Embedded Systems Programming

INTERRUPTS

M.Eng. Mariusz Rudnicki
Interrupts

- **Interrupt** is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.
- An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing.
- The processor responds by:
  - suspending its current activities,
  - saving its state,
  - and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event.
- After the interrupt handler finishes, the processor resumes normal activities.
HARDWARE INTERRUPTS

- Interrupts can be divided into:
  - hardware interrupts,
  - software interrupts.

- **Hardware interrupts** are used by devices to communicate that they require attention from the CPU or operating system.
- Hardware interrupts are implemented using electronic signals that are sent to the processor from an external device;
- Hardware interrupts are asynchronous and can occur in the middle of instruction execution.
SOFTWARE INTERRUPTS

- **Software interrupt** is caused either by:
  - an exceptional condition in the processor;
  - or a special instruction in the instruction set which causes an interrupt when it is executed;

- This type of interrupts are often called **traps** or **exceptions** and are used for errors or events occurring during program execution and they cannot be handled within the program itself:
  - divide-by-zero exception;
  - communication with disk controller to request data to be read or written to the disk.
**INTERRUPTS**

- Generally we can say:
  - Each interrupt has its own interrupt handler;
  - The number of hardware interrupts is limited by the number of interrupt request (IRQ) lines to the processor;
  - There may be hundreds of different software interrupts.
INTERRUPTS

- **Hardware interrupts were introduced as a way:**
  - to reduce wasting the processor's valuable time in polling loops,
  - waiting for external events.

- **Two hardware implementations:**
  - as a distinct system with control lines - Programmable Interrupt Controller (PIC) may be connected between the interrupting device and the processor's interrupt pin to multiplex several sources of interrupt onto the one or two CPU lines typically available,
  - integrated into the memory subsystem - interrupts are mapped into the system's memory address space.
INTERRUPTS

Types of interrupts:
- **Maskable interrupt (IRQ):** a hardware interrupt that may be ignored by setting a bit in an interrupt mask register's (IMR) bit-mask.
- **Non-maskable interrupt (NMI):** a hardware interrupt that lacks an associated bit-mask, so that it can never be ignored. NMIs are used for the highest priority tasks such as timers, especially watchdog timers.
- **Inter-processor interrupt (IPI):** a special case of interrupt that is generated by one processor to interrupt another processor in a multiprocessor system.
- **Software interrupt:** an interrupt generated within a processor by executing an instruction. Software interrupts are often used to implement system calls because they result in a subroutine call with a CPU ring level change.
- **Spurious interrupt:** an unwanted hardware interrupt generated by system conditions such as electrical interference on an interrupt line or through incorrectly designed hardware.
INTERRUPTS

- **Types of interrupts:**
  - **Precise interrupt:** an interrupt that leaves the machine in a well-defined state. Conditions:
    - The Program Counter (PC) is saved in a known place.
    - All instructions before the one pointed to by the PC have fully executed.
    - No instruction beyond the one pointed to by the PC has been executed (that is no prohibition on instruction beyond that in PC, it is just that any changes they make to registers or memory must be undone before the interrupt happens).
    - The execution state of the instruction pointed to by the PC is known.
  - **Imprecise interrupt:** an interrupt that does not meet above-mentioned requirements.

- **Interrupt storm.**
INTERRUPTS uC

- Programmable Multi-level Interrupt Controller

- Features
  - 3 interrupt levels
  - Round-robin scheduling for low-level interrupts
  - Programmable priority for low-level
Most (peripheral) modules have status flags, or interrupt flags, that can trigger execution of an interrupt. An interrupt is when a module signals to the AVR® CPU that the flow of the program code execution should be interrupted and a specific Interrupt Service Routine (ISR) program code should be executed. An interrupt is generated if all of the following requirements are meet:

- The condition that sets the interrupt flag is fulfilled.
- The specific interrupt is enabled in the module.
- The interrupt level is enabled in the Interrupt Controller.
- Global interrupts are enabled for the CPU.
INTERRUPTS uC

- The execution order of interrupts is determined first by their Interrupt Level and then their Interrupt Priority. Interrupt Level is specified individually for the Interrupts in the module where they originate, while the Interrupt Priority is determined by their fixed Interrupt Vector Address. Alternatively, for low-level interrupts, the Interrupt Priority can be based on a Round-robin scheme controlled by the Interrupt Controller.

- The Non-Maskable Interrupt (NMI) is a special interrupt that cannot be disabled, and is used to for system critical interrupts. NMI is used for critical failures such as a failure in the crystal oscillator.
void TimerCfg(){
    // interrupt configuration
    TCE0.PER = 14400;
    // timer overflow interrupt
    TCE0.INTCTRLA = TC_OVFINTLVL_LO_gc;
    // LOW level interrupt enabled
    PMIC.CTRL = PMIC_LOLVLEN_bm;

    // timer configuration
    TCE0.CTRLB = TC_WGMODE_NORMAL_gc;    // normal mode
    // TCC0.CTRLFSET = TC0_DIR_bm;        // count down
}

INTERRUPTS uC
{  

//Disable interrupts, just for safety  
USARTC0_CTRLA = 0;
//8 data bits, no parity and 1 stop bit  
USARTC0_CTRLC = USART_CHSIZE_8BIT_gc;
    //Enable receive and transmit  
USARTC0_CTRLB = USART_TXEN_bm | USART_RXEN_bm;

//Enable interrupts, just for safety  
USARTC0_CTRLA = USART_RXCINTLVL_LO_gc | USART_DREINTLVL_LO_gc;
/* Enable PMIC interrupt level low. */
    PMIC.CTRL |= PMIC_LOLVLEN_bm;
USARTC0.STATUS &=(0<<PIN7_bp);
    sei();
}

/* Enable PMIC interrupt level low. */
    PMIC.CTRL |= PMIC_LOLVLEN_bm;
Interrupts uC

ISR(USARTC0_DRE_vect){
  // Disable DRE interrupts.
  if (TxD_Data_count_C0 == FRAME_LEN){
    uint8_t tempCTRLA = USARTC0.CTRLA;
    tempCTRLA = ( tempCTRLA & ~USART_DREINTLVL_gm ) |
               USART_DREINTLVL_OFF_gc;
    USARTC0.CTRLA = tempCTRLA;
  }
  else{
    // Start transmitting.
    sendChar_USARTC0(C0_TxD_BUFFER[TxD_Data_count_C0]);
    TxD_Data_count_C0++;   }
}

ISR(USARTC0_RXC_vect){
  C0_RxD_BUFFER[RxD_Data_count_C0] = USARTC0.DATA;
  RxD_Data_count_C0++;
  if(RxD_Data_count_C0 == FRAME_LEN) RxD_Data_count_C0=1;
  return;
}
Cortex-M0 provides an interrupt controller as an integral part of the exception mode, named as “Nested Vectored Interrupt Controller (NVIC)”. It is closely coupled to the processor kernel and provides following features:

- Nested and Vectored interrupt support;
- Automatic processor state saving and restoration;
- Reduced and deterministic interrupt latency.

This NVIC architecture supports 32 (IRQ[31:0]) discrete interrupts with 4 levels of priority.

- All of the interrupts and most of the system exceptions can be configured to different priority levels.
- When an interrupt occurs, the NVIC will compare the priority of the new interrupt to the current running one’s priority. If the priority of the new interrupt is higher than the current one, the new interrupt handler will override the current handler.
When any interrupts is accepted, the starting address of the interrupt service routine (ISR) is fetched from a vector table in memory. There is no need to determine which interrupt is accepted and branch to the starting address of the correlated ISR by software. While the starting address is fetched, NVIC will also automatically save processor state including the registers “PC, PSR, LR, R0~R3, R12” to the stack. At the end of the ISR, the NVIC will restore the mentioned registers from stack and resume the normal execution. Thus it will take less and deterministic time to process the interrupt request.

The NVIC supports “Tail Chaining” which handles back-to-back interrupts efficiently without the overhead of states saving and restoration and therefore reduces delay time in switching to pending ISR at the end of current ISR. The NVIC also supports “Late Arrival” which improves the efficiency of concurrent ISRs. When a higher priority interrupt request occurs before the current ISR starts to execute (at the stage of state saving and starting address fetching), the NVIC will give priority to the higher one without delay penalty. Thus it advances the real-time capability.
void set_interrupt_vme(){
    _asm cli
    BuforAdresuIRQ_VME2 = _dos_getvect( VME_INTVECT ); //save old INTVEC
    _dos_setvect( VME_INTVECT, MVME_INTERRUPT ); //set new INTVEC
    andp( ICM_S, (unsigned char)~VME_MIRQ_S ); //INT15 enable
    outp( VIMR, ( RS_MINT | PORT0_MINT ) ); // odblokowanie CIO i obslugiwanych przerwan VME
    _asm sti
}

void clear_interrupt_vme()
{
    outp( VIMR, 0xFE ); // VME INT disable
    orp ( ICM_S, VME_MIRQ_S ); //
    _dos_setvect( VME_INTVECT, BuforAdresuIRQ_VME2 ); //restore old INTVEC
}
INTERRUPTS in DOS

#include <stdio.h>

typedef struct
{
    unsigned int count;
    unsigned int delay;
} timer_t;

#define TIMER count
#define DELAY delay

void _interrupt _far MVME_INTERRUPT()
{
    _asm{
        cli
        START_PRZERWANIA:
            mov dx,VIPR // odczytaj rejestr przerwan
            in al,dx
            and al,0xFE
            cmp al,0
            jz KONIEC_VME_INT
            test al,PORT0_MINT
            jnz PRZERWANIE_Z_PORTU_IO
            jmp KONIEC_VME_INT
    }
}
INTERRUPTS in DOS

```
PRZERWANIE_Z_PORTU_IO:
  mov  dx,VIACKR+2*PORT1_INT          //odpowiedz IACK
  in   al,dx
  cmp al,0x04                         //przerwanie od timera 1
    je CT1
  cmp al,0x02                         //przerwanie od timera 2
    je CT2
  cmp al,0x00                         //przerwanie od timera 3
    je CT3
  cmp al,0x3c
    je ADC_
  cmp al,0x06
    je KONIEC_VME_INT
  CT1:
    jmp START_PRZERWANIA

KONIEC_VME_INT:
  jmp EOI_VME
```
INTERRUPTS in DOS

CT2:
les bx,V1
mov BYTE PTR es:[bx],BYTE PTR CT2CSR_REG  //Kasuj flaga przerwania
mov BYTE PTR es:[bx],BYTE PTR RES_IPIUS|0x06

inc FIRST_SAMPLE
cmp FIRST_SAMPLE,0xFFFF
je STOP_CT2
jmp START_PRZERWANIA

STOP_CT2:
mov ax,FIRST_SAMPLE
xor ax,FIRST_SAMPLE
mov FIRST_SAMPLE,ax

jmp START_PRZERWANIA

CT3:
jmp START_PRZERWANIA
INTERRUPTS in DOS

ADC_
  inc ADCDATA
  inc FIRST_SAMPLE
  cmp FIRST_SAMPLE,0xFFFF
  je STOP_CT2
  jmp START_PRZEWANIA

EOI_VME:
  mov al,EOI8259 ; wyslij komende EOI do kontrolerow
  mov dx,IRR_M
  out dx,al
  mov dx,IRR_S
  out dx,al
 sti

} // koniec asm
}

#pragma check_stack (on)
INTERRUPTS in OS QNX Neutrino

- QNX

- Interrupt Source
  - PIC
  - uKERNEL

- Driver A
  - HANDLER()
    - ....
    - return event;
  - main()
    - ....

- Driver B
  - main()
    - ....
INTERRUPTS in OS ONX Neutrino

- Functions:
  
  ```c
  id = InterruptAttach (int intr,
          struct sigevent *(*handler)(void *, int),
          void *area, int size, unsigned flags);
  id = InterruptAttachEvent (int intr, struct sigevent *event,
                           unsigned flags);
  InterruptDetach (int id);
  InterruptWait (int flags, uint64_t *reserved);
  InterruptMask (int intr, int id);
  InterruptUnmask (int intr, int id);
  InterruptLock (struct intrspin *spinlock);
  InterruptUnlock (struct intrspin *spinlock);
  ```

- We have to obtain I/O permissions for these functions;
- ThreadCtl(_NTO_TCTL_IO, 0);
- We have to belong to a group of system administrators (root (userid 0)).
INTERRUPTS in RT Linux OS

- RT Linux uses VM concept limited to interrupt emulation.
- A layer of emulation software between Linux kernel and interrupt controller hardware.
- Prevents disabling of interrupts by Linux.
- cli, sti and iret are replaced by corresponding soft (emulated) versions.
INTERRUPTS in RT Linux OS

- There are two types of interrupts: Hard and Soft.
- Hard interrupts are the actual device interrupts and have less latency.
- First RT handler (if any) is called and if it permits sharing, the interrupt is passed on to Linux.
- Very limited set of kernel functions can be called from them.
INTERRUPTS in RT Linux OS

S_CLI :
  movl $0, SFIF

S_STI :
  sti
  pushfl
  pushl $KERNEL_CS
  pushl $1f
S_IRET
1:
INTERRUPTS in RT Linux OS

S_IRET:
  push %ds
  pushl %eax
  pushl %edx
  movl $KERNEL_DS, %edx
  mov %dx, %ds
  cli
  movl SFREQ, %edx
  andl SFMASK, %edx
  bsfl %edx, %eax
  jz 1f
  S_CLI
  sti
  jmp SFIDT (,%eax,4)
INTERRUPTS in RT Linux OS

1:

movl $1, SFIF
popl %edx
popl %eax
popl %ds
iret
INTERRUPTS in RT Linux OS

- Soft interrupts are at par with normal Linux interrupts.
- They don’t provide real time performance.
- Kernel functions may be called from them.
- Can be delayed for considerable periods of time.
- Serviced when system switches back to Linux.
- Used in implementation of RT fifos.
A specific IRQ can be requested with `rtl_request_irq(irq, handler, regs)`. An IRQ can be released using `rtl_free_irq(irq)`. The “handler” executes with hardware interrupts disabled. If it is necessary to receive further interrupts, re-enabling is done with `rtl_hard_enable_irq(irq)`. An interrupt driven thread is created as usual by calling `pthread_create()`. The thread calls `pthread_suspend_np()` and blocks. It is assumed that the interrupt handler will call `pthread_wakeup_np()` to wakeup this thread.
References

2. XMEGA Interrupts and the Programmable Multi-level Interrupt Controller.pdf
3. DS_NUC230_240(AE)_Series_EN_Rev1.01.pdf
5. https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RTapplication#Things_you_should_not_do_in_IRQF_NODELAY_context
References