Chapter 5 - Interrupts

Handling Asynchronous Events
Interrupts and Exceptions

Table 5.1: Exception vectors table of the PIC32 architecture.

<table>
<thead>
<tr>
<th>Exception Source</th>
<th>Memory Region</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset and NMI</td>
<td>Program</td>
<td>Normal reset and nonmaskable interrupt entry point.</td>
</tr>
<tr>
<td>On-chip debug</td>
<td>Program</td>
<td>Used by the ICD and EJTAG interfaces to enable in circuit debugging features.</td>
</tr>
<tr>
<td>Cache error</td>
<td>RAM or Program</td>
<td>Error condition specific to the cache mechanism.</td>
</tr>
<tr>
<td>TLB refill</td>
<td>RAM or Program</td>
<td>Not used on PIC32 because a fixed address translation scheme (FMT) is used in place of a full MMU.</td>
</tr>
<tr>
<td>General exception</td>
<td>RAM or Program</td>
<td>All other types of exceptions.</td>
</tr>
<tr>
<td>Interrupt</td>
<td>RAM or Program</td>
<td>The proper interrupt vector.</td>
</tr>
</tbody>
</table>

Interrupt – an internal or external event that requires “fast” attention from CPU.

MIPS core of the PIC32 treats all interrupts as *exceptions* (see Table above).

PIC32 can manage up to 64 interrupts. An interrupt service routine (ISR), a.k.a. interrupt handler (IH) is a special C function to be executed in response to an interrupt. An ISR or IH can be asynchronous to the main program execution to minimize (or more precisely, hide) the *interrupt latency*.

Interrupts are the “friendliest” kind of exceptions, because application won’t crash.
Interrupt Service Routines

- Interrupt service routines are not supposed to return any value (i.e. type void). Why?
- Also, no parameter can be passed to the ISR (therefore, parameter list is void). Why?
- ISRs cannot be called directly by any other (user) functions in the program. Why?
- Normally, an ISR should not invoke any other (user) function, including another ISR, either.
- PIC32’s MIPS core uses the crt0 code to put the interrupt vectors in RAM or PROG MEM.
Sources of Interrupt Events

Among the external sources for the PIC32MX, there are:
- 5 x external pins with level trigger detection
- 22 x external pins connected to the Change Notification (CN) module
- 5 x Input Capture (IC) modules
- 5 x Output Compare (OC) modules
- 2 x serial port interfaces (with UARTs)
- 4 x synchronous serial interfaces (SPI and I²C)
- 1 x Parallel Master Port
- (More on all these modules and interfaces later.)

Among the internal sources for the PIC32MX3, there are:
- 1 x 32-bit internal (core) timer
- 5 x 16-bit internal (peripheral) timers
- 1 x analog-to-digital converter (ADC)
- 1 x Analog Comparator (AC) module
- 1 x real-time clock and calendar
- 1 x flash controller
- 1 x fail-safe clock monitor
- 2 x software (programmable) interrupts
- 4 x direct memory access (DMA) channels

The interrupt control module can handle up to 96 independent events.
Sources of Interrupt Events

- UART (Universal Asynchronous Rx/Tx), for example, generates 3 kinds of interrupts:
  - Receipt of new data in the receive buffer
  - Transmit of data complete - transmit buffer is empty
  - Error occurred, need to re-establish communication

- 96 independent events can be managed. But only up to 64 vectors are allowed by the MIPS core of PIC32 due to hardware resources.
  - A solution has been to design the processor to group all the interrupts belonging to the same peripheral into the peripheral’s assigned interrupt vector.
## Interrupt Vector Table (PIC32FJ512MX360L)

<table>
<thead>
<tr>
<th>Vector Number</th>
<th>Vector Symbol</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>_EXTERNAL_4 VECTOR</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>_TIMER_5 VECTOR</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>_INPUT_CAPTURE_5 VECTOR</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>_OUTPUT_COMPARE_5 VECTOR</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>_SPI1 VECTOR</td>
<td>Groups all three SPI1 interrupts.</td>
</tr>
<tr>
<td>24</td>
<td>_UART1 VECTOR</td>
<td>Groups all three UART1 interrupts.</td>
</tr>
<tr>
<td>25</td>
<td>_I2C1 VECTOR</td>
<td>Groups all I2C1 interrupts.</td>
</tr>
<tr>
<td>26</td>
<td>_CHANGE_NOTICE VECTOR</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>_ADC VECTOR</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>_PMP VECTOR</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>_COMPARATOR_1 VECTOR</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>_COMPARATOR_2 VECTOR</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>_SPI2 VECTOR</td>
<td>Groups all three SPI2 interrupts.</td>
</tr>
<tr>
<td>32</td>
<td>_UART2 VECTOR</td>
<td>Groups all three UART2 interrupts.</td>
</tr>
<tr>
<td>33</td>
<td>_I2C2 VECTOR</td>
<td>Groups all I2C2 interrupts.</td>
</tr>
<tr>
<td>34</td>
<td>_FAIL_SAFE_MONITOR VECTOR</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>_RTCC VECTOR</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>_DMA0 VECTOR</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>_DMA1 VECTOR</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>_DMA2 VECTOR</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>_DMA3 VECTOR</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>_FCE VECTOR</td>
<td></td>
</tr>
</tbody>
</table>
Interrupt Flags & Priorities (7 Bits)

- **Interrupt Enable bit** (represented with the name of the interrupt source peripheral followed by the suffix -IE):
  - When cleared, the corresponding trigger event is prevented from generating interrupts.
  - When set, it allows the interrupt to be generated and serviced.
  - By default, all interrupt sources are disabled (i.e. cleared) at power-on.

- **Interrupt Flag bit** (with a suffix -IF) is set each time the specific trigger event is activated, independently of the status of the Enable bit.
  - Once set, it must be cleared by the user. In other words, it must be cleared before exiting the ISR. Otherwise, the same ISR will be immediately invoked again!

- **Group-Priority Level bits** (with a suffix -IP). Interrupts can have up to 7 levels of priority (from ipl1 to ipl7). Should two interrupt events occur at the same time, the higher priority event will be served first.
  - 3 bits are used to encode the priority level for each interrupt source.
  - At any given time, the PIC32 execution priority-level value is kept in the MIPS core status-register. Interrupts with a priority level lower than the currently kept value will be ignored.
  - At power on, all interrupt sources are assigned a default level of ipl0, disabling all interrupts.

- **Sub-Priority Level bits**. 2 bits are allocated to define 4 more levels of priority within a set priority group. If two events of the same group-priority level occur simultaneously, the event with the higher sub-priority goes first.
  - However, once an interrupt of a given priority group is selected, any following interrupts of the same level (even if of higher sub-priority) will be ignored until the current interrupt (flag) has been cleared.
# Natural Order Table (for Final IPL Arbitration)

<table>
<thead>
<tr>
<th>Natural Order</th>
<th>Macro Abbreviation</th>
<th>IRQ Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (highest)</td>
<td>CT</td>
<td>CORE_TIMER_IRQ</td>
<td>Core Timer Interrupt</td>
</tr>
<tr>
<td>1</td>
<td>CS0</td>
<td>_CORE_SOFTWARE_0_IRQ</td>
<td>Core Software Interrupt 0</td>
</tr>
<tr>
<td>2</td>
<td>CS1</td>
<td>_CORE_SOFTWARE_1_IRQ</td>
<td>Core Software Interrupt 1</td>
</tr>
<tr>
<td>3</td>
<td>INT0</td>
<td>EXTERNAL_0_IRQ</td>
<td>External Interrupt 0</td>
</tr>
<tr>
<td>4</td>
<td>T1</td>
<td>TIMER_1_IRQ</td>
<td>Timer 1 Interrupt</td>
</tr>
<tr>
<td>5</td>
<td>IC1</td>
<td>INPUT_CAPTURE_1_IRQ</td>
<td>Input Capture 1 Interrupt</td>
</tr>
<tr>
<td>6</td>
<td>OC1</td>
<td>_OUTPUT_COMPARE_1_IRQ</td>
<td>Output Compare 1 Interrupt</td>
</tr>
<tr>
<td>7</td>
<td>INT1</td>
<td>_EXTERNAL_1_IRQ</td>
<td>External Interrupt 1</td>
</tr>
<tr>
<td>8</td>
<td>T2</td>
<td>TIMER_2_IRQ</td>
<td>Timer 2 Interrupt</td>
</tr>
<tr>
<td>9</td>
<td>IC2</td>
<td>_INPUT_CAPTURE_2_IRQ</td>
<td>Input Capture 2 Interrupt</td>
</tr>
<tr>
<td>10</td>
<td>OC2</td>
<td>_OUTPUT_COMPARE_2_IRQ</td>
<td>Output Compare 2 Interrupt</td>
</tr>
<tr>
<td>11</td>
<td>INT2</td>
<td>_EXTERNAL_2_IRQ</td>
<td>External Interrupt 2</td>
</tr>
<tr>
<td>12</td>
<td>T3</td>
<td>TIMER_3_IRQ</td>
<td>Timer 3 Interrupt</td>
</tr>
<tr>
<td>13</td>
<td>IC3</td>
<td>_INPUT_CAPTURE_3_IRQ</td>
<td>Input Capture 3 Interrupt</td>
</tr>
<tr>
<td>14</td>
<td>OC3</td>
<td>_OUTPUT_COMPARE_3_IRQ</td>
<td>Output Compare 3 Interrupt</td>
</tr>
<tr>
<td>15</td>
<td>INT3</td>
<td>_EXTERNAL_3_IRQ</td>
<td>External Interrupt 3</td>
</tr>
<tr>
<td>16</td>
<td>T4</td>
<td>TIMER_4_IRQ</td>
<td>Timer 4 Interrupt</td>
</tr>
<tr>
<td>17</td>
<td>IC4</td>
<td>_INPUT_CAPTURE_4_IRQ</td>
<td>Input Capture 4 Interrupt</td>
</tr>
<tr>
<td>18</td>
<td>OC4</td>
<td>_OUTPUT_COMPARE_4_IRQ</td>
<td>Output Compare 4 Interrupt</td>
</tr>
<tr>
<td>19</td>
<td>INT4</td>
<td>_EXTERNAL_4_IRQ</td>
<td>External Interrupt 4</td>
</tr>
<tr>
<td>20</td>
<td>T5</td>
<td>TIMER_5_IRQ</td>
<td>Timer 5 Interrupt</td>
</tr>
<tr>
<td>21</td>
<td>IC5</td>
<td>_INPUT_CAPTURE_5_IRQ</td>
<td>Input Capture 5 Interrupt</td>
</tr>
<tr>
<td>22</td>
<td>OC5</td>
<td>_OUTPUT_COMPARE_5_IRQ</td>
<td>Output Compare 5 Interrupt</td>
</tr>
<tr>
<td>23</td>
<td>SPI1E</td>
<td>SPI1_ERR_IRQ</td>
<td>SPI 1 Fault</td>
</tr>
<tr>
<td>24</td>
<td>SPI1TX</td>
<td>SPI1_TX_IRQ</td>
<td>SPI 1 Transfer Done</td>
</tr>
<tr>
<td>25</td>
<td>SPI1RX</td>
<td>SPI1_RX_IRQ</td>
<td>SPI 1 Receiver Done</td>
</tr>
<tr>
<td>26</td>
<td>U1E</td>
<td>UART1_ERR_IRQ</td>
<td>UART 1 Error</td>
</tr>
<tr>
<td>27</td>
<td>U1RX</td>
<td>UART1_RX_IRQ</td>
<td>UART 1 Receiver</td>
</tr>
<tr>
<td>28</td>
<td>U1TX</td>
<td>UART1_TX_IRQ</td>
<td>UART 1 Transmitter</td>
</tr>
<tr>
<td>29</td>
<td>I2C1B</td>
<td>I2C1_BUS_IRQ</td>
<td>I2C 1 Bus Collision Event</td>
</tr>
<tr>
<td>30</td>
<td>I2C1S</td>
<td>I2C1_SLAVE_IRQ</td>
<td>I2C 1 Slave Event</td>
</tr>
<tr>
<td>31</td>
<td>I2C1M</td>
<td>I2C1_MASTER_IRQ</td>
<td>I2C 1 Master Event</td>
</tr>
<tr>
<td>32</td>
<td>CN</td>
<td>CHANGE_NOTICE_IRQ</td>
<td>Input Change Interrupt</td>
</tr>
<tr>
<td>33</td>
<td>AD1</td>
<td>ADC_IRQ</td>
<td>ADC Convert Done</td>
</tr>
<tr>
<td>34</td>
<td>PMP</td>
<td>_PMP_IRQ</td>
<td>Parallel Port Interrupt</td>
</tr>
<tr>
<td>35</td>
<td>CMP1</td>
<td>_COMPARATOR_1_IRQ</td>
<td>Comparator 1 Interrupt</td>
</tr>
<tr>
<td>36</td>
<td>CMP2</td>
<td>_COMPARATOR_2_IRQ</td>
<td>Comparator 2 Interrupt</td>
</tr>
<tr>
<td>37</td>
<td>SPI2E</td>
<td>SPI2_ERR_IRQ</td>
<td>SPI 2 Fault</td>
</tr>
<tr>
<td>38</td>
<td>SPI2TX</td>
<td>SPI2_TX_IRQ</td>
<td>SPI 2 Transfer Done</td>
</tr>
<tr>
<td>39</td>
<td>SPI2RX</td>
<td>SPI2_RX_IRQ</td>
<td>SPI 2 Receiver Done</td>
</tr>
<tr>
<td>40</td>
<td>U2E</td>
<td>UART2_ERR_IRQ</td>
<td>UART 2 Error</td>
</tr>
<tr>
<td>41</td>
<td>U2RX</td>
<td>UART2_RX_IRQ</td>
<td>UART 2 Receiver</td>
</tr>
<tr>
<td>42</td>
<td>U2TX</td>
<td>UART2_TX_IRQ</td>
<td>UART 2 Transmitter</td>
</tr>
<tr>
<td>43</td>
<td>I2C2B</td>
<td>I2C2_BUS_IRQ</td>
<td>I2C 2 Bus Collision Event</td>
</tr>
<tr>
<td>44</td>
<td>I2C2S</td>
<td>I2C2_SLAVE_IRQ</td>
<td>I2C 2 Slave Event</td>
</tr>
<tr>
<td>45</td>
<td>I2C2M</td>
<td>I2C2_MASTER_IRQ</td>
<td>I2C 2 Master Event</td>
</tr>
<tr>
<td>46</td>
<td>FSCM</td>
<td>FAIL_SAFE_MONITOR_IRQ</td>
<td>Fail-safe Clock Monitor Interrupt</td>
</tr>
<tr>
<td>47</td>
<td>RTCC</td>
<td>RTCC_IRQ</td>
<td>Real Time Clock Interrupt</td>
</tr>
<tr>
<td>48</td>
<td>DMA0</td>
<td>DMA0_IRQ</td>
<td>DMA Channel 0 Interrupt</td>
</tr>
<tr>
<td>49</td>
<td>DMA1</td>
<td>DMA1_IRQ</td>
<td>DMA Channel 1 Interrupt</td>
</tr>
<tr>
<td>50</td>
<td>DMA2</td>
<td>DMA2_IRQ</td>
<td>DMA Channel 2 Interrupt</td>
</tr>
<tr>
<td>51</td>
<td>DMA3</td>
<td>DMA3_IRQ</td>
<td>DMA Channel 3 Interrupt</td>
</tr>
<tr>
<td>52</td>
<td>FCE</td>
<td>_FLASH_CONTROL_IRQ</td>
<td>Flash Control Event</td>
</tr>
</tbody>
</table>

Di Jasio - Programming 32-bit Microcontrollers in C
We can declare an *Interrupt handler* function in C using one of the following syntax options:

- **Using the attributes syntax:**
  ```c
  void __attribute__((interrupt(ipl1), vector(0))) InterruptHandler(void) // name of IH
  {
    // your ISR code here...
  }
  ```

- **Using the pragma syntax:**
  ```c
  #pragma interrupt InterruptHandler ipl1 vector 0
  void InterruptHandler(void) // name of IH
  {
    // your ISR code here...
  }
  ```
ISR Syntax Similar to a C Macro

- A more compact alternative using something similar to the pre-defined macro:
  ```
  __ISR(v, ipl) // similar to a macro definition
  ```

- Implementation example:
  ```c
  void __ISR(0, ipl1) InterruptHandler (void)
  {
    // your ISR code here...
  }
  ```

- Prologue and epilogue code sequences will be inserted automatically by the compiler to provide safe context save and restore.
Interrupt Management Library

- To manage all interrupt sources, 2 libraries are provided.
- One is the small `<int.h>` provided as part of the standard PIC32 toolset. (int = interrupt).
- The other is the big `<plib.h>` provided as part of the peripherals support library.
- With either of these 2 libraries, we can have available to us a good number of functions and macros.
- Macros are prefixed with the lower case “m”.
<int.h> Support Functions & Macros

- **INTEnableSystemSingleVectoredInt( );**
  - This function follows a precise sequence of initialization of the interrupt control module (prescribed in the device datasheet) to enable the basic interrupt management mode of the PIC32.

- **INTEnableSystemMultiVectoredInt( );**
  - Same as the above, but this function enables the multivectored interrupt management mode of the PIC32.

- **mXXSetIntPriority(x);**
  - This is actually just a placeholder for a long list of similar macros (replace the XX with the interrupt Macro Abbreviations from the Interrupt Vector Table to obtain the specific macro name).
  - Assigns a given priority level \( x \in [0,7] \) to the specific interrupt source identified by the macro name.

- **mXXClearIntFlag( );**
  - This represents an entire class of macros that allow us to clear the interrupt flag (the -IF bit) of the chosen interrupt source.
The “single.c” Example

/*
** With Single Interrupt Vector: Use an ISR to service Timer2.
** Contrast this with the library-free code on P. 92 of Di Jasio.
*/
#include <p32xxxx.h>
#include <plib.h> // Library <plib.h> contains <int.h>.
int count; // Global counter gets zero’d by crt0 code.

void __ISR(0, ipl1) InterruptHandler(void)
{
    count++; // ipl7 will get a different set of prologue/epilogue.
    mT2ClearIntFlag(); // ISR increments count for every T2 period of 16.
    mT2ClearIntFlag(); // ISR must clear the interrupt flag before returning.
} // InterruptHandler

main()
{
    // 1. init timer T2
    PR2 = 15; // Same as setting PR2 = 0xf.
    T2CON = 0x8030; // How is Timer2 configured?

    // 2. init the interrupt source and IM
    mT2SetIntPriority(1); // Priority here must match that of the ISR, i.e. ipl1 or 1.
    INTEnableSystemSingleVectoredInt(); // Use single-vector interrupt management.
    mT2IntEnable(1); // Interrupt source is enabled the last.

    // 3. main infinite loop
    while(1); // What will happen if Timer2 interrupts happen too often?
} // main
Setting a Breakpoint in “single.c”
Managing Multiple Interrupts

- A lower-priority (ipl1) interrupt occurred, and is now being served, i.e. an ISR is running.
- What should the PIC32 do if a higher-priority interrupt arrives at this time?
- It will require immediate attention, and will interrupt the currently running ipl1-ISR.
- This case is called “nesting interrupt calls”.
- Done by manually re-enabling interrupts upon entry into the interrupt handler.
- The MIPS instruction “ei” supports this.
The “nesting.c” Example

/*
 ** With Single Vector Interrupt Nesting: One ISR for both T2 and T3 interrupts.
 */
#include <p32xxxx.h>
#include <plib.h>
int count;

void __ISR(0, ipl1) InterruptHandler(void) // __ISR_SINGLE() for later C32 compilers?
{
  // 1. re-enable interrupts immediately (nesting) upon entry
  asm("ei");
  // 2. check and serve the highest priority interrupt first
  if (mT3GetIntFlag())
  {
    count++;
    // clear the flag and exit
    mT3ClearIntFlag();
  } // Timer3
  // 3. check and serve the lower priority
  else if (mT2GetIntFlag())
  {
    // spend a long time here!
    while(1);
    // before clearing the flag and exiting
    mT2ClearIntFlag();
  } // Timer2
} // __ISR

// Wait! Where is the main( ) code?

Di Jasio - Programming 32-bit Microcontrollers in C
The "nesting.c" Example

main()
{
    // 4. init timers 2 and 3
    PR3 = 20;
    PR2 = 15;
    T3CON = 0x8030;
    T2CON = 0x8030;

    // 5. init interrupt sources and IM
    mT2SetIntPriority(1);           // ipl1 for Timer2, lower priority for Timer2
    mT3SetIntPriority(3);           // ipl3 for Timer3, higher priority for Timer3
    INTEnableSystemSingleVectoredInt(); // Use single-vector interrupt management.
    mT2IntEnable(1);
    mT3IntEnable(1);

    // main loop
    while(1);
} // main
Multiple Interrupts – Warning!

- In practice, you should never have more than 2 levels of such nesting on an MCU.
- If this does happen, think the word DESIGN.
- Is your priority scheme well designed?
- Are your interrupt handlers too convoluted?
- Or… do you have both of these problems?
- Our current design is to funnel all interrupts through a single ISR (i.e. 1 interrupt vector).
- This can create a noticeable latency when responding to a critical interrupt event.
Multi-vectored Interrupts

- To minimize interrupt latency, especially when high-priority interrupts arrive, PIC32 uses (both) vectored interrupts and multiple register sets to improve performance.

- PIC32MX uses a 64-vector table and 2 sets of 32 registers for ISR switching.

- Interrupts that belong to the same peripheral device are grouped into the same vector.

- This eliminates the need to sequentially test (more like look up) all the interrupt sources.

- What about the second set of 32 registers?
Multi-vectored Interrupts

- The second 32-register set frees us from the need to save the current context on the stack – a task that involves the *prologue*.

- But obviously, you see the catch here – with only 2 sets of registers, can’t do more than one switch (a.k.a. swap) at the same time.

- PIC32 decides to allow the use of the second “swap” register set ONLY when interrupts of *ipl7*, i.e. the highest priority, hit.

- Hence, rewrite the current code as follows… into 2-ISR code (p. 101 of Di Jasio).
Multi-vectored Interrupts

- Assigning a separate vector (that points to a separate ISR) to each group of interrupt sources, thus the term multi-vectored.
- You can see how this helps reduce the cost of testing to find the right interrupt source.
- Of course we can still use nesting to “start” a lower-priority ISR that will give way to higher-priority interrupts, just like before.
- In general, multi-vectored interrupt design is better and simpler, in most cases.
The “multiple.c” Example

/*
** With Multiple Vector Interrupt: 2 ISRs, one for Timer3, one for Timer2
*/
#include <p32xxxx.h>
#include <plib.h>

int count;

void __ISR(_TIMER_3_VECTOR, ipl7) T3InterruptHandler(void)
{
    // 1. Timer3 (T3) handler is responsible for incrementing count
    count++;

    // 2. clear the T3 flag and exit
    mT3ClearIntFlag();
} // T3 ISR

void __ISR(_TIMER_2_VECTOR, ipl1) T2InterruptHandler(void)
{
    // 3. re-enable interrupts immediately (nesting) - still need this “ei” now?
    asm("ei"); // remove and see what happens

    // 4. T2 handler code here
    while(1);

    // 5. clear the T2 flag and exit
    mT2ClearIntFlag();
} // T2 ISR
The “multiple.c” Example

main()
{
    // 5. init timers
    PR3 = 20;
    PR2 = 15;
    T3CON = 0x8030;
    T2CON = 0x8030;

    // 6. init interrupt sources and IM
    mT2SetIntPriority(1);
    mT3SetIntPriority(7);
    INTEnableSystemMultiVectoredInt();
    mT2IntEnable( );
    mT3IntEnable( );

    // 7. main loop
    while(1);
} // main
Multi-vectored Interrupts

- What difference do you observe now?
- Did you notice the priority level for T3?
- Note: only ipl7 gets the “swap” registers.
- So... what does that mean for T3’s ISR?
- What if your main() grows in size?
- Think *prologue*, i.e. the start code!
- When you are at ipl7, you save on *prologue*.
- Check the assembly code for the 2 ISRs.
- Good material for Lab Project No. 3 😊
- See p. 103 of Di Jasio for more details.
A Real-Time Clock?

- Use Timer1 to keep track of 1/10 seconds, seconds, and minutes.
- Use lower 8 pins of PORTA as binary display.
- For now, assume PIC32 runs at 16MHz with a peripheral clock. Then, use 1:64, i.e. 0b10 for the T1CON field TCKPS<1:0>.
- We want 0.1 second between interrupts.
- Set PR1 = 25000-1, period register value for Timer1, since 0.1s = (1/16MHz) * 64 * 25000.
- Disable JTAG, TRISA = 0xff00, PORTA = sec
A Real-Time Clock?

```c
/*
 ** A Real-Time Clock Using Single Vector Interrupts and 1 ISR for T1
 ** Source code file: clock.c
*/
#include <p32xxxx.h>
#include <plib.h>

int dSec = 0;       // counter for 1/10 second
int Sec = 0;        // counter for seconds
int Min = 0;        // counter for minutes

// 1. Timer1 interrupt service routine
void __ISR(0, ipl1) T1Interrupt(void)
{
    // 1.1 increment the tens of a second counter
    dSec++;
    
    if ( dSec > 9)
    {
        dSec = 0;
        Sec++;
        if ( Sec > 59)
        {
            Sec = 0;
            Min++;
            if ( Min > 59)
            {
                Min = 0;
            }
        }
    }
    // seconds

    // 1.2 clear the interrupt flag
    mT1ClearIntFlag();
} //T1 ISR
```

Di Jasio - Programming 32-bit Microcontrollers in C
main()
{
    // 2.1 init I/Os
    DDPCONbits.JTAGEN = 0; // disable JTAG port
    TRISA = 0xff00; // set PORTA pins 0..7 as output

    // 2.2 configure Timer1 module
    PR1 = 25000-1; // set the period register, internal clock
    T1CON = 0x8020; // enabled, prescaler 1:64, internal clock

    // 2.3 init interrupts
    mT1SetIntPriority(1);
    mT1ClearIntFlag();
    INTEnableSystemSingleVectoredInt();
    mT1IntEnable( 1); // why here? not in an ISR yet

    // 2.4. main loop
    while(1)
    {
        // your main code here
        PORTA = Sec;
    } // while
} // main
The "clock.c" Simulation
A Real-Time Clock? – Notes

- ISR was hit, after PR1 counted from 0 thru 24999, i.e. \(25000 \times 64 = 1.6M\) cycles passed.
- Look at the Stopwatch, why is it not exactly 100ms (0.1s) elapsed, given \(1.6M \times (1/16M)\)?
- Think *prologue* (i.e. init) “clock cycles”!
- \(0x00100401\) (Status SFR content) =
- \(0000 0000 0001 0000 0000 0001 0100 0000 0001\)
- \(\text{Status}<15:10> = 1 \implies \text{ipl is set to 1 for Timer1 inside the T1 ISR. Upon exit, it resets to 0.}\)
Secondary Oscillator

- Our real-time clock used the high-frequency internal clock as input to Timer1.
- We could have used Secondary Oscillator, a low-frequency device, for Timer1.
- This oscillator uses an inexpensive 32768-Hz crystal to operate in low-frequency.
- The oscillator is more power efficient.
- Secondary oscillator is very important in the low-power modes – especially when the main clock is disabled (when PIC32 hibernates).
How do we change the Timer1 source from the high-frequency internal clock to the secondary oscillator?

- Set the period register for the low-frequency oscillator: PR1 = 32768 – 1;
- Change the Timer1 configuration SFR as:
  - T1CON = 0x8002; // Crucial! TCS<1> = 1
- This enables Timer1, prescaler = 1:1, to use the secondary oscillator as source.
- But S.O. is not supported by MPLAB SIM 😞
Using the Secondary Oscillator

// 1. Timer1 interrupt service routine
void __ISR(0, ipl1) T1Interrupt(void)
{
    // 1.1
    Sec++;
    if (Sec > 59)
    {
        Sec = 0;
        Min++;
        if (Min > 59)
            Min = 0;
    } // minutes
    // 1.2
    mT1ClearIntFlag();
} // T1 ISR

- Change the period register PR1 to generate one interrupt every 32,768 cycles
  PR1 = 32768-1; // set the period register
- Change the Timer1 configuration SFR (the prescaler is not required anymore)
  T1CON = 0x8002; // prescaler 1:1, use 2nd oscillator
RTCC Module in PIC32MX

- PIC32MX already contains a Real-Time Clock and Calendar (RTCC) module.
- Once initialized, the RTCC module can use its Alarm function to generate an interrupt at a given year/month/day/hour/min/sec combo.
- To use the RTCC module, both the ISR and the main() need modification. More changes are needed for main() than for the ISR.
- The code has an incomplete main() for user code addition (e.g. I/O, menu, etc.)
Using the RTCC module

main()
{
    // 2.1 init I/Os
    DDPCONbits.JTAGEN = 0; // disable JTAG port
    TRISA = 0xff00; // set PORTA LSB as output
    // 2.2 configure RTCC module
    RtccInit(); // inits the RTCC
    // set present time
    rtccTime tm; tm.sec=0x15; tm.min=0x30; tm.hour=01;
    // set present date
    rtccDate dt;
    dt.wday=0; dt.mday=0x15; dt.mon=0x10; dt.year=0x07;
    RtccSetTimeDate(tm.l, dt.l);
    // set desired alarm to Feb 29th (rings every 4 years!)
    dt.wday=0; dt.mday=0x29; dt.mon=0x2;
    RtccSetAlarmTimeDate(tm.l, dt.l);
    // 2.2 init interrupts,
    mRTCCSetIntPriority( 1);
    mRTCCClearIntFlag();
    INTEnableSystemSingleVectoredInt();
    mRTCCIntEnable( 1);
    // 2.3. main loop
    while(1)
    {
        // user code here
        // ...
    } // while
} // main
Using the RTCC module

// 1. RTCC interrupt service routine
void __ISR(0, ipl1) RTCCInterrupt(void)
{
   // 1.1 your code here, will be executed only once a year
   // or once every 365x24x60x60x16,000,000 PIC32 cycles
   // that is once every 504,576,000,000,000 seconds?
   // (assuming a PIC32 running at 16MHz)

   // 1.2 clear the interrupt flag
   mRTCCClearIntFlag();
} // RTCCInterrupt ISR