

Interrupts

Referenced Text

- ◆ This set of notes summaries chapter four of:
 - An Embedded Systems Primer, David E. Simon, ISBN: 0-201-61569-X
 - 5 Copies in the Library

Response Problem

- ◆ Response problem is the difficult one of making sure that the embedded system reacts rapidly to external events, even if it is in the middle of doing something else
- ◆ The first approach to the response problem - the one that we will discuss in this lecture - is to use interrupts
- ◆ Interrupts can solve the response problem, but not without some difficult programming, and not without introducing some new problems of their own.

Microprocessor Architecture

- ◆ Before we can discuss interrupts sensibly, you must know something about how microprocessors work
- ◆ Most microprocessors and their assembly languages are fairly similar to one another in a general way
- ◆ We're going to discuss the parts that are similar; we have no need for the details that make microprocessors and assembly languages complicated and make them differ from one another

Assembly

- ◆ Human readable form of machine instructions
- ◆ Translate assembly to machine code using Assembler
 - One assembly instruction = One machine instruction
- ◆ One C statement = many machine instructions
- ◆ Most C compilers will produce a listing file that shows the assembly language that would be equivalent to the C.

Instruction Sets

- ◆ Every family of microprocessors has a different assembly language, because each family understands a different set of instructions
- ◆ Within each family, the assembly languages for the individual microprocessors usually are almost identical to one another

Registers

- ◆ General-purpose registers
 - holds a value that the processor is working with
 - Must move data to registers before doing any operations
 - Different micro family = different name and number of registers
 - We assume GP registers R1, R2, R3, etc
- ◆ Program Counter – or Instruction Pointer
- ◆ Stack Pointer

Assembly basics

- ◆ In a typical assembly language, when the name of a variable appears in an instruction, that refers to the address of that variable.
- ◆ To refer to the value of a variable, you put the name of the variable in parentheses.
- ◆ In most assembly languages anything that follows a semicolon is a comment, and the assembler will ignore it

Assembly Basics

- ◆ `MOVE R3, R2`
- ◆ `MOVE R5, (iTemperature)`
- ◆ `MOVE R5, iTemperature`
- ◆ Some CPU's can only do arithmetic in Accumulator
- ◆ Most do in any GP register
`ADD R7, R3`
- ◆ (Intel V's Motorola)
- ◆ `NOT R4`

Jump Instructions

◆ Unconditional Jump

◆ Labels

```
        ADD R1, R2
        JUMP NO_ADD
MORE_ADDITION:
        ADD R1, R3 ;
        ADD R1, R4
NO_ADD:
        MOVE (xyz), R1
```

◆ Conditional Jump (Branch)

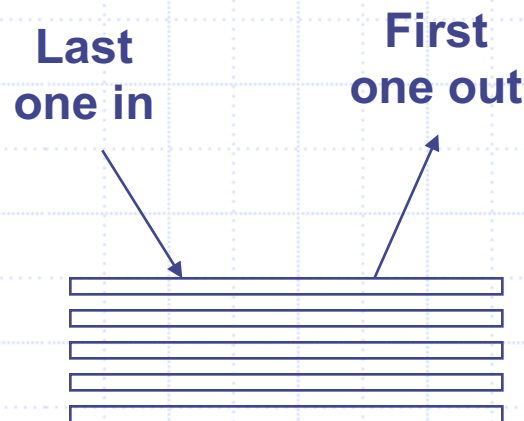
```
SUBTRACT R1, R5
JCOND ZERO, NO_MORE
MOVE R3, (xyz)
...
...
NOMORE:
...
```

The Stack

- ◆ Most assembly languages have access to a stack with PUSH and POP instructions.
- ◆ The PUSH instruction adjusts the stack pointer and adds a data item to the stack.
- ◆ The POP instruction retrieves the data and adjusts the stack pointer back.

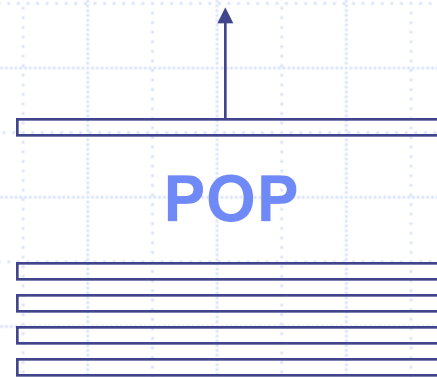
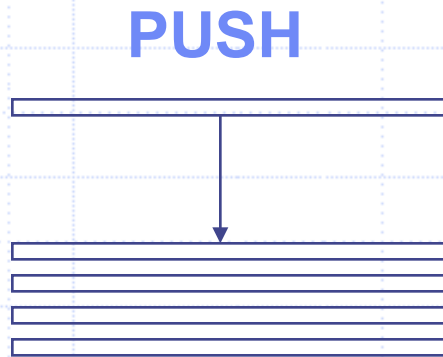
Stacks

- ◆ Most assembly languages have access to a stack with PUSH and POP instructions.
- ◆ Stack is just the piece of memory that the Stack Pointer register points to
- ◆ Helpful analogy: Stack of trays in a cafeteria



Computer Scientist Jargon

- ◆ End of the stack where insertions and deletions are made - **TOP** of the stack
- ◆ Place an item onto the stack - **PUSH** an item onto the stack
- ◆ Take an item off the stack - **POP** an item from the stack



Subroutines

- ◆ CALL instruction - get to subroutines or functions
- ◆ RETURN instruction - for getting back.

```
CALL ADD_EM_UP
MOVE (xyz), R1
...
...
ADD_EM_UP:
ADD R1, R3
ADD R1, R4
ADD R1, R5
RETURN
```

Subroutines

- ◆ The CALL instruction typically causes the microprocessor to automatically Push the address of the instruction after the CALL.
- ◆ In this case, the address of the MOVE instruction - onto the stack.
- ◆ When it gets to the RETURN instruction, the microprocessor automatically Pops that address from the stack to find the next instruction it should execute next (adjust program counter)

C and equivalent Assembly

```
x = y +  
    133;
```

```
MOVE    R1, (y)    ; Get the value of y into R1  
ADD     R1, #133   ; Add 133  
MOVE    (x), R1    ; Save the result in x
```

```
if (x >= z)
```

```
MOVE    R2, (z)    ; Get the value of z  
SUB     R1, R2     ; Subtract z from x  
JCOND   NEG. L101  ; Skip if the result is negative
```

```
{  
    z += y;  
}
```

```
MOVE    R1, (y)    ; Get the value of y into R1  
ADD     R2, R1     ; Add it to z.  
MOVE    (z), R2    ; Save the result in z
```

```
w = sqrt (z);
```

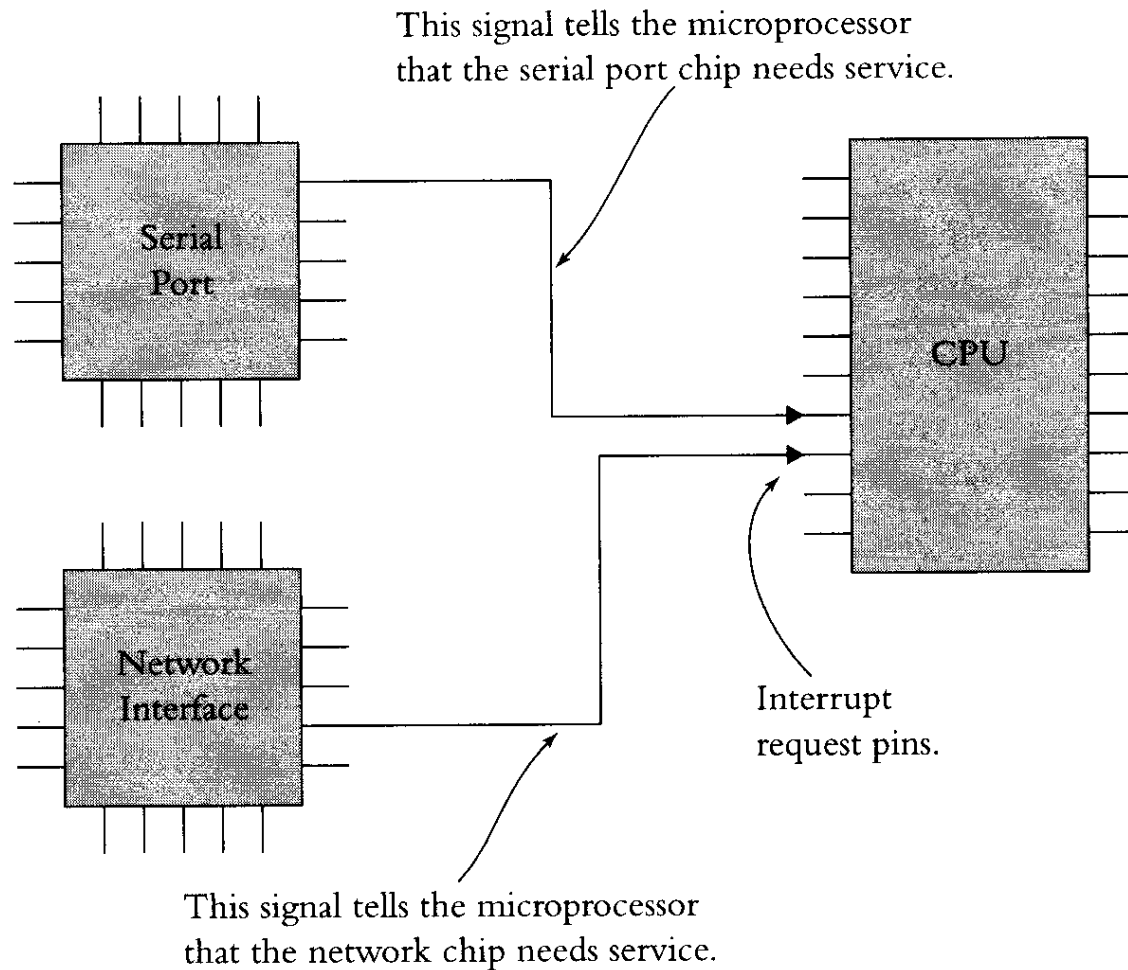
```
L101:   MOVE    R1, (z)    ; Get the value of Z into R1  
PUSH    R1         ; Put the parameter on the stack  
CALL    Sqrt      ; Call the sqrt function  
MOVE    (w), R1    ; The result comes back in R1  
POP     R1         ; Throw away the parameter
```


Interrupt Basics

- ◆ Most I/O chips, such as ones that drive serial ports or network interfaces, need attention when certain events occur.
- ◆ Serial port chip receives a character from the serial port
 - Read that character from where it is stored inside of the serial port chip
 - Store it somewhere in memory
- ◆ Finished transmitting one character
 - Needs the micro to send it the next char
- ◆ Almost any other kind of I/O chip needs the microprocessor's assistance for similar sorts of events

Interrupt Basics

Figure 4.2 Interrupt Hardware



ISR's –Interrupt Service Routines

- ◆ Interrupt request pin is asserted
 - Finish current instruction
 - Push the address of the next instruction on stack
 - Jump to ISR (adjust program counter)
- ◆ Serial Port ISR
 - Read char from register in serial port chip and write to memory
- ◆ ISR's must do some miscellaneous housekeeping chores
 - Reset the interrupt-detecting hardware within the microprocessor to be ready for the next interrupt.

ISR's

Figure 4.3 Interrupt Routines

Task Code

```
. . . .  
MOVE R1, (iCentigrade)  
MULTIPLY R1, 9  
DIVIDE R1, 5  
ADD R1, 32  
MOVE (iFarnht), R1  
JCOND ZERO, 109A1  
JUMP 14403  
MOVE R5, 23  
PUSH R5  
CALL Skiddo  
POP R9  
MOVE (Answer), R1  
RETURN  
. . .  
. . .  
. . .  
. . .  
. . .
```

Interrupt Routine

```
PUSH R1  
PUSH R2  
. . .  
!! Read char from hw into R1  
!! Store R1 value into memory  
. . .  
!! Reset serial port hw  
!! Reset interrupt hardware  
. . .  
POP R2  
POP R1  
RETURN
```

ISR's

- ◆ RETURN instruction
 - Micro pops the address of the next instruction it should do (the one it was about to do when the interrupt occurred) and resumes execution from there. (adjust program counter)
- ◆ ISR like subroutine with no CALL instruction
 - Done by H/W
- ◆ Intel x86 require a special RETURN instruction (IRET)

Saving and Restoring the Context

- ◆ Nearly all subroutines (incl, ISR's) need to use registers
 - Move data into to be able to manipulate
- ◆ Unreasonable to expect anyone to write an interrupt routine that doesn't touch any of the registers.
- ◆ Get around this problem by ISR saving the contents of the registers it uses at the start of the routine and to restore those contents at the end.
- ◆ Usually, the contents of the registers are saved on the stack.

Saving and Restoring the Context

- ◆ Pushing all of the registers at the beginning of an interrupt routine is known as saving the context; popping them at the end, as restoring the context.
- ◆ Failing to do these operations properly can cause troublesome bugs.
- ◆ The distressing thing about this type of bug would be that it would only show up occasionally.

Disabling Interrupts

- ◆ Variety of ways to disable Interrupts
- ◆ Instruct I/O chip not to from program
- ◆ Instruct CPU to ignore certain interrupts – value in special register
- ◆ Almost always a way to disable all interrupts and re-enable again at previous level – usually single assembly instruction

Disabling Interrupts

- ◆ Some Micro's use priorities
- ◆ Program specifies lowest level it will respond to
- ◆ Disable all by setting priority higher than highest
- ◆ Cannot disable NMI
- ◆ Enable all by setting priority very low
- ◆ Selectively enable with medium value
- ◆ Interrupt Mask

Common Questions

- ◆ How does the microprocessor know where to find the ISR when interrupt occurs?
 - Depends on microprocessor
 - Some assume fixed locations, e.g 8051
 - ISR for interrupt 1 at 0x0003 – you put it there
 - Most typical in Interrupt Vector Table
 - microprocessor looks up address of associated ISR
- Again programmers job to set up table

Common Questions

- ◆ How do microprocessors that use an interrupt vector table know where the table is?
 - Again, depends upon the microprocessor.
 - In some, the table is always at the same location in memory, at 0x00000 for the Intel 80186, for example.
 - In others, the microprocessor provides your program with some way to tell it where the table is.

Common Questions

- ◆ Can a microprocessor be interrupted in the middle of an instruction?
 - Usually not – assembly instruction completes before CPU reacts
 - Exceptions - Zilog Z80 and the Intel x86 families of microprocessors have single instructions that move potentially thousands of bytes of data. These instructions can be interrupted at the end of transferring a single byte or word and will resume where they left off when the interrupt routine returns.

Common Questions

- ◆ If two interrupts happen at the same time, which interrupt routine does the microprocessor do first?
 - Highest Priority
 - Control of priorities varies. Some only allow one ISR at each priority.

Common Questions

- ◆ Can an interrupt request signal interrupt another interrupt routine?
 - Yes.
 - Called Interrupt Nesting
 - Default on some
 - Some like x86 disable all interrupts whenever ISR is called so ISR must re-enable to allow nesting
 - Only a higher priority interrupt can interrupt
 - If lower priority interrupt occurs during higher priority ISR the CPU will finish high priority ISR and then execute lower priority ISR

Common Questions

- ◆ What happens if an interrupt is signalled while the interrupts are disabled?
 - Most cases the micro will “remember” and execute the ISR when interrupts re-enabled
 - If more than one interrupt while interrupts disabled
 - micro will execute ISR's in priority order
 - Interrupts are not really disabled- just deferred.

- ◆ What happens if I disable interrupts and then forget to re-enable them?
 - Nearly all processing in Embedded Systems involves ISR's so the system will grind to a halt!

Common Questions

- ◆ What happens if I disable interrupts when they are already disabled or enable interrupts when they are already enabled?
 - Nothing.

- ◆ Are interrupts enabled or disabled when the microprocessor first starts up?
 - Disabled.

Common Questions

- ◆ Can I write Interrupt routines in C?
 - Yes, usually.
 - Most compilers used for embedded systems allow non-standard keywords to tell the compiler that a particular function is an ISR.
 - The compiler will then add the code (assembly) to save and restore the context.
 - If using an x86 then compiler will use IRET instead of RET
 - C ISR's can deal with H/W
 - Must set up vector table with address of routine
 - Only write ISR's in assembly for speed – otherwise easier in C

Common Questions

```
void interrupt vHandleTimerIRQ (void)
{
    ...
    ...
    ...
}
```

Summary so far...

- ◆ One Assembly instruction = one machine instruction
- ◆ One C Statement may equal many machine instructions
- ◆ Move data to registers before processing
- ◆ Jump, conditional jump, call, return, push, pop
- ◆ IRQ pin, ISR
- ◆ ISR's save and restore context

Shared-Data Problem

- ◆ ISR's need to communicate with rest of code (task code)
- ◆ For various reasons it is neither possible nor desirable for Micro's to do all its work in ISR's
- ◆ ISR's need to signal task code to do follow-up work
- ◆ ISR and task code need to share one or more variables to communicate

Shared-Data Problem

Code for control of
Nuclear Reactor.

Sound alarm if
temps ever differ.

ISR called if either
temp changes or
periodically...

What's the bug?

```
static int iTemperatures[2];

void interrupt vReadTemperatures (void)
{
    iTemperatures[0] = !! read in value from hardware
    iTemperatures[1] = !! read in value from hardware
}

void main (void)
{
    int iTemp0, iTemp1;

    while (TRUE)
    {
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];

        if (iTemp0 != iTemp1)
            !! Set off howling alarm;
    }
}
```

Shared-Data Problem

Does this fix
the bug?

```
static int iTemperatures[2];

void interrupt vReadTemperatures (void)
{
    iTemperatures[0] = !! read in value from hardware
    iTemperatures[1] = !! read in value from hardware
}

void main (void)
{
    while (TRUE)
    {
        if (iTemperatures[0] != iTemperatures[1])
            !! Set off howling alarm;
    }
}
```

Shared-Data Problem

- ◆ Same bug just more subtle form.
- ◆ Micro will not interrupt individual assembly instruction but can and will interrupt C statements as each C statement can translate to many assembly instructions.
- ◆ Whenever an interrupt routine and your task code share data, be suspicious and analyse the situation to ensure that you do not have a shared data bug.

Assembly for if statement

if (iTemperatures[0] != iTemperatures[1]) —→ C code becomes
!! Set off howling alarm;

int happens here?

```
void interrupt vReadTemperatures (void)
{
    iTemperatures[0] = !! read val from H/W
    iTemperatures[1] = !! read val from H/W
}
```

...

```
MOVE    R1, (iTemperatures[0])
MOVE    R2, (iTemperatures[1])
SUBTRACT R1, R2
JCOND   ZERO, TEMPERATURES_OK
...
```

...

... ; Code goes here to set off the alarm

...

...

TEMPERATURES_OK:

...

...

Characteristics of the Shared-Data bug

- ◆ Bug only shows if int occurs during first of the 2 crucial assembly instructions
- ◆ Otherwise prog runs perfectly – instructions only take millisecs so likelihood of bug showing very small
- ◆ Very difficult to find
- ◆ Bug usually appears...
 - ◆ 5 o'clock in the afternoon, usually on Friday
 - ◆ Any time you are not paying very much attention
 - ◆ Whenever no debug equipment is attached to the system
 - ◆ After your product has landed on Mars
 - ◆ And, of course, during customer demos

Solving the Shared-Data Problem

- ◆ Disable interrupts whenever your task code uses the shared data
- ◆ Lib functions `enable()` and `disable()` usually available in embedded systems compilers (not always these names)
- ◆ No compiler smart enough to figure out when to disable/enable – must be done by programmer

```
void interrupt vReadTemperatures (void)
```

```
{
```

```
    iTemperatures[0] = !! read in value from hardware
```

```
    iTemperatures[1] = !! read in value from hardware
```

```
}
```

```
void main (void)
```

```
{
```

```
    int iTemp0, iTemp1;
```

```
    while (TRUE)
```

```
{
```

```
        disable(); /* Disable interrupts while we use the array */
```

```
        iTemp0 = iTemperatures[0];
```

```
        iTernp1 = iTemperatures[1];
```

```
        enable();
```

```
        if (iTemp0 != iTemp1)
```

```
            !! Set off howling alarm;
```

```
    }
```

```
}
```

Assembly for en/disabling interrupts

...

...

DI ; disable interrupts while we use the array

MOVE R1, (iTemperature[0])

MOVE R2, (iTemperature[1])

EI ; enable interrupts again

Single instruction
to enable and
disable ints

SUBTRACT R1, R2

JCOND ZERO, TEMPERATURES_OK

...

...

; Code goes here to set off the alarm

...

...

TEMPERATURES_OK:

...

NMI not disabled
thus never use
shared data in
NMI ISR

"Atomic" and "Critical Section"

- ◆ Atomic – part of program that cannot be interrupted.
- ◆ Critical Section – a set of instructions that must be atomic for the system to work properly.
- ◆ Shared-Data problem – task code uses data in a way that is not atomic
 - Solve by making critical section atomic.
- ◆ Atomic also means– cannot be interrupted by anything that might mess up the data it is using.
- ◆ If other interrupts change other data, the time of day, water pressures, steam pressures, etc.—while the task code is working with the temperatures, that will cause no problem.

```
static int iSeconds, iMinutes, iHours;
```

```
void interrupt vUpdateTime (void)  
{
```

```
    ++iSeconds;
```

```
    if (iSeconds >= 60)
```

```
    {
```

```
        iSeconds = 0;
```

```
        ++iMinutes;
```

```
        if (iMinutes >= 60)
```

```
        {
```

```
            iMinutes = 0;
```

```
            ++iHours;
```

```
            if (iHours >= 24)
```

```
            iHours = 0;
```

```
        }
```

```
    }
```

```
// Do whatever needs to be done to the hardware  
}
```

```
long iSecondsSinceMidnight (void)
```

```
{
```

```
    return ( (((iHours * 60) + iMinutes) * 60) + iSeconds);
```

```
}
```

vUpdateTime is triggered
by a H/W interrupt every
sec

Obvious Bug?

How far out could
iSecondsSinceMidnight()
be?

More Examples

Does this fix the Bug?

```
long iSecondsSinceMidnight (void)
{
    disable();
    return ( (((iHours * 60) + iMinutes) * 60) + iSeconds);
    enable();
}
```

Is this any better?

```
long iSecondsSinceMidnight (void)
{
    long iRetVal;

    disable();
    iRetVal = (((iHours * 60) + iMinutes) * 60) + iSeconds;
    enable();

    return (iRetVal);
}
```


More Examples

- ◆ Still a problem with above code
- ◆ What if `iSecondsSinceMidnight()` is called from within a critical section somewhere else in the program?
- ◆ Will cause bug by re-enabling interrupts too soon
- ◆ Some C-lib implementations of `disable()` return a Boolean value indicating whether ints were enabled or disabled when called
- ◆ Following code is best solution for `iSecondsSinceMidnight()`
 - Runs a little bit slower?

More Examples

```
long iSecondsSinceMidnight (void)
{
    long iRetVal;
    BOOL fInterruptStateOld; /* Interrupts already disabled? */

    fInterruptStateOld = disable();

    iRetVal = (((iHours * 60) + iMinutes) * 60) + iSeconds;

    /* Restore interrupts to previous state */
    if (fInterruptStateOld)
    {
        Enable();
    }

    return (iRetVal);
}
```

Another Solution

Does this solve the shared-data problem?

Does task code use the shared variable in a non-atomic way?


```
static long int iSecondsToday;

void interrupt vUpdateTime (void)
{
    ...
    ++iSecondsToday;

    if (iSecondsToday == 60 * 60 * 24)
        iSecondsToday = 0L;
    ...
    ...
}

long iSecondsSinceMidnight (void)
{
    return (iSecondsToday);
}
```

Another Solution



```
_iSecondsSinceMidnight:  
    MOVE    R1, (ISecondsToday)  
    RETURN
```

Two possible assembly
versions of
iSecondsSinceMidnight()

```
_iSecondsSinceMidnight:  
    MOVE    R1, (ISecondsToday)           ; Get first byte or word  
    MOVE    R2, (ISecondsToday+1) ; Get second byte or word  
    RETURN
```

Another Solution

- ◆ Do not depend on this last C implementation unless absolutely necessary
- ◆ Might work on your 32-bit micro today but what if you have to port it to a 16-bit micro tomorrow?
- ◆ Better to always disable interrupts even if not as efficient
 - Always think ahead!! Design robust code!!

The *volatile* Keyword

- ◆ Most compilers assume that a value stays in memory unless the program changes it - use that assumption for optimization.
- ◆ This can cause problems.
- ◆ The following code is an attempt to fix the shared-data problem in `iSecondsSinceMidnight()` without disabling interrupts.
- ◆ It is a fix that works, even on processors with 8 and 16-bit registers (as long as the while-loop in `iSecondsSinceMidnight()` executes in less than one second, as it will on any microprocessor).
- ◆ The idea is that if `iSecondsSinceMidnight()` reads the same value from `iSecondsToday` twice, then no interrupt can have occurred in the middle of the read, and the value must be valid.

```
static long int iSecondsToday;
```

```
void interrupt vUpdateTime (void)
```

```
{ ...  
  ++iSecondsToday;  
  if (iSecondsToday == 60L * 60L * 24L)  
  {  
    iSecondsToday = 0L;  
  }  
  ...  
}
```

```
long iSecondsSinceMidnight (void)
```

```
{  
  long iReturn;
```

```
  /* When we read the same value twice, it must be good. */
```

```
  iReturn = iSecondsToday;
```

```
  while (iReturn != iSecondsToday)
```

```
  {  
    iReturn = iSecondsToday;
```

```
  }  
  return (iReturn);
```

```
}
```

The *volatile* Keyword

- ◆ Some compilers will conspire against you, however, to cause a new problem.
- ◆ With optimizing switched on the compiler may decide not to reread a value into a register if it is already there assuming that it could not have changed in memory...
- ◆ Some compilers may even optimize out the entire while loop - this leaves us with same buggy old version
 - May be difficult to spot as source code looks OK
 - Be careful using optimization
- ◆ To fix add keyword `volatile` before declaration of variable
 - `static volatile long int iSecondsToday;`
- ◆ Could also just switch off optimization

Interrupt Latency

- ◆ Interrupts are a tool for getting better response
- ◆ How fast does system respond to interrupt?
- ◆ Depends on:
 1. Longest period that ints are disabled
 2. Time it takes to execute any ints of higher priority
 3. How long it takes CPU to stop what it is doing and start executing ISR
 4. How long it takes ISR to save context and do enough work to be considered a response

Interrupt Latency

- ◆ Interrupt Latency used to mean various combinations of above
- ◆ We use term to include all
- ◆ How to get the interrupt Latency time?
 - Factor 3 can be found in the Micro's documentation
- ◆ Factor's 1,2 & 4
 - Write code and measure
 - Count instructions and look up execution times in documentation
 - Works reasonably for smaller Micro's without instruction caches

Controlling Latency

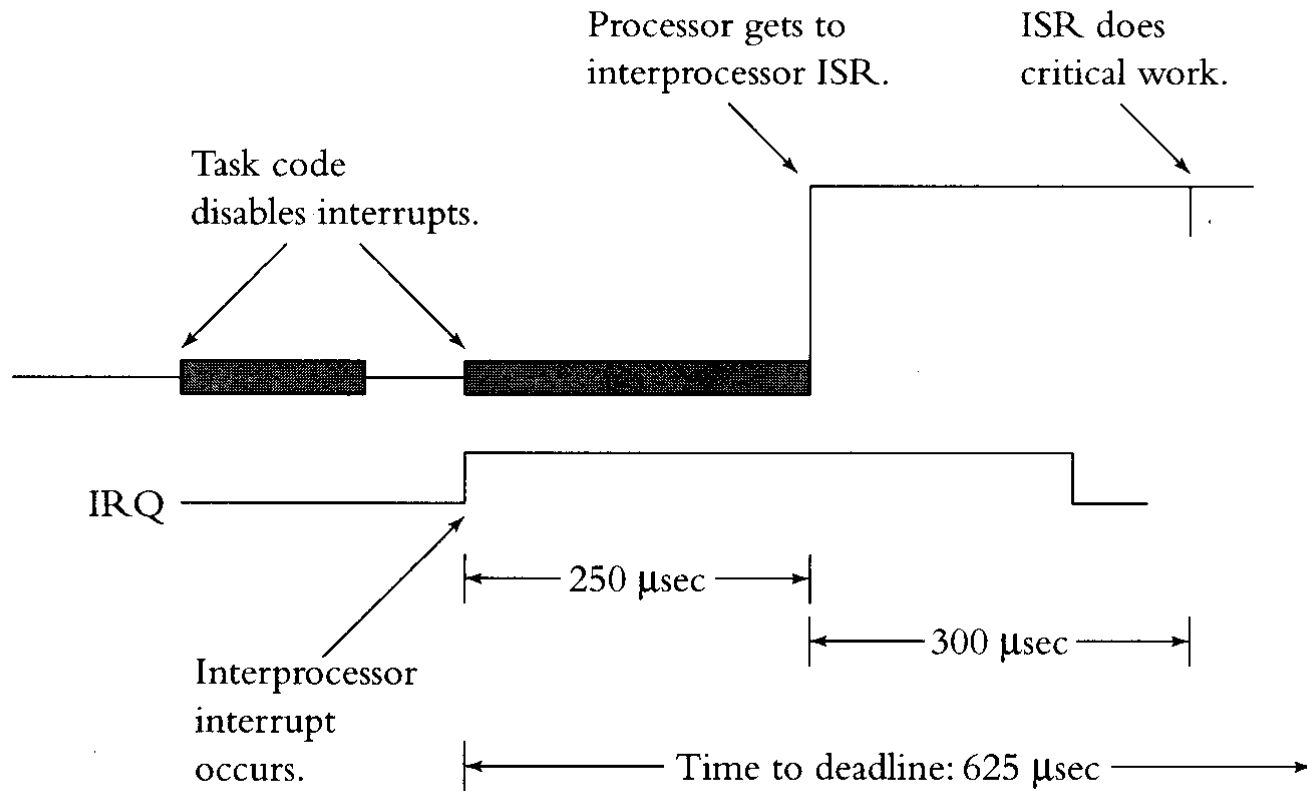
- ◆ Factor 4 – write efficient code!
- ◆ Factor 3 – not under S/W control
- ◆ Factor 2 – keep ISR's short
 - Otherwise response to low priority ints could be awful
- ◆ Factor 1 – Down to you!
 - Necessary to solve shared-data problem but must be careful not to trash response times
 - Shorter time ints disabled the better

Example

- ◆ Suppose that the requirements for your system are as follows:
- ◆ You have to disable interrupts for 125 μ sec for your task code to use a pair of temperature variables it shares with the interrupt routine that reads the temperatures from the hardware and writes them into the variables.
- ◆ You have to disable interrupts for 250 μ sec for your task code to get the time accurately from variables it shares with the interrupt routine that responds to the timer interrupt.
- ◆ You must respond within 625 μ sec when you get a special signal from another processor in your system; the interprocessor interrupt routine takes 300 μ sec to execute.
- ◆ Can this be made to work?

Interrupt Latency example

Figure 4.13 Worst Case Interrupt Latency



Interrupt Latency example

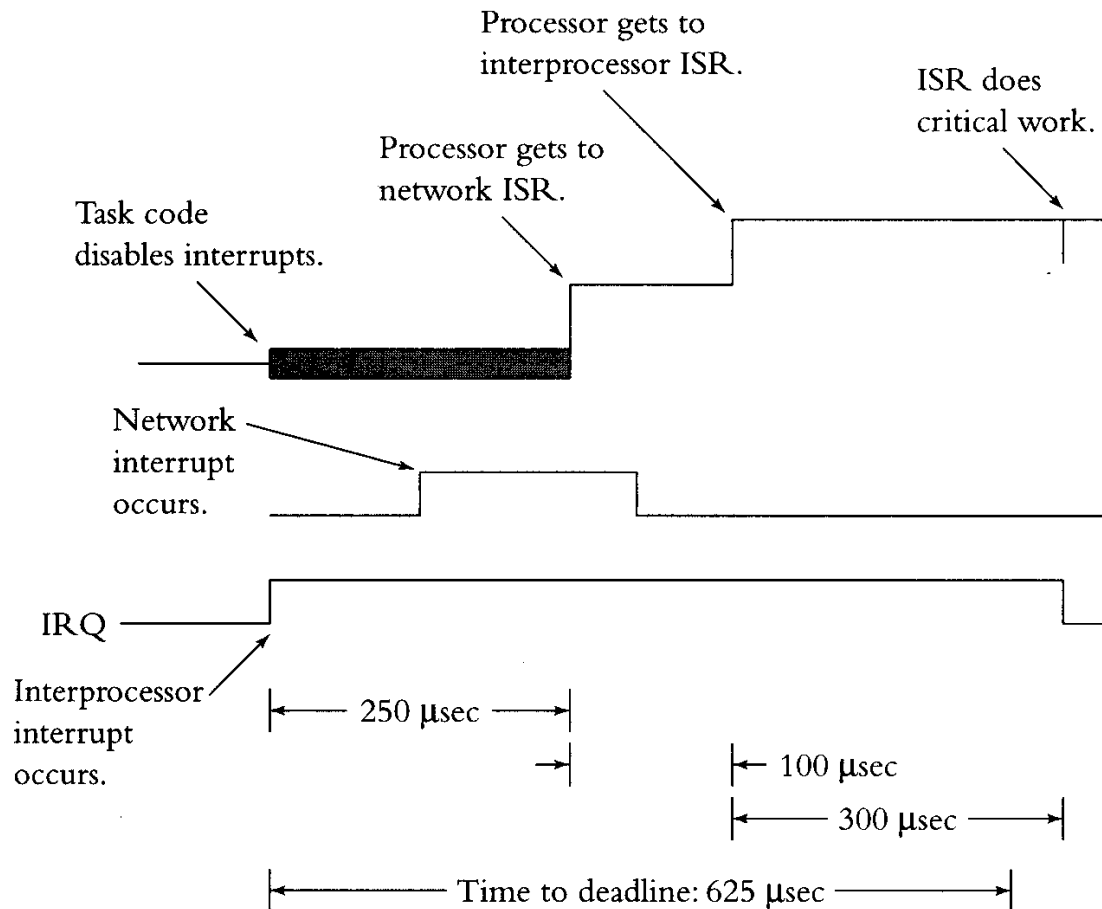
- ◆ Worst case response = $250 + 300 = 550\text{sec}$
 - Within 625 limit
- ◆ Max time ints disabled = $250\ \mu\text{sec}$ - not 375!
 - Why?
- ◆ What if H/W Engineer decides to cut costs by using a Micro that only runs half as fast?
 - All processing times are doubled
 - Does system still work?

Interrupt Latency example

- ◆ Suppose that we manage to talk the hardware group out of the idea of the slower processor, but now the marketing group wants to add networking capability to the system.
- ◆ Suppose that the interrupt routine for the network hardware will take 100 μ sec to do its work.
- ◆ Will the system respond to the interprocessor interrupt quickly enough?
- ◆ Depends on priorities assigned?

Interrupt Latency example

Figure 4.14 Worst Case Interrupt Latency



Alternatives to Disabling Interrupts

- ◆ Disabling Interrupts increases interrupt latency
- ◆ There are alternatives
- ◆ Disabling Interrupts is more robust
- ◆ Alternatives result in fragile code
- ◆ Only use alternatives as last resort

Alternative example...

```
static int iTemperaturesA[2];
static int iTemperaturesB[2];

static BOOL fTaskCodeUsingTempsB = FALSE;

void interrupt vReadTemperatures (void)
{
    if (fTaskCodeUsingTempsB)
    {
        iTemperaturesA[0] = !! read value from H/W
        iTemperaturesA[1] = !! read value from H/W
    }
    else
    {
        iTemperaturesB[0] = !! read value from H/W
        iTemperaturesB[1] = !! read value from H/W
    }
}
```

Alternative example... (cntd)

```
void main (void)
{
    while (TRUE)
    {
        if (fTaskCodeUsingTempsB)
        {
            if (iTemperaturesB[0] != iTemperaturesB[1])
                !! Set off howling alarm;
            else
                if (iTemperaturesA[0] != iTemperaturesA[1])
                    !! Set off howling alarm;
        }

        fTaskCodeUsingTempsB =
        !fTaskCodeUsingTempsB;
    }
}
```

What are the advantages of this code?

What are the disadvantages of this code?

Another alternative example...

```
#define QUEUE_SIZE 100
int iTemperatureQueue[QUEUE_SIZE];

int iHead = 0;    /* Place to add next item */
int iTail = 0;    /* Place to read next item */

void interrupt vReadTemperatures (void)
{
    /* If the queue is not full . . . */
    if ( !( ( iHead + 2 == iTail) || (iHead == QUEUE_SIZE - 2 && iTail == 0) ) )
    {
        iTemperatureQueue[iHead] = !! read one temperature;
        iTemperatureQueue[iHead + 1] = !! read other temperature;
        iHead += 2;
        if (iHead == QUEUE_SIZE)
            iHead = 0;
    }
    else
        !! throw away next value
}

Benjamin Toland
```

Another alternative example...(cntd)

```
void main (void)
{
    int iTemperature1, iTemperature2;

    while (TRUE)
    {
        /* If there is any data. . . */
        if (iTail != iHead)
        {
            iTemperature1 = iTemperatureQueue[iTail];
            iTemperature2 = iTemperatureQueue[iTail + 1];
            iTail += 2;

            if (iTail == QUEUE_SIZE) {
                iTail = 0;
            }

            if (iTemperature1 != iTemperature2)
                !! Set off howling alarm;
        }
    }
}
```

What are the advantages of this code?

What are the disadvantages of this code?

Another alternative example...(cntd)

- ◆ What happens if the task code moves the tail pointer before reading the data from the queue?
- ◆ What if this is executing on an 8-bit processor, the array is larger than 256 entries and as such the increment by 2 of the tail pointer in the task code is not atomic?
- ◆ This code is too complex and fragile. Only do this if disabling interrupts is the only way to reduce latency.

Summary

- ◆ One Assembly instruction = one machine instruction
- ◆ One C Statement may equal many machine instructions
- ◆ Move data to registers before processing
- ◆ Jump, conditional jump, call, return, push, pop
- ◆ IRQ pin, ISR
- ◆ ISR's save and restore context

Summary... cntd

- ◆ Disable all interrupts (except NMI) for critical sections
- ◆ ISR's and task code that share data – shared-data problem
- ◆ Critical sections, Atomic
- ◆ Volatile
- ◆ Interrupt Latency: Keep low by
 - Short ISR's
 - Disable ints for short periods
- ◆ Alternatives to disabling interrupts to solve shared-data prob
 - Fragile and complex
 - Only use if absolutely necessary