Exercise 3: Interrupts

1 Interrupt-based UART communication

The polling-based implementation UART communication from the last session is actually quite inefficient, especially when there are many tasks to perform simultaneously. Recall the button counter application in the first session. Imagine that when you press the button, the microcontroller is busy doing something else (e.g., UART communication) and it simply does not have a chance to check whether the button is pressed. Then your program will miss the event. In this section, we will learn another yet better way to do this job, namely to use *interrupts*.

Introduction to Interrupts

Figure 1 conceptually shows how an interrupt works: The bars represent the control flow. The top one is the main program and the bottom bar is the *Interrupt Service Routine* (ISR). When an interrupt occurs, the main program stops executing and the microcontroller begins to execute the ISR. Once the ISR is complete, the microcontroller returns to processing the main program where it left off. In this way, we can be assured that high-priority input is never missed. Also, using interrupts is a very efficient approach. Many embedded systems are called interrupt driven systems, because most of the processing occurs in ISRs, and the embedded system spends most of its time in a low-power mode.

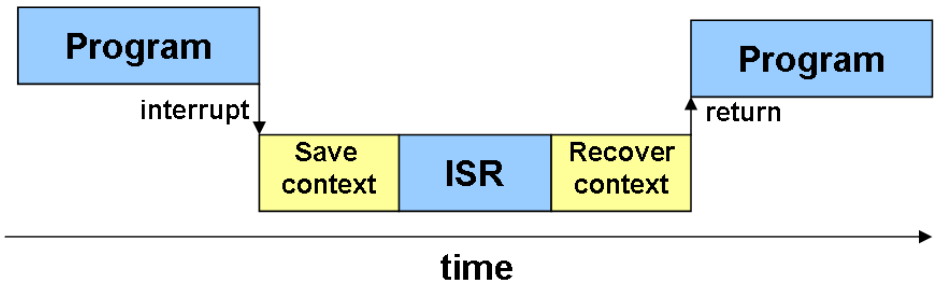

Figure 1: Serving an Interrupt

Note that before executing the ISR, there is a special piece of code which saves the *context* the program. The context typically includes current register values, the stack pointer, the return address and so on. After serving the interrupt, exactly the same context must be recovered in order to guarantee the correctness of original program. Usually, we don't bother to manually write the code to save/recover the context, since most compilers can generate those instructions for us. Nevertheless, some of these instructions are not necessary for normal function calls and the compiler will never generate them unless told. Hence, we need to add a special attribute to the ISR function so that the compiler knows about our intentions. The syntax for the GCC compiler that we are using is the following:

```
ISR(USART_RX_vect) {
  /* Implementation */
}
```

This statements defines a ISR called `USART_RX_vect`. The `ISR` macro is defined in `%WINAVR%\avr\include\avr\interrupts.h`. You can have a look at it if you are interested. Note that an ISR should not take any parameters and should not have a return value.

Usually, there are many entities in the microcontroller that can generate an interrupt, for example the UART, timers and so on. Upon detecting an interrupt, the microcontroller will first check the interrupt status register to find which type of interrupt it is and then invoke the corresponding service routine. The *Interrupt Vector Table* is a list of every interrupt service routine. It is located at a fixed location in program memory. Table 1 lists the interrupt vector table of ATmega8515.

Vector No.	Program Address	Source	Interrupt Definition
1	$000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog Reset
2	$001	INT0	External Interrupt Request 0
3	$002	INT1	External Interrupt Request 1
4	$003	TIMER1 CAPT	Timer/Counter1 Capture Event
5	$004	TIMER1 COMPA	Timer/Counter1 Compare Match A
6	$005	TIMER1 COMPB	Timer/Counter1 Compare Match B
7	$006	TIMER1 OVF	Timer/Counter1 Overflow
8	$007	TIMER0 OVF	Timer/Counter0 Overflow
9	$008	SPI, STC	Serial Transfer Complete
10	$009	USART, RXC	USART, Rx Complete
11	$00A	USART, UDRE	USART Data Register Empty
12	$00B	USART, TXC	USART, Tx Complete
13	$00C	ANA_COMP	Analog Comparator
14	$00D	INT2	External Interrupt Request 2
15	$00E	TIMER0 COMP	Timer/Counter0 Compare Match
16	$00F	EE_RDY	EEPROM Ready
17	$010	SPM_RDY	Store Program memory Ready

Table 1: Interrupt Vector Table of ATmega8515

Exercise 3.1

a) Why is there in general the need to save the program context before entering an ISR? Give an example why it is needed. Can you imagine why the context is not automatically saved and restored by the microcontroller?

b) What would happen on ATmega8515 if an interrupt occurred for which no service routine is specified in the code?

c) Which functions provided in the *WinAVR* standard library can be used to enable/disable interrupts globally?

d) How can we check whether interrupts are globally enabled? Write a function that will return whether interrupts are currently enabled. The signature of the function should be:

```
bool ien(void);
```

Note that in the interrupt vector table, the name of the service routines are already defined (`%WINAVR%\avr\include\avr\iom8515.h`). To implement an ISR, you should always use exactly the same name. For example, to implement a UART receiver ISR, use the name `USART_RX_vect`. Some important ISRs are listed in table 2.

Functionality	Name
UART, Receive Complete	`USART_RX_vect`
UART Data Register	`USART_UDRE_vect`
UART, Transmit Complete	`USART_TX_vect`
External Interrupt Request 0	`INT0_vect`

Table 2: Some Interrupt Function Names of ATmega8515

UART Interrupt

Exercise 3.2

a) How can we generate a transmitter interrupt (`UDRE` and `TXC`)? And how can we clear it?

b) How can we generate an `RXC` interrupt? And how can we clear it?

c) Develop an application with UART receiver interrupt enabled. Write a simple ISR for the `RXC` interrupt. In the ISR, set the state of the LEDs to show the character code of the received character like you did last time in polling mode. Use the terminal to send some data to the device and verify the functionality.

d) Develop the same *echo* application as in the previous exercises. Use interrupts to implement the functionality this time. To show that only interrupts are used, extend your program so the LEDs produce a "running light" with a fixed speed that is independent of the communication.

e) Verify your answer to the question in exercise 3.1 b) by omitting the receive interrupt and sending a character from the host. What happens to the "running light"?

Hints

- Make sure that the interrupt flag is cleared inside the ISR, otherwise an interrupt will occur once the ISR terminates and our program will spend most of its time in an endless ISR loop (extra question: why "*most* of its time" and not "*all* the time"?).

2 External Interrupts

Exercise 3.3

a) In the first session we have developed an application to increment a counter when a button is pressed. As we discussed, that implementation is not very "accurate". Now try to make it better by generating an *external interrupt* when a button is pressed. Note that the pin used for external interrupt 0 is `PD2`. Let your program have a global variable that holds the number of button presses and let the ISR only modify that variable. In the main program, an endless loop should apply the current value of that variable to the LEDs.

b) Implement the keyboard-like behavior when the button is continuously pressed.

Hints

- Remember what we said about the `volatile` keyword, which prevents the compiler from applying optimizations to your code. Interrupts are spontaneous events, so you might need to use the keyword in the context of ISRs as well!