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

12. Debugging – Troubleshooting in the Program

Written by Dávid Kiss

English translation by Xtalin Engineering Ltd.

Revised by Gábor Proksa

Processors can perform various operations on binary numbers, such as addition, comparison, and other logical operations. In order to do useful tasks for us, we need to tell the processor which operations to execute. These commands are called instructions, and the set of all the instructions understood by a specific processor is called an instruction set. You can assign a binary number to each instruction. By doing this, we have solved the problem, because if we want to give a command to the processor, then we need to send a binary number, which will tell the processor what to do.

The processor of a microcontroller works on the same principle: it reads numbers out of its program memory and executes instructions based on them.

How does the instruction get into the program memory? The easiest thing would be to write the binary numbers, belonging to the instructions, "manually" in the memory. We could do this if we only have a few instructions, but it would take a long time to type them in case of a complicated program, not to mention the amount possible of mistakes. That's why the assembly programming language was invented.

In assembly, the instructions are not referred to by binary numbers, but by short text (for example, "ADD" means the addition). After writing a program in assembly another program called the assembler translates this text into the binary instructions understood by the processor, called machine language.

The only problem with this method is that different processors may have different instruction sets, so each processor has different assembly code. This problem can be overcome by using high-level programming languages, such as the C language, which you are already familiar with from the previous chapters of the curriculum. C code lines are compiled into assembly language by a computer program, the compiler, and then the compiled assembly code will be assembled into machine binary code.

```c
int main(void)
{
    //PORTok inicializálása
    IOInit();

    //A végtelen ciklus minden egyes lefutásakor a PORTA lábak értéke az ellentettjére (negáltjára) változik.
    while (1)
    {
        PORTA = ~PORTA;
    }

    //A végtelen ciklus miatt ez a kód már nem fut le, a fordító azonban hiányolná a return-t
    return 0;
}
```

Figure 1 - C source code

The first figure shows some C source code, which is what we can read easily. Of course, our program does not consist only of a single file, the operation of a program can be described with several “*.c” and “*.h”

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).

files. The contents of such a "*.c" file can be seen in the figure above, and you will most often encounter these files during the curriculum.

Wildcards

(In computer science, "*" is typically used to mean any number of characters can be at the place of "*". So "*.c" means all C files. You can try a "*.c" or "*.jpg" in the search engine of our computer; the latter will list only pictures with jpg extension.)

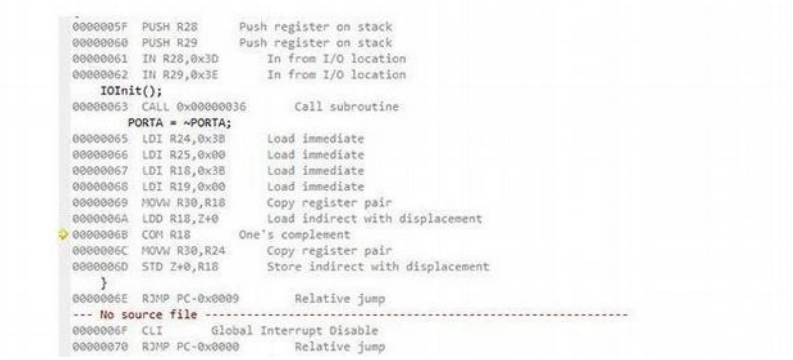

Figure 2 - Assembly source code

The first step in the compilation is done by the preprocessor and the compiler. The compiler converts the source of high-level C language into a lower-level, so-called Assembly language code. We can see this code in the middle step. This file still contains readable information for human mortals. The assembly code implements the functionality required by the C source with the processor's limited instruction set.

From here on, only one step separates us from the machine code, as it was mentioned before, the assembly file can be matched one-to-one to machine instructions, only the words have to be replaced with the corresponding byte sequences.

Figure 3 - Machine code

The last step is completely hidden from us, the computer solves this problem as well. We already had several source files in our previous simple programs, and they are compiled to as many of these small sequences as many "*.c" files we have. However, we can only upload one to the processor, so as a last step, the so-called linker links these files to one whole.

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).

Of course, this process is far from trivial, and the same C code can be implemented in several ways in assembly. The compiler *optimizes* the code, executes some things in a different order, if we do not use a variable, then the compiler will not create it unnecessarily, etc. We can turn off optimization and then what happens is exactly what we described. During the curriculum, we will switch optimization off to avoid any misunderstandings due to its operation.

There are still programs that are written in assembly language today. One of the reasons for this is to control execution time and program size. When using assembly, you know exactly how much space is occupied by each instruction and how many clock cycles does it take to execute them. Unfortunately, it is not always possible to predict in advance what kind of instructions the compiler will use to implement our program written in the language C.

The other reason is that we want full control over the processor. The code written in assembly is clearly translated into machine code, the processor executes exactly what we have written. The C compiler is also a program, made by humans so there may be errors in it, in some cases it can misinterpret our instructions, and output something different.

Inline assembly

In C, it is also possible to write some code lines in assembly, these are simply copied into the assembly program. So, in critical situations, we can take full control.

WHAT IS DEBUGGING?

Now that we know how our code is compiled, we can look for errors in it. During development, the first requirement is to write syntactically correct code. We have discussed this in earlier sections. However, correct syntax alone does not mean that the code behaves the way we designed it.

The processor executes commands millions of times every second, so we have no chance to follow it with the naked eye. On the other hand, even if we could, we can't really see the flow of things, because it is very difficult to look at the dance of electrons in the silicon chip (and not just because it is in a black plastic case). These problems are overcome by debugging, which is the process of troubleshooting. We have the ability to run our code step by step, track the status of the different registers and variables, and thus verify that the operation is correct.

Etymology

If you are well versed in the English language, you may wonder about the origin of the word "debugging". The correct definition of the word is "identifying and removing errors", but the question arises: Where does the term come from? In the old days, even when computing was in its infancy, and the computational capacity that is now in the small black case in front of you, filled an entire gym-sized room, debugging was a bit different. In this huge space, the various components were placed and connected together with many wires. These machines have not yet been programmed, the desired function has been achieved with the right connection with tens of thousands of wires. Of course, this complex construction usually did not work for the first time, it also contained errors that had to be discovered by the engineers. As a result of

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 long research work and experimentation, an actual bug was found, which shorted several wires together. After removing the bug, de-bugging, the system operated as expected so it turned out that there was no design error.

A SIMPLE EXAMPLE

The first example, which we hope will highlight the criticality of debugging as a function, is a very simple, single line small program. For a more spectacular result, connect an LED to the microcontroller's PA0 pin as we did in chapters 9 and 11. Upload the *CE12_1_LED_blinking* project into the microcontroller. If this is done successfully, we will see that the LED on the PA0 pin lights up. Let's just look at the uploaded code!

Based on the lessons learned from the *CE12_1_LED_blinking* software, the LED should be blinking, and as it has been discussed in the previous section, it is blinking as well. The processor runs the code very quickly, while the LED is blinking, but so fast that the human eye is unable to detect, so we see it as constantly lit.

First, let's look at what is the basis for what the remainder of this chapter of the curriculum will be about. To do this, just follow the instructions now and then we will look at the features in more detail later. Before we would compile the project, we clarify some of the settings. During the introduction it was said that optimization will not be our friend, so make sure that it is really turned off. This can be found in the project settings on the *Toolchain* tab under the *AVR/GNU C Compiler/Optimization* section. Select *None (-O0)* here.

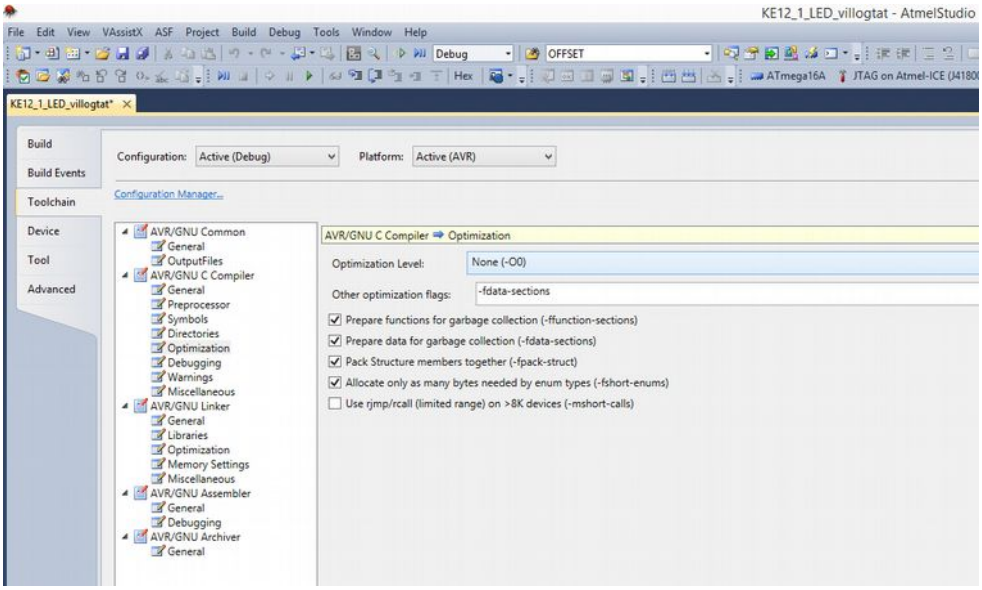

Figure 4 - Switching off optimization

Run the code now in debug mode! To do this, click on the *"Start Debugging"* button. We can see that the interface of Atmel Studio has changed slightly. In this mode, the code on the microcontroller runs the same way as before, but the computer keeps continuously communicating with it, so we can influence the running of the code, for example, to pause or move it line-by-line.

Let's try the latter too! In the first step, use the "Break all" (Ctrl + F5) button to stop the program from executing, then use the "Step Over" (F10) button to step through within the code per line (function call).

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).

In parentheses, you can find the key combination assigned to the functions, and we recommend that you use them now, so you don't have to search for them on the toolbar.

Step by step, we can see that the LED really changes its state. So, the program works correctly, only our eyes are not fast enough to detect the blinking. The LED blinking program written in the previous section also worked the same way, but it contained a delay. This is necessary for our eyes to be able to sense blinking. It may be interesting to try to find out how many ms delays are needed in order to be able to distinguish the individual flashes with the naked eye.

OVERVIEW AND FUNCTIONS OF THE DEBUGGING INTERFACE

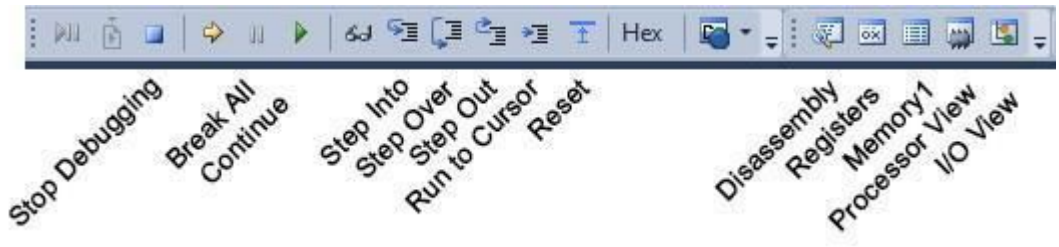

Figure 5 - Debugging menu bar

Now that we are beyond the first steps, let us get to know the rest of the buttons on this toolbar, which we will use henceforth. The first three buttons affect the code run.

STOP DEBUGGING

This button is used to stop debugging. It terminates the debug process and the connection between the microcontroller and our computer. It's important to stop debugging after we've finished debugging and wanting to change the code. If we do not act like this, then we will not be able to upload the modified program to the microcontroller because the programmer will be busy.

BREAK ALL

When we click this button, we are suspending our program. This should be imagined as if we could stop the clock in the microcontroller. Of course, this is not the case in reality, since we would not be able to communicate with it if it was.

CONTINUE

If we stop the program from running, somehow, we have to continue later. This button serves this purpose.

The following buttons are also related to code execution, so we can “move” the code, i.e. run at the pace we want.

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).

STEP INTO

It executes the following code sequence, so that if it is a function call or a block with braces, it enters it and executes its first line.

STEP OVER

Also executes the following line. If it is a function call or a block with a bracket, it does not enter, but the contents of the block are executed.

Suppose that an LED is not turned on and off by simple bitwise operation, but we have a SetLED() function. When you move to this function with the “Step Over” button, you only see that the LED turns on, but if you move on with “Step Into”, you get inside the function, where you can follow the steps to turn the LED on.

Encapsulation

In terms of coding, it is more elegant to use the SetLED() function, as it hides the bit-behaviors from the reader, and instead we call a function, whose name tells its function without commenting and implements the necessary operations in the background.

STEP OUT

We can quit the function, in which we are in currently. The program stops at the next function call. Of course, the rest of the function will be executed, but the run will not stop until the beginning of the next function.

RUN TO CURSOR

The code will be executed until the position of the cursor.

RESET

It is used to restart the processor. It is practically equivalent to pull out and re-plug the processor's power supply except the inconvenience that it is not needed to rebuild the debug connection.

Using the remaining buttons, we can open different views to get to know the internal state of the processor. The common feature of these views is that we can not only read the different registers, but we can interfere "by hand" into their state, that is, we can overwrite their values.

DISASSEMBLY

If you click here, the Assembly code will appear, which was compiled from our C file by the compiler. We may need this view if we want to know exactly what happens in the processor.

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).

REGISTERS

This view allows us to monitor the contents of the CPU registers. There are 32 such 8-bit registers in the microcontroller, these are temporary containers for the data we work with. For example, if we want to add two numbers, then these two numbers are first put into two such registers and only then can the processor add them together. If a loop is executed, the loop counter is also stored in such a register.

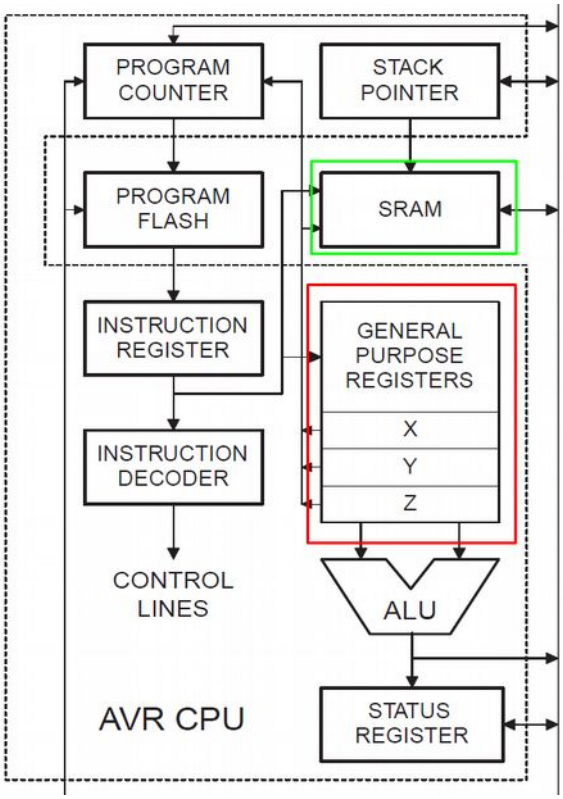

Figure 6 – the registers of the ATmega16A microcontroller

They can be seen in the figure as GENERAL PURPOSE REGISTERS. It is not to be confused with the concept of RAM, which is a memory area to be treated as a larger block. From this, the variables that are just needed are copied to the registers.

MEMORY1

We can see the memory map of the CPU here. We can see all available memory areas, in raw form, encoded in hexadecimal, and ASCII characters at the end of the rows.

PROCESSOR VIEW

This view provides access tothe contents of registers that are closely related to the processor execution unit. These are the following:

- Program Counter
- Stack pointer

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).

- X register
- Y register
- Z register
- Status register
- and finally, the 32 general purpose registers

I/O VIEW

This view will be most useful to us. Here, we can examine and modify the state of all peripherals. For example, you can easily determine if something has been configured incorrectly.

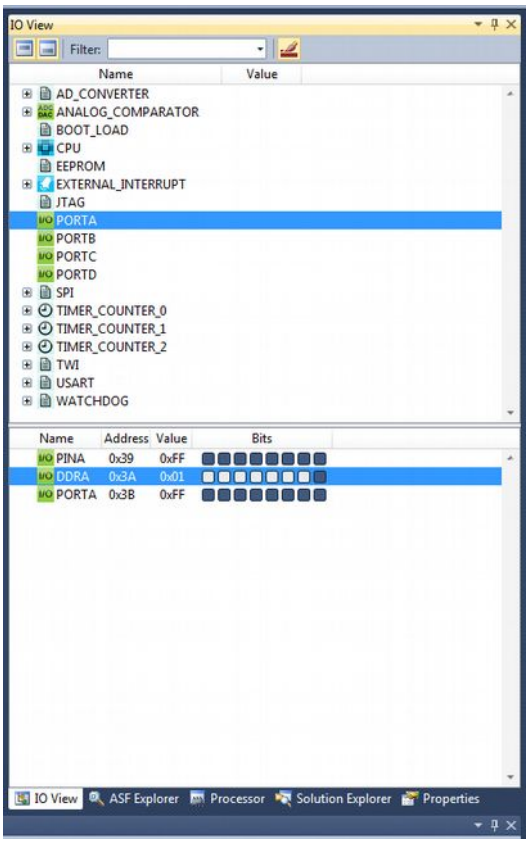

Figure 7 – I/O View panel

If PORTA is selected, the registers that belongs to port A will appear below. These are PINA, DDRA, and PORTA.

The PINA register stores the logical value that the controller has read from the given pins. DDRA means that a specific pin is configured for output or input. In the PORTA register we can write what value we want to see on that leg if it is configured as output.

If you click on the rightmost bit of PORTA (the first square on the right), then we can see how the LED can be switched on and off on the breadboard. Be patient for our configuration to become valid, we have to wait for the programming tool to print the register contents, and then read the new ones. It is also an interesting game to try to pinch these pins with your finger and step one on the program. If we are lucky,

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).

we may experience some change in the value of the inputs. The reason for this is to look for the processor's electronics structure, and the fact that these pins are not connected to anywhere, so they are “floating”.

WHEN THINGS GET MORE COMPLICATED

Let's complete our little program now with a few lines to see even more debug functionality. To do this, open the *CE12_2_LED_counter* project. The purpose of our change is not to change the state of the LED in every loop. To do this we will need a variable, which we will call *counter*. The value of this variable will be changed between 0 and 9.

You may have been guessing what tricks we're going to make. Our new program increases the *counter* variable by one in each loop, and when its value is smaller than a predefined number, in the example let it be 4, the LED will be turned on, otherwise it will be turned off.

PWM

Practically we will implement PWM by software. But what is PWM? The three-letter magic abbreviation is correctly resolved to Pulse Width Modulation.

You will meet several times with PWM later in the curriculum, we will build an analog PWM stage and we will also learn how to create a PWM signal using the digital peripherals built into the microcontroller.

Pulse width modulation can, for example, affect the brightness of LEDs, the torque (and therefore the speed) of motors. We rely on the fact that real physical systems, devices (such as our eyes or an electric motor) cannot follow the changes of physical quantities as quickly as we can modify them with a microcontroller.

That's exactly what happened in the first example, your eyes smoothed out the change. This is why you see, for example, a picture moving on your monitor, but actually it shows a series of still images really fast.

PWM is widely used. The closest practical example is in your pocket, so you can adjust the brightness of the backlight of your cell phone’s display.

There is an intentional mistake in the program, which we will now discover using the debugging tool. First, let's see what we see on the breadboard. The LED does not light up, although we have written a program that flashes the LED, according to the best of our knowledge.

Turn on the line numbering in the *Text Editor/All Languages* tab of the Tools / Options window by clicking the Line Numbers check box.

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).

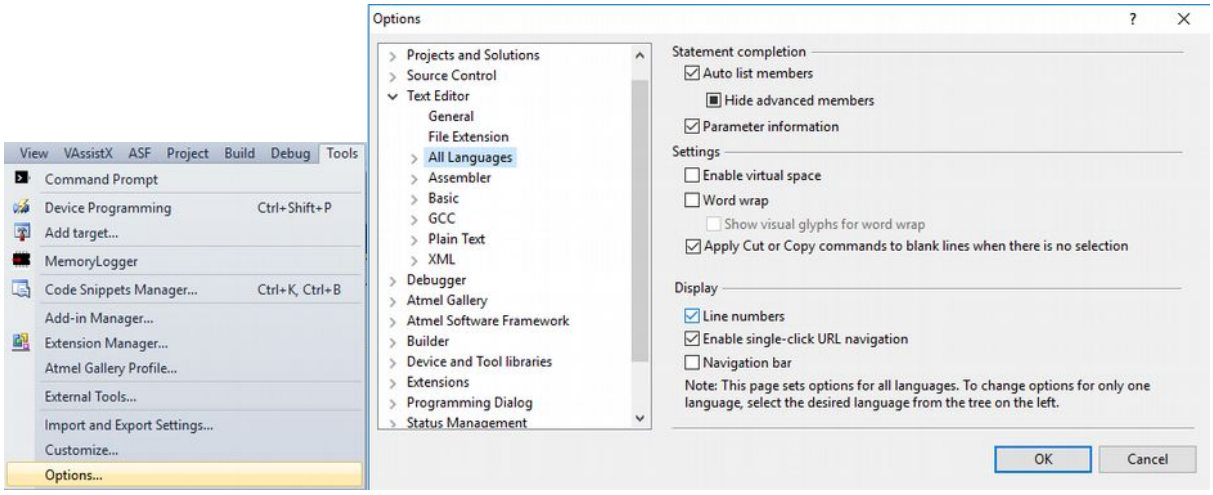

Figure 8 –Enable line numbering

1. Start the program in Debug mode with the "*Start Debugging" (F5)* button. Then the program starts running. The LED is still off.

2. In the next step, reset the processor using the "*Reset*" (Shift+F5) button. This is needed, because if we stop the processor at any time, it will be over a lot of program lines because it is much faster than us. We can even avoid this step by starting the debugging with the "*Start debugging and break*" (Alt+F5). Go ahead with the "*Step Over" (F10)* button. You can see that we have now jumped to line 14 where the `counter` is created.

```
10  int main(void)
11  {
12
13      //Continuously running counter
14      int counter = 0;
15      //Initializing ports
16      IOInit();
17
18      //Infinite loop
19      while (1)
20      {
```

Figure 9 – Creating the counter variable

3. The next step is the `IOInit()` function call on line 16.

4. If we step one more, we find ourselves on line 22. This line examines whether the value of the `counter` variable has reached 10, if not, it will increase it. We have to make it less than 10, so we don't have to increase it indefinitely. Step over it!

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).

```
18      //Infinite loop
19      while (1)
20      {
21          //Managing the counter
22          if(counter < 10)
23          {
24              counter++;
25          }
```

Figure 10 –Conditional test of the counter variable

5. Since the value of the `counter` variable was less than 10, it is increased as shown in line 24. Step over it!

```
18      //Infinite loop
19      while (1)
20      {
21          //Managing the counter
22          if(counter < 10)
23          {
24              counter++;
25          }
```

Figure 11 –Increasing the counter variable

6. Now we examine whether the `counter` variable is less than 4. If it is less, then we turn on the LED, if it is greater, then it will be turned off. Step over it!

```
27          //The state of the LED is based on the counter value
28          if(counter < 4)
29          {
30              //Turn ON the LED
31              PORTA = 0b0000001;
32          }
33          else
34          {
35              //Turn OFF the LED
36              PORTA = 0b0000000;
37          }
```

Figure 12 – Conditional test of the counter variable and then turning on the LED

7. The software stops on line 38, but it only indicates the end of the infinite loop, in which our program runs, step it over as well!

8. The LED is on, and we are on line 22 at the evaluation of the `counter` variable. In order find where things happen differently than our plan, it would be good to see the current value of the variables. To do this, move the mouse over a copy of the `counter`.

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).

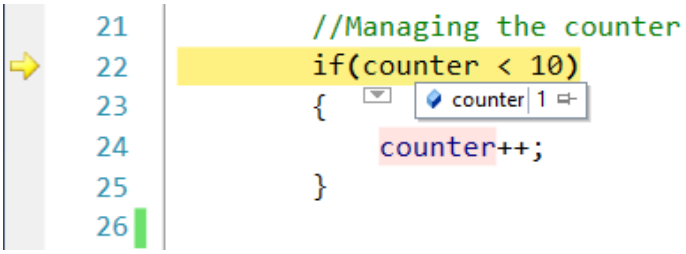

Figure 13 - visualization of the value of a variable

In this case, we can see the current value of the variable. If you click on the small drawing pin symbol, the small window will remain there.

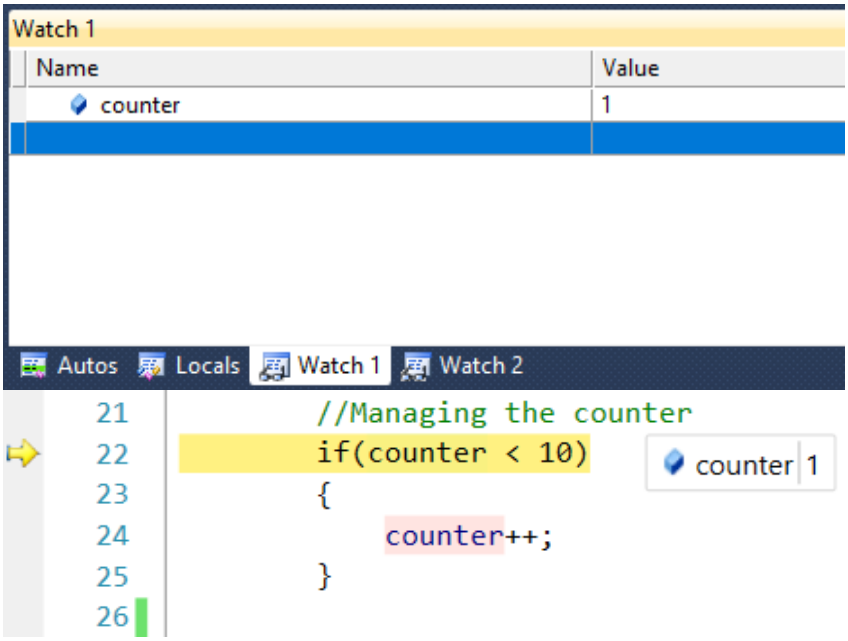

Figure 14 - Watch Window, continuous tracking of variables

Alternatively, you can display it in the *Watch Window* by typing the name of the variable in a free field of the *Name* column.

9. Now move the code by pressing the F10 button (it is more convenient than clicking the "*Step over*" button). We can see that the value of the`counter` is constantly increasing, and when it reaches four, the LED turns off, as the expression in line 28 becomes false, so we will jump to line 36.

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).

```
27          //The state of the LED is based on the counter value
28          if(counter < 4)
29          {
30              //Turn ON the LED
31              PORTA = 0b0000001;
32          }
33          else
34          {
35              //Turn OFF the LED
36              PORTA = 0b0000000;
37          }
38      }
```

Figure 15 – Turning off the LED

10. When the value of the counter reaches 10, an interesting phenomenon can be observed. The value of the`counter` changes no longer. This is because we only increase the value if the counter is less than 10. Yeah, but it's never going to be less than 4 such way. With this we found our fault!
11. Fix the code!
12. Don't look at the solution, write the fix!
13. Is it ready?
14. Have you tried it out?
15. Does it work?
16. If so, good job! If you couldn't figure out, don't worry, here's the solution:

```
21          //Managing the counter
22          if(counter < 10)
23          {
24              counter++;
25          }
26          else
27          {
28              counter = 0;
29          }
```

Figure 16 – The necessary fix

Indeed, the counter must be reset if it has reached its maximal desired value.

17. Try the fixed version (if it wasn't good until now), then experiment with other values between 0 and 9 instead of 4. We can see that the higher this value, the brighter the LED will light.

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 THINGS GET MUCH MORE COMPLICATED

Take one more knack into our program. Use the push button on PB0 to adjust the brightness of the LED. To do this, you need to initialize pin 0 on port B as input and connect a button to this pin. Our circuit on breadboard looks like this:

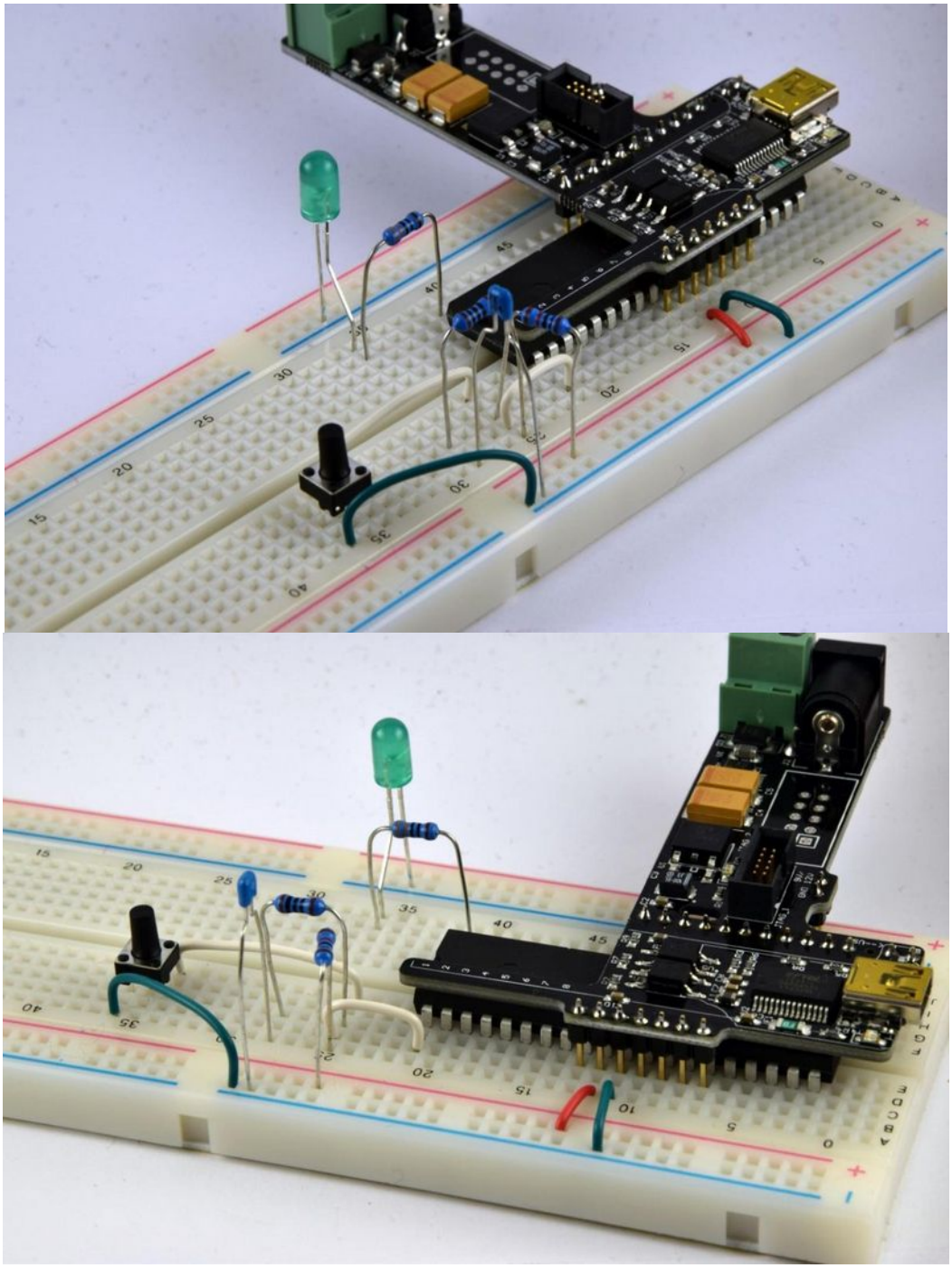

Figure 17 – Implementation of a circuit using a push-button on a breadboard

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).

Let's look at how this circuit works, as we may need a push button later too. The figure below shows the circuit schematic.

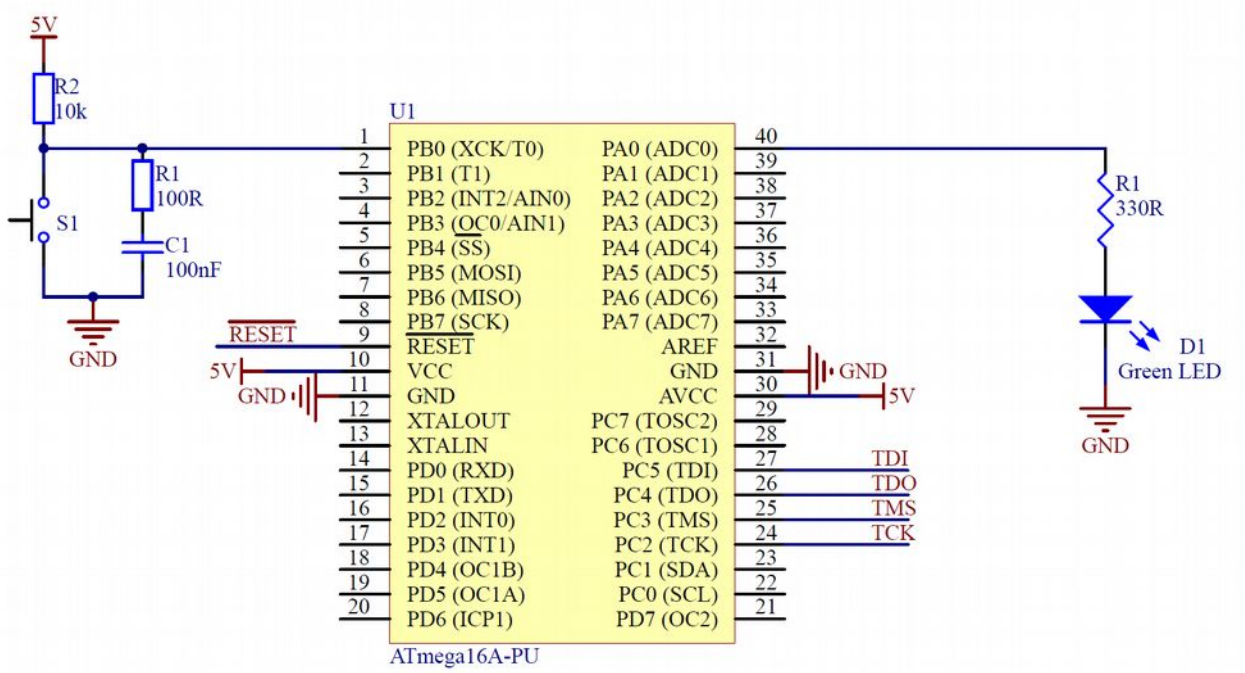

Figure 18 - LED circuit using push-button

The push-button circuit (on the left) consists of a pull-up resistor, a switch, and series resistor and capacitor, which implements a low-pass RC filter (we will see a bit below that what is this, and why we need it).

The most important of these is the pull-up resistor and the switch, let's look at this section now:

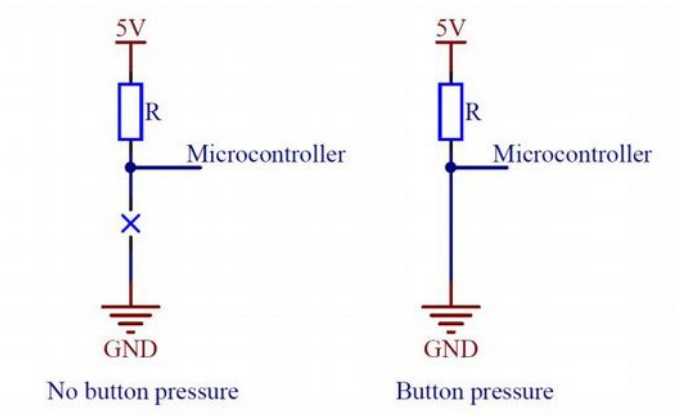

Figure 19 – The states of the push buttons

The circuit on the left shows the state when the button is not pressed while the circuit on the right shows the pressed state. The following figure shows the voltage change on the microcontroller pin.

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).

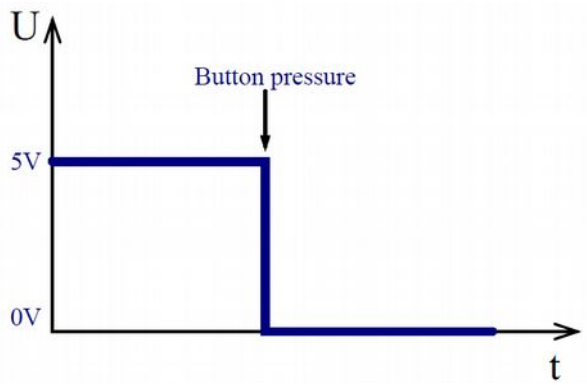

Figure 20 – The moment of button presses on the microcontroller pin

When are not pressing the button, there is virtually no current flowing through the resistor, so there is no voltage drop across it, that's why the supply voltage appears on the microcontroller pin; when the button is pressed, then current starts flowing, so the voltage drops to near 0 V. (Thought-provoking task: how to connect the button so that the voltage appears when the button is pressed?)

The remaining two components create a low-pass filter. This means that the resistor and capacitor are acting as a short-circuit for all fast-changing signals, but in the case of DC signals it is as if they weren't be there. This behaves as a filter, it lets the slowly changing signals through, but fast-changing signals are blocked. This is needed, because the internal mechanism of the push-button tends to bounce during key pressing. Bouncing is a rather unfavorable behavior or mechanical switches where the contacts close and release several times in rapid succession, which would register as more button presses on the microcontroller, even though we only wanted to press the button once. This phenomenon is shown in the figure below.

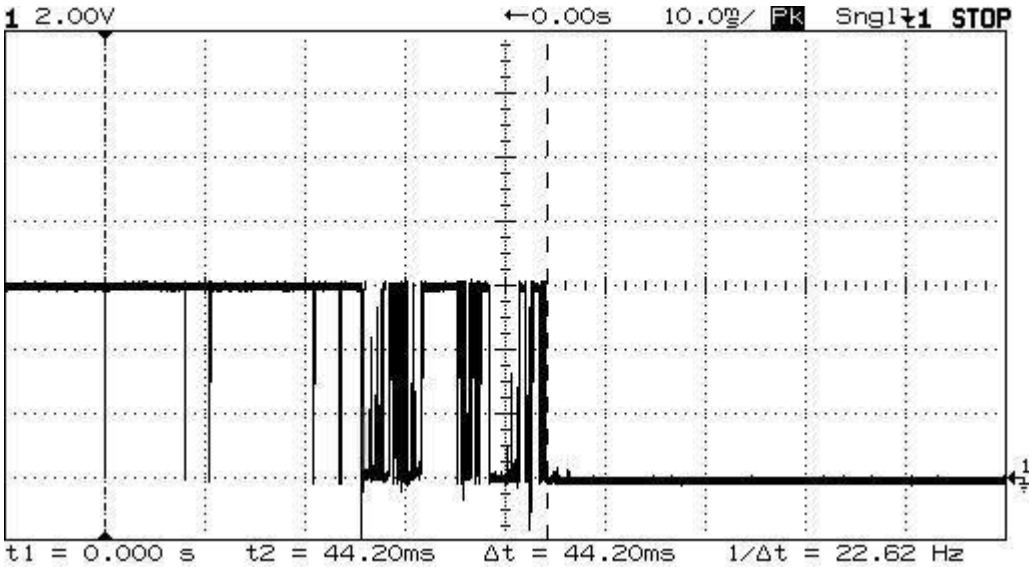

Figure 21 – The bouncing push-button

These so-called high-frequency components can be filtered out with our low-pass RC filter because they are fast-changing signals and filter will not let them through.

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).

Let's go back to our program, open the *CE12_3_LED_pushbutton* project. Previously, I promised that using this software, we will be able to adjust the brightness of the LED using the push-button. However, in practice it doesn't work.

What went wrong? Why is nothing happening when pressing the button? To find out we will learn about one of the most useful features of the debug environment. We can mark certain lines in our program, and execution will stop when the microcontroller reaches them. When this happens, we can view the current value of our variables and registers. Such markings are called breakpoints. We can place them by clicking on the gray bar on the left-hand side of the program. If the placement is successful, a red circle will appear indicating that the program will stop running when this line is reached. But where should we stop the program? Let's look at whether it notices the button press, so put a breakpoint in the section which tests it, that is, line 36.

```
33          //Part for handling the pushbutton
34          //The pullup resistor keeps the input high in default state
35          //When the button is pushed, it connects the input to ground, that must be detected
36          if((PINB & 0x01) == 0)
37          {
38              //Increment duty, when the button is pushed
39              duty++;
40              //Switch off the LED while the button is pushed (brightness is adjusted)
41              PORTA = 0b0000000;
42              //Wait while the button is pushed
43              //We need to wait, because we want to increase the duty variable only once when the button is pressed
44              while((PINB & 0x01) == 0);
45          }
```

Figure 22 – Setting a breakpoint

When pressing the button, the program should be stopped at the breakpoint, but unfortunately this is not the case. The first thing we can suspect here is the hardware failure, but if you have built the circuit as we have shown above, there shouldn't be such a mistake. Still, it is worth checking if everything is properly connected on the breadboard. The easiest way to do this is to measure the voltage using a multimeter on the microcontroller's pin to which the push-button is attached. When the button is not pressed, you should see a value close to 5 V on the multimeter, and close to 0 V while the button is pressed. If you do the measurements, you will see that the results are not what you'd expect. Perhaps the pin behaves differently than we would like, for example, it drives the point we want to measure as an output.

But where's the problem? Think about what happens to the ports and pins in the software. In the first step it is initialized, and then we look at its value in each loop. In this case, it is not so difficult to find out which part of the theory does not work properly in practice. Let's look at the initialization code snippet once more. The `DDRB = 0xFF` line may catch our eyes. This sets each bit of the DDRB register to 1, which means that each pin is configured as output. Here's the problem, write it over to `DDRB = 0x00`. Now all pins of port B are configured as inputs.

Let's restart the program by leaving the breakpoint in place. You can see that if you press the button now, the program will be stopped on the right line. You can view the current value of the duty variable by dragging the mouse over any occurrence of it in the code, or you can set its watch in *"Watch window"*. In this way, the correct operation of the program can be verified: when the value of the `duty` variable is less than the value of the`counter`, the LED is lit. As a further observation, it can be noted that the value of the `duty` variable will be increased by one at every button pressure. The interesting, first perhaps incorrect part comes after 9, because the value of the variable will be 10. This is not a problem, the next

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).

code snippet will take effect and will reset the value of the variable, this can be checked by stepping the program.

```
47          //Reseting duty variable if needed
48          if (duty >= 10)
49          {
50              duty = 0;
51          }
```

Figure 23 – Reset the counter of the push-button

SUMMARY

Hopefully, these few simple examples above have successfully highlighted software debugging processes and its tools. We will also implement more complex software later, and these tools will become useful companions for development, programming, and debugging.

Useful tips

Sometimes an error occurs because the different files generated during compilation are not from the same compilation. Do not work in a folder that synchronizes your computer for some cloud-based service (Google Drive, Dropbox, etc.), as it can re-download files that should have been overwritten, or somehow interfere with the compilation process. If you still get suspicious error messages, then use the Build / Clean Solution command to remove all files generated during the compilation, then recompile. In this case, the compilation process starts from scratch again the next time and the different references will be rebuilt.

There may be a problem with the optimization, which can be changed in the Project->Properties window. For us, the only unacceptable option besides turning optimization off is “Optimize for size -Os”.

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).