



Interrupts

Simon Chapter 4

Microprocessor architecture, IRQs, ISRs,
saving and restoring context, disabling, the
shared-data problem, latency

Microprocessor Architecture

- Instructions
 - Instructions in Assembly Language
 - Assembler converts assembly language into binary numbers for execution
 - Instructions in higher level language get reduced to one or more assembly instructions
- Registers
 - General-purpose (R1, R2, etc.)
 - Program Counter – address of next instruction to be executed
 - Stack Pointer – memory address of next available location on stack

Assembly Language

- Moving data from one place to another

MOVE R3, R2 (DST <- SCR copies value of R2 into R3)
MOVE R5, (iTemperature) (copies value of iTemperature into R5)
MOVE R5, iTemperature (copies address of iTemperature into R5)

OUR PROCESSOR---

mov.b R15, R10 (SCR -> DST copies value of R15 into R10)

- Other Operations

ADD R7, R3 (adds R7 + R3 and stores result in R7)
NOT R4 (inverts value of all bits in R4)

- Calling Subroutines/Functions

- (see example)

Calling Turn_LED_ON()

```

    Turn_LED_ON();
00F8A8  12B0 F8C2      call    #Turn_LED_ON

```

```

    P10UT |= LED_ON;
Turn_LED_ON:
00F8C2  D3D2 0021      bis.b   #0x1,&P10UT
}
00F8C6  4130          ret
    P10UT &= LED_OFF;
Turn_LED_OFF:
00F8C8  C3D2 0021      bic.b   #0x1,&P10UT
}

```

Interrupts Basics

- A signal from hardware that it needs a “service”
- IRQ – Interrupt Request – pin to uP
- Routine to provide a “service”
 - “Interrupt service routine”
 - “ISR”
 - “INT”
 - “Interrupt handler”
- An interrupt temporarily halts processing of our code example.

Saving the Context

- Interrupts use registers, too.
- What happens if interrupt wants to use R14 and R15?
- First part of ISR is to save registers to stack
 - "Saving the Context"
 - "Pushing stuff on the stack"
- ISR does its "service" processing
- Finally, ISR restores register's values from stack
 - "Restoring the Context"
 - "Popping stuff off the stack"
- ISR "Returns"

Disabling Interrupts

- Why?
 - May need to do something without interruption
 - To protect data that both might access
 - Priorities
- Usually one instruction (DINT, EINT)
- Nonmaskable interrupt – cannot be masked
 - For catastrophic events
 - Should not share data

Questions about Interrupts

- How does the processor know where to find the interrupt code with an interrupt occurs? (Vector table)
- Can a microprocessor be interrupted in the middle of an ASM instruction? (usually not)
- Can a microprocessor be interrupted in the middle of a C instruction? (usually yes)
- If two interrupts happen at the same time, which is handled first? (priority levels)
- Can an interrupt be interrupted by another interrupt? (often yes – interrupt nesting, often configurable)
- What happens if an interrupt is signaled when interrupts are disabled? (variable. May ignore, may service later)
- What happens if I disable interrupts and forget to reenale them? (Usually system failure sooner or later)

The Shared-Data Problem

- The regular code and the interrupt code will use some of the same variables.
- For example, interrupt collects temperature data periodically
- Main routine evaluates the temperature values
- Nuclear Reactor Monitoring System Example –
 - ISR collects temperature values in two places
 - Normally, temperature values should be the same.
 - If temperatures NOT equal, catastrophic meltdown could be beginning, so set off alarm.

The Shared-Data Problem

```
static int iTemperatures[2];

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

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

Fixing the Problem...

```
static int iTemperatures[2];

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

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

Assembly Language Equivalent

```
MOVE          R1, iTemperatures[0]
MOVE          R2, iTemperatures[1]
SUBTRACT      R1, R2
JCOND         ZERO, TEMPERATURES_OK
.
.
; code to set off alarm goes here
.
.
TEMPERATURES_OK:
.
.
```

Does this really happen?

- These are “fiendish bugs”
- Don’t occur all the time
- Very timing critical – can be within a fraction of a microsecond to occur
- When these bugs tend to occur:
 - Friday at 5pm,
 - when no debugger available,
 - when products lands on Mars,
 - during customer demos.
- “Because these bugs often show themselves only rarely an are therefore difficult to find, it pays to avoid putting these bugs into your code in the first place.” – D. Simon

Fixing the Problem for Real

```
static int iTemperatures[2];

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

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

Assembly Language Equivalent

```
DI          ; disable interrupts
MOVE       R1, iTemperatures[0]
MOVE       R2, iTemperatures[1]
EI          ; enable interrupts again
SUBTRACT   R1, R2
JCOND     ZERO, TEMPERATURES_OK
.
.
; code to set off alarm goes here
.
.
TEMPERATURES_OK:
.
.
```

“Atomic” and “Critical Section”

- “Atomic” – parts of the program that cannot be interrupted
- Problems happen when task code uses shared data *in a way that is not atomic*
- “Critical section” – the parts of the code that must be atomic for the system to work correctly.
- “Volatile” – keyword to tell compiler not to optimize – do not assume variable value does not change in memory.

Interrupt Latency

- The amount of time it takes for the system to respond to the interrupt
- How long does this take?
- Depends...
 - Length of any current interrupts in progress
 - Length of other higher priority interrupts
 - Time for uP to finish current housekeeping
 - Time for ISR to save context and jump

Make Interrupt Routines SHORT!

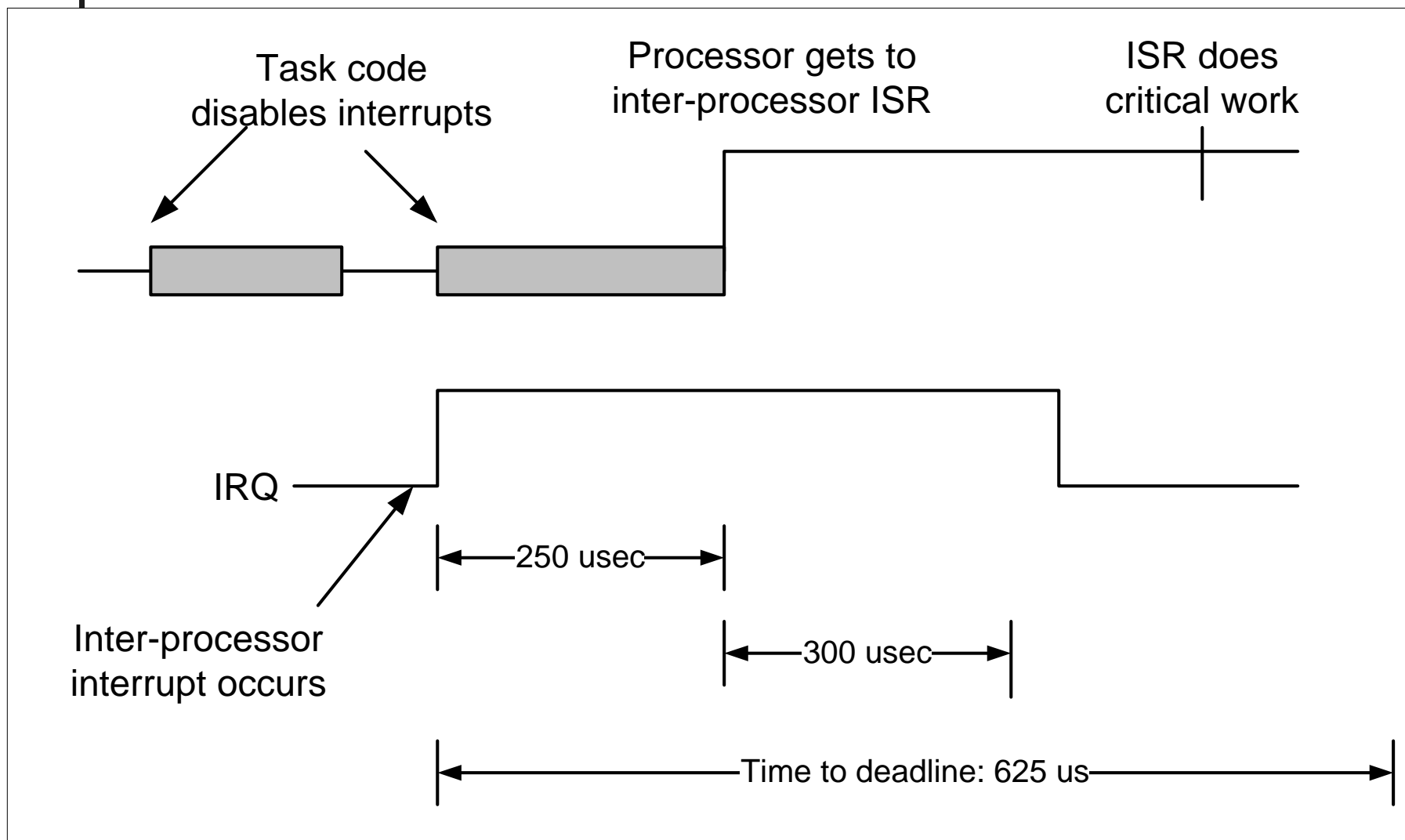
- Time burned in ISR affects timing of the rest of the system.
- Periodic ISRs eat duty cycle
- Set relative priorities logically

- Example – ISR in response to gas leak should not call fire department within the ISR.

Disabling Interrupts

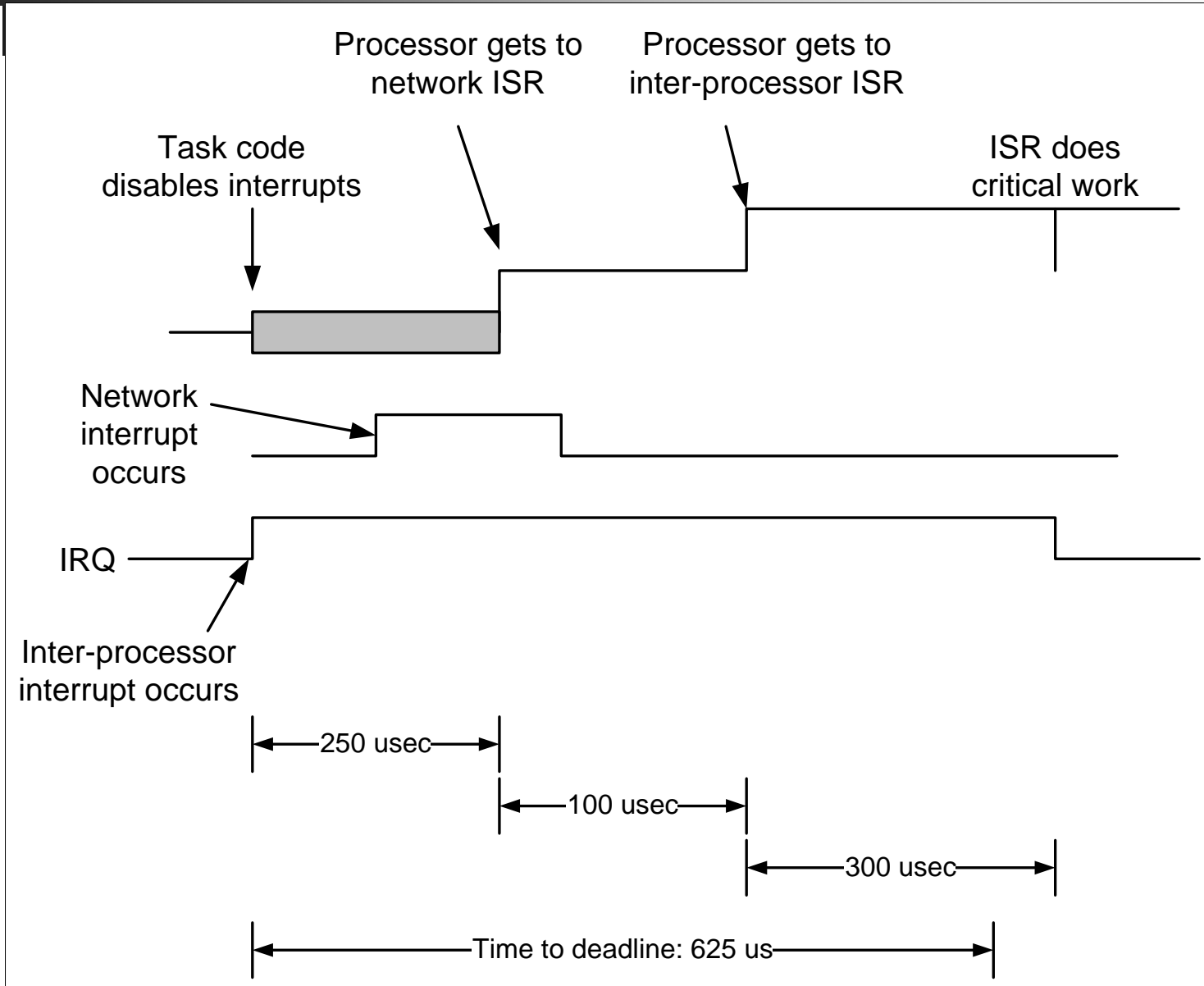
- Major cause of latency
- Find a balance for shared-data problem
- Example:
 - Disable interrupts for 125 usec to use temperature variables
 - Disable interrupts for 250 usec to store timer values
 - Must respond within 625 usec after “special signal”, interprocessor routine takes 300 usec to execute
 - Can this all work?

Worst Case Interrupt Latency



What happens if HW specs processor half as fast?

Added Network Capability



Interrupts on the MSP430

- #pragma directive used to assign interrupt vectors
 - #pragma tells compiler how to compile something
 - Describes source of interrupt
 - #define TIMERA0_VECTOR (9*2u) /*0xFFF2 Timer A CC0 */

```
// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
    P1OUT ^= 0x01;           // Toggle P1.0
    CCR0 += reload;         // Add Offset to CCR0
}
```