ECE3411

Lecture 6

USART Interrupt

Sung Yeul Park
Department of Electrical & Computer Engineering
University of Connecticut
Email: sung_yeul.park@uconn.edu

Derived from Lecture 2b, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider
Lecture 5, ECE3411 – Fall 2024, by John Chandy

Based on the AVR128DB48 datasheet and material
from Bruce Land's video lectures at Cornell

Programmed I/O

- CPU has direct control over I/O
  - Sensing status
  - Read/write commands
  - Transferring data
- CPU polls I/O module and waits for it to complete operation
- Wastes CPU time

```
unsigned char USART_Receive(void)
{
    /* wait for data to be received */
    while ( !(UCSR0A & (1<<RXC0)) ) ;

    /* get and return received data from buffer */
    return UDR0;
}
```

Interrupts

- I/O is slow – Instead of waiting for the I/O to complete, let the I/O tell the CPU when it's done
- CPU can do useful work and get interrupted when the I/O is ready
- Advantages
  - Overcomes CPU waiting
  - No repeated CPU checking of device
  - I/O module interrupts when ready

Polling vs. Interrupts

- Polling
  - Frequent/regular events as long as device can be controlled at user level.
  - Compiler knows which registers in use at polling point. Hence, do not need to save and restore registers (or not as many).
  - Other interrupt overhead avoided (pipeline flush, trap priorities, etc).
- Interrupts
  - Infrequent/Irregular events
  - Regular/predictable service of events
  - Overhead of polling instructions is incurred regardless of whether or not handler is run. This could add to inner-loop delay.
  - Device may have to wait for service for a long time.

Interrupt Control Flow

- Processor executes main code
- Interrupt happens and processor stops main code
- Processor saves main code state (program counter and registers)
- Processor jumps to interrupt service routine (ISR) corresponding to interrupt
- Upon completion of ISR, processor restores main program state and resumes main code
- It takes about 11 cycles to go in and out of an ISR; another 20-30 cycles to save state of the MCU

Interrupts

50	0x64	TCA1_CMP1 TCA1_LCMP1	Normal: Timer/Counter Type A Compare 1 Interrupt Split: Timer/Counter Type A Low Compare 1 Interrupt			X	X
51	0x66	TCA1_CMP2 TCA1_LCMP2	Normal: Timer/Counter Type A Compare 2 Interrupt Split: Timer/Counter Type A Low Compare 2 Interrupt			X	X
52	0x68	ZCD1_ZCD	Zero Cross Detector Interrupt			X	X
53	0x6A	USART3_RXC	Universal Synchronous Asynchronous Receiver and Transmitter Receive Complete Interrupt			X	X
54	0x6C	USART3_DRE	Universal Synchronous Asynchronous Receiver and Transmitter Data Register Empty Interrupt			X	X
55	0x6E	USART3_TXC	Universal Synchronous Asynchronous Receiver and Transmitter Transmit Complete Interrupt			X	X

USART Interrupts

Name	Vector Description	Conditions
RXC	Receive Complete interrupt	• There is unread data in the receive buffer (RXCIE) • Receive of Start-of-Frame detected (RXSIE) • Auto-Baud Error/ISFIF flag set (ABEIE)
DRE	Data Register Empty interrupt	The transmit buffer is empty/ready to receive new data (DREIE)
TXC	Transmit Complete interrupt	The entire frame in the transmit shift register has been shifted out and there are no new data in the transmit buffer (TXCIE)

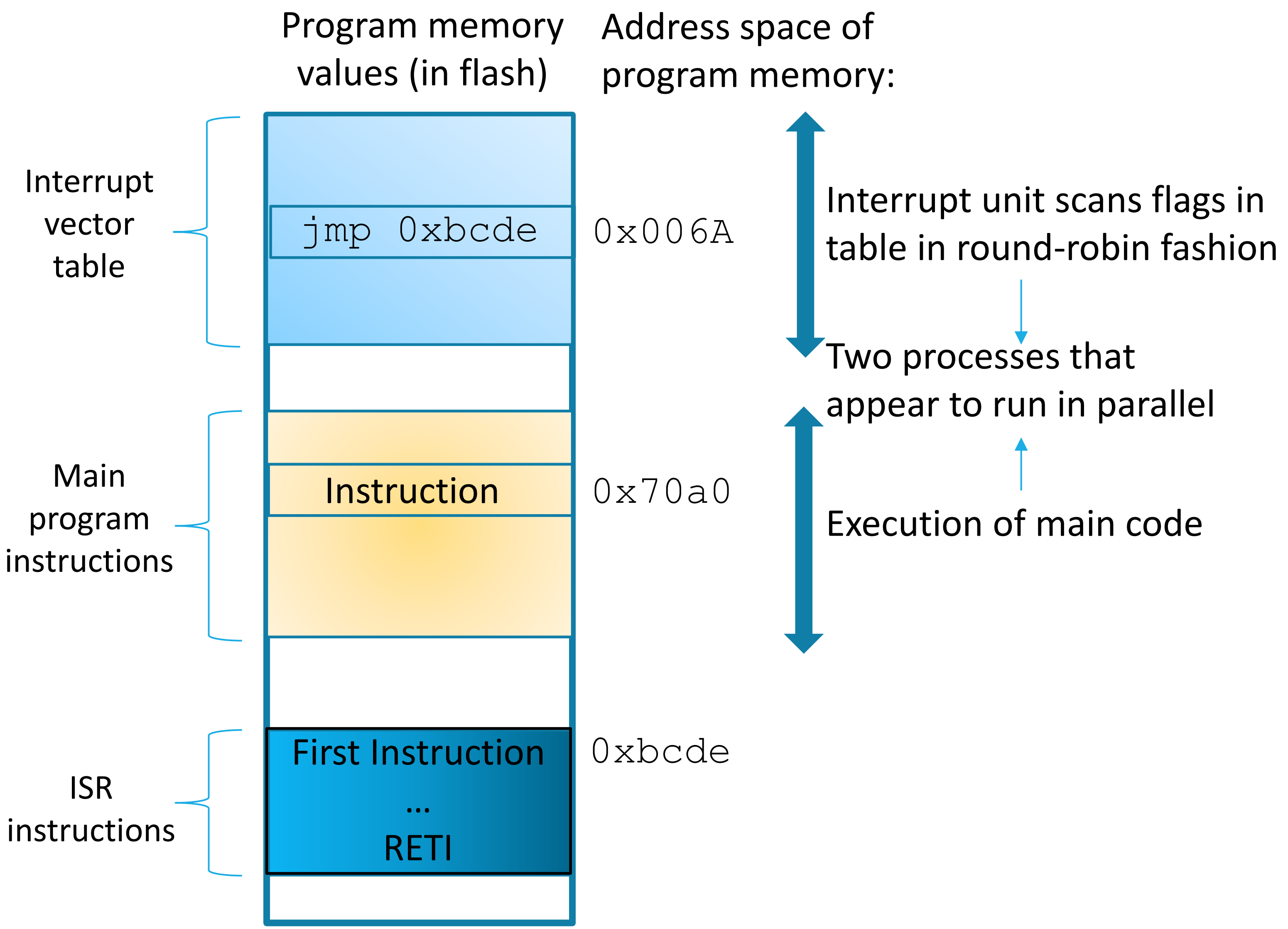
When an interrupt condition occurs, the corresponding interrupt flag is set in the STATUS (USARTn.STATUS) register.

An interrupt source is enabled or disabled by writing to the corresponding bit in the Control A (USARTn.CTRLA) register.

An interrupt request is generated when the corresponding interrupt source is enabled, and the Interrupt flag is set. The interrupt request remains active until the Interrupt flag is cleared. See the USARTn.STATUS register for details on how to clear Interrupt flags.

Execution of an ISR

1. Character receive complete → UART RX HW event which makes flag `UCSR3A & (1<<RXC)` non-zero
2. If USART3.CTRLA `|= (1<<RXCIE)`, i.e., ISR UART RX is enabled, then the interrupt unit sees flag when checking for the UART RX HW event
3. Program counter (PC) points at `0x70a0`, corresponding instruction is executed to completion
4. `0x70a0` is pushed on to the PC stack
5. Clear I-bit in SREG (disable all interrupts)
6. CPU jumps to the ISR indicated by the address at the Interrupt vector table at position `0x006A` for USART RXC
7. ISR USART RXC is executed
8. RETI instruction at end of ISR resets the I-bit in SREG, checks if any other interrupts are ready, and if not, the PC stack pops the value `0x70a0`
9. PC gets the next address and main program continues its execution



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 `RXC0`
  - After clearing this flag, the same HW event happens again which will again set the interrupt flag vector for `RXC0` (which will be handled after the current interrupt)
  - But more interrupts for `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.

USART CTRLA

Bit	7	6	5	4	3	2	1	0
	RXCIE	TXCIE	DREIE	RXSIE	LBME	ABEIE		RS485
Access	R/W	R/W	R/W	R/W	R/W	R/W		R/W
Reset	0	0	0	0	0	0		0

- RXCIE: Receive character complete interrupt enable
- TXCIE: Enables interrupt for both members in TX queue being empty
- DREIE: Enables interrupt if the first of the output pipeline is empty. Ready to transmit.
- RXSIE: Receive start frame interrupt enable
- LBME: Loop-back mode enable
- ABEIE: Auto-baud error interrupt enable
- RS485: Enable RS-485 mode

```
// Enable interrupts for RX complete and data register empty
USART3.CTRLA = USART_RXCIE_bm | USART_DREIE_bm;
```

Program Layout

```
#include <avr/interrupt.h>
int main()
{
    USART3.CTRLA |= USART_DREIE_bm; // enable UART TX interrupt
    sei();
    while (1) {
        ...
    }
}

ISR(USART3_DRE_vect)
{
    ...
}
```

Problems

- Long-running ISRs
  - Example: An ISR calls the `printf` function. This can take a long time (milliseconds), and no interrupts can be processed during this time.
  - As solution, buffer the characters to be printed, so that ISR can complete quickly.
  - However, if interrupts happen frequently, you may overflow the buffer and characters may get dropped.

Problems

- On the AVRDX series, interrupts are not turned off when you enter the ISR
  - That could create a nested interrupt → not recommended
- Memory of one event deep: e.g.,
  - MCU handles a first flag of `RXCIE`
  - After clearing this flag, the same HW event happens again which will again set the interrupt flag vector for `RXCIE` (which will be handled after the current interrupt)
  - But more interrupts for `RXCIE` 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.

Polling Method

- `scanf` uses:

```
int usart_receive_data(void* ptr)
{
   USART_t* usart = (USART_t*)ptr;
   while( !(usart->STATUS & USART_RXCIF_bm) );
   uint8_t c = usart->RXDATAL;
   return c;
}
```

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

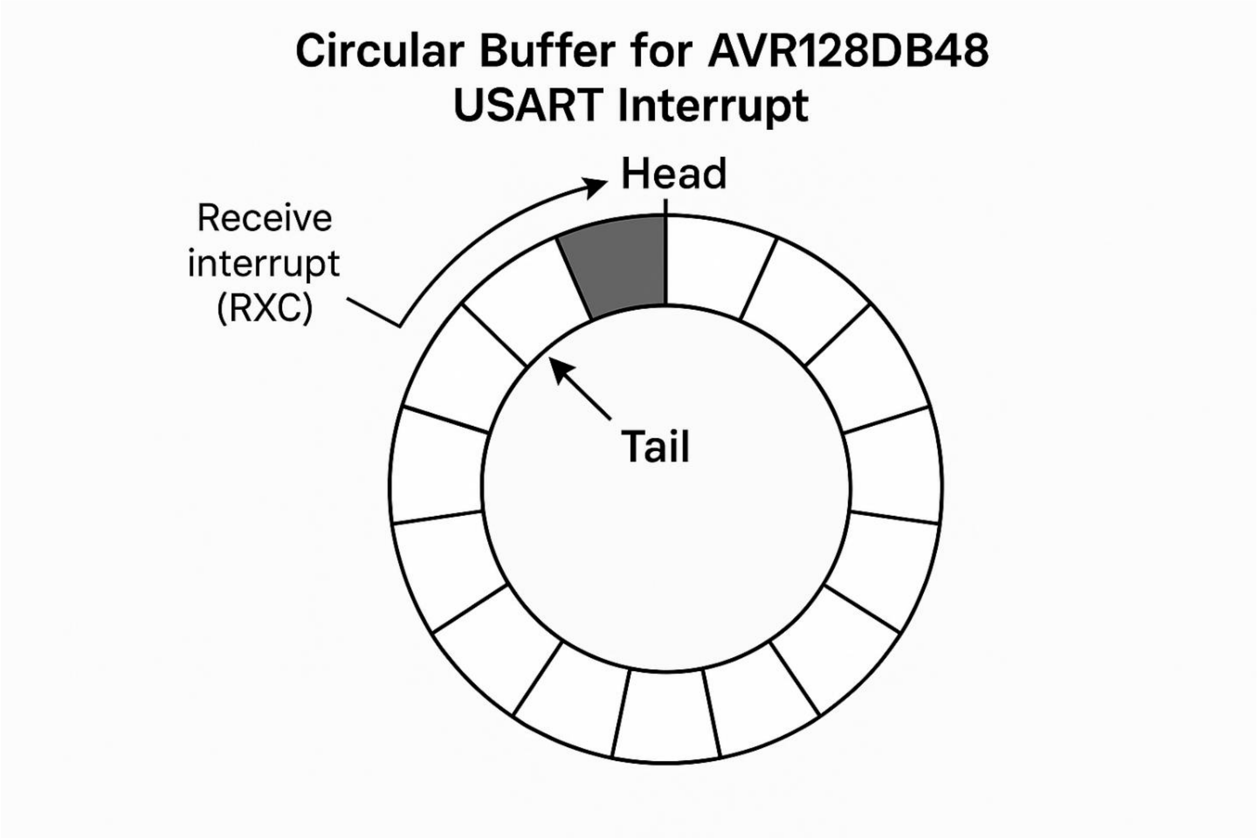
ISR(USART3_RXC_vect)

```
#include <avr/interrupt.h>

ISR(USART3_RXC_vect)
{
   c = USART3.RXDATAL;
   …
   putchar(c);
   …
}
```

- Use an ISR to receive the character as soon as it arrives.

Circular Buffer for USART Interrupt



A circular buffer (also called a ring buffer) is essential for handling USART interrupts on the AVR128DB48 microcontroller. It provides a smooth way to buffer incoming and outgoing data without blocking your main program execution.

A circular buffer uses a fixed-size array with read and write pointers that wrap around when they reach the end. This creates a continuous "circular" data flow, preventing buffer overflow and allowing efficient interrupt-driven communication.

This implementation provides robust, interrupt-driven USART communication that won't block your main program execution, making it ideal for real-time applications on the AVR128DB48.

Sample Code

```c
#define F_CPU 4000000UL  // 4MHz default
clock

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdbool.h>

#define BAUD_RATE 9600
#define BUFFER_SIZE 64

// Circular buffer structure
typedef struct {
char buffer[BUFFER_SIZE];
volatile uint8_t head;
volatile uint8_t tail;
volatile uint8_t count;
} circular_buffer_t;

// Global buffers
volatile circular_buffer_t tx_buffer = {0};
volatile circular_buffer_t rx_buffer = {0};

void buffer_put(volatile circular_buffer_t*
buf, char data) {
  if (buf->count < BUFFER_SIZE) {
buf->buffer[buf->head] = data;
buf->head = (buf->head + 1) % BUFFER_SIZE;
buf->count++;
}
}

bool buffer_get(volatile circular_buffer_t*
buf, char* data) {
if (buf->count > 0) {
*data = buf->buffer[buf->tail];
buf->tail = (buf->tail + 1) % BUFFER_SIZE;
buf->count--;
return true;
}
return false;
}

void USART3_Init(uint32_t baud) {
// Set baud rate to 9600
//USART3.BAUD = (F_CPU * 64) / (16 * 9600);

uint16_t baud_setting = (F_CPU * 64) / (16
* baud);

// Set baud rate
USART3.BAUD = baud_setting;

// Enable interrupts for RX complete and
data register empty
USART3.CTRLA = USART_RXCIE_bm |
USART_DREIE_bm;

// Set frame format: 8N1
USART3.CTRLC = USART_CHSIZE_8BIT_gc;

// For USART3 (already configured on
Curiosity Nano)
PORTB.DIRSET = PIN0_bm;  // PB0 as output
(TX)
PORTB.DIRCLR = PIN1_bm;  // PB1 as input
(RX)

// Enable transmitter and receiver
USART3.CTRLB = USART_TXEN_bm |
USART_RXEN_bm;
}

// RX Complete Interrupt
ISR(USART3_RXC_vect) {
char received_data = USART3.RXDATAL;
buffer_put(&rx_buffer, received_data);
}
```

Sample Code

```
// Data Register Empty Interrupt (TX)
ISR(USART3_DRE_vect) {
char data_to_send;

if (buffer_get(&tx_buffer, &data_to_send))
{
USART3.TXDATAL = data_to_send;
} else {
// Disable DRE interrupt when buffer is
empty
USART3.CTRLA &= ~USART_DREIE_bm;
}
}

void USART3_SendChar(char data) {
buffer_put(&tx_buffer, data);

// Enable DRE interrupt to start
transmission
USART3.CTRLA |= USART_DREIE_bm;
}

void USART3_SendString(const char* str) {
while (*str) {
USART3_SendChar(*str);
str++;
}
}

bool USART3_ReceiveChar(char* data) {
return buffer_get(&rx_buffer, data);
}

int main(void) {

PORTB.DIRSET = PIN3_bm;  // PB3 as Output

USART3_Init(BAUD_RATE);
sei();  // Enable global interrupts

USART3_SendString("Interrupt-driven USART
Demo\r\n");
USART3_SendString("Commands: LED_ON,
LED_OFF, STATUS\r\n");

char command_buffer[32];
uint8_t cmd_index = 0;

while(1) {
char received_char;

if (USART3_ReceiveChar(&received_char)) {
if (received_char == '\r' || received_char
== '\n') {
command_buffer[cmd_index] = '\0';

// Process commands
if (strcmp(command_buffer, "LED_ON") == 0)
{
PORTB.OUTCLR = PIN3_bm; // Turn on LED
USART3_SendString("LED turned ON\r\n");
}
else if (strcmp(command_buffer, "LED_OFF")
== 0) {
PORTB.OUTSET = PIN3_bm; // Turn off LED
USART3_SendString("LED turned OFF\r\n");
}
else if (strcmp(command_buffer, "STATUS")
== 0) {
USART3_SendString("System Status: OK\r\n");
}
else {
USART3_SendString("Unknown command\r\n");
}

cmd_index = 0;
USART3_SendString("> ");
}
else if (cmd_index < sizeof(command_buffer)
- 1) {
command_buffer[cmd_index++] =
received_char;
USART3_SendChar(received_char);  // Echo
character
}
}
}

return 0;
}
```

Lab practice#5: Changing Frequency and Position using USART Interrupts

- With last lab, the board stops blinking while waiting for input from the user
- Instead of checking for user input in the main while loop, use an interrupt on RXCI. You can not use `scanf` in the ISR because it is a blocking function.
- Behavior should be the same as last lab, except you no longer wait for 5 seconds to ask whether to change frequency or position.
- As soon as the number digit is entered, the frequency or position should change immediately – no need to press return/enter
- The blinking should continue while waiting for input
- When a new frequency or position has been entered, ask again whether frequency or position needs to be changed

Tips

- You will need to print the "`Do you want to change the frequency or position? (F/P)`" at the beginning and also whenever the user selects a valid frequency or position.
- Make sure you enable the USART receive interrupt. Check the `CTRLA` register to find the correct bit to set. Also make sure you call `sei()` in order to enable global interrupts.
- The USART Receive ISR will get the character from the RXDATAL register and save that to a variable that your main loop will check.

Tips

- The ISR or your main function will need to print out the character that the user input – otherwise, you won’t see what you typed in.
- Depending what you are waiting for, you will ask for the frequency/position or change the frequency/position values. Use a mode flag with three possible states – waiting for F/P, waiting for frequency value, or waiting for position value.
- Since you need to print `Frequency:` when the user selects F or `Position:` when the user selects P, you can either print directly in the ISR or set a flag that the main function checks to print the prompt.