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

Copyrights

This curriculum is the intellectual property of the partnership led by Xtalin Engineering Ltd., as the coordinator. The materials are designed for educational use and are therefore free to use for this purpose; however, their content cannot be modified or further developed without the written permission of Xtalin Engineering Ltd. Re-publication of the materials in an unchanged content is possible only with a clear indication of the authors of the curriculum and the source of the original curriculum, only with the written permission of Xtalin Engineering Ltd.

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

21 - Immediate Events in the Software, Interrupts

Written by Péter Farkas, Csaba Apró

English translation by Xtalin Engineering Ltd.

Revised by Szabolcs Veréb

Better an interrupt today, than an operating system tomorrow :)

INTRODUCTION

The electronic devices and systems that surround us usually don't work on their own, but they respond to events in the outside world. The systems and their inputs can be relatively simple, such as a remote and its buttons, or complicated such as the autopilot of an aircraft. Even though an aircraft autopilot sounds very exciting, we will stick to a simpler example to understand one of the key systems inside a microcontroller and how to use it.

DEMO CIRCUIT

In this chapter we will write a program which can count between 0 and 7 and increases or decreases the counter by one on button press. The current counter value is displayed in binary with three LEDs.

On the demo circuit to buttons are connected to two pins of the ATmega16A microcontroller, one to PD2, and one to PD3. These buttons will act as the up and down counter input buttons respectively. The three LEDs are connected to PA0, PA1, and PA2. You can see the schematic in the figure below.

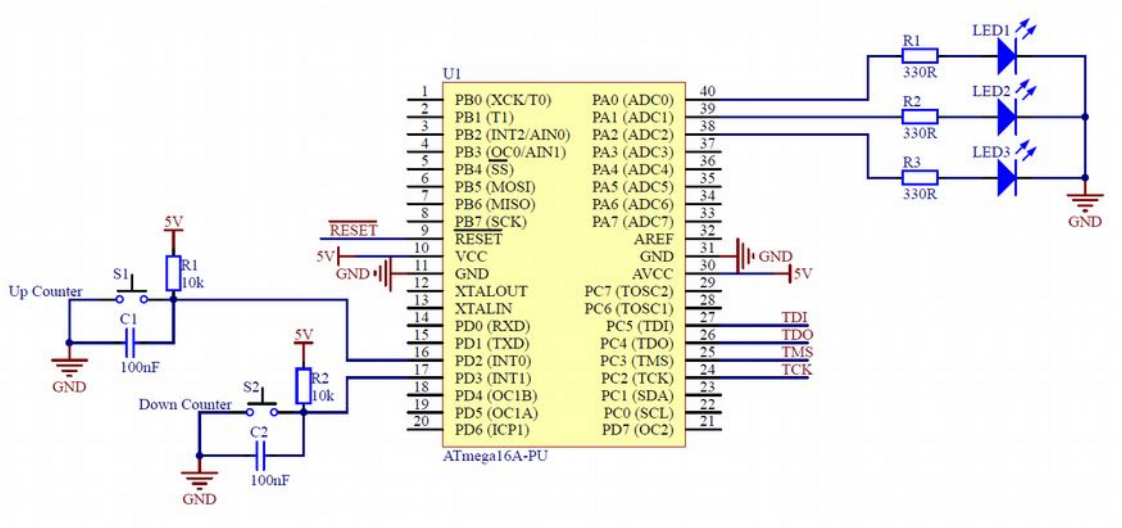

Figure 1 – Schematic of the demo circuit

Erasmus+

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

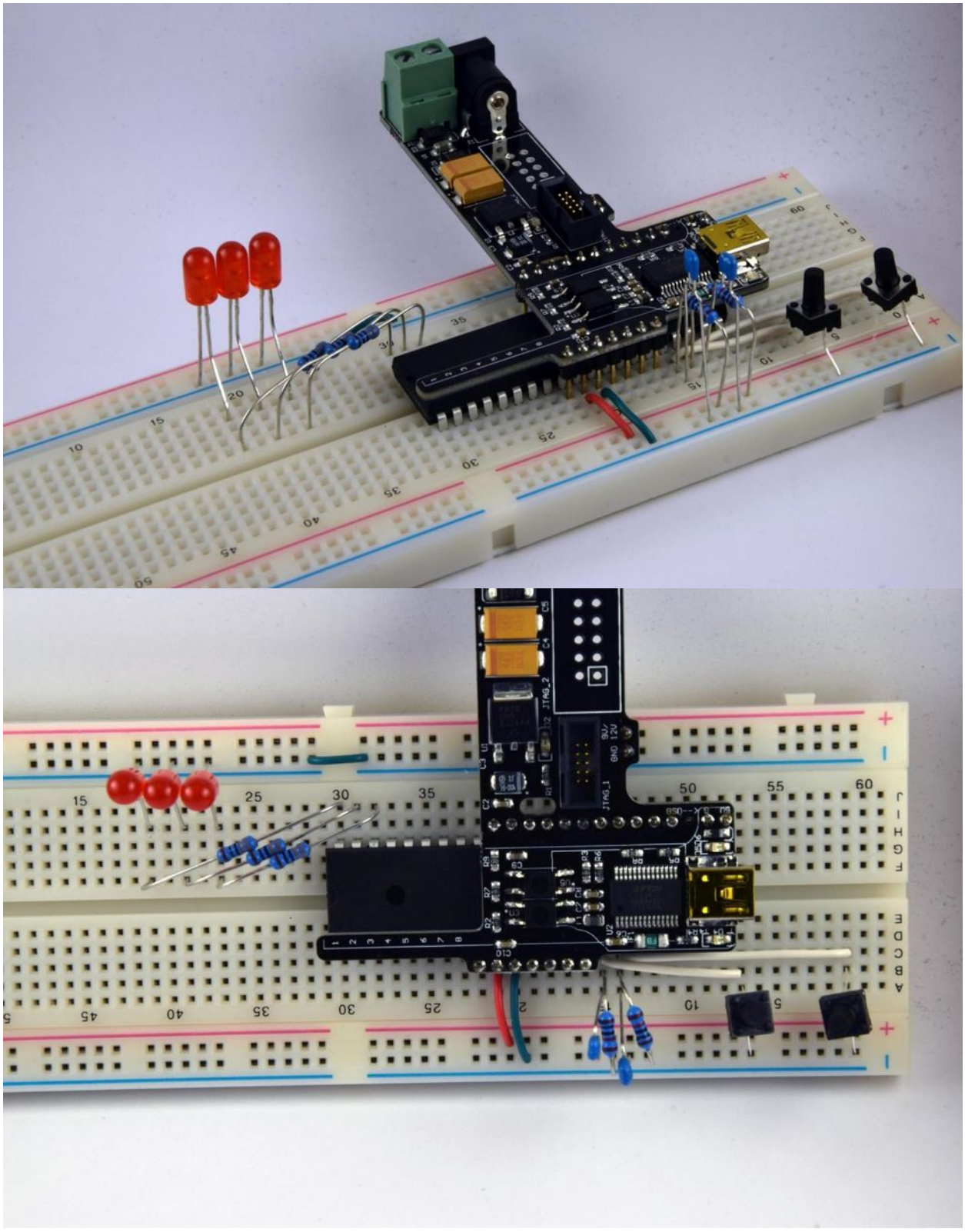

Figure 2 – Assembled demo circuit on breadboard

As you can see, there is a pull-up resistor between the buttons and the power supply.

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

Debouncing

A capacitor is connected to the input pin of the microcontroller and ground to filter the effect of switch bounce.

An ideal push button would immediately close the circuit the moment it is pressed and open it when released. In reality, before the circuit is closed the two contacts touch and come apart multiple times rapidly, similar to how a ball is bouncing up and down when thrown. This happens every time we turn on the lights, but our eyes can't see it. However, a microcontroller is much faster than our eyes, so it can detect these changes and you will see multiple button presses in software. The connected capacitor filters these fast changes. In the figure below you can see the waveforms on the microcontroller input while pressing a button. The first pictures contain the signal without filtering, and in the second you can see the effect of the filtering.

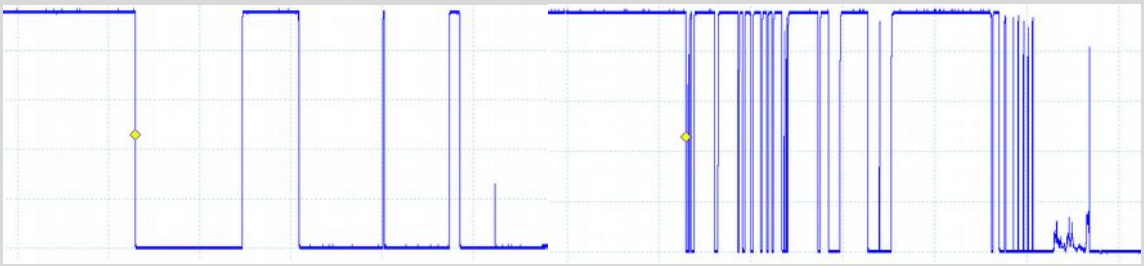

Figure 3 – Bounce of button

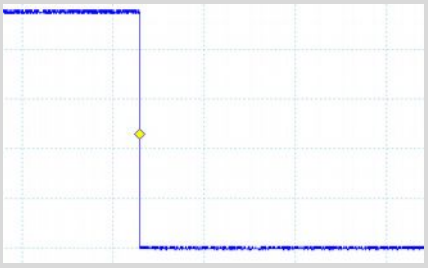

Figure 4 – Debounce using hardware

You can see that the filter capacitor gets rid of the bouncing effect, so we don't have to do anything special in software, the microcontroller will only see one input change. Of course, there are options to debounce in software too, but let's start at the beginning.

How to Detect a Button Press?

One solution is to enter an infinite loop in the main part of our program and read the input which has the button connected to it continuously. If the input is 1, someone pressed the button, and we can do the counter increase or decrease.

In our example there is a small difference, since the button input has a logic high (1) value when the button is not pressed. If you look at the schematic from before you can see that if the button is not active (not pressed) the pull-up resistors connect the input to the supply voltage, making it logic high level. When 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).

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

button is pressed, the input gets connected to ground via the button. This is called an “active low” connection. It also means that the button press can be detected in software with reading a “0” from the PD2 or PD3 input.

The program begins with IO initialization as usual; we configure the ports as input and output the same way we have learned before in io.c. The main program code looks like the following:

```
#include "../Headers/main.h"
volatile uint8_t cntr = 0;
int main(void)
{
	//IO initialization
	IOInit();
	//Endless cycle
	while (1)
	{
		//If we push the button that is connected to the PD2 pin AND
		// the counter hasn't reached the maximum yet
		if (!(PIND & 0x04) && (cntr < 7))
		{
			cntr++;
		}
		//If we push the button that is connected to the PD3 pin AND
		// the counter hasn't reached the minimum yet
		else if (!(PIND &  0x08) && (cntr > 0))
		{
			cntr--;
		}
		//Lighting up the LEDs accordingly to the counter
		PORTA = cntr;
	}

	return (0);
}
```

After initialization at the beginning of main (`IOInit()`) the software increments or decrements the value of a counter if one of the buttons is pressed, while taking into account the maximum and minimum values of the counter. At the end of the loop we set the output according to the counter.

The exclamation mark in the `if` conditions is the logical negation (not) operator, so the value of `!(PIND & 0x04)` is true if the PD2 pin has a logical low value (the third bit from the right in the `PIND` register is zero), which is exactly when the button is pressed.

What do we see when running the program? At the moment we press either button, all LEDs turn on or off.

That’s not exactly what we intended, but what causes the weird behaviour? Think about how long an average button press is. No matter how fast you press and release the button, you’ll surely keep it pressed for a few hundred milliseconds. What is the clock speed of the microcontroller? Completing one iteration of the infinite loop takes a lot more than one clock cycles, but with the microcontroller running at 8 MHz

Erasmus+

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

it still only takes a few milliseconds (or less). So, while our super-quick button press seems very fast to us, the microcontroller has time to examine the input a few hundred times while the button is pressed. As the input is always 0 during that time, it increases the counter until it reaches the limit (0 or 7). This all happens so fast that the individual counting of the LEDs cannot be perceived by our eyes, we only see the end result of no LEDs or all LEDs lighting up.

How can we solve this problem? We could try and save the current state of the button (0 or 1), and only increase or decrease the counter if the previous button state was 0 (pressed), and the current button state is 1 (released). This means we wouldn't wait for the button state, but for the moment the button is released (0->1 transition, rising edge of the signal), and only change the counter then.

There is an issue with this solution as well, that we are not seeing in this simple example. In a real-world application the microcontroller has many tasks that it needs to do in every iteration, and that usually takes time. To illustrate the problem, add a `for` loop inside our main loop that does nothing useful, but count to X. It could seem that this has no purpose, but we only need something that runs for a while to emulate a microcontroller busy with real-world tasks.

```
//Create a big enough variable outside the main function
uint32_t i;

//Imitating a task which requires a lot of time somewhere within the while(1)
for (i=0; i<500000; i++)
{
}
```

When running the altered code two things can happen after you press a button. Either nothing happens, if you have pressed the button at the wrong time, or did not hold it down long enough, or the software works, and it counts up and down one-by-one.

Think about why this happens! For the better part of the main loop our microcontroller is busy with counting to X, so it can only look at the buttons for a short amount of time before going back to counting again. If the button is pressed when the microcontroller is not looking, it will miss the event, and nothing happens. This is obviously not a reliable operation.

There should be a way of detecting a rising edge in the signal (button release) while doing something else. Luckily, others have ran into the same problem before, and came up with the solution called…

Erasmus+

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

THE INTERRUPT

The name i*nterrupt* really speaks for itself.

We can use dedicated inputs of the microcontroller, where if a pulse or a logic level change occurs the microcontroller interrupts whatever it was doing and executes a special function, we can specify the contents of. In the function we can react to the event and after that the microcontroller continues execution from where it left off.

This function is called *external interrupt* indicating that the microcontroller enters an interrupted state because of an external event of one of its digital inputs. We call this function *interrupt handler*.

Now we can see that this will provide solution for the above-mentioned problems, however more questions might arise.

- What other types of interrupts exist?
- How does the microcontroller recognize the change of the input while it is running the main loop?
- What happens when new (one or more) interrupt events occur while the interrupt routine is still running?
- Will we be able to notice this?
- Will this end the running interrupt routine?

These questions will be answered if we understand how this part of the microcontroller works.

We can set interrupts for several peripherals of the controller, such as the digital input, timers, ADCs, communication peripherals. Each peripheral hardware operates according to the values set in their respective configuration registers and they change values in the output registers during their operation.
In the simplest case, the microcontroller pin works as a digital input because the configuration bits associated to the controller has been set for this purpose. When digital inputs are configured as interrupts, also means that the interrupt bits (a bit of a hardware register) of a certain pin sets to 1 during the interrupt event. The technical term for this is “to set the interrupt flag to 1”. The key is that this will allow the hardware to set the interrupt while the programme is still running.

We can set interrupts for several peripherals of the microcontroller too (for example: digital input, timer, ADC, communicational peripherals). Each periphery and hardware work accordingly to the set values in the belonging configuration registers during their operation they change the values of the output registers. In the simplest case the given leg of the microcontroller works as a digital input because we set its belonging bits accordingly to our desired operation. If we configure a digital input as an interrupt input, then we also achieve that if an interrupt event happens then the belonging interrupt bit of the given leg will be set to 1 (a bit of a hardware register). (The technical term for this is the following: it set the i*nterrupt flag* to 1) The main essence is that it can happen with a hardware at the same time while the program runs.

Erasmus+

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

Datasheet of the ATmega16A contains a table called *interrupt vector table* (IVT) which summarises all the possible interrupt inputs:

Table 11-1. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	TIMER2 COMP	Timer/Counter2 Compare Match
5	$008	TIMER2 OVF	Timer/Counter2 Overflow
6	$00A	TIMER1 CAPT	Timer/Counter1 Capture Event
7	$00C	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	$00E	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	$010	TIMER1 OVF	Timer/Counter1 Overflow
10	$012	TIMER0 OVF	Timer/Counter0 Overflow
11	$014	SPI, STC	Serial Transfer Complete
12	$016	USART, RXC	USART, Rx Complete
13	$018	USART, UDRE	USART Data Register Empty
14	$01A	USART, TXC	USART, Tx Complete
15	$01C	ADC	ADC Conversion Complete
16	$01E	EE_RDY	EEPROM Ready
17	$020	ANA_COMP	Analog Comparator
18	$022	TWI	Two-wire Serial Interface
19	$024	INT2	External Interrupt Request 2
20	$026	TIMER0 COMP	Timer/Counter0 Compare Match
21	$028	SPM_RDY	Store Program Memory Ready

Figure 5 – Interrupt vectors in the datasheet

Reading on the datasheet, it also shows that in case there are multiple interrupt flags set to 1 at a given time, the handler with the smaller interrupt vector number will run first. For instance, in case of INT1 and INT0 pins, INT0 will have priority.

The ATmega16A microcontroller has a level 1 interrupt controller, meaning that the running of the interrupt handler cannot be interrupted by another handler. The interrupt flag might get set however it can be handled only after the interrupt handler routine has completed its course.

Multiple Interrupts

In case of complex microcontrollers, we can set the priority of multiple interrupts, we can even set priority levels as well. This is called as Multi-level Interrupt Controller. In a system like this, an interrupt can only be interrupted by a higher-level handler that has higher priority. However, if there are two interrupt requests at the same time, the one with the higher priority will run first. So, if a new interrupt is detected the system stops the execution, it runs a new interrupt then goes back to the original interrupt and when it is finished with that, it returns to the main program.

Sticking with a similarly simple example as Atmel, the PIC18F series has two user-selectable priority levels. During the configuration of the peripheral interrupt you can assign the interrupt source to either high or low priority. This way, during the execution the higher priority interrupt can disable the low-level interrupt, however the ones with same priority cannot disable each other.

Erasmus+

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

Now we can see when new and different types of interrupts arrive while the interrupt handler runs then the corresponding flags will be set however, the occurrence of the same type of interrupt (for example INT1 input) will not be registered by the hardware.

So, the interrupt is a very effective tool, but you have to know how to use it. But be careful, the interrupt handler name might be confusing! It is possible, although not recommended to implement all the event handling code in the handler. Always try to implement the bare minimum what is necessary inside the handler (such as saving some data specific to the interrupt – button pushed, critical temperature above 100 ℃, etc.). Everything else should be done in the main program when we really need the result (for example we turn on the fan). This way we can decrease the chance we miss another interrupt.

SETTING UP AN INTERRUPT

In order to use this function of the microcontroller, we need to enable the interrupt request of the given peripheral and the global interrupt of the microcontroller. This decision can be implemented by setting the corresponding bit (with the `sbi()` macro) of the peripheral register and setting the global interrupt enable bit to 1.

Sometimes we do not want the main program to be interrupted at a certain point (time and, or safety-critical task) by one or even all the peripheral interrupts.

We are able to achieve this by setting the corresponding bits to 0 (by using the `cbi()` and/or `cli()` macros).

This way the interrupts that come in during the process will only be handled after the re-enablement.

Enable or Disable Global Interrupts

While running the interrupt routine, the controller automatically modifies some of the bits. After entry, it blocks the interrupts and deletes the flags belonging to these interrupts, at the end of the handler it enables the interrupts again (as an effect of the RETI instruction automatically embedded by the compiler. The technical terms used in this case are 'to enable (Cli command, `cli()`) and disable (Sei command, `sei()`) global interrupt flags.

Erasmus+

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

SOLUTION OF THE COUNTER PROBLEM

After getting to know the interrupts in detail, let's get back to our counter task.

From now on, we will handle the buttons with interrupts, and we will increase or decrease the counter value in the handler. We have to complete the previously used output/input function, `IOInit()` by setting and enabling interrupts. This will be implemented in the following program lines:

```
	//INT0 at rising edge, i.e. ISC00=ISC01=1 page 66
	sbi(MCUCR, ISC00);
	sbi(MCUCR, ISC01);
	//INTR1 at rising edge, i.e. ISC10=ISC11=1 page 67
	sbi(MCUCR, ISC10);
	sbi(MCUCR, ISC11);
	//External interrupt enable page 68
	sbi(GICR, INTF0);
	sbi(GICR, INTF1);
	//Global interrupt enable page 9
	sbi(SREG, 7);
```

We have to modify the content of the demo project main.c file for the following:

```
#include "../Headers/main.h"
volatile uint8_t cntr = 0;

// + button interrupt
ISR(INT0_vect)
{
	//If the counter hasn't reached the maximum yet, then increase it
	if (cntr < 7)
	{
		cntr++;
	}
}

// - button interrupt
ISR(INT1_vect)
{
	//If the counter hasn't reached the minimum yet, then decrease it
	if (cntr > 0)
	{
		cntr--;
	}
}

int main(void)
{
	//IO initialization
	IOInit();

	//Infinite loop
```

Erasmus+

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

```
	while (1)
	{
		//Lighting up the LEDs accordingly to the counter
		PORTA = cntr;
	}

	return (0);
}
```

Let's see how it works in detail: when calling the `IOInit()` function, following the setup of the direction of the digital pins, the `sbi(reg, n)` macros are used to set the bits of the microcontroller registers to 1. The first four `sbi(reg, n)` instructions setup the external interrupts, the following two enables them and the last one enables the global interrupts. The global enablement and disablement of interrupts can be also done by the `sei()` and `cli()` parameterless macros, which makes our program more readable.

In the file main.c. you can find new types of functions, called ISR (Interrupt Service Routine, also called Interrupt Handler) functions that have the parameter `xxx_vect`. These are the interrupt handler functions that were mentioned earlier, they are only executed when an interrupt happens. Their parameters start with a name corresponding to the interrupt vector (found on page 44 of the datasheet under Interrupt Vector section) and ends with the suffix `_vect`.

In our code, we increment or decrement the counter while paying attention to the limits in our interrupt handlers.

Within the infinite loop of the main function the only thing left to do is set port A to the value of the counter, lighting up the LEDs.

Because the button is debounced with a (filter) capacitor and we configure the external interrupt to detect a rising edge, nothing will happen if we push either of the buttons until we release them. Then a 0->1 transition will occur on the input pin which generates an interrupt, and the control goes to the appropriate handler.

ISR Functions

The ISR functions are not the typical functions we are used to, because `ISR()` is just a macro which lets the compiler know that the content is an interrupt handler.

The compiler automatically puts the RETI instruction at the end of the generated assembly for this block, which re-enables the global interrupts disabled by hardware at the start of the interrupt handler. This way we don't have to put an enable interrupts instruction at the end of our handler.

It is also worth to mention that we cannot call interrupt handlers explicitly like other normal functions we define from our code.

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

TIMERS AND INTERRUPTS

If it hasn't been clear by now, we would like to emphasise again: interrupts are a very important element of embedded software development. The reason for this is that in microcontrollers most peripherals are capable of generating interrupts. An interrupt can not only come from an external pin of the controller, but it can also be generated by an internal peripheral such as the analogue-digital converter signalling that the conversion is complete (more about this in the next chapter), or if any communication peripheral receives a message it can fire an interrupt to let us know.

From the title of this section you might have already guessed that usually we can get an interrupt if a timer peripheral has "expired".

In a previous chapter we have already talked about timers but then we did not use interrupts. Let's have a look at an example where we utilize a timer with interrupts.

In the previous example we have solved the bouncing button problem with hardware debouncing in the form of two filter capacitors. Let's remove these now and implement software debouncing instead.

After removing the capacitors try to slowly press and release the button. In most cases the counter will count more than once in its direction. This is because of the button bounce we have already talked about, and it is very visible with this software.

To avoid it we do the following: in the interrupt handler of the buttons we disable any interrupts coming from the same source and start a timer which we have set up previously to have close to a 100 ms period. When the timer "expires" another interrupt handler will get called, where we re-enable the interrupts coming from the button.

This solution makes the system only detect the first rising edge, because the other edges that are caused by the bounce will get ignored as the interrupt is disabled. The approximate 100 ms set in the timer is enough time for the button to stop bouncing, but it is small enough to almost always detect two consecutive intentional button presses.

Create a new source file called "timer.c" in which – according the above – configure the TIMER1 peripheral with a prescaler of 8. The contents of the `TimerInit()` function are the following:

```
    //prescaler of 8, according to page 108
    cbi(TCCR1B, CS12);
    sbi(TCCR1B, CS11);
    cbi(TCCR1B, CS10);
```

Because the clock frequency of our controller is 8 MHz, the clock signal of the timer will be one-eighth of that. Because the timer has a 16-bit counter it will overflow in:

$$\frac{8\ [\mathrm{MHz}]}{8 \bullet 2^{16}} \approx 15.26\ [\mathrm{Hz}] \Longrightarrow \frac{1}{15.26\ [\mathrm{Hz}]} = 65.536\ [\mathrm{ms}]$$

We will use this time to debounce the buttons.

The contents of "io.c" can remain the same, as at startup we want the button interrupts to be active.

Erasmus+

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

In the “main.c” file, both external interrupt handlers have to be amended with a reset timer and the overflow interrupt enable which is accomplished with the following lines:

```
	//Timer1 counter reset
	TCNT1 = 0;
	//Timer1 overflow interrupt enable
	sbi(TIMSK, TOIE1);
```

Furthermore, the external interrupts have to disabled so the microcontroller won’t recognize the rising edges coming from the bounce.

```
	//INTR0 interrupt disable
	cbi(GICR, INTF0);
```

Then a new interrupt hander has to be created for the timer overflow, in which we re-enable external interrupts, clear all interrupt flags that might have been set before the timer expired, and disable the interrupt of the timer.

```
//Timer1 overflow interrupt
ISR(TIMER1_OVF_vect)
{
	//External interrupts enable, page 67
	sbi(GICR, INTF0);
	sbi(GICR, INTF1);

	//External interrupt flags clear, page 68
	cbi(GIFR, INTF0);
	cbi(GIFR, INTF1);

	//Timer1 overflow interrupt disable
	cbi(TIMSK, TOIE1);
}
```

Finally we have to call `TimerInit()` and `IOInit()` in main, and set the LEDs by assigning the counter value to PORTA.

Try to put the counter loop in main that keeps the controller busy and see what happens!

```
	//Imitating a time-consuming task somewhere within the while(1)
	for (i=0; i<500000; i++)
	{
	}
```

While not always immediately, but all button presses are correctly registered, and the LED counter is updated. The processor spends most of its time counting up in the for loop, then updating the LEDs according to the counter, and getting back to the for loop again. But interrupts are fired at each button press, the value of the counter is updated “in the background”.

Erasmus+

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.

Exact timing

In our example it is not really important whether we wait for exactly a 100 ms or for around 65 ms after each button press, so we were using the timer mode which was simpler to implement.

If you need to wait exactly a 100 ms, we have to use the timer in Output Compare mode. Here, the timer generates an interrupt when the counter reaches an exact pre-programmed value contained in OCR1AH and OCR1AL and/or OCR1BH and OCR1BL registers.

It may very well be that the 65 ms turns out to be too short (it depends on the push button itself, and how fast are you pressing it) and we need to leave more time to debounce.

To overcome that problem, we can use the timer in Output Compare mode as mentioned earlier, or we can choose a bigger prescaler, making it run slower. However, by implementing any of these we leave a greater window where the external interrupts are disabled, so we can increment or decrement the counter slower. The optimal solution is always application-specific, and depends on many factors such as how long does our button bounce, is there a free timer available, how fast impulses we need to detect, etc.

FINAL WORDS

We have reached our goal set at the beginning of the chapter; we have learned about interrupts and their quite important role in embedded systems. We have seen one example for internal and external interrupts each, which I hope will be useful for you in the rest of the curriculum, and in hobby projects or later studies.

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

This is xxx personal copy - distribution is strictly prohibited.

http://crystalclearelectronics.eu | All rights reserved Xtalin Engineering Ltd.