Exercise 3: Interrupts

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 from 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 interrupts work. The bars represent the control flow. The top bar corresponds to the main program and the bottom bar to 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. Using this mechanism, we can drastically reduce the probability to miss important events.

Furthermore, using interrupts is a very efficient approach. Some 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 from which it is only awakened by interrupt requests.

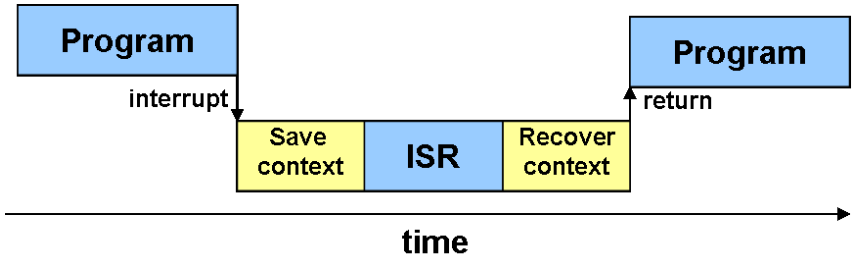

Figure 1: Serving an Interrupt

Note that before executing the ISR, there is a special piece of code that saves the *context* the program. The context typically includes current register values, the stack pointer and the address of the next instruction to execute (*program counter*). After serving the interrupt, exactly the same context must be recovered in order to guarantee the correctness of the main program. Usually, we will not manually write the code to save and 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:

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

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

This statements defines a ISR called `USART_RX_vect`. Note that an ISR does not take any parameters and does not have a return value.

Usually, there are many subsystems in the microcontroller that can generate an interrupt, for example communication channels like the UART, internal sources like timers or external sources

like I/O pins. 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 shows the interrupt vector table of ATmega168. You can also find this information in section 11.4 of the ATmega168 manual. Exactly the names in the "ISR name" column **have to be used** for ISR functions handling the respective interrupts.

No.	Address	ISR Name	Interrupt Definition
1	0x0000	(none)	Power-on, Brown-out and Watchdog System Reset
2	0x0002	INT0_vect	External Interrupt Request 0
3	0x0004	INT1_vect	External Interrupt Request 1
4	0x0006	PCINT0_vect	Pin Change Interrupt Request 0
5	0x0008	PCINT1_vect	Pin Change Interrupt Request 1
6	0x000A	PCINT2_vect	Pin Change Interrupt Request 2
7	0x000C	WDT_vect	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA_vect	COMPA Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB_vect	COMPB Timer/Counter2 Compare Match B
10	0x0012	TIMER2_OVF_vect	OVF Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT_vect	CAPT Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA_vect	COMPA Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB_vect	COMPB Timer/Coutner1 Compare Match B
14	0x001A	TIMER1_OVF_vect	OVF Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA_vect	COMPA Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB_vect	COMPB Timer/Counter0 Compare Match B
17	0x0020	TIMER0_OVF_vect	OVF Timer/Counter0 Overflow
18	0x0022	SPI_STC_vect	SPI Serial Transfer Complete
19	0x0024	USART_RX_vect	USART Rx Complete
20	0x0026	USART_UDRE_vect	USART Data Register Empty
21	0x0028	USART_TX_vect	USART Tx Complete
22	0x002A	ADC_vect	ADC Conversion Complete
23	0x002C	EE_READY_vect	EEPROM Ready
24	0x002E	ANALOG_COMP_vect	Analog Comparator
25	0x0030	TWI_vect	2-wire Serial Interface
26	0x0032	SPM_READY_vect	Store Program Memory Ready

Table 1: Interrupt Vector Table of ATmega168

Exercise 3.1

a) Why is there in general the need to save the program context before entering an ISR? Give a concrete example for a case when it is needed. Can you imagine why the context is not automatically saved and restored by the microcontroller (e.g., "in hardware")?

b) Which functions from the *WinAVR* library can be used to enable/disable interrupts globally?

c) How can be checked whether interrupts are globally enabled? Write a function with the following signature that returns 1 or 0 depending on whether interrupts are enabled or not:

```
char ien(void);
```

UART Interrupts

Read the ATmega168 manual about UART transmit and receive complete interrupts to find the answers to the following questions.

Exercise 3.2

a) How can transmitter interrupts be enabled (transmit complete and data register empty)?

b) How can receive complete interrupts be enabled? When is the `RXC0` flag cleared?

c) Develop an application with UART receive complete interrupt enabled. 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, rather slow speed that is independent of the communication.

e) Try to find out what happens if an interrupt occurs for which no service routine is specified. Do this by commenting out the receive interrupt's ISR and sending a character from the host. What happens to the "running light"? If the effect does not match your expectations, try to find an explanation.

Hints

- Make sure interrupts are enabled globally at the correct points in time ;)
- Some interrupts require the interrupt flag to be manually cleared by the ISR, otherwise a subsequent interrupt will occur once the ISR terminates and the program will spend most of the time in an ISR loop. Read section 19.6.3 of the ATmega168 manual for more information.

  By the way, why does the above paragraph read "*most of* the time" and not "*all* the time"?

External Interrupts

In the first session we have developed an application to increment a counter when a button is pressed. As we discussed, that implementation might be vulnerable to bouncing effects and might suffer from timing issues. We will now improve the program by generating an *pin change interrupt* when the button is pressed.

Exercise 3.3

a) Read chapter 12 of the ATmega168 manual to learn about external interrupts. Implement the button press counter from a previous exercise using pin change interrupts (leave the switches connected to port 'C'). 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 (meaning that when continuously pressing the button, the character gets sent multiple times after a short period until the button is released).

Hints

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

Sleep Modes and Interrupt Driven Design

Atmega168 can be switched to different power saving modes during idle times. This is of special importance in scenarios where the controller is battery-powered.

Exercise 3.4

a) Read chapter 9 of the ATmega168 manual to learn about power management. Which factors determine how much energy the microcontroller uses? Also consider factors that arise due to inefficient programs and algorithms and give examples for each case.

b) Which sleep modes and register setup can be used for each of the following scenarios when only the specified functionality must be guaranteed? For each case, specify the configuration that saves as much energy as possible.

i) Handling of pin change interrupts within very short amount of time (only a few clock cycles until interrupt handling begins), given an external clock is used.

ii) Handling of pin change interrupts with a maximum delay of a few milliseconds.

iii) Handling of UART events.

iv) Handling of timer events (for each timer individually).

c) Implement an echo application on the serial port that is as energy-efficient as possible. This means that the MCU should switch to a well-suited power saving mode as often and quickly as possible and the number of instructions executed should be minimized. List the energy-saving measures your approach considers to implement the desired behavior.