

INTERRUPTS

Embedded Software Design

熊博安

國立中正大學資訊工程研究所

pahsiung@cs.ccu.edu.tw

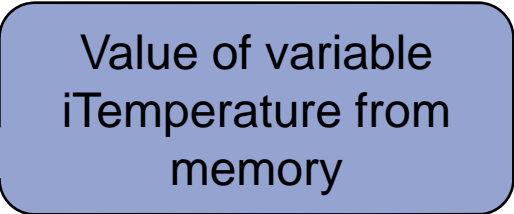
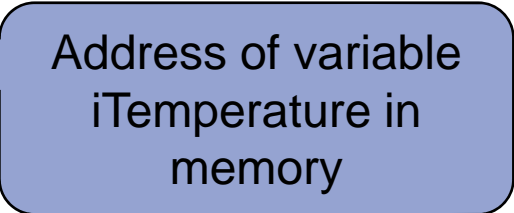

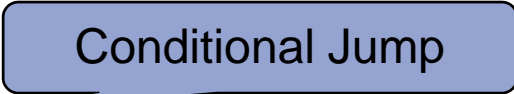
Contents

- Microprocessor Architecture
- Interrupt Basics
- Shared-Data Problem
- Interrupt Latency

Microprocessor Architecture

- Assembly language = human-readable microprocessor instructions
- 1 assembly instruction = 1 CPU instruction
- 1 C statement = 1 or more CPU instructions
- Every family of microprocessors has a different assembly language
- Microprocessor registers:
 - general-purpose: R1, R2, R3, ...
 - special: program counter, stack pointer, ...

Assembly-Language Instructions

- MOVE R3, R2
- MOVE R5, (iTemperature) 
- MOVE R5, iTemperature 
- ADD R7, R3
- NOT R4
- JUMP NO_ADD 
- NO_ADD: MOVE (xyz), R1
- SUBTRACT R1, R5
- JCOND ZERO, NO_MORE 

Assembly-Language Instructions

- CALL ADD_EM_UP
 - MOVE (xyz), R1
 - ...
 - ADD_EM_UP:
 - ADD R1, R3
 - ADD R1, R4
 - ADD R1, R5
 - RETURN
-
- ```
graph TD; CALL[CALL ADD_EM_UP] -- green --> ADD_EM_UP[ADD_EM_UP:]; ADD_EM_UP -- green --> ADD[ADD R1, R3]; ADD -- green --> ADD2[ADD R1, R4]; ADD2 -- green --> ADD3[ADD R1, R5]; ADD3 -- green --> RETURN[RETURN]; RETURN -- red --> MOVE[MOVE (xyz), R1];
```

# C and Assembly Language

**Figure 4.1** C and Assembly Language

```
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

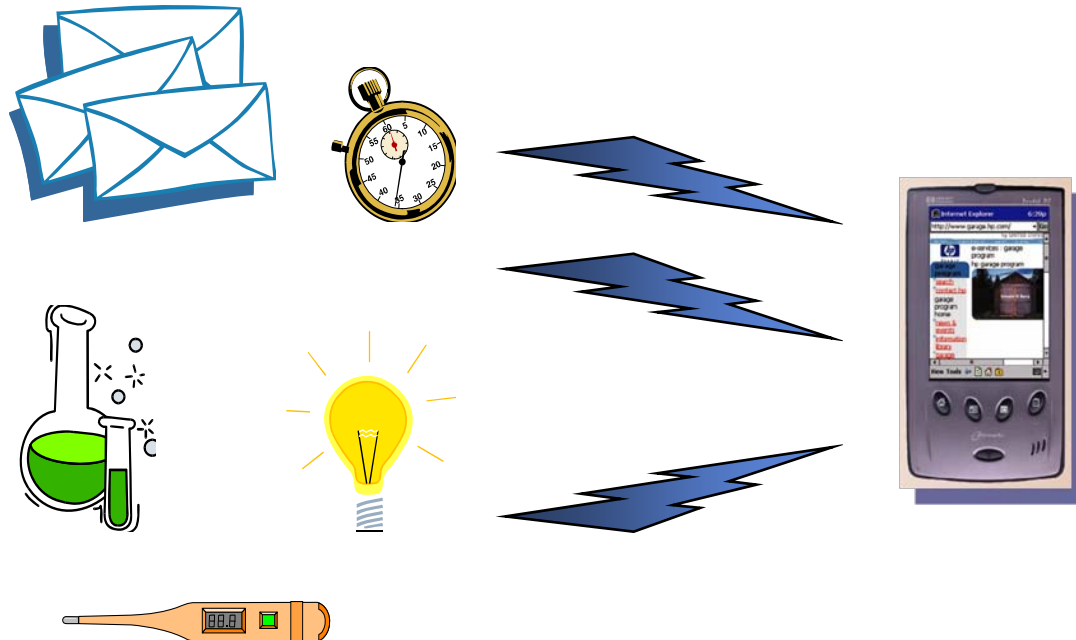
if (x >= z)
 MOVE R2, (z) ; Get the value of z
 SUBTRACT R1, R2 ; Subtract z from x
 JCOND NEG, L101 ; Skip if the result is negative

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
```

# Context Awareness

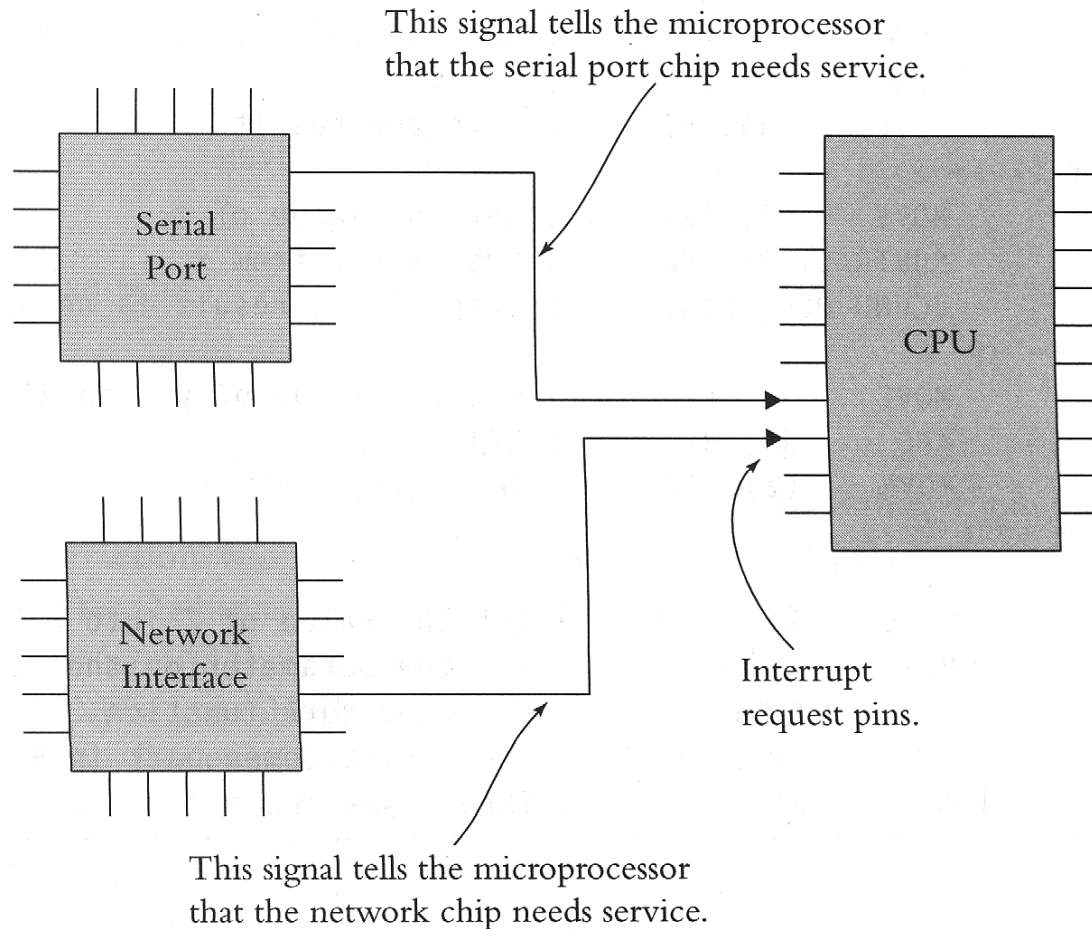
- Since embedded systems are closely relevant to the contexts, how can these external events be noticed?



# I/O Interrupts

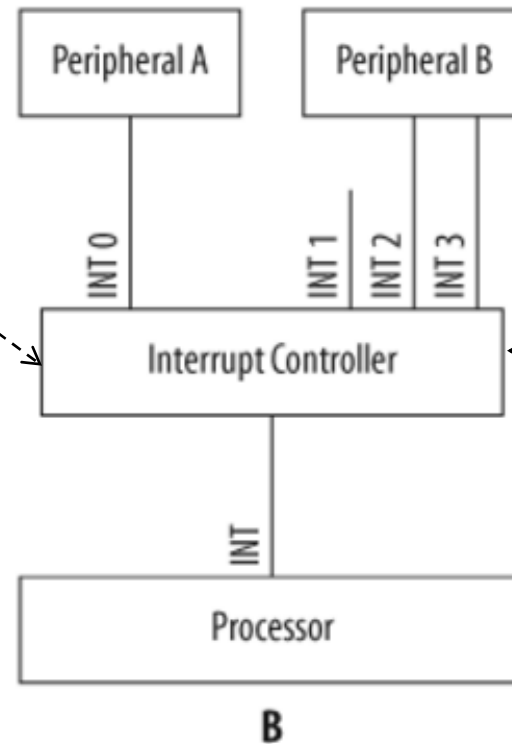
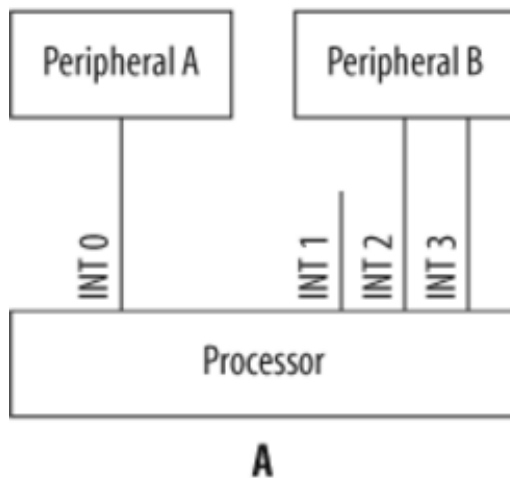
- The most common approach is to use interrupts.
  - Interrupts cause the microprocessor in the embedded system
    - to suspend doing whatever it is doing, and
    - to execute some different code instead.
- Interrupts can solve the response problem, but not without some difficult programming, and without introducing some *new problems* of their own.

# Interrupt Hardware



# Interrupt Wiring

Functions: disabling, prioritizing, showing active interrupts



PXA255 has an internal interrupt controller

# Partial interrupt list (PXA255)

---

## Interrupt number Interrupt source

|    |            |
|----|------------|
| 8  | GPIO Pin 0 |
| 9  | GPIO Pin 1 |
| 11 | USB        |
| 26 | Timer 0    |
| 27 | Timer 1    |
| 28 | Timer 2    |

# Exception vs. Interrupt vs. Trap

---

## ■ Exception

- Software interrupt: synchronous event
- Eg: divide by zero

## ■ Interrupt

- Hardware interrupt: asynchronous event
- Eg: Timer

## ■ Trap

- Internal interrupt: synchronous event
- Eg: undefined instruction

# Interrupt Basics

---

- Microprocessor **detects** interrupt request (IRQ) signal is asserted
- **Stops** executing instructions
- **Saves** on stack the address of next instruction
- **Jumps** to interrupt service routine (ISR) and executes it
- **Returns** from ISR
- **Pops** address from stack
- **Continues** execution of next instruction

# Interrupt Service Routine (ISR)

- Do whatever **needs** to be done when the interrupt signal occurs
- E.g.: character **received** at serial port chip →
  - **read** character from chip
  - **put** it into memory
- Miscellaneous **housekeeping**:
  - **save/restore** processor context
  - **reset** interrupt-detecting hardware in CPU
  - **enable** processing interrupts of lower priorities
- Last instruction: **RETURN** or return from ISR

# ISR vs. Procedure Calls

- To execute ISRs, there is NO **CALL** instruction.
  - The microprocessor does the call automatically.
- An array, *interrupt vector*, of addresses is used to point to appropriate ISRs.
  - ISRs must be loaded when the computer is turned on.
- Interrupts can be **masked**.
- The **context** need to be saved.

# 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

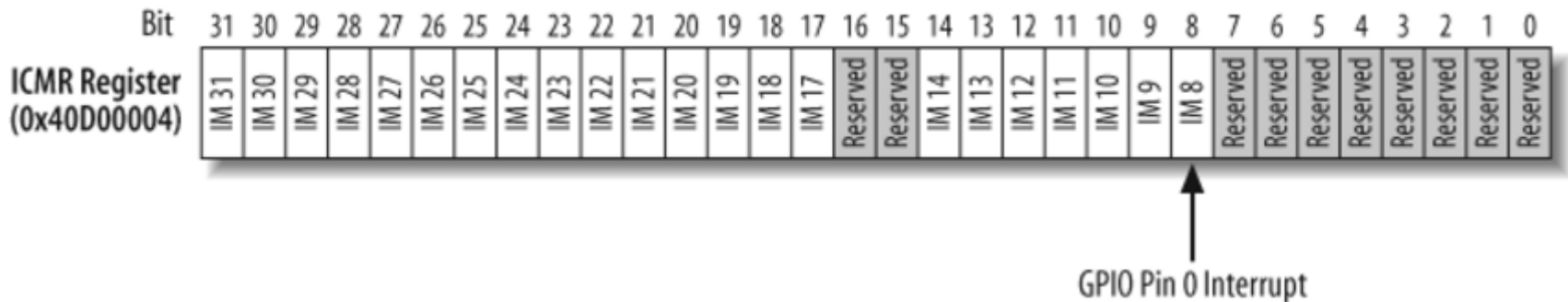
```
PUSH R1
PUSH R2
. . .
!! Read char from hw into R1
!! Store R1 value into memory
. . .
!! Reset serial port hw
!! Reset interrupt hardware
. . .
POP R2
POP R1
RETURN
```

# Disabling Interrupts

- Two ways:
  - At **Source**: tell I/O chip to stop interrupting
  - At **Destination**: inform CPU to ignore IRQs
- **Selective disabling** can be performed
  - Write a value in a special register
- Two types of interrupts
  - **Maskable**: can be disabled, normal I/O
  - **Nonmaskable**: cannot be disabled, power failure, catastrophic event, etc.  
(ISR must **not share data** with task code!)

# PXA255 Interrupt Controller Mask Register

- **ICMR**: at 0x40D00004, for **individual** interrupt enabling and disabling
  - 1: interrupt allowed
  - 0: interrupt masked



# Current Program Status Register

- **CPSR**: for **global** interrupt enabling and disabling

- FIQ bit and IRQ bit

- Cannot use C to change CPSR

- Must use assembly instructions

- `MRS{<cond>} Rd,<psr>` ; `Rd = <psr>`
- `MSR{<cond>} <psr>,Rm` ; `<psr> = Rm`
- `MSR{<cond>} <psrf>,Rm` ; `<psrf> = Rm`

**where**

- `<psr>` = CPSR, CPSR\_all, SPSR or SPSR\_all
- `<psrf>` = CPSR\_flg or SPSR\_flg

# Disabling Interrupts

---

- **Priority-based** interrupt disabling/enabling
- Assign a **priority** to each IRQ
- Program specifies **lowest acceptable priority**
  - All IRQ with **lower** priorities will be **disabled**
  - All IRQ with **higher** priorities will be **enabled**

# ARM v5TE Processor

## Exception/Interrupt Priorities

| Priority    | Exception/Interrupt Source                  |
|-------------|---------------------------------------------|
| 1 (highest) | Reset                                       |
| 2           | Data abort                                  |
| 3           | Fast Interrupt Request (FIQ)                |
| 4           | Interrupt Request (IRQ)                     |
| 5           | Prefetch Abort                              |
| 6 (lowest)  | Undefined instruction or software interrupt |

ARM v6: an imprecise abort between IRQ and prefetch abort

# Some Common Questions

---

- How do CPUs know where to find **ISR**?
  - At fixed location: e.g.: 8051 IRQ1: 0x0003
  - In IVT (Interrupt Vector Table): addresses of each ISR
- How does CPU know where **IVT** is?
  - At fixed location: e.g.: 80186: 0x00000
  - Depends on CPU

# ARM Interrupt Vector Table

- Addresses in IVT are at fixed locations

| Exception/Interrupt source | Address    |
|----------------------------|------------|
| Reset                      | 0x00000000 |
| Undefined instruction      | 0x00000004 |
| Software interrupt         | 0x00000008 |
| Prefetch abort             | 0x0000000C |
| Data abort                 | 0x00000010 |
| IRQ                        | 0x00000018 |
| FIQ                        | 0x0000001C |

# Partial Interrupt Map for the Arcom Board

---

## Interrupt number Interrupt source

|    |               |
|----|---------------|
| 8  | Ethernet      |
| 11 | USB           |
| 21 | Serial Port 2 |
| 22 | Serial Port 1 |
| 26 | Timer 0       |
| 27 | Timer 1       |
| 28 | Timer 2       |

# Interrupt Map

```
/* **** */
* Interrupt Map
* **** */

#define ETHERNET_INT (8)
#define USB_INT (11)
#define SERIAL2_INT (21)
#define SERIAL1_INT (22)
#define TIMER0_INT (26)
#define TIMER1_INT (27)
#define TIMER2_INT (28)
```

Should install  
a **default** ISR  
for any **unused**  
IRQ in IVT!!!

# Some Common Questions (contd)

---

- Can CPU be interrupted **within an instruction**?
  - Usually not
  - Except: inst with large data movements
- 2 IRQ at same time, which one to service?
  - According to **priorities**
- Can an IRQ interrupt an ISR?
  - **Yes**: default behavior on some CPUs

# Some Common Questions (contd)

- Interrupt nesting:
  - **80x86**: all IRQs disabled automatically, ISR must reenables interrupts
  - **Others**: automatic, high-priority IRQ can interrupt low-priority ISR
- Interrupts signaled, while IRQs are disabled?
  - Interrupts are **remembered** by microprocessor
  - Serviced after IRQs are **enabled**, in **priority-order** (actually only deferred)

# Some Common Questions (contd)

---

- IRQs disabled, forgot to enable?
  - System **halted**
- CPU start up: interrupts enabled or disabled?
  - **Disabled**
- ISR in C? **Yes!**

# ISR in C?

- 3 methods

- Compiler-specific keyword: **interrupt**

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

- Compiler-specific #pragma to declare ISR

- GNU gcc keyword: **\_\_attribute\_\_**

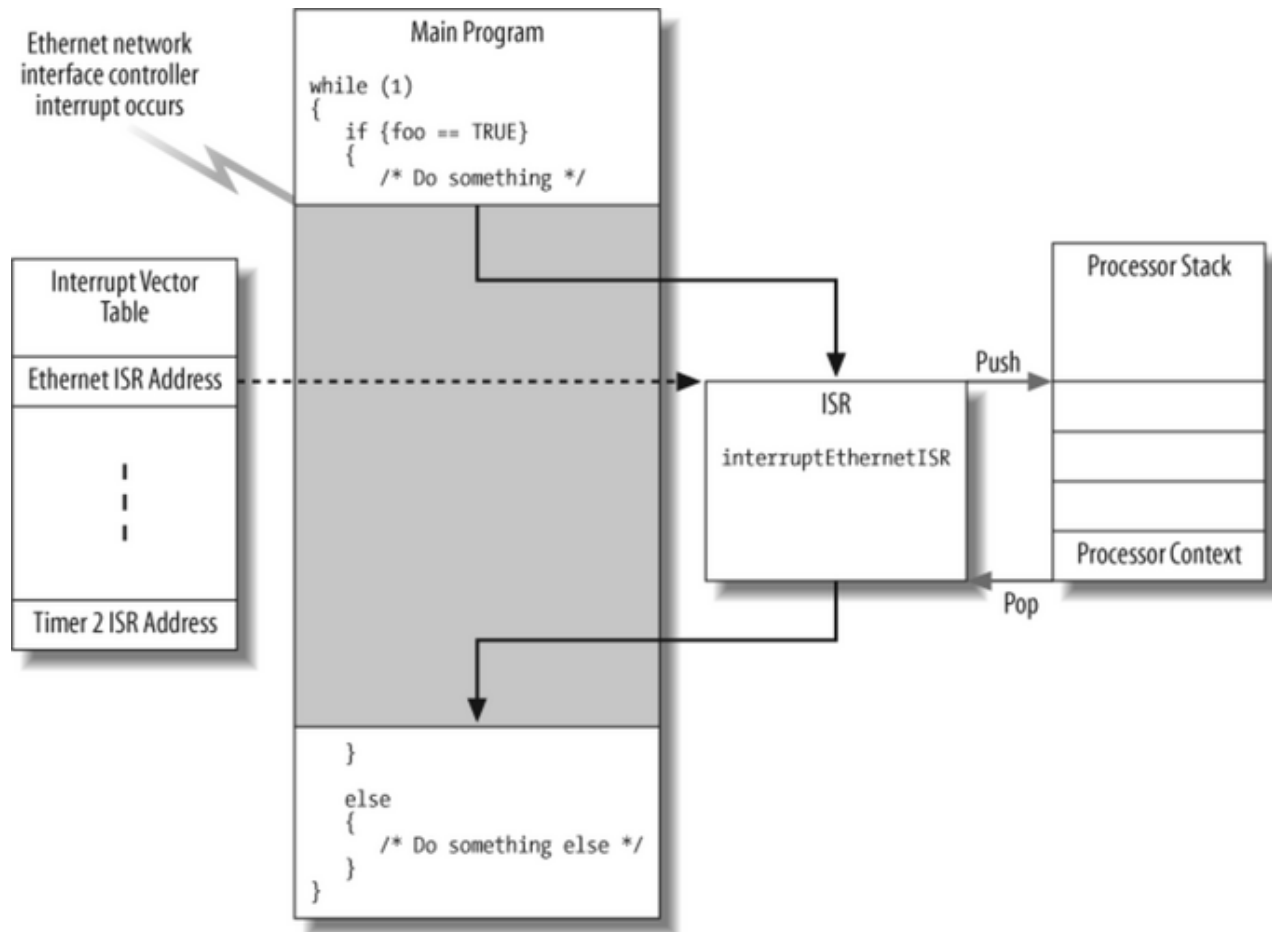
```
void vHandleTimerIRQ() __attribute__((interrupt(*IRQ)));
```

- Compiler will **automatically** add code to **save and restore the context**

- Compiler will add **RETURN** at end

- C is a little slower than assembly, if speed is not of concern, C can be used to write ISR!

# Software Flow During Interrupt

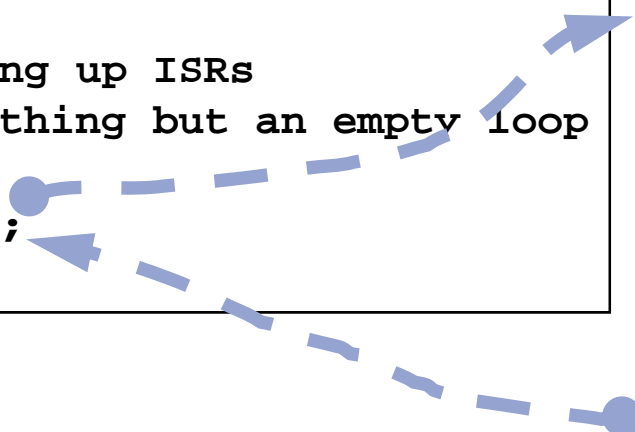


# The Shared-Data Problem

# A Powerful ISR Design

- Can the ISR do **everything** you want?
  - Yes, but this is very **impractical**.

```
main()
{
 int i;
 // setting up ISRs
 // do nothing but an empty loop
 while(1)
 i=i+0;
}
```

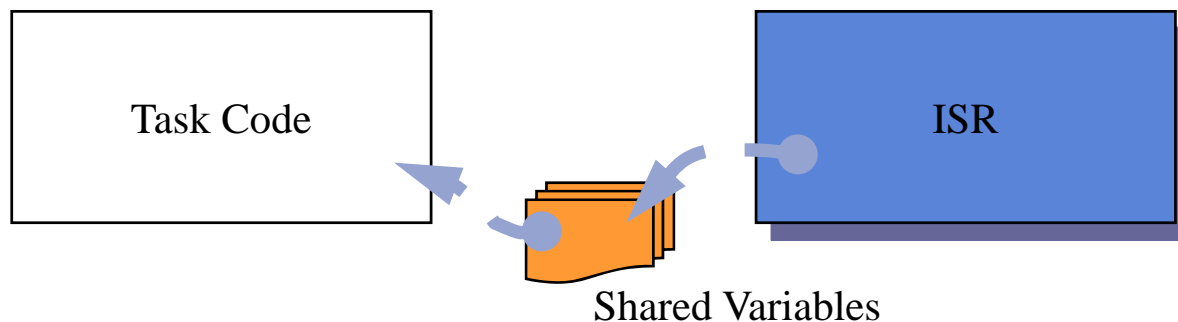


A dashed blue arrow originates from the `i=i+0;` line in the `while(1)` loop of the `main()` function and points to the `SampleISR()` function, indicating a call or interrupt.

```
SampleISR()
{
 newVal:= ReadADConverter()
 call StartNewConversion()
 ...
 ...
 ...
}
```

# A Practical Approach

- The ISR should do the **necessary** tasks
  - **moving data** from I/O devices to the memory buffer or vice versa
  - **handling** emergent signals
  - **signaling** the task subroutine code or the kernel
- The ISR needs to **notify** some other procedure to do follow-up processing.



# Accessing the Shared Data

- However, this may cause the shared data problem.

```
static int iTemperatures[2];

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

**ISR**

```
void main (void)
{
 int iTemp0, iTemp1;

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

**Task Code**

**Interrupt**

# The Shared Data Problem

---

- The **unexpected** interrupt
  - may cause two readings to be **different**,
  - even though the two measured temperatures were always the **same**.
- How can we **improve** on this?
  - **Compare** these temperature measurements **directly**!

# Directly Testing

## ■ The new code

```
static int iTemperatures[2];

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

**ISR**

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

**Task Code**

.....● **Testing**

# A Bug Harder to Be Detected

## ■ The compiler translation

```
if (iTemperatures[0] != iTemperatures[1])
 !! Set off howling alarm;
```

```
·
·
·
 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:
 ·
 ·
 ·
```

● **Interruptible**

# Characteristics of Shared-Data Bug

- A bug that is **very hard to find**.
  - It does **not** happen every time the code runs.
  - The program **may run correctly** most of the time.
  - However, a **false alarm** may be set off sometimes.
- How can you trace back if the embedded system has already **exploded**?
- Remember the **Therac-25 accidents**?
  - There were many **shared** variables!!!

# Characteristics of Shared-Data Bug

---

- Whenever ISR and task code share data
  - Be **suspicious!!!**
  - **Analyze** the situation to locate any such bugs!!!

# Solving the Shared-Data Problem

# Solving Shared-Data Problem

---

- **Solution:** Disable interrupts before using shared data in task code, enable them after!
- C compilers have **functions** to disable and enable interrupts
- Processors have **instructions** to disable and enable interrupts
- **Problem:** Compilers not smart enough to automatically add disable/enable instructions around shared code. Users must DO IT!

# Solving Shared-Data Problem: Disabling Interrupts in C

```
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)
 {
 disable (); /* Disable interrupts while we use the array */
 iTemp0 = iTemperatures[0];
 iTemp1 = iTemperatures[1];
 enable ();

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

# Solving Shared-Data Problem: Disabling Interrupts in Assembly

```

:
DI ; disable interrupts while we use the array
MOVE R1, (iTemperature[0])
MOVE R2, (iTemperature[1])
EI ; enable interrupts again

SUBTRACT R1, R2
JCOND ZERO, TEMPERATURES_OK
:
:
; Code goes here to set off the alarm
:
:
TEMPERATURES_OK:
:
:
```

# Atomic Instructions/Sections

- A More Convenient Way
  - Use some atomic instructions supported by the hardware.
- An instruction is *atomic* if it cannot be interrupted.
- The collection of lines can be atomic by adding an **interrupt-disable** instruction.
- A set of instruction that must be atomic for the system to work properly is often called a critical section.

# Another Example

- A buggy program
  - imprecise answer

**Interrupt**

```
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);
}
```

# Interrupts with a Timer

---

- Bug:
- ISecondsSinceMidnight() **interrupted!**
- iSeconds, iMinutes, iHours **changed** by vUpdateTime()
- **Wrong answer** by ISecondsSinceMidnight()

# Several Possible Solutions

## ■ Disable the interrupt

```
long 1SecondsSinceMidnight (void)
{
 disable ();
 return ((((iHours * 60) + iMinutes) * 60) + iSeconds);
 enable (); /* WRONG: This never gets executed! */
}
```

## A buggy solution

- Interrupts will not be appropriately enabled.

# A Better Code

## ■ Changing the timing of return()

```
long lSecondsSinceMidnight (void)
{
 long lRetVal;

 disable ();

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

 enable ();

 return (lRetVal);
}
```

# The Best Way

## ■ The nested interrupts

```
long lSecondsSinceMidnight (void)
{
 long lRetVal;
 BOOL fInterruptStateOld; /* Interrupts already disabled? */

 fInterruptStateOld = disable ();

 lRetVal =
 (((iHours * 60) + iMinutes) * 60) + iSeconds;
 /* Restore interrupts to previous state */
 if (fInterruptStateOld)
 enable ();

 return (lRetVal);
}
```

# Another Potential Solution

## ■ Doing things in the ISR

```
static long int lSecondsToday;

void interrupt vUpdateTime (void)
{
 :
 :
 ++lSecondsToday;
 if (lSecondsToday == 60 * 60 * 24)
 lSecondsToday = 0L;
 :
 :
}

long lSecondsSinceMidnight (void)
{
 return (lSecondsToday);
}
```

# Is the solution correct?

- Depends on if the microprocessor's registers are **large enough to hold a long integer**

- ISecondsSinceMidnight() in assembly

YES!

```
■ MOVE R1, (ISecondsToday)
■ RETURN
```

← ATOMIC

NO!

```
■ MOVE R1, (ISecondsToday)
■ MOVE R2, (ISecondsToday+1)
■ ...
■ RETURN
```

← NOT ATOMIC

# Solution without disabling interrupts

■ (needs  
volatile  
keyword)

```
static long int lSecondsToday;

void interrupt vUpdateTime (void)
{
 :
 :
 ++lSecondsToday;
 if (lSecondsToday == 60L * 60L * 24L)
 lSecondsToday = 0L;
 :
 :
}

long lSecondsSinceMidnight (void)
{
 long lReturn;

 /* When we read the same value twice, it must be good. */
 lReturn = lSecondsToday;
 while (lReturn != lSecondsToday)
 lReturn = lSecondsToday;

 return (lReturn);
}
```

# Compiler optimizations defeat the purpose of the solution

## ■ Solution:

read twice ISecondsToday, if not changed then no interrupt occurred between the reads

## ■ Problems:

- After “IReturn = ISecondsToday;”, compiler **saves ISecondsToday** and assumes that it has not changed since the last read, thus **using the saved value** instead of newly reading it
- while loop is optimized out of existence!

# The volatile Keyword

- Need to declare:

```
static volatile long int lSecondsToday;
```

- Compiler will know that the variable is **volatile** and **each reference is read from memory**
- If this keyword is not supported, you can still get the similar result by turning off the compiler optimizations.

# Interrupt Latency

# Interrupt Latency

How **fast** does my system respond to each interrupt?

- ■ Longest period of interrupt **disabled** time
- ■ Total execution time of **higher priority** interrupts
- Time for microprocessor to **stop execution, switch context, start ISR execution**
- ■ **Execute ISR** till an initial response

CPU  
docs

Simulation, Estimation

# To Reduce Interrupt Latencies

---

- Factor 4: Initial response of ISR →  
Write efficient code
- Factor 3: Context switch →  
Hardware dependent
- Factor 2: Exec of higher-priority ISRs →  
Write short ISRs!
- Factor 1: Disable interrupts →  
To solve shared-data problem,  
Shorten period

# Make your ISRs short

---

- Factory Control System
- A task to monitor a gas leak detector
  - High priority
    - **Call** fire dept
    - **Shut down** affected factory part
- Call fire dept
  - Needs several **seconds**
  - **Delays** all other lower priority interrupts
  - Therefore, **remove** telephone call from ISR

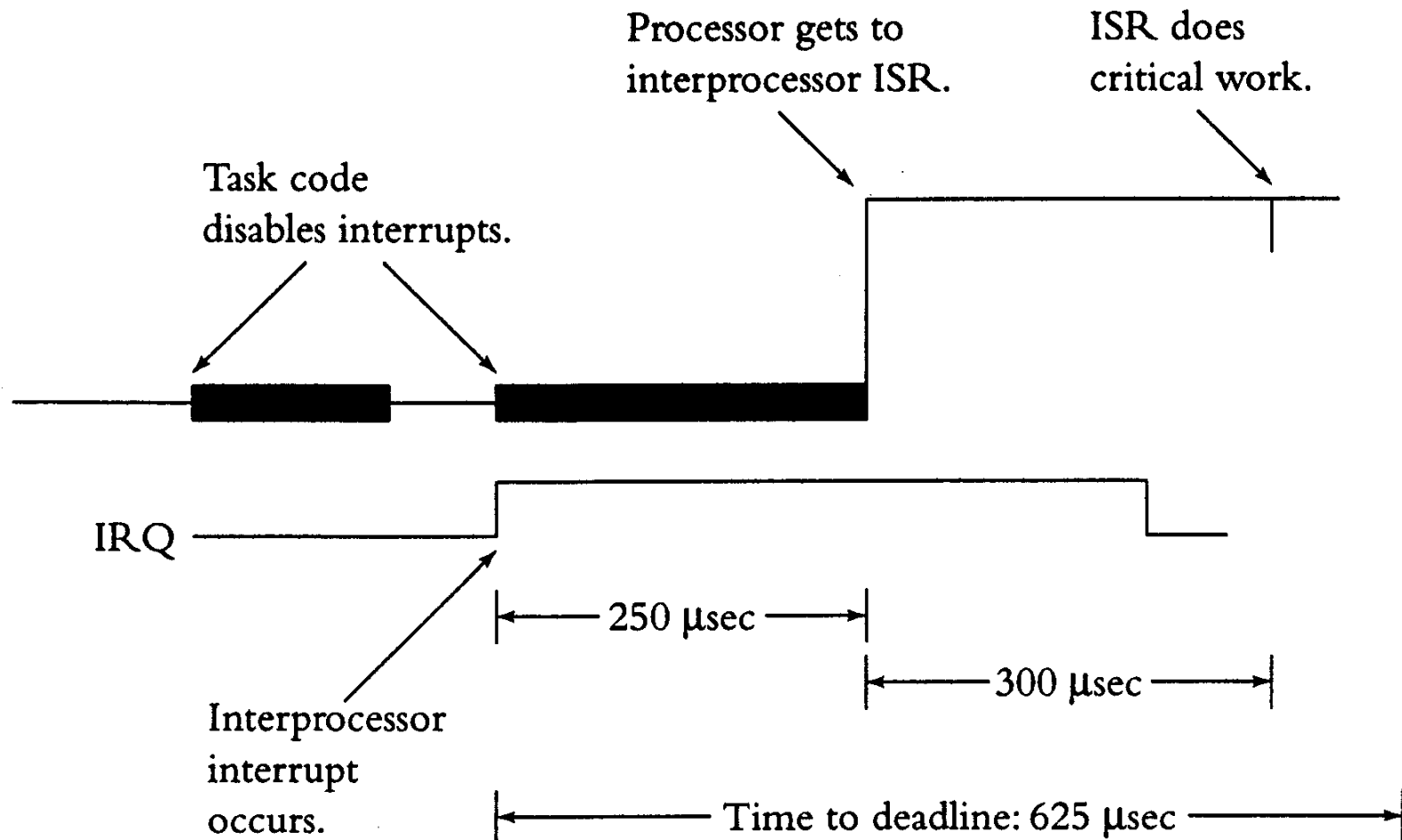
# Disabling Interrupts

---

System requirements:

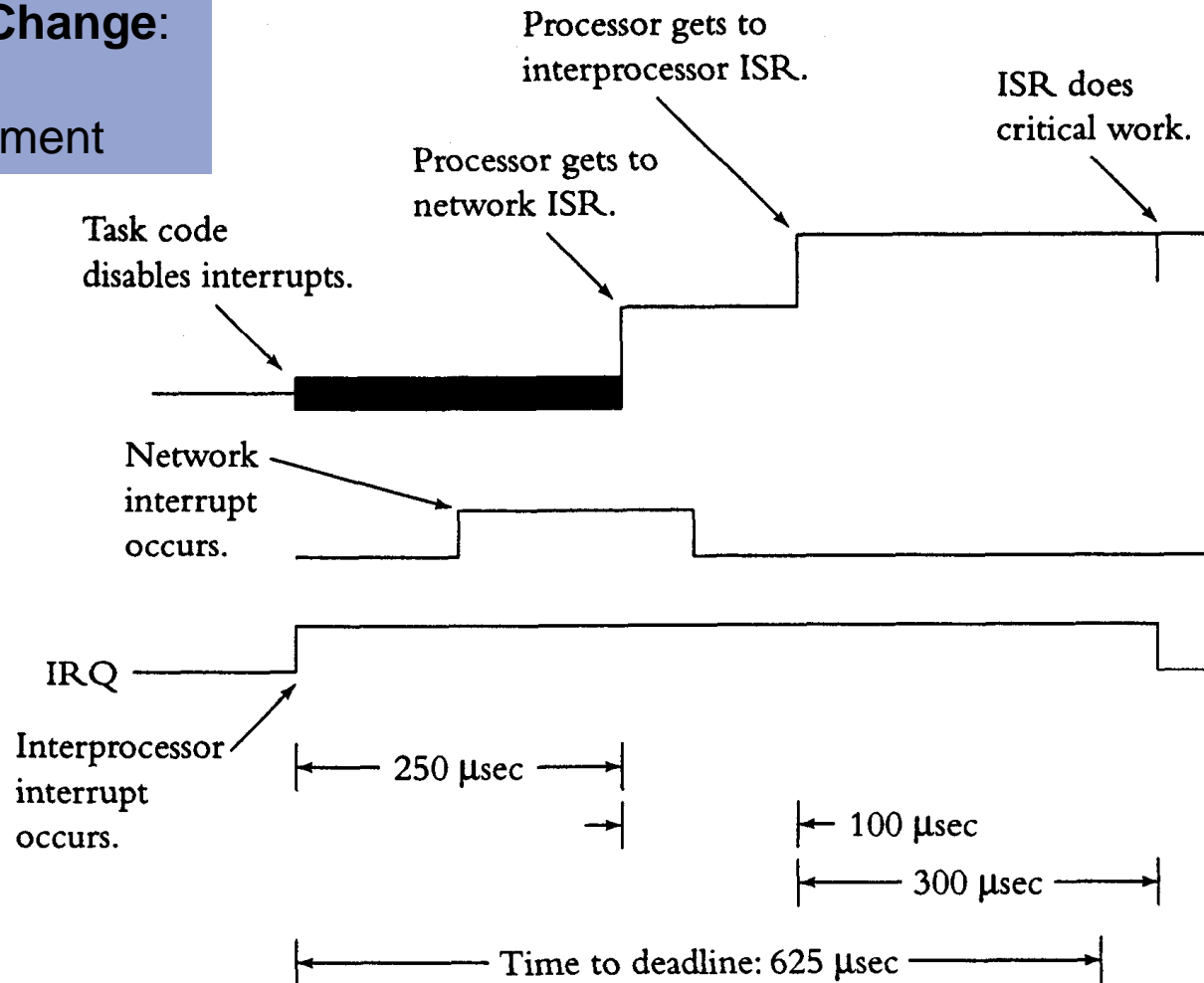
- **Disable** interrupts for **125 microseconds** to read 2 shared temperature variables
- **Disable** interrupts for **250 microseconds** to get time from shared variables
- **Respond** within **625 microseconds** to special signals, **ISR** takes **300 microseconds** to execute

# Worst Case Interrupt Latency



# Worst Case Interrupt Latency

## Design Change: Network Enhancement



# Alternatives to Disabling Interrupts

## ■ Two variable sets

```
static int iTemperaturesA[2];
static int iTemperaturesB[2];
static BOOL fTaskCodeUsingTempsB = FALSE;

void interrupt vReadTemperatures (void)
{
 if (fTaskCodeUsingTempsB)
 {
 iTemperaturesA[0] = !! read in value from hardware;
 iTemperaturesA[1] = !! read in value from hardware;
 }
 else
 {
 iTemperaturesB[0] = !! read in value from hardware;
 iTemperaturesB[1] = !! read in value from hardware;
 }
}
```

```
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;
 }
}
```

# Two Variable Sets

---

- This simple mechanism solves the shared-data problem, because the ISR will **never write** into the set of temperatures that the task code is **reading**.
- However, the while-loop may be executed **twice** before it sets off the alarm.

# A Queue Approach

- The shared-data problem is also eliminated.

```
#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
}
```

# A Queue Approach

## ■ main()

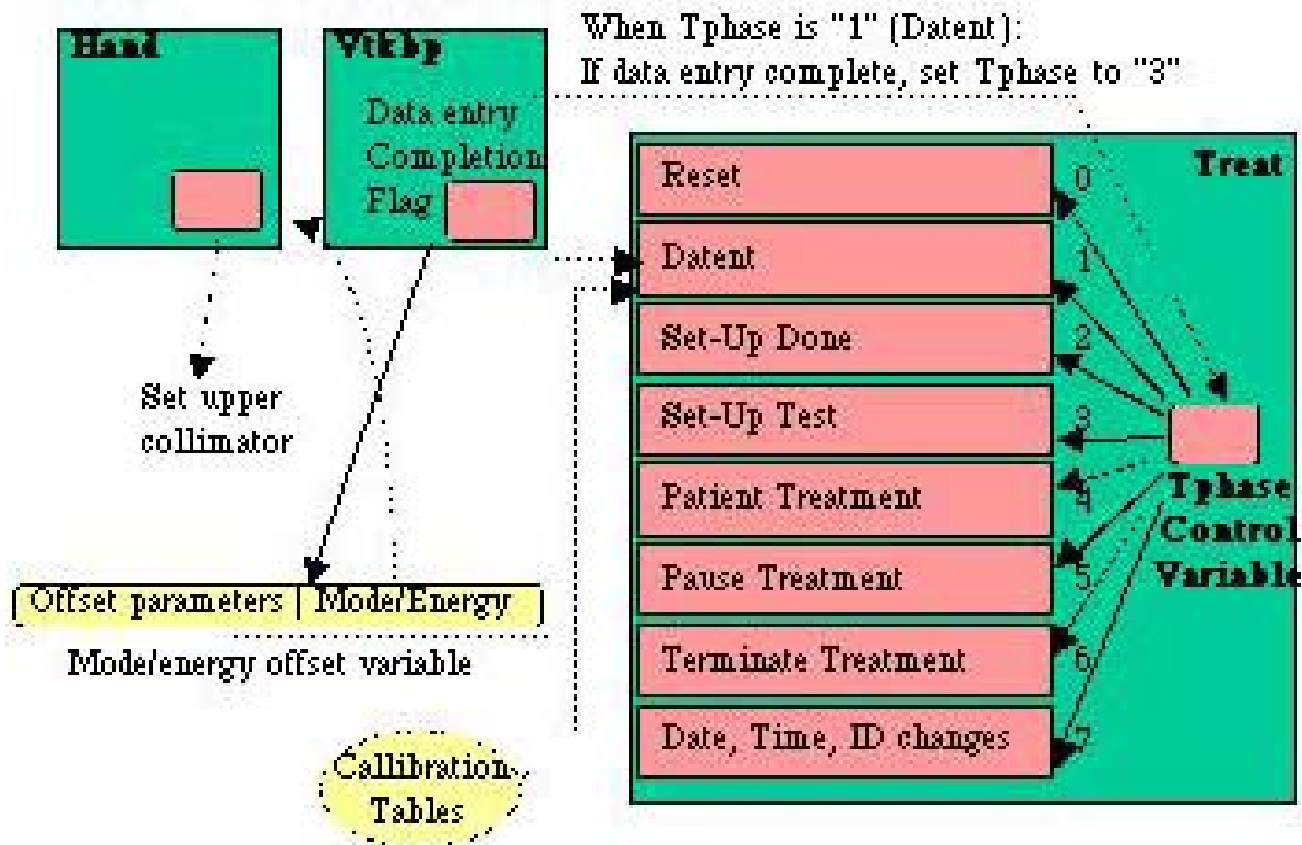
```
void main (void)
{
 int iTemperature1, iTemperature2;

 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;
 !! Do something with iValue;
 }
 }
}
```

# A Queue Approach

- However, this code is very **fragile**.
- Either of these seemingly minor changes can cause **bugs**
  - In task code (main):
    - **Reversing** the following order could cause shared data bug
      - **Read data first**
      - **Update iTail pointer second**
    - If the following statement is **non-atomic**, shared data bug could occur
      - **iTail += 2;**

# Shared Data in Therac-25 Software



**Tasks and subroutines in the code blamed for the Tyler accidents**

# Improved Blinking LED Program

---

- Use a timer to re-write the blinking LED program
  - More **accurate** than trial-and-error approach
  - Delay routine eliminated
  - Timer **device driver** is used for delay
    - Interrupts processor after 500 ms

# How Timers Work?

## ■ Timer

- A **peripheral** that measures elapsed time
- **Counts down** processor cycles or clocks

## ■ How does a timer work?

- Setup an **interval register** in peripheral
- Uses a **clock** to keep count of number of ticks elapsed since timer started
- Number of clock ticks **compared** to the value in timer interval register
  - Equal → a **timer interrupt** is generated (if enabled)

# Timer clocks

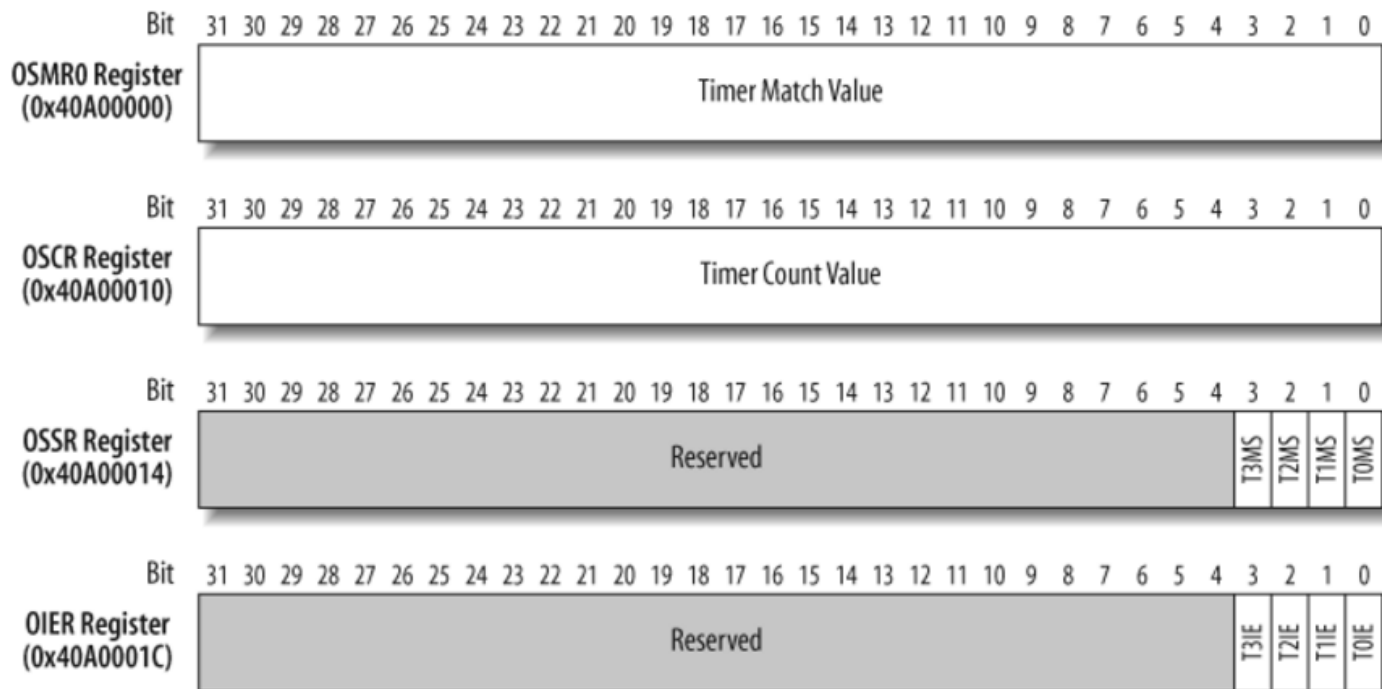
---

- Timer counts cycles from
  - Processor's **main clock** signal, or
  - A **separate clock** signal fed into timer peripheral
- Can be configured by programming timer's **configuration registers**
  - Modern processor has **multiple** internal clock sources to drive timer

# PXA255 Timer

- 4 timers: 0, 1, 2, 3
- Counts up

PXA255 Processor  
Timer 0 registers



# PXA255 Timer

- Timer **Count** Register (OSCR)
  - Contains a count incremented on rising edge of timer clock (3.6864 MHz)
- Timer **Match** Register (OSMR $n$ )
  - Timer values for 4 different timers
  - Compares value of OSMR $n$  to the OSCR
    - Equal → interrupt generated
- Timer **Interrupt Enable** Register (OIER)
  - To enable interrupts for the 4 different timers
- Timer **Interrupt Status** Register (OSSR)
  - To check if interrupts are enabled

# Watchdog Timers

- Kicking the dog!
- A special hardware **fail-safe** mechanism
  - Intervenes when software stops functioning
  - Periodically reset by software
  - If software crashes or hangs, the watchdog timer soon expires, causing system to be **reset automatically**
- **Must implement resetting in main processing loop and not in ISR**
  - Main loop might hang, while all IRQ/ISRs work!

# Blinking LED: main()

```
#include "led.h"
#include "timer.h"
int main(void) {
 /* Configure the green LED control pin. */
 ledInit();

 /* Configure and start the timer */
 timerInit();

 while (1) ;

 return 0;
}
```

# Timer Initialization

- Use **bInitialized** variable to ensure timer registers are configured only once!
- Clear any pending interrupt
  - Write bit 0 (TIMER\_0\_MATCH) to OSSR (TIMER\_STATUS\_REG)
- Calculate interrupt interval
  - PXA255 clock is **3.6864** MHz

Timer Match Register Value = Timer clock x Timer interval  
= 3,686,400 Hz x 0.5 seconds  
= 1,843,200 = **0x001C2000 = TIMER\_INTERVAL\_500MS**

# Timer Initialization

- Algorithm to setup timer in PXA255
  - **Read** current count value in timer count register **OSCR**
  - **Add** interval offset to current count value. This is the amount of time before next time-out.
  - **Program** new interval value into timer match register 0, **OSMR0**.

```
TIMER_0_MATCH_REG =
(TIMER_COUNT_REG +
TIMER_INTERVAL_500MS);
```

# Timer Initialization

- Timer Enabling is done in **two** places
  - **Timer peripheral**
    - Bit 0 (TIMER\_0\_INTEN) in the 32-bit OIER (TIMER\_INT\_ENABLE\_REG)
  - **Interrupt controller**
    - Timer 0 is mapped to interrupt number 26
    - Set bit number 26 (TIMER\_0\_ENABLE) in ICMR (INTERRUPT\_ENABLE\_REG)
- Finally, **bInitialized** is set to TRUE.

# Timer Initialization

```
#define TIMER_INTERVAL_500MS (0x001C2000)
void timerInit(void) {
 static int bInitialized = FALSE;
 /* Initialize the timer only once */
 if(bInitialized == FALSE) {
 /* Ack outstanding timer interrupts */
 TIMER_STATUS_REG = TIMER_0_MATCH;
 /* Initialized timer interval */
 TIMER_0_MATCH_REG = (TIMER_COUNT_REG + TIMER_INTERVAL_500MS);
 /* Enable timer interrupt in timer peripheral */
 TIMER_INT_ENABLE_REG |= TIMER_0_INTEN;
 /* Enable timer interrupt in interrupt controller */
 INTERRUPT_ENABLE_REG = TIMER_0_ENABLE;
 bInitialized = TRUE;
 }
}
```

# Timer ISR

```
#include "led.h"

void timerInterrupt(void) {
 /* Ack timer 0 interrupt */
 TIMER_STATUS = TIMER_0_MATCH;
 /* Change state of green LED */
 ledToggle();
 /* Set new timer interval */
 TIMER_0_MATCH_REG = (TIMER_COUNT_REG +
 TIMER_INTERVAL_500MS;
}
```