ECE3411 – Fall 2016

Lecture 2c.

Interrupts & Interrupt Service Routines (ISRs)

Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

Copied from Lecture 2c, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

Based on the Atmega328P datasheet and material
from Bruce Land's video lectures at Cornel

UCONN

Interrupts

Lower range of program storage in flash:

11.4 Interrupt Vectors in ATmega328P

Table 11-6. Reset and Interrupt Vectors in ATmega328P

VectorNo.	Program Address(2)	Source	Interrupt Definition
1	0x0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete

If you want to set the mask bit of an interrupt, i.e., you enable a certain interrupt, then you ***must*** write a corresponding ISR (interrupt service routine).

The table contains the address of the ISR that you write (upon the HW event that will cause the interrupt, the program counter will jump to the address indicated by the table to execute the programmed ISR).

Program memory has 2^16 registers
→ an address has 16 bits, e.g., 0xabcd
→ 0xabcd is stored in two 8-bit registers
→ Interrupt vector table associates interrupt vectors to addresses 0x0000, 0x0002, 0x0004 etc. (by increments of 2)

Table 11-6. Reset and Interrupt Vectors in ATmega328P (Continued)

VectorNo.	Program Address(2)	Source	Interrupt Definition
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

Program Layout

- Initialization procedure:
  - Set up tables,
  - Initialize timers,
  - Do bookkeeping before you can put on interrupts
  - Turn on the master interrupt bit: This is the I-bit in register SREG, the C-macro sei() does this for you

6.3.1 SREG – AVR Status Register

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – I: Global Interrupt Enable**

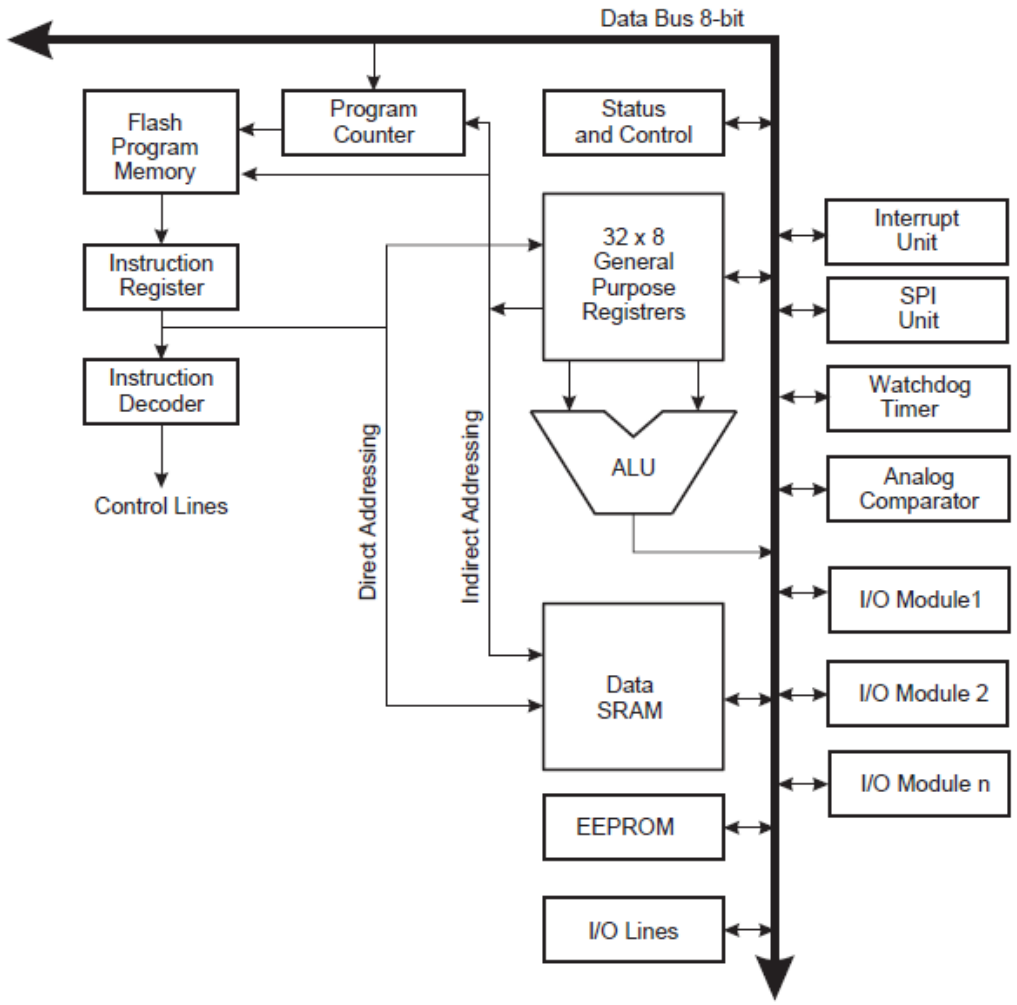
The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable Register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

Program Layout

- Main() executes slow background code forever → you never exit main in a MCU
  - Interrupt driven tasks are asynchronously called from main, for example:
    - a HW timer may cause a HW event every 1000 cycles, upon which in the corresponding ISR a SW counter is incremented;
    - upon reaching an a-priori defined maximum value, the background code calls a corresponding procedure which executes some task, and upon returning the SW counter is reset to 0
- ISRs have no parameters, no return value, they save CPU state (and C does this for you); they are called by HW events:
  - E.g., the bit value for position RXC0 in UCSR0A goes to high when receiving a character is completed
  - If the mask bit in UCSR0B for position RXCIE0 is set [meaning that an interrupt is enabled for the flag (UCSR0A & (1<<RXC0))], then the MCU will jump to the address of the ISR as indicated by the interrupt vector table for source USART, RX.
  - It takes about 75 cycles to go in and out of a ISR; another 32 cycles to safe state of the MCU (32 registers); another 7/8 cycles overhead.

AVR Architecture

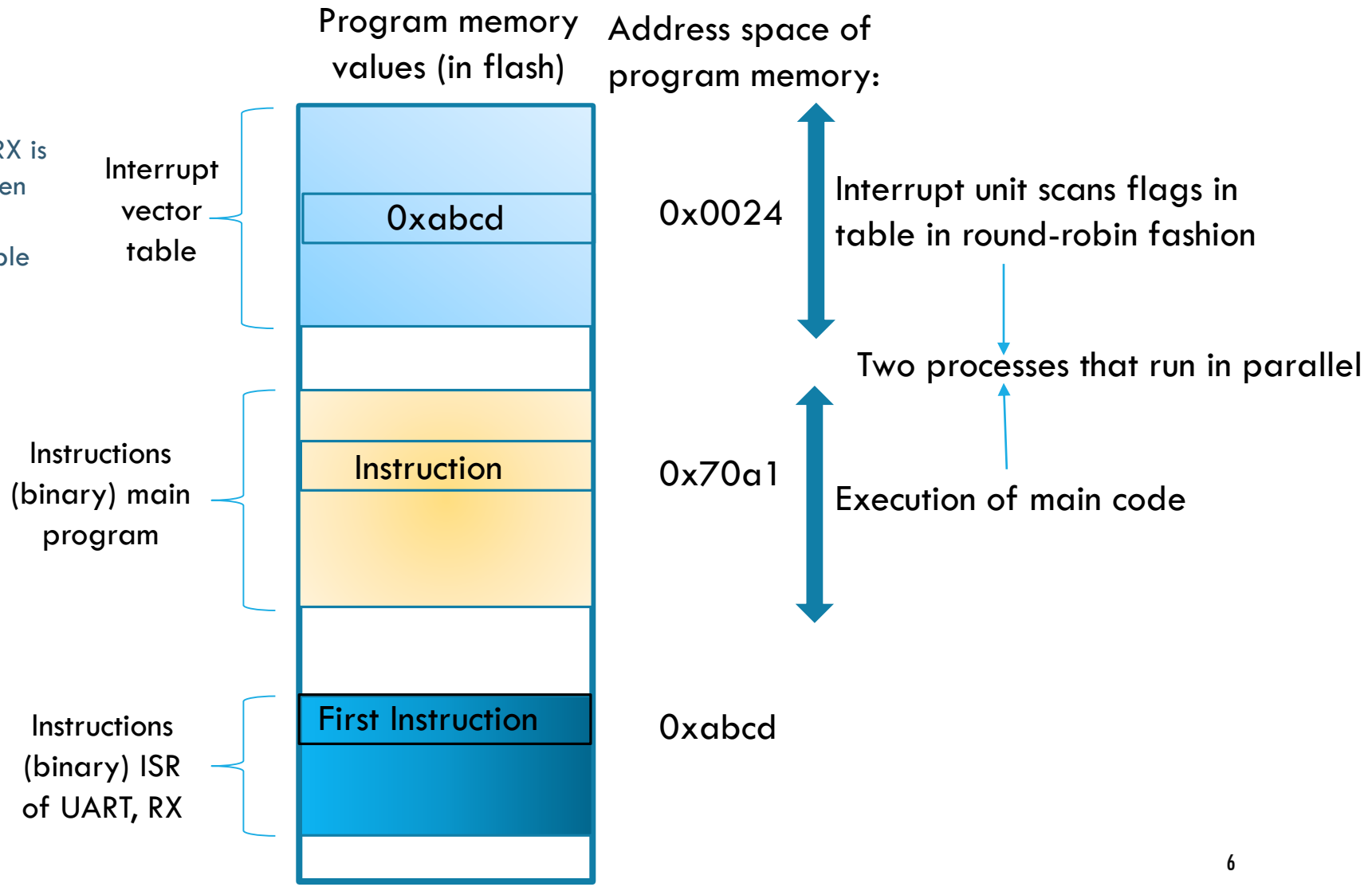
Figure 6-1. Block Diagram of the AVR Architecture



Execution of an ISR

1. Character receive complete
→ UART RX HW event which makes flag UCSR0A & (1<<RXC0) non-zero
→ If UCSR0B |= (1<<RXCIE0), i.e., ISR UART RX is enabled, then the interrupt unit sees flag when checking for the UART RX HW event
→ Interrupt unit looks at the Interrupt vector table at position 0x0024 for UART RX, and reads address 0xabcd

2. Program counter (PC) points at 0x70a1, corresponding instruction is executed to completion
→ 0x70a1 is pushed on to the PC stack
→ PC becomes 0xabcd
→ ISR UART RX is executed
→ Upon return from interrupt, the PC stack pops the value 0x70a1
→ PC gets the next address and main program continues its execution



Execution of an ISR

1. Some HW event sets a flag in some register, e.g., (UCSR0A & (1<<RXC0)) goes to high → If the corresponding interrupt is enabled, e.g., by initially programming UCSR0B |= (1<<RXCIE0), then this flag is detected by the Interrupt Unit of the MCU which scans the flags which correspond to the vector table in round robin fashion
2. If the CPU is executing an ISR, then finish the ISR, else finish current instruction
3. Push the Program Counter (PC) on the stack
4. Clear the I-bit in SREG (after this, none of the interrupts are enabled)
5. Clear flag in register UCSR0A as in UCSR0A &= ~(1<<RXC0)
6. The CPU jumps to the vector table and clears the corresponding flag
7. The CPU jumps to the ISR indicated by the address in the vector table
8. The compiler created a binary which saves state, executes your ISR code, restores state, and returns: return from interrupt (RETI)
9. RETI enables the I-bit in SREG and re-checks the interrupt flag registers in the vector table (since other HW events may have occurred in the meantime)

Problems

- Example: An ISR with print statement calls the print procedure, which buffers the characters to be printed in HW since printing is slow.
- Now, the HW executes the printing statement in parallel with the rest of the ISR.
- The ISR finishes.
- Before the print statement is finished the ISR is triggered again
- Not even a single character may be printed!!

Problems

- In your ISR you may enable the master interrupt bit → this creates a nested interrupt → not recommended

- Memory of one event deep: e.g.,
  - MCU handles a first flag of (UCSR0A & (1<<RXC0))
  - After clearing this flag, the same HW event happens again which will again set the interrupt flag vector for (UCSR0A & (1<<RXC0)) (which will be handled after the current interrupt)
  - But more interrupts for (UCSR0A & (1<<RXC0)) are forgotten while handling the current interrupt (first flag)!!
  - You need to write ***efficient*** ISR code to avoid missing HW events, which may cause your application to be unreliable.

Interrupt Service Routine (ISR)

- http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__interrupts.html
- #include <avr/interrupt.h>

USART_RXC_vect	SIG_USART_RECV, SIG_UART_RECV	USART, Rx Complete	ATmega16, ATmega32, ATmega323, ATmega8
USART_RX_vect	SIG_USART_RECV, SIG_UART_RECV	USART, Rx Complete	AT90PWM3, AT90PWM2, AT90PWM1, ATmega168P, ATmega3250, ATmega3250P, ATmega328P, ATmega3290, ATmega3290P, ATmega48P, ATmega6450, ATmega6490, ATmega8535, ATmega88P, ATmega168, ATmega48, ATmega88, ATtiny2313
USART_TXC_vect	SIG_USART_TRANS, SIG_UART_TRANS	USART, Tx Complete	ATmega16, ATmega32, ATmega323, ATmega8
USART_TX_vect	SIG_USART_TRANS, SIG_UART_TRANS	USART, Tx Complete	AT90PWM3, AT90PWM2, AT90PWM1, ATmega168P, ATmega328P, ATmega48P, ATmega8535, ATmega88P, ATmega168, ATmega48, ATmega88, ATtiny2313

- We need to program ISR(USART_RX_vect)

ISR(USART_RX_vect)

- fscanf uses:

```
int uart_getchar(FILE *stream)
{
   …
   while ( !(UCSR0A & (1<<RXC0)) ) ;
   c = UDR0;
   …
   uart_putchar(c, stream);
   …
}
```

- During the while loop other tasks need to wait → fscanf's implementation is blocking
- Need non-blocking code: write a ISR which waits until the character is there

ISR(USART_RX_vect)

```
....

// To make sure that the program does not need to wait
// we write our own get_string procedure.
// This requires a circular buffer, index, and a ready flag.
#define r_buffer_size 50
char r_buffer[r_buffer_size];
int r_index;
volatile char r_ready;

// After getstr(), the USART receive interrupt is enabled.
// The interrupt loads r_buffer, when done r_ready is set to 1.
void getstr(void)
{
  r_ready = 0; // clear ready flag
  r_index = 0; // clear buffer
  UCSR0B |= (1<<RXCIE0);
}
....
```

The volatile keyword warns the compiler that the declared variable can change at any time without warning and that the compiler shouldn't optimize it away no matter how static it seems

ISR(USART_RX_vect)

- The while loop represents task-based programming, which we repeat throughout the course: While a string is being inputted by the user, other tasks (e.g. Task2) can be executed in parallel
  - No stalling → Efficient execution
  - Modularity as a Computer System Design Principle

- The getstr() can be called in any subroutine, not only in the main while loop
- Inside getstr() r_ready and r_index are reset to 0 → Task_InterpretReadBuffer() may also call getstr() at the end of its code; Here we show the reset explicitly in the main while loop.
- It is often more natural to merge Task() and ResetCond(), especially if the reset should happen at the start of a task rather than at the end

```
int main(void)
{
  // Initializations etc.
  sei(); // Enable global interrupt
  getstr();
  etc.

  while(1)
  {
    if (r_ready == 1) {Task_InterpretReadBuffer(); getstr();}
    if Condition2 { Task2(…); ResetCond2; }
    etc.
  }

  return 0;
}
```

ISR(USART_RX_vect)

```
ISR(USART_RX_vect)
{
  char r_char = UDR0;
  // Echo character back out over the system such that a human user
  //can see this
  UDR0 = r_char;

  if (r_char != '\r')        // compare to the enter character
  {
    if (r_char == 127) // compare to the backspace character
                       // (using '\b' instead of 127 does not work!)
    {
       putchar(' ');      // erase character on screen
       putchar('\b');     // backspace
       --r_index;         // erase previously read character in r_buffer
     }
    else
    {
       r_buffer[r_index] = r_char;
       if (r_index < r_buffer_size-1) {r_index++;}
       else {r_index = 0;}
    }
  }
  else
  {
       putchar('\n');                 // new line
       r_buffer[r_index]=0;           //strings are terminated
                                      // with a 0
       r_ready = 1;
       UCSR0B ^=(1<<RXCIE0);  // turn off receive
interrupt
  }
}
```

Implements a simple line editor; we can add more line editing commands from the original uart_getchar(…)!