Microcomputers

PIC24 Interrupts

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 (even as simple as a PIO pin) 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

// Normal Program flow
main() {
    instr1
    instr2
    instr3
    ...
    instrN // Interrupt occurs
    instrN+1
    instrN+2
    ...
    ...
}

// Interrupt Service Routine (ISR)
_ISR InterruptName() {
    // ISR responsibilities
    (a) save µC context
    (b) service interrupt
    (c) restore µC context
}

retfie // return from interrupt

• ISR is called by interrupt generation logic (i.e. hardware). main() code does not call ISR explicitly.

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

Interrupt Vector Table

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

Derived from Figure 7-1, PIC24HJ128GP502 datasheet
Interrupt Sources

<table>
<thead>
<tr>
<th>IRQ</th>
<th>Primary Name</th>
<th>Alternate Name</th>
<th>Vector Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>Reserved</td>
<td>AllReservedTrap</td>
<td>Reserved</td>
</tr>
<tr>
<td>N/A</td>
<td>Dec0Status</td>
<td>Status0</td>
<td>Dec0Status</td>
</tr>
<tr>
<td>N/A</td>
<td>AddressErr</td>
<td>AddressErr</td>
<td>Address Err</td>
</tr>
<tr>
<td>N/A</td>
<td>StackErr</td>
<td>StackErr</td>
<td>Stack Err</td>
</tr>
<tr>
<td>N/A</td>
<td>MemoryErr</td>
<td>MemoryErr</td>
<td>Memory Err</td>
</tr>
<tr>
<td>N/A</td>
<td>GND38.31</td>
<td>AllGND38.31</td>
<td>Reserved</td>
</tr>
<tr>
<td>N/A</td>
<td>ReservedTrap</td>
<td>ReservedTrap</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>INT0</td>
<td>INT0</td>
<td>Interrupt 0</td>
</tr>
<tr>
<td>1</td>
<td>INT1</td>
<td>INT1</td>
<td>Interrupt 1</td>
</tr>
<tr>
<td>2</td>
<td>INT2</td>
<td>INT2</td>
<td>Interrupt 2</td>
</tr>
<tr>
<td>3</td>
<td>INT3</td>
<td>INT3</td>
<td>Interrupt 3</td>
</tr>
<tr>
<td>4</td>
<td>INT4</td>
<td>INT4</td>
<td>Interrupt 4</td>
</tr>
<tr>
<td>5</td>
<td>INT5</td>
<td>INT5</td>
<td>Interrupt 5</td>
</tr>
<tr>
<td>6</td>
<td>INT6</td>
<td>INT6</td>
<td>Interrupt 6</td>
</tr>
<tr>
<td>7</td>
<td>INT7</td>
<td>INT7</td>
<td>Interrupt 7</td>
</tr>
<tr>
<td>8</td>
<td>INT8</td>
<td>INT8</td>
<td>Interrupt 8</td>
</tr>
<tr>
<td>9</td>
<td>INT9</td>
<td>INT9</td>
<td>Interrupt 9</td>
</tr>
<tr>
<td>10</td>
<td>INT10</td>
<td>INT10</td>
<td>Interrupt 10</td>
</tr>
<tr>
<td>11</td>
<td>INT11</td>
<td>INT11</td>
<td>Interrupt 11</td>
</tr>
<tr>
<td>12</td>
<td>INT12</td>
<td>INT12</td>
<td>Interrupt 12</td>
</tr>
<tr>
<td>13</td>
<td>INT13</td>
<td>INT13</td>
<td>Interrupt 13</td>
</tr>
<tr>
<td>14</td>
<td>INT14</td>
<td>INT14</td>
<td>Interrupt 14</td>
</tr>
<tr>
<td>15</td>
<td>INT15</td>
<td>INT15</td>
<td>Interrupt 15</td>
</tr>
<tr>
<td>16</td>
<td>INT16</td>
<td>INT16</td>
<td>Interrupt 16</td>
</tr>
<tr>
<td>17</td>
<td>INT17</td>
<td>INT17</td>
<td>Interrupt 17</td>
</tr>
<tr>
<td>18</td>
<td>INT18</td>
<td>INT18</td>
<td>Interrupt 18</td>
</tr>
<tr>
<td>19</td>
<td>INT19</td>
<td>INT19</td>
<td>Interrupt 19</td>
</tr>
<tr>
<td>20</td>
<td>INT20</td>
<td>INT20</td>
<td>Interrupt 20</td>
</tr>
<tr>
<td>21</td>
<td>INT21</td>
<td>INT21</td>
<td>Interrupt 21</td>
</tr>
<tr>
<td>22</td>
<td>INT22</td>
<td>INT22</td>
<td>Interrupt 22</td>
</tr>
<tr>
<td>23</td>
<td>INT23</td>
<td>INT23</td>
<td>Interrupt 23</td>
</tr>
<tr>
<td>24</td>
<td>INT24</td>
<td>INT24</td>
<td>Interrupt 24</td>
</tr>
<tr>
<td>25</td>
<td>INT25</td>
<td>INT25</td>
<td>Interrupt 25</td>
</tr>
<tr>
<td>26</td>
<td>INT26</td>
<td>INT26</td>
<td>Interrupt 26</td>
</tr>
<tr>
<td>27</td>
<td>INT27</td>
<td>INT27</td>
<td>Interrupt 27</td>
</tr>
<tr>
<td>28</td>
<td>INT28</td>
<td>INT28</td>
<td>Interrupt 28</td>
</tr>
<tr>
<td>29</td>
<td>INT29</td>
<td>INT29</td>
<td>Interrupt 29</td>
</tr>
<tr>
<td>30</td>
<td>INT30</td>
<td>INT30</td>
<td>Interrupt 30</td>
</tr>
<tr>
<td>31</td>
<td>INT31</td>
<td>INT31</td>
<td>Interrupt 31</td>
</tr>
</tbody>
</table>

Derived from Table 7-4, MPLAB C30 C COMPILER USER’S GUIDE

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) that 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 ENABLE 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

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

ISR Exit: From RETFIE to program resumed.

ISR Overhead Terminology

- **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 Calculation

- Percentage of CPU time taken up by one ISR:
  \[ \text{ISR\%} = \left( I_{\text{entry}} + I_{\text{body}} + I_{\text{exit}} \right) \times \frac{\text{Fisr}}{\text{Fcy}} \times 100 \]

- **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!

- **EXAMPLE:** ISR CPU Percentage for FCY = 40 MHz, \( I_{\text{BODY}} = 50 \) instr. Cycles

<table>
<thead>
<tr>
<th>( T_{\text{isr}} = 10 \text{ ms} )</th>
<th>( T_{\text{isr}} = 1 \text{ ms} )</th>
<th>( T_{\text{isr}} = 100 \mu\text{s} )</th>
<th>( T_{\text{isr}} = 10 \mu\text{s} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01%</td>
<td>0.14%</td>
<td>1.43%</td>
<td>14.3%</td>
</tr>
</tbody>
</table>

Interrupt Vectors in Memory

- 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.

- Unhandled interrupts use _DefaultInterrupt
- Math Error Trap Vector
Our _DefaultInterrupt ISR

// Persistent storage for an error message, reported at reset by printResetCause.
static _PERSISTENT const char* sz_lastError;

// Persistent storage for a timeout error, to be reported if a watchdog reset occurs.
_PERSISTENT const char* sz_lastTimeoutError;

static _PERSISTENT INTTREGBITS INTTREGBITS_last;
// Make INTTREGBITS_last also accessible as a word. This is like
// uint16_t u16_INTTREGlast except that INTTREGBITS_last and
// u16_INTTREGlast refer to the same data.
#define u16_INTTREGlast BITS2WORD(INTTREGBITS_last)

_PERSISTENT error variables used for tracking errors across resets

_Lecture 6-14 Electrical & Computer Engineering – Microcomputers
Our _DefaultInterrupt ISR (continued)

void _ISR _DefaultInterrupt(void) {
    u16_INTTREGlast = INTTREG;
    reportError("Unhandled interrupt, ");
}

void reportError(const char* sz_errorMessage) {
    //ignore if a previous error has already been triggered
    if (sz_lastError == NULL) {
        sz_lastError = sz_errorMessage;
        asm ("reset");
    }
}

_DefaultInterrupt is the name of the default ISR used by the PIC24 compiler. This version saves the interrupt cause (INTTREG) then does a software reset

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.
Our _DefaultInterrupt ISR (continued)

// printResetCause() code fragment
void printResetCause(void) {
  // Code omitted for brevity
  // Function defined in c:\microchip\lib\common\pic24_util.c
  if (sz_lastError != NULL) {
    outString("Error trapped: ");
    outString(sz_lastError);
    if (u16_INTTREGlast != 0) {
      outString("Priority: ");
      outUint8(INTTREGBITS_last.ILR);
      outString("", Vector number: ");
      outUint8(INTTREGBITS_last.VECNUM);
    }
    outString("\n\n");
    sz_lastError = NULL;
    u16_INTTREGlast = 0;
  }
}

After reset, printResetCause() prints the error message

If the last rest was caused by an unhandled interrupt, print the priority (ILR) and the vector number (VECNUM)

Clear _PERSISTENT error variables

Output from the _DefaultInterrupt ISR

(a) Simplified test code (trap_test.c) to generate a Math Error Trap
int main (void) {
  volatile uint8 u8_zero;
  configBasic(HELLO_M80);
  while (1) {
    outString("Hit a key to start divide by zero test... ");
    inChar();
    outString("OK. Now dividing by zero. ");
    u8_zero = 0;
    u8_zero = 1/u8_zero;
    // generate divide-by-zero (Math Error) trap
  } // end while (1)
}

(b) Console Output
Reset cause: Power-on.
Device ID = 0x00008F10 (PIC24HJ32GP202), revision 0x0000F501 (A2)
Fast S2 Osc with PLL
trap_test.c, built on Jun 6 2008 at 10:17:53
hit a key to start divide by zero test... ok. Now dividing by zero.

Reset causes: Software Reset.
Error trapped: Unhandled interrupt. Priority: 0x3B, Vector number: 0x47
_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.
A User-Provided ISR

// Code in C, Interrupt Service Routine for MathError
void _ISRFAST _MathError (void) { // Clear the error and continue
_MATHERR = 0; // Clear the _MATHERR flag to signal trap is handled
RCOUNT = 0; // Clear the RCOUNT to break repeat loop
}

; In Assembly
[MathError:]
; Clear _MATHERR flag
bclr .b INTCON1, #4 ; _MATHERR=0
clr RCOUNT ; RCOUNT=0
retfie ; Return from interrupt

These ISRs just clear the _MATHERR interrupt flag and return. If the interrupt flag is not cleared, get stuck in an infinite interrupt loop.

_MATHERR address inserted into interrupt vector table (IVT)

Original file c:\microchip\chap9\trap_testHandled.c

Change Notification Interrupts

When enabled, triggers an interrupt when a change occurs on a pin.
Use Change Notification to wake from Sleep

#include "pic24_all.h"

//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
}

Use Change Notification to wake from Sleep (continued)

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 > 0
    _CNIE = 1;  // enable 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")
    }
}
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>SD1I</td>
<td>_SD1IR = 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.)

- Mapping outputs to RPx pins.

<table>
<thead>
<tr>
<th>Output Name</th>
<th>Function Name</th>
<th>Example Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Port Pin</td>
<td>NULL</td>
<td>_RPnR = 0;</td>
</tr>
<tr>
<td>UART1 Transmit</td>
<td>U1TX</td>
<td>_RPnR = 3;</td>
</tr>
<tr>
<td>UART1 Rdy. To Send</td>
<td>U1RTS</td>
<td>_RPnR = 4;</td>
</tr>
<tr>
<td>SPI1 Data Output</td>
<td>SD01</td>
<td>_RPnR = 7;</td>
</tr>
<tr>
<td>SPI1 Clock Output</td>
<td>SCK1OUT</td>
<td>_RPnR = 8;</td>
</tr>
<tr>
<td>SPI1 Slave Sel. Out.</td>
<td>SS1OUT</td>
<td>_RPnR = 9;</td>
</tr>
<tr>
<td>Output Compare 1</td>
<td>OC1</td>
<td>_RPnR = 10;</td>
</tr>
<tr>
<td>Output Compare 2</td>
<td>OC2</td>
<td>_RPnR = 19;</td>
</tr>
</tbody>
</table>
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 INTxEP bit (‘1’ is falling edge, ‘0’ is rising edge).

- On the PIC24HJ128GP502, INT1 and INT2 must be brought out to remappable pins (RPx).

- INT0 is assigned a fixed pin location.
Use INT1 to wake from Sleep mode

```c
#include "pic24_all.h"

// 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
}
```

Original file c:\microchip\chap9\int1_wakeup.c

Use INT1 to wake from Sleep mode (cont.)

```c
int main (void) {
  configBasic(HELLO_MSG);
  /** Configure the switch ***********/
  CONFIG_SW1();
  CONFIG_INT1_TO_RP(13);  // Map INT1 to RP13 (RB13 pin)
  /** Configure INT1 interrupt **/
  _INT1IF = 0;  // Clear the interrupt flag
  _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")
  }
}
```

Original file c:\microchip\chap9\int1_wakeup.c
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 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 T3IF for the interrupt flag.
T2IF Period

- The T2IF flag is set at the following period ($T_{t2if}$):
  \[ T_{t2if} = (PR2+1) \times PRE \times Tcy = (PR2+1) \times PRE/Fcy \]
- 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:
  \[ T_{t2if} = 65536 \times 1/Fcy \]
- We typically want to solve for PR2 given a desired Tt2if and given a PRE value:
  \[ PR2 = \left( T_{t2if} \times Fcy /PRE \right) - 1 \]

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>
<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

---

**Programmable the configuration register**

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

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

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:

\[
T2CONbits.TON = 1; \quad \text{//Set TON bit = 1, turn timer on}
\]
# Square Wave Generation

- Timer2 configured to generate an interrupt every 15 ms. An output pin is toggled in the ISR, so square wave has period of 30 ms.

```c
#include "pic24_all.h"

#define WAVEOUT _LATB2 // state of _RB2

inline void CONFIG_WAVEOUT()
{
    // Use RB2 for output
    CONFIG_RB2_AS_DIG_OUTPUT();
}

// Interrupt Service Routine for Timer2
void _ISRFAST _T2Interrupt (void)
{
    WAVEOUT = !WAVEOUT; // Toggle output
    _T2IF = 0; // Clear the timer interrupt bit
}
```

Original file c:\microchip\chap9\squarewave.c

---

# Square Wave Generation (cont.)

```c
#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 PRx + 1
    PR2 = msTo16Ticks (ISR_PERIOD, getTimerPrescale(T2CONbits)) - 1;
    TMR2 = 0; // Clear timer2
    _T2IF = 0; // Clear interrupt flag
    _T2IP = 1; // Choose a priority
    _T2IE = 1; // Enable the interrupt
    T2CONbits.TON = 1; // Turn on the timer
}
```

Original file c:\microchip\chap9\squarewave.c

---
Square Wave Generation (cont.)

```c
int main (void) {
    configBasic(HELLO_MSG);
CONFIG_WAVEOUT(); // PIO Config
configTimer2(); // TMR2 config
// Interrupt does work of generating the square wave
while (1) {
    doHeartbeat(); // Ensure that we are alive
} // End while (1)
}
```

After configuration, the ISR does the job of generating the square wave. The while(1) loop does nothing.

Original file c:\microchip\chap9\squarewave.c

Switch Sampling

```c
#define CONFIG_LED1() CONFIG_RB14_AS_DIG_OUTPUT()
#define LED1 _LATB14 //led1 state

/// Switch1 configuration
inline void CONFIG_SW1() {
    CONFIG_RB13_AS_DIG_INPUT(); // use RB13 for switch input
    ENABLE_RB13_PULLUP(); // enable the pullup
}

#define SW1_RAW _RB13 // raw switch value
#define SW1 u8_valueSW1 // switch state
#define SW1_PRESSED() (SW1==0) // switch test
#define SW1_RELEASED() (SW1==1) // switch test
```

Original file c:\microchip\chap9\ledtoggle_timer.c
Switch Sampling (cont.)

// debounced switch value that is set in the timer ISR
// any variable written by an ISR should be declared volatile
volatile uint8_t u8_valueSW1 = 1; // initially high

// Interrupt Service Routine for Timer3
void _ISRFAST _T3Interrupt (void) {
    u8_valueSW1 = SW1_RAW; // sample the switch
    _T3IF = 0; // clear the timer interrupt bit
}

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

Switch state is stored in a variable.

typedef enum {
    STATE_RESET = 0, STATE_WAIT_FOR_PRESS, STATE_WAIT_FOR_RELEASE
} STATE;

STATE e_LastState = STATE_RESET; // print message when state changes
void printNewState (STATE e_currentState) {
    if (e_LastState != e_currentState) {
        switch (e_currentState) {
            case STATE_WAIT_FOR_PRESS:
                outString("STATE_WAIT_FOR_PRESS\n"); break;
            case STATE_WAIT_FOR_RELEASE:
                outString("STATE_WAIT_FOR_RELEASE\n"); break;
            default: break;
        }
    }
    e_LastState = e_currentState; //remember last state
}
Switch Sampling (cont.)

int main (void) {
  STATE e_mystate;
  // Configuration not shown
  e_mystate = STATE_WAIT_FOR_PRESS;
  while (1) {
    printNewState(e_mystate); // debug message when state changes
    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; e_mystate = STATE_WAIT_FOR_PRESS;
        }
        break;
      default: e_mystate = STATE_WAIT_FOR_PRESS;
    } //end switch (e_mystate)
    doHeartbeat(); // ensure that we are alive
  } // end while (1)
}

Switch Sampling (cont.)

#define ISR_PERIOD 15  // in ms
void configTimer3(void) {
  // ensure that Timer2,3 configured as separate timers.
  T2CONbits.T2 = 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 = msTo16Ticks(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
}
Semaphores

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

Semaphores

- Semaphores are basic programming synchronization mechanisms
- They are commonly used for:
  - Establishing an order between two events
    - e.g. establishing an order (i.e. synchronizing) two pieces of code
  - Mutually exclusive access to a certain resource
- Semaphores may be binary (0/1), or counting
- For the purposes of our examples, a semaphore is a flag set by an ISR when an IO event occurs.
  - The main() code is generally responsible for clearing the flag.
Press & Release Semaphore

```c
typedef enum { STATE_WAIT_FOR_PRESS = 0, STATE_WAIT_FOR_RELEASE,
               } ISRSTATE;
volatile uint8_t u8_valueSW1 = 1; // Initially high
volatile uint8_t u8_pnrSW1 = 0;
void _ISRFAST _T3Interrupt (void) { // ISR for Timer3
  static ISRSTATE e_isrState = STATE_WAIT_FOR_PRESS;
  u8_valueSW1 = SW1_RAW; // sample the switch
  switch (e_isrState) {
    case STATE_WAIT_FOR_PRESS:
      if (SW1_PRESSED() && (u8_pnrSW1 == 0))
        e_isrState = STATE_WAIT_FOR_RELEASE;
      break;
    case STATE_WAIT_FOR_RELEASE:
      if (SW1_RELEASED()) {
        e_isrState = STATE_WAIT_FOR_PRESS;
        u8_pnrSW1 = 1; // set press & release semaphore
        break;
      }
    default: e_isrState = STATE_WAIT_FOR_RELEASE;
  }
  _T3IF = 0; // clear timer interrupt bit
}
```

Do not process another press & release until the last has been handled

ISR is now a state machine!

Press & Release Semaphore main() code

```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); // debug message when state changes
    switch (e_mystate) {
      case STATE_WAIT_FOR_PNR1:
        LED1 = 0; // turn off the LED
        if (u8_pnrSW1) {
          u8_pnrSW1 = 0; // clear semaphore
          e_mystate = STATE_WAIT_FOR_PNR2;
        }
        break;
    }
  }
```
Press&Release Semaphore main() code

```c
case STATE_WAIT_FOR_PNR2:
    LED1 = 1;  // turn on the LED
    if (u8_pnrSW1)
        u8_pnrSW1 = 0;  // clear semaphore
    // decide where to go
    if (SW2) e_mystate = STATE_BLINK;
    else e_mystate = STATE_WAIT_FOR_PNR1;
    break;

case STATE_BLINK:
    LED1 = !LED1;  // blink while not pressed
    DELAY_MS(100);  // blink delay
    if (SW1_PRESSED()) e_mystate = STATE_WAIT_FOR_RELEASE3;
    break;

case STATE_WAIT_FOR_RELEASE3:
    LED1 = 1;  // Freeze LED1 at 1
    // use u8_pnrSW1 instead of SW1_RELEASED because
    // u8_pnrSW1 is set on a release, and must be cleared.
    if (u8_pnrSW1)
        u8_pnrSW1 = 0;
    e_mystate = STATE_WAIT_FOR_PNR1;
    break;

default:
    e_mystate = STATE_WAIT_FOR_PNR1;
    } // end switch(e_mystate)
doHeartbeat();  // ensure that we are alive
    } // end while (1)

Differences from original solution:

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 one right choice.
- Golden Rules:
  - The ISR should do its work as fast as possible.
  - Do not put long (or any) software delays into an ISR.
  - An ISR should never wait for I/O, the I/O event should trigger the ISR!
  - 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
- Interrupt-driven LED/Switch IO using periodic interrupts