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

17 - PWM with Microcontroller

Written by Endre Szesztay

English translation by Xtalin Engineering Ltd.

Revised by Szabolcs Veréb

INTRODUCTION

In previous chapters, we have learned about the essence of PWM. We have seen, that PWM can be used to implement analogue (continuous) power or voltage control without analogue components by switching a voltage (or current) rapidly. This feature is important in two cases:

- If we would like to create an analogue signal in a purely digital system (by purely digital system we mean that we have logic circuits and signals only, which can only be switched on or switched off, for example 5 V and 0 V)
- If we would like to control high power signals, such as in motor control.

In the light of above it isn't surprising that most microcontrollers contain a PWM control unit. This is a counter/timer peripheral (hardware unit) with PWM functions or a separate peripheral designed specifically to create PWM signals. To make it simple: we set the frequency and the duty cycle, and the PWM signal magically appears on one pin of the microcontroller. This pin can even drive an LED or a switching element (FET) directly.

It's important to note that after these values are set, the generation of the PWM signal happens independently from the CPU, so it doesn't take valuable processor time away from other tasks. We will talk about this later.

HOW DOES A PWM PERIPHERAL WORK?

In order to use the PWM controller let's have a look at its operation in detail. A PWM controller is usually responsible for the following functions, although these might vary in different microcontrollers:

- Setting the PWM frequency (period of time) and duty cycle - these are mandatory
- Generating an interrupt at the end of each cycle – usually they also provide this feature (We will talk about interrupts in detail later. Basically, the processor is able to stop the currently running program based on certain external signals and start to run a specified program section. After that, it returns to the original program section).
- Generating further interrupts at the peripheral's different events, for example when the output PWM signal changes (0->1, or 1->0 transition)
- Managing several PWM outputs which have different duty cycles

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).

- Setting output (pins): we can choose at which pin of the microcontroller the created signal would be accessible; or when the internal counter reaches the threshold, what transition should happen on the output: 0->1 or 1->0 (polarity).
- If there are more outputs, setting the timings between each other (turn on and off at the same time or work alternating)
- Synchronizing several completely independent PWM controllers
- Controlling DMA – this is a hardware element which makes it possible to move data between the memory and the peripheral without using the CPU.
- Managing other input signals, as an example: an external error signal can block the operation of the PWM.

Let's have a look at some examples, what kind of tasks we can implement with a microcontroller PWM unit:

- Producing a square wave signal with set frequency. Here we don't change the duty cycle of the signal (50% for a square wave), but we alter the frequency. This can be useful if we would like to produce a sound with varying tone.
- Generating a fixed-frequency PWM signal with the duty cycle set in the program. We can use this to continuously regulate motors, or the brightness of an LED, and after filtering we can create an analogue voltage too.
- Generating a varying PWM signal from cycle to cycle. This is more complicated but there is an opportunity for the duty cycle to be different in each PWM cycle. When using this the program of the CPU has to be synchronized with a signal generated by the PWM unit. Quick controls can be implemented like this which are commonly used to control motors.

Of course, the PWM unit can be used in several different ways as well, and this depends only on the imagination of the programmer.

Regulation vs Control

The upper last two sections apparently say the same thing but in one of them I use the word control while in the other I use the word regulation. The difference is that the word regulation is used for that case when we change the speed of the motor by setting voltage on the motor to a previously calculated value. Like this the speed won't be accurate, only approximate, but the motor will do what we want: spin slower or faster.

In comparison control is a more complex process. Using the example above we continuously measure the speed of the motor and based on the measurements we modify the voltage on the motor. (If the motor spins too fast then we slowly decrease the voltage and vica versa.) We call this repeated measurement-calculation-intervention process control. In specialized literature you can meet the „feedback" expression too, which basically means the same thing but here the sensing and measuring is emphasised. The result of the measurement is the feedback signal, which gets "fed back" to the controller to achieve control.

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).

Implementing the control, we have mentioned above is way more complex than it look at first glance. The reason for this is that controllers can easily get into a positive-feedback loop, where the result is not a steady sate, but a continuously oscillating output instead. Those who are interested can learn more about control by searching for “PID”, or “PI controller” on the internet.

We talked about that the PWM controller is a peripheral of microcontrollers. But what exactly is a peripheral?

For example, generation of a PWM signal can be implemented in software too. A counter loop has to be created and the state of an output pin has to be changed at the right times. This is a very simple task; however, it would consume a lot of CPU time. Because of that, specialized hardware elements are built inside the microcontroller for tasks like this, and these are called peripherals.

The peripherals work on their own and they don’t consume CPU resources. They have their own internal logic; they get the clock signal; they have a direct connection to the pins of the microcontroller and the program needs set them up and read and write the required data.

Here are some of the most common type of peripherals:

- ports: these peripherals make the pins of the microcontroller available for the program; we can read the state of the inputs and we can set the state of the outputs with the help of simple registers.
- counters – timers: through registers, we can set the speed of the counter, and we can read the current value anytime in software. This is perfect for time measurement.
- PWM: we will talk about this in detail in this chapter
- communication peripherals, for example: RS232, USB
- analogue – digital (ADC) and digital – analogue (DAC) converters. With these, the pins of the microcontroller can be used to measure or generate not only digital signals such as 0 V or 5 V but any voltage within the entire supply voltage range.
- interrupt controller. In this curriculum, you will learn more about this later

The next important question is how the program running on the CPU can communicate with the peripherals, in our case the PWM controller. Peripherals have registers for this purpose and we can control our peripherals by writing proper codes (numbers) into these registers. Registers nothing but special section of memory from the CPU’s perspective so it is usually able to read and write them. However, they do much more than simple memory does because the data (value of each bit) inside them directly affects the operation of the hardware (the logic circuit in peripherals).

Don’t confuse peripheral registers with the CPU’s own registers; the latter is used in the internal logic of the CPU and only machine code can directly refer to them and we're not dealing with that right now.

The peripheral registers can be accessed by simple memory read and write actions, similarly as the any other memory but on a special address. The datasheet of the microcontroller describes the operation of the peripherals and the addresses of the registers where they can be accessed and the meaning of each bits (effect) inside the register.

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).

Our Atmel processor has an 8-bit architecture, which means that the registers are also 8-bit wide. There are registers where each bit means something different, and there are registers where whole register is an 8-bit number.

There are some peripherals where a 16-bit number is used, with two registers, one acting as the lower byte (Lo) and one acting as a higher byte (Hi), and the registers have to be written separately. Handling of registers and PWM peripheral will be discussed in detail.

Interrupts

I mentioned while talking about the features of PWM peripherals, that they can generate an interrupt. We will talk about this in more detail in a later chapter, but I just explain the essence of it in a few lines. The interrupt mechanism generally means that the CPU stops execution of our code in response to an external signal and it starts to run another part of the program (this is called the interrupt routine). Interrupts are used to notify the CPU immediately, so the program is able to respond to an external event. It's important that after the interrupt has been served (the interrupt routine finished) the CPU should be able to get back to run the previous part of code that was interrupted in a way the internal state of the CPU is the same as it was before (CPU registers are unvaried) so the interrupted program doesn't recognize anything from serving the interrupt. There are special commands and mechanisms in the CPU and the microcontroller to handle the interrupts; on the one hand during an interrupt the CPU has to save the internal registers' state, and on the other hand the code which serves the interrupt should be able to identify the origin of the interrupt.

Almost all peripherals are able to generate some kind of interrupt through which the CPU can quickly respond to the events of the outside world; this can be an external signal on a pin, receiving data through some kind of communication channel, or the end of a timer. The interrupt controller is also a separate peripheral which allows you to set which units can request an interrupt from the CPU and in what order of priority.

Operation of the PWM Peripheral

In the Atmega16A microcontroller we use, the timer peripherals can generate a PWM signal too.

Counters (revision)

The essence of the counter peripheral is the counter register. This is an 8- or 16-bit register which is able to increase or decrease its value by one synchronized with a periodic signal. For example, if the value of the counter (register) is 12 then it will have the following values 13, 14…etc. after each period of the clock signal.

If the counter has 8 bits, then usually after 255 its value changes to 0 and the counting continues from here. The value of the counter register can be read or overwritten by the CPU at any time, but you don't need to do this during normal PWM production.

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 a timer you can implement timing, measure time, and counting tasks too. Timing or time measuring can be implemented if the counter is driven from a constant clock source, counting can be implemented if we connect an external impulse in place of the clock signal. TCNTn is short for “timer-counter”, which is the register name of the peripheral in Atmega16A. After the register a number signifies which timer peripheral we are talking about, as there are multiple inside the controller. For example: TCNT2 means the TCNT register of the second timer.

Operation of the PWM Peripheral

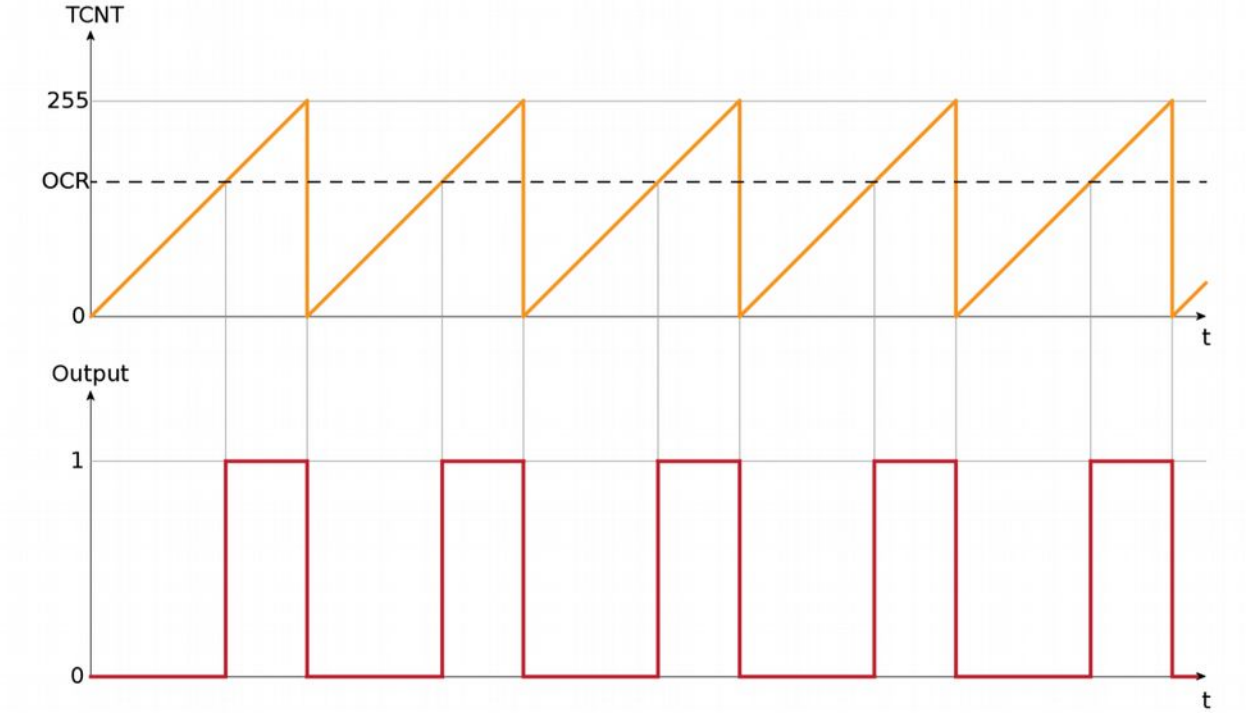

Figure 1 – Operation of PWM peripheral

The figure above shows the typical operation of the PWM peripheral. The datasheet of the microcontroller calls this “Fast PWM Mode”, where the counter always counts up. In the figure above the top waveform represents the value of the counter register TCNTn, and as you can see it is counting up to 255, and then starting from 0 again, creating this sawtooth wave.

The other operation mode supported by the microcontroller is called “Phase Correct PWM Mode”, where the counter does not go back to 0 after reaching 255, but starts to count down instead, creating a triangle wave. This is important, if you need to synchronize multiple PWM peripherals together, but we won’t be using this mode in the curriculum.

To generate a PWM signal we need another value that does not count, to which we can compare our counter after every count. This register is called the OCRn (Output Compare Register), and you can see its value in the figure above represented by a dashed line. If the counter value is smaller than the OCR register’s value, the PWM output is “0”, when it is bigger, the output is “1” (this can be reversed by a setting). This way, modifying the OCR value from 0 to 255, sets the duty cycle of the PWM signal from 0% to 100%.

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).

USING THE PWM PERIPHERAL

ADJUSTING LED BRIGHTNESS

After learning about the inner operation of the PWM peripheral, let's see which registers we need to set in the Atmega16A to use it.

The simple program below changes the brightness of an LED continuously from minimum brightness to maximum brightness and then back. To achieve this, it produces an approximately 31 kHz PWM signal, and the duty cycle will determine the brightness of the LED.

A full example can be found on the website of the curriculum.

The hardware, that is required to operate the program, is very simple to build. You only need to connect an LED and a 330 Ω resistor between the ATMega16A's appropriate pin and the GND. This pin is the chip's 21st pin, its name is „PD7 (OC2)" and a detailed description can be found in the datasheet of the microcontroller. Its name refers to the fact that this pin has two functions: basically, it is the D port's 7th pin, but the OC signal of the 2nd timer can also be accessed here. The OC is the abbreviation of „Output Compare" i.e. this is where the result of the comparison between the counter (`TCNT2`) and the `OCR2` register comes out.

To make our program work correctly, we have to do the following main steps:

- Set the IO ports properly so that the produced PWM signal can appear on the microcontroller's pin. This is done by the `IOInit()` function.
- The registers of the timer-counter peripheral have to be set correctly so that the PWM can works in the right mode. This is done by the `PWMInit()` function.
- Finally, the compare value has to be set i.e. the duty cycle of the PWM. This is done by the infinite cycle which is in the `main()` function; it continuously and slowly changes the comparing level and because of that the brightness of the LED also changes continuously.

The comments in the source code explain the tasks of the certain parts but we will see it in detail. If we would like to write our program, then it is required to look at the Timer/Counter units in the microcontroller's datasheet.

The program consists of a single `main.c` file, which contains all the required code. The program starts to run in the `main()` function and this calls the `IOInit()` then the `PWMInit()` functions. The `IOInit()` does simple register writes and the comments show the meanings of the registers. The `PWMInit()` function sets the operation of the timer (PWM). The comments embedded into the code show what they refer to, in order to understand it more deeply it is recommended to review the datasheet.

Setting multiple bits

Instead of the `sbi()` (set bit, set bit to 1) and `cbi()` (clear bit, set bit to zero) commands defined in the compat/deprecated.h file there is a different way to set and clear bits of a register, using bitwise "or" and

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).

"and" operations. The following operations sets the three counter select bits (CS22, CS21, and CS20) of the timer control register of timer 2 to "001".

```
TCCR2 = TCCR2 | (0<<CS22) | (0<<CS21) | (1<<CS20);
```

This is a bit more difficult to understand than the `sbi` and `cbi` commands, that's why we are using them rather than these bitwise operations. However, it is worth to look at how the `sbi` and `cbi` macros are defined. Macros are `#define` expressions, that the pre-processor processes before compilation, they can take parameters, similar to functions. If we look at the expression `sbi(TCCR2, WGM20);`, every part is a macro definition. The macro `sbi` is defined as the following:

```
#define sbi(port, bit) (port) |= (1 << (bit))
```

As you can see it is basically a short notation for the same bitwise operations we have shown above. Since macros are processed before compilation, using the `sbi` macro is equivalent to writing the bitwise operations ourselves, but it makes our code more readable. The `TCCR2` is also a macro:

```
#define TCCR2 _SFR_IO8(0x25)
```

As you can see it uses another macro, `_SFR_IO8()`, which creates a memory address out of an I/O address. This is important so that we can assign a value to the register, so for example the `TCCR2 = 0;` becomes a valid command.

Of course, the `WGM20` is a macro too which simply defines `WGM20` as the constant 6:

```
#define WGM20 6
```

This expresses what the datasheet shows on page 125, that the WGM20 bit is the 6th bit in the byte which contains it.

The final parts of the `main()` function have only one thing to do, change the value of the `OCR2` register continuously, and reverse the counting direction when it reaches one of the boundaries. This is the register (Output Compare Register) of the contains the value, which is compared with the counter after each step, determining the duty cycle of the PWM signal.

```
/*
 * Chapter 17
 *
 * Adjusting the brightness of the LED on the PD7 pin of the microcontroller,
 * by linearly changing the duty ratio of the PWM
 */

#include "../Headers/main.h"

int main(void)
{
	//Initialization of outputs and inputs
	IOInit();

	//Initialization of PWM
	PWMInit();
```

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).

```
	//Start counting upwards
	bool cntr_up = true;

	//Infinite loop
	while (1)
	{
		//If PWM duty ratio increases
		if (cntr_up)
		{
			//If it is lower than MAX
			if (OCR2 < PWM_DUTY_MAX)
			{
				//Increase
				OCR2++;
			}

			//If it has reached the maximum
			else
			{
				//Changing direction
				cntr_up = false;
			}
		}

		//If PWM duty ratio decreases
		else
		{
			//If it is larger than MIN
			if (OCR2 > PWM_DUTY_MIN)
			{
				//Decrease
				OCR2--;
			}

			//If it reached the minimum
			else
			{
				//Changing direction
				cntr_up = true;
			}
		}

	//Delay in order to make the changes visible
	_delay_ms(PWM_DELAY_MS);
	}

	return 0;
}

void PWMInit()
```

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 PWM frequency (page 120 of the datasheet) fpwm=fclkIO/(N*510),
	 * where N is the division of the clock signal.
	 * From this N=fclkIO/fpwm*1/510, namely for a frequency around 10 kHz
	 * N=8e6/10e3*1/510=1.569 is required..
	 * There is only 1,8,32,64,128,256,1024 divisions in the microcontroller,
	 * by choosing N=1
	 * fpwm=15.686kHz
	 *
	 */

	//Simple clock signal division for the timer2
	cbi(TCCR2, CS22);
	cbi(TCCR2, CS21);
	sbi(TCCR2, CS20);
	//Non-negated PWM operation, the OC2 output is 0 if the counter is bigger
	//than the threshold
	sbi(TCCR2, COM21);
	cbi(TCCR2, COM20);
	//Phase correct PWM mode
	cbi(TCCR2, WGM21);
	sbi(TCCR2, WGM20);

	/* Instead of the sbi() and cbi() functions,
	 * which are definied in the compat/deprecated.h file,
	 * we can set each bit with the following solution which is based
	 * on the following "or" connections:
	 * TCCR2 = TCCR2 | (0<<CS22) | (0<<CS21) | (1<<CS20);
	 * This is kind of messy, so it's recommended to use the functions above.
	 */
}
```

Further Possibilities

In the code above, the brightness is changed by changing the value in the OCR2 (8-bit long) register. If you'd like to play with the code or try your own program, then you only need to write a number between 0 and 255 into this register. In the case of 0 the duty cycle and so the brightness of the LED will be zero while in case of 255 you'll get full brightness.

Out of bounds values

If you write a value which is out of the range of 0..255, in principle that is an illegal command (according to the rules of the C language) however the code can be built and it will run. The value of the register will be the lower 8 bits of the value we have written. So, for example writing 256 would mean that you wrote 0 and, writing 257 means 1, and so on.

Erasmus+

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

BUZZER - PLAYING A MELODY

Another example project named *CE17_2_PWM_zummer* is also available on the website of the curriculum, which can play melodies.

Building the hardware isn't complicated here either: a speaker has to be connected to the microcontroller through a resistor but now we must use pin 19th the chip, which is PD5 (OC1A). The reason for this is that we are using the first timer peripheral for this task (16-bit Timer/Counter1). This is a 16-bit timer (as the name suggests) instead of an 8-bit one, so we can adjust the output timing in much smaller steps.

The structure of this program is a little bit more complicated. This is partly because its operation is more complicated, and partly because the program is separated to multiple source files for easier maintainability. Storing program functions that belong together in separate files is a general practice in software development. This keeps files short and easy to understand. The code is contained by the files `main.c`, `io.c`, `pwm.c`and `music.c`, and their associated `main.h`, `io.h`, `pwm.h`and `music.h` headers.

In this example the timing peripheral works in *"Clear Timer on Compare Match"* (CTC) mode. This means that when the values of the counter and the comparator (OCR) register are the same then not only the output changes state but also the counter is reset, and the counting starts over. This operation is explained on the datasheet's 74th page.

Figure 14-5. CTC Mode, Timing Diagram

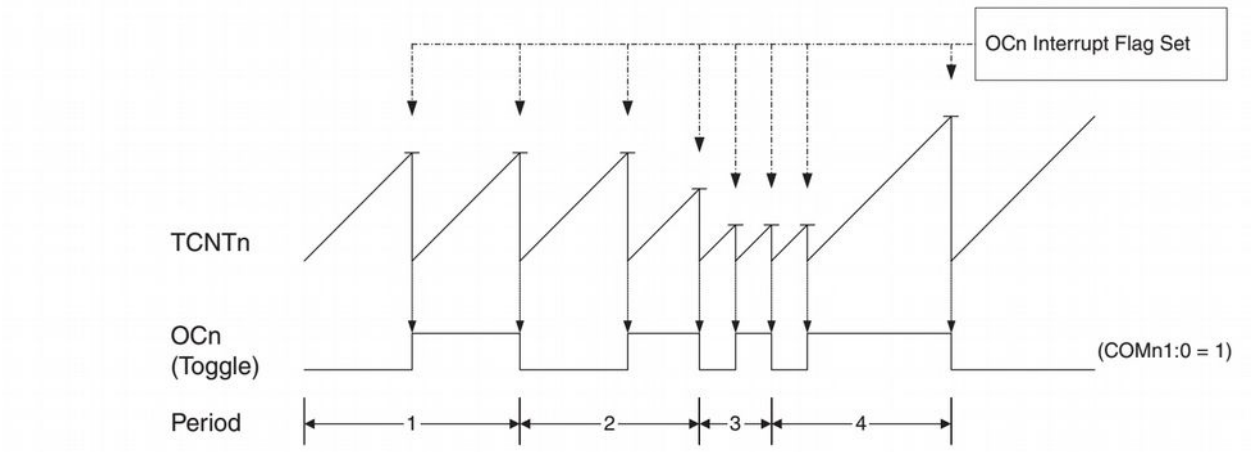

Figure 2 – Operation of timer peripheral (ATMega16A datasheet)

The signal produced during CTC operation mode always has a duty cycle of 50%, although its frequency changes according to the value of the OCR register. This makes this mode suitable for producing a variable frequency signal.

The buzzer program plays two simple melodies through the speaker. To play a melody you only have to change the frequency of the signal, the duty cycle can stay constant. The program works very simply, the OCR values representing sounds are taken from a table in memory, they are pre-programmed.

We are using a 16-bit timer this time, where both the counter (TCR1), and the output compare (OCR) registers are 16-bit wide, meaning their value is not restricted to be in the range of 0.255, but it can go from 0 to 65535. We can create a higher resolution PWM signal this way, which is needed to play tunes.

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).

Managing 16-Bit Registers

It's important to know how to read and write 16-bit registers with an 8-bit processor. An 8-bit processor can only move 8 bits of data in one instruction, and generally has 8-bit registers. The internal representation and operations on 16-bit numbers are done by the compiler, but the registers of the processor when we store or read the registers, we cannot simply assign a 16-bit value to an 8-bit register. The solution is that the processor has two 8-bit registers representing the 16-bit OCR1A register, called OCR1AL and OCR1AH, and we have to assign to them separately. The OCR1AL contains the lower 8 bits while the OCR1AH contains the upper/higher 8 bits. An assignment can be done like this:

```
//Upper 8 bit
OCR1AH = (note >> 8);
//Lower 8 bit
OCR1AL = note & 0xFF;
```

There are some registers where it is necessary to do a read or write operation to both parts at the same time, such as the timer counter, or output compare registers. The timer peripheral increments or decrements the counter independently from the CPU, so if we read the first half of the current value from one register, by the time the CPU gets around to read the second half, the value might have already changed, resulting in a false value in the CPU. Similarly, when writing to the output compare register, having one half of the number being a new value, and the other half an old value can mess up the PWM output.

To solve this problem there is a second 8-bit register hidden in the processor. When we read or write the *lower part* of a 16-bit register, then the upper 8-bit is written into, or read from this hidden register by the CPU. This means that when handling 16-bit registers the *order of accessing* the lower and upper parts is very important.

When *reading* a 16-bit register, such as theOCR1A we should always *read the lower byte first* (in this case OCR1AL). This way simultaneously to our reading operation the CPU copies the current value of the higher byte to the hidden register. When we next read the higher byte, the CPU will give us the correct value from the hidden register, which is coherent with the lower byte.

Writing a 16-bit register is the other way around, *the upper byte must be written first*. When writing the upper byte (such as OCR1AH), the CPU writes the value into the hidden register instead of the actual register. Then, when we write the lower byte, the CPU copies the previously saved higher byte to the real register as well in one operation, avoiding an incoherent value in the peripheral. You can read more about this operation on page 87 under section 16.3 of the datasheet.

Further Possibilities

The inner workings of the melody playing code is explained in the comments by the code, so we will omit the explanation here. However, I suggest after understanding the operation of the PWM peripheral, you make changes to the example code according to your own ideas, and create your own PWM 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).