BASIC INTERRUPTS AND I/O

AN INTRODUCTION TO INTERRUPTS AND I/O

AVR EIVIND, AVRFREAKS.NET

OCT 2002

TABLE OF CONTENTS

Lets' get physical

This article is a small project for you people who are just getting into the AVR, and perhaps even microcontrollers in general. We take things a step further than in the previous article from AVRfreaks, where we simply concentrated on assembling a simple (virtual) LED-flashing application in AVRstudio. No external hardware requirements were done. But this time we'd like to expand a little, and add some external hardware.

The natural place to start is the **STK500** (http://www.avrfreaks.net/Tools/showtools.php?ToolID=115). It is a very nice development board for the AVR, reasonably priced (~USD79) and provides all the environment we need to test some pretty real applications on the AVR out in the wild.

We're gonna start out with some simple counting controlled by external interrupts and exposed on the nice LEDs of the STK500. Then we'll add a speaker (Oh yeah!), and before we know it we'll have a miniature amusement park on our desks; with lights AND noise and buttons to push! Perhaps fire as well, if something REALLY goes wrong.

This is what we'll use:

1. **AVRstudio** 3 or 4
2. **STK500** development kit, all set up with your computer and ready to go
3. An **AT90s8515** microcontroller (usually comes with the STK500)
4. Some small **speaker** that works, including wires soldered in place

The setup is based on a Windows configuration, but it is very possible use some other software as well, since we won't concentrate much on the use of AVRstudio besides assembling the project. If you are a Linux user, you could use:

- WinAVR (http://www.avrfreaks.net/Tools/showtools.php?ToolID=376)(i.e. avrasm) for the assembly
- uisp for programming (http://savannah.nongnu.org/projects/uisp/)

The program will be written in **assembly**, because:

- assembly is very "machine-near" and provides a very educative approach to what goes on inside the processor during our program
- high-level languages and different compilers all have different notations and routines for doing the same thing. Learning a compiler and the respective C-style (e.g.) is a story of itself.

The code for this project is something we found among leftovers from O'Guru *Sean Ellis*; which we brutally and without due respect ripped apart. Shame on us.

Basic interrupts

An interrupt is a flow control mechanism that is implemented on most controllers, among them the AVR. In an MCU application interacting with the outside world, many things are happening at the same time, i.e. not in a synchronized manner, that are to be handled by the microcontroller.

Examples: a switch pressed by the user, a data read on the UART (serial port), a sample taken by the ADC, or a timer calling to say that "time is up!". All these events neeeds to be handled by the MCU.

Instead of polling each instance round-Robin style to ask whether they are in need of a service, we can have them call out themselves when they need attention. This is called "interrupts", since the peripheral device (e.g. a switch pressed) *interrupts* the main program execution. The processor then takes time out of the normal program execution to examine the source of the interrupt and take the necessary action. Afterwards, normal program execution is resumed.

An interrupt service in other words is just like a subroutine; except that it is not anticipated by the processor to occur at a particular time, since there are no explicitly placed calls to it in the program.

What's in a name?

When reading this article you will from time to time get the feeling that you are confronting a term possibly denoting an actual physical entity or an entity in some sense relevant to the current activity; namely playing around or building serious applications with the AVR...: ***INT0, INT1, GIMSK, PORTB, PB7*** etc...

You are sure to come across such names in any assembly code, Atmel appnote, AVRfreaks Design Note or any posting in the AVRforum.

One might think these are just common names used by individuals accustomed to the jargon, but we will try to use them consciously - in the sense that these names actually denote actual memory locations in the AVR you will be programming.

The mapping of these name to actual memory locations is in the part's **def** file (*def.inc). Have a look at the included def file for the 8515 each time you come across an unknown term:

8515def.inc **Example snippet;** only a few lines are shown

(~6kB)
File included in zip

```
;***** I/O Register Definitions
.equ SREG  =$3f
.equ SPH   =$3e
.equ SPL   =$3d
.equ GIMSK=$3b
 ..   ..    ..
 ..   ..    ..
```

When including this file in the assembly program file, all I/O register names and I/O register bit names appearing in the data book will be known to the assembler and can be used in the program.

Note that some high-level language compilers may use proprietary terms other than these. But they *will* have files similar to this def file, defining the memory space of the AVRs. As previously stated; this is another story.

Another document that will prove very useful to anyone working with the AVR, is this document:

8515 datasheet

(~2MB)
File included in zip

The datasheet. The datasheet is the ultimate reference for **any** AVR microcontroller. It even includes an instruction set summary; look up every instruction you don't know when you come across it!

In this article, we will be using the 8515. Keep this .pdf close for reference.

The datasheet. The datasheet is the ultimate reference for **any** AVR microcontroller. It even includes an instruction set summary; look up

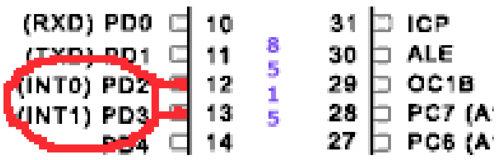
Now you know where to look when anything unknown pops up. Let's move on!

Structure of an interrupt-driven program on the AVR

Take a deep breath. This is the heaviest part.

We are going to write an "interrupt-driven" program where the main loop simply does nothing but wait for interrupts to occur. What interrupts?

External interrupts INT0 and **INT1** on pins **PD2** and **PD3**



The interrupts are handled in turn, and a return to the main program is performed at the end of each interrupt service (that's what I call it; "service").

This is a rather wide topic with many pitfalls. But we need somewhere to start and will mainly discuss aspects concerning elements of our little example application. The main important thing that constitutes such elements in a program is:

1. Setting the interrupt vector jump locations: **.org**
2. Setting the correct interrupt mask to enable desired interrupts: **GIMSK**
3. Make necessary settings in control registers: **MCUCR**
4. Globally enable *all* interrupts: **SREG**

Setting the interrupt vector jump locations: .org

The lowest part of the AVR program memory, starting at address $0000, is sometimes referred to as the *"Program memory vector table"*, and the actual program should start beyond this space. The vector table is reserved for storing interrupt vectors; i.e. locations to jump to when this or that interrupt is calling. This means that each interrupt has a reserved memory location, and when a particular interrupt comes in, the MCU looks in this location to find the address where code that handles this interrupt resides.

8515 Vector table			**Example**; only the few first vectors are shown
Program memory address	Vector	Comment	The number of interrupts available varies from processor to processor.
$0000	Reset	Start address of Reset handler is stored here	
$0001	INT0	Start address of code to handle external INT0 is stored here	
$0002	INT1	Start address of code to handle external INT1 is stored here	
etc...	...	...	

The .org directive

In assembly code, the **.org** directive is used to set vector jump locations. This assembler directive (or "command", if you like) tells the assembler to set the location counter to an absolute value. It is **not** part of the AVR instruction set, it is just a command that the assembler needs to make sure the program code is mapped correctly when making a binary for the AVR. **Example:**

Sample Code

```
; Interrupt service vectors
; Handles reset and external interrupt vectors INT0 and INT1

.org $0000
      rjmp Reset  ; Reset vector (when the MCU is reset)

.org INT0addr
      rjmp IntV0  ; INT0 vector (ext. interrupt from pin PD2)

.org INT1addr
      rjmp IntV1  ; INT1 vector (ext. interrupt from pin PD3)

; - Reset vector - (THIS LINE IS A COMMENT)
Reset:
      ldi   TEMP,low(RAMEND)  ; Set initial stack ptr location at ram
end
      out   SPL,TEMP
      ldi   TEMP, high(RAMEND)
      out   SPH, TEMP
      ...
      ...
```

Note that **labels** are used instead of absolute numbers to designate addresses in assembly code - The assembler stitches it all together in the end. All we need to do is tell the assembler where to jump when e.g. the **reset** vector is calling, by using the name of the code block meant for handling resets.

A **label** denotes a block of code, or *function* if you like; which is not terminated with a *"}"*, an *.endfunc* or anything like that. The only thing that ends a code block definition, is it being released by another block name, followed by a colon (":").

This also implies, unlike with functions in e.g. C, that all blocks are run by the processor consecutively, unless the flow is broken up by un/conditional jumps, returns, interrupts etc. In assembly, *the whole file* is the *main()* function, and the flow control is more like Basic...

Please also note the first lines of the **reset handler**. This is where the **stack** is set up. The stack is used to hold return addresses in the main program code when a sub- or interrupt routine is run; i.e. when a "digression" from the main program is made.

For any interrupt service or subroutine to return to the main program properly; the stack must be placed outside their vector space. The **SP** is namely initialized with the value **$0000**, which is the same location as the reset vector. This goes for **any** program, especially such as this, where we are involving several interrupt vectors besides the *reset* vector.

For AVRs with more than 256 bytes SRAM (i.e. none of the Tinys, nor 2343 and 4433), the Stack Pointer register is two bytes wide and divided into **SPL** and **SPH** (low and high bytes).

Setting the interrupt mask: GIMSK

The GIMSK register is used to enable and disable individual external interrupts.

GIMSK									General Interrupt Mask register
Bit	7	6	5	4	3	2	1	0	**Note** that only the INT0 and INT1 bits
	INT1	**INT0**	-	-	-	-	-	-	are writable. The other bits are reserved
Read/write	R/W	R/W	R	R	R	R	R	R	and always read as zero
Init. value	0	0	0	0	0	0	0	0	

We are going to use the external interrupts INT0 and INT1 for the switches on the STK500. These interrupts are enabled by setting INT0 and INT1 in GIMSK; i.e. bits 6 and 7.

General control register: MCUCR

MCUCR									MCU general control register
Bit	7	6	5	4	3	2	1	0	The bits in MCUCR allow general processor control. Consult the datasheet for an in-depth description of the registers and the individual bits.
	SRE	**SRW**	**SE**	**SM**	**ISC11**	**ISC10**	**ISC01**	**ISC0z0**	
Init. value	0	0	0	0	0	0	0	0	

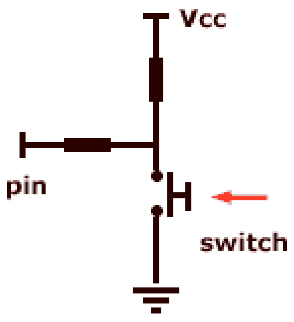
We will be using bits 0,1,2 and 3 in this register to control the interrupt from

INT0 and INT1. These bits control how to sense the external interrupts; either by level, falling edge on pin, or rising edge of pin:

ISCx1	ISCx0	Description
0	0	Low level on INTx pin generates interrupt
0	1	Reserved
1	0	Falling edge on INTx pin generates interrupt
1	1	Rising edge on INTx pin generates interrupt

We will use the **rising edge** of the switches on the STK500 to trig the interrupt; so the 8515 must be programmed to trig external interrupts on rising edges of each pin PD2 and PD3. Hence; all the ISCx bits must, for our program, be set to "1".

You can see on the diagram to the right how pushing the switch will close the lower branch and pull the pin low. Hence; releasing the switch causes a rising edge when the branch is re-opened and the pin is pulled high.



Globally enable all interrupts: SREG

In addition to setting up the interrupts individually, the **SREG** (Status Register) **bit 7** must also be set to globally enable all (i.e. any) interrupts.

SREG									Status register
Bit	7	6	5	4	3	2	1	0	The bits in SREG indicate the current state of the processor.
	I	**T**	**H**	**S**	**V**	**N**	**Z**	**C**	

Init. value	0	0	0	0	0	0	0	0

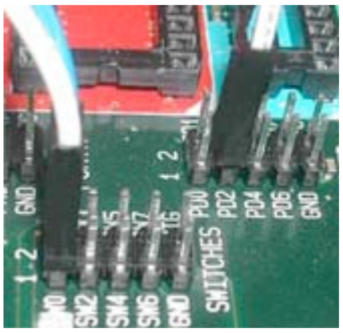
All these bits are cleared on reset and can be read or written by a program. **Bit7 (I)** is the one we are currently interested in; as setting this bit enables all interrupts. Vice versa, resetting it disables all interrupts.

In AVR code, we have an instruction of its own to set this flag; ***sei***:

```
                ; lots and lots of initialisation, and then...
        sei     ; this instruction enables all interrupts.
                ;...and off we go!
```

Real code part 1

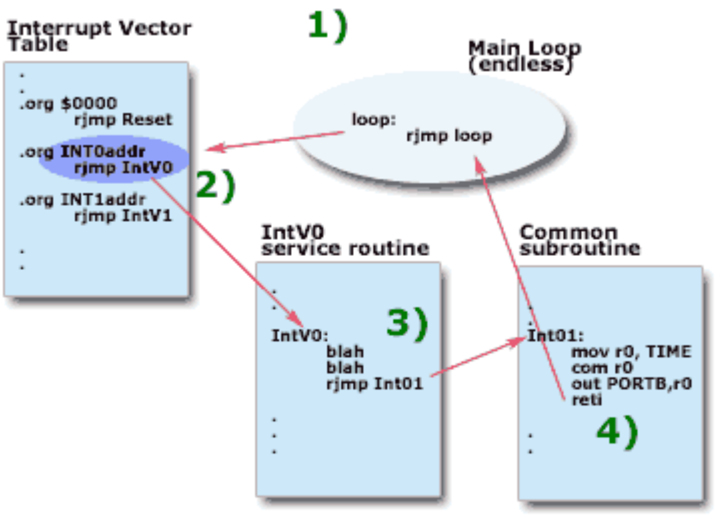
OK, let's start with the real code. Assuming you're already able to assemble your own code and even program the part in the STK500 - we'll just dig through the code.



Just remember to couple the switches with the appropriate inputs on the 8515; namely PORTD input pins **PD2** and **PD3**. Use any two switches on the STK500 you like; on the picture to the right I used switches **SW0** and **SW1**. Also connect the LEDs to PORTB with a 10-pin ISP connector cable.

When running this code on the STK500, at first all LEDs will be off. Press the switches a few times, and you will realize one of them counts something up, the other one down, and the results are reflected on the LEDs.

Let's have an overview of the program. Here's an example snapshot, after initialization:



```
.
.org $0000
        rjmp Reset
.org INT0addr
        rjmp IntV0
.org INT1addr
        rjmp IntV1
.
.
```

```
loop:
        rjmp loop
```

```
.
IntV0:
        blah
        blah
        rjmp Int01
.
.
```

```
.
Int01:
        mov r0, TIME
        com r0
        out PORTB,r0
        reti
.
```

1. A switch is pressed -> ext. INT0 generated
2. The vector for INT0 is found
3. Code at the according location is run, and jumps to a common subroutine
4. The common subroutine returns to the main loop by **reti** instruction

This is what our code will do. Nothing more. Besides initialization, the short routine for handling the other switch (generating **INT1**) and a few directives for the assembler, that's it all.

8515def.inc

(~6kB)

File included in zip

Just to make sure I'm still not kidding you; have a look in the 8515def.inc file and search for "INT0addr" and "INT1addr". Lo and behold; they are **real** addresses. **Reset** is placed at **$0000**.

OK, here is the entire program code, with some excessive comments removed (these are still left in the available file). Look up any unknown instruction for full understanding while you read through it. You can click each code block label to jump to their respective comments below.

INTs_1.asm **Source for first part of program**

(~3kB)

File included in zip

```
;---------------------------------------------------------------
; Name:           int0.asm
; Title:    Simple AVR Interrupt Verification Program
;---------------------------------------------------------------

.include "8515def.inc"

; Interrupt service vectors

.org $0000
          rjmp Reset          ; Reset vector
.org INT0addr
          rjmp IntV0          ; INT0 vector (ext. interrupt from
pin D2)
.org INT1addr
          rjmp IntV1          ; INT1 vector (ext. interrupt from
pin D3)

;---------------------------------------------------------------
;
; Register defines for main loop

.def      TIME=r16
.def      TEMP=r17
.def      BEEP=r18

;---------------------------------------------------------------
;
; Reset vector - just sets up interrupts and service routines
and
; then loops forever.

Reset:
          ldi       TEMP,low(RAMEND)         ; Set stackptr to
ram end
          out       SPL,TEMP
          ldi       TEMP, high(RAMEND)
          out       SPH, TEMP
```

```
        ser     TEMP            ; Set TEMP to $FF to...
        out     DDRB,TEMP       ; ...set data direction to
"out"
        out     PORTB,TEMP      ; ...all lights off!

        out     PORTD,TEMP      ; ...all high for pullup on
inputs
        ldi     TEMP,(1<<DDD6)          ; bit D6 only
configured as output,
        out     DDRD,TEMP       ; ...output for piezo
buzzer on pin D6

        ; set up int0 and int1

        ldi     TEMP,(1<<INT0)+(1<<INT1) ; int masks 0
and 1 set
        out     GIMSK,TEMP
        ldi     TEMP,$0f        ; interrupt t0 and t1 on
rising edge only
        out     MCUCR,TEMP
        ldi     TIME,$00        ; Start from 0

        sei                     ; enable interrupts and off
we go!


loop:
        rjmp    loop            ; Infinite loop - never
terminates

;----------------------------------------------------------------
;
; Int0 vector - decrease count

IntV0:
        dec     TIME
        rjmp    Int01   ; jump to common code to display
new count


;----------------------------------------------------------------
;
; Int1 vector - increase count

IntV1:
        inc     TIME    ; drop to common code to display
new count

Int01:
        mov     r0,TIME         ; display on LEDs
        com     r0
        out     PORTB,r0
        reti
```

OK, lets go through the code step by step, though at a pace. It may be easier if you have the source printed out next to you while reading the following comments: The **first lines** includes the define file for the 8515; thus making all register and I/O names known to the assembler. What happens next is the Interrupt vector table is defined. At **$0000**, the **reset** vector is set up. This is where the 8515 wakes up in the morning - everything is supposed to start from here. Also, the **INT0** and **INT1** vectors are set up, and their handling routines named **IntV0** and **IntV1**, respectively. Look up their labels down the code, and you can see where they are declared.

Following this, registers r16, r17 and r18 have labels put onto them. This is a way to make **variables** in assembly - only we also get to decide where they are placed in memory. Where? In registers r16, r17 and r18... hence; they are all **one byte** wide.

The reset label

Py-haa! The reset label contains all initialization code; this block is run at start-up. The first 4 lines sets up the stack pointer, as mentioned earlier. Note how the *ldi*(load immediate) instruction is used to hold any value temporarily before writing to the actual location by *out*. *low()* and *high()* are macros returning the immediate values of their arguments, which are memory locations defined in the .def file.

The next six lines sets the *Data Direction Registers* of ports PORTB (used for LEDs) and PORTD (switches). Please check the datasheet under "I/O Ports" for functional descriptions of these registers. Now, notice this line:

```
        ldi     TEMP,(1<<DDD6)
```

This line of code simply (!) means: *"Load TEMP register with a byte value of 1 shifted DDD6 places leftwards"*. Ok. Then what is DDD6? From the .def file, we find that this value is 6, and it is meant to point to the 6th bit of the PORTD Data Direction Register. The value loaded into **TEMP** and then into **DDRB**, becomes **01000000** in binary. Hence, the bit in this position in the DDRB register is set. So what happens? That pin (pin PD6) is to be used for a special twist in the next stage of the program, so that particular pin is set as an **output**; the others will be **inputs** For now, just notice the notation.

You can probably imagine what happens if you combine such notation in an addition? Well, this is what happens next, when the **GIMSK** register is loaded, and then the **MCUCR**. Please refer to the previous section or the datasheet for a description of these registers and why they are set this way.

Only thing remaining in the **reset** block now, is to call our friend the **sei** instruction for enabling the interrupts we have just set up.

The loop label

The **loop** label simply contains *nothing* but a call to itself. It's an equivalent of writing while(1); in C. After reset is run, the program pointer falls through to the loop block and it will run forever only interrupted by - *interrupts*.

The IntV0 label

This label comtains the handling code for the **INT0** interrupt. Whenever that interrupt calls, this code will be run. It will simply decrement the TIME register. Then it just jumps to a common block called...:

The Int01 label

This block consists of common code that displays the value of TIME (r16) on the

LEDs connected to PORTB. **Note** that the value is inverted by 1's complement (*com* instruction) before written to PORTB, since a low value means no light and vice versa. This block then performs a return to wherever it was called from through the *reti* instruction - which was from the **loop** label.

The IntV1 label

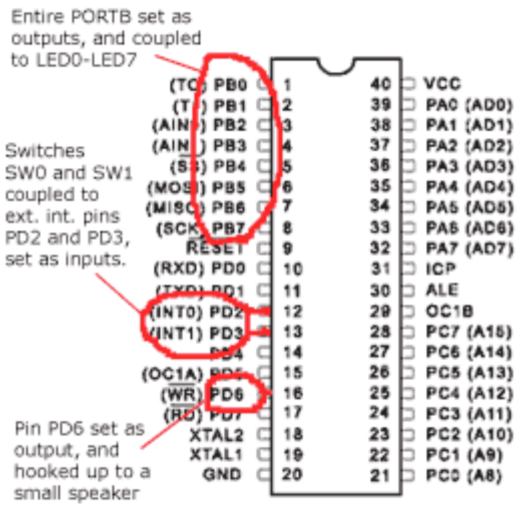
You've probably figured that this code runs every time the switch connected to pin PD3 is pressed (i.e. *released*, due to our MCUCR settings). It increases TIME. Then it *just falls through to the common routine Int01*, since it contains *no jump or return instruction*. We could just as well have put in an

rjmp Int01

here as well. But we don't need it. Though it may be good common practice to be consequent with this :-)

The timer overflow interrupt

After stating our success with the previous experiment, we are going to ameliorate this little design a little more. We are going to add another interrupt into the application; a *Timer Overflow* interrupt for the **timer/counter 0** of the 8515. Also, we're going to supply a small speaker to make some noise.



If you think the hardware requirements for this project are getting too demanding now, you don't **have** to hook up the speaker. It is for illustrative purposes only, and your code will work perfectly well without it.

Every which way; you will see how to set up the timer overflow interrupt and write handling code for it.

Timer overflow 0

The 8515 has **two** timer/counters; one 8 bits wide and one 16 bits wide. This means that they are capable of counting from any value you set, until they reach their limit which is determined by the number of bits available (256 or 65535, respectively). Then they will issue an interrupt, if you have set it up to do so. Upon overflow; the Timer/Counter just keeps counting "around" the range... so if you have set the timer to start from some special value and want it to start from there again; you will have to reset it to that value. What we need to do in the code, is to add three little blocks of code more. These are (could you guess them?):

1. Another interrupt vector, for the TimerOverflow 0 interrupt: **OVF0addr**
2. Initialization code for timer/counter 0: **TIMSK, TCCR0,TCNT0**
3. The interrupt handling subroutine.

OVF0addr

This is the name set in the **8515def.inc** file for the location where this interrupt vector should reside (check it, I may be pulling your leg). We add these two lines of code to the vector block:

```
; -- new interrupt vector -

.org OVF0addr
        rjmp TimerV0    ; T/C0 overflow vector
```

You are **very able** to read this now, and realize that it is just like the previous .org's in this program. Let's move on!

TIMSK, TCCR0,TCNT0

Together, these 3 registers are all we need consider to have timing interrupts in an application on the AVR.

TCCR0 controls the operation of Timer/counter 0. The count is incremented for every clock signal at the input of the timer. But the clock input can be selected, and **prescaled** by **N**. We'll just consider the 3 lowest bits of this register:

TCCR0									**Timer/Counter0 register**
Bit	7	6	5	4	3	2	1	0	**Note** that bits 7-3 are reserved, and always read as zero
	-	**-**	**-**	**-**	**-**	**CS02**	**CS01**	**CS00**	
Read/write	R	R	R	R	R	R/W	R/W	R/W	
Init. value	0	0	0	0	0	0	0	0	

This table shows the different settings of these 3 control bits:

CS02	CS01	CS00	Description
0	0	0	Stop the timer/counter
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	Ext. pin T0, falling edge
1	1	1	Ext. pin T0, rising edge

TIMSK; the Timer/Counter Interrupt Mask register is simply a "mask" register for enabling/disabling interrupts just like you have already seen with the **GIMSK** register:

TIMSK									**Timer/Counter Interrupt Mask register**
Bit	7	6	5	4	3	2	1	0	**Note** that bits 4,2 and 0 are reserved, and always read as zero
	TOIE1	**OCIE1A**	**OCIE1B**	**-**	**TICIE1**	**-**	**TOIE0**	**-**	
Read/write	R/W	R/W	R/W	R	R/W	R	R/W	R	
Init. value	0	0	0	0	0	0	0	0	

...and again; the only thing you really need to know for this little tutorial; is the position of one special little bit: this one is called **"Timer/Counter0 Overflow Interrupt enable"**, abbreviated **"TOIE0"** and found in bit postition 1 of this register. To enable our Timer interrupt; set this bit (to "1"). **TCNT0** is the actual "Timer/Counter" register. This is where the timing and counting is done, in accordance with the settings in **TCCR0**. This is simply a register for storing a counter value; there are no special bits in it. It is entirely readable/writable; so you can load it with any desired starting value for your counting if you like. **Note** that it does not reset itself automatically, even if an interrupt is issued.

This is already becoming old news to you now, since it's just more or less another instance of registers controlling similar functions that you have already heard about regarding the external interrupts...

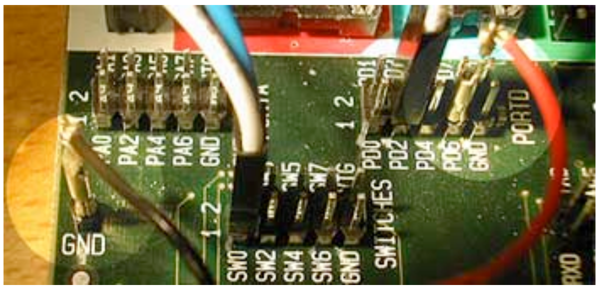
Real code part 2

For illustrating the Timer0 Overflow interrupt; we connect a small speaker to an output pin of the 8515. Each Timer0 overflow interrupt will toggle the pin. The result is that the speaker will buzz with a base frequency proportional to the frequency with which the pin is toggled. I.e. the base frequency will be:

CLK/2*Prescale*(256-TIME)

where TIME is the current value in the TIME register (r16).

Also, the two switches affecting the value in TIME will make the buzz frequency waver up or down.



The picture shows how to connect the speaker. We have chosen pin **PD6** for no particular reason.

Huh?	**Why is that the formula for the base frequency?**
?	The Timer/counter counts one up from some value every clock cycle until it overflows. Then we reset it, to repeat the drill. Let's say the timer can only count to 1 before overflow. Flipping the pin every time, will give us one cycle of a square-wave like waveform every 2 flips, right? (up a while, then down a while, repeat...). Hence, the base frequency would be: CLK/2 Now; the Timer/Counter register is 8 bits wide, and can count from any value it is set (TIME) to 255. The formula becomes: CLK/2*(256-TIME) Besides; we have a prescaler which, when set to N, makes the Timer count just every Nth cycle... CLK/2*N*(256-TIME)

These are the three snippets of code to insert. Please consult the complete source code (INTs_2.asm, available below) to where the snippets are inserted:

INTs_2.asm	**Source snippets** for second part of program
(~3kB) File included in zip	

```
;---------- CODE SNIPPET #1 - OVF0addr vector ----------
---------
; inserted BELOW the existing vector defs
;--------------------------------------------------------
---------
.org OVF0addr
        rjmp TimerV0    ; T/C0 overflow vector

.
.

;---------- CODE SNIPPET #2 - Initializing
TIMSK,TCCR0,TCNT0 ----
; inserted in the Reset: label, right before the 'sei'
call
;--------------------------------------------------------
---------
        ldi     TIME,$80        ; Start from 128. NB!
        out     TCNT0,TIME                   ; set
Timer/counter also.

        ldi     TEMP,(1<<TOIE0) ; timer overflow interrupt
enable 0
        out     TIMSK,TEMP

        ldi     TEMP,$02        ; clock prescaler = clk/8
        out     TCCR0,TEMP
.
.
;---------- CODE SNIPPET #3 - handling the Timer
overflow int. --
; new subroutine label, inserted at the end of the file
;--------------------------------------------------------
---------
TimerV0:
        out     TCNT0,TIME      ; reset time

        com     BEEP
        ori     BEEP,$BF         ; bit 6 only
        out     PORTD,BEEP

        reti                    ; important!
```

CODE SNIPPET #1

This part simply declares the Timer Overflow vector address. **NB!** It is **imperative** that you declare the interrupt vectors in the **same order as they appear in the .inc file**. If not, you will get segment overlap errors. That is; the **.org** directives must be placed according to the memory locations they represent, in ascending order. So an easy rule to use is to just enter them in the same order as in the .inc file!

CODE SNIPPET #2

First, we set the TIME register to a higher value (0x80 = 128 decimal) for starters, and load it into the Timer/Counter register. It's just a more fitting start value if you run the 8515 on a low CLK freq. Then the relevant interrupt enable bit is set in the **TIMSK** register, and the prescaling bits in the Timer Control register are set to **Prescale=8**.

This way, if the 8515 runs @ **1.23MHz**; the speaker will buzz with a base frequency equal to 1.23E6/2*8*127 = **<u>605.3 Hz</u>**

CODE SNIPPET #3

The important issues in handling this interrupt is:

- Resetting the Timer/Counter - it won't do that itself!
- Flipping the beep pin
- Returning to the program

Resetting Timer/Counter is obviously done by loading the value of TIME (r16) into TCNT0, and returning from the interrupt routine is done by issuing a **reti** instruction. Flipping the beep pin (PD6) is a little curious, however: This is done by inverting every bit in BEEP (r18) with the **com** instruction, and then OR'ing it with this value 0xbf = 10111111 b (note the 6th position is '0'). Follow the sequence below:

BEEP	00000000
after com:	11111111
'OR' with:	10111111
Result:	11111111
after com:	00000000
'OR' with:	10111111
Result:	10111111
etc...	...

As you may see; whichever value is in BEEP, the 6th bit of it will flip every time... So, the pin will toggle up and down, and the speaker beeps this little song: *"...10101010101010..."*. Haha.

Now, these were the very basic basics of interrupts and I/O. Feel free to experiment with what you have learnt in this article; use other prescaler settings, try other flanks of external interrupt triggering, write programs that use switches to make flow control decisions, whatever...

Good luck!