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

10 - IT Basics

Written by Olivér Pintér, Gábor Proksa

English translation by Xtalin Engineering Ltd.

Revised by Ádám Szabó

In this chapter of the curriculum, we get to know the basics of informatics, which we start with a little revision and later we learn the basics of the C programming language. By the end of this chapter, we will be able to create simple programs. These will not be executed in a microcontroller environment, but on a PC for simplicity.

NUMERAL SYSTEMS (REVISION)

First, we'll repeat a little what we learned about numeral systems. In everyday life, the decimal numeral system is widespread, we are learning it since primary school. The reason behind this is simple: we have ten fingers, so we can easily count and illustrate with them.

On the other hand, in the IT system the binary numeral system is widespread, because it is easy to be implemented there.

THE DECIMAL NUMERAL SYSTEM

The digits of the decimal numeral system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and their place-values are 1, 10, 100, 1000, 10000, ... i.e. the powers of 10. In mathematics, there is usually no special mark. There is no special prefix in language C, so we can use them in the usual way, we'll see that later.

THE BINARY NUMERAL SYSTEM

The numbers used for the binary numeral system are 0 and 1, and their place-values are 1, 2, 4, 8, 16, 32, 64, ... i.e. the powers of 2. In mathematics it is usually denoted by 2 in the lower index that the number is understood in binary numeral system. In language C, prefix "0b" serves this purpose. Let's look at an example of how to describe a binary number. In mathematics a binary number looks like 10111011001_2, while in C language `0b10111011001` will mean the same.

THE HEXADECIMAL NUMERAL SYSTEM

For the base 16 or hexadecimal numeral system, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (= 10), B (= 11), C (= 12), D (= 13), E (= 14), F (= 15) digits are used, and the place-values are 1, 16, 256, 4096, 65536, ... i.e. the powers of 16. In case of hexadecimal numeral system, the letters [A-F] are also considered to be digits, as we can simply and not confusingly represent them.

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).

In mathematics, it is usually denoted by 16 in the lower index that the number is understood in hexadecimal numeral system. In the language C, the prefix "0x" is intended to indicate hexadecimal numbers.

Let's look at an example here as well. In mathematics, we write a hexadecimal number $\texttt{f32ab6527a83c3}_{16}$, while in C language `0xf32ab6527a83c3` will mean the same, the lowercase and uppercase are equivalent for the description of the number.

CONVERSION BETWEEN NUMERAL SYSTEMS

Sometimes you may want to convert from one numeral system to another. Conversion from the decimal numeral system to another numeral system can also be done by the methods learned in primary school, that is, by recording the integer and the residue given after division by the base. It is iterated, i.e., repeated until the integer is 0. Afterwards to get the number in the new based numeral system, the residues should be read from bottom to top. Let's look at an example. We would like to convert 37 into binary numeral system. The procedure takes the following form as described above.

```
37| 1
18| 0
 9| 1
 4| 0
 2| 0
 1| 1
 0|
```

So, 37 can be described as $100101_{\ 2}$ in binary numeral system. In case of other-based numeral system, the same must be done. You can easily convert from binary numeral system to hexadecimal numeral system if you split the binary number into four groups and determine their value. Do you still remember how this happens? If not, feel free to read the "Revision of Basic Mathematics" chapter.

Reproduction to the decimal numeral system can also be done easily. To do this, we need to multiply the digits with the appropriate power, then add them at the end. For example, the $100101_{\ 2}$ binary number in the decimal numeral system is $1 \cdot 2^5 + 1 \cdot 2^2 + 1 \cdot 2^0 = 37$ or the $1B_{16}$ hexadecimal number converted into decimal numeral system is $1 \cdot 16^1 + 11(= B) \cdot 16^0 = 27$.

LOGICAL OPERATIONS (REVISION)

Let's briefly recall what we learned about logical operations, and Boolean algebra. We can execute operations with logical variables, the simplest of these is the NOT logical operation, which assigns the opposite to our logical variable. For example, $A = 1$has a negation, that is, $\bar{A} = 0$.

There are more types of operations that can be performed with two logical variables. For example, the "or" operation will result 1 being assigned to our logical variable if one of the variables contained 1. For example, $A = 1, B = 0$, then $A|B = 1$.

The "and" operation, on the other hand, only assigns 1 to our variable if both variables were set to 1. For example, $A = 1, B = 1$, then $A\&B = 1$

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 use the opposite or negatives for these operations, which means that when, for example, the result of the “and” operation would give zero, then the “no and” or “nand”, will give the negated of this, i.e. we get 1.

An important operation is the “XOR” or “exclusive or” operation, which assigns to our variable 1 if only one of the variables was 1. For example, $A = 1, B = 0$, then $A \otimes B = 1$

The logical operations can also be written in the form of a truth table, which you can see in the “Revision of Basic Mathematics” section.

REPRESENTATION OF NUMBERS

In informatics, numbers are typically represented in binary numeral system, which means that 0 and 1 are included in the description of a number. **Natural numbers** can be easily described by writing the number itself in binary form. But let's think about that if we want to describe an integer, such as -3 then, how can we do it because we don't have "-" sign only 0 and 1? Before we answer this, introduce some of the concepts used in information technology. In information technology, the smallest, or elementary unit of the data, is the **bit**, its name comes from the English **bi**nary digi**t** expression. It can represent two values 0 or 1, but we can couple multiple meaning to this. For example, it can logically be true or false, yes or no, but we can think of it as a sign for "+" or "-". Anything that can be described with two states can be matched. The next important concept is the byte. You've heard many times that 8 bits mean 1 byte, but why is it so important? Because we usually use it as the base unit of digital information, for example, typically the memory can be addressed with byte accuracy, it is common during data transmission to determine how many bytes can be sent or received in a second and our characters are also represented in bytes, that is, 8 bits.

How to write letters in binary

Our characters are also encoded as integers in the so-called ASCII table. This is an acronym composed of the initials of the American Standard Code for Information Interchange. The ASCII table contains English alphabet letters, numbers, punctuation marks, and control characters. These are 7-bit unsigned integers, that is, a set of 128 characters. For example, the code for the 'a' is the decimal 97.

To support different languages, where different characters from special English alphabet are present, this character set is expanded to 8 bits, and that is how we are able to write letters in Hungarian.

The sign problem for **integers** is solved if we use a so-called sign bit. This means that if the first bit of our binary number is 0 then our number is positive, if 1 then it’s negative. Let's look at an example: *0b00001001* has a sign bit and data bits. The sign bit is 0, so it will be a positive number, followed by the *0001001* bit sequence, which is 9, so the bit sequence *0b00001001* denotes 9. Let's look at the number *0b10001001*, acting like the previous one, we can see that it will mean -9.

Two zeros? Complement!

In the previous description mode, the so-called negative zero will also be included in the range of values, and the negative and the positive range will not be separated completely. A bit more complicated description mode is the two’s complement. If our number is a positive integer, then it is the same as above.

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).

However, if it is negative, it will look in two's complement that we take the opposite of the bits of its positive equivalent and we add one. This method eliminates the negative zero and completely separates the positive and negative ranges from each other. In addition, with this method addition has traced back to subtraction, so less complicated ALUs (arithmetic logic unit, it is the microcontroller's logic unit) can be designed. That is why this is the most basic form of representation so far. Let's look at how it looks the -9 in two's complement. 9 is described by bit series *0b00001001*, taking the opposite of each bit, then we obtain the bit sequence *0b11110110*. Add to this 1 more and we get the -9 , which will be *0b11110111*.

What if we want to describe a fraction, how could we do that? Fractions can be described with the so-called **fixed-point arithmetic**. This means that the place of the decimal (or much rather binary, because we are in binary numeral system now) point is predefined. For example, we would like to describe 9.5 with a fixed-point arithmetic so that it consists of 1-bit fraction and 4 bits of the integer part. We can do this by taking the integer, that is, 9. We've seen it appear as 1001. After that, I'll also look at the fraction of 0.5, which is the minus first power of 2. So, I can represent 9.5 as *0b10011*, where we know that the last bit is the fraction because we fixed it in advance. If we wanted to describe the same number with 2-bit fraction part, what would we get? The correct solution is *0b100110*, because the last two bits represent the fraction, but we only need the minus first power of 2, not the minus second power of 2, so it is replaced by 0.

Moving on, what if we can't describe the number that we want to represent as a fraction, only as a real number. The representation of **real numbers** is made possible by **floating-point arithmetic**. Thus, we are able to cover a much broader range than before. The name of the arithmetic came from the fact that the decimal point is not in a fixed place, but "floats", and it can get to anywhere. Storing our number in the following format:

$$\text{valuable digit} \bullet \text{base}^{\text{exponent}}$$

This means that our valuable digit is scaled with the help of an exponent, where in the basic information technology it is typically 2. Let's look at an example here, now choose 10 for the sake of better understanding and describe 300 with different exponents. If I choose the exponent for 1 then it would look like: $30 \cdot 10^1$, and if we choose it for 2 then $3 \cdot 10^2$.

ENDIANNESS (BYTE ORDER)

We have learned that our data is stored in bytes in the memory. The question may arise that how we store more bytes of data. The byte order or endianness will determine where the most significant byte will be placed (most significant byte, shortened MSB). There are basically two common representations, big-endian and little-endian. In the case of big-endian, the big front principle prevails, i.e. the MSB will be stored at the lowest memory address, while in the case of little-endian, the MSB will be stored on the highest memory address.

For example, the hexadecimal written 16-bit number shall be *0x1A2B*. We would like to store this number in the memory from address *100*, because we can address our memory by byte, but to store this number we will need two addresses. In case of big-endian *0x1A* will be stored at address *100* and *0x2B* at address *101*. In case of little-endian, it will be exactly reversed, at address *100* it will be *0x2B* and MSB will go to the higher address, that is, it will go to address 101. For example, the importance of endianness can be

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).

encountered when the microcontroller communicates with an external device, as the data should be sent in a byte order as required by the current communication protocol.

The endianness war

It should be noted that it is not really possible to decide which one is better, there is no significant advantage that would favor one. Perhaps the little-endian is predominant in the world because Intel's x86-based processors prefer this byte order.

INTRODUCTION TO PROGRAMMING

When we get a task, we usually only know what the machine should do. The way, how the machine does this, is what we have to figure out, design it and tell it all. But how do we tell what to do?

The machines must have a so-called instruction set, which is the set of operations that he knows. We give him instructions through his own machine language. Because machines see the world in binary, it means a series of 1 and 0. A series of 0s and 1s implementing a program is called machine code. It would be quite complicated for the human eye and it would be difficult for us to work efficiently (not everyone is capable of that, what is seen in the Matrix movie).

It is slightly more readable for the human eye if the instructions are given in text form. In this form, we can give instructions in the "Assembly language", such as the "ADD" instruction, which means that something has to be added. The words in the codes written in the assembly language are compiled by the assembler (the software that generates the code that can be interpreted by the machines), which means that the compiler replaces the words for bit sequences. It is obvious that neither this would be too convenient for us, because also simpler tasks were able to implement only with long programs, which would be hard to understand, and they would consist of very few small steps.

Instead of the previous ones, we will learn the C language, which is widely used because of its generally usable nature, or because this language is still sufficiently low-level (close to the hardware) to be suitable for us. Programs written in the C language will be compiled to Assembly language and then machine codes are made from it. You can read about the machine code and the compilation process in more detail in a later section of the curriculum.

Before getting to know the C language, it's important to clarify some concepts. The **algorithm** is the solution for a task, which includes the instructions to be performed in succession. For example, the sandwich making algorithm could be:

1. Take the bread!
2. Cut a slice from it!
3. Spread butter on the bread!
4. Put salami on it!
5. Put cheese on it!
6. You're ready!

This simple example shows that the sequence of instructions makes the algorithm and that after a certain step we can prepare our sandwich.

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).

Declarative programming languages

As described above, a task is solved by finding the solution by us and passing it step by step to the machine that will execute it all the way. This is called imperative programming language. C is also an imperative language.

There are also declarative programming languages. Unlike the previous one, in this case we have to specify the correct solutions, but the solution will be invented by the machine itself. Thus, a task can be programmed more easily, but its range of workability is much narrower than for imperative languages. For example, such a declarative language is typically the language used in the databases, where we just say what data we are looking for and then the program will figure out how to solve it.

The next important concept is the **variable**. We can give the variables a name that can be used to find them later, we can refer to them and store values in them. When used, the stored value can be read or replaced with the new one.

Generally, our variables have a **type** that represents their range of values and operations that can be executed on them. For example, the integers where the range of value includes the numbers -1, 2, 6, and the operations that can be executed with them, such as addition and subtraction.

Like all languages, programming languages also have linguistic elements and rules or syntax (structure) that we need to learn in the same way as when we learn English. Fortunately, the development environments in which the codes are usually made are to our help and indicate when we have made a syntactic error and until we have an error in our code we cannot compile it into a language that the machine can understand.

THE C PROGRAMMING LANGUAGE

We have arrived to the point where we can start to learn a programming language. The C language has a static type system, which means that we have to say the type of all the variables we want to use.

So, when you create a variable, you have to say its type, its name, and then you can give it an initial value, in other words initialize it. Initialization is not mandatory. In the example, an int (integer briefly, the different types will be discussed later in detail) variable will be created called " `a`" and another with the name `apple`, and we will also give them an initial value. The semicolon at the end of the line is important, indicating the end of an instruction.

```
int a = 3;

int apple = -1;
```

In this example, the type "int" is also a **keyword**. We can use keywords to refer to different language elements. It is worth to choose a name for the variable that refers to the purpose for which it was created. For example, it is advisable to create a variable, which contains the length of the circle radius, such as `int radius` or `int R`, so we will definitely know later what it means.

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).

BASIC TYPES

Let's look at the basic types in C now. In the previous example we have already seen the `int` type. Its size is typically 32 bits (on a PC), which means that its range of values spreads from -2^{31} to $2^{31}-1$. Our variables are signed by default (this is also a keyword), but if we want to create them without a sign, we must put the `unsigned` keyword in front of the type. Then its range of value spreads from 0 to $2^{32}-1$. It is created as follows.

```
unsigned int a = 102;
```

The integer has also a shorter, 16-bit and longer 64-bit version, the keyword to the former is `short`, while to the latter is `long`.

The next important type is `char`, which is used to store characters. This is an 8-bit variable, so its range of values can be from -128 to 127 in the case of a sign, while in the case of unsigned from 0 to 255 (`unsigned char`). Characters can be specified between ‘‘ marks, such as:

```
char letter = 'a';
```

As we have seen, these types are fixed-point values. For floating-point arithmetic (called fractions in computer science), `float` and `double` types are used. The `float` has an accuracy of 6 decimal places, while the `double` (double precision) has 15 decimal places. Of course, the need for storing a `double` is accordingly higher, 64 bits, while the `float` type is stored at 32 bits. They are created as follows:

```
float perimeter = 25.7;
```

```
double area = 12.342;
```

The following table sums up the basic types in the C language.

Datatype:	Range of values:	Size:	Accuracy (digit):
char	$-128 \ldots 127$	8 bits	
unsigned char	$0 \ldots 255$	8 bits	
int	$-2^{31} \ldots 2^{31}-1$	32 bits	
unsigned int	$0 \ldots 2^{32}-1$	32 bits	
short int	$-32\ 768 \ldots 32\ 767$	16 bits	
unsigned short int	$0 \ldots 65\ 535$	16 bits	
long int	$-2^{63} \ldots 2^{63}-1$	64 bits	

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).

unsigned long int	$0 \ldots 2^{64} - 1$	64 bits	
float	$-3.4 \cdot 10^{38} \ldots 3.4 \cdot 10^{38}$	32 bits	6
double	$-1.7 \cdot 10^{308} \ldots 1.7 \cdot 10^{308}$	64 bits	15

Of course, there are other types, but to use them, we need to assign an external file to our program. For example, the `bool` type, which contains *true* and *false* in its range of values. To use them, we need to assign the `stdbool.h` file to our program. For logical variables it is recommended to use this type.

Further on, you can find types like`uint8_t`. Don't be frightened of this, it just means that it is an unsigned (u = unsigned) 8-bit integer variable. In many cases, we do not need the 32-bit coverage range, only a fraction of this is enough, so we can handle our memory efficiently, which is very significant, especially in case of a microcontroller.

What if, for example, I want to store a lot of data related to each other? Create a variable named for each one? This would be quite inconvenient, that's why we have arrays. You can store multiple values of the same type within a single array. We need to say how many elements it can store at most to create them, such as `int area[3];` so our variable called `area` will be able to store 3 integer values. We can refer to the individual elements with their index, i.e. their serial number. The first element in our array is the element with index 0 (usually the first element in the IT is assigned to the index 0), while the 3rd element has the index 2.

```
int number[3];
number[0] = 12;
number[1] = -2;
number[2] = 4;

We can give the initial value this way too:
int number[3] = {12, -2, 4};
```

We have seen how we can store characters, but it would be inconvenient for a text to be given by character. Therefore, we can easily store texts in **strings**. The strings resemble the arrays, basically the characters are stored in an array, except that the last element of the strings will always be **'\0'**, which will indicate the end of the string. We can create a string like the this:

```
char text[] = "Hello";
This is equivalent to:
char text[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

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 not want to modify, change the value of a variable, so we want to make it **constant**. We have the possibility, with the `const` keyword we can do this when creating variable, for example:

```
const float PI = 3.14;
```

In this case, it is important to give a value to it when creating a constant, because we cannot do that later.

C compilers also have an optimization function. So, where they can, they optimize our code, simplify it, and make it more effective. We can choose the degree of optimization ourselves, or we can even turn it off.

Optimization sometimes works against us, for example, after turning on optimization, a code that has worked before suddenly does not behave as expected. The compiler, and thus the optimization, only works in one file at a time. One of the easiest example of this is that if a variable cannot be changed in a file, then the compiler replaces it with a constant so it optimizes the code. This, in turn, results an incorrect operation if the value of the variable can be changed in another file (/ translation unit). In such cases, we need to indicate this to the compiler, which we can do with the `volatile` keyword before the variable type.

OPERATORS

We can describe an expression (which is made up of operator(s) and operand(s)) by operators or operational marks. The following table shows some examples:

Expression:	Value:	Explanation:
3+2	5	3 and 2 are the operands, and the operator is the addition
2*(3+1)	8	Similarly, to the previous ones, 2, 3, and 1 are the operands, and *, + and () are the operators. Addition, and then multiplication.
2<5	True	Logical expression, comparison with an operator. It is true, because 2 is smaller than 5.

There are many types of operators, and between them there is order of operations, just like in mathematics, with this is called operator precedence. The following table shows operands of C language in precedence order. Don't be frightened if it is not clear for the first time, in time when you start to get to know and use C, you will understand what it means. You can see here the important operators used in the later chapters of the curriculum.

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).

Operator precedence	Operator	Description	Associativity
1	**++ --**	**Suffix/postfix increment decrement**	Left to right
	()	**Function call**	
	[]	Subscript	
	.	Structure and union reference	
	->	Structure and union dereference	
	(*type*){*list*}	Compound literal(C99)	
2	**++ --**	**Prefix increment decrement**	Right to left
	+ -	**Unary plus and minus**	
	! ~	**Logical NOT and bitwise NOT**	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Sizeof	
	_Alignof	Alignment requirement(since C11)	
3	*** / %**	**Multiplication, division, residue**	Left to right
4	**+ -**	**Addiction and subtraction**	
5	**<< >>**	**Bitwise left and right shift**	
6	**< <=**	**< and ≤ operator for relational operations**	
	> >=	**> and ≥ operator for relational operations**	
7	**== !=**	**= and ≠ operator for relational operations**	
8	**&**	**Bitwise AND**	
9	**^**	**Bitwise XOR (Exclusive OR)**	
10	**|**	**Bitwise OR**	
11	**&&**	**Logical AND**	
12	**||**	**Logical OR**	

13	?:	Conditional operator
		
		expression ? true : false
		
		if (expression)
		  true;
		else
		  false;
			Right to left

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).

14	**=**	**Basic assignment**	Right to left
	+= -=	**Addition assignment, subtraction assignment**	
	*= /= %=	Multiplication, Division, and Modulo assignment	
	<<= >>=	Bitwise left or right shift assignment	
	&= ^= |=	Bitwise AND, XOR, or OR assignment	
15	,	**comma**	Left to right

INSTRUCTIONS OF THE C LANGUAGE

We've already learned a lot of instructions in the past, just think about when we created a variable that was also a kind of instruction, or the terms and expressions we looked at. The" ; " or semicolon serves to terminate the instructions, but if it is alone, it indicates an empty instruction.

Let's look at other elements now. Several instructions can be placed in an instruction block, which is bounded by braces {}. This instruction block will thus essentially mean a complex instruction and look like this.

```
{
      instruction1;
      instruction2;
      …
      instructionN;
}
```

Let's look at a specific example in which we create three variables and then we initialize them.

```
{  //begin of the instruction block
     int a,b,c;  //creation of three integer variable
     a = 2;      //we initialize variable "a"
     b = 3;      //we initialize variable "b"
     c = a*b;    //we multiply "a" by "b", store result in "c"
   //so in variable ”c” 6 will be stored, which is also an assignment
}  //end of the instruction block
```

We can add comments to our code that helps understanding it. There are two ways to mark the comments. After the // sign, everything we write on the same line will be a comment, while text written between /* and */ may span multiple lines and all of it will be a comment. Comments are not included in the machine code; they only provide information to us. The compiler simply ignores them during the compilation process.

Branches

Based on what we know so far, the instructions are executed one after another in order, this is called sequential execution.

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).

But what if we execute one or more instructions only some of the time? **Branches** or conditional branches provide a solution to this. Several of these statements are available.

The first kind is the `if()` branch. It is used if you want to set the execution of an instruction or instruction block to a condition. The syntax looks like this:

```
if(condition)
{
      instruction;
}
```

This means that the instruction will only be executed if the condition in the argument of the `if()` statement is met, otherwise this entire block will not be executed. The following is an example in C where we enter the `if()` branch as our condition is be met.

```
int number = 2;  //a variable created for the example.

// the start of the if() branch
if(number < 3)   // if the value of "number" is less than 3, we enter
                 // the branch (2 < 3 is true)
{
      number = number + 5;  //we add 5 to the "number"
}
```

The value stored in the `number` variable is less than 3, so the instruction in our branch will be executed and we will add 5 to our `number` variable. Its value will be 7 at the end of the branch. For example, if the value of our `number` variable would have been 4 initially, then our condition would not have been met and we would have skipped the branch, and not increase the variable by 5. Those who are proficient in mathematics, have certainly noticed that the line in the conditional block makes no sense from a mathematical point of view, because something cannot be equal something plus 5. In the course of programming, we have to set aside a bit of mathematics and interpret the equal sign as an assignment instruction. On the left side of the equal sign is where we want to store the result of the expression on the right side - this is now the `number` variable, while the `number` on the right indicates the current value of the variable. The instruction can be written as "taking the current value of the `number` variable and adding five (right side), then store the result to the `number` variable (left side)". Thus, the result of the instruction is that the value of the variable has been increased by five.

The if-else structure is essentially the completion to the previously branch type. In addition to the `if` branch, there is now another branch called `else`. The overall structure is as follows.

```
if(condition)
{
      instruction;
}
else
{
      instruction;
}
```

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).

Similarly, if the condition in the `if()` statement is met, the instructions on the `if` branch will be executed. Otherwise, the other branch will play a role, the `else` branch. The instruction block there will be executed. The following example written in C shows an extension of the previous one.

```
int number = 6; //a variable created for the example

// the start of if-else branching
if(number < 3)  //if the value of "number" is less than 3, we enter the
                // branch (now 6 < 3 is false)
{
      number = number + 5;  //we add 5 to the "number"
}
else // otherwise, if the “number” was not less than 3, we enter this
     // branch
{
      number = number - 2;  //we subtract 2 from the variable "number”
}
```

The value of our `number` variable has now been set to 6, so when we examine our condition, we get false, because 6 is larger than 3. Therefore, the instruction in the `else` branch will be executed. We will subtract 2 from our `number` variable, and in the end, we will have 6-2, i.e. 4 in our `number` variable, after our branch.

We might need more branching because, for example, we will have to do something depending on the value of a variable. The `switch-case` structure can be used for this. The general syntax is:

```
switch(integer value)
{
      case value1: instuctions;
      case value2: instuctions;
      case value3: instuctions;
      ...
      case valueN: instuctions;
      default: instuctions;
}
```

In operation, we will enter a `case` branch corresponding the integer value placed in the `switch()`. It is important that the individual `case` branches are not followed by braces. You can also specify a so-called `default` branch that you may use if the value passed to the `switch()` does not match the value in any `case`. Use of the `default` branch is not obligatory.

Let's look at the following example in C to understand the operation of the switch structure.

```
int number = 0; // a variable created for the example

// the start of switch-case
int jump = 2;    // we will jump to the branch corresponding to this
                 // variable

switch(jump)     // we jump to the branch corresponding to the value 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 variable ”jump”
{
      case 1:
            number = number - 5; //do this if the value of “jump” is 1
      case 2:
            number = number + 2; //do this if the value of “jump” is 2
      case 3:
            number = number * 3; //do this if the value of “jump” is 3
}
```

In the `switch()` statement, the variable `jump` is read, which means that we will enter the branch corresponding to the value of the `jump`. In this case, we jump to `case 2`, because the value of the `jump` variable is 2. In the branch of `case 2`, we add 2 to our `number` variable, so its value changes to 2, then we move on to the next branches. In `case 3` we multiply our `number` variable by 3, so at the end of the `switch()` the value of our `number` variable will be 6 already. So, essentially, the `case 1` branch was skipped in our example. It is a little difficult to see through that not only the `case 2` branch is running, but also all the others that are after it. Let's think about it, this would result a hardly transparent code and complicated logic for many branches. Usually, we just want a single branch to be executed.

You can do this by using the `break` keyword. When we reach this so-called **branching instruction**, our branching is interrupted, and we see jump out of the `switch`. This keyword can of course also be used for the structures described above and later, although it should be noted that its use will result more difficult transparency. Let's look at the previous example, but with the known `break` instruction and a `default` branch.

```
int number = 0; // a variable created for the example

// the start of switch-case
int jump = 2;    // we will jump to the branch based this variable
switch(jump)     // we jump to the branch corresponding to the value
                 // of the variable ”jump”
{
      case 1:
            number = number - 5; //do this if the value of “jump” is 1
            break;
       case 2:
            number = number + 2; //do this if the value of “jump” is 2
            break;               //we jump out of the switch ()
                                 //because of the break
      case 3:
            number = number * 3; //do this if the value of “jump” is 3
            break;
      default:
            number = 0;          // do this if the value of “jump” is
                                 // different from the previous ones
            break;
}
```

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).

Similarly, to before, we will move to the `case 2` branch and increase our `number` variable by 2. Then, as a result of the `break` instruction, we exit the `switch` structure, so the following branches will not be executed compared to the previous example. For example, if you select 4 for the initial value of the jump variable, then we move to the `default` branch, where the value of our number variable is reset.

Loops

Often, we want to execute an instruction block repeatedly. This may be a predetermined repetition or only while some condition is fulfilled. Loops are used for this purpose. The first one we look at is the `while()` loop. The syntax for it is:

```
while(condition)
{
      instruction;
}
```

Our loop will run and the instruction(s) in it again and again as long as the condition in the`while()` part is fulfilled. Let's look at a simple example implemented in C.

```
int x = 0; // a variable created for the example

// the start of the while loop
while(x < 2)  //We examine the condition, as long as it is true, we
enter the loop again and again
{           //the begin of the instruction block of the while loop
      x++;  //the variable "x" will be incremented by 1, we could
             //have written x=x+1 as well
}            //the end of the instruction block of the while loop
```

First, we create an integer called `x`, with an initial value of 0. The `while()`loop begins with the test condition, if that evaluates to true, then the instruction block is executed. The initial value of `x` is 0, so 0<2 is true, we enter the instruction block. Here, the value of `x` is increased by 1, so `x` becomes 1. After the end of the instruction block, we jump back to test the condition of the `while` loop. Remember, we learned that our loop runs as many times as the condition remains true. Now 1<2 is still true, so we re-enter the instruction block, increasing x, so its value is now 2. We then jump back to test the condition of the `while()` loop . The value of `x` is then 2, so the expression 2<2 is false, our program jumps to the part after the `while()` loop. We could see in this simple example, that the instructions inside the `while()` loop were executed 2 times in a row.

The `for()` loop, is a counter loop that does not differ in function from the `while()`loop, they can be converted into one another. The syntax of the `for()` structure: is:

```
for(initialization; condition; increment)
{
      instruction;
}
```

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).

During *initialization*, we give an initial value to a variable, and we will use it as a counter for our loop. The variable can be created here, it doesn't need to exist before the statement. In the middle is the *condition*, which means the same as before, if it evaluates to true, we enter into the loop again and again. The third part is the *increment*, here we implement the loop counter iteration. Let's look at the following simple example to illustrate what has been described.

```
int x[10];       // an array created for the example

//the start of the for loop
for(int i = 0; i < 10; i++) //the initialization of the loop counter,
                            //and the description of the condition and
                            //iteration
{
      x[i] = i * 7;  //assignment to the i-th element of "x" array
}                    //the end of the instruction block
```

For the sake of this example, we have created an array called `x`, with 10 elements. The we begin the `for()` loop. We first create the counter between the parentheses of the `for()`statement, this is called `i`, which is a common name of loop counters. Its initial value is set to 0 and then we specify the condition. Now our condition is `i<10`, that is, our loop will be executed until `i` is less than 10. The last member of the parentheses describes how to move our counter. This `i++` means that at the end of each loop, the value of our counter is increased by 1.

Let's look at it step-by-step: after initialization, `i` is 0, which fulfills the condition, so we give a value of the first element (which is at index 0) in the array `x`. This value is `i*7`, in this case 0*7=0. At the end of the instruction block of the `for()`loop, when all the instructions in it were executed, but before the next loop condition is checked, we will increase our counter by 1 (`i++`). We then re-evaluate our condition, but with the value of `i=1`. This is still less than ten, so our `for()` loop will be executed again, and then the counter will increase, so its value will be 2. The value of the second element (index 1) of the array `x` has been changed to 1*7. In the next condition test, 2<10 will be evaluated, so the value of `x[2]` will be 14. This will go up to `x[9]=9*7`. Then `i` will be 10 and will not enter the `for()`loop again. With this loop, we produced the multiples of 7 from 0 to 63 in array `x`.

We've already learned a branching instruction that was the `break`. Now we get to know another very important branching instruction, which is the `return`. With this, we are able to finish our program and return from our functions (we will see in a bit what functions are). There can be two ways: it can be placed on its own like this: `return;` or an expression can follow the keyword. For functions, for example, we can return the output of a function with it.

Functions

Functions in the programming have a similar meaning like in mathematics. They describe some kind of a relationship. Functions can also be called subroutines or methods. Usually, we assign a series of instructions to functions that we use several times during our program, making our code more transparent and efficient.

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 function is built up as follows:

```
return_type function_name(parameters)
{
      ...function_body...
      return return_value;
}
```

All functions have a name that we can use to reach them. The return type is placed before the function name which is the type of variable the function returns with. Additionally, functions have a parameter list, through which we can pass input variables. In the function body, there are the instructions placed, and every function terminates with the `return` instruction, which gives the function output back. Let's look at a simple example:

```
/*This function can add two numbers.
  We have two inputs, the first and the second number.
  As a result we expect an integer, which is the sum
*/
int add(int first_number, int second_number)
{
      //creating a variable called sum, where we will store the result
      int sum;

      //The instruction which implements the addition
      sum = first_number + second_number;

      //The end of our function (subroutine), which terminates with
returning the calculated result
      return sum;
}
```

The example implements a function for adding two numbers, which has two input parameters. These are two integer numbers that are added up in the function body. The result is stored in a variable, and then our function is terminated with the `return` statement, which returns the calculated sum, that is, the output of our function.

It may also can happen that there is no input parameter or no return value. The `void` type is used to indicate this. The `void` keyword means empty. We will look at an example of how we can use our functions later.

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).

STRUCTURE OF A C PROGRAM

Now we have learned about the elements of the C language, let's see how a C program looks like. In the programming language courses, the first program we see is the *"Hello world!"*, let's look at this now! On this small program, the structure of a general program can be well illustrated.

```
#include <stdio.h>
//Our main program, the entry point of our program
int main(void)
{
    /* It prints out the Hello world! text to the console */
    printf("Hello world!\n");
    //The end ouf our program is terminated with return, 0 indicates
that there were no errors
    return 0;
}
```

Let's look at it line-by-line. Using the `#include <name>` on the first line, we can insert and import pre-written parts of a program (such as `stdbool.h` for booleans mentioned earlier). Stdio stands for **ST**andar**D** **I**nput **O**utput. The stdio also includes the `printf()` function along with many others, which can be used to print characters to the standard output, usually a terminal screen. The `.h` tells you the extension of the file itself. The `.h` is the abbreviation of the header file. Such files contain prototypes of functions and constants. This is followed by the main program: `int main(void)`, which is the entry point for all C programs. It can be seen that this is essentially a function that will return an integer and has no input parameters. The function body is located between braces.

We provide our codes with comments, as we did in the examples above, and we have to write our comments among `/* */` signs if we want to write comments over lines, but if we want to write a comment in a given line, we can do it by placing `//` signs. These are meant for us who write the program, the compiler ignores these lines as if they were not there.

This is followed by the already mentioned `printf()` function, which has the text `"Hello world!\n"` within, which is given as a string. As a reminder, the `strings` are a series of characters. The `\n` means a new line. Besides `\n`, an important character is the `\r`, they usually coexist in a form of `\r\n` and their meaning: go to the beginning of the next line. These are called **control characters**. Finally, our program ends with `return 0;` which terminates running the program and 0 indicates that everything has been successfully completed. The result of the compiled and executed code is the *Hello world!*, which is printed to the terminal. You can try out the your C code at:

https://www.onlinegdb.com/online_c_compiler

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).

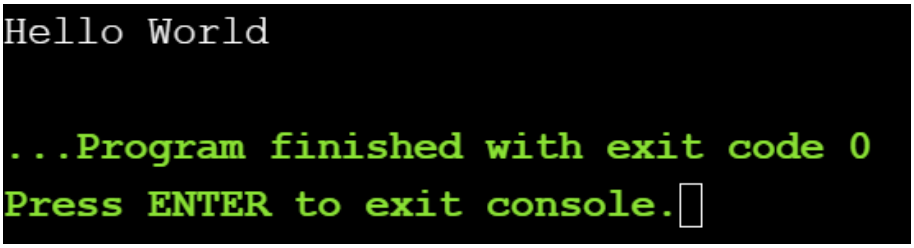

Figure 1 - Hello world!

You can try out this program if you want. You don't have to download anything to do this, because C compilers are available online. Search for "c online compiler" or use the one we have linked above.

Let's look at another simple program that adds two integers. This is very similar to the previous *"Hello world!"* program.

```c
#include <stdio.h>
//Our main program, the entry point to the program
int main(void)
{
    /* It prints the result of the operation to the terminal */
    printf("Result:%d\n", 5+3);
    //The end ouf our program is terminated with return, 0 indicates
that there were no errors
    return 0;
}
```

We see an obvious difference at `printf()`. With `printf()`, we can not only display simple text, but also the result of a mathematical expression. The %d will be replaced by the result of our expression. %d indicates that an int type value will be replaced. For floating point numbers we have to write `%f`, for characters %c, and for strings %s. The result of our program is as follows.

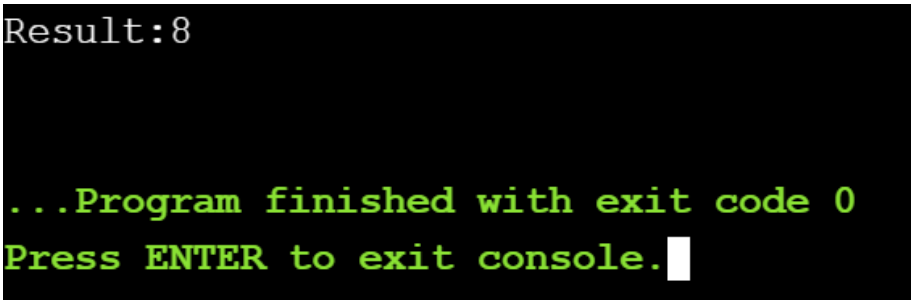

Figure 2 - output of a calculating program, that shows how to use printf()

Program arguments

Our main program may also have input parameters. Generally, our main function can be described in the following forms: int main (int argc, char * argv [], char * env []).

The first parameter is the number of arguments that you want to pass, and then the pointer to arguments and possibly environmental variables. The pointer specifies the address of the value stored in the memory Parameters can be skipped from behind, as we have seen in our examples.

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).

In the stdio directory there are many other useful functions beside *printf()*. We will now learn about two more functions that you will use in later sections. These are *getchar()* and *puts()* functions. Let's look at an example what they can be used for.

The `int getchar(void)` function can read a character from the standard input (such as from the terminal using a keyboard or file). The function returns with the scanned character and does not have an input parameter. The following simple program demonstrates its use.

```
#include <stdio.h>
// Our main program, the entry point to the program
int main(void)
{
      char c;         // We'll read a character into this variable

      printf("Press a button: "); // Prints the given text to terminal
      c = getchar();              // Function which scans a character
      printf("Input character: %c", c);      // Prints the value stored
                                             // in variable "c"
      // The end of our program is terminated with return,
      // 0 indicates that there were no errors
      return 0;
}
```

When we start, we create a variable called *c*, in which we will store the character scanned from the terminal. We use a text message to indicate to ourselves what to do, what the machine is waiting for. When our program reaches the *getchar* function, it will wait until a character then the *enter* key is pressed. After pressing *enter*, our program continues with the following command, which will also be a text message. As a control, we will print the just now scanned character, which will put an end to our program. The following two figures show the running of our program.

Figure 3 - Waiting for a character to be pressed

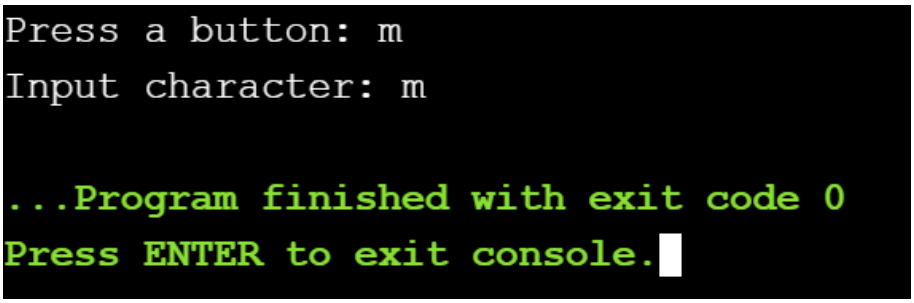

Figure 4 - Printing the scanned character successfully

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).

With the `int puts(const char *str)` function, we can send a string to the standard output (screen, file, or printer). The function expects a string as an input parameter. You may have noticed that if we only print something, then why is there a return value? In the return value we get information about the state of the function's run, whether it was successful or not. The following simple program illustrates the use of this function.

```
#include <stdio.h>
// Our main program, the entry point to the program
int main(void)
{
      char text[] = “I am a string!”; // Making a string

      puts(text);      // This function prints the given string
                       // to the terminal

      // The end ouf our program is terminated with return,
      // 0 indicates that there were no errors
      return 0;
}
```

We create a string called *text*, where we put the phrase *"I am a string!"*. This string is passed as a parameter to the *puts()* function, which prints the text we have given to the screen.

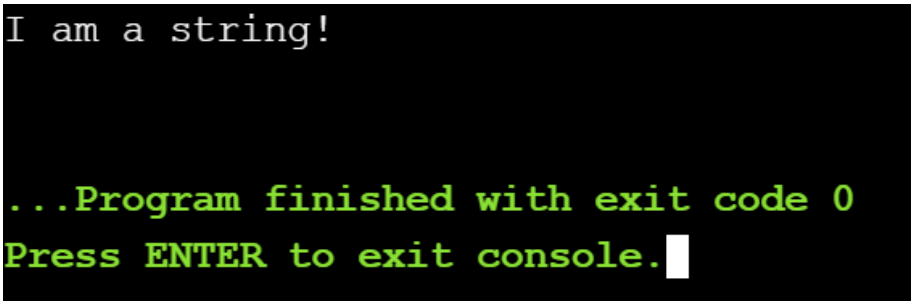

Figure 5 - Printing a string using the puts() functionl

If you remember, I promised that we will see how to use the function we wrote. Our last example will show you this. The functions we have created are located outside the `main` function. We have seen many examples of calling a function, let's think about `printf()`. The following code contains the previously presented `add()` function and its use.

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).

```c
#include <stdio.h>

// A function to add two numbers, which was previously created
int add(int first_number, int second_number)
{
      //creating a variable called sum, where we will store the result
      int sum;

      //The instruction which implements the addition
      sum = first_number + second_number;

      //The end of our function (subprogram), which terminates with
      //returning the calculated result
      return sum;
}

// Our main program, the entry point to the program
int main(void)
{
      int result; // We will store the output of the function here

      result = add(1, 4); //We call the “add” function with
                          //inputs 1 and 4
      printf("Result:%d\n", result); //We print the value
                                     //stored in “result”
      result = add(12, 21); // We call the “add” function with
                            //inputs 12 and 21
      printf("Result:%d\n", result); // We print the value
                                     //stored in “result”
      result = add(215, -15); // We call the “add” function with
                              //inputs 215 and -15
      printf("Result:%d\n", result); // We print the value
                                     // stored in “result”

      return 0;
}
```

When we look at the main program, we can see that our function receives two integers that correspond to the list of parameters. For example, in case of `add(1,4)`, the `first_number` will be replaced by 1, while the `second_number` will be replaced by 4. In each case, the result will be stored in a variable called `result`. It can be seen that we can call the function as many times as we want; we just need to refer to it and give it the right parameters. The output of the program will be the following:

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).

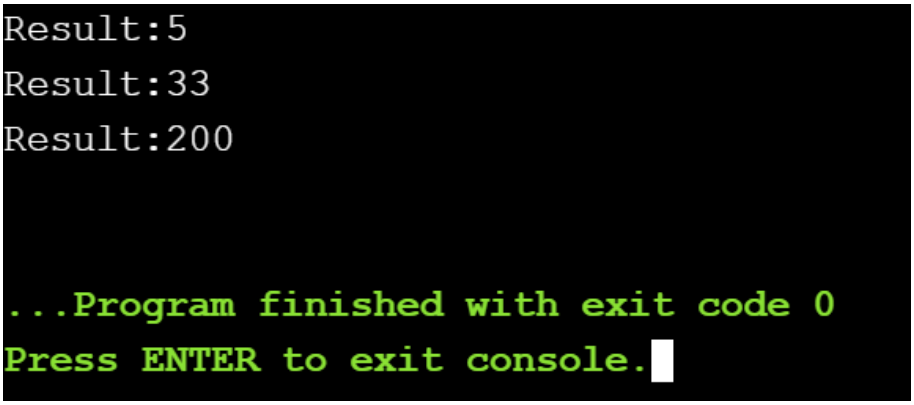

Figure 6 - Output of a program that shows how to use the add function

With this, we have also completed the understanding of the basics of programming and the C language, and have completed our first programs. Of course, the C language also has more complex elements that go beyond this curriculum, but after learning the basics and with enthusiasm, you can continue to learn the C language (and later even other languages).

SOFTWARE INTEGRATION

In a real, industrial environments, we can quickly reach a state where a person do cannot understand the whole system, and the code to be written is so big and complicated that we have to break it down into smaller parts, this is called modularization. Our goal is to extend the entire system to larger services and then to break them down to smaller and smaller features - modules. In this way, we can have our software modules that are already easily transparent to the system and can be developed and tested much easier. In order for these small modules to work together properly, the connection points and interactions between them must be defined, these are called interfaces. If they are properly defined and maintained, we not only win with the modularity to simply modify and test individual modules, but also to replace them and add new modules - new features - easily and the unnecessary ones can be removed from our system. This is a very important aspect in a large industrial environment, but of course we can also benefit from it, when coding.

For a more serious system, different software developers are responsible for developing the modules. The task of the software integrator (also called coordinator) is to maintain the interfaces and to transform and configure the modules into a unified system, while testing is done by others.

In most industries, software integration and configuration are supported by different standards. The maintenance and traceability of the source code is supported by various version control systems. The essence of version control is that we can restore or reproduce a particular state of the source code at any time. There are several possible implementations of this. Perhaps the easiest way is to put our code into a new folder every day with the date written on it. This can be a very effective solution for small systems or if only one developer works on it. But as our code becomes complicated or others are involved in the development, the situation becomes practically unmanageable. Therefore, it is more appropriate to use an advanced version control system. The essence of these is that the source code is stored on a central server, which is available to everyone, so that the most up-to-date code is always available.

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).

If you need to change it, you can recover the changed code at any stage of the development, so it is always available the latest state for everyone. Another great advantage of these systems is that they allow the creation of different development branches. This comes in handy when a function or module can be implemented in several ways and we want to know which one is the most appropriate solution. For example, if you want to try to handle a peripheral with an interrupt or polling (see later in the chapter about interrupts). In this case, we can create a branch for both options, leaving the original code intact, and we can write and try both solutions completely independently. In this situation, obviously the new date-named folder solution would make the organizing quite complicated. In contrast, there are plenty of usable version control systems to help keep tracking of this. There are also completely free ones (for example, SVN, Git), it is worth to try them out, and use them, because they make the work much easier.

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).