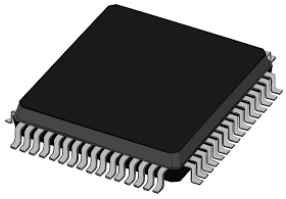
Computer Organization

Lab and Exercise 4



EITF70
EITA15

Polling and Interrupts

Goals

- Understand when and how to use polling
- Understand when and how to use interrupts
-
-

Christoffer Cederberg
Jonathan Sönnerup
2020

Contents

Introduction

Lab Equipment

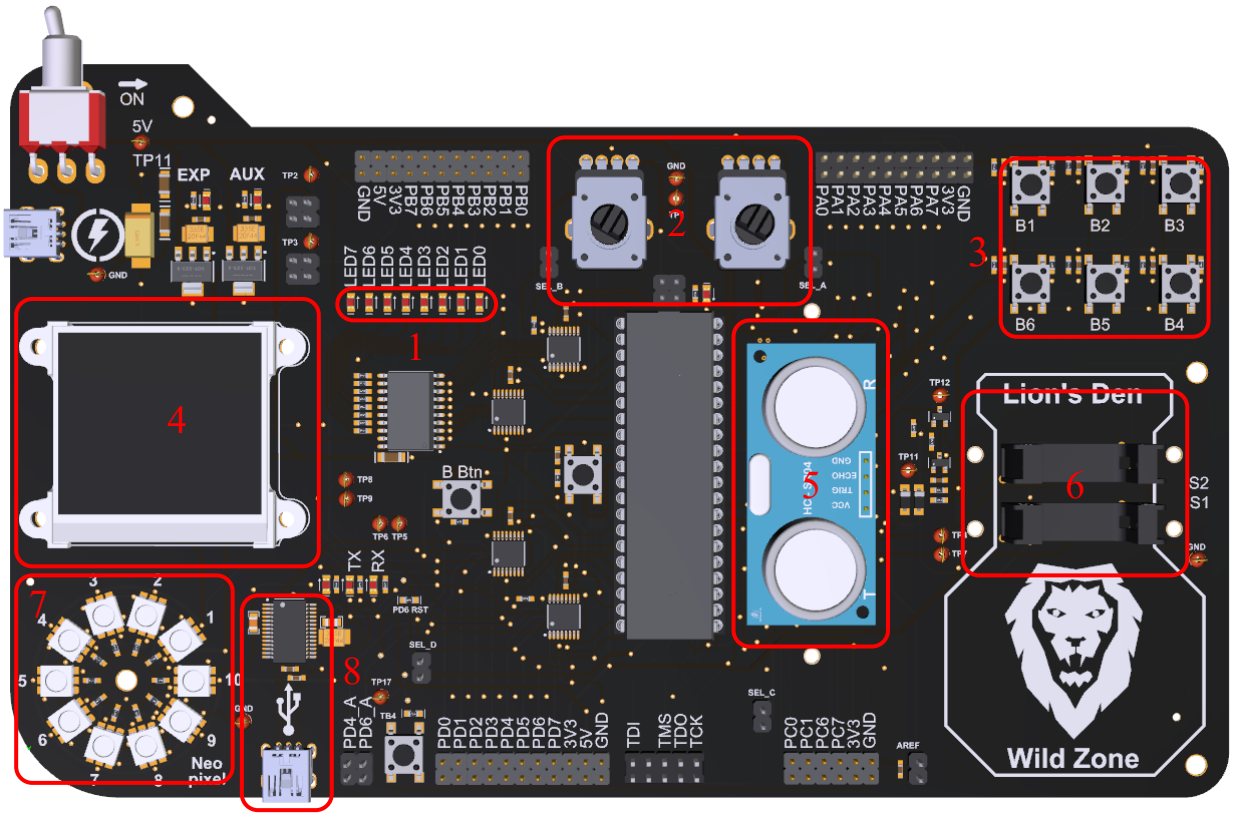


Figure 1: A picture of the lab PCB and the external components that will be used during the lab.

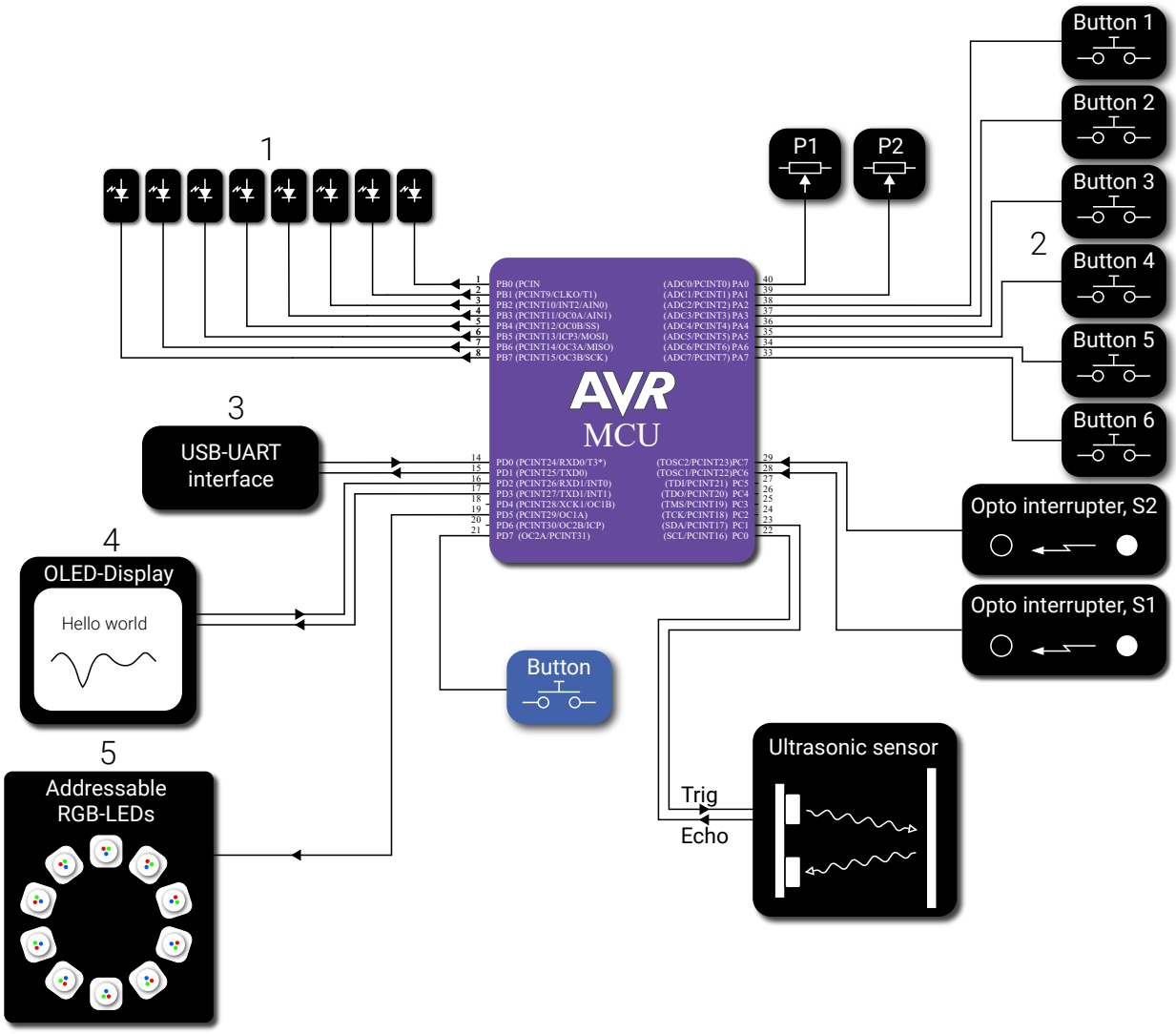


Figure 2: Block schematic of the circuit board with the used IO blocks coloured black.

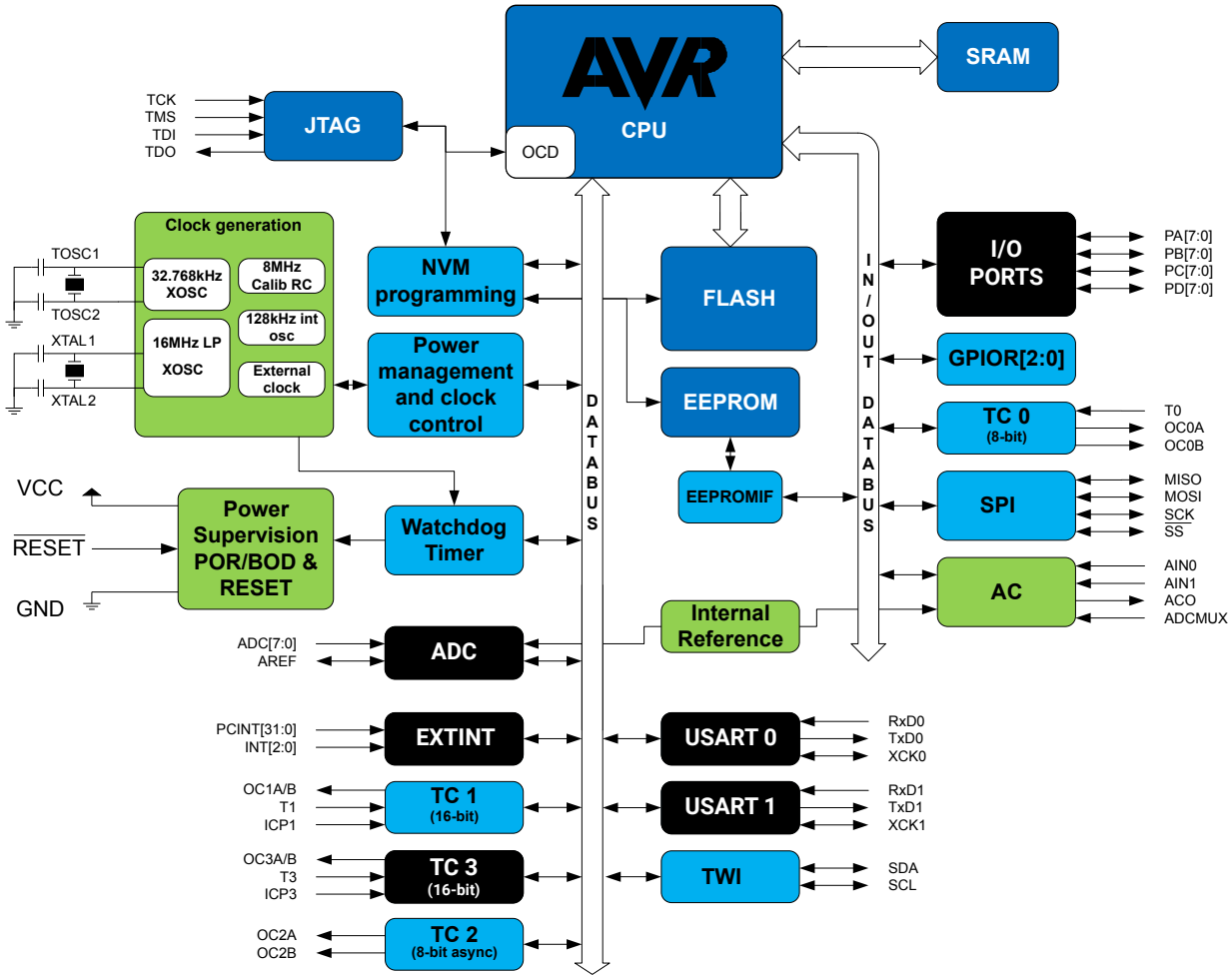


Figure 3: Block diagram for the ATmega1284P with the internal I/O units that will be used during this lab coloured black.

Chapter 1

Polling

Polling is the process where the computer or controlling device waits for an external device to check for its readiness or state, often with low-level hardware. For example, when a printer is connected via a parallel port, the computer waits until the printer has received the next character. These processes can be as minute as only reading one bit. This is sometimes used synonymously with busy-wait polling. In this situation, when an I/O operation is required, the computer does nothing other than check the status of the I/O device until it is ready, at which point the device is accessed. In other words, the computer waits until the device is ready. Polling also refers to the situation where a device is repeatedly checked for readiness, and if it is not, the computer returns to a different task. Although not as wasteful of CPU cycles as busy waiting, this is generally not as efficient as the alternative to polling, interrupt-driven I/O.

In a simple single-purpose system, even busy-wait is perfectly appropriate if no action is possible until the I/O access, but more often than not this was traditionally a consequence of simple hardware or non-multitasking operating systems.

Polling is often intimately involved with very low-level hardware. For example, polling a parallel printer port to check whether it is ready for another character involves examining as little as one bit of a byte. That bit represents, at the time of reading, whether a single wire in the printer cable is at low or high voltage. The I/O instruction that reads this byte directly transfers the voltage state of eight real world wires to the eight circuits (flip flops) that make up one byte of a CPU register.

Polling has the disadvantage that if there are too many devices to check, the time required to poll them can exceed the time available to service the I/O device. - Wikipedia

1.1 Lab Exercises

1.1.1 The Lions Den

Rustin Cohle has grown tired of fighting criminals and started to manage an amusement park close to Fort Macomb, New Orleans. The biggest attraction is the "Lions Den". It consists of two regions, one smaller cage, where the lions eat and sleep, and one bigger fenced area, where the lions can roam around freely, i.e., a wild zone, see Figure 1. The wild zone is hilly, thus it is difficult to see if there are any lions or not. This poses a problem since the caretakers want to clean the two regions twice a week. One of the co-workers has proposed that a security systems should be installed. After some time, Rust decided to place two sensors in the passage between the den and the wild zone. The two sensors, labeled S1 and S2, are separated by a suitable distance so that when a lion goes from one region to another it will always, at some point, cover both sensors. Furthermore only one lion fits in the hallway and for some strange reason they cannot walk backwards.

When an obstacle is placed between the two ends of a sensor, the output will be high, otherwise it will be low. The sensor labeled with S1 is connected to port C and pin 6 and the other, to pin 7 on the same port.

For a complete passage from the den to the wild zone the sensor signals will be as following:

$$(\mathrm{S2}, \mathrm{S1}) : 00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00 \tag{1.1}$$

For a complete passage from the wild zone to the den the sensor signals will be as following:

$$(\mathrm{S2}, \mathrm{S1}) : 00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00 \tag{1.2}$$

Rust has asked you write an application that counts the number of lions that are in the wild zone. As the go-getter you are you gladly sign up to program the AVR that controls the system. Rust gives you more or less total freedom when it comes the structure of the application. The only requirement is that the sensors **should not** constantly be polled in a while-loop, which only breaks when a lion walks through the hallway. Doing so should be done with caution since it will lock the processor, i.e., other functions will never or rarely be called. For an example, see Listing 1.1 below.

```c
#include "true_den.h"

//  some variables

int main(void) {

    // some initialization code

    while (1) {

        your_lion_code();

        func_to_be_added(); // Might never run or very seldomly
    }
}

void your_lion_code() {

    while(!(S1 || S2));      // Loop that only breaks when either senor 1 or sensor 2 becomes
    equal to 1.
    .
    // Other code
}
```

Listing 1.1: An example of bad practice whe it come to polling.

Tasks:

- Create a new project in Atmel Studio.

> **Home Assignment 1.1**
> How do you initialize the used I/O-pins as inputs? The pins can be found on port C, use the schematic in Figure 2.

- Create a function that initializes the used I/O-pins using the answer from the above home assignment.

> **Home Assignment 1.2**
> Write code that polls the opto interrupters and counts the number of lions that are out in the wild zone.

– Remember to initialize the I/O-pins before using them.
– Do not forget to mask the input.
– You may have to buffer a certain amount of previous sensor states.
– Also, make sure that you only sample the sensor if the value has changed.

- Create a function using the code from the above home assignment.

Lab Question 1.1
Is it necessary to wait for the whole sequence that is generated when a lion passes through the hallway?

Lab Question 1.2
How fast can an obstacle move through the sensor array without it being missed?

Lab Question 1.3
OPTIONAL: Suddenly the lions have gained the ability to walk backwards. Do the necessary changes to your code!

1.1.2 High-tech Lion Cage

Rustin Cohle was very pleased with your work and has requested your assistance again. After they installed the system you designed, they wanted to add some functionality. One of the technicians had some good ideas, which resulted in that he added the following features:

- It is now possible to see how many of the lions that are in the wild zone by using a terminal, such as YAT. The baud rate is 500000 baud, no parity and one stop bit.
- Inside the den, there are two light sources which intensity is modulated by a PWM signal. The intensity is displayed when a specific button is pressed.
- The lions are feed through a small hatch. To ensure that there are no lions close by, the technician installed an ultra sonic sensor. If a lion is closer than 15 cm a warning is displayed in the terminal when a button is pressed.

The worried Rust did not tell what was wrong, he only said that he trusted you to figure it out.

The added features, which are neatly packed in two functions `security_system_init()` and `security_system_run()`, can be found in the library, `true_den`, on the following path `S:\Courses\eit\EITA15\lab4`. They are used to interfaced the I/O units on the lab PCB. The potentiometers, `P1` an `P2`, controls the intensity of `LED7` and `LED6`. Button 1 is used to check if it is safe to open the hatch and button 2 to display the light settings.

Tasks:

- Create a new project in Atmel Studio.
- Add the `true_den` library.

– If you don't remeber how to add the library, watch this video from lab 1: `https://youtu.be/e00Ef4CDI9U`.
– Video on how to configure YAT: `https://youtu.be/-L1S_aBHdf0`. IMPORTANT! It differs from lab 2.

- Copy the code from the previous assignment.
- Call the function `security_system_init()` after the initialization of the lion cage (and before the while-loop) and add the function `security_system_run()` in the while-loop. To send the number of lions that are in the wild zone use the function `send_lions(uint8_t)`. The input parameter is the variable you use to count the lions. It should preferably be of the type `uint8_t`. In Listing 1.2 an example is shown.

- Try your application by testing all the park functionalities except the lion cage. Everything should work except the ultra sonic sensor which might be broken.

Lab Question 1.4
If a lion moves fast through the senors is it still registered? If not, why? Isn't that a strange(r) thing(s)? Are there any limitations to consider while polling sensors?

```c
#include "true_den.h"

//  Your variables

int main(void) {

    // Your initialization code

    security_system_init();

    while (1) {

        // Lion cage code

        send_lions(your_count_var); // Enter the variable that keeps track of
                                    // the number lions that are in the wild zone

        security_system_run(); // This function runs the security system
    }
}
```

Listing 1.2: An example of how to add the functions.

You are now done with this part, show your work to a lab assistant! _____ ☑

Chapter 2

Interrupts

In system programming, an interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, the processor resumes normal activities. There are two types of interrupts: hardware interrupts and software interrupts. Lab exercise 4 will focus on hardware interrupts.

Hardware interrupts are used by devices to communicate that they require attention from the operating system. Internally, hardware interrupts are implemented using electronic alerting signals that are sent to the processor from an external device, which is either a part of the computer itself, such as a disk controller, or an external peripheral. For example, pressing a key on the keyboard or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position. Unlike the software type (described below), hardware interrupts are asynchronous and can occur in the middle of instruction execution, requiring additional care in programming. The act of initiating a hardware interrupt is referred to as an interrupt request (IRQ). - Wikipedia

2.1 Exercises

Answers to the questions can be found in Appendix **??**.

Interrupt Handling

2.1 Assume you have an assembly routine along with an interrupt routine to handle timer overflows, shown below.

```
start:
    ...
    ldi r18, 24
    ldi r19, 30
    add r19, r18
    ...
    rjmp start

TIMER1_OVF_vect:
    ...          ; handle stuff
    ldi r18, 50  ; store 50 in r18
    reti         ; return from interrupt
```

(a) Assume an interrupt happens between the two load instructions, what will the result of the add instruction be after the interrupt has been handled?

(b) From the calling conventions it is known that register 18 does not have to be saved, if used. Then, is it correct to implement the interrupt this way?

2.2 You have implemented a program that uses the ultrasonic sensor on a drone to measure the distance to the ground. One timing critical part of the code is shown below, where you send the trig pulse to the sensor.

```
int main()
{
    ...
    // send trig pulse
    PORTC |= (1 << TRIG);
    _delay_us(10);
    PORTC &= ~(1 << TRIG);
    ...
}
```

(a) Assume an interrupt occurs right after you set the trig pin to high, and the interrupt takes 48 μs to execute, how long will the trig pulse be?

(b) This might cause problems with the sensor. How would you fix the problem, assuming you can not optimize the interrupt routine further?

2.3 You recently bought some kick ass LEDs to light up your guitar. You feel that the intensity is way to bright and you want to dim the LEDs. Unfortunately, the PWM controller is broken in your microcontroller, so you can only use the timer as a counter.

(a) How would you be able to generate a PWM signal using interrupts?

(b) Implement it!

2.4 The AVR Atmega family has one processor core, thus can only run piece of code (thread) at a time. This was cool back in the 80s, but does not even impress kids anymore. Having only one core, it is physically impossible to run several threads at the same time. However, that does not prevent us from running several independent threads. From the code below, we see that we can not call both functions, since we will get stuck in one of the loops forever.

```
void thread1()
{
    while (1) {
        // run the latest AI algorithms, trying to make money ...
    }
}

void thread2()
{
    while (1) {
        // play the next episode on Netflix
    }
}
```

(a) Using a timer and interrupts, how can you implement the functionality to run both threads alternately, i.e., you shall switch between the threads (context switch)?

(b) **Bonus** Implement it! The authors did :)

2.2 Lab Exercises

2.2.1 Resolve the Issue - Part I

The problems that was introduced by Rust's technicians will be addressed now. What the well willing man did not know was that all the I/O-pins on the Atmega1284 can be configured to be a source that generates interrupt, a so called pin change interrupt (PCINT). This is good news. Inside the microcontroller there are circuits that monitor the state of the I/O pins. When the state is changed, when it goes from high to low and low to high, an interrupt service routine, ISR, is invoked. By using this feature it can be assured that the signals from the sensors would not be missed. Two registers are used to configure the I/O-pins so that they act as interrupt sources. The first is the Pin Change Mask Register, `PCMSK` (there are actually four of them, one for each port) and the other is the Pin Change Interrupt Control Register, `PCICR`.

Tasks:

- Create a new project in Atmel Studio.

  Home Assignment 2.1
  Look on page 13 in the datasheet to see which `PCINT` (Pin Change Interrupt) the sensors are connected.

  The sensors are connected to port C, sensor 2 to pin 7 and and sensor 1 to pin 6.

  Home Assignment 2.2
  Write code that enables the correct Pin Change Interrupt in the corresponding Pin Change Mask Register, `PCMSK`. See page 92, 93, 94 and 95 in the datasheet for the details.

  Home Assignment 2.3
  How do you enable the pin change interrupt for the corresponding port in the Pin Change Interrupt Control Register, `PCICR`? See page 90 in the datasheet.

- Create a function using the code from the above home assignment.

  In order to get the interrupt to work, an h-file, `interrupt.h`, needs to be imported . It can be found in the folder `avr`. To include a an h-file the following command can be used `<folder/file.h>`.

- Import the h-file into your project.

  When all of the tasks are done the pin change interrupt circuit inside the microcontroller will request an interrupt when the I/O-pin changes. This request is only granted if the Global Interrupt Flag in the status register of the microcontroller is set. The provided function `sei()` does exactly that.

- After the initialization code, call the `sei()` function so that the microcontrollers global interrupt flag is set.

  When the above is done it is time to add the interrupt service routine. As mentioned before the routine will be invoked every time the state of the pins are changed. The interrupt service routine is preferably

added at the bottom of the code. See Listing 2.1 for a generic AVR interrupt service routine. This is how all AVR interrupt routines look.

```
ISR(Interrupt_vector)
{

}
```

Listing 2.1: A generic interrupt service routine.

Every interrupt service routines has an input. The input is an interrupt vector which designates a interrupt service routine to a specific source. In Listing 2.1 the `Interrupt_vector` has to be changed to the one that corresponds to the pin change interrupt. The name of that vector is `PCINT2_vect`.

- Replace the `Interrupt_vector` with the one for pin change interrupt, `PCINT2_vect`.

  At this point it is time to verify if the `PCMSK` and `PCICR` register is properly configured. This can be done by using a global counter variable. But first there is something to consider. An ISR is never called from the code, like normal functions. This can cause problems when the compiler optimizes the code. Since it seems like the ISR is never called there is no reason to update the value of a variable used in the routine. If that variable is used elsewhere in the code, it is not guarantied that the current value is the correct one. Because of this, variables that are used in an interrupt service routine must be declared as a `volatile` variable, this `volatile` keyword was used for the same reason in lab 2 when pointing at something in the memory that could change by external means. The `volatile` qualifier ensures that the variable value is updated every time it is used. See the Listing 2.2 below to see how to use the `volatile` qualifier while declaring a variable.

```
volatile uint8_t var1;
```

Listing 2.2: Declaration of a volatile variable.

- Declare a global counter variable and increment it in the ISR.
- Put a break point at the variable in the ISR and start the application in debug mode.
- Move an obstacle through the sensors. Make sure that the execution is halted when an obstacle is passed through the sensors.
- When the execution is halted add the variable to the watch.

If you do not remember how to add and use break points and adding a variable a watch refer to this video: `https://youtu.be/Rx7aLKsiyRA`.

- Stop the debug session and remove the break point by left clicking on it. (*It is very important that the application is stopped.*)
- Start the application in debug mode and move an obstacle (it has to be big enough so that it can cover both sensors) through the two sensors. Pause the application.

Lab Question 2.1
What is the value of the variable?

2.2.2 Resolve the Issue - Part II

Now it is time the create an application that does the same thing as before, counting and keeping track of the lions. But this time the application should be driven by the interrupt service routine that was created and verified in the previous assignment. This means that the code that counts the lions and the `send_lions(uint8_t)` should be inside the interrupt service routine. It is important that no other part of the code is in the interrupt service routine. See Listing 2.3.

```
#include "true_den.h"

//  Your variables

int main(void) {

    // Your initialization code

    security_system_init();

    while (1) {

        security_system_run(); // This function runs the security system
    }
}

ISR(PCINT2_vect) {

    // Lion cage code

    send_lions(your_count_var); // Enter the variable that keeps track of
                                // the number lions that are in the wild zone

}
```

Listing 2.3: A generic interrupt service routine.

Tasks:

- Change the code that keeps track of the lions so that it utilizes the interrupt.

– Do not forget to use the `volatile` qualifier for variables that are used in th ISR.

> **Lab Question 2.2**
> Are there any disadvantages to consider while using interrupts? Can it be known which part of the code that is executed just before an interrupt?

You are now done with this part, show your work to a lab assistant! _____ ☑

2.2.3 Interrupts Gone MAD

In the previous assignment the interrupt solved all problems. This is not always the case and interrupts must be used with caution. In this assignment one of the drawbacks with interrupts will be encountered. The USART peripheral unit and the code for the Neo-pixels will be used for this purpose.

Tasks:

- Create a new project in Atmel Studio.
- Copy the content of `isr_gone_mad.c` to your `main.c`.

> **Home Assignment 2.4**
> How do you enable the USART RX complete interrupt. For details please refer to the datasheet.

- Use the USART initialization function from lab 2. But this time enable the RX complete interrupt.
- Add an interrupt service routine at the bottom mark in the code. The interrupt vector that is used for the RX complete interrupt is this: `USART0_RX_vect`.
- Add a global variable of the type `uint8_t`. This variable should be used to store the received data. Remember to declare it with the `volatile` qualifier.

Do not forget to enable the global interrupt. This is done by calling the `sei()` function.

When using the USART peripheral unit in this configuration, i.e, utilizing the interrupt functionality, there is no need to use a loop to wait until the data has been received. The `UDR0` register can simply be read in the interrupt service routine since it is invoked when the data is received. After the data has been read it is possible to send data back. This is done by writing to the `UDR0` register.

```
ISR(...) {

    a_var = UDR0;    // Recieve

    UDR0 = a_var;    // Transmit

}
```

- Connect the USB cable to the USB-UART converter.
- Configure the terminal YAT with the same settings used in the USART initialization routine. Start the terminal.
- Add a break point in the ISR at the line where the received data is written to the global variable. Start the application in debug mode.
- Send a character from YAT. If the application halts on your break point, you have most likely configured the USART module correct! ;)
- Add the variable containing the received data to the watch.

- Start the execution again and then pause it.

If you do not remember how to set up YAT watch this video: `https://youtu.be/qrL23q0g4VU`.

Lab Question 2.3
What is the value of the variable? What did you send from the terminal? Is it correct? (*Hint: An ASCII table might help.*)

- Send a string of 200 characters from YAT repeatedly.

You can change the speed of the LEDs by pushing button 5 and move your hand closer to the ultrasonic sensor. Higher speed is more likely to show the disturbances from the UART interrupt.

Lab Question 2.4
Do you observe anything strange regarding the Neo-pixels while transmitting the string?

Lab Question 2.5
What is causing the Neo-pixels to randomly light up? (*Hint: When does the interrupt occur? Is it possible to know what part of the code that is executed when the interrupt occurs?*)

Lab Question 2.6
How could you solve the problem?

What does the `cli()` function do?

Lab Question 2.7
At what cost is the problem solved?

You are now done with this part, show your work to a lab assistant! _____ ☑