Crystal Clear Electronics

The development of the Crystal Clear Electronics curriculum was supported by the European Commission in the framework of the Erasmus + programme in connection with the “Developing an innovative electronics curriculum for school education” project under “2018-1-HU01-KA201-047718” project number.

Erasmus+

The project was implemented by an international partnership of the following 5 institutions:

- Xtalin Engineering Ltd. – Budapest
- ELTE Bolyai János Practice Primary and Secondary Grammar School – Szombathely
- Bolyai Farkas High School – Târgu Mureș
- Selye János High School – Komárno
- Pro Ratio Foundation working in cooperation with Madách Imre High School – Šamorín

XTALIN

Contact http://crystalclearelectronics.eu/en/
info@kristalytisztaelektronika.hu

11 - Microcontrollers II.

Written by János Tóth

English translation by Xtalin Engineering Ltd.

Revised by Gábor Proksa

By the end of this chapter, you will able to implement the running lights of K.I.T.T. from the Knight Rider series.

Introduction to the Development Environment

Creating a New Project

During the previous parts, we have installed the development environment, now let's get a closer look at it, and create our project. This can be achieved from several places, the easiest way for me is the file menu. Choose the *New* menu then the *Project* menu.

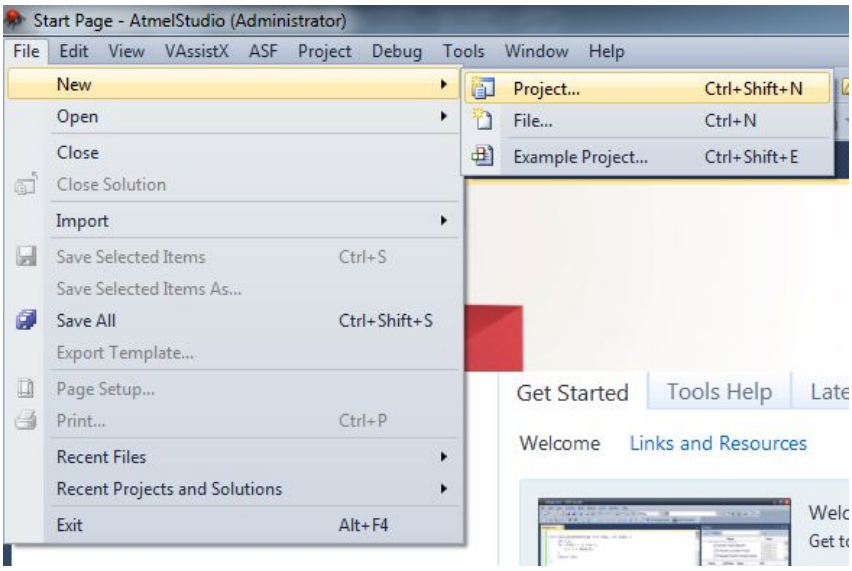

Figure 1 - Create new project menu

After that, we need to set the type of the project. We can choose from C/C++ projects, or even assembly projects. Choose C/C++ on the left, then choose *„GCC C Executable Project"*. This project type indicates that we would like to write our code in C language, and we would like the project to be able to run (executable). Give a name for the project in the *Name* towards the bottom of the window, I've chosen the MyAppl name, but you can give it a different name if you'd like. With *Location* option you can set the directory where you would like to save your project, but the default location works fine too. Click on the *OK* button to move on.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

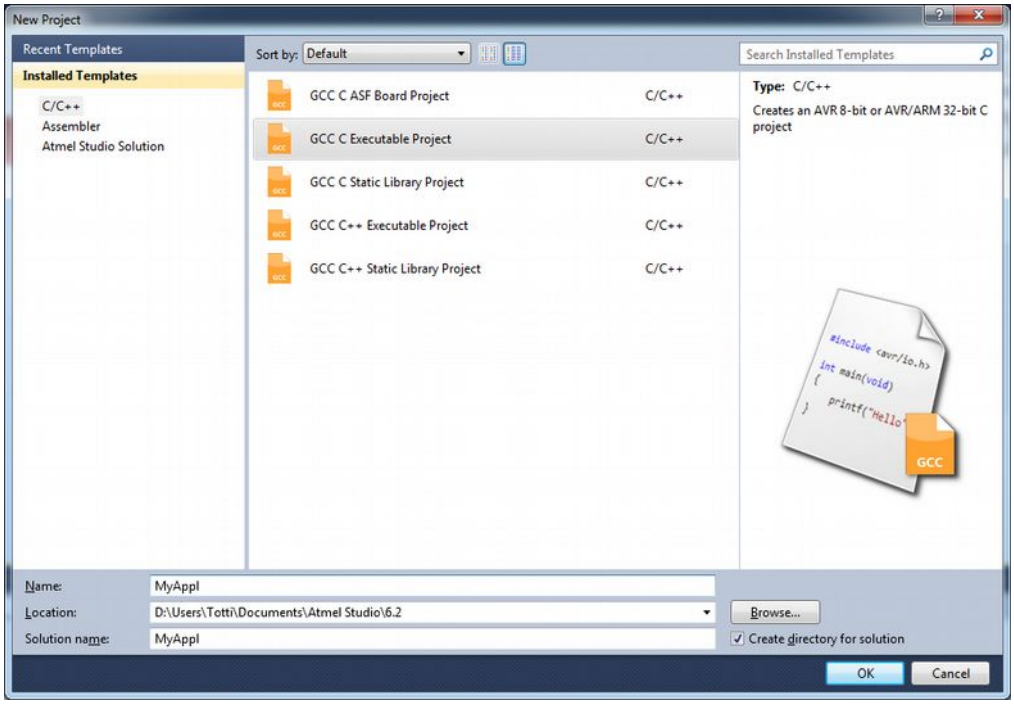

Figure 2 - New project window

In the next window we can choose the type of microcontroller we want to use. There is a search field in the upper right corner to help the selection. In our case, search for the *ATmega16A* keyword to quickly find the correct device. Of course, if you are using another microcontroller, choose the correct one. Under the device info pane, you can find the controller's datasheet and the supported programmer devices. Pressing the *OK* button creates our project.

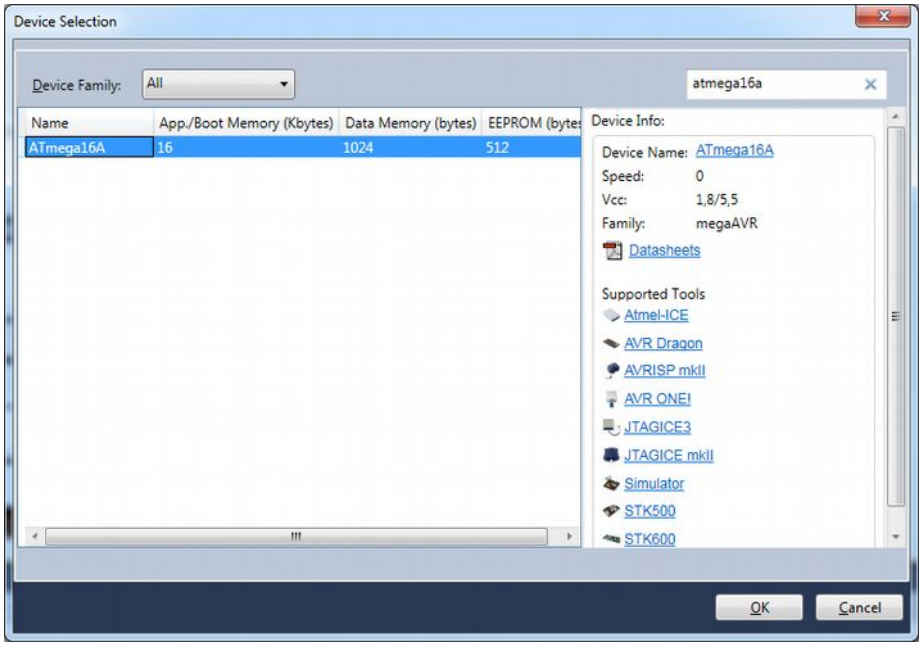

Figure 3 - Device selection, ATmega16A

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

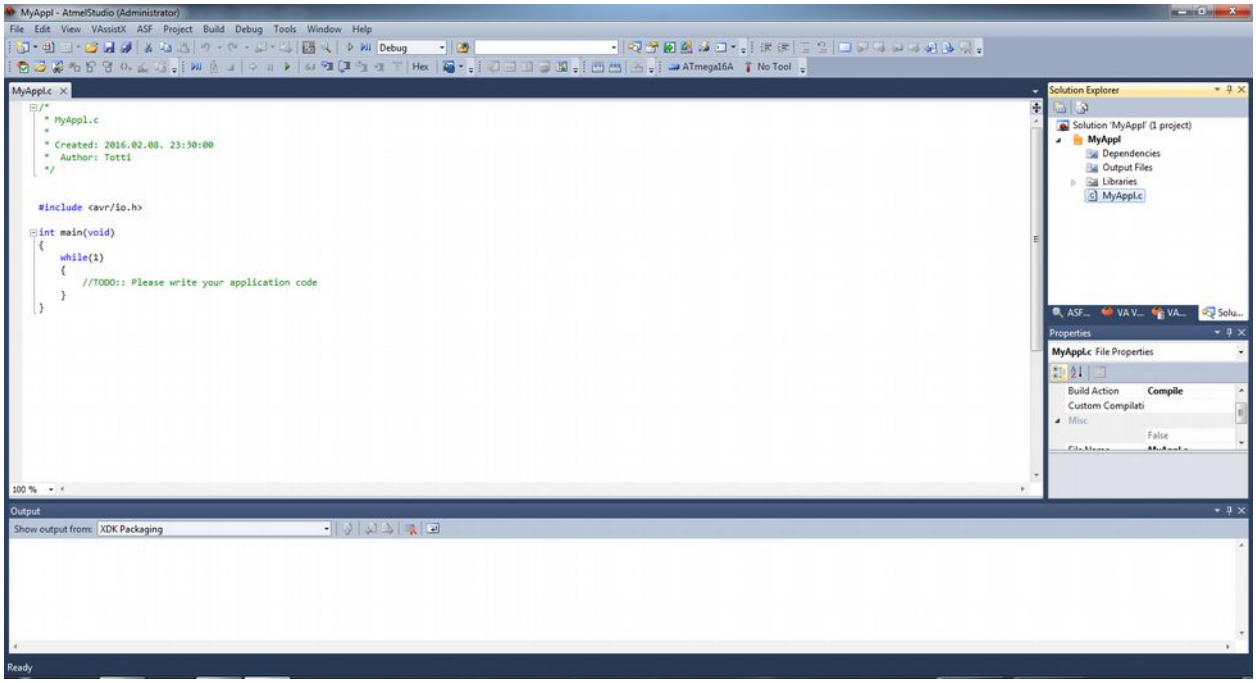

Figure 4 - The created blank project

When the project is created, the development environment creates a C file, with the same name as the project. This file contains a main function with an infinite loop (the while(1) with the curly braces) inside. Now, we can start to develop software for our microcontroller.

FUSE BITS - MEANING AND SETUP

Before we talk about FUSE bits, we must clarify a very important definition, because you will surely see it during embedded development: the register. Registers are used to store data and with their help, the peripherals inside our microcontroller can be configured, or data can be sent to, or read from them. The size of a register is usually given in bits, and can vary between 8, 16 or 32 bits (or even 64 bits in newer microcontrollers). You can imagine a register as a shelf divided into equal sections corresponding to its size. We can place something or take something from these sections, which correspond to the bits of a register. In the following picture, you can see one of the registers called the PORTA register, from the microcontroller's datasheet. This is an 8-bit register and every single bit can be written or read. There are some registers where some or even all bits can only be read and not written. The initial value means the default value of each bit after turning on the microcontroller. In this case every bit will be 0.

PORTA – Port A Data Register

Bit	7	6	5	4	3	2	1	0	
	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
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	

Figure 5 - Structure of Port A register

Previously I've mentioned that we store data in registers, and that in microcontrollers, we use memory to store data. You can guess from here, that registers are nothing but small sections of memory. In order to

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

reach these small parts of the memory, we need their addresses, which tell us where exactly these sections can be found inside the whole memory. Similarly, to how a postman also needs to know the street, house number, floor and door number to successfully deliver mail; knowing only the name of the city isn't enough. All the registers have a fix address, that never changes. Remembering these addresses would be hard, and in many cases the registers are on different addresses in different models of microcontrollers, which means we would have to re-write our whole program if we wanted to use another controller, and hope that we didn't mess up any of the addresses. To avoid that we can use a bit of help. It's much easier if we only need to write down, that the PORTA register's value is 0, instead of searching for the address of the PORTA register in the datasheet and writing the value 0 to this address. For most microcontrollers, the manufacturer usually publishes a header file, where all the names and addresses of the registers can be found. This header file can be included in our program, and we can refer to registers by their names. When the compiler creates machine code from our code, it will translate our register names to the correct address using the header file. If we want to use a microcontroller where the registers have different addresses, we only need to change the header file and not our code.

After talking about the registers, let's move on to the FUSE bits. At this point it probably won't surprise you that the FUSE bits are inside a register too, namely the FUSE register. The most important settings of the microcontroller can be set with the help of these FUSE bits. Be careful, a badly set FUSE bit permanently sever the connection between us and the microcontroller, in this case, we won't be able to program our microcontroller anymore, and we need to replace it. You can see the FUSE bits of the ATmega 16 in the picture below.

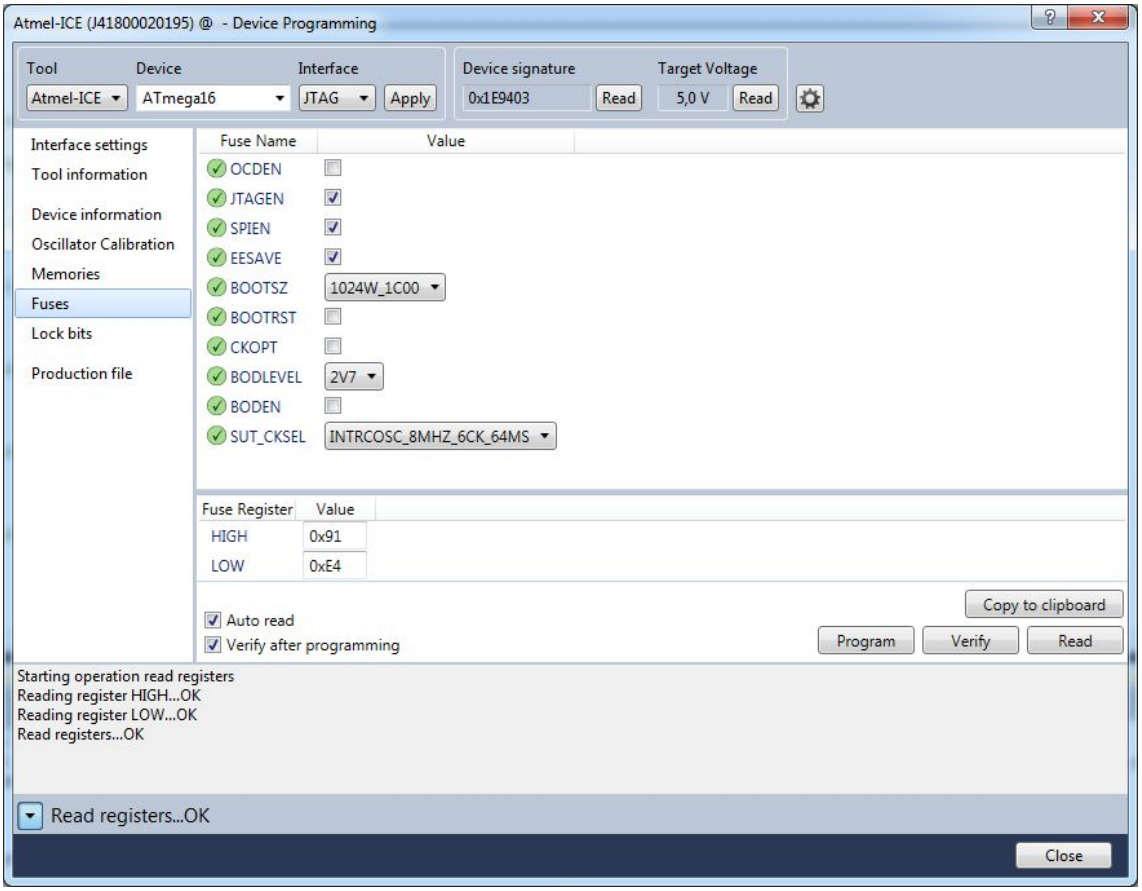

Figure 6 - FUSE register setup

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

The description of every single bit can be found in the datasheet, now we'll discuss only those, that will be used further on. One of the most important bit is the SUT_CKSEL, which is not just one bit but a setting that has 6 bits. With it you can set which oscillator the microcontroller will get the clock signal from. In our example the built-in 8 MHz internal oscillator is selected, but later you can build a circuit with a more accurate external oscillator. In that case you will have to choose the correct oscillator here. This setting is very important, because if you set a wrong value, then the timings will be wrong, and the peripherals won't work properly.

In order to save the settings, you must click on the *Program* button while the microcontroller is connected to the programmer and the PC. When you click *Program*, the values of the FUSE register are saved inside the microcontroller, and the microcontroller will use these values at the time of next launch.

Warning! Make sure that the JTAGEN fuse is set before you program the fuse bits to your device (it is set by default, the checkbox is selected). If you unset the JTAGEN fuse, and program that setting, you won't be able to program your microcontroller anymore, and you will have to replace it.

BLINKING AN LED

So far, we have learned about the development environment and the FUSE bits. Now, let's go over why the circuit built in the 9th chapter is working.

Load the *CE11_1_LED_blink_delay* project into the development environment. You can do this by clicking *File → Open → Project/Solution*, and choosing the project file in the pop-up window.

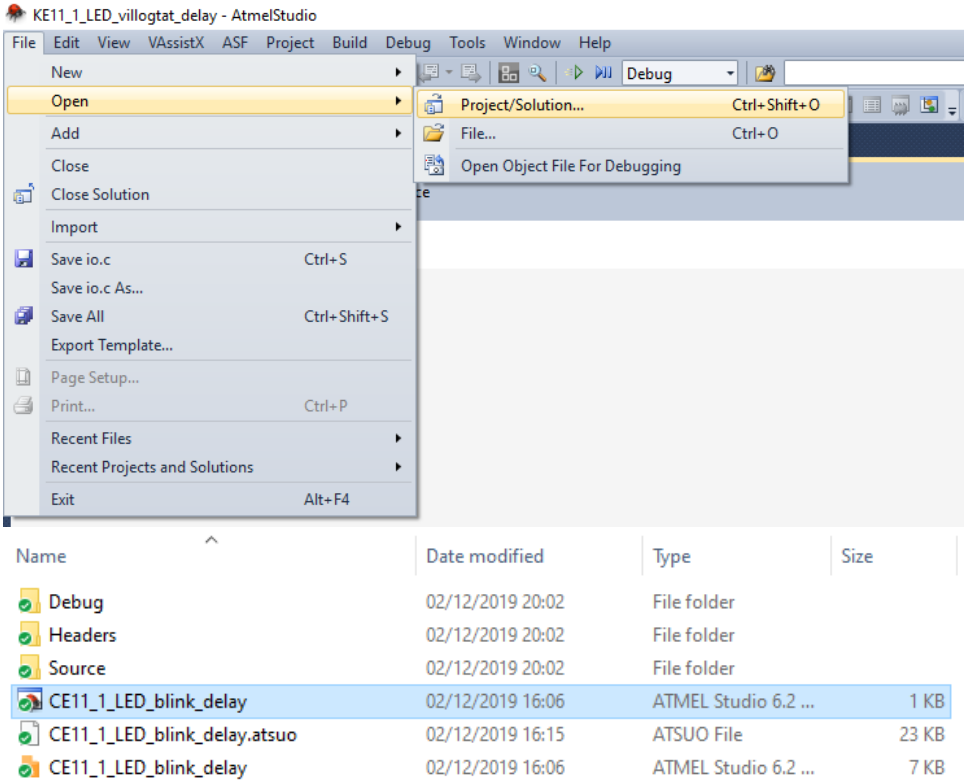

Figure 7 - Open project

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

When the project finishes loading, and you can see the relevant files of the project inside the *Solution Explorer* window (upper right), open the "*main.c*" file. This file contains the *main* function that is first executed after switching on the microcontroller.

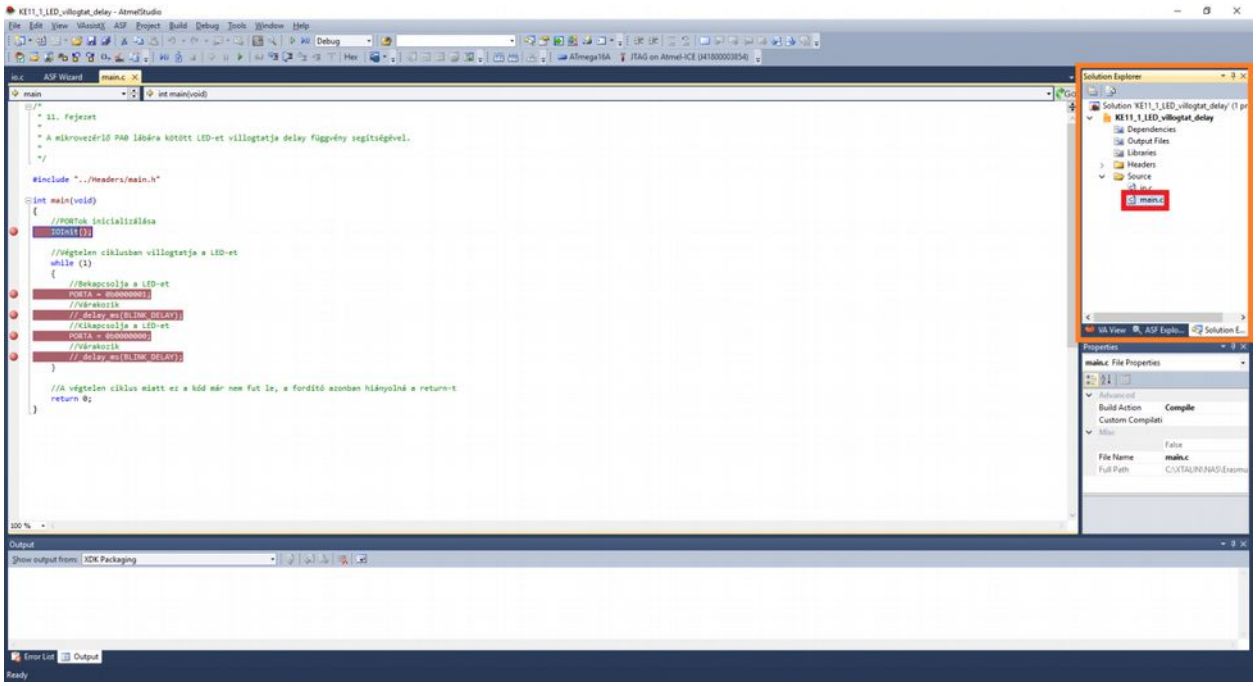

Figure 8 - main.c file

The first command calls another function, the `IOInit()` function. It was already mentioned in chapter 9, that the PA0…7, PB0…7, PC0…7, PD0…7 pins can be used as digital inputs and digital outputs. In order to light up our LED, we need to set the PA0 pin to logic high. This means, that an internal switch connects the chosen pin and the supply voltage together. If we set PA0 to logic low, then the pin is connected to ground, and the LED won't light up.

We have also mentioned in chapter 9, that if we would like to set a pin's logic value, then the pin has to be set up as a digital output. We can use the register called DDR (Data Direction) for this purpose. As described in the datasheet, each port has its own DDR register, for example, for PORTA the register is called DDRA.

In the DDR register each bit corresponds to one of the pins of that port. The DDRA register contains the data direction description for each pin of PORTA. According to the datasheet, if the value of a given bit is 1, then the corresponding pin works as a digital output. So, for the PA0 pin to be a digital output, we have to set rightmost bit (bit zero) inside the DDRA register to 1. This can be easily achieved if we assign the value of the register as a binary number. In case of a binary representation, the value of every single bit is represented by a 0 or a 1. Right click on the `IOInit()` function, then choose the *Goto Implementation* option from the drop-down menu (or use the Alt+G keyboard shortcut). The development environment will search for the place where the function is defined, and offer the results. Choose the upper option (*io.c*), where we can see exactly what the function does. Look at the line below:

```
DDRA = 0b00000001;
```

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Here, the function sets the value of the DDRA register. As you have seen in chapter 10, the “0b” prefix tells the compiler we would like it to interpret the following number as a binary number. We can see in the example code, that 8 digits follow the prefix, which means an 8-bit number, that is exactly the size of the DDRA register. In the first position, the highest place-value can be found, in our case this is bit 7. This is the same way we write the numbers in the decimal numeral system. In the example code, you can see, that only the final number has a value of 1, and the others are all 0, so it sets bit zero to 1 and all the others to 0.

Assigning a value to set the register is recommended only if you would like to set all bits in the register at once to a known state. If you want to set only 1 bit, using some built-in functions is recommended. To set and clear (write to 1 and to 0 respectively) you can use the sbi (setbit) and cbi (clearbit) functions. These functions are declared inside the ”compat/`deprecated.h`” files. To use them, we need to include this file into our code with an `#include` directive placed in the “.c” file. Our case is no exception, we have to write `#include ”compat/deprecated.h”` at the beginning of our file. Setting a bit to 1 with the sbi function looks like this:

```
sbi(register, bitposition);
```

In our example, setting the zero bit of the DDRA register to 1, happens with the following command: `sbi(DDRA,0);`. If we would like to set a bit to 0, then we use the cbi function. As an example, to revert the 0 bit of the DDRA to 0, we can do this using the following command:

```
cbi(DDRA, 0);
```

Let’s have a look at what type of registers belong to the individual ports. Apart from the DDRx registers, there are PORTx and PINx registers as well. The PORTx register performs two functions depending on the direction of the given pin. If a certain pin is set up as an output, then we can set the pin’s value to logic high or logic low by the setting the PORTx register’s corresponding bit. If we write 1 to the position, then the corresponding pin will be set to logic high, in case of 0, it will be set to logic low. The other scenario is, that the pin is set up as an input. In that case, we can turn on the pull-up resistor of the pin with the PORTx register and read the status (logic high or low) of the pin with the PINx register. If you think of the register as a variable with 8 bits (the PINx register is 8-bit wide) then each bit position corresponds to pin of the port. For example, if the value of PINB is 0b00000001, then the zero pin (PB0) has a value of logic high, all the others have a value of logic low. If the value of PIND is 0b01001010, then the 6th, 3rd And 1st pins (PD6, PD3, PD1) have logic high values, the others (PD7, PD5, PD4, PD2 and PD0) have logic low values.

Looking further in the `IOInit()` function you can see that, we turn off the pull up resistors for every port, and set up all the pins as inputs except the PA0 pin. So, in this function, the initialization of the ports and pins happens (inputs and outputs), which is why it is called “IO init”.

Looking back at the *main.c* file, after the initialization an infinite `while(1){}` loop follows. The commands inside the infinite loop will be executed over and over until we restart the controller, or an error occurs. The first command of the loop sets the PA0 pin to logic high value through the PORTA register. We already know how this works from the previous section. After running the code, the LED connected to the pin will light up. In order to flash the LED, it’s not enough to just turn it on, we’ll have to turn it off as well, then turn it on again, and so on. Thus, after turning it on, inside the cycle, we need to

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

turn it off too, which can be achieved by setting the PORTA register's 0 bit to 0. Now build the circuit which we tested in chapter 9 again.

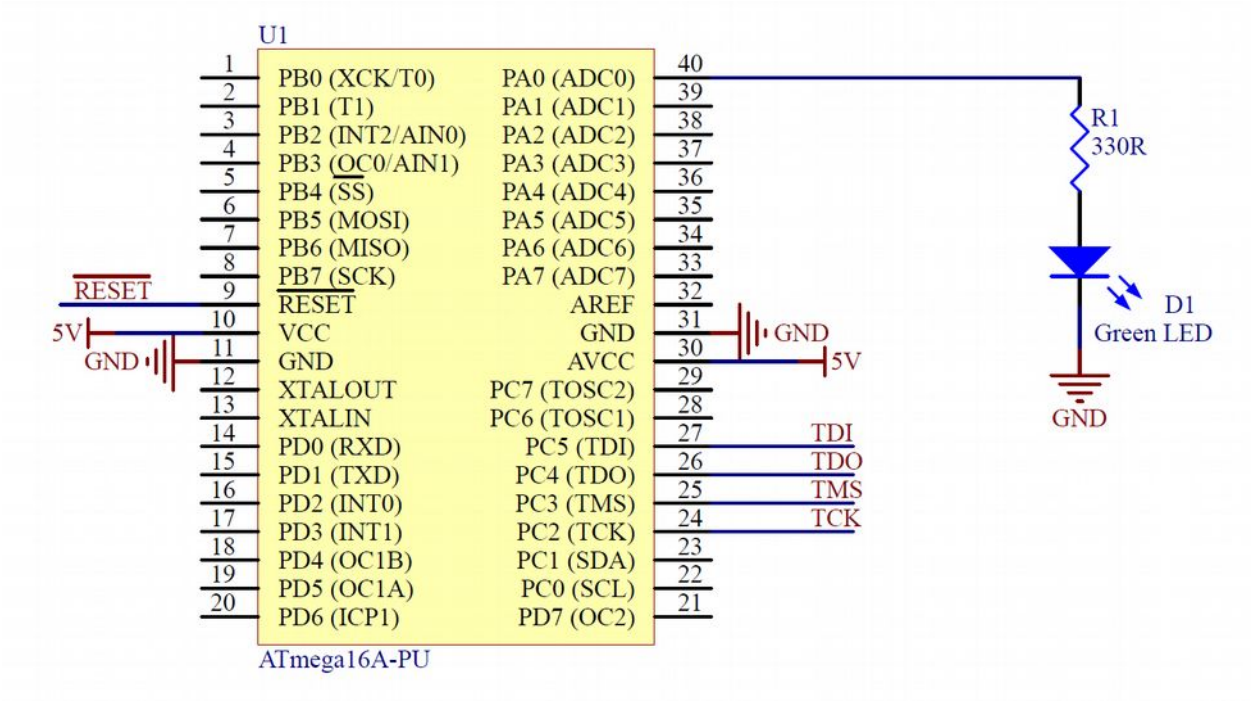

Figure 9: Wiring diagram of blinking an LED

Then upload the example code to the microcontroller without changing anything by clicking on the *Start Debugging* button.

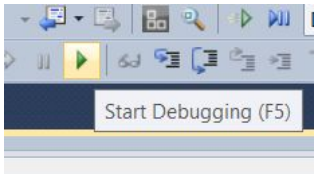

Figure 10: Upload of program (reminder)

We can see, if the main loop contains only these two commands, the LED won't flash, but continuously light up instead. But why? It turns out it is flashing, but we just cannot see it. During the previous chapters, we have already talked about that the microcontroller executes a command after every single clock signal. The clock signal of the microcontroller is 8 MHz. This means, that it executes 8 million commands under a second. Our main loop has only two commands to execute, so we turn off the LED 4 million times under a second, and then back on. Our eyes cannot see a change this fast. We should put a delay between the turn on and turn off commands somehow. Think about how we could achieve this. How to put the execution on hold in the microcontroller a certain period of time? We will talk about this a little bit later, but for now let's use the pre-existing `_delay_ms()` function, which will make the controller stop and wait for some amount of time, specified by the parameter of the function in milliseconds. Use this function after calling each of the two commands, so the processor will wait for a certain time after turning the LED on or off. Let the delay be 500 for example, therefore we will turn on and off the LED every half second. Write `_delay_ms(500);` after both turning on and turning off the LED:

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

```
while (1)
{
	//Turns on the LED
	PORTA = 0b0000001;
	//Wait
	_delay_ms(500);
	//Turns off the LED
	PORTA = 0b0000000;
	//Wait
	_delay_ms(500);
}
```

This ensures, that the LED flashes only once a second. Upload the code to the controller and check its operation.

DELAY VERSUS COUNTING CLOCK SIGNAL

In the previous section we have talked about the way we can make the microcontroller wait, by using the `_delay_ms` function. Is there another way to achieve the same effect? Let's have a look at the things we've learnt so far. We know that the microcontroller executes a command in every clock signal cycle. We also know the clock signal frequency. From these two we can calculate the number of commands that are executed under a certain amount of time. For example, if we have a variable, and we continuously increase its value inside a cycle, then we'll get a counter which steps forward one after every clock signal. All we have to do is move the counter until it reaches a value we have calculated for our desired delay.

An example: The clock signal of the microcontroller is 8 MHz, so 8 million commands are executed each second. Create a variable and increase its value inside a loop, until it reaches 8 million. In this case, if we check the variable at the end of every loop, then 1 second has elapsed when the counter reaches 8 million. We can implement a delay algorithm using this method, which can be used if we do not require very accurate timing, just an approximate delay. On the other hand, the `_delay_ms` function provides very accurate timing. An example function:

```
void _own_delay_ms(unsigned int delay)
{
	//internal variables
	static unsigned int counter;
	static unsigned int time;

	//reset the elapsed time
	time = 0;
	 //until it reaches the required time
	while(delay > time)
	{
		//at the beginning of every cycle reset the counter
		counter = 0;
		//increase the value of the counter until 8000
```

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

```
                //theoretically we wait 1 second with this
                while(counter <= 8000)
                {
                      counter++;
                }
                //increase the value of elapsed time
                time++;
          }
    }
```

Copy this function before the main function, and replace our calls of `_delay_ms(500)` to `_own_delay_ms(500)`.

```
    while(1)
    {
         //Turns on the LED
          PORTA = 0b0000001;
          //Waits
          _own_delay_ms(500);
          //Turns off the LED
          PORTA = 0b0000000;
          //Waits
          _own_delay_ms(500);
     }
```

After uploading the code, we can see that the LED is flashing, but it's much slower than we expected. One period is just under about 6-7 seconds instead of 1. The reason for this is we are not only executing one instruction in each loop (`counter++`), but also a bunch of other instructions:

1. `counter` reading the value of the variable from the memory to the operational register
2. `8000`, writing it into the operational register as a value
3. executing the comparing operation(<=)
4. jumping to the `counter++` or the `time++` lines according to the result of the previous operation
5. increasing the value of the `counter` or the `time` variable
6. writing the value of the `counter` or the `time` variable to the memory
7. jumping to the beginning of the cycle

It isn't that easy to calculate how much time it takes to run a delay cycle like this, for this reason it's inappropriate for precise timing as we've mentioned before. If we write 1195 instead of 8000, then the delay will be quite acceptable, but it's still not accurate enough.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

It's worth to mention about the example, that the variables may overflow. We know that our variables are stored in finite sized memory, so they cannot be arbitrary big. An unsigned 8-bit variable (`char`) can have a value from 0 to 255. Why would that be a problem? Let's have a look at this number represented in binary, to see what happens at the time of overflow. Remember, the processor also works with in binary. If we write the decimal 178 in binary, stored on 8 bits we'll get 0b10110010. Add decimal 150 to this number, which looks like 0b10010110.

```
 10110010
+10010110
101001000
```

The result cannot be represented with an 8-bit number because it is larger than 255. However, we store the result as an 8-bit number in memory, so the highest place-value won't have a place to go. In technical terms this is called an overflow (like overfilling water in a glass). In this case, the calculated result will be 72 (0b**01001000**) instead of the expected 328 (0b**101001000**). This could cause a big problem with any mathematical operations, so you have to watch out for the variables' range.

Variable Type	Size	Range
`char`	8 bit	**0** - **255** or **-128** - **127**
`unsigned char`	8 bit	**0** - **255**
`signed char`	8 bit	**-128** - **127**
`int`	16 bit	**-32768** - **32767**
`unsigned int`	16 bit	**0** - **65535**
`long`	32 bit	**-2147483648** - **2147483647**
`unsigned long`	32 bit	**0** - **4294967295**

Think about increasing the variable in the previous example until it reaches 8000 (or 1195). If my counter was an 8-bit variable, then there wouldn't be any problems until 255, each cycle would increase the value by 1. However, if we added 1 to the maximum value of 255, we wouldn't get 256, but 0 again. This way the counter would never reach 8000 or 1195, or any value higher than 255, and the program would stay in an endless loop forever.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

BLINKING 2 LEDS

In the previous section we have seen, how we can flash an LED with our microcontroller. We have learnt how we can configure the pins to be outputs, and how we can turn on and turn off the LED connected to a pin. We've also examined how we can introduce delays in our code.

In this section, we will make things a little more complicated. We don't want to flash only a single LED, but two LEDs. First, we have to build the circuit by modifying the previous circuit a little bit. Connect a second LED and resistor to the PA2 pin, similarly to the first one on PA0.

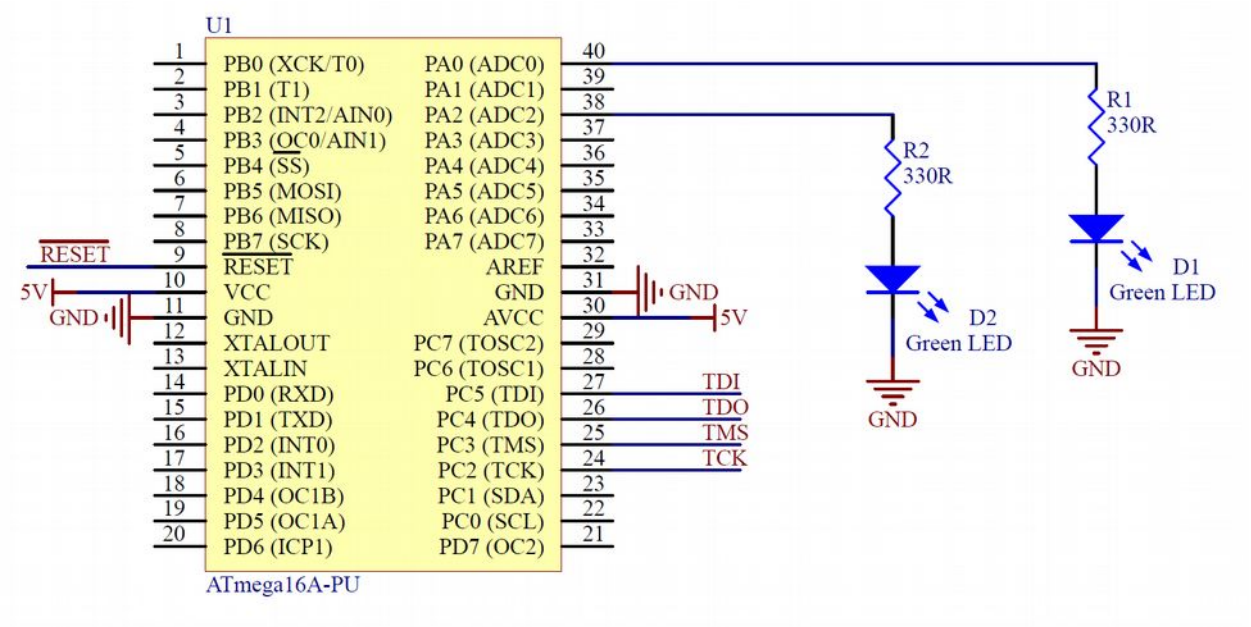

Figure 11 - Circuit for blinking two LEDs

Now it's time to modify the code. Open the *CE11_2_LED_blink_delay* project first to keep our previous example as well. Remember what we did that time:

1. The pins have to be configured as an output
2. The pull up resistors of the pins have to be turned off
3. Changing the state of the pin at the right time

The configuration of the pins, i.e. the `IOInit()` function, can be found in the *io.c* file just like last time, so let's make the changes here first. We already know that we can set the direction of the pins with the DDRx register. In this case, we would like to configure PA0 and PA2 pin as an output. In our example code we configure the whole DDRA register with a single command, so it's makes sense to modify this. Alternatively ever bit could be set with calling `cbi()` and `sbi()` functions. Since we entered the value of the register in binary, we simply need to change to 2nd bit's position to 1 as well. The command will look like the following:

```
DDRA = 0b00000101;
```

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

This file doesn't need any further modifications, so you can save and close it. If you have connected the LED to a pin of another port, let's say PB5, then we should set the value of the DDRB register's 5th bit instead.

This is the end of the configuration process; you can now control the pins. We will do this in the *main.c* file just like last time. We will modify the status inside an infinite loop too, as we did it at the time of LED flashing. Currently, we set the whole port during state change by assignment, but if we would like to control more than one pin independently, this isn't the best solution. Let's use the already learnt `sbi()` and `cbi()` functions instead. With them, it's guaranteed, only that particular pin will change its state, and the others will remain the same.

At first, we will flash the LEDs alternately, as long as one lights up, the other doesn't. Let the time between each status change be 1 second, so in every second the other LED will light up. The initial state should be such, that the LED on the PA0 pin lights up first and then 1 second later the one on PA2 pin does. To achieve this, in the infinite loop the first command should be `sbi(PORTA,0);` and the second command should be `cbi(PORTA,2);`. Then we wait for 1 second, so the next command is `_delay_ms(1000);` Next, we have to make changes to the states again, setting the PA0 pin to logical low and the PA2 pin to logical high. Thus, the following two commands are `cbi(PORTA,0);` and `sbi(PORTA,2);` Now the two pins are inverted, only a final but very important command is missing, waiting for 1 second again. If we would omit the second `_delay_ms(1000);` command the infinite loop would start over immediately, switching the two LED-s back to the first state.

```
while (1)
{
	//Turns on the LED1, turns off the LED2
	sbi(PORTA,0);
	cbi(PORTA,2);
	//Waiting
	_delay_ms(1000);
	//Turns off the LED1, turns on the LED2
	cbi(PORTA,0);
	sbi(PORTA,2);
	//Waiting
	_delay_ms(1000);
}
```

Save the code, then compile and download it to the microcontroller. If you did everything right, you should get a compiled code without errors and the LED-s flash as expected.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

A small programming trick for inversion

We would like to show a small trick which can be used to implement the same flashing or simply to switch a pin's state back and forth. If we would like to invert a pin's state, we can use bitwise operators as well.

Let's stick to the code where we would like to flash two LEDs. We're inverting the state of the PA0 and PA2 pins at specified intervals. An initial value is needed. Let's put this before the infinite loop, because we would like to set this only once. Like this: `PORTA = 0b00000001;` With that, we have set the PA0 pin to logic high value and the PA2 pin to logic low value.

Now for the trick. We have the initial state and we would like to invert the state of the two pins. Use the command `PORTA ^= 0b00000101;` This will change the state of the two pins in a single instruction.

What exactly did we do? We know, that we have to modify the PORTA register's two bits, and leave the others as they are. The previous code contains language abbreviations, the `^=` operator is just the short way of writing:

```
PORTA = PORTA ^ 0b00000101;
```

The two code does the same thing. Take a look at the longer one to better see how it works. We've already talked about bitwise operators in previous chapters, so I won't explain those. Here we are using the exclusive or operator. We know, when using the exclusive or, the result will be 1 if the bits on the right and left side differ from each other, which is the property we are exploiting here. Where we don't want to modify the state, we write 0, where we would like to modify the state, we write 1. The following table will help you to understand why this works:

Original state	Modifier bit	New state
0	0	0
0	1	1
1	0	1
1	1	0

The table is identical to the the truth table of the exclusive or operator. We see that if the modifier bit is 0, the operation retains the original state, while if the modifier bit is 1, the operation flips the bit. The complete alternate LED blinking can be implemented as shown below, with only the exclusive or operator and a delay inside the infinite loop.

```
PORTA = 0b00000001;
while (1)
{
    PORTA ^= 0b00000101;
    _delay_ms(1000);
}
```

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).