Introduction to interrupts

Reference: Simon chapter 4

Motivation for interrupts

- Imagine I have 300 watering tanks in my greenhouse

- I have one microprocessor that constantly monitors the water level at every single tank.

- It checks tank #1, then tank #2, and so on... Unfortunately it takes 30 seconds per measurement

- I really don't want to wait a long time if I want to know how much water is in a particular tank.

Interrupts

- **Interrupts** will tell the microprocessor to stop running whatever code he is working on, and execute a different code instead!

- Interrupts solve the response problem, but it makes programming a little bit harder.

Assembly language

- Assembly (**ASM**) is the human-readable form of the instructions the microprocessor really knows how to do.
- A program (**assembler**) translates the assembly language into binary numbers such that the microprocessor can execute them.
- Each assembly instruction corresponds to a single microprocessor instruction.

C-code vs ASM

```
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
     SUBTRACT 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
```

- These days is quite un-usual to write code in ASM.
- Most projects use C or some variant.
- When we write the code in C, then our instructions are converted into multiple microprocessor instructions.

Registers

- Every family of microprocessors has a different assembly language, because each architecture requires a different set of instructions.
- The typical microprocessor has a different set of storage elements, **registers**, which are just flip-flops.
- Registers hold temporary data that will be used elsewhere.
- For example if we want to multiply two numbers, we need to store them in registers before sending them to the ALU.
- Registers names vary for each architecture, but they are usually named R1, R2, R3, ...

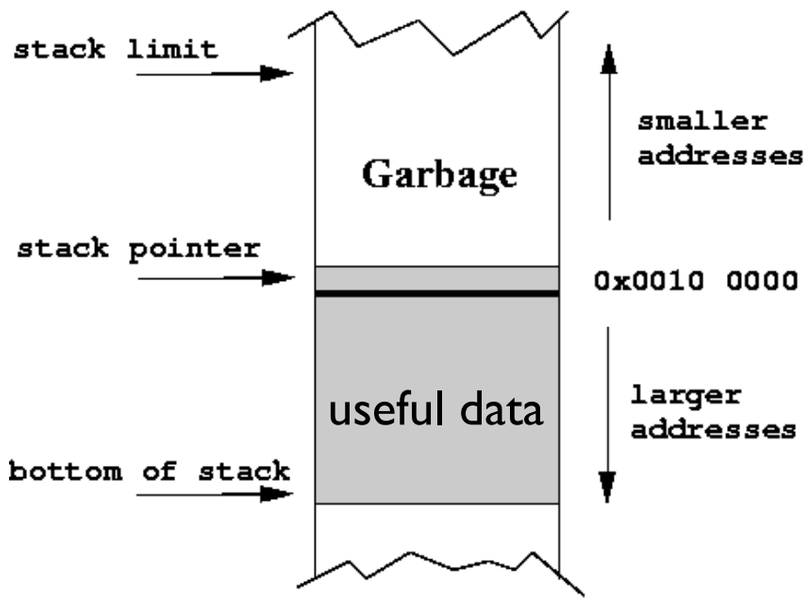
Special purpose registers

- A **general purpose register** is used for storing temporary data that is *in transit*.

- A **program counter** is a special purpose register that stores the address of the next microprocessor instruction.

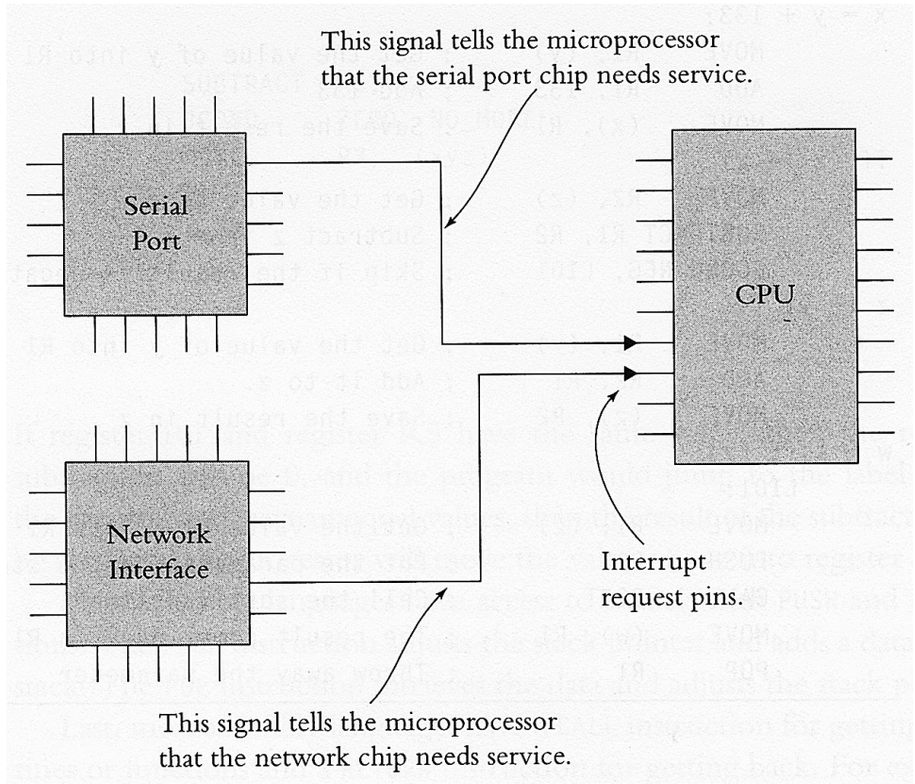
- A **stack pointer** is a special purpose register that store the address of the last free position in memory.

The stack pointer corresponds to the RAM address of the last useful data. The contents of addresses before the stack pointer, can be replaced with new data.



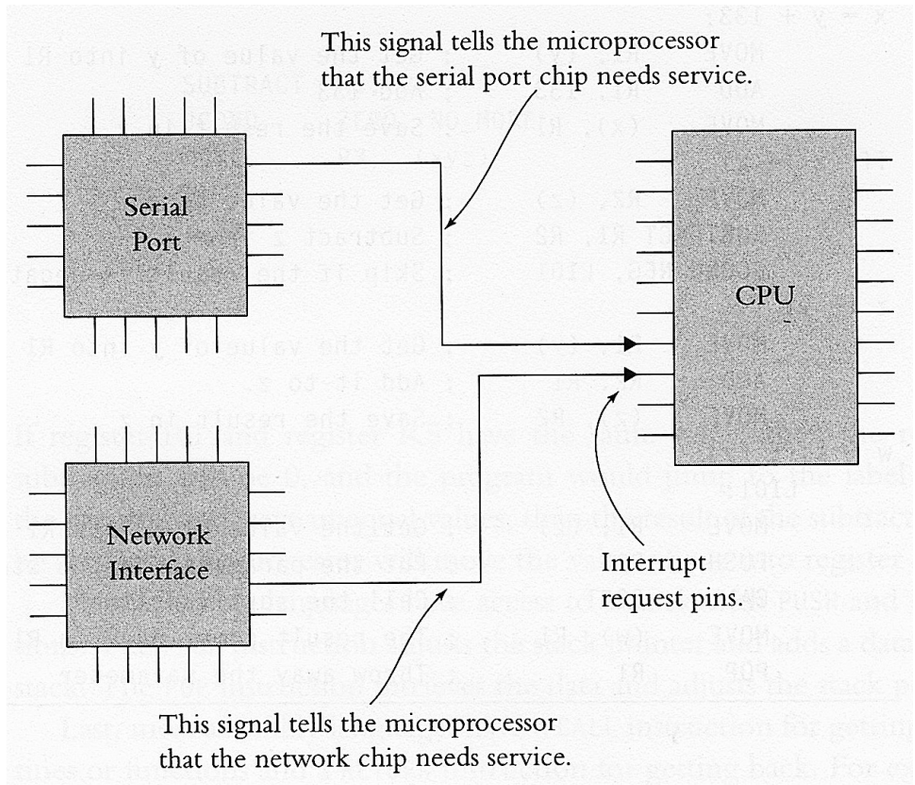
Interrupt basics

- Interrupts start with a signal from the hardware.
- Most I/O chips (serial ports, network interfaces, ...) need attention when certain events occur.
- Example: when a serial port chip receives a character from, it needs the microprocessor to read that character from where it is stored inside the serial port, into some memory location.



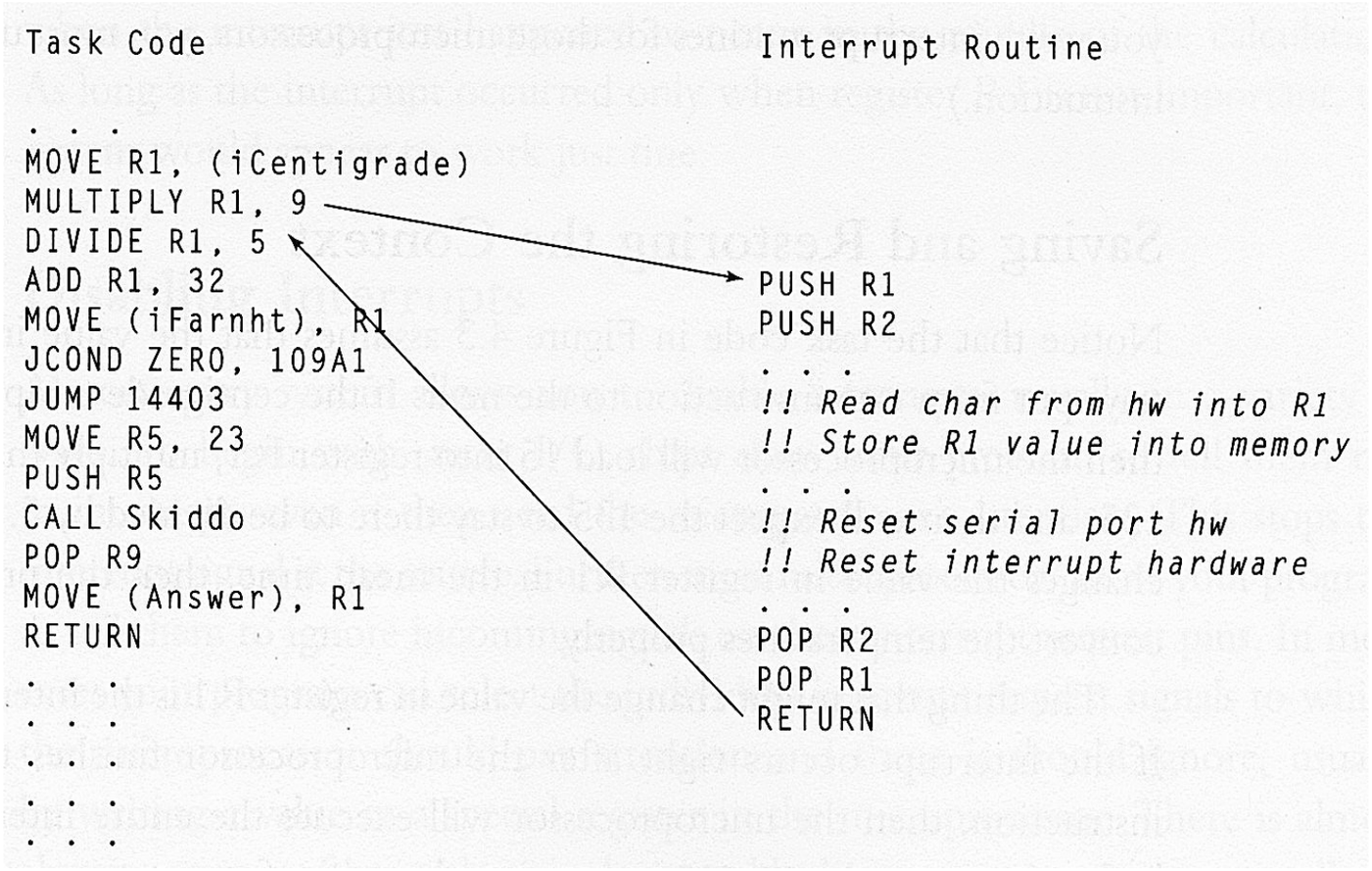
Interrupt request

- When the interrupt request (IRQ) is asserted it stops executing the current sequence of instructions and jumps to an **interrupt routine**.
- Interrupt routines are code blocks you write with whatever needs to be done when an interrupt is requested.
- An interrupt routine is normally called **interrupt handler** or **interrupt service routine** (ISR).



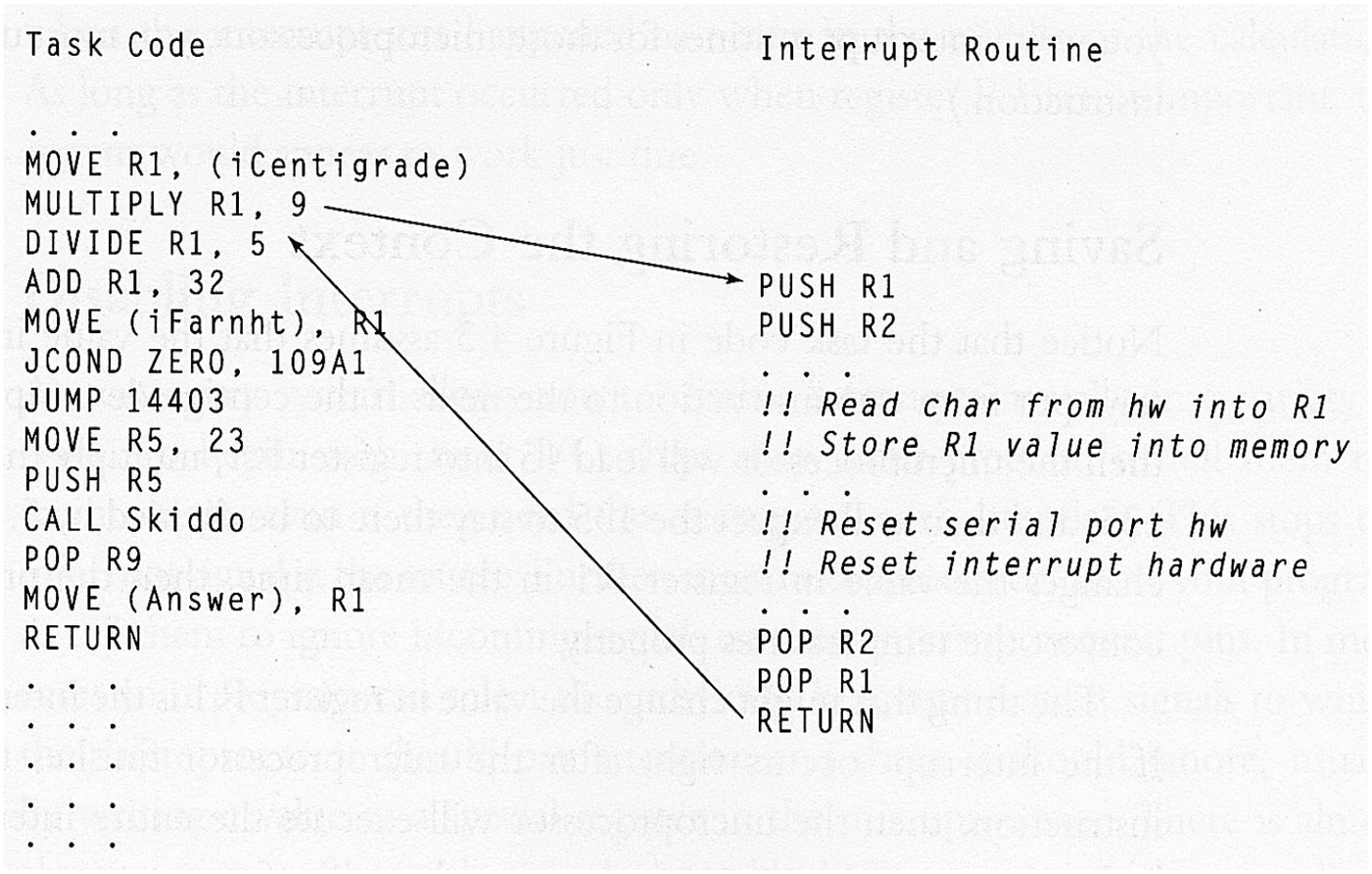
Example of an interrupt routine

- When the interrupt comes from the serial port chip, it means a new character has arrived.
- The interrupt routine must then read that character at once, and when done, the processor can resume its previous execution.
- There is no call to the **ISR**, since this section of the code will only be entered upon an interrupt assertion.



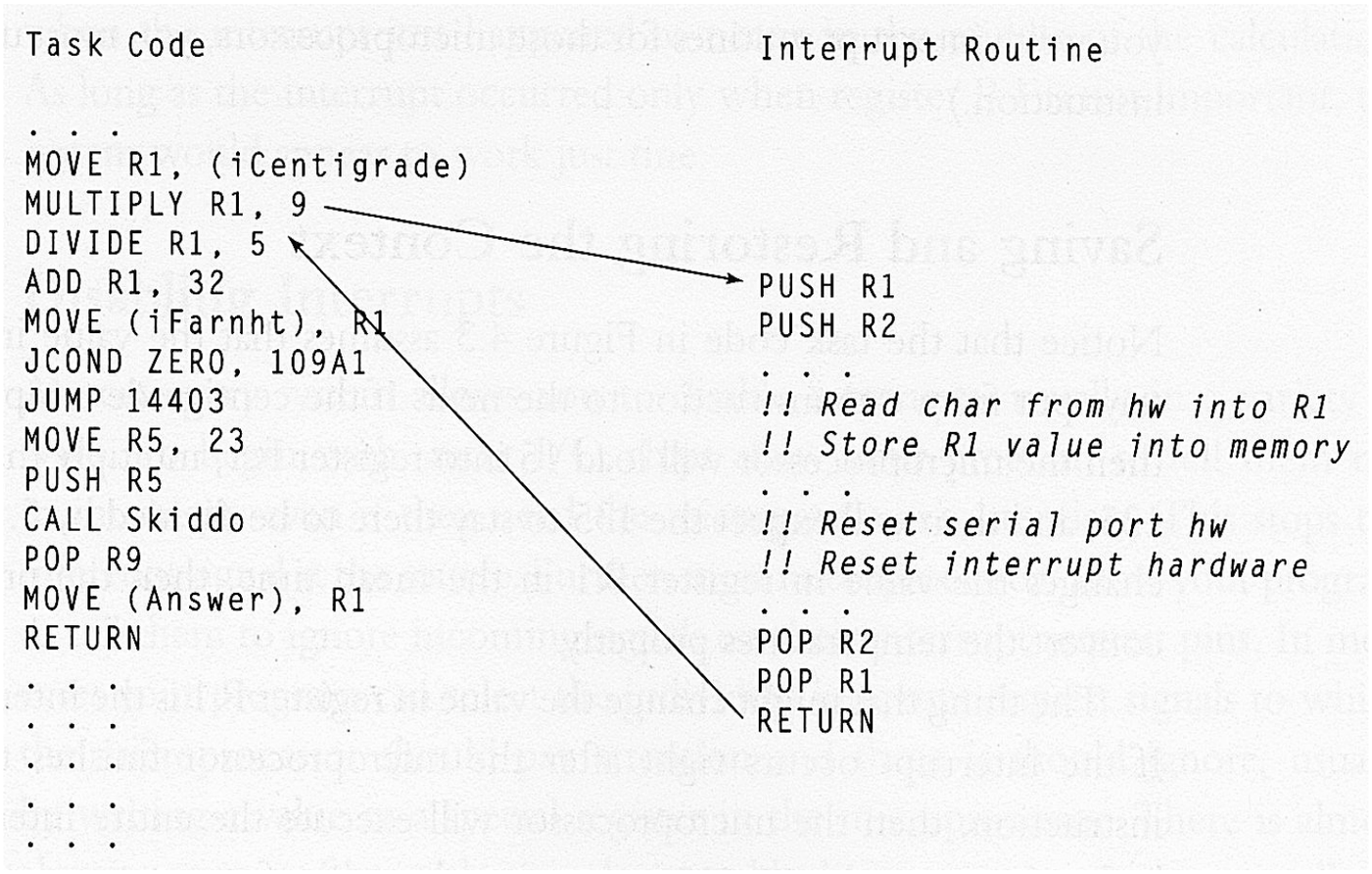
Task code

- We call the **task code** to the code that is <u>not</u> part of the interrupt routine.
- The task code is busily performing a conversion between Celsius to Fahrenheit.
- It move the Celsius into register R1 and performs the appropriate calculations and stores the result in R1.



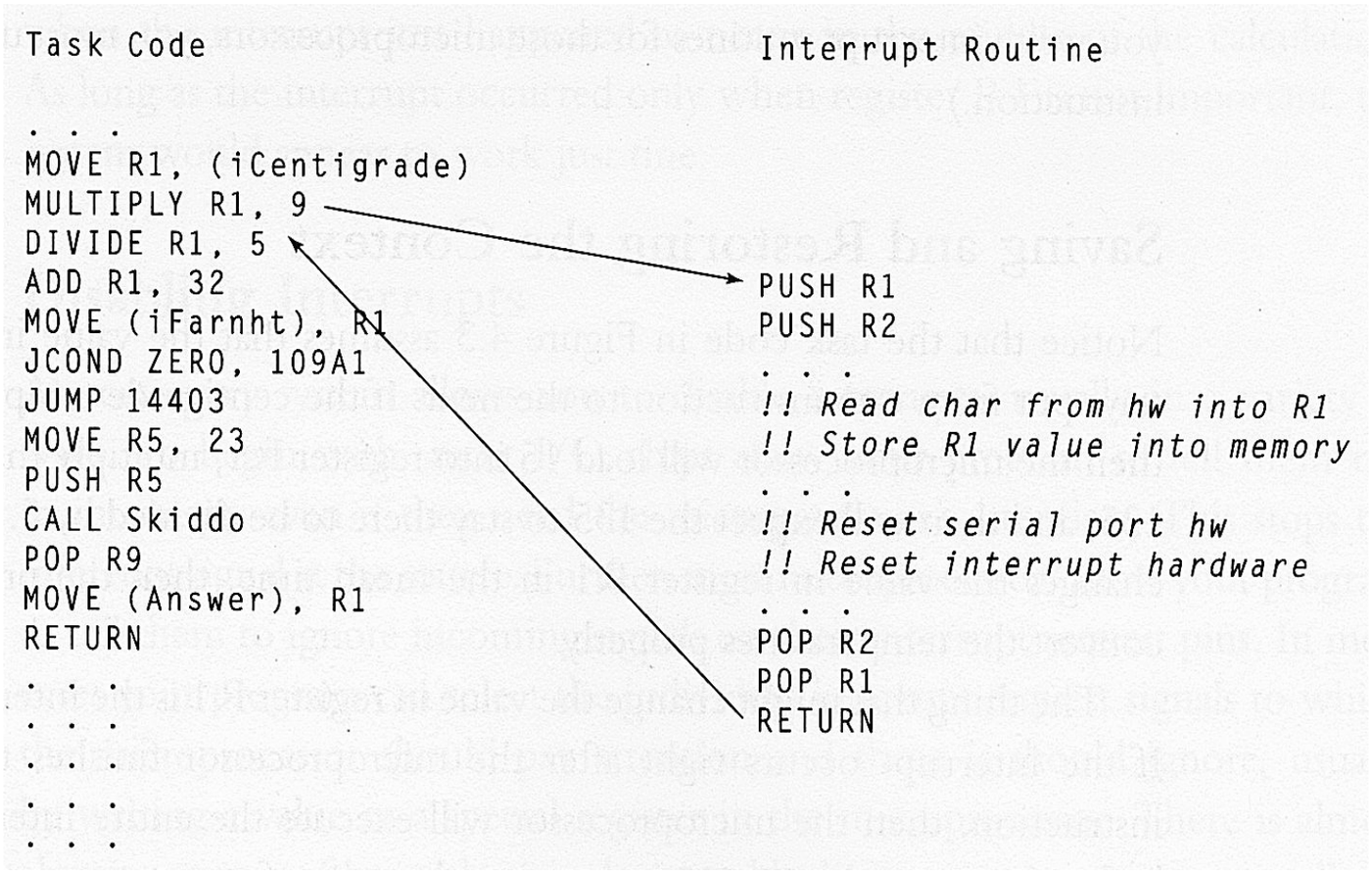
Interrupt routine

- When the interrupt happens, the task code will be suspended and all instructions from the interrupt routine are executed.
- When all instructions are done, the RETURN keyword will tell the microprocessor to resume the task code.



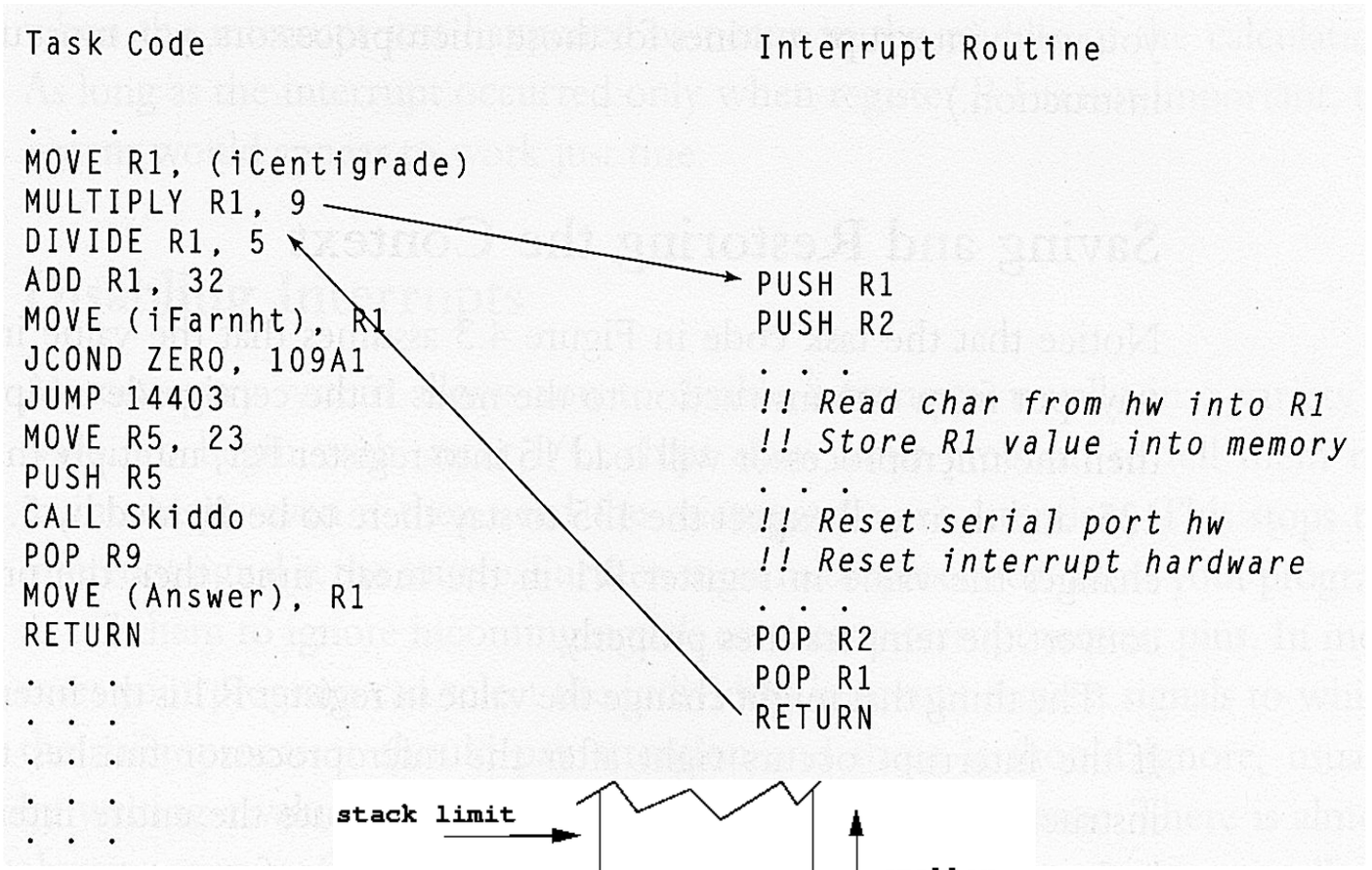
What is task code doing?

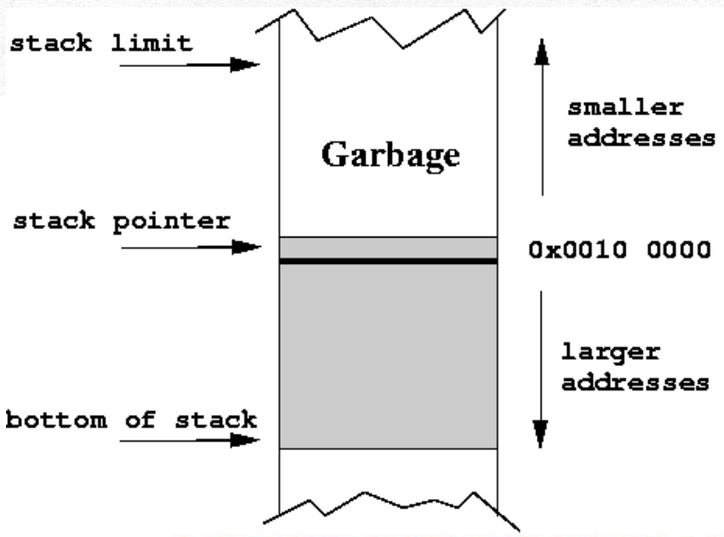
- The task code puts the temperature in R1
- It multiplies it by 9 and stores the result in R1
- Then it divides it by 5 and also stores it in R1.
- Adds 32 to it and also stores it in R1.
- ... So if the value of R1 changes in the interrupt, then the computation will be wrong!



Stack PUSH and POP

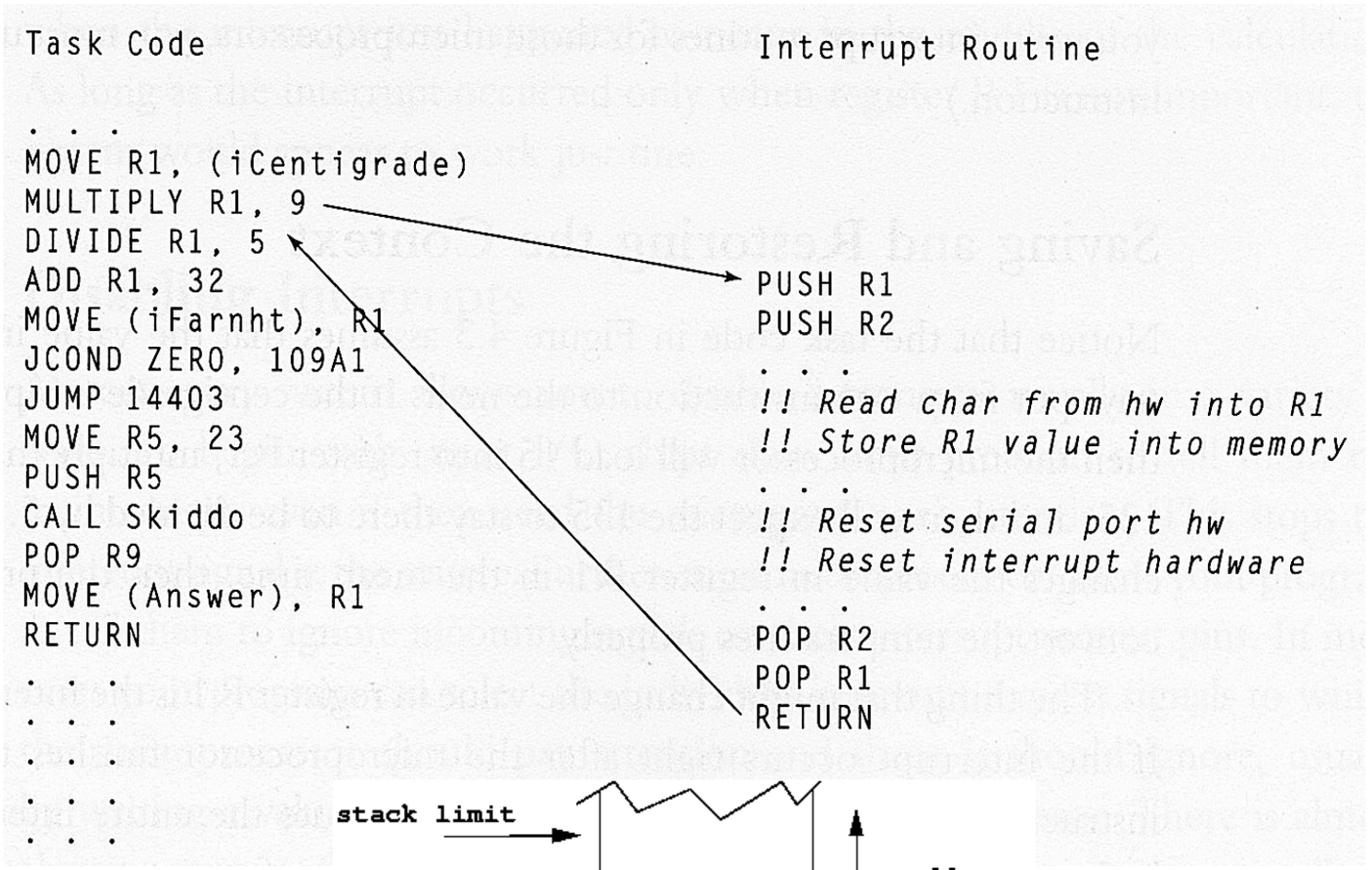
- It is very hard for the interrupt routine not to use registers like R1.
- So we must save relevant register content into the stack.
- That is why we must add the PUSH R1 and PUSH R2 operations, and then POPing them at the end.
- PUSH means we "put stuff" into the stack. POP means we "take stuff" from the top of the stack.

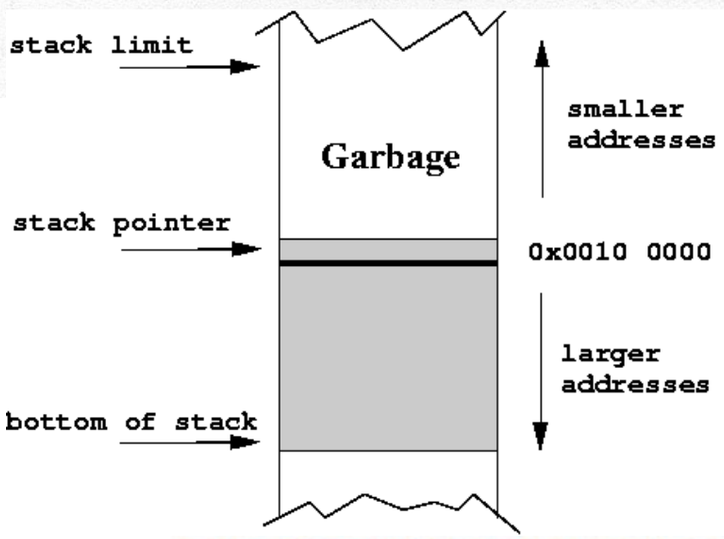




Saving and restoring context

- When we are storing all register contents we are **saving context**.
- **Restoring context** happens when we restore register contents.
- If we did not perform the saving and/or restoring context, then the program would work *most of the times*.
- Some times it would calculate correctly... others it would not!





Disabling interrupts

- Through a single instruction you can tell almost every system to disable (and then enable) interrupts.
- However, most microprocessors have a single pin that contains an interrupt that cannot be disabled. This is the **non-maskable interrupt**.
- If you must use this non-maskable interrupt, and you don't want to get into trouble, make sure the interrupt routine and the task code do not share data.
- Why do you care about these interrupts? For example, you can use them to allow your system to recover from a power failure.

Interrupt priority

- Also, microprocessors can assign a priority to each interrupt request signal.
- This means you can specify the priority of each interrupt.
- You can then disable all interrupts by setting the acceptable priority higher than that of any interrupt.
- You can enable all interrupts by setting the acceptable priority very low.

Common interrupt questions

How does the microprocessor know where to find the ISR?

Q: How does the microprocessor know where to find the interrupt routine when the interrupt occurs?

A: Two different ways:

1. Older microprocessors architectures assume that the interrupt service routine is at a fixed location.

2. On all recent microprocessors there is a table somewhere in memory with all **interrupt vectors**, which contain the addresses of the interrupt routines. So, when an interrupt occurs, the microprocessor will look up the address of the interrupt routine in this interrupt vector table.

How do microprocessors know where the interrupt vector table is?

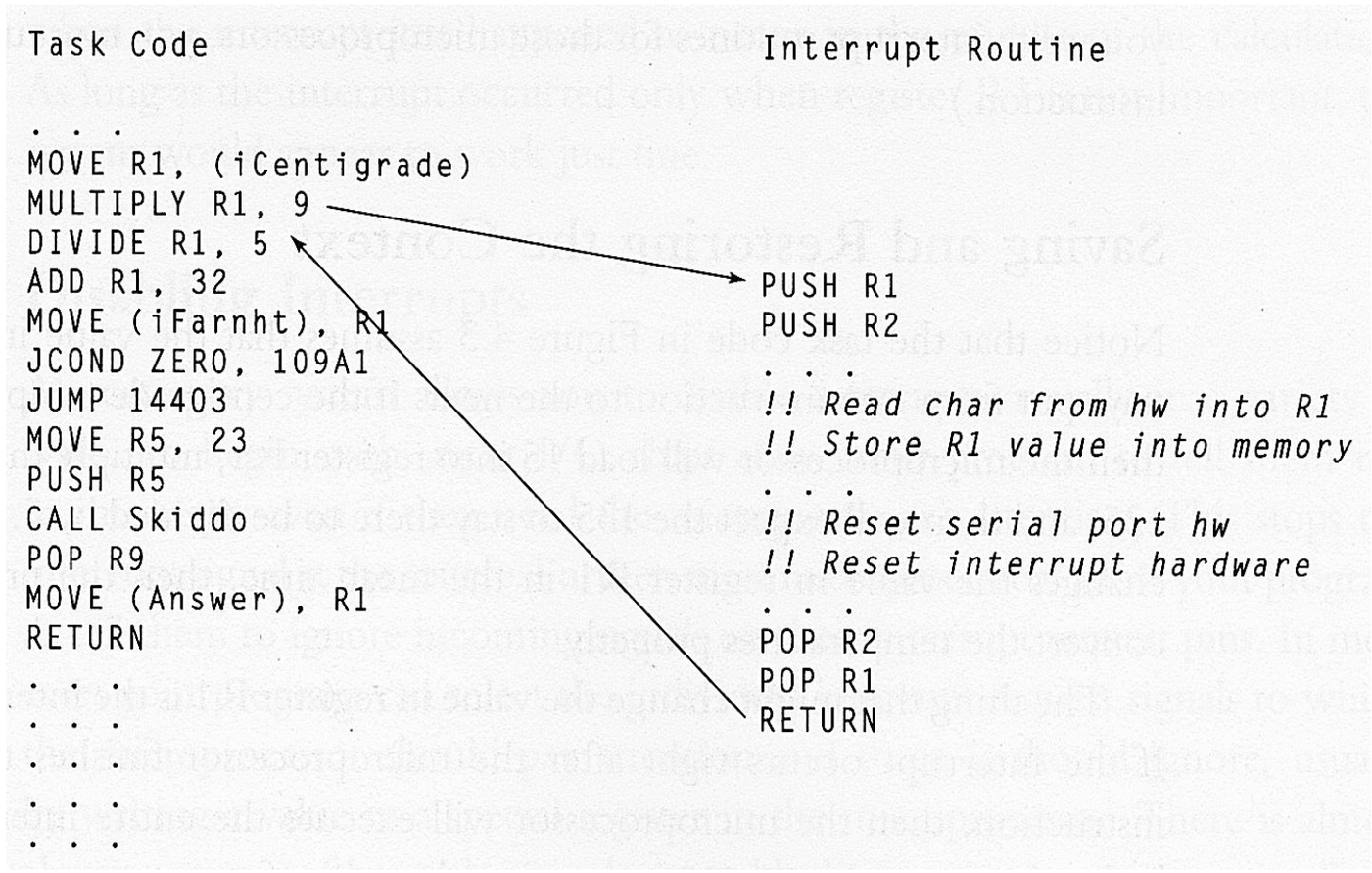
Q: How do microprocessors (that use an interrupt vector table) know where the table is?

A: The interrupt vector table is always at the same location in memory. For the Intel 80186 is at position 0x00000

Can a microprocessor be interrupted in the middle of an instruction?

Q: Can a microprocessor be interrupted in the middle of an instruction?

A: Usually not. In almost every case, the microprocessor will finish the instruction that it is working on before jumping to the interrupt routine.



On two simultaneous interrupts, which one goes first?

Q: If two interrupts happen at the same time, which interrupt routine does the microprocessor do first?

A: Almost every microprocessor assigns a priority to each interrupt signal, and the microprocessor will do the interrupt routine associated with the higher-priority signal first.

Signaled interrupt when interrupts are disabled

Q: What happens if an interrupt is signaled while the interrupts are disabled?

A: In most cases the microprocessor will remember the interrupt until interrupts are reenabled, at which point it will jump to the interrupt routine.

If more than one interrupt is signaled while interrupts are disabled, the microprocessor will do them in priority order when interrupts are reenabled.

Multiple interrupts

Q: Can an interrupt request signal interrupt another interrupt routine?

A: Yes (on most micro-processors). This is called **interrupt nesting**.

A higher-priority interrupt can interrupt a lower-priority interrupt routine. A lower priority interrupt will wait until the higher priority interrupt is done.

The Intel x86 microprocessors disable all interrupts automatically whenever they enter any interrupt routine; therefore, the interrupt routines must reenable interrupts to allow nesting.

I forgot to reenable the interrupts

Q: What happens if I disable interrupts and then forget to reenable them?

A: The micro-processor will execute no more interrupt routines until they are re-enabled.

When interrupts are reenabled, they will sequentially be processed by priority order.

Enabling / disabling interrupts multiple times

Q: What happens if I disable interrupts when they are already disabled or enable interrupts when they are already enabled?

A: Nothing. You just wasted a clock cycle.

Default interrupt state

Q: Are interrupts enabled or disabled when the microprocessor first starts up?

A: Most often, they are disabled.

Can I write my interrupt routines in C?

Q: Can I write my interrupt routines in C?

A: Yes, usually. Most compilers used for embedded-systems code allows you to tell the compiler that a particular function is an interrupt routine. For example:

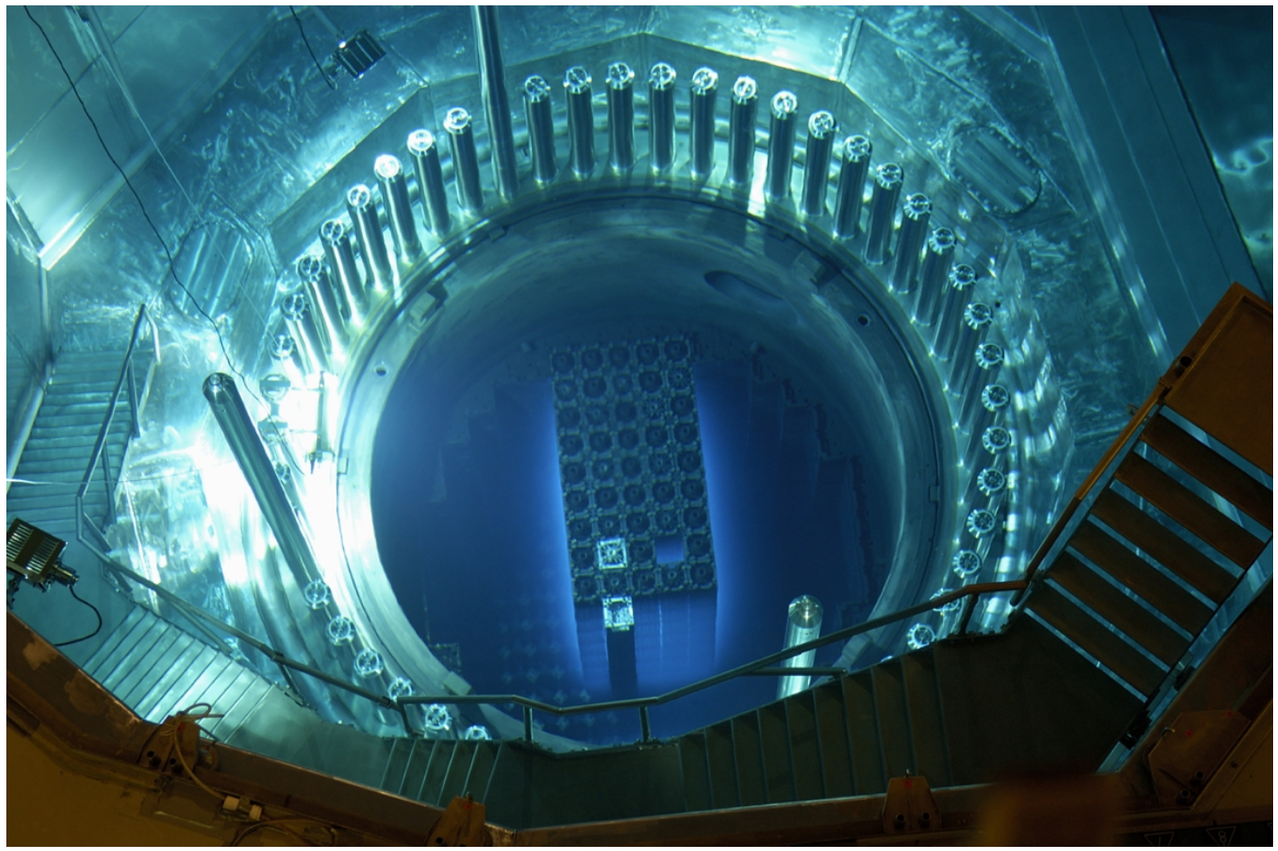
void interrupt vHandleTimerIRQ (void) { **(...)** }

The most common reason for writing interrupt routines in assembly language is that on many micro-processors you can write faster code in assembly language than you can in C.

By the way, you can embed ASM code inside you C code.

Shared-data problem

Premise for the shared-data problem



- This is a nuclear reactor.
- Periodic temperature measurements occur at two separate locations.
- If the temperature at these two locations are different then sound the alarm!
- Different temperatures means we may have a nuclear fallout!

CPE 355 - Real Time Embedded Kernels - Spring '12

Nuno Alves (nalves@wne.edu), College of Engineering

WESTERN NEW ENGLAND UNIVERSITY | WNE

Code #1

Shared-data problem

- One big problem arises as soon as your interrupt routines need to communicate with the rest of your code.
- It is often not possible or desirable for microprocessors to complete all its work in interrupt routines.
- The interrupt routines and the task code must share one or more variables so that they can communicate with one another.

```
static int iTemperatures[2];

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

void main (void) {
  int iTemp0, iTemp1;
  while (TRUE) {
    iTemp0 = iTemperatures[0];
    iTemp1 = iTemperatures[1];
    if (iTemp0 != iTemp1)
    { !! Set off howling alarm }
    }
}
```

Notation to make life easier

The !! means I am omitting some nasty code that we don't need to know how to implement... We just need to know the outcome.

```
static int iTemperatures[2];

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

void main (void) {
  int iTemp0, iTemp1;
  while (TRUE) {
    iTemp0 = iTemperatures[0];
    iTemp1 = iTemperatures[1];
    if (iTemp0 != iTemp1)
    { !! Set off howling alarm }
    }
}
```

What is the problem with this code?

- Suppose both temperatures have been 73 degrees for a long time.
- iTemperatures[0]=73 and iTemperatures[1]=73
- Suppose an interrupt happens here, right after iTemp0=73
- Now both temperatures have increased to 74.
- When interrupt resumes iTemp1=74 but iTemp0 is still 73!

```
static int iTemperatures[2];

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

void main (void) {
  int iTemp0, iTemp1;
  while (TRUE) {
    iTemp0 = iTemperatures[0];
    iTemp1 = iTemperatures[1];
    if (iTemp0 != iTemp1)
    { !! Set off howling alarm }
    }
}
```

Code #2

Shared-data problem

- What about this code? Does it solve the problem?
- Unfortunately it doesn't... Remember a single instruction in C, may be many assembly instructions.

```
MOVE R1, iTemperatures[0])
MOVE R2, (iTemperatures[1])
SUBTRACT R1,R2
JCOND ZERO, TEMPERATURES_OK
;  Code to sound the alarm

TEMPERATURES_OK:
```

```
Problem static int iTemperatures[2];
void interrupt vReadTemperatures  (void)
{
     iTemperatures[0] = !!  read from hardware
     iTemperatures[1] = !!  read from hardware
}

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

- Look at the equivalent ASM code.
- If an interrupt occurs after the line `MOVE R1, iTemperatures[0])`, the same problem will arise.

Shared-data problem

- What about this code? Does it solve the problem?
- Unfortunately it doesn't... Remember a single instruction in C, may be many assembly instructions.

```
MOVE R1, iTemperatures[0])
MOVE R2, (iTemperatures[1])
SUBTRACT R1,R2
JCOND ZERO, TEMPERATURES_0K
;  Code to sound the alarm

TEMPERATURES_OK:
```

```
Problem static int iTemperatures[2];
void interrupt vReadTemperatures  (void)
{
     iTemperatures[0] = !!  read from hardware
     iTemperatures[1] = !!  read from hardware
}

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

- Look at the equivalent ASM code.
- If an interrupt occurs after the line `MOVE R1, iTemperatures[0])`, the same problem will arise.

Characteristics of the shared data problem

- The problem with both codes is that the iTemperatures array is shared between the interrupt routine and the task code.

- Shared data bugs are difficult to find since they do not happen every time the code runs.

- Whenever an interrupt routine and your task code share data, carefully study the problem so that you do not have a shared-data bug.

Solving the shared data problem

Solving shared data problem with method #1

- We can disable interrupts whenever the task code uses shared data.
- While the interrupts are disabled, the hardware can assert the interrupt signal requesting service ... but the microprocessor will not jump to the interrupt routine.

```
static int iTemperatures[2];
void interrupt vReadTemperatures  (void)
{
  iTemperatures[0] = !!read some data
  iTemperatures[1] = !!read some data
}

void main  (void)
{
  int iTemp0, iTempi;
  while (TRUE) {
  //Disable interrupts while we use the array
  disable ();
  iTemp0 = iTemperatures[0];
  iTemp1 = iTemperatures[1];
  //Enable interrupts again
  enable ();
  if  (iTemp0 != iTemp1)!! Set off howling
alarm;
    }
}
```

Method #1 using assembler

- No C compilers or assemblers are smart enough to figure out when it is necessary to disable interrupts.

```
;disable interrupts
DI

;use the arrays
MOVE R1,(iTemperature[0])
MOVE R2,(iTemperature[1])

;enable interrupts again
EI
SUBTRACT R1, R2
JCOND ZERO,TEMPERATURES_OK

;  Code goes here to set off the alarm

TEMPERATURES_OK:
```

```
static int iTemperatures[2];
void interrupt vReadTemperatures  (void)
{
  iTemperatures[0] = !!read some data
  iTemperatures[1] = !!read some data
}

void main  (void)
{
  int iTemp0, iTempi;
  while (TRUE) {
  //Disable interrupts while we use the array
  disable ();
  iTemp0 = iTemperatures[0];
  iTemp1 = iTemperatures[1];
  Enable interrupts again
  enable ();
  if  (iTemp0 != iTemp1)!! Set off howling
alarm;
    }
}
```

"Atomic" sections

- A part of a program is said to be **atomic** if it cannot be interrupted.
- This is the atomic section of this program.
- The shared-data problem arises when an interrupt routine and the task code share data, and the task code uses the shared data in a way that is not atomic.

```
static int iTemperatures[2];
void interrupt vReadTemperatures  (void)
{
  iTemperatures[0] = !!read some data
  iTemperatures[1] = !!read some data
}

void main  (void)
{
  int iTemp0, iTempi;
  while (TRUE) {
  //Disable interrupts while we use the array
  disable ();
  iTemp0 = iTemperatures[0];
  iTemp1 = iTemperatures[1];
  Enable interrupts again
  enable ();
  if  (iTemp0 != iTemp1)!! Set off howling
alarm;
    }
}
```

"Critical" sections

- The disable/enable interrupt functions that use the shared data, is what made that section atomic.

- A set of instructions that **must** be atomic for the system to work properly is often called a **critical section**.

- This particular atomic section, is also critical!

```
static int iTemperatures[2];
void interrupt vReadTemperatures  (void)
{
  iTemperatures[0] = !!read some data
  iTemperatures[1] = !!read some data
}

void main  (void)
{
  int iTemp0, iTempi;
  while (TRUE) {
  //Disable interrupts while we use the array
  disable ();
  iTemp0 = iTemperatures[0];
  iTemp1 = iTemperatures[1];
  Enable interrupts again
  enable ();
  if  (iTemp0 != iTemp1)!! Set off howling
alarm;
    }
}
```

Another shared data problem example

Interrupts with a timer

- The function ISecondsSinceMidnight returns the number of seconds since midnight.

- A hardware timer asserts an interrupt signal every second, which causes the microprocessor to run the interrupt routine vUpdateTime to update the static variables that keep track of the time.

```
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 other stuff in hardware
}

long ISecondsSinceMidnight  (void)
{
    return  ( (((iHours * 60) + iMinutes) *
60) + iSeconds);
}
```

Where is the problem?

- If the hardware timer interrupts while the microprocessor is doing the arithmetic in lSecondsSinceMidnight, then the result might be wrong.

```
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 other stuff in hardware
}

long lSecondsSinceMidnight  (void)
{
    return  ( (((iHours * 60) + iMinutes) *
60) + iSeconds);
}
```

How bad can it be? Very bad!

- Suppose that the time is 3:59:59.
- The function ISecondsSinceMidnight might read iHours as 3.
- However, if the interrupt occurs and changes the time to 4:00:00.
- ISecondsSinceMidnight will read iMinutes, and iSeconds as 0.
- Returning a value that makes it look as though the time is 3:00:00, which is 1 hour off!!!

```
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 other stuff in hardware
}

long ISecondsSinceMidnight  (void)
{
     return  ( (((iHours * 60) + iMinutes) *
60) + iSeconds);
}
```

An even worse solution!

Why is this so bad?

```
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 other stuff in hardware
}

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

Why is this so bad?

- Well... The function *ISecondsSinceMidnight* will always return before enabling the interrupts.
- Interrupts will never be enabled, so the time obtained through the function *vUpdateTime* will never change.

```
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 other stuff in hardware
}

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

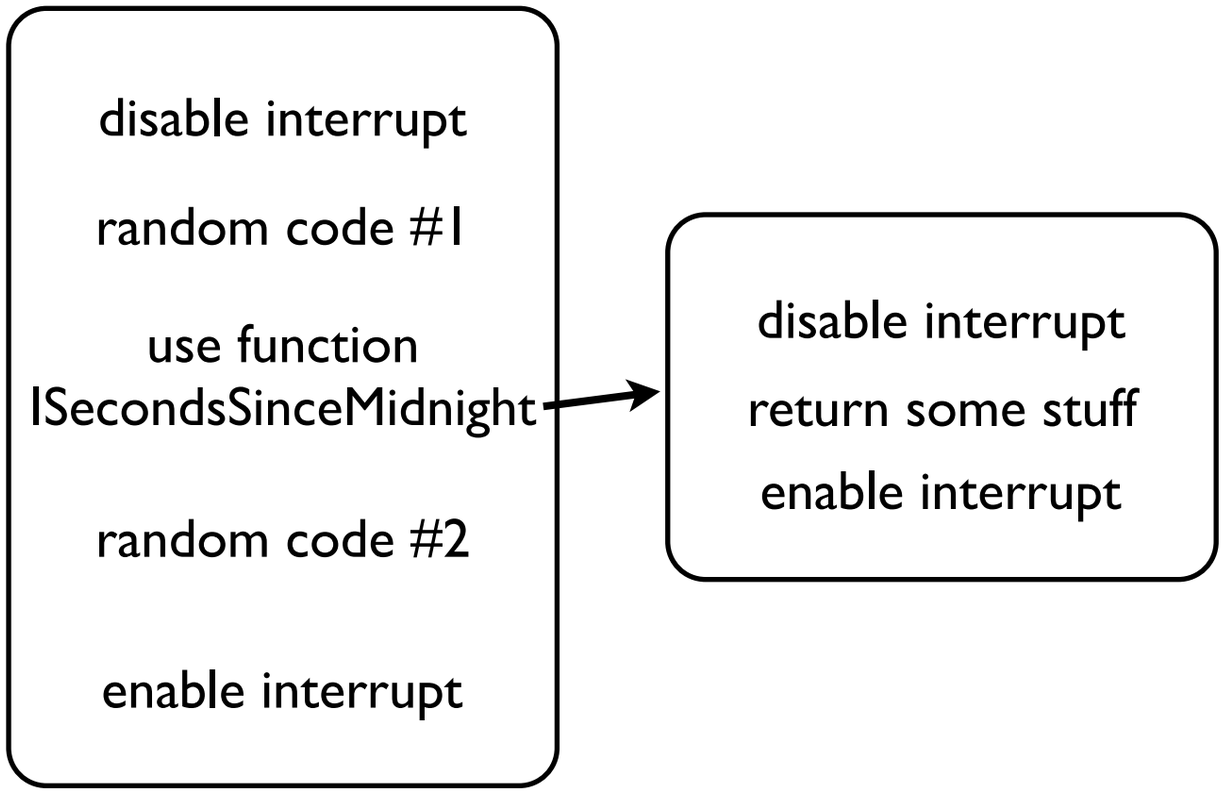
Slightly better solution... but still has problems

- Suppose that ISecondsSinceMidnight is called from inside a critical section somewhere else in the program
- Critical section means that interrupts are disabled.
- A bug will be caused since we are now enabling interrupts inside a critical section.

```
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 other stuff in hardware
}

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

Visual representation




```
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 other stuff in hardware
}

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

- If an interrupt happens sometime during *random code #2*, then it will be processed immediately!

The “best” solution

```
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 other stuff in hardware
}

long lSecondsSinceMidnight (void) {
    long lReturnVal;
    bool fInterruptStateOld;
    fInterruptStateOld = disable (); //are interrupts already disabled?

    IReturnVal =(((iHours * 60) + iMinutes) * 60) + iSeconds;
    if (flnterruptStateOld) { enable (); } //Restore interrupts to previous state

    return  (lReturnVal);
}
```

- If interrupts are already disabled, don’t disable / enable them again!
- The disable() function not only disables the interrupts, but it will also tell you if they are already disabled.
- A bit slower, than previous code.

Interrupt latency

Interrupt latency

- Because interrupts are a tool for getting better response from our systems, and because the speed which an embedded system can respond is always of interest ...
- How fast does my system respond to each interrupt?
- **Interrupt latency** is the amount of time it takes for a system to respond to an interrupt.

How fast does my system respond to an interrupt?

Depends on 4 factors:

1. The longest period of time where that interrupt is disabled.

2. The time it takes to execute the interrupt routines for interrupts that are of higher priority than the one in question.

3. How long it takes the microprocessor to stop what it is doing, do the necessary bookkeeping, and start executing instructions within the interrupt routine.

4. How long it takes the interrupt routine to save the context and then run the interrupt code.

Get some timing information

How do I get the times associated with the four factors listed on the previous slide?

- Read microprocessor documentation.
- Write the code and measure how long it takes to execute.
- Count the instructions of various types and how long each type of instruction takes.

If I want to make my system to respond faster to interrupts, the shorter the period during which interrupts are disabled, the better my response will be.

Example #1

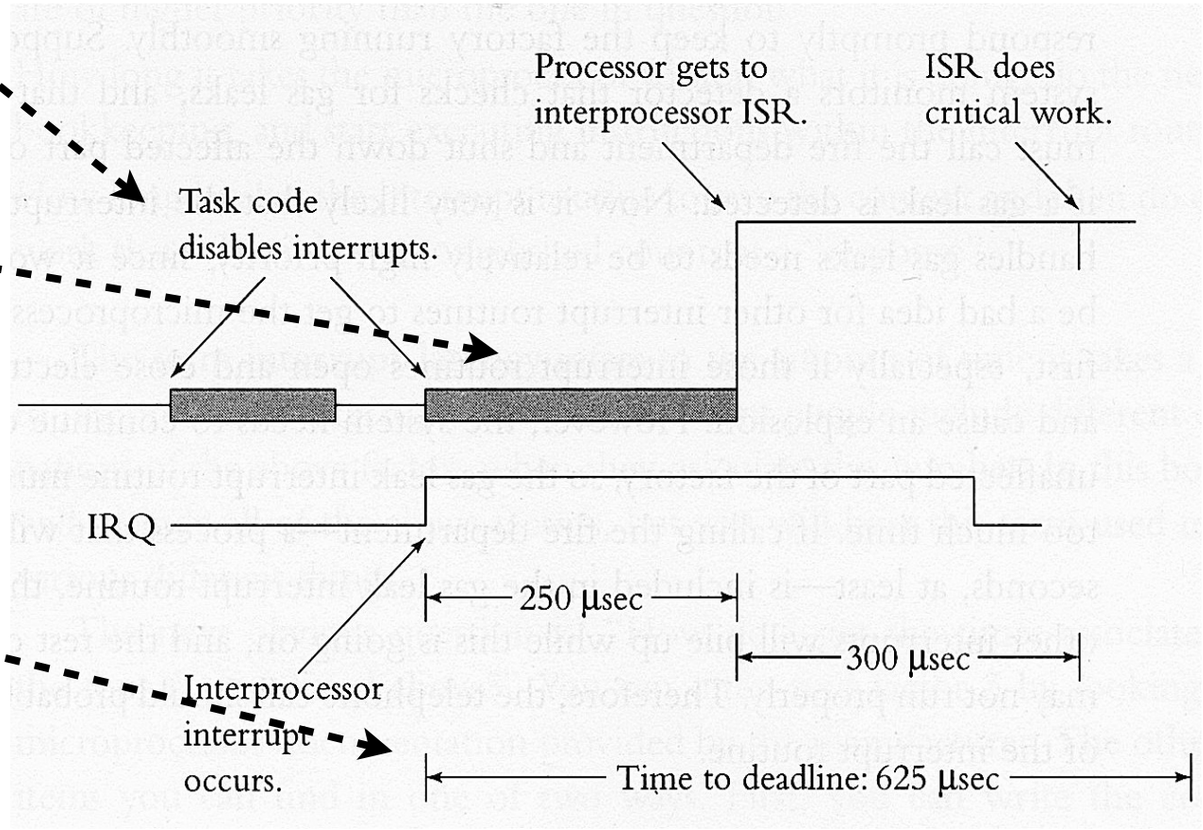
Suppose that the requirements for your system are as follows:

- Disable interrupts for 125µsec while you run **random code A**.
- Disable interrupts for another 250µsec while you some **random code B**.
- Whenever some other processor asks for resources, you must give them within 625µsec.
- It takes 300µsec for your system to give the other processor his desired resources (interprocessor interrupt routine).

Question: Is it possible to implement this system?

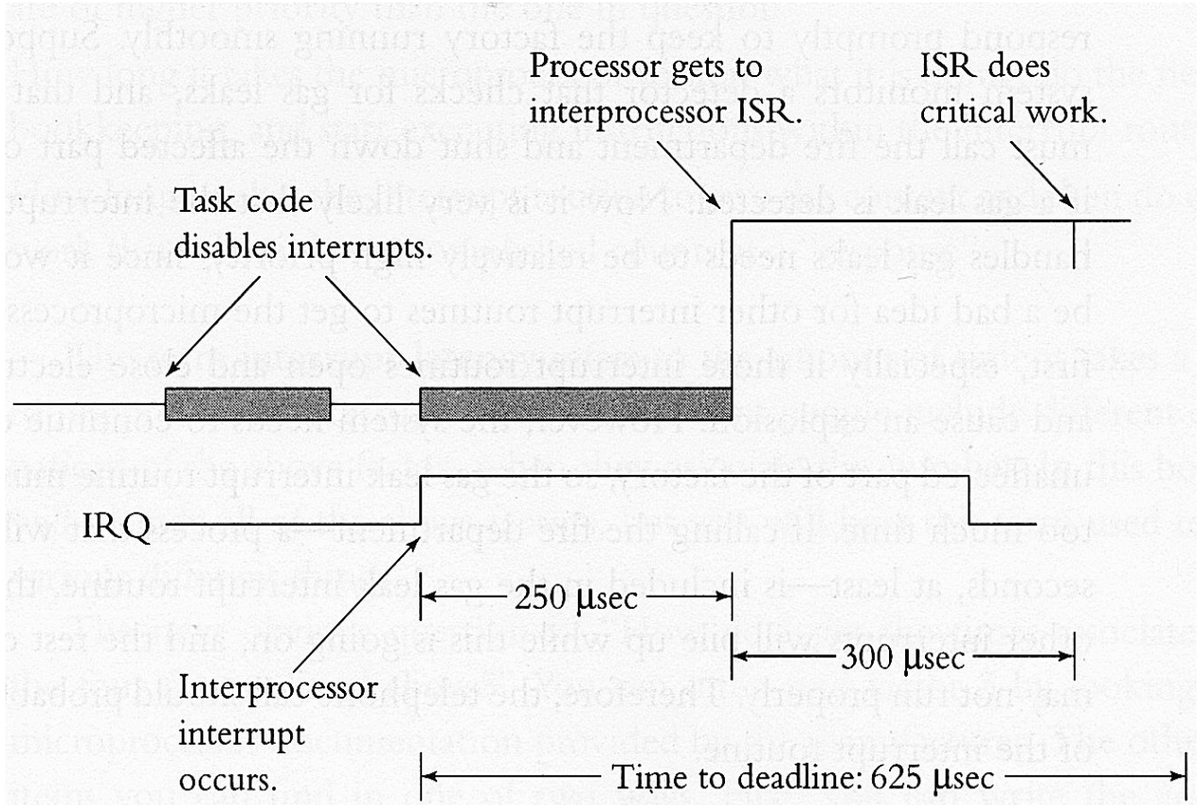
Worst case interrupt latency

- Disable interrupts for 125µsec while you run **random code A**.
- Disable interrupts for another 250µsec while you some **random code B**.
- Whenever some other processor asks for resources, you must give them within 625µsec.
- It takes 300µsec for your system to give the other processor his desired resources (interprocessor interrupt routine).



We can definitely implement this system

- Interrupts are disabled in our system for at most 250µsec.
- The worst case scenario happens when the longest period when interrupts are disabled (the 250µsec mentioned before).
- The interrupt routine needs 300µsec to run.
- The longest time it can take to run is 300µsec+250µsec = 550µsec.
- This is within the desired response time of 625µsec.



Example #2

- Will this system work with a processor that is half the speed?
- This means all the processing times are doubled: interrupts are disabled for twice as long, the interrupt service routine takes twice as long, but the 625-µsec deadline remains the same.
- Will the system meet its deadline? **No!**
- Interrupts will be disabled for up to 500 µsec at a time, and the interrupt service routine needs 600µsec to do its work.
- The total of is 1100 µsec, much longer than the 625-µsec deadline.

Example #3

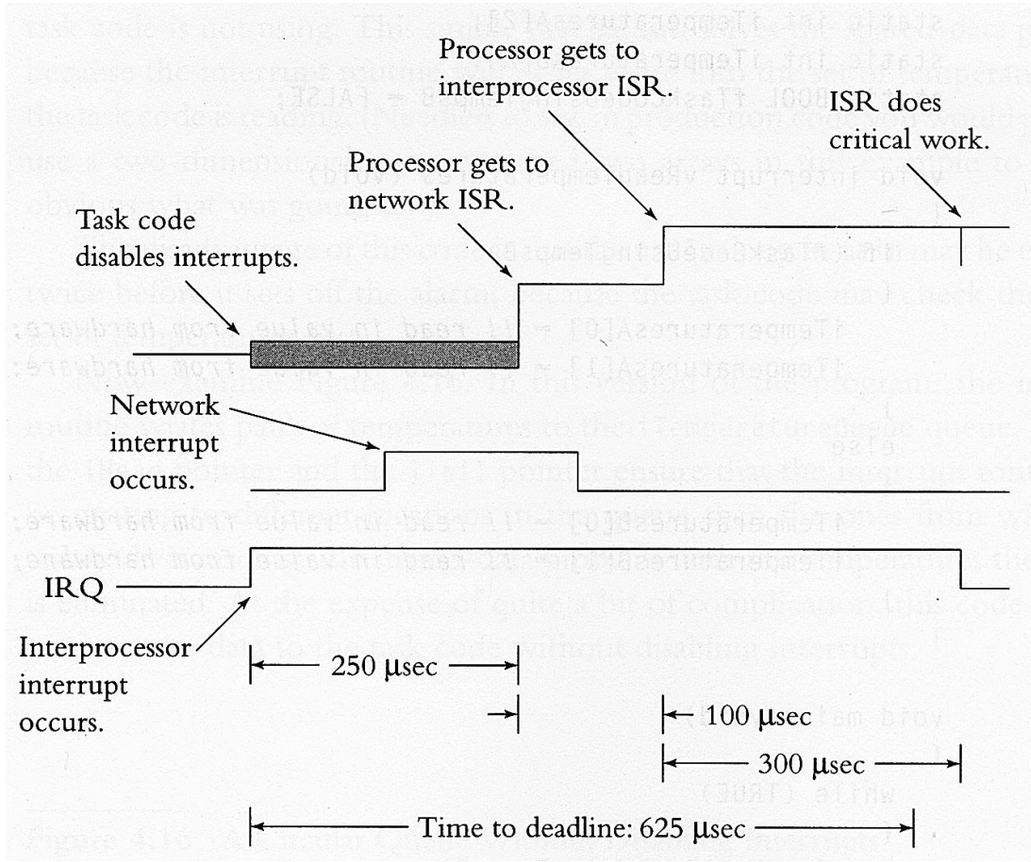
- Some random code requires interrupts to be disabled for 250µsec.
- A network card was added to the system, and its hardware interrupt takes100µsec to process.
- Whenever some other processor asks for resources, you must give them within 625µsec.
- It takes 300µsec for your system to give the other processor his desired resources (interprocessor interrupt routine).

Question: Is it possible to implement this system? **It depends...**

Importance of interrupt priorities

- Interrupts disabled for 250µsec.
- A network card interrupt takes100µsec to process.
- Time to deadline 625µsec.
- Interprocessor interrupt routine takes 300µsec to process.

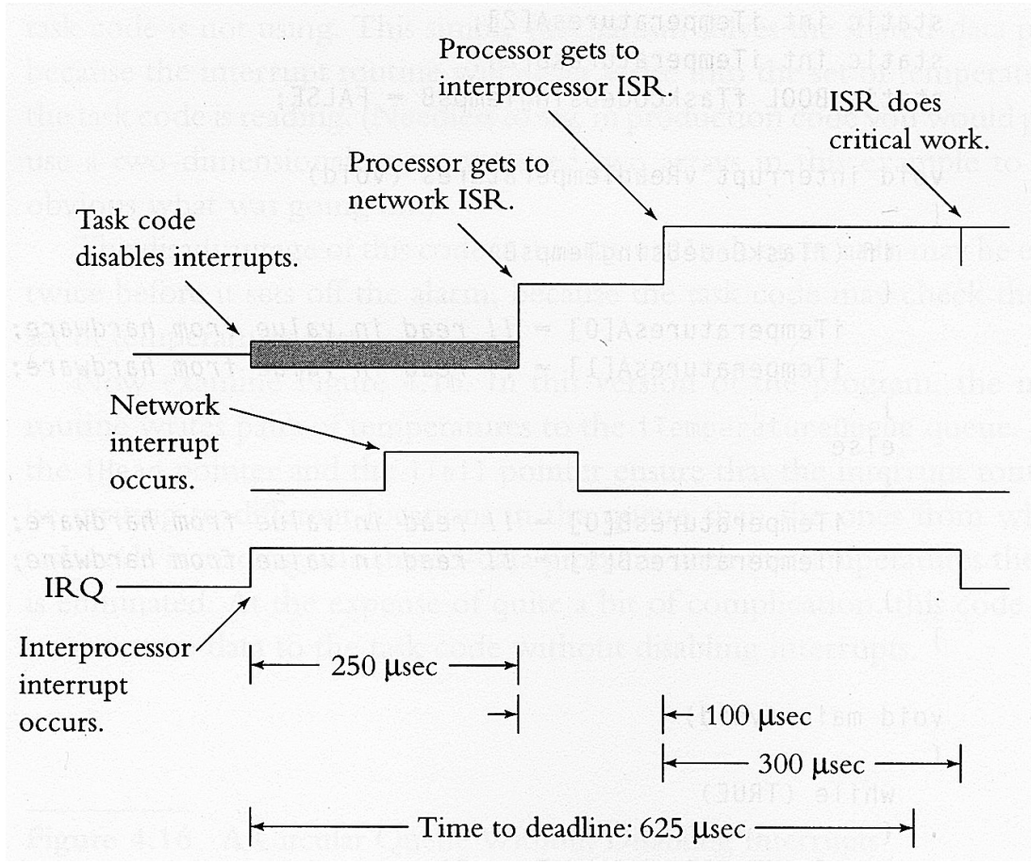
  Only if I assign a lower priority to the network card we will be able to meet the deadline!



In this diagram, the network card is assigned a higher priority and the deadline will NOT be met!

Why won't the deadline be met if the network ISR has a higher priority?

- In most micro-controllers, only one thing can be running at the same time!
- Higher priority means an interrupt will be executed first.
- If the Network ISR takes 100µsec to be done, the Processor ISR will have to wait until done.
- If the Processor ISR has higher priority, it will be done first, and it could even interrupt the Network ISR.



In this diagram, the network card is assigned a higher priority and the deadline will NOT be met!

Polling

I/O Systems

- Many devices are operating independently of the processor – except when communication happens
- We say that these devices are acting **asynchronously** of the processor
- The processor must have some way of knowing that something has changed with the device (e.g., that it is ready to send or receive information).
- Up till now we discussed interrupts.

Polling is a bad way to check the state of the device

With polling, the processor continually checks the state of the device:

```
do {
    x = PINB & 0x10;
}while(x == 0);
y = PINC …
```

This piece of code will loop until x is different than 0.

Why is polling so bad?

- In embedded systems, we are typically managing many devices at once.
- We can potentially be waiting for a long time before the state changes.
- We call this busy waiting.
- The processor is wasting time that could be used to do other tasks.

I/O By Polling: An Alternative

One alternative is to do something while we are waiting...

```
do {
    x = PINB & 0x10;
    <go do something else>
}while(x == 0);
y = PINC …
```

Polling works great ... but:

- We have to guarantee that our "something else" does not take too long (otherwise, we may miss the event)

- Depending on the device, "too long" may be very short

When to use polling instead of interrupts?

In generally we avoid polling I/O like the plague!

However, we use polling approach for situations in which:

- We don't want to deal with shared data problems in interrupts.
- We know the event is coming very soon (nsec to µsec range).
- We must respond to the event very quickly (nsec to µsec range).