SIMULATION SESSION

INTERRUPTS, CORRUPTION AND RACE CONDITIONS

INTRODUCTION

In this session you will study a very important phenomena – **race conditions** and **data corruption**. It must be emphasized that **this is a very important topic**, especially if working with safety critical systems.

The problem of race conditions is (in)famously difficult to demonstrate as they are 'stochastic' (occur at random and with low probability). For illustrative purposes, the examples in this tutorial are somewhat contrived to illustrate what would otherwise be rare and difficult to observe software errors.

At the end of this session, you should be able to do the following:

INTENDED LEARNING OUTCOMES

- Program and implement two hardware timer interrupts to interrupt execution at regular intervals
- Enable or disable nested interrupts
- Identify a critical section
- Describe a race condition
- Avoid corruption of shared variables through temporarily elevation of interrupt priority
- Describe meaning and identify the appropriate use of the keyword **volatile**

TASK 1 - USING TWO TIMER INTERRUPTS TO DEMONSTRATE NESTING

The PIC24 has 5 hardware timers that can run independently of the main thread of code. We are going to use **TIMER1** AND **TIMER2** to illustrate the concept of nested interrupts.

In the first example, we shall look at configuring and using TWO timer **interrupts**. These two timers will interrupt at the same frequency, but will be slightly out of phase (in time)[1].

Figure 1 illustrates the concept of two **nested** interrupts. In this exercise, ISR1 represents TIMER1 and ISR2 represents TIMER2.

- TIMER1 has the same period register and clock speed as TIMER2
- TIMER1 will start a few clock cycles **ahead** of TIMER2 (so its interrupt will be called first)
- TIMER2 has a **higher** priority than TIMER1
    - Therefore **TIMER2 will interrupt TIMER1**.

[1] Refer to the lecture notes on how the timer is programmed if you are unsure, or look at the source code in task 1.

NOTE - for all precise timing, you MUST use the MPLAB internal simulator AND NOT the VSM debugger

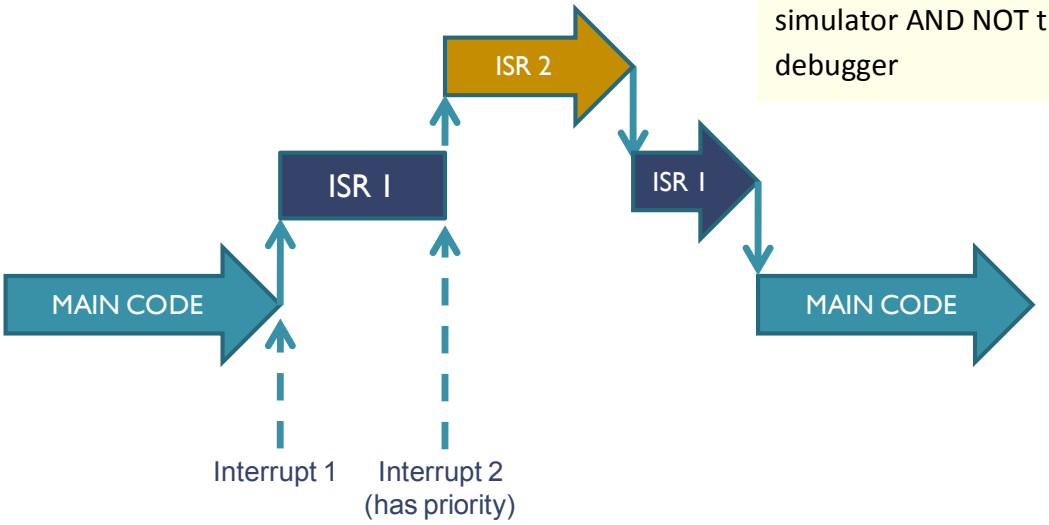

Figure 1 - Nested interrupts

1. Open the project in the folder Task 1
   - Study this code carefully and read all the comments.
   - Run the code to see what it does.
   - **Now set the debugger to the internal simulator (not the VSM)**
   - If not already done so, add the **Interrupt Flag Status** (IFS) **register IFS0** to the watch window – ensure the format is in binary (see Figure 2)
   - Note that TIMER1 is setup almost identically to TIMER2 (same pre-scale etc.)
   - Note that TIMER2 has a higher priority than TIMER1
   - Note the starting values of TIMER1 and TIMER2 – I've given one a slight head start!
2. Build the code
3. Set a break point at the start of BOTH timer service routines and run code.
   - The program should enter the **_T1Interrupt()** function
4. STEP through the code
   - The TIMER1 ISR should be interrupted by TIMER2

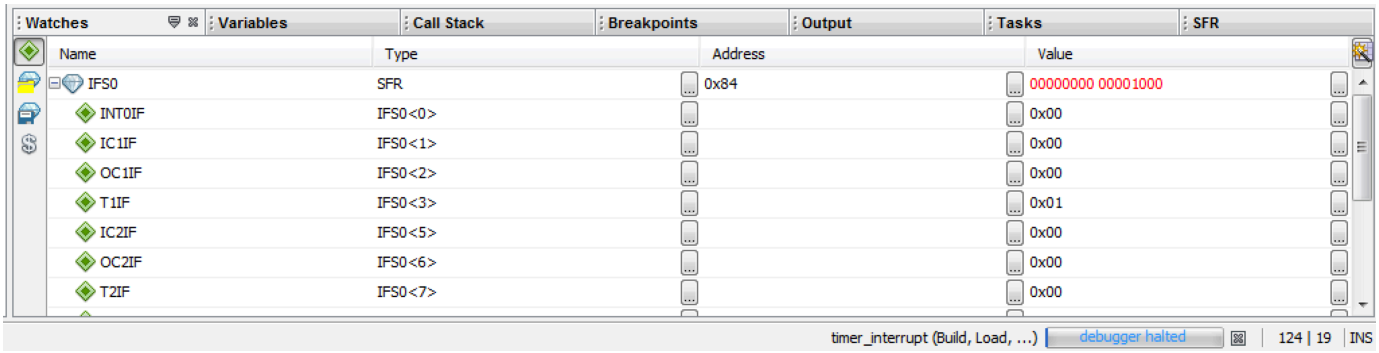

Figure 2 - Watching the Interrupt Flag Status Register IFS0

TASKS
Q. What was the value of IFS0 when the code entered the TIMER1 interrupt?
Q. What was the value of IFS0 when the code entered the TIMER2 interrupt?
Q. What caused the program to jump into the _T2Interrupt() function?
Task: Repeat the experiment above, but this time, you must set both timers to have **equal priority.** **Describe the precise sequence of events – noting the value if IFS0**
Q. Note at all ISR's include some code to reset the interrupt status flag. **What happens if you don't do this?**

5. You should have noticed that when TIMER2 has the same priority to TIMER1, they are no longer nested. Even through the interrupt flag for TIMER2 was raised, it had to wait for TIMER1 to finish.
    - **This is the scenario described in** Figure 3**.**

6. It is sometimes useful to dynamically change the interrupt priority to **temporarily** avoid nesting of interrupts – this is often done when running a **critical section (more on this later).**
7. Now we will look at simple way to avoid nesting of interrupts
8. Restore the interrupt priority of TIMER2 to 5
9. Change the line that reads _NSTDIS = 0;
   to _NSTDIS = 1;
    - NSTDIS stands for NeST DISable

TASKS
Repeat the experiment above, again noting the value of the interrupt flag bits IFS0 **Describe the precise sequence of events – noting the value if IFS0**
Q. What impact does nesting have of the timing precision of our TIMER routines? (Discuss with the tutor if unsure)

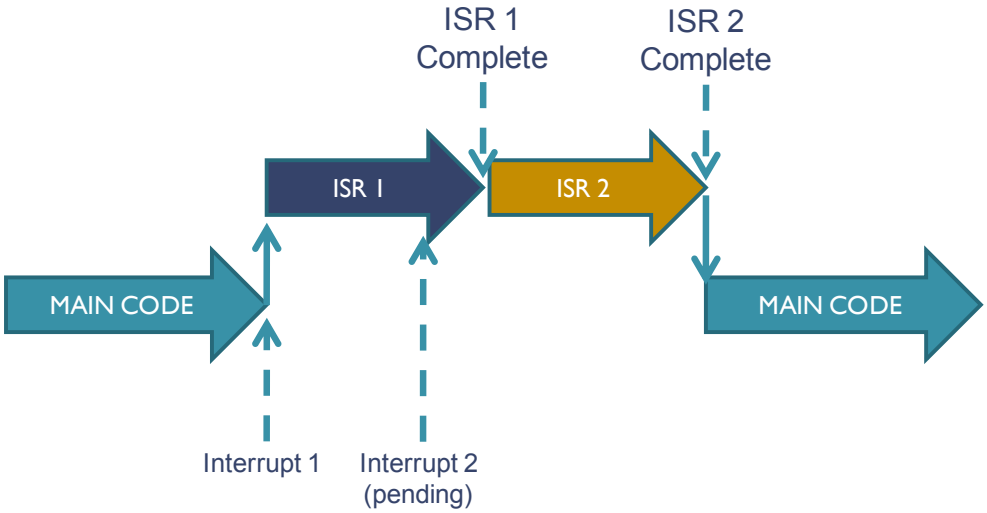

Figure 3. This represents two possible scenarios:

(i) Two interrupts with nesting disabled

(ii) Two interrupts with nesting enabled, but the priority of ISR2 <= priority of ISR1.

NOTE - for all precise timing, you MUST use the MPLAB internal simulator AND NOT the VSM debugger

TASK 2 – CRITICAL SECTIONS

The timers are **internal** interrupt sources, and with care, we can sometimes synchronise them so they never interrupt each other. For some tasks, this can be difficult or impossible to achieve.

Furthermore, external interrupts (key presses, serial data etc…) are not under internal control can occur **at any time.** The developer often has no control at which point an external interrupt may occur.

Turning off nesting of interrupts can also result in poor response times. **For some real-time control and interfacing problems, nesting of interrupts may be deemed essential**[2] to ensure the highest priority interrupts have sufficient timing accuracy.

As a general rule, we can often assume that an external interrupt will occur at **RANDOM**. This can be **during** ANY of your C Program statements[3]!

Refer back to the lecture notes on race conditions. If we have a **shared variable** and we meet the following criteria, then without sufficient protection, **that variable could be corrupted**.

(a) It is referenced in two or more places in functions[4] with different interrupt priority
(b) One of those references performs a write operation
(c) Nesting of interrupts is allowed OR one operation is in the main() routine

Note the word RANDOM above. This is highlighted for a reason. Data corruption depends on the actual sequence that occurs. The term race condition implies this (it is a race to modify the variable). As external interrupts occur at random points, the phenomena of data corruption may only happen at random.

The probability of data corruption is often very low, and is therefore very hard to detect, but its consequence can be catastrophic.

There is a famous case incident where patients in a hospital were killed due to such a software error.

YOU MIGHT WANT TO RESEARCH THE FAMOUS **THERAC-25** RADIATION THERAPY MACHINE ACCIDENT WHICH KILLED THREE PEOPLE AND GAVE RADIATION OVERDOSES TO MANY OTHERS. THIS WAS CAUSED BY A SOFTWARE FAULT INVOLVING A RACE CONDITION.

Given that such events are random, it is difficult to test software for race condition even under controlled conditions.

However, through creating a 'contrived' example (where the probability is made higher), it is possible to at least demonstrate the phenomena of data corruption (we will do something similar next term when using multiple threads on shared variables).

---

[2] There are alternative strategies, but there is no time to go into this at this point
[3] This applies to any language except assembly language. It is still possible to have a 'race condition' in assembly language however, but not at 'statement level'
[4] Don't forget that main is a function, and by default has the lowest priority.

1. Open the project in Task 2. Build and run – the display should static
2. Now enable nested interrupts by changing the line that reads _NSTDIS=1 to _NSTDIS=0
3. Run and repeat – note the change in behaviour

What has happened here? By turning on interrupt nesting[5], all you have changed is the timing.

4. Set the debugger to the internal simulator (not the VSM)[6]

5. For now, disable nesting (set _NSTDIS=1) and build the code
6. Set a break point in both timer service routines (see Figure 4a and Figure 4b).
7. Add a watch to the count variable. *Carefully observe and make note of the value of 'count' as you step through the code*
8. Run until a break point is reached
9. Now step through the code
    a. You should find the counter variable 'count' counts up to 20 in the first ISR, then counts back down to zero in the second ISR
    b. You will notice that interrupt nesting is currently switched **off**
    c. **Confirm that when the code returns to main(), the value of count is 0 every time.**
10. When the second interrupt exits, the count is displayed as zero.
11. So far, so good, it's a simple program and not a very useful one at that – Now we are going to switch **on** nested interrupts
12. In the code, switch nested interrupts back on (set _NSTDIS=0)
13. Repeat the experiment, this time with nesting ON.
    a. Note that the TIMER2 ISR has **higher** priority than TIMER1 ISR
    b. Carefully observe the value of count when the TIMER2 ISR returns and TIMER1 ISR resumes

Question	
What is the value of count after both Interrupt Service Routines (ISR) have completed?	
Comment:	

14. To explain this, you might want to now repeat the experiment with the disassembly listing turned on
    a. Click Window->Debugging->Disassembly
    b. Restart the code
    c. Carefully observe and make note of the value of 'count' as you step through the code
    d. Once you hit the first breakpoint, step through the disassembly listing **one instruction at a time (Debug->Step Instruction[7])**.
    e. Note the *precise* point that Timer2 interrupts Timer1
    f. Note what happens when the Timer2 ISR returns – what happens to count?
    Hint – the C compiler has automatically preserved registers being used on the **stack** when an interrupt occurs.

[5] The ability for one interrupt to pre-empt another of lower priority – see lecture notes or ask the tutor if unsure

[6] See the video on the module site if your are unsure how to do this

[7] On some later version of MPLABX, you can use Ctrl-Alt-F7. You can also customize the toolbar to assist 

g. Note the value of count once the execution returns to the main function. It should be zero, but it is not – why?

TASKS
Describe the precise sequence of events – noting the value of count and any relevant registers

```
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt()
{
    //*****************************
    //BEGINNING OF CRITICAL SECTION
    //*****************************
    count++;
```

Figure 4a - break point in ISR1

```
void __attribute__((interrupt, no_auto_psv)) _T2Interrupt()
{
    //*****************************
    //BEGINNING OF CRITICAL SECTION
    //*****************************
    count--;
```

Figure 4b – breakpoint in ISR2

TASKS
Describe the precise sequence of events – noting the value of count and WREG0 (address 0)
ADVANCED:
At which point are registers preserved on the stack when a ISR is called? **Write down the assembly code that does this.** **Hint: W15 is the stack pointer**

PREVENTING CORRUPTION

There are a number of strategies for avoiding corruption. Careful design, avoiding nested interrupts and avoidance of shared variables may be possible, but there are times when nested interrupts are needed and race conditions may be difficult to predict. In such cases, the following is the simplest strategy to avoid data corruption:

- Identify all critical sections
- Try to ensure all critical sections are as short as possible
- Protect critical sections by temporarily elevating the interrupt priority to the maximum while they are being executed
    - This prevents them from being interrupted

Microchip have provided two macros to help us do this.

```
SET_AND_SAVE_CPU_IPL        //sets the current IPL to a given value

RESTORE_CPU_IPL             //restores the current IPL to a previous value
```

1. Open the project in task 3
2. Inspect the changes in the interrupt service routine
3. Set break points at the start of each ISR
4. Run the code
5. Single step and verify that the critical sections are NOT interrupted and that no data corruption occurs

Question
Q. What is the value of Interrupt Priority during each critical section?
Q. Why is it important to keep critical sections short? (hint – think about timing issues)

THE VOLATILE KEYWORD

You may have noticed that some variables are prefixed with the keyword **volatile.** The keyword volatile tells the compiler the variable might unexpectedly change its value (due to an interrupt).

- Variables marked as volatile will always be explicitly stored in data memory. Each time it is used, it will be read into a register, modified, and written back.
- Variables not marked as volatile MIGHT be optimised persistently stored in a register to avoid the need for multiple read and write operations (this makes faster optimised code).

If you must have a variable shared between ISR's and/or main, it **must** be marked as volatile. **The protection mechanism given above is only guaranteed if the variable is marked as volatile**.