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

16 - Accurate Timing - Timers

Written by László Treszkai

English translation by Xtalin Engineering Ltd. and László Treszkai

Revised by Gábor Proksa

In this chapter of the curriculum, you can learn about the basics of timer peripheral programming that will benefit your electrical engineer or software developer career.

Introduction

In this chapter we will write the software for a digital clock in two ways: first without using the timer peripheral and then with it. Throughout the examples we did not provide complete code snippets, and we left some questions unanswered. Although you can find the complete working code examples on the website of the curriculum, we suggest that you try to write the missing parts yourself and think about the questions, in order to learn the most. (Which is why you are reading this curriculum in the first place, right?) We can pick up a cookbook and select a good-looking recipe, but just by reading them one cannot learn to cook.

Let's start in the Stone Age first: we have a processor and we want to build a digital clock that looks good for the next part of *Back to the Future*.

What Does a Digital Clock Software Look Like Without Using the Timer Peripheral?

What Does the Simplest Digital Clock Do?

Mine shows 17:23:42 right now. Oops, already 17:23:43, what's going on? You could say that a clock continuously displays the time in seconds from when it was turned on. In practice, this means that the value of a counter is increased every second, starting from 0 at midnight, and returning to 0 after 86,400 seconds (we say that the counter *overflows*). The value of this counter is displayed on the clock, usually in the "hour: minute: second" format.

On a "binary" clock, LEDs indicate the hours, minutes, and seconds as binary digits. For example, in Figure 1, the LEDs in the bottom row indicate the seconds: LEDs for 32, 8, and 2 are lit, which translates to 32 + 8 + 2 = 42 seconds. Can you tell the time from the display?

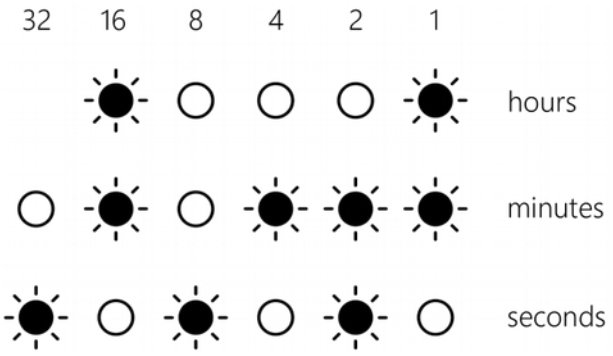


Figure 1 – Binary clock example

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).

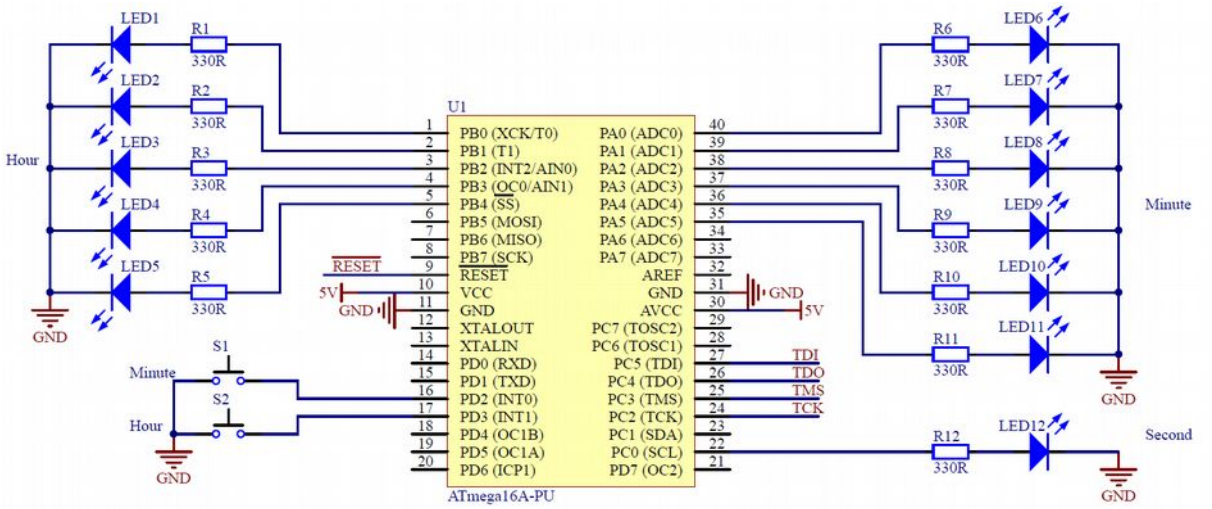
How can a Digital Clock Tell the Time?

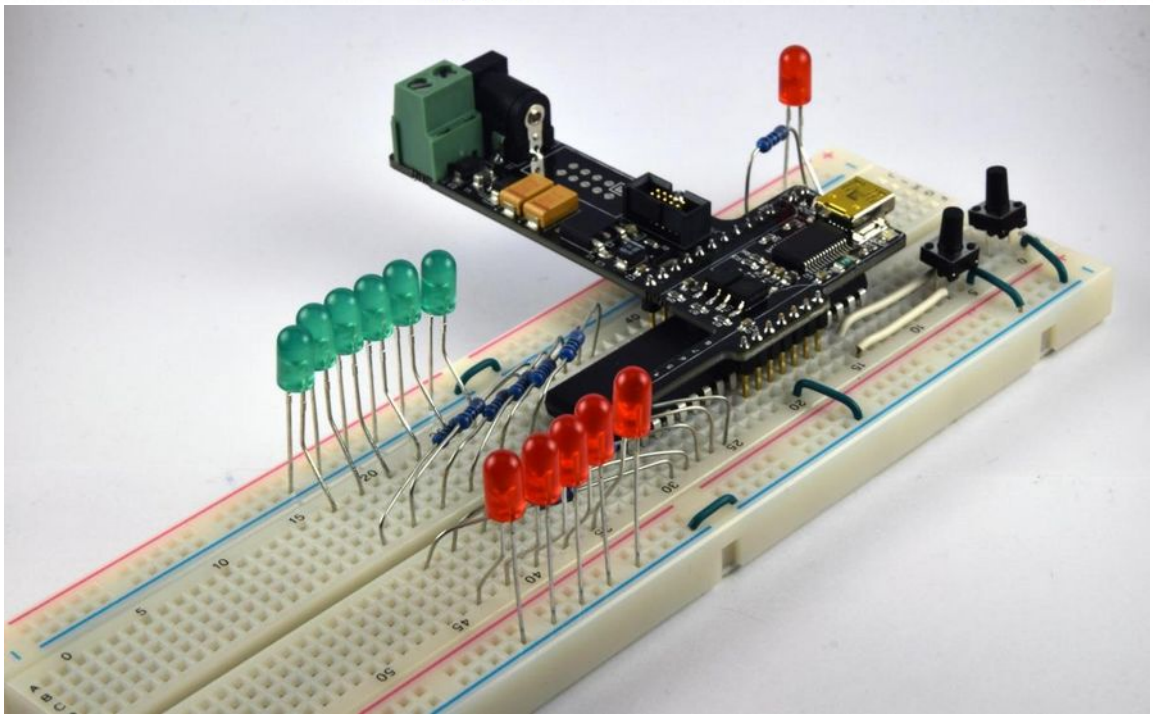
Usually it can't. It simply starts counting from 0:00:00 when you turn it on, and the time needs to be set manually. In the example code, we will solve this in the simplest possible way: one button to increase the minute counter and another to increase the hour value. This solution is far from perfect: it's instructive to think about how to make it easy to *use*, not easy to *develop*. Examine the user interface of the items around you, how do they solve similar problems?

What Should Our Gadget Look Like?

The LEDs, used as a light source, can be connected between the microcontroller pin and ground – i.e. the negative supply voltage – via a resistor connected in series. So if you want an LED to light up, you need to set the output of the microcontroller to high.

By default, the inputs are pulled up by the microcontroller's internal pull-up resistor and the push buttons will pull them down to the ground when pressed. Thus, pressing a button can be detected in software by reading a low value on the input.





Erasmus+

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

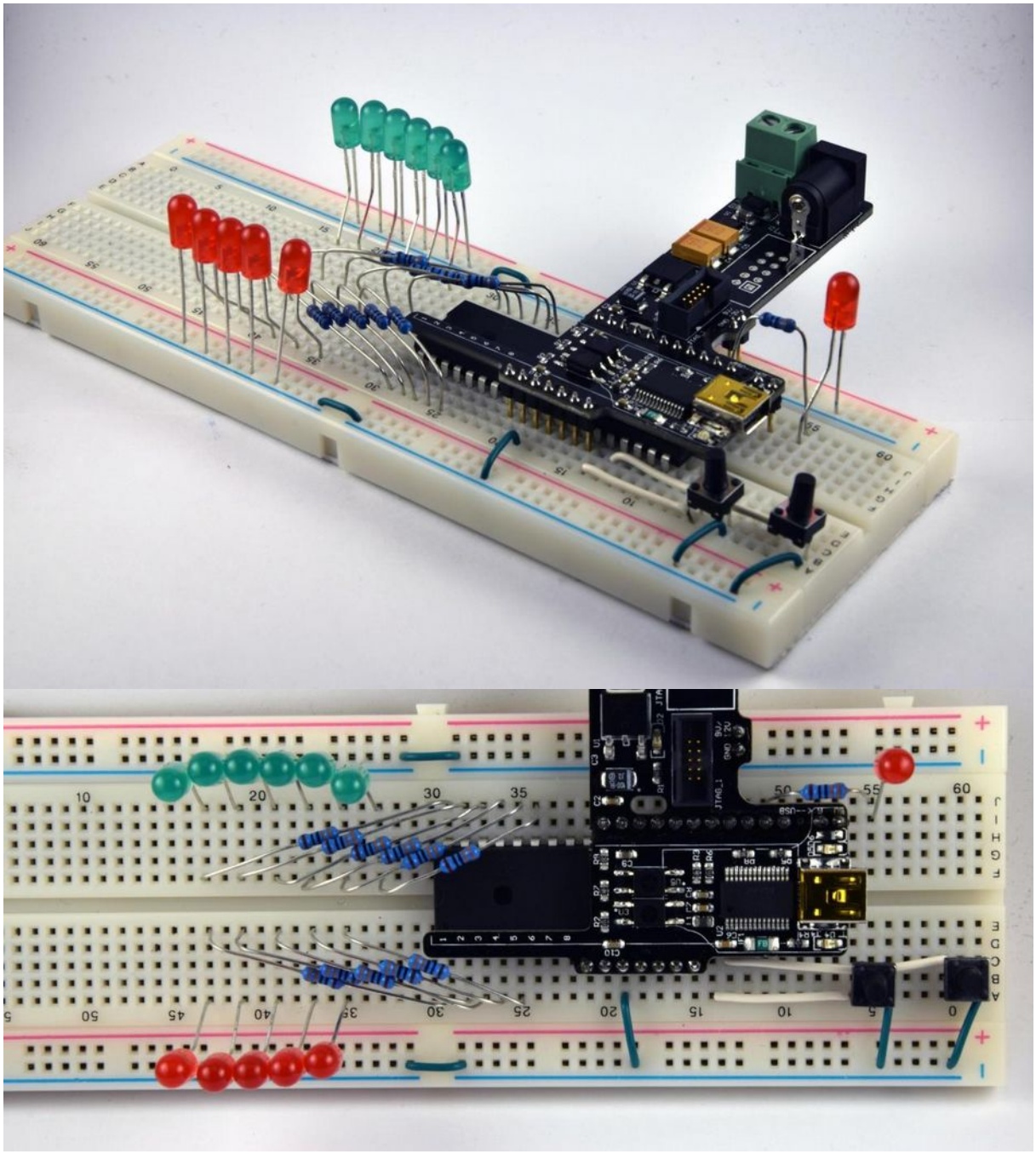


Figure 2 – Schematic and assembly of our clock

You have to decide what kind of LED and resistor to choose. These two questions are related, because increasing the value of the resistance causes the current of the LED to decrease, thus decreasing its brightness. The brightness of an LED increases proportionally with the current flowing through it.

In general, we would like a light source to be as brightly as possible, but the maximal current of both the LED and the microcontroller outputs is limited. The former can be read from the LED datasheet, but most of the LEDs will endure up to 20 mA. The current limit of the outputs is in the microcontroller's datasheet in section 27.2, under the "DC Characteristics" table. The outputs of port A can be loaded with a total of 100 mA, and the outputs of ports B and C *altogether* can be loaded with a total of 100 mA in case of PDIP package.

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).

Port B has 5 LEDs and Port C has 1 LED; if the same current flows through these 6 LEDs, it means a maximum of 16 mA per output. We get the same numbers for port A as well.

We need the current and the voltage in order to determine the value of the resistance. The voltage on it depends on the voltage drop on the LED, which depends on the colour of the LED and the current flowing through it. On the red and green LEDs like the one supplied in the component package, there is a voltage drop of about 2 V at a current of 16 mA, so if our supply voltage is 5 V, then 3 V can be measured on the resistor connected in series. (The exact voltage-current characteristic depends on the type of LED, so a graph in the datasheet of that LED describes the current as a function of voltage.)

The resistance value can finally be calculated using Ohm's Law:

$$R = \frac{U}{I} = \frac{3\ [\mathrm{V}]}{16\ [\mathrm{mA}]} \approx 0.19\ [\mathrm{k\Omega}] = 190\ [\Omega]$$

The *higher* the resistance value, the *less* current flows through the LED, so the value of our resistance should be *at least* this high. (Because of the tolerance of resistors, it is not worth to calculate it with more precision than two decimals.)

WHAT SHOULD OUR PROGRAM LOOK LIKE?

After the initialization at the beginning, we perform three tasks in an infinite loop: first we scan the state of the push buttons, then update the display, and finally wait a second until the next update. Something like this:

```
int main(void)
{
    uint32_t time = 0;
    IoInit();
    while (1)
    {
        time = SetTime(time);
        UpdateDisplay(time);
        WaitSecond();
        time++;
        if (time >= 86400)
        {
            time = 0;
        }
    }
    return 0;
}
```

The `IoInit()` function initializes the I/O ports, which is described in section 12.2.1 of the datasheet. The bits of the DDRx registers describe whether the corresponding pin is used as input or output (where x is the port identifier, i.e. A, B, C or D). For example, if bits 0 and 2 of DDRB are 1 (i.e., the register value is $0000\ 0101_2 = 4 + 1 = 5$), then PB0 and PB2 will be outputs, and the other PBn pins will be inputs. If a pin is used as an input, then the internal pull-up resistor can be switched on by setting the corresponding bit of the PORTx register to 1. Based on these, you can write the `IoInit()` function alone. I trust you will succeed! :)

The `SetTime()` function reads the state of the push buttons, adds 60 or 3600 to the `time` variable depending on which button was pushed, and returns with the new value of `time`. The state of the

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).

PD2 input is contained in bit 2 of the `PIND` register. (Where the numbering of bits, of course, starts from 0, starting with the least significant bit.) Remember that when the push button is pressed, a low value can be read on the input.

With the `UpdateDisplay()` function, we only handle digital outputs using the knowledge we have acquired in the previous chapters of the curriculum. The only challenge is calculating the hours, minutes, and seconds from the `time` variable, which contains the seconds. To do this, it is a good start to try to calculate the digits for each digit of a decimal number. For example, if the input of our function is 527, the following will be true:

$$527 \,\%\, 10 = 7$$

$$527 \,/\, 10 = 52$$

$$52 \,\%\, 10 = 2$$

$$52 \,/\, 10 = 5$$

(The % percent symbol denotes calculating the remainder after integer division.) Based on these, write the `UpdateDisplay()` function. There will be a slight difference between your solution and the above example because while the digits of a decimal number are always less than 10, the seconds and minutes need only be less than 60, and the hours less than 24. Managing the outputs will be very simple if the hours, minutes, and seconds are indicated by consecutive bits of each port. For example, if the `minutes` variable contains the value of minutes, then the

```
PORTA = minutes;
```

statement sets all the minute outputs at the same time.

The purpose of the`WaitSecond()` function is to wait for a second while spending time on counting. Well, not really counting, but increasing the value of a local variable.

```
#define SECOND_COUNTER_MAX 1000000

void WaitSecond(void)
{
    volatile uint32_t i;

    /* This for loop has no effect outside of the WaitSecond() function, and the compiler
also knows this. But the compiler does not know that time is elapsing while this loop
is running, although that's exactly what we want to exploit. If i was not a volatile
variable, then the entire for loop might be deleted by the compiler during optimization.
*/
    for (i = 0; i < SECOND_COUNTER_MAX; i++)
    {
        /* Intentionally empty loop body. The waiting is achieved by increasing the
index variable i one by one. */
    }
}
```

On the website of the curriculum there is a version of the example code that is yet to be completed. Fill in the missing parts marked with “TODO”, upload the program to the microcontroller, fix any possible bugs and admire your cool gadget!

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).

Tracking changes in our software

When we constantly modify the software during development, it may happen that some of the changes don't work, so we would like to return to an earlier version. An obvious solution is to make a copy of the source code before making major changes – but it will soon lead to clutter, which makes it easy to lose track of which version followed which, or what the change was. A version control system stores all the changes together with our comments in a database that can be easily searched later. I used the free Git system and the TortoiseGit program to develop the example code on the website; you can view my edit history with them and keep track of yours.

What Assumptions Were Made During Software Development? What Did We Have to Measure or Experiment With? What Problems Did We Face?

First, we need to know how long it takes to execute a *for* loop. This can be calculated knowing the clock frequency and the type of microcontroller, but that would be difficult: chapter 30 of the microcontroller datasheet describes how many clock cycles each machine instruction takes. It is easier to measure how fast the clock goes with some arbitrary value of the `SECOND_COUNTER_MAX` macro and adjust the counter value accordingly. For example, if you see that your clock is five times faster than it should be, then reduce the value of the macro to one-fifth, and then perform the measurement again to make a finer adjustment. I leave you to do this measurement: in the example program, that macro contains only an approximate value, with which our clock will not tick at the right pace. Perform the `SECOND_COUNTER_MAX` macro calibration as described above. (Note: We could use the `_delay_ms()` macro provided by Atmel in the `WaitSecond()` function, but that wouldn't solve all of our problems.)

Measurement Technology

If we have the same absolute error in two measurements, the result will be more accurate if the measured value is higher.

For example, we want to measure the rate of an unknown clock using an accurate clock. This can be done by measuring with the accurate clock how much time it takes for the clock to be measured to advance by 1 minute. This measurement will have a fixed error, say, 0.2 seconds due to the manual handling of the stopwatch.

If we measure the time it takes to advance by an hour instead of a minute, its error will still be 0.2 seconds. But note that in the latter case the error is the (0.2 / 3600) = 0.0055% of the measured value, while in the former case (0.2 / 60) = 0.33%, which is 24 · 60 · 0.33% = 4.7 minutes of slip every day! That's the difference between a useless clock and a barely useful one.

Second, we assumed that the execution time of our functions is the same in each cycle. This is not true, for example, if there is an if-else branch where the *if* and *else* branches are not the same length. If we used interrupts (which will be discussed in a later chapter of the curriculum), then it would be very difficult to calculate their runtime.

Third, we assumed during the development phase that the execution time of the `while` loop is constant. However, if you change the program or the compiler optimization settings, the `while` loop will no longer take exactly as long, so you need to re-calibrate the `SECOND_COUNTER_MAX` constant.

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).

Finally, while the `WaitSecond()` function is running, the microcontroller cannot execute any other task. (For example, to control the clock buttons or backlight.) Trying to do this would cause our clock to be delayed.

All these problems are solved by using the timer peripheral.

What is the Timer Peripheral Good For?

A microcontroller is much more than a processor: in addition to the operation-executing core (CPU), it includes additional circuits called peripherals to facilitate communication with the environment.

The main function of the timer/counter peripheral is to count the edges of square waves. These are usually the edges of an internal clock signal (which arrive at a fixed rate, so they are a good indication of the elapsed time, hence the term *timer*), but the peripheral can also count the edges of external square waves, for example, the rotation of the axis of a washing machine or an internal combustion engine (which arrive not necessarily at a fixed rate, therefore the name *counter*).

The burden of measuring time is lifted from the `main` function's shoulder by the timer, so the software developer has less work to do, and yet they have a more reliable time source. While the core of the microcontroller deals with the execution of instructions, the timer continuously increases the value of a counter in the background.

With the help of the timer, we can, among other things, simply ask how much time has elapsed since the start of our microcontroller (this is called *uptime*), measure delays of milliseconds or microseconds, or execute certain tasks periodically (for example, to read analog signals a thousand times in a second, or to turn the heater on/off once per hour).

This peripheral is also capable of generating PWM signals, which is discussed in another chapter.

How Does The Timer Work?

There are three timer peripherals in the ATmega16A microcontroller that are named Timer/Counter0, 1, 2. Of these, Timer/Counter0 is the simplest, which has an 8-bit wide counter and is capable of generating a PWM signal. Timer/Counter1 is 16-bit wide and is capable of generating two PWM signals in parallel, while recording the time of an external event. Timer/Counter2 is only 8-bit wide, but it has its own oscillator, so it can count at a frequency independent of the system clock.

In section 14.2 of the datasheet you can see the structure of Timer/Counter0 peripheral, and the following figure shows us the parts that are relevant for us:

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 14-1. 8-bit Timer/Counter Block Diagram

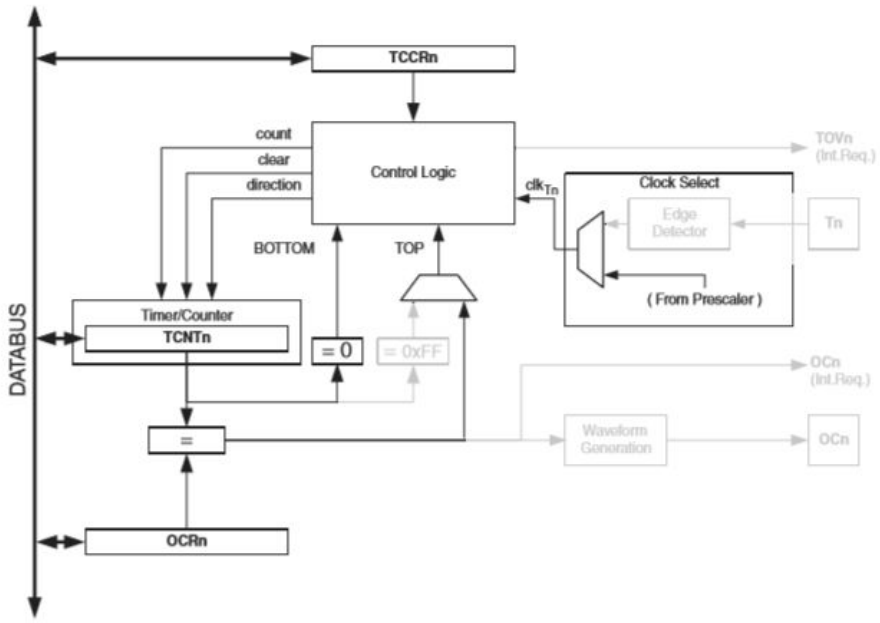


Figure 3 - The internal structure of Timer/Counter0

The TCNTn* register is the counter controlled by the *timer control logic* according to the settings in the TCCRn register. (Here *n* is the number of the counter, i.e., in case of Timer/Counter0, the corresponding registers areTCNT0, TCCR0, etc.) These settings in case of Timer0 affect the following:

- whether the source of the input signal should be an external signal or an internal clock through the so-called *prescaler* (CS0 bits);
- whether the maximum value of the counter is 0xFF or the value of the OCR0 register (WGM0 bits);
- other settings related to PWM signal generation (COM0 bits).

The frequency of counting is determined by the settings of the clock source and the prescaler, which will be described on the following pages.

WHAT IS THE PRESCALER?

If the microcontroller uses a 16 MHz clock for the maximum performance, and if the counter value is increased by one for each rising edge of the clock, then the 8-bit counter reaches its maximum value of 255 in 16 µs. Although it is possible to measure longer times with such a fast counter, a slower counter is often more practical. This is solved by the prescaler, which divides the frequency of the clock source at its input by an integer, for example, dividing the frequency of the 16 MHz clock by eight, so its output will be a 2 MHz clock. This is also required to generate hardware PWM signals of longer periods that would not be possible without a prescaler with an 8-bit counter and a fast clock.

Figure 14-9 of the datasheet shows the timer signals when the prescaler is in use. (In contrast, in figure 14-8, the prescaler is not used.) The prescaler will be used in our program.

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 15-2. Prescaler for Timer/Counter0 and Timer/Counter1(1)

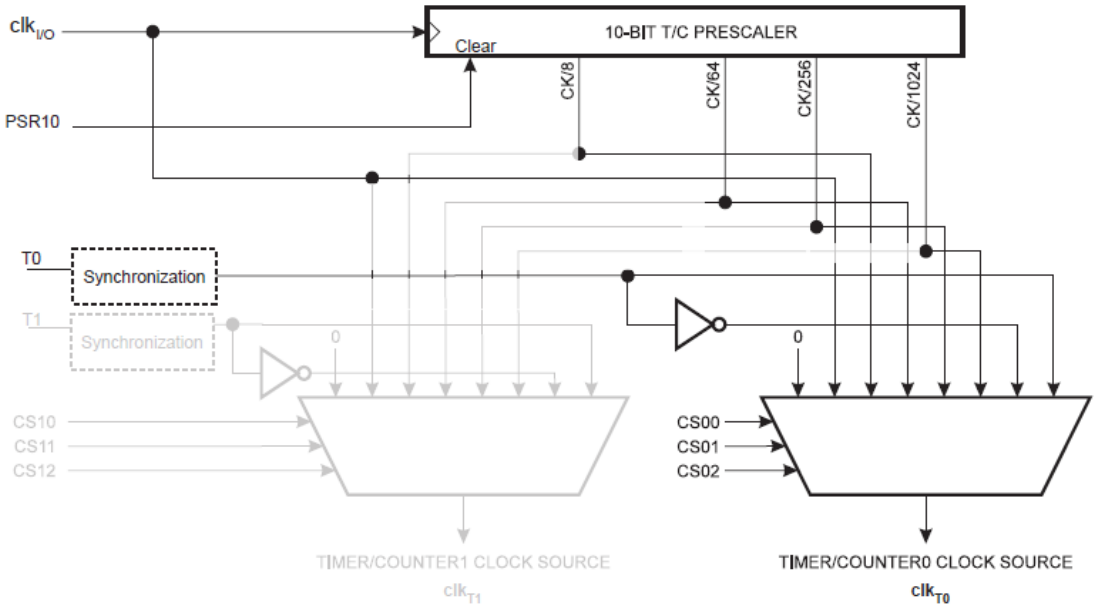


Figure 4 - The internal structure of the prescaler

WHAT KIND OF CONFIGURATION OPTIONS DOES THE TIMER PERIPHERAL HAVE?

One of the two most important configuration options defines the *mode of operation* of the counter.

In *Normal* mode, the counter starts from 0 and the counter value is increased by one until it reaches the maximum value of its data type (i.e. 255 for an 8-bit counter, 65535 for a 16-bit counter). Then the counter overflows and continues counting from 0. This overflow event is indicated by the `TOVn` bit, which is named after the term "timer overflow". This mode can be selected for Timer/Counter0 by setting the `WGM0[1:0]` bits of the `TCCR0` register to 0 (see Table 14-2. of the datasheet). (The bits of a variable are usually denoted by the numbers in square brackets after the name of the variable, where 0 denotes the least significant bit. The expression `TCNT1[15:8]` denotes the bits from 15 to 8 of `TCNT1` variable.)

In *clear timer on compare match (CTC)* mode, the counter starts from 0, and is also increased one by one. As soon as the counter value would be greater than the number set in the `OCRn` (Output Compare Register) register, the counter is reset and resumes counting from 0. At the same time, the Output Compare Flag (`OCFn`) bit is set to 1 until it is reset to zero by our program. (See figure 14-11 in the datasheet.)

Another important setting is the clock source, which can be selected using the `CS0[2:0]` bits of the `TCCR0` register (see table 14-6 in the datasheet). We have the opportunity of stopping the timer, using the main clock of the microcontroller, counting the prescaled version of the main clock, or the signal on the `T0` pin. The external clock signal arriving at the `T0` pin can be used to count impulses from outside the microcontroller: for example, an output signal of a sensor that is on the wheel of a car near to a cogwheel. The output of such a sensor is a digital signal, that is, it can take two values: in its initial state it usually indicates a high value (i.e. a voltage close to the supply voltage), and it indicates

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 low value if it is close to a cog. The microcontroller detects how many edge changes happen in a unit time, and from this the speed of the wheel can be calculated.

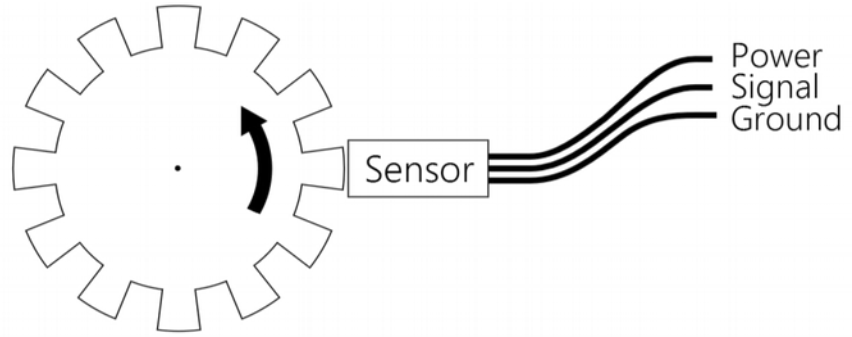


Figure 5 – Angular velocity measurement using a cogwheel sensor

WHAT KIND OF OPPORTUNITIES DO WE HAVE IF WE WANT ACCURATE TIMING?

Chapter 8 of the datasheet describes what kind of clock sources can be used with this microcontroller; these are summarized in the table below.

	Frequency	Price	Accuracy	Start-up time
External high frequency crystal	max. 16 MHz	$0.10…$1.50	10…100 ppm	16k clock periods
External resonator	max. 8 MHz	$0.50	0,1…0,5%	1k clock periods
External low frequency crystal	32 kHz	$0.10…$0.50	10…100 ppm	1k / 32k clock periods
External RC-oscillator	max. 12 MHz	$0.02…$0.10	5…10%	6-18 clock periods
Internal RC-oscillator	1 / 2 / 4 / 8 MHz	$0.00	1…3%	6 clock periods
External clock source	max. 16 MHz	?	?	6 clock periods

Cost is an important design aspect, which means not only the price of the components, but also the design engineer's time. In terms of both price and simplicity, the microcontroller's internal RC oscillator wins, but it is orders of magnitude less accurate than an external crystal.

If we need accurate and fast timing, an external crystal is usually the best solution.

An external clock setting is required when there are multiple microcontrollers in a circuit, and they need to operate at exactly the same frequency.

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).

What Are the Costs?

It is obvious why the cost of the components is important: if you need to make 1000 pieces of a circuit you save $200.00 with a component which is cheaper by $0.20 each. However, our invested time should never be disregarded!

Life is very short, and on whatever we spend time on, the less we have left for other, perhaps more important or more enjoyable things.

Engineering salaries are high, so it is in their employer's interest to make them work as quickly and efficiently as possible.

It is often necessary to work on tight deadlines to reach a fixed date (for example, the Christmas gift should be wrapped up by 24 December) or to be faster than the competition (for example, our next-generation mobile phone should come out sooner than theirs).

The sooner we complete a project, the sooner we can enjoy its benefits (for example, the joy of Christmas lights or the proceeds from a sale).

If we add up all of these, saving $200.00 is not always worth it.

How to Use The Timer For Our Digital Clock?

How accurate should our clock be?

Clock precision is measured by much the clock is allowed to deviate in a given timeframe, e.g. in 100 seconds. For example, a 10 MHz oscillator with a frequency tolerance of ±0.5% delivers between 9,950,000 and 10,050,000 pulses in 1 second. For more accurate devices, tolerance is given in ppm (*part per million*), equivalent to 0.0001% (one millionth of the nominal frequency).

I expect my watch not to slip more than half a minute in a month: this means one second a day. This is rounded down to $1/(24 \cdot 60 \cdot 60) = 11$ ppm.

It Matters Whether It's a Little Bit Too Accurate or a Little Bit Too Inaccurate

Why did I round down when the result was 11.57 ppm?

Because my requirement is that it should be at least this precise: if my watch was delayed by 12 ppm, it would no longer fulfil my requirements. You should always pay attention to the sign of the tolerance the teacher does not care if you are early in the class, but they care much more if you are late! If the car's braking distance is half a meter shorter than necessary, we are happy, but if it's half a meter longer…

It is impossible to make an RC-oscillator with such precision, and it would be difficult to find such a ceramic oscillator, if there is even any. In fact, with this 11 ppm we have set the bar high enough: when writing this chapter of the curriculum, the most accurate quartz that's easily available offers 10 ppm. Only 1% accuracy is guaranteed for the calibrated internal oscillator, which could cause a delay of quarter of an hour per day. That wouldn't be very practical.

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).

Calibration of Prototypes

Actually, only the circuits in mass production require the clock signal to be *accurate*; if only few pieces are produced, it is enough if the clock signal is *stable*. If only one piece is produced, we should only take into account how much the oscillator frequency varies over time and at different temperatures. The initial error can be corrected in the software, which is called *calibration*.

So, we need to have a quartz in the clock to be sufficiently accurate. (It is no coincidence that the term "quartz watch" is used.) Even so, we have two options: the quartz can be used either as the main clock of the microcontroller, or only for the input of the Timer/Counter2 external clock (then the main clock could be the internal RC oscillator). The advantage of the former solution is that the power consumption is lower if a slow crystal is used as the main clock instead of the minimum 1 MHz RC oscillator. The advantage of the second solution is that the higher operating frequency allows more operations per unit time (for example, more complex display management), and we don't need to worry about fuse settings. I chose the second solution for the example program, so we will use the Timer/Counter2 peripheral.

The properly modified circuit diagram is shown in Figure 6. Although the use of an external crystal usually requires the use of two capacitors between the two legs of the crystal and the ground, according to section 8.9 of the datasheet, it is not required in this case.

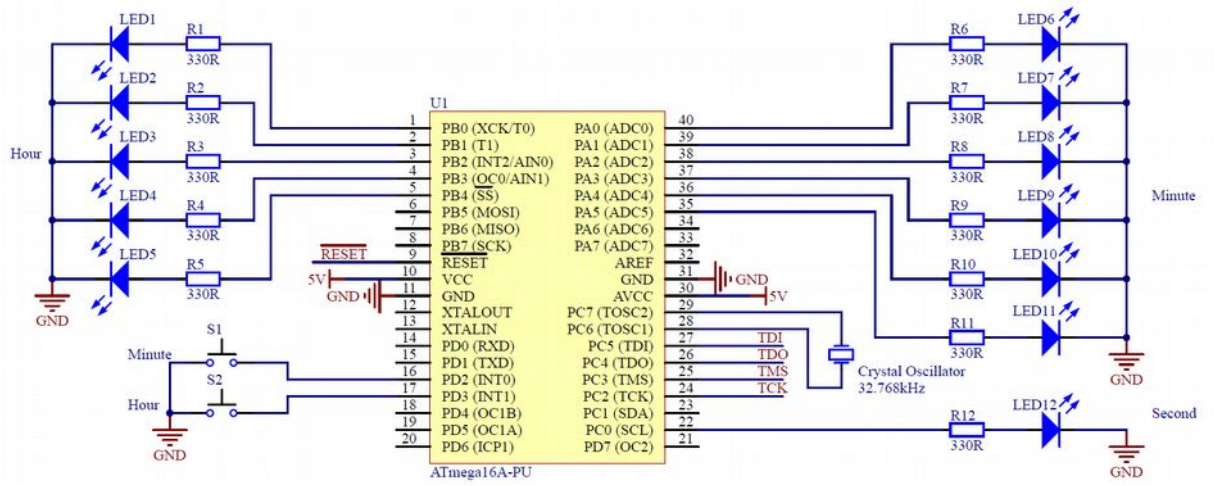
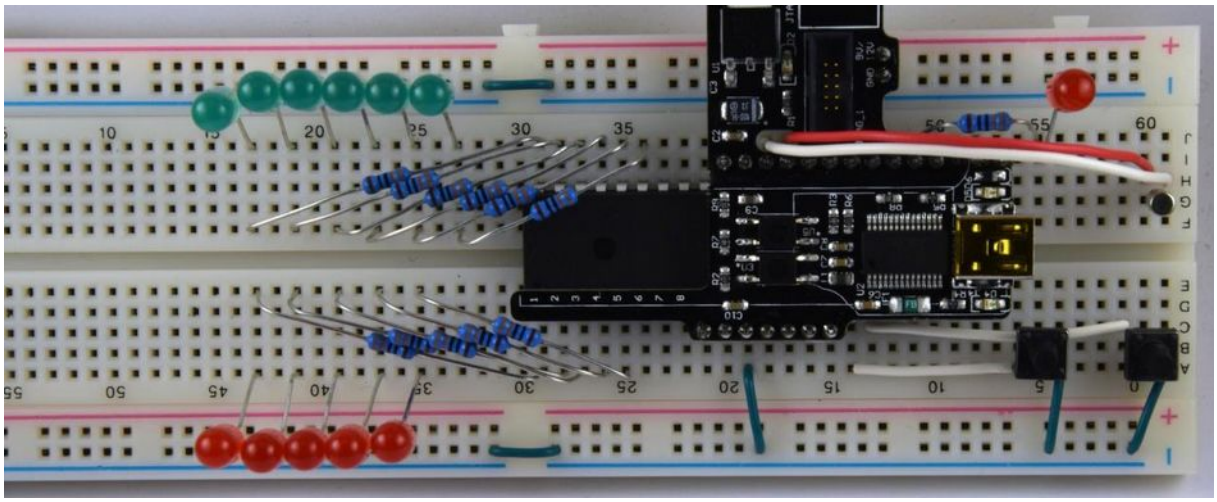


Figure 6 – Schematic of the clock using an external crystal



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).

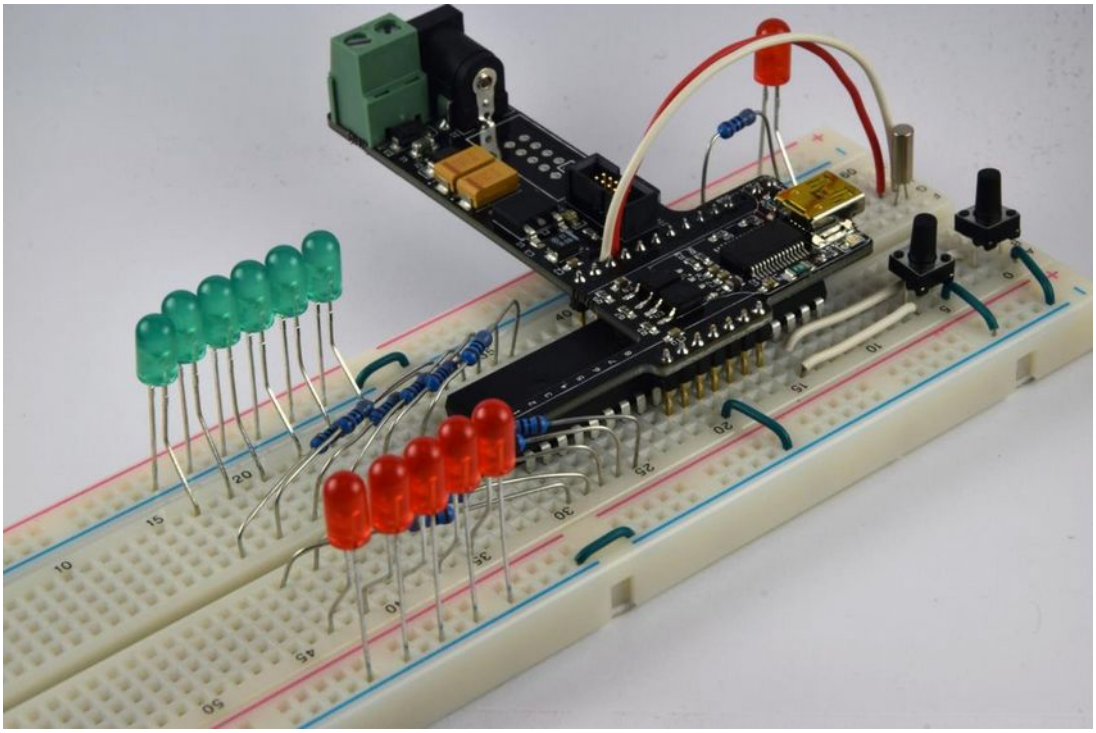


Figure 7 – The assembled circuit

WHAT SHOULD THE NEW SOFTWARE LOOK LIKE?

The concept of the new software is the following: at start-up, we set up and start the timer peripheral, then we execute three tasks in an infinite loop: scan the state of the buttons, update the display, and wait until one second has elapsed. Like this:

```
int main(void)
{
    uint32_t time_seconds = 0;

    IoInit();
    SetupTimer();

    while (1)
    {
        time_seconds = SetTime(time_seconds);

        UpdateDisplay(time_seconds);

        while (CheckIfSecondElapsed() == false)
        {
            // intentionally empty loop core
        }

        time_seconds++;
        if (time_seconds >= 86400)
        {
            time_seconds = 0;
        }
    }

    return 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).

WHAT IS HAPPENING HERE?

Let's start with the `SetupTimer()` function. This function configures the Timer/Counter2 peripheral by following these steps:

1) It configures the timer to use the 32 768 Hz oscillator between the TOSC[1:2] pins as input, by setting the AS2 bit in the ASSR register.

2) It configures the prescaler to divide the above 32 768 Hz clock signal by 1024. Thus, the control logic of the counter will have a 32 Hz clock.

3) Set the timer to Clear Timer on Compare Match (CTC) mode. In this mode, the counter counts between 0 and OCR2 and at every overflow the OCF2 bit is set to 1 (see figure 17-11 of the datasheet).

4) Sets OCR2, i.e. the upper limit of counting to 31, so the counter will count between 0 and 31, which means that an overflow will be generated at every 32 nd beat. The 32 Hz internal clock signal causes the timer peripheral to set the OCF2 bit to 1 once per second. This flag will then be periodically restored to 0 by our program.

5) It starts the timer.

The prescaler setting in the ATmega16A microcontroller also starts the timer (see table 17-6 in the datasheet). Accordingly, combining points 2 and 5, the implementation of the function may look like this:

```
void SetupTimer( void )
{
    // TODO: 1. Setting the timer input to TOSC[1:2] pins
    // TODO: 2. Setting the Timer Mode to CTC mode
    // TODO: 3. Setting OCR2
    // TODO: 4. Setting the prescaler, which will divide 1024
}
```

It is worth to mention that I write the program in the same order as you read it in this section: first, I design the high level concept in my head and try to write it (currently only the `main` function), then the functions (`SetupTimer`, `CheckIfSecondElapsed`) and data structures (if there is one in the program), and finally I look through the datasheet, searching for details such as which register configures the prescaler. If the program is more complicated, or if I get stuck in a thought, then I'm not ashamed to pick up paper and pencil to clarify my thoughts on drawings or diagrams.

It also helps a lot if you explain how the program works to a competent friend or colleague: we often notice during the explanation that our program has errors or is unnecessarily complicated. The best programmers I know regularly ask for the opinion of their colleagues, and not only when their software doesn't work. (This activity is known as "code review".) The best time to review our code is right after the compilation of our program: the compiler has already found the most trivial mistakes, but we have not yet spent time on uploading and debugging faulty code.

How Can Our Dog Help in Software Development?

It is interesting that the recipient of our explanations does not necessarily need to be a competent expert, or even a living person. The essence of "rubber duck debugging" is explaining our program line by line to a rubber duck. By putting it into words what our program does and what we want it to do,

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).

any difference will be much more obvious. The best programmers are considered as mad geniuses regardless, so a conversation with a rubber toy will not change this opinion.

Well, let's get back to the program! The `CheckIfSecondElapsed()` function also deals with the timer: this function must return `true` if the timer has reached the set maximum value (i.e. if another second has elapsed) or `false` if it has not.

The advantage of using a timer peripheral is that this function is very simple: the peripheral works in the background regardless of what operation the microcontroller is executing. The moment a full second elapses, the hardware will set the`OCF2` flag in the `TIFR` register (Timer Interrupt Flag Register), and this flag will remain in this state until the software sets it back to 0. This is shown in the figure 17-11 of the datasheet and the `OCF2` bit description in section 17.11.6. The value of the `TIFR` register can be changed in a strange way: the value of a bit can be reset by writing 1 into it; if you write 0, it does not change the bit value.

```
bool CheckIfSecondElapsed(void)
{
    bool second_elapsed_b;

    if (0 != (TIFR & (1 << OCF2)))
    {
        // Resetting the OCF2 bit
        TIFR = (1 << OCF2);

        second_elapsed_b = true;
    }
    else
    {
        second_elapsed_b = false;
    }

    return second_elapsed_b;
}
```

Why Is It Better That I Used the second_elapsed_b Variable, Instead of Using 2 Return Statements in the Function?

Although the above simple function would have been understandable anyway, in general it is easier to understand, debug and maintain a function that has only one exit point. When you have completed your own projects, notice that there is a lot of time in troubleshooting and fixing errors during software development.

What Does the "_b" Mean at the End of the second_elapsed_b Variable?

We save ourselves (and those who later want to understand our program) from headaches beyond measure if we write the unit of measurement in the variables' names. Boolean variables are often denoted by the "_b" suffix: from this it is visible at a glance that the `second_elapsed_b` variable has values `true` or `false`, depending on whether a second has elapsed or not. It is interesting, that NASA's space probe, the Mars Climate Orbiter, was lost due to a preventable error. There, one software module returned the engine-generated momentum in imperial units, as opposed to the required metric units. That's how the result of a $193 million development burned up in the Mars atmosphere.

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 `UpdateDisplay()` function is the same as it was in the previous program, because the function was written to be independent of the timing. On the other hand, if you later want to use a 7-segment display instead of the LEDs, or write the time to a serial port in text format, you will only need to make few changes in the software. If you want to use the same function in another project, it will be easy to move large sections of the program. (Provided that they are well documented: we humans tend to overestimate our memory and believe that we will remember our design decisions correctly after many years have passed. This is often not the case, so we make things easier for our future self or for our colleagues by describing our tricky decisions and explaining what we did and why.)

The curriculum website has a version of the example code that is yet to be completed. Fill in the missing parts marked with “TODO”, upload the program to the microcontroller, fix any bugs and admire your much cooler gadget!

SUMMARY

In this chapter of the curriculum you have learned about the following:

- what difficulties we face with timing without using the timer peripheral,
- how to use the timer peripheral,
- how to make software development easier,
- what are the qualities of a good engineer: able to formulate problems and solve them, and able to admit their own mistakes.

The timer peripheral was used in this section in timer mode. In another chapter it is discussed how you can apply them to generate PWM signals.

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).