Embedded Software

TI2726-B

4. Interrupts

Koen Langendoen
Embedded Software Group
What is an Interrupt?

- Asynchronous signal from hardware
- Synchronous signal from software
- Indicates the need for attention
- Indicates the need for execution change
Why Interrupts?

- Avoid wasting time in polling loops for external events
- Make a system reactive and simple
- Make debugging more fun 😊
Interrupts: Principle

Program:

```
MOVE R1, (var-addr)
MULT R1, 9
DIVIDE R1, 5
ADD R1, 32
...
```

ISR:

```
PUSH R1
...
POP R1
RET
```
Where is the interrupt code?

- Intel 8051 - default locations for interrupt service routines
- Interrupt vector table
- Where is the interrupt vector table?
  - Default location (Intel 80186)
  - Programmable location
Some questions...

- Can a microprocessor be interrupted in the middle of an instruction?
- Which interrupt goes first? Interrupt during another interrupt?
- What happens immediately after the interrupts are enabled?
- What is the status of interrupts when the processor starts?
ATmega2560

- too complex
- 100 pins
- 54 interrupt sources
Roll your own: FPGA + Soft Core

- Xilinx XC3S200
  - 50 MHz, 200k gates
  - I/O, LEDs, SSD, buttons, RS232

- X32 soft core
  - 32-bit processor
  - peripherals: UART, PWM, timer, decoder
X32 Peripheral Interrupts

buttons, LEDs, SSD, switches, timers, RS232, off-board I/O ports

Peripheral 0 - K

IRQ₀  ....... IRQₙ

Interrupt Controller

multiple IRQ lines per peripheral possible (e.g., tx+rx IRQ per UART)
X32: Interrupt Sources

- CPU: divide-by-0, overflow (disable unless needed!)
- Buttons, switches, I/O ports
  - positive AND negative edge-triggered
- Timers: counter value > threshold reg
- UART: rx buffer char received / tx buffer empty
- ...

X32: Interrupt Controller

Interrupt Controller

IRQ0

IRQn

Interrupt Controller

IEk, priok, vectork

IE_global

interrupt

acknowledge

vector

priority

exec level

CPU

FSM for each IRQ:

IRQ . IE

IE_global . (priority > exec level)

stdby

scheduled

IE'

servicing / interrupt

acknowledge
X32 Interrupts

- IRQ controller preprocesses multiple IRQ’s
- Each device: (IRQ #, priority, associated with IRQ vector)
- Vectored IRQ
- Interrupts NOT disabled automatically
- Automatic ISR preemption if prio IRQ > prio current ISR
- Normal call saves context -> no interrupt keyword
X32: Demo

- Demo ..
- (x32_projects.tgz, leds.c, **buttons.c**)
void isr_read_temps(void)
{
    iTemp[0] = peripherals[..];
    iTemp[1] = peripherals[..];
}

void main(void)
{
    ...
    while (TRUE) {
        NOT ATOMIC!
        tmp0 = iTemp[0];
        tmp1 = iTemp[1];
        if (tmp0 != tmp1)
            panic();
    }
}
Finding this bug...

- Can be very tricky
  - The bug does not occur always!
- Frequency depends on
  - The frequency of interrupts
  - Length of the critical section
- Problem can be difficult to reproduce

- Advise: double check the access on data used by ISR!
Solving the Data-Sharing Problem?

```c
void isr_read_temps(void)
{
    iTemp[0] = peripherals[..];
    iTemp[1] = peripherals[..];
}

void main(void)
{
    ...
    while (TRUE) {
        if (iTemp[0] != iTemp[1])
            panic();
    }
}
```

MOVE R1, (iTemp[0])
MOVE R2, (iTemp[1])
SUBSTRACT R1,R2
JCOND ZERO, TEMP_OK
...
...
TEMP_OK:
...
Solution #1

- Disable interrupts for the ISRs that share the data

```c
... while (TRUE) {
    !! DISABLE INT
    tmp0 = iTemp[0];
    tmp1 = iTemp[1];
    !! ENABLE INT
    if (tmp0 != tmp1)
        panic();
}
```

The critical section is now atomic.
X32: Demo

- Demo ..
- (x32_projects.tgz, critical.c)
Atomic & critical section

- A part of a program is atomic if it cannot be interrupted
  - Interrupts and program code share data
- *atomic* can also refer to mutual exclusion
  - Two pieces of code sharing data
  - They can be interrupted
- The instructions that must be atomic = *critical section*
Be careful!

```c
static int iSeconds, iMinutes;

void interrupt vUpdateTime(void)
{
    ++iSeconds;
    if (iSeconds>=60) {
        iSeconds=0;
        ++iMinutes;
    }
}

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

too little, too late 😞
Function calls and enable()

- enable() can be a source of bugs!

```c
void function1 () {
    ...
    // enter critical section
    disable();
    ...
    temp = f2();
    ...
    // exit critical section
    enable();
    ...
}

int f2 () {
    ...
    disable();
    ...
    enable();
    ...
}
```

should test if this is fine
More on shared-data...

```c
static long int lSecondsToday;

void interrupt vUpdateTime()
{
    ...
    ++lSecondsToday;
    ...
}

long lGetSeconds()
{
    return (lSecondsToday);
}
```
static long int lSecondsToday;

void interrupt vUpdateTime()
{
    ++lSecondsToday;
}

long lGetSeconds()
{
    long lReturn;

    lReturn = lSecondsToday;
    while (lReturn!=lSecondsToday)
    {
        lReturn = lSecondsToday;
    }

    return (lReturn);
}

ingenious code without interrupts
Any issues here?

```c
volatile static long int lSecondsToday;

void interrupt vUpdateTime()
{
    ++lSecondsToday;
}

long lGetSeconds()
{
    long lReturn;

    lReturn = lSecondsToday;
    while (lReturn!=lSecondsToday)
        lReturn = lSecondsToday;

    return (lReturn);
}
```

Otherwise compiler might optimize this code!
Interrupt Latency

- Quick response to IRQ may be needed
- Depends on previous rules:
  - The longest period of time in which interrupts are disabled
  - The time taken for the higher priority interrupts
  - Overhead operations on the processor (finish, stop, etc.)
  - Context save/restore in interrupt routine
  - The work load of the interrupt itself
- worst-case latency = t_maxdisabled + t_higher prio ISRs + t_myISR + context switches
Example #1

Interrupt execution time – worst case analysis
Example #2

Interrupt execution time – worst case analysis – in the presence of a second interrupt
Alternatives to disable()

- Do not disable interrupts but write ingenious code
  - By alternating data buffers (Fig. 4.15/page 108)
  - By using queues (Fig. 4.16/page 109)
  - ISR and (main) code never access the same data
- Problem: code becomes error-prone and hard to read
- Rule of the thumb:
  - *Keep it simple*, just disable interrupts as long as:
    - keep the critical sections SHORT
    - keep the ISRs SHORT (to minimize latency)
Avoiding interrupts #1

```c
static int temp[2];
static bool busy = FALSE;

void interrupt readTemp()
{
    if (!busy) {
        temp[0]=...;
        temp[1]=...;
    } else {
        // try again later
    }
}

void main(void)
{
    while (TRUE) {
        busy = TRUE;
        if (temp[0]!=temp[1]) ... ;
        busy = FALSE;
        ...
    }
}
```

Idea: use a Boolean flag to protect critical section
Avoiding interrupts #1

```c
static int tempA[2];
static int tempB[2];
static bool useB = FALSE;

void interrupt readTemp() {
    if (useB) {
    } else {
        tempB[0]=…; tempB[1]=…;
    }
}

void main(void) {
    while (TRUE) {
        if (useB)
            if (tempB[0]!=tempB[1]) … ;
        else
            if (tempA[0]!=tempA[1]) … ;
        useB = !useB;
    }
}
```

Alternating data buffers example (page 108)
Avoiding interrupts #2

- Make use of a circular queue (example page 109)
  - Interrupt “produces” temperature readings
  - Main code “consumes” temperature readings

- Operation
  - Interrupt adds readings to the queue (modifies the head pointer)
  - Main code extracts readings from the queue (modifies the tail pointer)
Questions?