Interrupts (I)

Lecturer: Sri Parameswaran
Notes by Annie Guo
Lecture overview

- Introduction to Interrupts
  - Interrupt system specifications
  - Multiple Sources of Interrupts
  - Interrupt Priorities
- Interrupts in AVR
  - Interrupt Vector Table
  - Interrupt Service Routines
  - System Reset
    - Watchdog Reset


## CPU Interacts with I/O

Two approaches:

- **Polling**
  - Software queries I/O devices.
  - No hardware needed.
  - Not efficient.
    - CPU may waste processor cycles to query a device even if it does not need any service.

- **Interrupts**
  - I/O devices generate signals to request services from CPU.
  - Need special hardware to implement interrupt services.
  - Efficient.
    - A signal is generated only if the I/O device needs services from CPU.
Interrupt Systems

- An interrupt system implements interrupt services
- It basically performs three tasks:
  - Recognize interrupt events
  - Respond to the interrupts
  - Resume normal programmed task
Recognize Interrupt Events

- **Interrupt events**
  - Associated with interrupt signals:
    - In different forms, including levels and edges.
  - Can be multiple and synchronous
    - Namely, there may be many sources to generate an interrupts; a number of interrupts can be generated at the same time.

- Approaches are required to
  - Identify an interrupt event among multiple sources
  - Determine which interrupts to serve if there are multiple simultaneous interrupts
Respond to Interrupts

- Handling interrupt
  - Wait for the current instruction to finish.
  - Acknowledge the interrupting device.
  - Branch to the correct *interrupt service routine* (interrupt handler) to service interrupting device.
Resume Normal Task

- Return to the interrupted program at the point it was interrupted.
Interrupt Process Control

- Interrupts can be enabled or disabled
- Can be controlled in two ways:
  - Software control
    - Allow programmers to enable and disable selected/all interrupts.
  - Hardware control
    - Disable further interrupts while an interrupt is being serviced
Interrupt Recognition and Acknowledgement Hardware

Interruption Device

Signal Conditioning

IRQ-FF
Set
Reset

Pending Interrupt

Interrupt Signal to Sequence Controller

Interrupt Ack from Sequence Controller

SEQUENCE CONTROLLER

Interrupt Enable

Disable Interrupt Instruction

Enable Interrupt Instruction

Return from Interrupt Instruction

CPU
Interrupt Recognition and Ack.

- An Interrupt Request (IRQ) may occur at any time.
  - It may have rising or falling edges or high or low levels.
  - Frequently it is an active-low signal
    - multiple devices are wire-ORed together.
      - Recall open-collector gates
- Signal Conditioning Circuit detects these different types of signals.
- Interrupt Request Flip-Flop (IRQ-FF) records the interrupt request until it is acknowledged.
  - When IRQ-FF is set, it generates a pending interrupt signal that goes towards the Sequence Controller.
  - IRQ-FF is reset when CPU acknowledges the interrupt with INTA signal.
Interrupt Recognition and Ack. (cont.)

- Interrupts can be enabled and disabled by software instructions, which is supported by the hardware Interrupt Enable Flip-Flop (INTE-FF).
- When the INTE-FF is set, all interrupts are enabled and the pending interrupt is allowed through the AND gate to the sequence controller.
- The INTE-FF is reset in the following cases.
  - CPU acknowledges the interrupt.
  - CPU is reset.
  - Disable interrupt instruction is executed.
Interrupt Recognition and Ack. (cont.)

- An interrupt acknowledge signal is generated by the CPU when the current instruction has finished execution and CPU has detected the IRQ.
  - This resets the IRQ-FF and INTE-FF and signals the interrupting device that CPU is ready to execute the interrupting device routine.

- At the end of the interrupt service routine, CPU executes a return-from-interrupt instruction.
  - Part of this instruction’s job is to set the INTE-FF to re-enable interrupts.

- Nested interrupts can happen if the INTE-FF is set during an interrupt service routine.
  - An interrupt can therefore interrupt interrupting interrupts.
Multiple Sources of Interrupts

- To handle multiple sources of interrupts, the interrupt system must
  - Identify which device has generated the IRQ.
    - Using polling approach
    - Using vectoring approach
  - Resolve simultaneous interrupt requests
    - using prioritization schemes.
Polled Interrupts

- Software, instead of hardware, is responsible for finding the interrupting source.
  - The device must have logic to generate the IRQ signal and to set an “I did it” bit in a status register that is read by CPU.
  - The bit is reset after the register has been read.
Polled Interrupts Execution Flow

1. Device generates IRQ
2. CPU polls status registers of all devices
3. CPU found the interrupting device
4. CPU executes the service routine for that device
Polled Interrupt Logic

- Logic to generate IRQ
- Logic to reset IRQ when status register is read
- Logic to read status register and reset “I did it” bit
- Logic to set “I did it” bit
- Status register

Data

Address

Control
Vectored Interrupts

- CPU’s response to IRQ is to assert INTA.
- The interrupting device uses INTA to place information that identifies itself, called vector, onto the data bus for CPU to read.
- CPU uses the vector to execute the interrupt service routine.
Vectored Interrupting Device

**Hardware**

- Logic to generate IRQ
- Logic to reset IRQ
- Vector Information
- Three-State Driver

- INTA
- IRQ
- Data
- Address
- Control
Multiple Interrupt Masking

- CPU has multiple IRQ input pins.
- Masking enables some interrupts and disables other interrupts.
- CPU designers reserve specific memory locations for a vector associated with each IRQ line.
- Individual disable/enable bit is assigned to each interrupting source.
Multiple Interrupt Masking Circuit

Circuit Diagram:

- CPU
  - IRQ 0
  - IRQ 1
  - IRQ 2
  - ...
  - IRQ n

Interrupts:
- Interrupt 0
- Interrupt 1
- Interrupt 2
- Interrupt 3

Interrupt Enable Register:
- IRQ0E
- IRQ1E
- IRQ2E
- IRQ3E

IRQ0
IRQ1
IRQ2
IRQ3
Interrupt Priorities

When multiple interrupts occur at the same time, which one will be serviced first?

Two resolution approaches:

- Software resolution
  - Polling software determines which interrupting source is serviced first.

- Hardware resolution
  - Daisy chain.
  - Others
Software Resolution

Device generates IRQ

CPU polls status registers of all devices

CPU found the interrupting device

CPU executes the service routine for that device
Daisy Chain Priority Resolution

<table>
<thead>
<tr>
<th>CPU</th>
<th>Device 1</th>
<th>Device 2</th>
<th>Device n</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTA</td>
<td>INTA</td>
<td>INTA</td>
<td>INTA</td>
</tr>
<tr>
<td>IRQ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Data
- Address
- Control
Daisy Chain Priority Resolution (cont.)

- CPU asserts INTA that is passed down the chain from device to device. The higher-priority device is closer to CPU.
- When the INTA reaches a device that generated the IRQ, that device puts its vector on the data bus and does not pass along the INTA. So lower-priority devices do NOT receive the INTA.
Other Priority Resolutions

- **Separate IRQ Lines.**
  - Each IRQ line is assigned a fixed priority. For example, IRQ0 has higher priority than IRQ1 and IRQ1 has higher priority than IRQ2 and so on.

- **Hierarchical Prioritization.**
  - Higher priority interrupts are allowed while lower ones are masked.

- **Nonmaskable Interrupts.**
  - Cannot be disabled.
  - Used for important events such as power failure.
Transferring Control to Interrupt Service Routine

- Hardware needs to save the return address.
  - Most processors save the return address on the stack.
- Hardware may also save some registers such as program status register.
  - AVR does not save any register. It is programmers’ responsibility to save program status register and conflict registers.
- The delay from the time the IRQ is generated by the interrupting device to the time the Interrupt Service Routine (ISR) starts to execute is called *interrupt latency*.
Interrupt Service Routine

- A sequence of code to be executed when the corresponding interrupt is responded by CPU.
- Interrupt service routine is a special subroutine, therefore can be constructed with three parts:
  - Prologue:
    - Code for saving conflict registers on the stack.
  - Body:
    - Code for doing the required task.
  - Epilogue:
    - Code for restoring all saved registers from the stack.
    - The last instruction is the return-from-interrupt instruction.
Software Interrupt

- Software interrupt is the interrupt generated by software without a hardware-generated-IRQ.
- Software interrupt is typically used to implement system calls in OS.
- Some processors have a special machine instruction to generate software interrupt.
  - SWI in ARM.
- AVR does NOT provide a software interrupt instruction.
  - Programmers can use External Interrupts to implement software interrupts.
Exceptions

- Abnormalities that occur during the normal operation of the processor.
  - Examples are internal bus error, memory access error and attempts to execute illegal instructions.
- Some processors handle exceptions in the same way as interrupts.
  - AVR does not handle exceptions.
Reset

- Reset is a type of interrupt in most processors (including AVR).
- Nonmaskable.
- It does not do other interrupt processes, such as saving conflict registers. It initialize the system to some initial state.
Non-Nested Interrupts

- Interrupt service routines cannot be interrupted by another interrupt.
Nested Interrupts

- Interrupt service routines can be interrupted by another interrupt.
AVR Interrupts

- Basically can be divided into internal and external interrupts
- Each has a separated interrupt vector
- Hardware is used to recognize interrupt
- To enable an interrupt, two control bits must be set
  - the Global Interrupt Enable bit (I bit) in the Status Register
    - Using SEI
  - the enable bit for that interrupt
- To disable all maskable interrupts, reset the I bit in SREG
  - Using CLI instruction
- Priority of interrupts is used to handle multiple simultaneous interrupts
Set Global Interrupt Flag

- Syntax: `sei`
- Operands: none
- Operation: `I ← 1`.
  - Sets the global interrupt flag (I) in SREG. The instruction following `SEI` will be executed before any pending interrupts.
- Words: 1
- Cycles: 1
- Example:

  ```c
  sei ; set global interrupt enable
  sleep ; enter sleep state, waiting for an interrupt
  ```
Clear Global Interrupt Flag

- Syntax:  
  - cli
- Operands: none
- Operation: I $\leftarrow$ 0
- Clears the Global interrupt flag in SREG. Interrupts will be immediately disabled.
- Words: 1
- Cycles: 1

Example:
```
in r18, SREG ; store SREG value
cli ; disable interrupts
; do something very important here
out SREG, r18 ; restore SREG value
```
Interrupt Response Time

- The interrupt execution response for all the enabled AVR interrupts is basically four clock cycles minimum.
  - For saving the Program Counter (1 clock cycle)
  - For jumping to the interrupt routine (3 clock cycles)
Interrupt Vectors

- Each interrupt has a 4-byte (2-word) interrupt vector, containing an instruction to be executed after MCU has accepted the interrupt.
- The lowest addresses in the program memory space are by default defined as the section for Interrupt Vectors.
- The priority of an interrupt is based on the position of its vector in the program memory
  - The lower the address the higher is the priority level.
- RESET has the highest priority
# Interrupt Vectors in Mega64

<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Program Address(^{(1)})</th>
<th>Source</th>
<th>Interrupt Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x0000(^{(1)})</td>
<td>RESET</td>
<td>External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset</td>
</tr>
<tr>
<td>2</td>
<td>0x0002</td>
<td>INT0</td>
<td>External Interrupt Request 0</td>
</tr>
<tr>
<td>3</td>
<td>0x0004</td>
<td>INT1</td>
<td>External Interrupt Request 1</td>
</tr>
<tr>
<td>4</td>
<td>0x0006</td>
<td>INT2</td>
<td>External Interrupt Request 2</td>
</tr>
<tr>
<td>5</td>
<td>0x0008</td>
<td>INT3</td>
<td>External Interrupt Request 3</td>
</tr>
<tr>
<td>6</td>
<td>0x000A</td>
<td>INT4</td>
<td>External Interrupt Request 4</td>
</tr>
<tr>
<td>7</td>
<td>0x000C</td>
<td>INT5</td>
<td>External Interrupt Request 5</td>
</tr>
<tr>
<td>8</td>
<td>0x000E</td>
<td>INT6</td>
<td>External Interrupt Request 6</td>
</tr>
<tr>
<td>9</td>
<td>0x0010</td>
<td>INT7</td>
<td>External Interrupt Request 7</td>
</tr>
<tr>
<td>10</td>
<td>0x0012</td>
<td>TIMER2 COMP</td>
<td>Timer/Counter2 Compare Match</td>
</tr>
<tr>
<td>11</td>
<td>0x0014</td>
<td>TIMER2 OVF</td>
<td>Timer/Counter2 Overflow</td>
</tr>
<tr>
<td>12</td>
<td>0x0016</td>
<td>TIMER1 CAPT</td>
<td>Timer/Counter1 Capture Event</td>
</tr>
</tbody>
</table>
## Interrupt Vectors in Mega64 (cont.)

<table>
<thead>
<tr>
<th>#</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0x0018</td>
<td>TIMER1 COMPA, Timer/Counter1 Compare Match A</td>
</tr>
<tr>
<td>14</td>
<td>0x001A</td>
<td>TIMER1 COMPB, Timer/Counter1 Compare Match B</td>
</tr>
<tr>
<td>15</td>
<td>0x001C</td>
<td>TIMER1 OVF, Timer/Counter1 Overflow</td>
</tr>
<tr>
<td>16</td>
<td>0x001E</td>
<td>TIMER0 COMP, Timer/Counter0 Compare Match</td>
</tr>
<tr>
<td>17</td>
<td>0x0020</td>
<td>TIMER0 OVF, Timer/Counter0 Overflow</td>
</tr>
<tr>
<td>18</td>
<td>0x0022</td>
<td>SPI, STC, SPI Serial Transfer Complete</td>
</tr>
<tr>
<td>19</td>
<td>0x0024</td>
<td>USART0, RX, USART0, Rx Complete</td>
</tr>
<tr>
<td>20</td>
<td>0x0026</td>
<td>USART0, UDRE, USART0 Data Register Empty</td>
</tr>
<tr>
<td>21</td>
<td>0x0028</td>
<td>USART0, TX, USART0, Tx Complete</td>
</tr>
<tr>
<td>22</td>
<td>0x002A</td>
<td>ADC, ADC Conversion Complete</td>
</tr>
<tr>
<td>23</td>
<td>0x002C</td>
<td>EE READY, EEPROM Ready</td>
</tr>
<tr>
<td>24</td>
<td>0x002E</td>
<td>ANALOG COMP, Analog Comparator</td>
</tr>
</tbody>
</table>
## Interrupt Vectors in Mega64 (cont.)

<table>
<thead>
<tr>
<th>#</th>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0x0030(3)</td>
<td>TIMER1 COMPC Timer/Counter1 Compare Match C</td>
</tr>
<tr>
<td>26</td>
<td>0x0032(3)</td>
<td>TIMER3 CAPT Timer/Counter3 Capture Event</td>
</tr>
<tr>
<td>27</td>
<td>0x0034(3)</td>
<td>TIMER3 COMPA Timer/Counter3 Compare Match A</td>
</tr>
<tr>
<td>28</td>
<td>0x0036(3)</td>
<td>TIMER3 COMPB Timer/Counter3 Compare Match B</td>
</tr>
<tr>
<td>29</td>
<td>0x0038(3)</td>
<td>TIMER3 COMPC Timer/Counter3 Compare Match C</td>
</tr>
<tr>
<td>30</td>
<td>0x003A(3)</td>
<td>TIMER3 OVF Timer/Counter3 Overflow</td>
</tr>
<tr>
<td>31</td>
<td>0x003C(3)</td>
<td>USART1, RX USART1, Rx Complete</td>
</tr>
<tr>
<td>32</td>
<td>0x003E(3)</td>
<td>USART1, UDRE USART1 Data Register Empty</td>
</tr>
<tr>
<td>33</td>
<td>0x0040(3)</td>
<td>USART1, TX USART1, Tx Complete</td>
</tr>
<tr>
<td>34</td>
<td>0x0042(3)</td>
<td>TWI Two-wire Serial Interface</td>
</tr>
<tr>
<td>35</td>
<td>0x0044(3)</td>
<td>SPM READY Store Program Memory Ready</td>
</tr>
</tbody>
</table>
Interrupt Process

- When an interrupt occurs, the Global Interrupt Enable I-bit is cleared and all interrupts are disabled.
- The user software can set the I-bit to allow nested interrupts.
- The I-bit is automatically set when a Return from Interrupt instruction – RETI – is executed.
- When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served.
  - Reset interrupt is an exception.
Initialization of Interrupt Vector Table in Mega64

- Typically an interrupt vector contains a branch instruction (JMP or RJMP) that branches to the first instruction of the interrupt service routine.
- Or simply RETI (return-from-interrupt) if you don’t handle this interrupt.
Example of IVT Initialization in Mega64

.include "m64def.inc"
.cseg
.org 0x0000
rjmp RESET ; Jump to the start of Reset interrupt service routine
          ; Relative jump is used assuming RESET is not far
jmp IRQ0  ; Long jump is used assuming IRQ0 is very far away
reti     ; Return to the break point (No handling for this interrupt).
...
RESET:   ; The interrupt service routine for RESET starts here.
...
IRQ0:    ; The interrupt service routine for IRQ0 starts here.
The ATmega64 has five sources of reset:

- Power-on Reset.
  - The MCU is reset when the supply voltage is below the Power-on Reset threshold (VPOT).

- External Reset.
  - The MCU is reset when a low level is present on the RESET pin for longer than the minimum pulse length.

- Watchdog Reset.
  - The MCU is reset when the Watchdog Timer period expires and the Watchdog is enabled.
RESET in Mega64 (Cont.)

- Brown-out Reset.
  - The MCU is reset when the supply voltage VCC is below the Brown-out Reset threshold (VBOT) and the Brown-out Detector is enabled.

- JTAG AVR Reset.
  - The MCU is reset as long as there is a logic one in the Reset Register, one of the scan chains of the JTAG system.

- For each reset, there is a flag (bit) in MCU Control and State Register MCUCSR.
  - These bits are used to determine the source of the RESET interrupt.
RESET Logic in Mega64
Watchdog Timer

- A peripheral I/O device on the microcontroller.
- It is really a counter that is clocked from a separate On-chip Oscillator (1 Mhz at Vcc=5V)
- It can be controlled to produce different time intervals
  - 8 different periods determined by WDP2, WDP1 and WDP0 bits in WDTCR.
- Can be enabled or disabled by properly updating WDCE bit and WDE bit in Watchdog Timer Control Register WDTCR.
Watchdog Timer (cont.)

- Often used to detect software crash.
  - If enabled, it generates a Watchdog Reset interrupt when its period expires.
    - When its period expires, Watchdog Reset Flag WDRF in MCU Control Register MCUCSR is set.
      - This flag is used to determine if the watchdog timer has generated a RESET interrupt.
  - Program needs to reset it before its period expires by executing instruction WDR.
Watchdog Timer Diagram

Source: Atmega64 Data Sheet
Watchdog Timer Control Register

- WDTCR is used to control the scale of the watchdog timer. It is an I/O register in AVR

Source: Atmega64 Data Sheet
WDTCR Bit Definition

- **Bits 7-5**
  - Not in use. Reserved.

- **Bit 4**
  - Watchdog change enable
    - Named WDCE
      - Should be set before any changes to be made

- **Bit 3**
  - Watchdog enable
    - Named WDE
      - Set to enable watchdog; clear to disable the watchdog

- **Bits 2-0**
  - Prescaler
    - Named WDP2, WDP1, WPD0
      - Determine the watchdog time reset intervals
## Watchdog Timer Prescale

<table>
<thead>
<tr>
<th>WDP2</th>
<th>WDP1</th>
<th>WDP0</th>
<th>Number of WDT Oscillator Cycles</th>
<th>Typical Time-out at V\textsubscript{CC} = 3.0V</th>
<th>Typical Time-out at V\textsubscript{CC} = 5.0V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16K (16,384)</td>
<td>17.1 ms</td>
<td>16.3 ms</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>32K (32,768)</td>
<td>34.3 ms</td>
<td>32.5 ms</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>64K (65,536)</td>
<td>68.5 ms</td>
<td>65 ms</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>128K (131,072)</td>
<td>0.14 s</td>
<td>0.13 s</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>256K (262,144)</td>
<td>0.27 s</td>
<td>0.26 s</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>512K (524,288)</td>
<td>0.55 s</td>
<td>0.52 s</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1,024K (1,048,576)</td>
<td>1.1 s</td>
<td>1.0 s</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2,048K (2,097,152)</td>
<td>2.2 s</td>
<td>2.1 s</td>
</tr>
</tbody>
</table>

Source: Atmega64 Data Sheet
Examples

- Enable watchdog

; Write logical one to WDE

ldi r16, (1<<WDE)
out WDTCR, r16
Examples

- Disable watchdog
  - Refer to the data sheet

; Write logical one to WDCE and WDE

ldi r16, (1<<WDCE)|(1<<WDE)
out WDTCR, r16

; Turn off WDT
ldi r16, (0<<WDE)
out WDTCR, r16
Examples

- Select a prescale
  - Refer to the data sheet

```assembly
; Write logical one to WDCE and WDE
ldi r16, (1<<WDCE)|(1<<WDE)
out WDTCR, r16

; set time-out as 1 second
ldi r16, (1<<WDP2)|(1<<WDP1)
out WDTCR, r16
```
Watchdog Reset

- Syntax: \textit{wdr}
- Operands: none
- Operation: reset the watchdog timer.
- Words: 1
- Cycles: 1
Example

- The program in the next slide is not robust. May lead to a crash. Why? How to detect the crash?
; The program returns the length of a string.

.include "m64def.inc"
.def i=r15 ; store the string length when execution finishes.
.def c=r16 ; store s[i], a string character

.cseg

main:
    ldi ZL, low(s<<1)
    ldi ZH, high(s<<1)
    clr i
    lpm c, z+
loop:
    cpi c, 0
    breq endloop
    inc i
    lpm c, Z+
    rjmp loop
endloop: ...

s: .DB 'h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd'
Reading Material

- Chapter 8: Interrupts and Real-Time Events. Microcontrollers and Microcomputers by Fredrick M. Cady.
- Mega64 Data Sheet.
  - System Control and Reset.
  - Watchdog Timer.
  - Interrupts.
Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
   - Bit operations
     - sei, cli
     - sbi, cbi
   - MCU control instructions
     - wdr
1. What is the function of the following code?

```assembly
; Write logical one to WDCE and WDE
ldi r16, (1<<WDCE)|(1<<WDE)
out WDTCR, r16

; set time-out as 2.1 second
ldi r16, (1<<WDP2)|(1<<WDP1)|(1<<WDP0)
out WDTCR, r16

; enable watchdog
ldi r16, (1<<WDE)
out WDTCR, r16

loop: oneSecondDelay ; macro for one second delay
wdr
rjmp loop
```
2. How an I/O device signals the microprocessor that it needs service?
Homework

3. Why do you need software to disable interrupts (except for the non-maskable interrupts)?