figure 6-1 of the PIC24 FRM datasheet (DS70224B), Microchip Technology, Inc.
<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>0x00001A</td>
<td>10</td>
<td>_OC1Interrupt</td>
<td>OC1 – Output Compare 1</td>
</tr>
<tr>
<td>0x00001E</td>
<td>11</td>
<td>_T1Interrupt</td>
<td>T1 – Timer1 Expired</td>
</tr>
<tr>
<td>0x000020</td>
<td>13</td>
<td>_IC2Interrupt</td>
<td>IC2 – Input Capture 2</td>
</tr>
<tr>
<td>0x000022</td>
<td>14</td>
<td>_OC2Interrupt</td>
<td>OC2 – Output Compare 2</td>
</tr>
<tr>
<td>0x000026</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>_M12C1Interrupt</td>
<td>M12C1 – I2C1 Master Events</td>
</tr>
<tr>
<td>0x00003A</td>
<td>27</td>
<td>_CN1Interrupt</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>32</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>

*Copyright Delmar Cengage Learning 2008. All Rights Reserved.*

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
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 interrupt condition occurs, which varies by the interrupt.

The priority bits set the interrupt priority.

The enable bit must be a ‘1’ AND the interrupt priority > 0 for the ISR to be executed (interrupt is enabled). (NOTE: the interrupt does not have to be enabled 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 PIC24 CPU will get stuck in an infinite loop executing the ISR.

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>2000 (ISR)</th>
<th>2002</th>
<th>2004</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC+2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>INST Executed</td>
<td>Tcy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Priority</td>
<td>Peripheral interrupt event occurs at or before midpoint of this cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

(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>INST Executed</td>
<td>ISR last instr.</td>
<td>RETFIE 2nd cycle</td>
<td>PC</td>
<td>PC+2</td>
<td>PC+4</td>
<td>PC+6</td>
</tr>
<tr>
<td>CPU Priority</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

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:
ISR% = [(Ientry + Ibody + Iexit) x Fisr]/Fcy x 100

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

<table>
<thead>
<tr>
<th>Tisr</th>
<th>0.01%</th>
<th>0.14%</th>
<th>1.43%</th>
<th>14.3%</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 μs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 μs</td>
<td></td>
<td></td>
<td></td>
<td></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.

(a) Code for default interrupt handler

```c
_PERSISTENT const char* sz_lastError;
_PERSISTENT char* sz_lastTimeoutError;
_PERSISTENT UINT32BITS INTTREGBITS_last;
#define u16_INTTREGLast \
BITS2WORD(INTTREGBITS_last)

void _ISR_DefaultInterrupt(void) {
    u16_INTTREGLast = INTTREG;
    reportError("Unhandled interrupt, ");
}

void reportError(const char*
    sz_errorMessage) {
    sz_lastError = sz_errorMessage;
    asm ("reset");
}

void printResetCause(void) {
    ...
    if (u16_INTTREGLast != 0) {
        Output error message saved from last reset
        outString("Error trapped: ");
        outString(sz_lastError);
        if (sz_lastInterrupt != 0) {
            If last reset was caused by an unhandled interrupt, print the priority (ILR) and vector number (VECNUM)
            outString("Priority: ");
            outUInt8(INTTREGBITS_last.ILR);
            outString("", Vector number: ");
            outUInt8(INTTREGBITS_last.VECNUM);
        }
        outString("\n\n");
        sz_lastError = NULL;  }
    u16_INTTREGLast = 0;  }
    Clear _PERSISTENT error variables.
}

_V 2.0

12
```
Output from the _DefaultInterrupt ISR

(a) Simplified test code (\textit{trap\_test.c}) to generate a Math Error Trap

\begin{verbatim}
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}{Generates divide-by-zero \textcolor{red}{(Math Error) trap}}
    doHeartbeat();
  } // end while (1)
}
\end{verbatim}

(b) Console Output

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

\begin{verbatim}
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
_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{verbatim}
In C

```c
void __ISR __attribute__((interrupt)) MathError (void) {
    // only action is
    // to clear the error
    _MATHERR = 0;
    RCOUNT = 0;
}
```

In Assembly (C30)

```
MathError:
    push PSVPAG ;save
    push W8 ;save
    mov.b #0,W8
    mov W8, PSVPAG ;PSVPAG = page 0
    pop W8 ;restore
    ;clr _MATHERR flag
    bclr.b INTCON1,#4 ;_MATHERR = 0
    clr RCOUNT ;RCOUNT = 0
    pop PSVPAG ;restore
    retfie ;return from interrupt
```

In include\pic24_util.h

```c
#define __ISR __attribute__((interrupt)) __attribute__((auto_psv))
```

In C

```c
void __ISRFAST MathError (void) {
    // only action is
    // to clear the error
    _MATHERR = 0;
    RCOUNT = 0;
}
```

In Assembly (C30)

```
MathError:
    ;clr _MATHERR flag
    bclr.b INTCON1,#4 ;_MATHERR = 0
    clr RCOUNT ;RCOUNT = 0
    retfie ;return from interrupt
```

In include\pic24_util.h

```c
#define __ISRFAST __attribute__((interrupt)) __attribute__((no_auto_psv))
```

(c) MPLAB Program Memory

<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Opcode</th>
<th>Label</th>
<th>Dis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0000</td>
<td>040C02</td>
<td>goto _reset</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0002</td>
<td>000000</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0004</td>
<td>000D8C</td>
<td>_DefaultInterrupt</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0006</td>
<td>000D8C</td>
<td>_DefaultInterrupt</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0008</td>
<td>000D8C</td>
<td>_DefaultInterrupt</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>000A</td>
<td>000D8C</td>
<td>_DefaultInterrupt</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>000C</td>
<td>00109A</td>
<td>_MathError</td>
<td></td>
</tr>
</tbody>
</table>

Math Error Trap vector now contains address of _MathError ISR.

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.

V 2.0
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
}  

// 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("pwrsav #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</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("pwrsav #0")
  }
}
Hardware 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:

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

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

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

$$= 65536 \times 25 \text{ ns} = 1638400 \text{ ns} = 1638.4 \text{ us} = 1.6384 \text{ ms}$$
Timer 2 Block Diagram

The Timer 3 block diagram is the same, with TMR3, PR3 used for these registers and T31F for the interrupt flag.

Figure redrawn by author from Fig 11-2 found in the PIC24H32GP202 datasheet (DS70289B), Microchip Technology, Inc.
T2IF Period

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

$$T_{t2if} = (PR2+1) \times PRE \times T_{cy} = (PR2+1) \times \frac{PRE}{F_{cy}}$$

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:

$$T_{t2if} = 65536 \times \frac{1}{F_{cy}}$$

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

$$PR2 = \left( \frac{T_{t2if} \times F_{cy}}{PRE} \right) - 1$$
Example T2IF Periods

PR2/PRE Values for $T_{t2if} = 15$ ms, $F_{cy} = 40$ 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>
<tr>
<td>(invalid)</td>
<td>(invalid)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
## 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>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Bit 15: TON: Timer2 On Bit
- When T32 = 1: Starts 32-bit Timer2/3
- When T32 = 0: Starts 16-bit Timer2
- 0 = Stops 32-bit Timer2/3
- 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: Gated time accumulation enabled
- When TCS = 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)

---

### Include File Excerpts:

```c
/* T2CON: TIMER2 CONTROL REGISTER */
#define T2_ON 0x8000
#define T2_OFF 0x0000
#define T2_IDLE_STOP 0x2000
#define T2_IDLE_CON 0x0000
#define T2_GATE_ON 0x0040
#define T2_GATE_OFF 0x0000
#define T2_PS_1_1 0x0000
#define T2_PS_1_8 0x0010
#define T2_PS_1_64 0x0020
#define T2_PS_1_256 0x0030
#define T2_32BIT_MODE_ON 0x0008
#define T2_32BIT_MODE_OFF 0x0000
#define T2_SOURCE_EXT 0x0002
#define T2_SOURCE_INT 0x0000
```

---

**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.

---

V 2.0

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

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

Just write a 16-bit value to the Timer2 configuration register to configure Timer2:

\[
T2CON = 0x0020; //\text{Timer off, Pre}=64, \text{Internal clock}
\]

More readable:

\[
T2CON = T2\_OFF | T2\_IDLE\_CON | T2\_GATE\_OFF |
T2\_32BIT\_MODE\_OFF | T2\_SOURCE\_INT |
T2\_PS\_1\_64;
\]

This is actually:

\[
T2CON = 0x0000 | 0x0000 | 0x00000 |
0x0000 | 0x0000 |
0x0020;
\]

Can also set individual bit fields:

\[
T2CON\_bits.TON = 1; //\text{Set TON bit}=1, \text{turn timer on}
\]
#define WAVEOUT_LATB2 //state
inline void CONFIG_WAVEOUT() {
    CONFIG_RB2_AS_DIG_OUTPUT(); //use RB2 for output
}

//Interrupt Service Routine for Timer2
void _ISRFAST _T2Interrupt (void){
    WAVEOUT = !WAVEOUT; //toggle output
    _T2IF = 0; //clear the interrupt bit
}

#define ISR_PERIOD 15 // in ms
void configTimer2(void) {
    //T2CON set like this for documentation purposes.
    //could be replaced by T2CON = 0x0020
    T2CON = T2_OFF | T2_IDLE_CON | T2_GATE_OFF
         | T2_32BIT_MODE_OFF
         | T2_SOURCE_INT
         | T2_PS_1_64; // results in T2CON= 0x0020
    //subtract 1 from ticks value assigned to PR2 because period is PR2 + 1
    PR2 = msToU16Ticks (ISR_PERIOD, getTimerPrescale(T2CON)) - 1;
    TMR2 = 0; // clear timer2 value
    _T2IF = 0; // clear interrupt flag
    _T2IP = 1; // choose a priority
    _T2IE = 1; // enable the interrupt
    T2CONbits.TON = 1; // turn on the timer
}

int main (void) {
    configBasic(HELLO_MSG);
    CONFIG_WAVEOUT(); // PIO Config
    configTimer2(); // TMR2 config
    // ISR does the work!
    while (1) {
        doHeartbeat(); // ensure that we are alive
    } // end while (1)
}
Switch Sampling

A Timer3 periodic Timer interrupt is used to sample the switch.

Switch state is now stored in a variable!
#define ISR_PERIOD 15 // in ms

void configTimer3(void) {
    // ensure that Timer2,3 configured as separate timers.
    T2CONbits.T32 = 0; // 32-bit mode off
    // T3CON set like this for documentation purposes.
    // could be replaced by T3CON = 0x0020
    T3CON = T3_OFF | T3_IDLE_CON | T3_GATE_OFF
            | T3_SOURCE_INT
            | T3_PS_1_64; // results in T3CON= 0x0020
    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) {
    STATE e_mystate;
    //... config not shown ...
    e_mystate = STATE_WAIT_FOR_PRESS;
    while (1) {
        printNewState(e_mystate);
        switch (e_mystate) {
            case STATE_WAIT_FOR_PRESS:
                if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE;
                break;
            case STATE_WAIT_FOR_RELEASE:
                if (SW1_RELEASED()) {
                    LED1 = !LED1;     //toggle LED
                    e_mystate = STATE_WAIT_FOR_PRESS;
                }
                break;
            default:
                e_mystate = STATE_WAIT_FOR_PRESS;
                break;          //end switch(e_mystate)
        } //end switch(e_mystate)
        doHeartbeat();    //ensure that we are alive
    } // end while (1)
}

Switch Sampling
(cont.)

removed from end of loop as the ISR periodically samples the input.
Semaphores

Will use a ‘button press & release’ semaphore to implement this as one state named

SW1 pressed & released. Will now be implemented as one state named

WAIT_FOR_PNR1

SW1 pressed & released. Will now be implemented as one state named

WAIT_FOR_PNR2

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”. 
Press&Release Semaphore

ISR is now a state machine!

A semaphore is a flag set by an ISR when an IO event occurs. The main() code is generally responsible for clearing the flag.

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”. V 2.0
```c
typedef enum {
    STATE_RESET = 0,    STATE_WAIT_FOR_PNR1,    STATE_WAIT_FOR_PNR2,
    STATE_BLINK,    STATE_WAIT_FOR_RELEASE3
} STATE;

int main (void) {
    STATE e_mystate;
    ... config not shown ...
    e_mystate = STATE_WAIT_FOR_PNR1;
    while (1) {
        printNewState(e_mystate);
        switch (e_mystate) {
            case STATE_WAIT_FOR_PNR1:
                LED1 = 0; // turn off the LED
                if (u8_pnrSW1) {
                    u8_pnrSW1 = 0; // clear
                    e_mystate = STATE_WAIT_FOR_PNR2;
                }
                break;
            case STATE_WAIT_FOR_PNR2:
                LED1 = 1; // turn on the LED
                if (u8_pnrSW1) {
                    u8_pnrSW1 = 0; // clear semaphore
                    if (SW2) e_mystate = STATE_BLINK;
                    else e_mystate = STATE_WAIT_FOR_PNR1;
                }
                break;
            case STATE_BLINK:
                LED1 = !LED1; DELAY_MS(100); // blink if not pressed
                if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE3;
                break;
            case STATE_WAIT_FOR_RELEASE3:
                LED1 = 1; // Freeze LED1 at 1
                if (u8_pnrSW1) {
                    u8_pnrSW1 = 0;
                    e_mystate = STATE_WAIT_FOR_PNR1;
                }
                break;
            default:
                default:
                e_mystate = STATE_WAIT_FOR_PNR1;
        } // end switch (e_mystate)
        doHeartbeat(); // ensure that we are alive
    } // end while (1)
}
```

**Differences:**

Only one state used for each press and release.

Use the `u8_pnrSW1` semaphore to determine when press/release occurred.
Another Solution

Put entire FSM into the ISR instead of using a press\&release semaphore.

Now use a doBlink semaphore to tell the main() code when to blink the LED.

Do not Blink in ISR!
This delays exit from ISR.
Dividing Work between the ISR and main()

There are usually multiple ways to divide work between the ISR and main().

The ‘right’ choice is the one that services the I/O event in a timely manner, and there can be more than right choice.

Golden Rules:

The ISR should do its work as fast as possible.
Do not put long software delays into an ISR.
An ISR should never wait for I/O, the I/O event should trigger the ISR or the ISR should just sample the input!
An ISR is never called as a subroutine.
What do you have to know?

• How interrupts behave on the PIC24 μC
• Interrupt Priorities, Enabling of Interrupts
• Traps vs. Interrupts
• Change notification Interrupts
• Timer2 operation
• Periodic Interrupt generation
• Switch sampling using periodic timer interrupts