CS4101 嵌入式系統概論

Interrupts

Prof. Chung-Ta King

Department of Computer Science

National Tsing Hua University, Taiwan

Materials from *MSP430 Microcontroller Basics*, John H. Davies, Newnes, 2008

Inside MSP430 (MSP430G2551)

oscillators | selectors | dividers | clocks

VLO
LFXT1
XT2 (if present)
DCO
RSELx, DCOx, MODx
LFXT1Sx
SELS
DCOCLK
SELMx
/1/2/4/8 DIVAx
/1/2/4/8 DIVSx
/1/2/4/8 DIVMx
ACLK auxiliary clock
SMCLK sub-system master clock
MCLK master clock
peripherals
CPU and a few peripherals

incl. 16 Registers
MDB
Emulation 2BP
JTAG Interface
Spy-Bi Wire
Brownout Protection
Wat WDT+ 15-Bit
2 CC Registers
Universal Serial Interface SPI, I2C
RST/NMI

Timer Block
TASSELx
IDx
Timer Clock
MCx
TACLK
ACLK
SMCLK
INCLK
00
01
10
11
Divider 1/2/4/8
15
16-bit Timer TAR
0
Clear
RC
Count Mode
EQU0
TACLR
Set TAIFG
CCR
CCR
CCR
CCISx
CMx
logic
COV
SCS
CCI2A
CCI2B
GND
VCC
Capture Mode
Timer Clock
Sync
CCI
15
TACCR2
0
Comparator 2
EQU2
CAP
SCCI
Y A EN
Set TACCR2 CCIFG
OUT
EQU0
Output Unit2
Timer Clock
D Set Q
Reset
OUT2 Signal
POR
OUTMODx

Introduction

- When MSP430 processor executes the following code, it will loop forever
- Question: How can it do other things, e.g. handling external events or falling into low-power modes?

```
StopWDT  mov.w #WDTPW+WDTHOLD,&WDTCTL
SetupP1  bis.b #001h,&P1DIR ; P1.0 output
Mainloop xor.b #001h,&P1OUT ; Toggle P1.0
Wait     mov.w #050000,R15  ; Delay to R15
L1       dec.w R15          ; Decrement R15
         jnz   L1           ; Delay over?
         jmp   Mainloop     ; Again
```

Option 1

- Put codes that handle external events in your main program → polling

```
StopWDT   mov.w #WDTPW+WDTHOLD,&WDTCTL
SetupP1   bis.b #001h,&P1DIR ; P1.0 output
Mainloop xor.b #001h,&P1OUT ; Toggle P1.0
Wait      mov.w #050000,R15  ; Delay to R15
L1        dec.w R15          ; Decrement R15
          jnz   L1           ; Delay over?
          bit.b #B1,&P2IN    ; Test bit B1
          jnz ButtonUp       ; Jump if not zero
ButtonUp:bis.b #LED1,&P2OUT ; Turn LED1 off
          jmp   Mainloop     ; Again
```

Sample Code 1 for Input from Lab 2

```
#include <msp430.h>
#define LED1 BIT0    //P1.0 to red LED
#define B1 BIT3      //P1.3 to button
void main(void){
  WDTCTL = WDTPW + WDTHOLD; //Stop watchdog timer
  P1OUT |= LED1 + B1;
  P1DIR = LED1; //Set pin with LED1 to output
  P1REN = B1;   //Set pin to use pull-up resistor
  for(;;){   //Loop forever
    if((P1IN & B1) == 0){  //Is button down
      P1OUT &= ~LED1; }    // Turn LED1 off
    else{                  //Is button up
      P1OUT |= LED1;  }    // Turn LED1 on
  }
}
```

Option 2

- Keep your program unchanged and force the processor to jump to the code handling the external event when that event occurs
- Requirements:
  - Must let the processor know when the event occurs
  - Must let the processor know where to jump to execute the handling code
  - Must not allow your program know!!

    → you program must execute as if nothing happens

    → must store and restore your program state

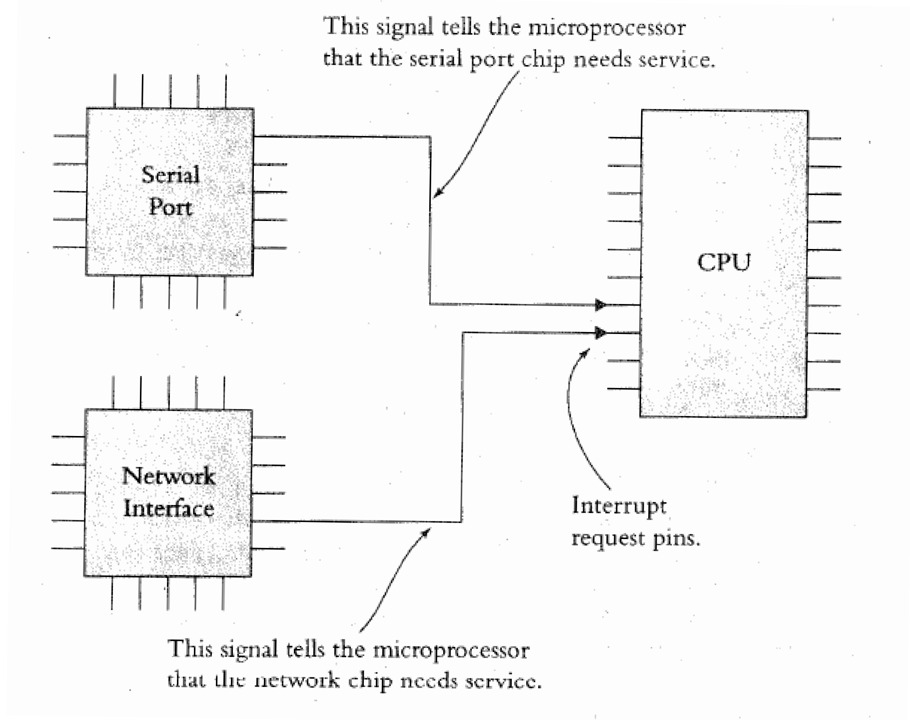
This is called interrupt!

Outline

- Introduction to interrupt
- The shared-data problem
- Interrupts of MSP430

Interrupt: Processor's Perspective

- How does the processor know when there is an interrupt?
  - Usually when it receives a signal from one of the IRQ (interrupt request) pins



Interrupt: Processor's Perspective

- What does the processor do in handling an interrupt?
  - When receiving an interrupt signal, the processor stops at the next instruction and saves the address of the next instruction on the stack and jumps to a specific *interrupt service routine* (ISR)
  - ISR is basically a subroutine to perform operations to handle the interrupt with a RETURN at the end
- How to be transparent to the running prog.?
  - The processor has to save the "state" of the program onto the stack and restoring them at the end of ISR

Interrupt Service Routine

- The following shows an example of an ISR

Task Code

```
...
MOVE R1, R7
MUL R1, 5
ADD R1, R2
DIV R1, 2
JCOND ZERO, END
SUBTRACT R1, R3
...
...
END: MOVE R7, R1
...
```

ISR

```
PUSH R1
PUSH R2
...
;ISR code comes here
...
POP R2
POP R1
RETURN
...
```

Interrupt: Program’s Perspective

- To a running program, an ISR is like a subroutine, but is invoked by the hardware at an unpredictable time
  - Not by the control of the program’s logic
- Subroutine:
  - Program has total control of when to call and jump to a subroutine

Disabling Interrupts

- Programs may disable interrupts
  - In most cases the program can select which interrupts to disable during critical operations and which to keep enabled by writing corresponding values into a special register
  - *Nonmaskable* interrupts cannot be disabled and are used to indicate critical events, e.g. power failures
- Certain processors assign *priorities* to interrupts, allowing programs to specify a threshold priority so that only interrupts having higher priorities than the threshold are enabled

Where to Put ISR Code?

- Challenges:
  - Locations of ISRs should be fixed so that the processor can easily find them
  - But, different ISRs may have different lengths
    → hard to track their starting addresses
  - Worse yet, application programs may supply their own ISRs; thus ISR codes may change dynamically
- Possible solutions:
  - ISR is at a fixed location, e.g., in 8051, the first interrupt pin always causes 8051 to jump to 0x0003
  - A table in memory contains addresses of ISR
    → the table is called *interrupt vector table*

How to Know Who Interrupts?

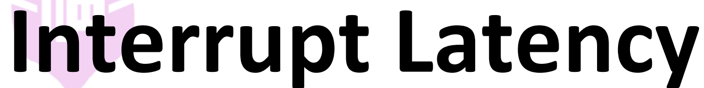
- Simple answer: according to interrupt signal
  - One interrupt signal corresponds to one ISR
- Difficult problem: same interrupt signal shared by several devices/events
  - Option 1: inside the corresponding ISR, poll and check these devices/events in turn
    → devices are passive
  - Option 2: devices/events provide the address of ISRs
    → devices are proactive
    → vectored interrupt

Some Common Questions

- Can a processor be interrupted in the middle of an instruction?
  - Usually not
  - Exceptions: critical hardware failure, long-running instructions (e.g. moving data in memory)
- If two interrupts occur at the same time, which ISR does the process do first?
  - Prioritize the interrupt signals
- Can an interrupt signal interrupt another ISR?
  - Interrupt nesting usually allowed according to priority
  - Some processor may require re-enabling by your ISR

Some Common Questions

- What happens when an interrupt is signaled while the interrupt is disabled?
  - Processors usually remember the interrupt signals and jump to the ISR when the interrupt is enabled
- What happens when we forget to re-enable disabled interrupts?
- What happens if we disable a disabled interrupt?
- Are interrupts enabled or disabled when the processor first starts up?



Interrupt Latency

- *Interrupt latency* is the amount of time taken to respond to an interrupt. It depends on:
  - Longest period during which the interrupt is disabled
  - Time to execute ISRs of higher priority interrupts
  - Time for processor to stop current execution, do the necessary 'bookkeeping' and start executing the ISR
  - Time taken for the ISR to save context and start executing instructions that count as a 'response'
- Make ISRs short
  - Factors 4 and 2 are controlled by writing efficient code that are not too long.
  - Factor 3 depends on HW, not under software control

Sources of Interrupt Overhead

- Handler execution time
- Interrupt mechanism overhead
- Register save/restore
- Pipeline-related penalties
- Cache-related penalties

Outline

- Introduction to interrupt
- The shared-data problem
- Interrupts of MSP430

The Shared-Data Problem

- In many cases the ISRs need to communicate with the task codes through shared variables.
- Example:
  - Task code monitors 2 temperatures and alarm if they differ
  - An ISR reads temperatures, e.g. on time up

```
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;
    }
}
```

The Shared-Data Problem

- Now, consider the assembly code:
  - When temperatures are 70 degrees and an interrupt occurs between the two MOVES to read temperatures
  - The temperatures now become 75 degrees
  - On returning from ISR, iTemp[1] will be assigned 75 and an alarm will be set off even though the temperatures were the same

```
MOVE     R1, (iTemperatures[0])
MOVE     R2, (iTemperatures[1])
SUBTRACT R1, R2
JCOND    ZERO, TEMPERATURES_OK
.
.
.
; Code goes here to set off the alarm
.
.
```

The Shared-Data Problem

- Problem is due to shared array iTemperatures.
- These bugs are very difficult to find as they occur only when the interrupt occurs in between the first 2 MOVE instructions, other than which code works perfectly.

Solving Shared-Data Problem

- Disable interrupts during instructions that use the shared variable and re-enabling them later

```
while (TRUE)
{
  disable();      // Disable interrupts
  iTemp0 = iTemperatures[0];
  iTemp1 = iTemperatures[1];
  enable();       // Re-enable interrupts
  ...
}
```

Solving Shared-Data Problem

- "Atomic" and "Critical Section"
  - A part of a program that cannot be interrupted
- Example:
  - An ISR that updates iHours, iMinutes and iSeconds every second through a hardware timer interrupt:

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

Outline

- Introduction to interrupt
- The shared-data problem
- Interrupts of MSP430

Know When an Interrupt Occurs

- An interrupt will be detected and serviced if
  - The global interrupt-enable (GIE) bit in Status Register (SR) in CPU is set
  - A peripheral device enables interrupt
    → For Timer_A: TAIE bit in TACTL register, CCIE bit in TACCTLx register
  - The peripheral signals an interrupt
    → For Timer_A: TAIFG, CCIFG

15	... bits ...	0
R0/PC	program counter	0
R1/SP	stack pointer	0
R2/SR/CG1	status register	
R3/CG2	constant generator	
R4	general purpose	
⋮		
R15	general purpose	

Ex: Timer_A Interrupt Enabling

TACTL

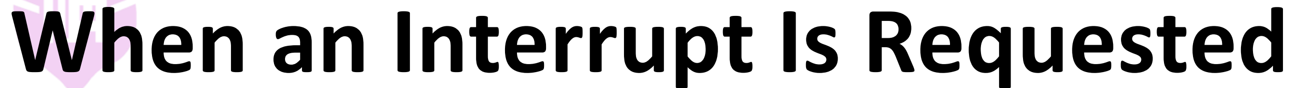
15	14	13	12	11	10	9	8
						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
IDx		MCx			TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

TACCTL

15	14	13	12	11	10	9	8
CMx		CCISx		SCS	SCCI	Unused	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	r0	rw-(0)

7	6	5	4	3	2	1	0
OUTMODx			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)



When an Interrupt Is Requested

- Any currently executing instruction is completed. MCLK is started if the CPU was off.
- The PC, which points to the next instruction, is pushed onto the stack.
- The SR is pushed onto the stack.
- The interrupt with the highest priority is selected.
- The interrupt request flag is cleared automatically for vectors that have a single source.
- The SR is cleared, and maskable interrupts are disabled.
- The interrupt vector is loaded into the PC and the CPU starts to execute the ISR at that address.

These operations take about 6 cycles

After an Interrupt Is Serviced

- An interrupt service routine must always finish with the *return from interrupt* instruction **`reti`**:
  - The SR pops from the stack. All previous settings of GIE and the mode control bits are now in effect.
    → enable maskable interrupts and restores the previous low-power mode if there was one.
  - The PC pops from the stack and execution resumes at the point where it was interrupted. Alternatively, the CPU stops and the device reverts to its low-power mode before the interrupt.

Where to Find ISRs?

- The MSP430 uses *vectored interrupts*.
  - Each ISR has its own vector, which is stored at a predefined address in a *vector table* at the end of the program memory (addresses 0xFFC0–0xFFFF).
  - The vector table is at a fixed location, but the ISRs themselves can be located anywhere in memory.

Interrupt Source	Interrupt Flag	System Interrupt	Word Address	Priority
Power-up/external reset/Watchdog Timer+/flash key viol./PC out-of-range	PORIFG RSTIFG WDTIFG KEYV	Reset	0FFFEh	31 (highest)
NMI/Oscillator Fault/ Flash access viol.	NMIIFG/OFIFG/ ACCVIFG	Non-maskable	0FFFCh	30
			0FFFAh	29
			0FFF8h	28
			0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer_A2	TACCR0 CCIFG	maskable	0FFF2h	25
Timer_A2	TACCR1 CCIFG, TAIFG	maskable	0FFF0h	24
			0FFEEh	23
			0FFECh	22
ADC10	ADC10IFG	maskable	0FFEAh	21
USI	USIIFG USISTTIFG	maskable	0FFE8h	20
I/O Port P2 (2)	P2IFG.6, P2IFG.7	maskable	0FFE6h	19
I/O Port P1 (8)	P1IFG.0 to P1IFG.7	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
Unused			0FFDEh 0FFCDh	15 - 0

Sample Code

- Toggle LEDs using interrupts from Timer_A in up mode

```
#include <io430x11x1.h> // Specific device
#include <intrinsics.h> // Intrinsic functions
#define LED1 BIT0
#define LED2 BIT4
void main (void) {
  WDTCTL = WDTPW|WDTHOLD; // Stop watchdog timer
  P1OUT = LED1;   P1DIR = LED1;
  TACCR0 = 49999; // Upper limit of count for TAR
  TACCTL0 = CCIE; // Enable interrupts
  TACTL = MC_1|ID_3|TASSEL_2|TACLR;
  // Up mode, divide clock by 8, clock from SMCLK, clear
  __enable _interrupt(); // Enable interrupts (intrinsic)
  for (;;) { // Loop forever doing nothing }
}
// Interrupt service routine for Timer_A
#pragma vector = TIMERA0_VECTOR
__interrupt void TA0_ISR (void){
  P2OUT ^= LED1|LED2; // Toggle LEDs
}
```

Summary

- Interrupts: a subroutine generated by the hardware at an unpredictable time
- Issues to consider:
  - How to set up and know there is an interrupt?
  - How to know where is the interrupt service routine?
  - Must not interfere the original program
  - The shared-data problem
- MSP430 interrupt mechanism