

# NET3001

## Interrupts

# Interrupts concepts

- Programs runs **synchronously**
  - CPU: fetch-execute cycle (clocked operation).
  - Activities performed according to order of instructions in program
- What is an interrupt?
  - Special **event** requiring CPU to stop normal program execution and perform some service related to that event.
  - Examples: timer completion, I/O event, timeout, I/O completion, illegal opcode, arithmetic overflow, divide-by-0, etc.
- Interrupts allow for **asynchronous** execution of service routines
  - Example : Classroom questions.

# Why are Interrupts Used ?

- Coordinate the CPU with activities of I/O devices
  - I/O devices under control of their own circuitry
  - Data for I/O not under the control of CPU (fire alarm)
  - Data transfer require CPU and I/O device to **synchronize**
    - Device Response times: at least one order of magnitude slower than instruction execution
    - CPU response time: want immediate action, preventing CPU from being tied up
- Remind the CPU to perform routine tasks.
- Provide a graceful way to handle software/hardware errors

# Interrupt Mechanism

- Originally appeared as synchronization mechanism between CPU - I/O
  - End of I/O: device informs data transfer successful.
  - Timer: a given amount of time elapsed
  - Parity error in memory (memory checking circuits).
  - Wrong Opcode: Control Unit detects non existing opcode
  - Requires a wired connection to the CPU



- Hardware Interrupt : signal issued by circuitry (event)
- CPU interrupt response: temporarily branches to execute services provided by **Interrupt Service Routine (ISR)** (executing program delayed for a while).
  - Hardware event triggers software response (internal CPU circuitry)

# Real Time Programming

## Polling

- Polling: CPU asks all the devices whether there is anything to do.
  - **sequential programming**: next instruction determined by control transfer instructions:

```
/* polling */  
while (1) {  
    if (keyPressed)  
        doKeyStuff();  
    if (timePassed)  
        doOtherStuff();  
}
```

# Real Time Programming Interrupts

- Physical events (or time) causes interrupt
- Interrupted processing: CPU doesn't "know" it was interrupted
  - temporarily suspend current thread of control
  - runs ISR
  - resumes suspended thread of control

# Sources of Interrupts

- external events
  - keypress, limit sensors
- timeout
  - passage of time
- with the above hardware interrupt mechanism in place, we can use it for other purposes
  - Make easier development of low level applications
  - illegal instruction
  - divide by 0
  - software interrupt

# Interrupt Sources

- Software Interrupts
  - sometimes called “traps” or “System Calls” or “software interrupt”
  - if you have an “operating system”, this is an easy way to request a service
- Interrupt circuitry activated as result of the execution of an instruction
- Synchronous in nature because :
  - occur as result of execution of an instruction.
  - precise instant when interrupt is going to occur can be identified
- ***NOTE: not available on MSP430***

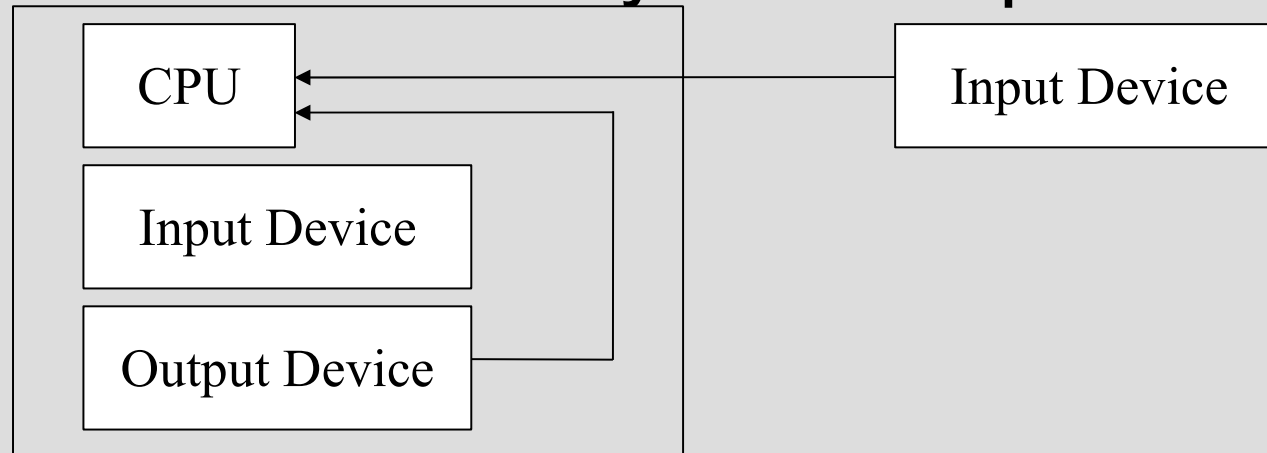
# Interrupt Sources

## Exceptions

- Extension to the hardware error interrupts
- Used to signal software errors (e.g. Divide overflow)
- Provide a graceful way to exit or handle the error
- They are still “synchronous”
- They are implemented using the same hardware Interrupt circuitry
- common example: illegal instruction trap
- ***NOTE: not available on MSP430***

# HW Interrupt Sources

- External : Circuitry is off chip
- Internal : Circuitry is on chip



- MSP430 Interrupt Sources
  - External : P1, P2 and NMI
  - Internal : Many, to be covered in coming lectures.

# Interrupt Properties Maskable

- Interrupt Maskability
  - **Maskable** interrupts can be ignored/postponed by the CPU
    - Enabled before interrupting the CPU (setting an associated enable flag).
  - **NonMaskable** interrupts cannot be ignored by the CPU
- Interrupt request *pending*: active but not yet serviced
  - May or may not be serviced (depending on whether or not it is enabled)

# Global Interrupt masking

- there is a bit (in the status register) which disables *all* maskable interrupt sources
  - MSP430:

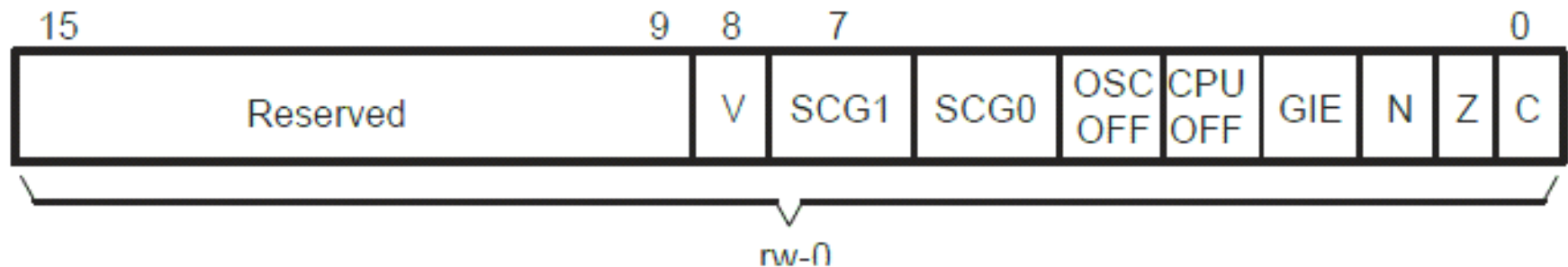


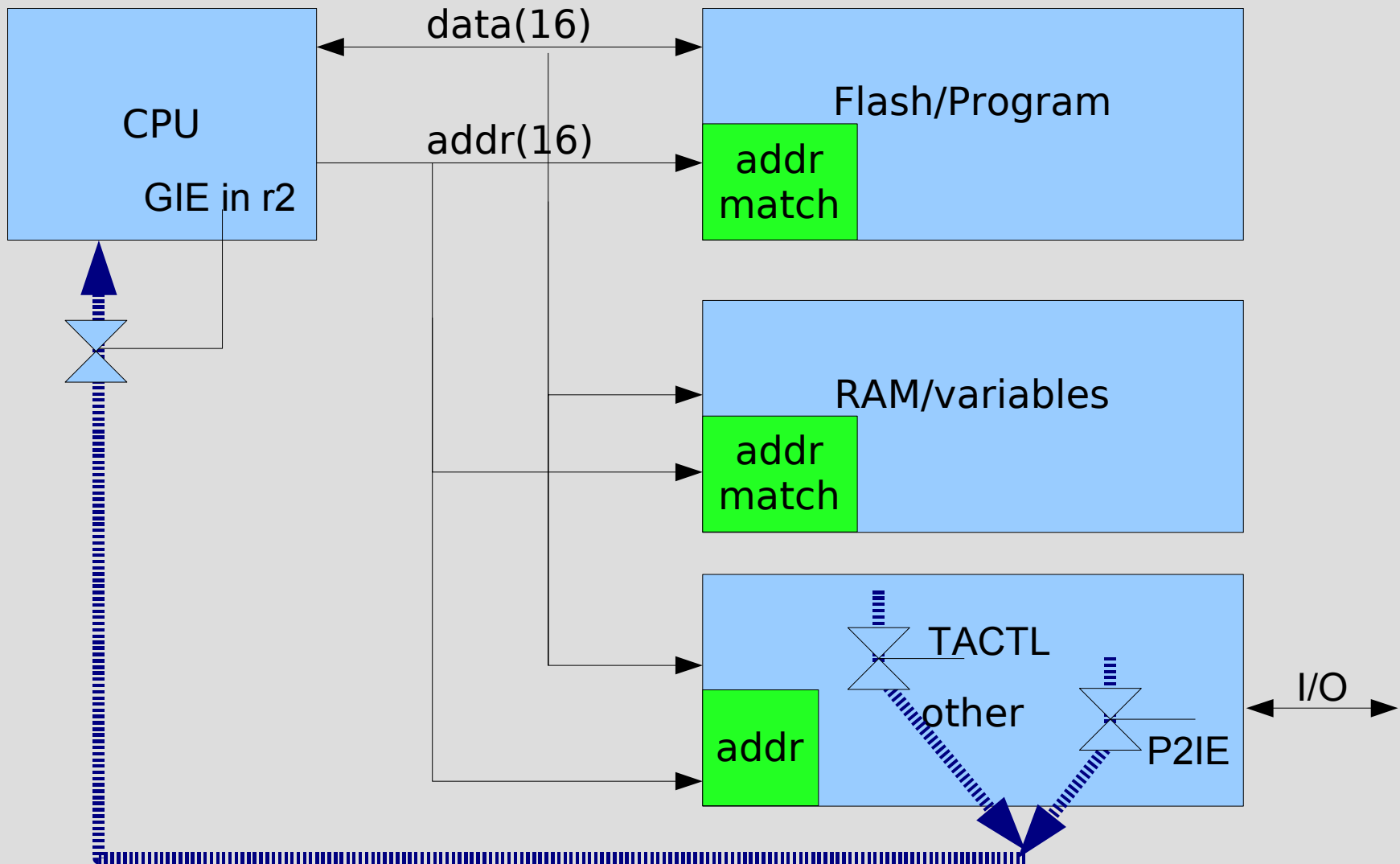
Table 3-1 describes the status register bits.

- **GIE** bit (CCR) globally enables/disables all maskable interrupts
  - `EINT` Sets I bit (Enables all maskable interrupts)
  - `DINT` Clears I bit (Disables all maskable interrupts)
- Nonmaskable interrupts: not affected.

# Local Interrupt masking

- in addition to the global enable, each I/O section has its own interrupt enable capability
  - for example,
    - TIMER A: TACTL, bit 1 enables interrupts
    - PORT2: P2IE has 8 bits, one for each interrupt
    - A/D converter: ADC10CTL0, bit 3 enables interrupt
  - several are gathered in the IE1/2 registers
    - IE2 bits 0 & 1 are UART interrupts

# Microcontroller Interrupt Flow



# MSP430 Interrupt Processing Sequence

- When does the MCU recognize interrupt requests?
  - When it completes the execution of the current instruction
    - Why?
  - Instructions are atomic

# Interrupt Processing Sequence

On the MSP430, the interrupt service cycle includes:

1. finish the current instruction
2. save the PC value (r0) on the stack
3. save status (r2) on the stack
4. clear the status register  
*this turns GIE off! and disables further interrupts and disables any low-power mode*
5. fetch the appropriate vector into r0

# Interrupt Properties : Priority

- What happens if two interrupts occur at the same time ?
  - CPU assigns **priority** to each potential source
- First, priority ...
  - MSP430 Priority : Assigned in order of their position in the vector table
    - Increasing priority at higher address
      - Reset : highest priority
      - Timer : lower priority

# MSP430 Interrupt Vector Table

Address	Usage
0xFFFFE	power on address; also used by watchdog reset
0xFFFFC	non maskable interrupt (not used in this course)
0xFFFFA	timer B main counter interrupt
0xFFFF8	timer B comparison interrupt
0xFFFF6	not used
0xFFFF4	watchdog timer
0xFFFF2	timer A main counter interrupt
0xFFFF0	timer A comparison interrupt
0xFFEE	uart receive interrupt
0xFFEC	uart transmit interrupt
0xFFEA	ADC finished interrupt
0xFFE8	not used
0xFFE6	Port 2 interrupt
0xFFE4	Port 1 interrupt
0xFFE2/0	not used

# Interrupt Service Routine

- ISR is the term usually used for the subroutine that handles the interrupt
  - on the way in, you know that interrupts are disabled
    - GIE is clear, so *general* interrupts are blocked
    - and the IFLG flag for *this* interrupt is set, blocking further interrupts of *this* type
  - some time in the routine, you probably want to clear the IFLG
    - to allow more interrupts of *this* type
  - when you execute RTI, the GIE bit will be reset *automatically*
    - you may wish to set GIE yourself, but this is not a common technique

# Interrupt Properties : Priority

- What if an interrupt occurs during the execution of the previous ISR?
  - Answer: [interrupt processing sequence](#)
  - **Scenario 1:** ISR for a low-priority interrupt is running when a higher-priority interrupt occurs.
  - **Scenario 2:** ISR for a high-priority interrupt is running when a lower-priority interrupt occurs.
  - **Scenario 3:** ISR is running when a reset occurs.

# Steps of Interrupt Programming

## Step 1. Initialize the interrupt vector table (Install the vector)

- assembly: use the `.vectors` section and position the `.word` pointer to your ISR
- C: add the phrase `interrupt(vector_number)` in front of your ISR

## Step 2. Write the interrupt service routine

- assembly: finish your code with `rti`
- C: add the phrase `interrupt(vector_number)` in front of your ISR; the compiler will automatically terminate your routine with `rti`

## Step 3. Re-enable the interrupt (if maskable)

- Locally enable maskable interrupts (IFLG)
- Optionally, globally enable interrupts

# Port 2 Key Wakeup

- When used as a general purpose I/O port :
  - Port 2 Input Register (**P2IN**)
  - Port 2 Output Register (**P2OUT**)
  - Port 2 Data Direction Register (**P2DIR**)
- When used as an interrupt source :
  - Port 2 Interrupt Enable Register (**P2IE**)
    - Enables interrupts on associated pin
  - Port H Interrupt Flag Register (**P3IFG**)
    - Bit is set when an active edge detected on associated pin
    - Also generates interrupt, if enabled
    - Read: To determine which pin is set
    - Write: To clear the pin (to acknowledge the interrupt)
  - Port 2 Input Edge Select Register (**P2IES**)
    - Sets the active edge of a bit as rising (0) or falling (1)
    - Also activates associated pull-up/down device, if enabled
- The other registers for Port 2 are **P2SEL** and **P2REN**

# Simple Keypad Driver

- Program displays all keystrokes on the 7 segment display
  - Runs forever
  - Interrupt-driven : Port 2 interrupts signal key press

```
main()
{
    ; Init Hardware
    ; Enable interrupts
    eint();
    for (;;)    {
        while (key == $FF)
            {}
        show7Seg(key);
        key = $FF
    }
}
```

```
key db $FF
```

```
interrupt(PORT2_VECTOR) KISR(void)

    small delay
    measure voltage
    convert to key number
    key = key number

    acknowledge irq
    reti
```

# Example : Interrupt Driven Keypad

```
unsigned char key = 0xFF;

int main(void) {
    P2DIR = 0xF0;      // 7 segment display is output, top bits
    P2IE = 1;         // enable keyboard interrupt
    P2IES = 0;        // select rising edge
    eint();           // enable interrupt
                    // sets the GIE in r2

    while(1) {
        // wait for a key to appear
        while (key == 0xFF)
            ;
        // light the 7 segment display
        P2OUT = (P2OUT & 0x0F) | (key<<4);
        // and clear out the key
        key = 0xFF;
    }
}
```

# Example : Interrupt Driven Keypad ISR

```
interrupt(PORT2_VECTOR) key_isr(void) {
    int r;

    // measure the A/D value on keyboard pin
    r = measureKey();
    // convert this to a key value and set the global
    key = convertKeyMeasuretoKey(r);

    // force the interrupt flag down,
    // to get ready for next
    P2IFG &= 0xFE;

} // <-----the compiler puts a "rti" here
// and adds this routine to the vector table
```

# Programming Interrupts in C

- ISR must be declared as a special kind of subroutine with certain properties
  - the address of the ISR is added to the vector table
  - the ISR ends with an “reti” instruction
    - so you must not call your ISR as a normal subroutine
  - the ISR preserves all registers

```
interrupt (TIMERB1_VECTOR) timerB_isr(void) {  
    // do things at interrupt time  
    // and reset the interrupt request  
}    // <---finishes with a “reti” instruction
```

# ISR

- your *Interrupt Service Routine* should
  - perform any high speed operations required
  - do as little as possible
  - finish by resetting the interrupt request
    - on the MSP430, some types of interrupt are *automatically* reset, such as TimerA and TimerB
    - most interrupts require a manual reset
    - for example, the keyboard (Port 2 interrupts) must finish with the line
      - `P2IFG = P2IFG & 0xFE;`
    - this process is often called “acknowledging the interrupt”

# Vector Table

- the “interrupt” keyword in C adds the address of the ISR to your vector table
- a blank version vector table is included in the “libraries” and gets filled in by the linker

# C Example : Interrupt Driven Keypad

```
interrupt (PORT2_VECTOR) KISR(void) { // you can use any name
    int r;

    r = measureKeyboard();
    key = convertKeyMeasuretoKey(r);

    P2IFG &= 0xFE; // acknowledge the interrupt
}
```

# C Example : Interrupt Driven Keypad

```
/* this version turns interrupts back on immediately */

interrupt (PORT2_VECTOR) KISR(void) { // you can use any name
    int r;

    eint(); // turn global interrupts back on
            // we won't get another keyboard interrupt, because
            // the P2IFG bit is still set
            // we MIGHT get a timer interrupt, but that's ok

    r = measureKeyboard();
    key = convertKeyMeasuretoKey(r);

    P2IFG &= 0xFE; // acknowledge the interrupt
}

```

# C Example : Interrupt Driven Keypad (assembler)

```
.text
kISR:  push r15

      call #measureKeyboard
      ; returns the key value in r15

      ; this routine takes r15 as its argument and returns r15
      call #convertKeyMeasuretoKey

      ; save it
      mov r15, &key

      ; clear the interrupt request bit (acknowledge)
      bic.b #1, &P2IFG
      pop r15
      reti

.section .vectors, "a"
.org 6
.word kISR
.org 0x16
.word start
```

# Interrupt Latency & Overhead

- **Latency** (Response time)

- Delay from the time that an interrupt occurs until it is serviced.

- **Overhead**

- Time taken to perform a service, over and above the service itself.

- Subroutine overhead

- push registers to save
- establish the “frame” r1->r4

- Interrupt overhead.

- Saving registers ( $\leq 4$  cycles)
- RTI ( 5 cycles)
- 9 cycles = 9 usec for a 1MHz clock.

- Interrupts do provide short latency (if enabled) but incur overhead.

# Summary of Interrupt Coding

- in Assembler

- in main code
  - enable interrupts `EINT`
- in ISR
  - save/restore all regs4-15
  - ack the interrupt
  - finish with `RETI`
- write the IRQ vector

- in C

- in your main code
  - enable interrupts `eint()`
  - declare the ISR as `interrupt(int_number)`
    - this automatically adds to the vector table
    - this automatically saves all registers
    - this automatically finishes with `RETI`
- in your ISR
  - ack the interrupt

- in both cases
  - be quick in the ISR, do your heavy work in the main code

# Interrupt-Driven versus Polled I/O

## Polling

**Pro:** shortest CPU latency (response time) if busy-waiting

**Con:** either CPU can't do other work **or** the device's latency dependent on the amount of other work.

## Interrupts

**Pros:** - Deterministic latency for a device

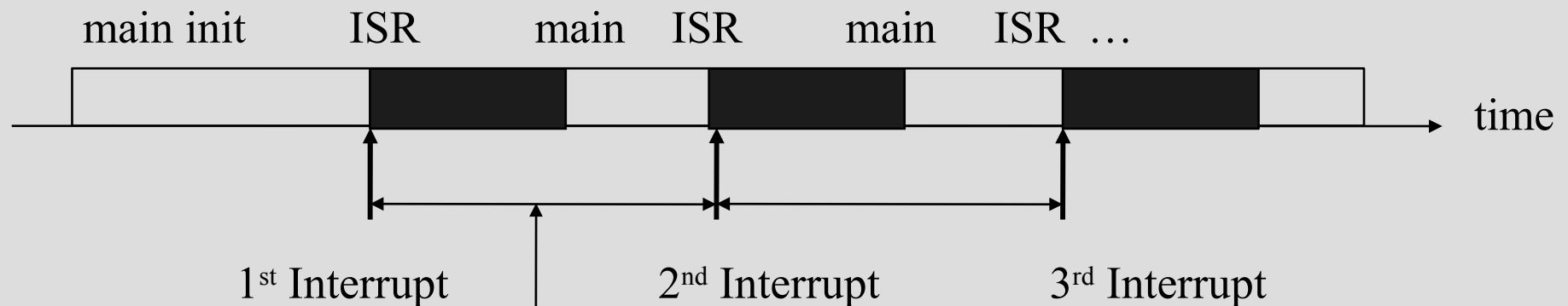
- CPU can perform other work when idle
- Software structure remains clean when handling many interrupts

**Con:** Interrupt Overhead

- Saving and restoring of CPU status and other registers.

# Analysis of Interrupt-Driven Program Behaviour

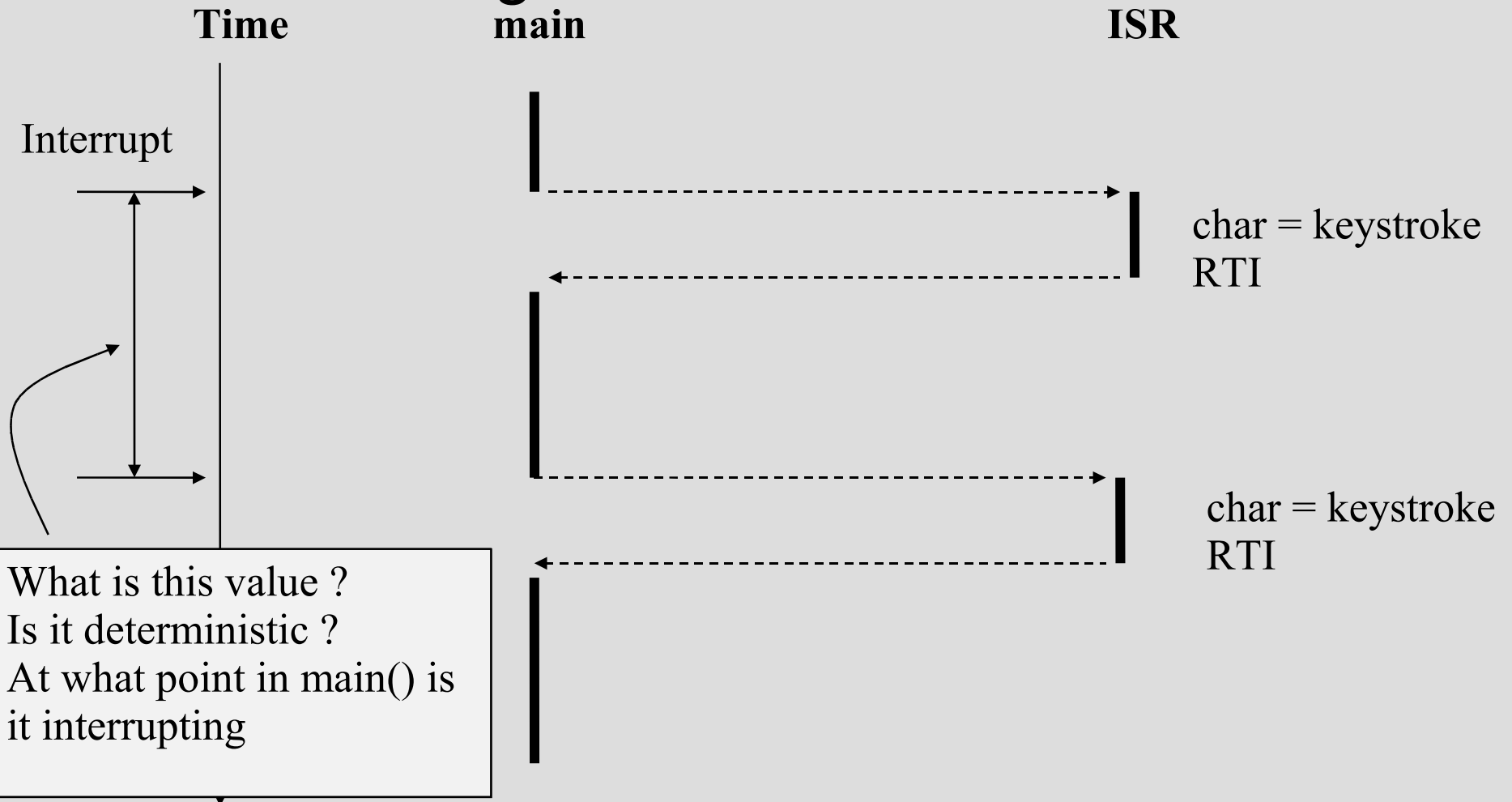
- The software is separated into background and foreground **threads of control**
- CPU Utilization Diagram



What is this value ?  
Is it deterministic ?  
At what point in main() is it interrupting

# Behaviour Analysis of Interrupt-Driven Program

- Thread Diagram



# Event-Driven Thinking : An Interference Scenario

*In the main loop :*

- *Where can interrupts happen ?*
- *Can any data be lost ?*

waiting:

```
cmp.b #0xFF, &key
```

```
jeq waiting
```

```
mov.b &key, r15
```

```
mov.b #0xFF, &key
```

```
rla.b r15      ; move key to upper 4 bits
```

```
rla.b r15
```

```
rla.b r15
```

```
rla.b r15
```

```
mov.b r15, P2OUT ; and display on the 7 segment
```

```
br #waiting
```

# Event-Driven Thinking : An Interference Scenario

*In the main loop :*

- *Where can interrupts happen ?*
- *Can any data be lost ?*

waiting:

```
cmp.b #0xFF, &key  
jeq waiting
```

```
rla.b &key          ; move key to upper 4 bits
```

```
rla.b &key
```

```
rla.b &key
```

```
rla.b &key
```

```
mov.b &key, P2OUT   ; and display on the 7 segment
```

```
mov.b #0xFF, &key
```

```
br #waiting
```

# Event-Driven Thinking : An Interference Scenario

*Interference : what are the chances of "key" being corrupted?*

*Critical Sections : how can we protect "key" and prevent the corruption?*

*Just an introduction .... more to come.*

# Types of Computer Event Flow

- Polled, inline code, no O/S
  - like the code we've been creating
- Polled, inline code, with an O/S
  - like the code you wrote in earlier courses
- Interrupt based I/O, no O/S
  - like the code for the next few assignments
- Interrupt based I/O, serviced by the O/S
  - like small operating systems: OS9
- Cooperative multi-tasking
  - like Windows 3/95, original Mac OS
- Preemptive multi-tasking
  - like QNX
- Preemptive multi-tasking with virtual memory
  - like Unix, Linux, WindowsNT/XP, Mac OSX

# Simple code flow

- Polled, inline code, no O/S
  - written just as a loop of read/test/decide/act
  - it's tricky to appear to do 2 or more things at once
  - it's difficult to modify
- Polled, inline code, with an O/S
  - again, a loop, except the access to I/O is done by calling back into the O/S for services (via subroutine calls)

# OS interaction with interrupts

- **Interrupt based I/O, no O/S**
  - quicker response to events
  - still need a main task that handles the “doing 2 things at once” loop
- **Interrupt based I/O, serviced by the O/S**
  - the O/S deals with common tasks:
    - get a key, detect an action
    - measure the passage of time
    - measure the battery
    - read/write to disk/flash
  - you still end up calling the O/S for services, via subroutines

# More sophisticated O/S interaction

- Cooperative multi-tasking
  - The situation is turned around: THE O/S CALLS YOUR PROGRAM AS A SUBROUTINE!
    - you must write your code as a subroutine, and it should act quickly and return ASAP
    - yes, you ARE allowed to call back into the O/S
      - this creates deep nests of calling/call-back
  - the rest of this course deals with cooperative multitasking

# More sophisticated O/S interaction

- Preemptive multi-tasking
  - You can write your program as if it's the only one, and the O/S will switch back and forth between other programs, making it appear that 2 things are happening at once
  - You can call on the O/S for services, but quite often, the O/S will “wait” on your call, for the service to finish.
  - not covered in this course