Introduction to embedded programming in C

Programming is a powerful skill. Not just useful, not just good for your career, not just a moneymaker, but powerful.

Embedded programming

In this section we are going to learn to program an Atmel family microcontroller (attiny44 in the examples below) using AVR-GCC toolchain in C language. Programming is a huge topic. It will take long time to master.

For learning to code you **have to** code. There is a difference between **knowing** the path and **walking** the path. Walk-the-path.

- Basics about writing code
  - Important: Always add Author, date, description and license to the header of your code
  - Always comment your code
  - Init and loop sections
  - Digital output
  - Analog output (PWM)
  - Digital input
  - Analog input (ADC) 8 bit/10bit
  - Pull up / down resistors (10k)
  - Multitasking? Polling
  - Debounce buttons
- Embedded Programming. 3 levels depending on proficiency:
  - Beginners: Arduino IDE
  - Intermediates: C
  - Advanced: Assembly
- Why C? https://www.youtube.com/watch?v=ERY7d7W-6nA&feature=youtu.be

- Why Assembly? https://en.wikipedia.org/wiki/Apollo_Guidance_Computer

I know Arduino scripting language. Do I still need to learn C?

Please watch this video https://www.youtube.com/watch?v=ERY7d7W-6nA

The datasheet

The first thing you should know about embedded programming is that **you cannot program a microcontroller without reading the datasheet** of that specific microcontroller.

> Download the Attiny 44 Datasheet and browse through it.

Setting up AVR-GCC Toolchain on Linux

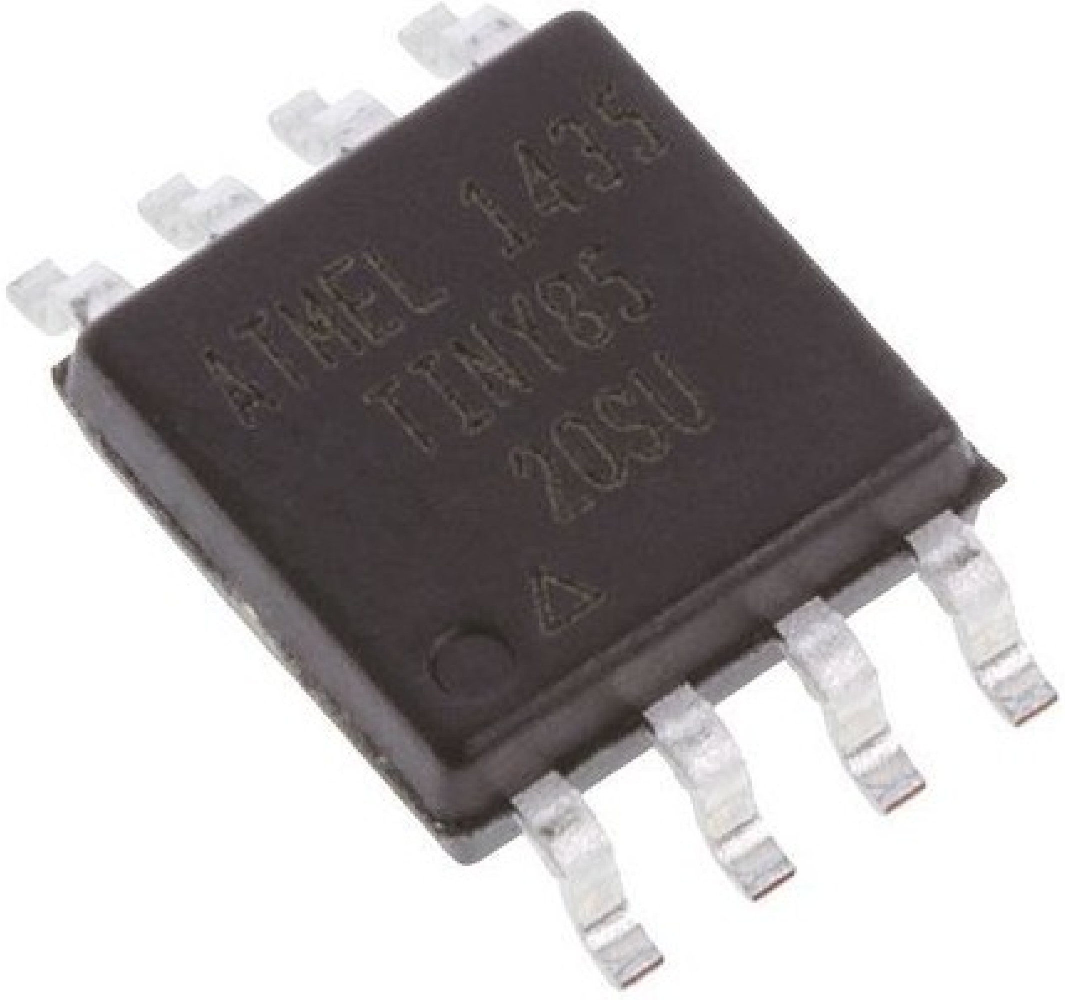
For writing code you just need a text editor, like atom. AVR-GCC is a tool-chain that will help you with the software development process, but doesn't do anything about burning the final executable (the hex file) to the microcontroller. For that we need to install ** AVR D**ownloader ** U**ploa**DE**r (avrdude).

In Ubuntu

```
sudo apt install avrdude gcc-avr binutils-avr avr-libc
```

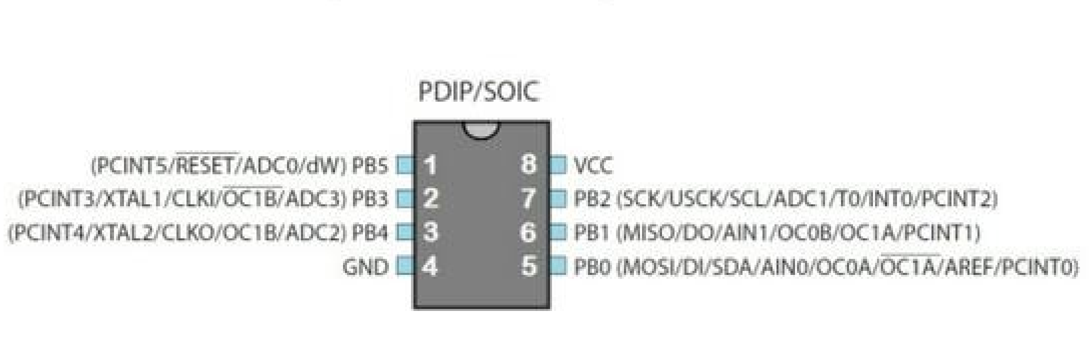
The basic idea behind a microcontroller

A microcontroller is an electronic device with a CPU, memory and some other hardware that interfaces with the external world through a number of pins.



Some pins have a fixed function and cannot be changed, like VCC or GND. The rest of the pins can be configured as inputs (for receiving data from sensors) or outputs (to move motors or turn on lights).

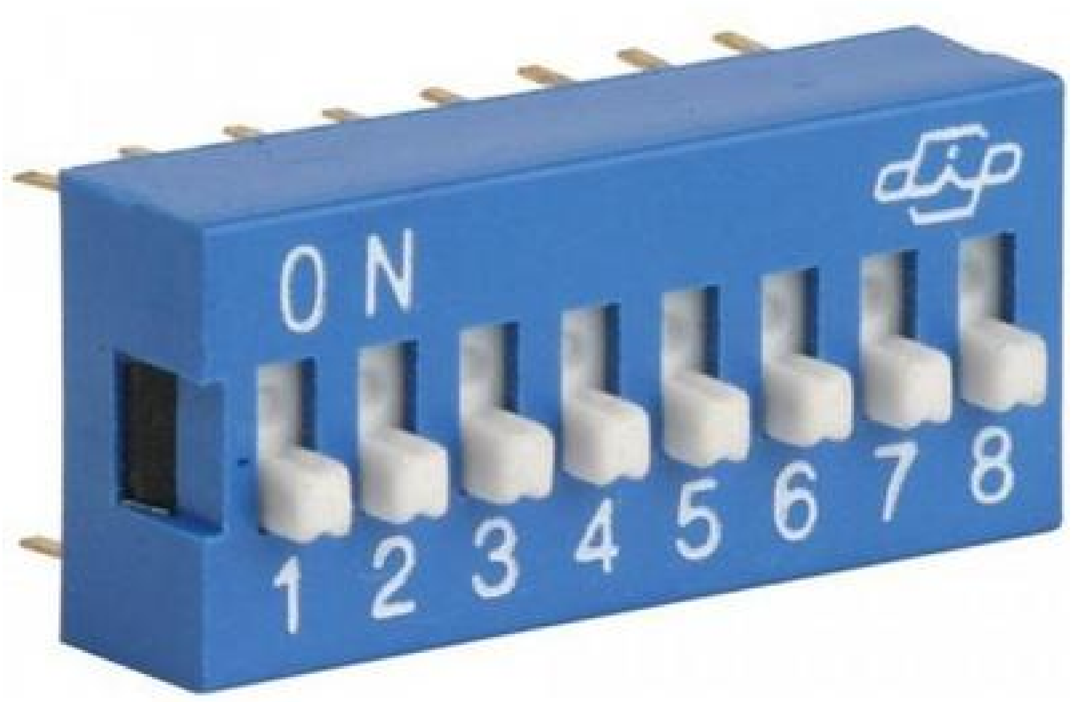
ATtiny85 Pin Configuration


PDIP/SOIC
(PCINT5/RESET/ADC0/dW) PB5 1
(PCINT3/XTAL1/CLKI/OC1B/ADC3) PB3 2
(PCINT4/XTAL2/CLKO/OC1B/ADC2) PB4 3
GND 4
8 VCC
7 PB2 (SCK/USCK/SCL/ADC1/T0/INT0/PCINT2)
6 PB1 (MISO/DO/AIN1/OC0B/OC1A/PCINT1)
5 PB0 (MOSI/DI/SDA/AIN0/OC0A/OC1A/AREF/PCINT0)

The general workflow is that you hook your inputs and outputs to the microcontroller pins and then later in your code you program the logic that enables reading from the sensors and actions in the actuators. This way, someone with little knowledge in electronics engineering can design and program a circuit board.

Setting Inputs and Outputs. The DDR (Data Direction Registers)

To tell the microcontroller if a pin is an input or an output we have modify a register called Data Direction Register. A register is like a DIP switch with 8 independently switches that can be set to 0 (off) or 1 (on).



There is one of these for each port A and B, named DDRA and DDRB.

By default all pins are configured as input.

Exercise: Read this tutorial https://www.arduino.cc/en/Tutorial/DigitalPins

PORT Registers

PORT registers write data (0 or 1) to the pins, when they are configured as outputs. If the pins are set as inputs, PORT activates or deactivates the internal pull-up resistor.

PIN Registers

PIN registers read data from the pins when they are configured as inputs.

Hello world program

If you are creating a new program, do not start from scratch. Start by editing an existing program.

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>

int main (void)
{

 DDRB = 0b00000100; // set PB2 as output in DDRB

 while(1) {
          // set PB2 high to turn led on
          PORTB = 0b00000100;
          _delay_ms(1000);
          // set PB2 low to turn led off
          PORTB = 0b00000000;
          _delay_ms(1000);
          }
}
```

The button and LED program

```
#define F_CPU 1000000UL
#include <avr/io.h>

// This program turns LED ON when the button is pressed

int main (void)
{

  DDRB |= (1 << PB2);
  // set PB2 as output in DDRB

  DDRA &= ~(1 << PA3);
  // set PA3 as input in DDRA

  PORTA = 0B00001000;
  // SET PULL UP RESISTOR IN PA3

 while(1) {
   if (testbit(PINA,PA3))

   PORTB = 0b00000000;
   // set PB2 low to turn led off

   else

   PORTB = 0b00000100;
   // set PB2 high to turn led on
        }
}
```

Bitwise operations

So far we have been writing a full register. But it is also possible to operate in a single bit of a register. The syntax in C is rather complex. That is why the first thing we will do is defining macros.

Making your life easier: Macros

Macros are aliases that you can use to avoid typing complex syntax.

```
#define setbit(register, bit)   (register) |=  (1 << (bit))
#define clearbit(register, bit) (register) &= ~(1 << (bit))
#define testbit(register, bit)  (register) &   (1 << (bit))
```

Then you can simply use

```
setbit(PORTA, PA0)
clearbit(PORTB, PB3)
testbit (PORTA, PA2)
```

This makes your code much easier to understand. The precompiler will substitute the macros out at compile time and you will be left with the exact same code in your AVR.

Compiling the program

The microcontroller does not understand C code. C, like other programming languages was created for humans. Microcontrollers only understand hex instructions, but those are difficult to write or remember. It is necessary to compile the program to obtain a `.hex` file that will be flashed in the microcontroller.

The Makefile

http://www.ladyada.net/learn/avr/avrdude.html

The programmer

To flash the `.hex` code to the microcontroller we need a ISP programmer. We are going to fabricate our custom ISP.

Back to Summary

Introduction to interrupts

- Understanding interrupts
- Activating the interrupts
- Pin change interrupts
- Supercharged interrupt pins
- The interrupt routine ISR

Understanding interrupts

> You are in your room playing video-games. It is dinner time and **your mum gives you a call to go immediately**. You stop what you are doing, you eat your dinner and when you finish, you continue playing video-games. In this story, you are a microcontroller. The video-game is the current task. **Your mother is an interrupt**. And dinner is the interrupt subroutine.

So far you have been dealing with polling when you need to do multiple tasks in a microcontroller. We are going to introduce a **powerful piece of hardware** inside the microcontroller, the **interrupts**.

Interrupts are **events that require immediate attention by the microcontroller**. When an interrupt event occurs the microcontroller pauses the current task and attend to the interrupt by executing a subroutine. After that, the microcontroller returns to the task it had paused and continue its normal operation.

Activating the interrupts

> **Note:** This chapter is based on the attiny 44 microcontroller

We are going to follow a worked example on how to create a simple interrupt.

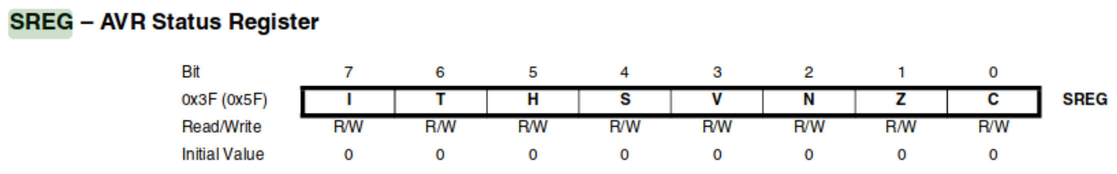
> **Learn by doing:** Create a new file using, and initialise it with all the information you usually add to the header (author, date, description, license)

For using interrupts you need to include the interrupt library in the header of your C code:

```
#include <avr/interrupt.h>
```

Also if we want to use interrupts in our application, we have to set (write a 1) in the the bit 7 of the **S**tatus **REG**ister (SREG).

> **Learn by doing:** Check in the datasheet what bit 7 is doing



SREG – AVR Status Register

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

This can be done either by setting the bit as usual in C.

Learn by doing: How would you set bit 7 to the SREG register in C?

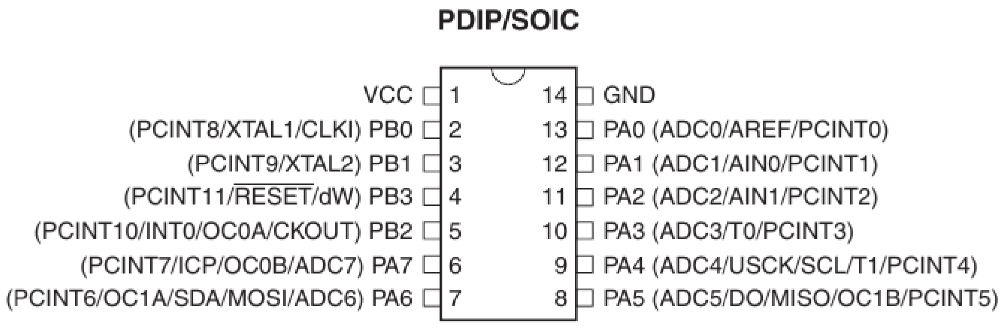
You can alternatively call the instruction SEI (set global interrupt):

```
sei(); //activates global interrupts
```

Pin change interrupts

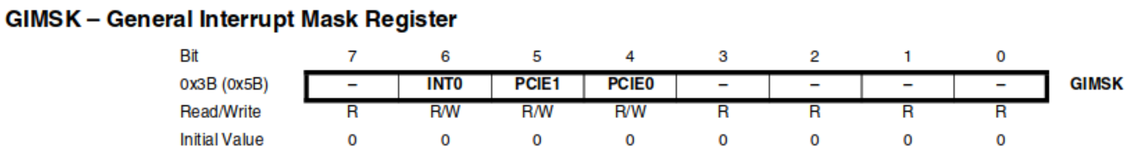
One specific kind of interrupt event is when an **input pin** changes its status, from 0 to 1 or from 1 to 0. They are called **pin change interrups**.

Inside the Attiny44 there are **two** pin change interrupts, one in PORTA and one in PORTB. They will trigger whenever it finds a falling edge or rising edge. So determining if it is a falling edge or rising edge must be done in the interrupt subroutine.



PDIP/SOIC
VCC 1
(PCINT8/XTAL1/CLKI) PB0 2
(PCINT9/XTAL2) PB1 3
(PCINT11/RESET/dW) PB3 4
(PCINT10/INT0/OC0A/CKOUT) PB2 5
(PCINT7/ICP/OC0B/ADC7) PA7 6
(PCINT6/OC1A/SDA/MOSI/ADC6) PA6 7
14 GND
13 PA0 (ADC0/AREF/PCINT0)
12 PA1 (ADC1/AIN0/PCINT1)
11 PA2 (ADC2/AIN1/PCINT2)
10 PA3 (ADC3/T0/PCINT3)
9 PA4 (ADC4/USCK/SCL/T1/PCINT4)
8 PA5 (ADC5/DO/MISO/OC1B/PCINT5)

If we want to use them, We have to activate them by setting the corresponding bits in the GIMSK register



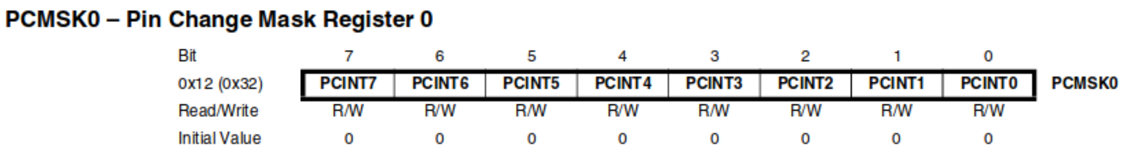
GIMSK – General Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x3B (0x5B)	–	INT0	PCIE1	PCIE0	–	–	–	–	GIMSK
Read/Write	R	R/W	R/W	R/W	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

```
GIMSK |= (1<<PCIE0); //enable pin change interrupt in PORTA (pins PA0 to PA7)
```

Learn by doing: How would you activate pin change interrupts in PORTB?

And then we have to tell the MCU which pin or pins we will be listening to, using the PCMSK0 register.



PCMSK0 – Pin Change Mask Register 0

Bit	7	6	5	4	3	2	1	0	
0x12 (0x32)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

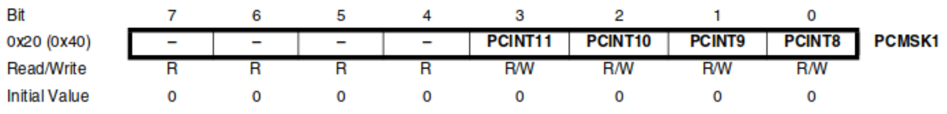
```
PCMSK0 |= (1<<PCINT7); //enable PC interrupt in PA7
```

Same for PORTB, we first activate them in the GIMSK register

```
GIMSK |= (1<<PCIE1); //enable pin change interrupt in PORTB (pins PB0 to PB2)
```

And then we tell the MCU which pins we want to listen using the PCMSK1 register.

PCMSK1 – Pin Change Mask Register 1



Bit	7	6	5	4	3	2	1	0	
0x20 (0x40)	–	–	–	–	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Supercharged interrupt pins

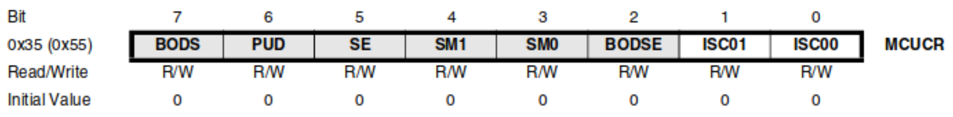
One of the pins of PORTA is a bit special. It is pin **PA0** which can be triggered only at falling, only at rising or only at low level. If we want to use this feature we have to activate the INT0 bit in the GIMSK register

```
GIMSK |= (1<<INT0); //this would enable INT0 in PA0
```

And set or clear the bits 0 and 1 of the MCUCR (MCU control register)

MCUCR – MCU Control Register

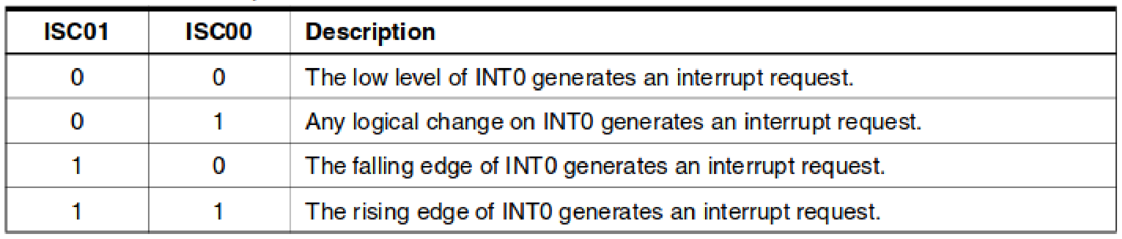
The External Interrupt Control Register A contains control bits for interrupt sense control.



Bit	7	6	5	4	3	2	1	0	
0x35 (0x55)	BODS	PUD	SE	SM1	SM0	BODSE	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Using this table:

Table 9-2. Interrupt 0 Sense Control

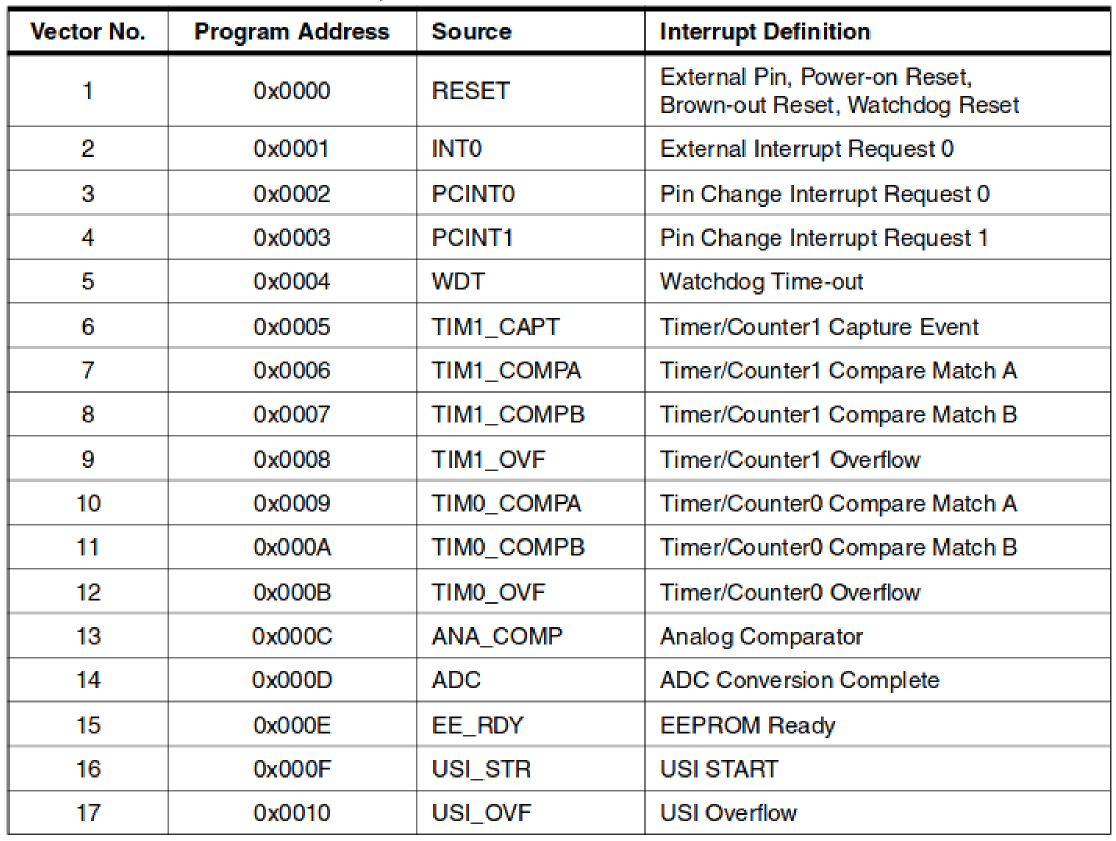


ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

The interrupt routine ISR

Ok, the interrupt is set and ready. Now what? So far we have activated the desired interrupts using the registers. Depending of which interrupt is triggered (pin change in port A or B or INT0) it will go to a specific subroutine called vector. These are the available vectors in the attiny44:

Table 9-1. Reset and Interrupt Vectors



Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	PCINT0	Pin Change Interrupt Request 0
4	0x0003	PCINT1	Pin Change Interrupt Request 1
5	0x0004	WDT	Watchdog Time-out
6	0x0005	TIM1_CAPT	Timer/Counter1 Capture Event
7	0x0006	TIM1_COMPA	Timer/Counter1 Compare Match A
8	0x0007	TIM1_COMPB	Timer/Counter1 Compare Match B
9	0x0008	TIM1_OVF	Timer/Counter1 Overflow
10	0x0009	TIM0_COMPA	Timer/Counter0 Compare Match A
11	0x000A	TIM0_COMPB	Timer/Counter0 Compare Match B
12	0x000B	TIM0_OVF	Timer/Counter0 Overflow
13	0x000C	ANA_COMP	Analog Comparator
14	0x000D	ADC	ADC Conversion Complete
15	0x000E	EE_RDY	EEPROM Ready
16	0x000F	USI_STR	USI START
17	0x0010	USI_OVF	USI Overflow

Say we have a button in PA7, connected to ground and an LED in pin PB2 also connected to ground. This would be a subroutine to control the LED with the button using interrupts:

```
ISR(PCINT0_vect)
{
 if ( PINA & (1<<PINA7) ) // test PINA7 if true then
  PORTB &= ~(1<<PB2); //turn off LED
 else
  PORTB |= (1<<PB2); //Turn on LED
}
```

Learn by doing: Create a new file using, and initialize it with all the information you usually add to the header (author, date, description, license). Write a program that blinks and LED only when you *release* the button.

TODO interrupt priority

Back to Summary

Timer Interrupts

- Understanding timer interrupts
- Activating timer interrupts
- Setting the mode of operation
- The interrupt routine ISR
- Worked Example

Understanding timer interrupts

In the last chapter we learned about the timer counter and the required registers to make it work.

We also had learned before about interrupts. An interrupt is an event that occurs when a particular trigger (either from an internal or external source) happens. The interrupt allows the program to stop what it was doing, perform a short task, and then return to where it left off in the main program.

Today we will learn about interrupts associated with timers. Timers can generate interrupts when they overflow or reach certain value. That will be very helpful to make our program to execute a task at regular intervals of time while executing at the same time other tasks.

Activating timer interrupts

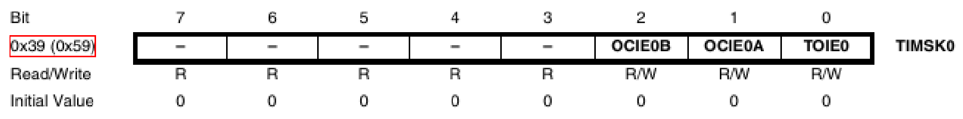
To enable timer interrupts, first we have to include the interrupt library and enable global interrupts

```
#include <avr/interrupt.h>
sei();
```

Setting the mode of operation

Next thing we have to do is set the **mode of operation**. When will the interrupt be triggered? There is a dedicated register where we can specify that. Remember that there are 2 timers, a 8-bit and a 16-bit. For `TIMER0` (8-bit) we have:

TIMSK0 – Timer/Counter 0 Interrupt Mask Register



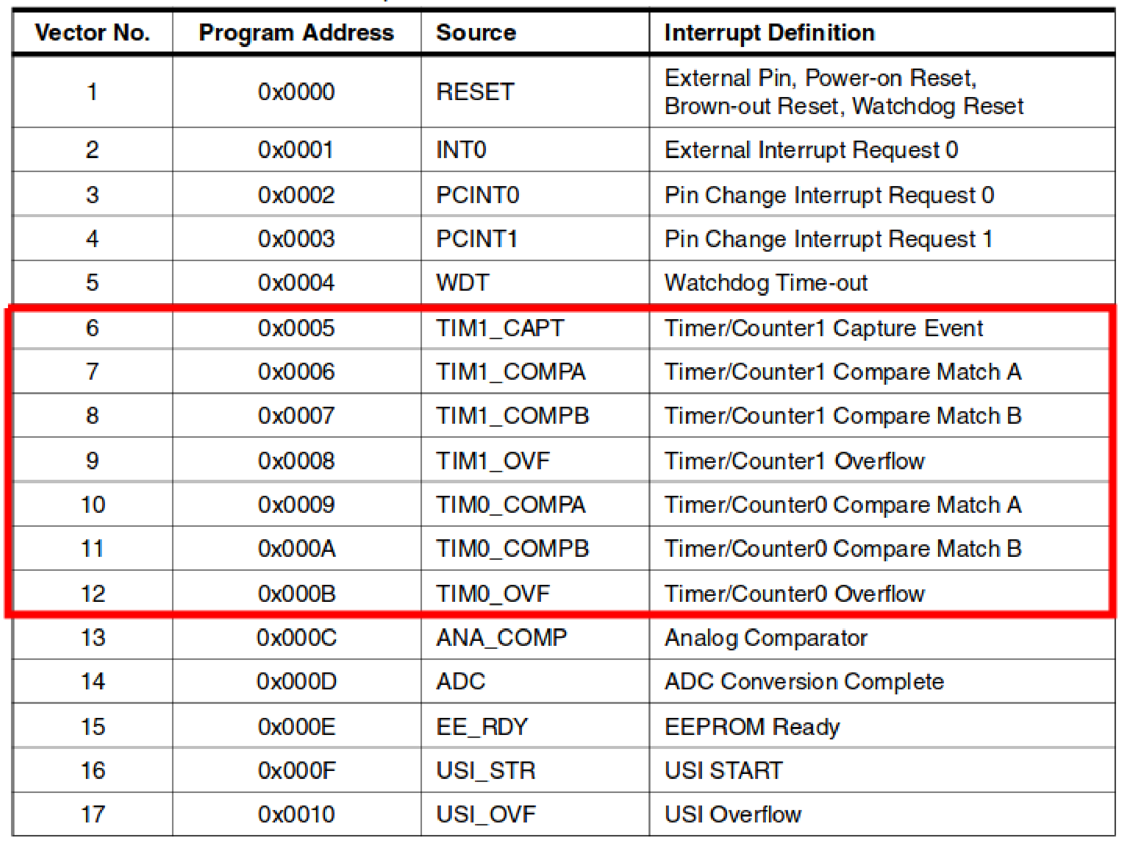
Bit	7	6	5	4	3	2	1	0	
0x39 (0x59)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Setting `TOIE0` bit will enable the interrupt in normal mode of operation. That means, the interrupt will be triggered when the counter overflows.
- Setting `OCIE0A` bit will enable the interrupt in CTC mode of operation when `TCNT0=OCR0A` . That is, when the counter `TCNT0` reaches a certain value (`OCR0A`) that we have decided beforehand.
- Similarly, setting `OCIE0B` bit will enable the interrupt in CTC mode of operation when `TCNT0=OCR0B` .

The interrupt routine ISR

As we saw before, the interrupt routine needs to have a specific name called vector. And these are the attiny44 vectors associated with timer interrupts:

Table 9-1. Reset and Interrupt Vectors



Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	PCINT0	Pin Change Interrupt Request 0
4	0x0003	PCINT1	Pin Change Interrupt Request 1
5	0x0004	WDT	Watchdog Time-out
6	0x0005	TIM1_CAPT	Timer/Counter1 Capture Event
7	0x0006	TIM1_COMPA	Timer/Counter1 Compare Match A
8	0x0007	TIM1_COMPB	Timer/Counter1 Compare Match B
9	0x0008	TIM1_OVF	Timer/Counter1 Overflow
10	0x0009	TIM0_COMPA	Timer/Counter0 Compare Match A
11	0x000A	TIM0_COMPB	Timer/Counter0 Compare Match B
12	0x000B	TIM0_OVF	Timer/Counter0 Overflow
13	0x000C	ANA_COMP	Analog Comparator
14	0x000D	ADC	ADC Conversion Complete
15	0x000E	EE_RDY	EEPROM Ready
16	0x000F	USI_STR	USI START
17	0x0010	USI_OVF	USI Overflow

Worked Example

In this example we are going to consider all the above to set up a 1 second timer on an attiny44 microcontroller with an external 20MHz clock source. So first thing is activating global interrupts:

```
sei();
```

At 20Mhz, a clock cycle is 50ns. We will count time using `TIMER1` (the 16-bit timer). Counting to 65535 at this speed with take exactly 3.277ms. That is definitely too fast for our purpose.

So we will have to trigger the timer with a **prescaler** value (by setting the Clock Select bits `CS12:10`) in Timer/Counter1 Control Register B (`TCCR1B`). This allows us to slow down the timer by a factor of 1, 8, 64, 256 or 1024 compared with the clock source. However, it is unlikely that counting to 65535 at any of these speeds with take exactly one second as we see in the table below:

Pre-scaler value	Time
1	3.277 ms

Pre-scaler value	Time
8	26.216 ms
64	209.728 ms
256	838.912 ms
1024	3.355392 s

We see that we will have to **set the pre-scaler to 1024** in order to be able to count at least a second.

But what number shall we count up to? As you see in the table above, counting to 65535 at this speed would take 3.355392 seconds. Therefore 1 second will take 65535/3.355392=19531.25 increments. To set this as the maximum the timer should count to, we set Output Compare Register 1 A (`OCR1A`) to 19531.

```
OCR1A   = 19531; // set the CTC compare value
```

Next thing is setting the timer to CTC (Clear Timer on Compare) mode. To use this mode, the Waveform Generation Mode bits (`WGM13:10`) in `TCCR1A` and `TCCR1B` must be set accordingly.

```
TCCR1B |= (1 << WGM12); // configure timer1 for CTC mode
```

> **Think about it:** As you can see, 1 second is 19531.25 clock cycles. But actually, we are counting 19531 clock cycles. Can you derive how much this count will be off after a year?

Let's set the **interrupt**. To ensure that a signal is received when the timer reaches 1 second, the Timer/Counter1 Output Compare A Match interrupt (`OCIE1A`) bit must be set in the Timer/Counter1 Interrupt Mask Register (`TIMSK1`).

```
TIMSK1 |= (1 << OCIE1A); // enable the CTC interrupt
```

Finally we have to set the ISR routine. In this case setting the `OCIE1A` bit corresponds to the `TIM1_COMPA_vect` Interrupt Service Routine.

```
ISR(TIM1_COMPA_vect) {
  // here the routine that
  // toggles the LED
}
```

> **Learn by doing:** Write the routine that toggles the LED

That was everything required to prepare the timer and the interrupt, so the only thing remaining is triggering it. Remember that the timer will start as soon as we set the pre-scaler:

```
TCCR1B |= ((1 << CS10) | (1 << CS12)); // start the timer   at 20MHz/1024
```

> **Learn by doing:** Write another routine that toggles the LED every minute.

Back to Summary

Timer/counters in the attiny44

- The problem of the blinking LED
- The timers/counters
- The Prescaler
  - Setting the prescaler for `TIMER0`
  - Setting the prescaler for `TIMER1`
- Operation modes
  - Normal mode
  - CTC mode
  - Selecting the mode of Operation

The **timer/counter** is a **piece of hardware** inside the microcontroller that can be used to **count and control time**. Actually, it does not count time, because it doesn't know anything about time. It should be actually named counter because **it just counts clock cycles**. What makes timers so interesting is the fact that they are **totally independent of the CPU**. The timers run parallel to the CPU without its intervention.

> **Note:** All the information listed here is a extract from the attiny44 datasheet. It just organized differently in order to be easier to understand. Other microcontrollers will have different register names.

The problem of the blinking LED

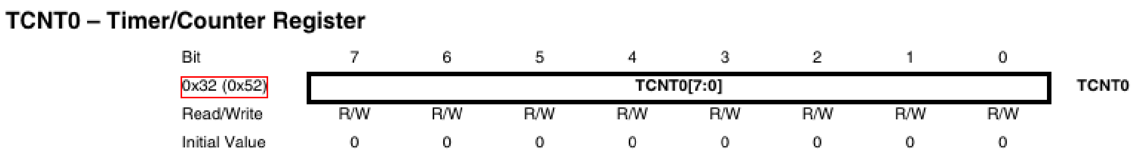
Let's see the classical blinking LED code, where we have an LED in PA3, connected to ground.

```c
#include <avr/io.h>
#include <util/delay.h>

int main (void) {
    DDRA |= (1 << PA3); // set LED pin as output

while(1) {
      PORTA |= (1 << PA3);  // turn on the LED
      _delay_ms(1000);      // wait a second
      PORTA &= ~(1 << PA3); // turn off the LED
      _delay_ms(1000);      // wait a second
         }
  }
```

The problem when counting time this way, is that **the microcontroller cannot do anything else while we are waiting with the delay**. As your coding abilities increase you will notice a limitation in what you can achieve due to this problem. In fact, **you should never use delays**.

The timers/counters

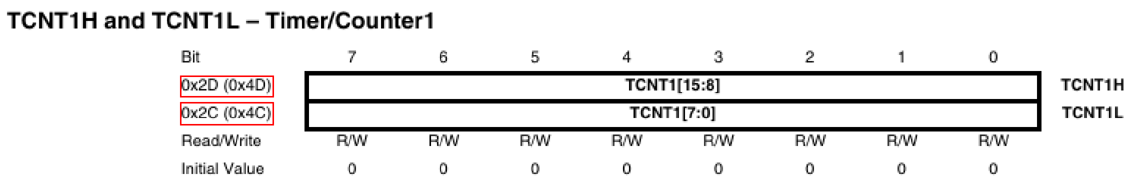
The ATtiny44 has two independent internal timers. One 8-bit (`TIMER0`) and another 16-bit (`TIMER1`), each with 2 PWM channels (more on this later). **Timers are actually registers** that increase automatically from 0 to 255. For the 8-bit, the count is stored in the `TCNT0` register:



TCNT0 – Timer/Counter Register

Bit	7	6	5	4	3	2	1	0	
0x32 (0x52)	TCNT0[7:0]								TCNT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

For the 16-bit register, the count from 0 to 65535 is stored in the `TCNT1` register, which in reality is split as two 8-bit registers, high `TCNT1H` and low `TCNT1L` :



TCNT1H and TCNT1L – Timer/Counter1

Bit	7	6	5	4	3	2	1	0	
0x2D (0x4D)	TCNT1[15:8]								TCNT1H
0x2C (0x4C)	TCNT1[7:0]								TCNT1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

When they reach the maximum value, they overflow back to zero and start counting again. So far, so good.

The Prescaler

By default, the timer register increment at the speed of the microcontroller (if a microcontroller clock speed of 1 MHz is used, then 1,000,000 times per second). So at every tick of the clock you count 1/1,000,000 s.

> **Question:** Do you foresee the problem?

That is an impressive resolution. But we may want to time large time intervals as well. Sometimes we want to count a few seconds. So the problem is that, at 1Mhz speed, the register `TCNT0` is going to overflow too fast, exactly at 0.000256s.

Somehow we need to slow down the speed at which the counter is updating. One way if doing that it is hooking a slower oscillator or crystal. Actually most watches use a 32768Hz oscillator.

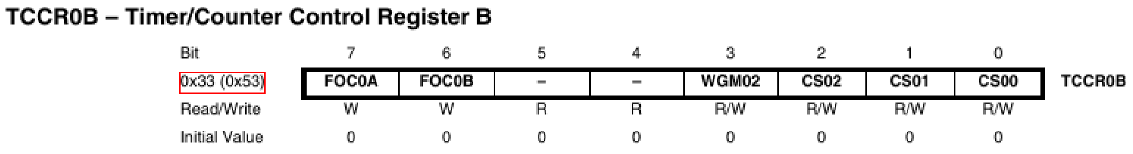
> **Question:** 32768Hz, Why this weird number? Clue: Power of two.

Another way to slow down the count is only increasing the count when a certain amount of clock cycles has occurred. This is called **prescaling**. For the AtTiny, the prescaler can be set to 8, 64, 256 or 1024 compared to the system clock.

If we set the prescaler to 1024. Then the register `TCNT0` will increase the count every 1024 clock cycles, thus every 1024/1000000 = 0.001024 s and will overflow at 0.262144 s.

Setting the prescaler for `TIMER0`

The prescaler for `TIMER0` is set using the control register `TCCR0B` :



TCCR0B – Timer/Counter Control Register B

Bit	7	6	5	4	3	2	1	0	
0x33 (0x53)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

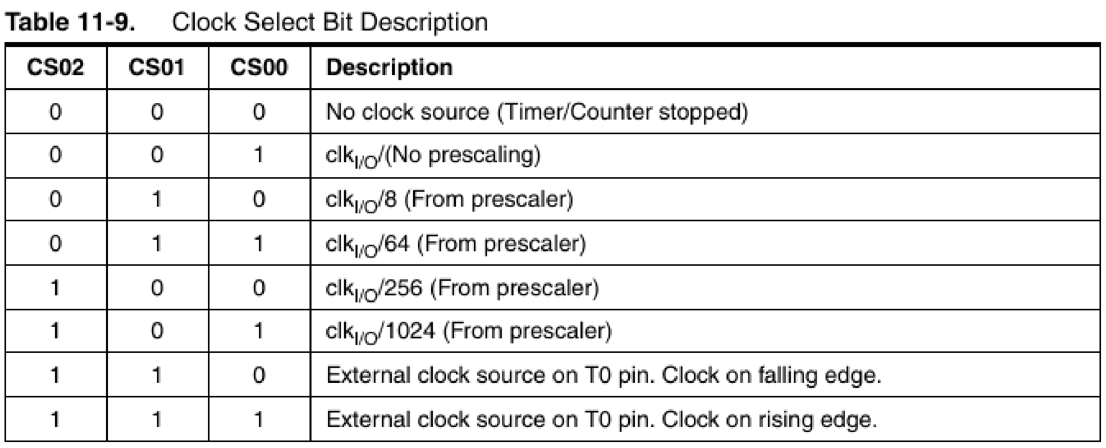
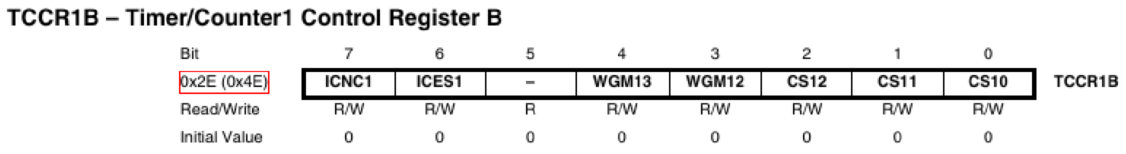


Table 11-9. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Setting the prescaler for `TIMER1`

The prescaler for `TIMER1` is set using the control register `TCCR1B` :



TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
0x2E (0x4E)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

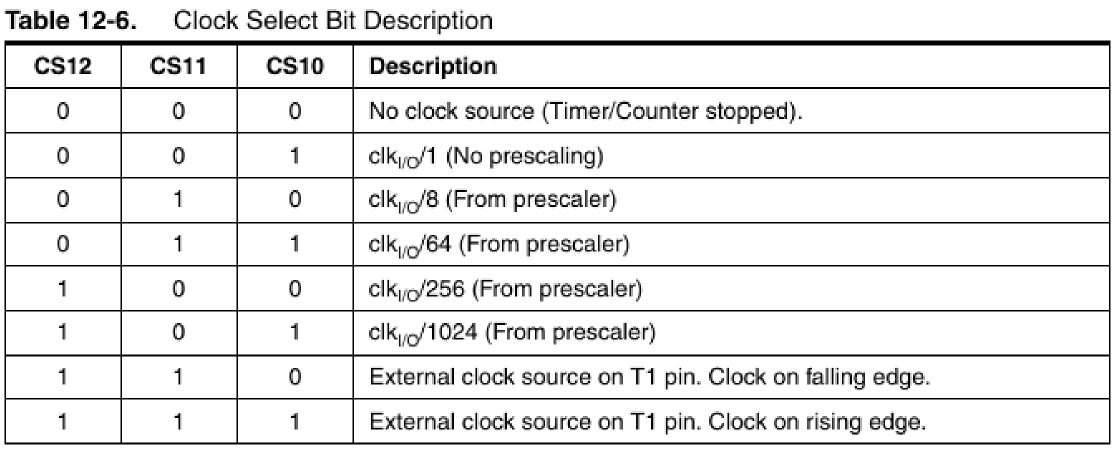


Table 12-6. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

WARNING! Once you set the prescaler, the counter will start!

Let's see an example of using `TIMER0` to blink an LED every 124 counts at 1024 prescaler.

```
#include <avr/io.h>

int main(void)
```

```
{
    // connect led to pin PA3
    DDRA |= (1 << PA3);

    // initialize timer
    // set up timer with prescaler = 1024
    TCCR0B |= (1 << CS02)|(1 << CS00);

    // initialize counter
    TCNT0 = 0;

    // loop forever
    while(1)
    {
        // check if the timer count reaches 124
        if (TCNT0 >= 124)
        {
            PORTA ^= (1 << PA3);    // toggles the led
            TCNT0 = 0;             // reset counter
        }
    }
}
```

Operation modes

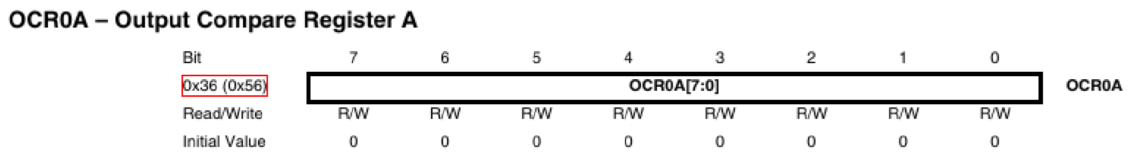
Timers have different modes of operation. We will focus in two for now: Normal mode and CTC mode. Let's explain what these modes do, one at a time.

Normal mode

By default the timers will run in this mode. They will count from 0 to their maximum value 255/65535, and then overflow. The problem with this mode is that it is uncommon that we want to count up to that specific number, that is why the have the CTC mode.

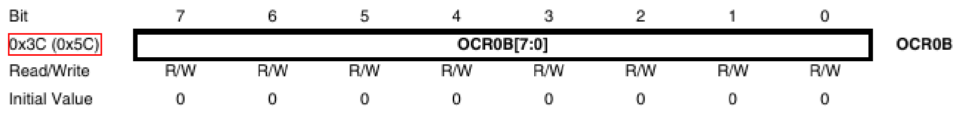
CTC mode

This is **clear timer on compare match** mode. This is the most common case. We will load a specific value in a register and then we will count up to that number. In each timer you can set up to 2 independent values that will trigger 2 independent interrupts. These for `TIMER0` :


OCR0A – Output Compare Register A

Bit	7	6	5	4	3	2	1	0	
0x36 (0x56)	OCR0A[7:0]								OCR0A
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

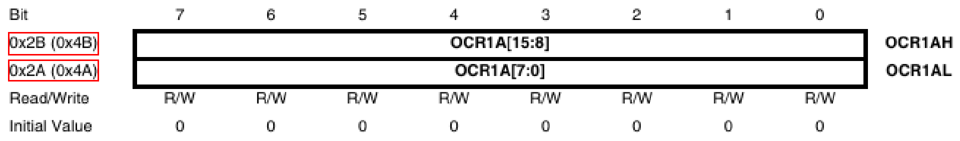
OCR0B – Output Compare Register B



Bit	7	6	5	4	3	2	1	0	
0x3C (0x5C)	OCR0B[7:0]								OCR0B
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

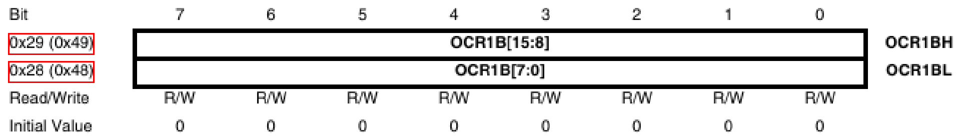
And these for `TIMER1` :

OCR1AH and OCR1AL – Output Compare Register 1 A



Bit	7	6	5	4	3	2	1	0	
0x2B (0x4B)	OCR1A[15:8]								OCR1AH
0x2A (0x4A)	OCR1A[7:0]								OCR1AL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

OCR1BH and OCR1BL – Output Compare Register 1 B

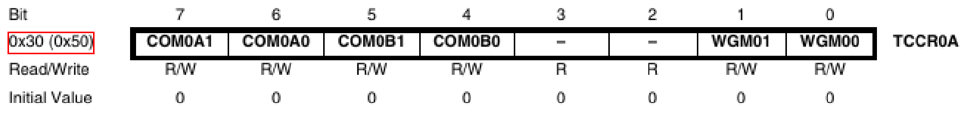


Bit	7	6	5	4	3	2	1	0	
0x29 (0x49)	OCR1B[15:8]								OCR1BH
0x28 (0x48)	OCR1B[7:0]								OCR1BL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Selecting the mode of Operation

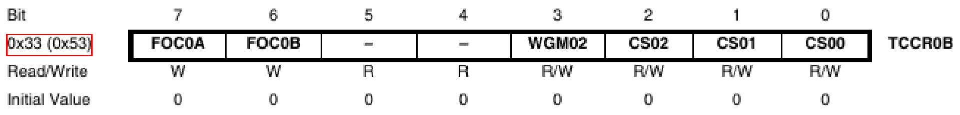
Operation modes are set with the control registers `TCCR0A` and `TCCR0B` for the 8-bit `TIMER0` , according to the following table.

TCCR0A – Timer/Counter Control Register A



Bit	7	6	5	4	3	2	1	0	
0x30 (0x50)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR0B – Timer/Counter Control Register B



Bit	7	6	5	4	3	2	1	0	
0x33 (0x53)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 11-8. Waveform Generation Mode Bit Description

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on[1]
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Note: 1. MAX = 0xFF
BOTTOM = 0x00

And `TCCR1A` and `TCCR1B` for the 16-bit `TIMER1` .

TCCR1A – Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0	
0x2F (0x4F)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
0x2E (0x4E)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

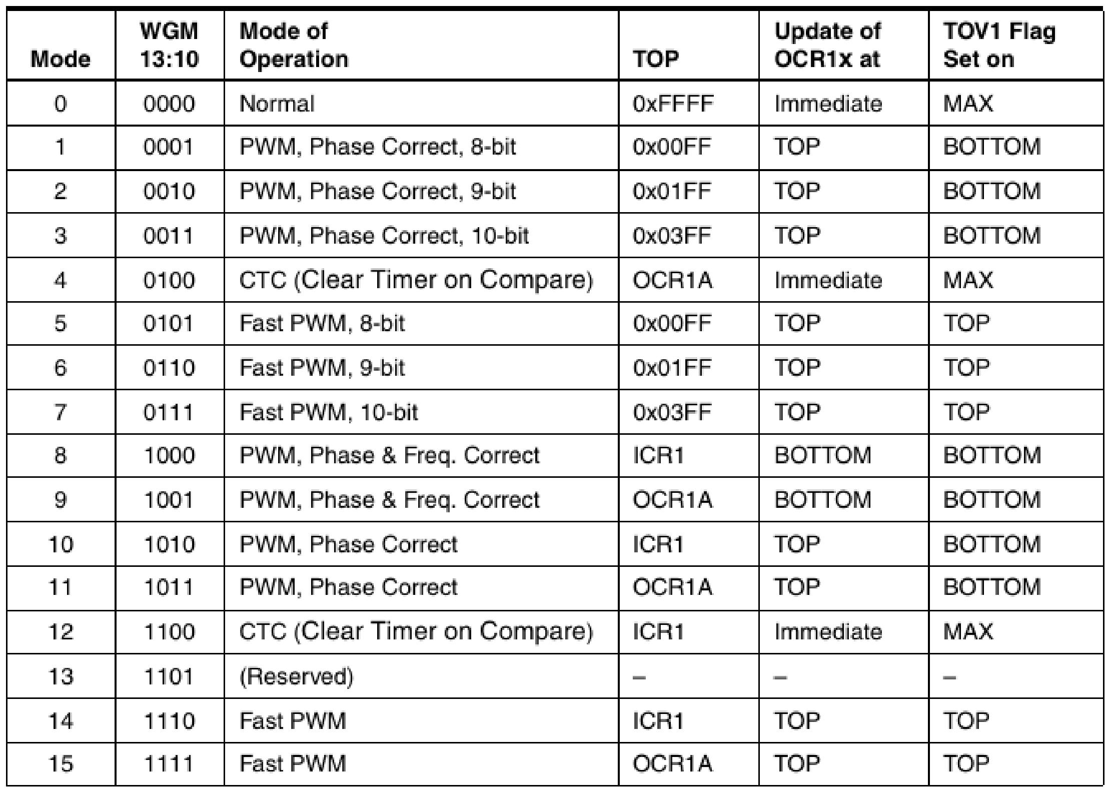


Table 12-5. Waveform Generation Modes

Mode	WGM 13:10	Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0000	Normal	0xFFFF	Immediate	MAX
1	0001	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0010	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0011	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0100	CTC (Clear Timer on Compare)	OCR1A	Immediate	MAX
5	0101	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0110	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0111	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1000	PWM, Phase & Freq. Correct	ICR1	BOTTOM	BOTTOM
9	1001	PWM, Phase & Freq. Correct	OCR1A	BOTTOM	BOTTOM
10	1010	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1011	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1100	CTC (Clear Timer on Compare)	ICR1	Immediate	MAX
13	1101	(Reserved)	–	–	–
14	1110	Fast PWM	ICR1	TOP	TOP
15	1111	Fast PWM	OCR1A	TOP	TOP

Learn by doing: Write a piece of code that counts exactly 1 second. Assume you are using a 8 Mhz resonator.

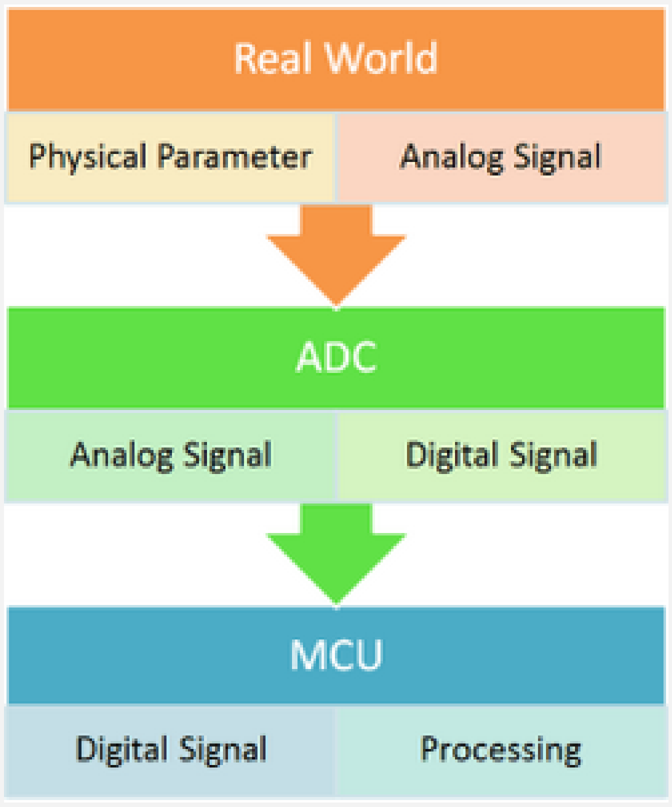
Back to Summary

Analog to Digital Converter (ADC)

- Analog signals
- ADC Prescaler
- Conversion time
- Selecting the channel(s)
- Modes of Operation & Starting the Conversion
- Beware the smoke

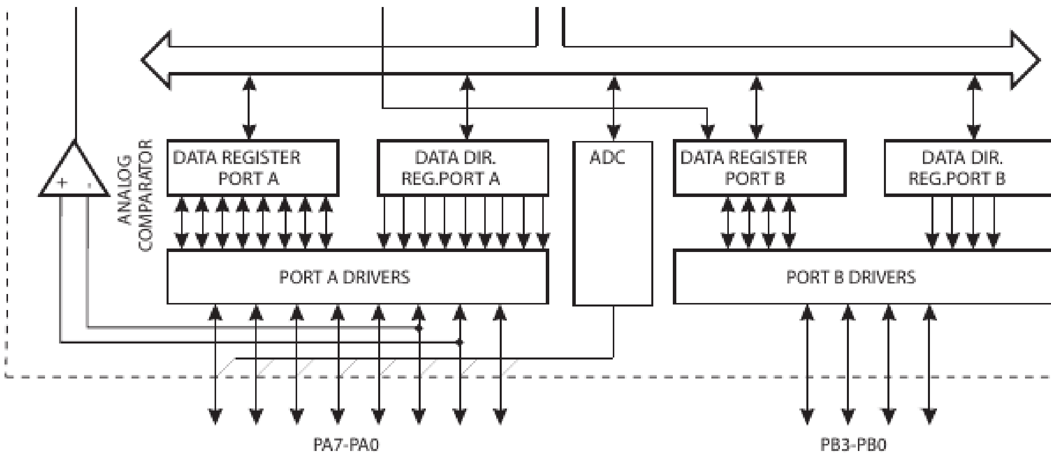
Analog signals

The microcontroller is a digital device so it can only read and write digital signals. In order to read real world analog signals, it has a piece of hardware called **analog to digital converter or ADC** which converts an continuous analog value into a discrete digital value.



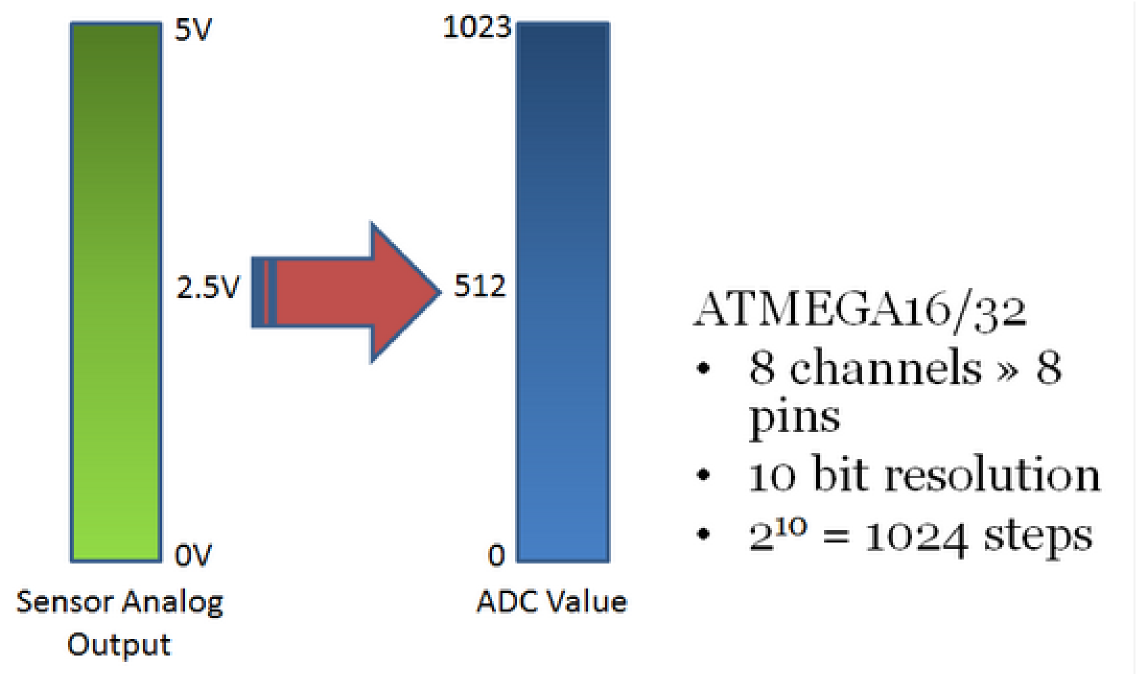
Signal Acquisition Process

In the attiny44, the ADC is hooked to PORTA (pins PA0 to PA7) only. You cannot read



analog signals in PORT B.

It is a 10-bit ADC. That means it has a resolution of 2E10=1024. It has 8 channels through a multiplexer. That means you can convert up to 8 analog signals (not at the same time, of course).



8 channel 10 bit ADC

- It has 12 differential ADC channel pairs with 1x/20x gain

ADC Prescaler

The ADC of the AVR converts analog signal into digital signal at some regular interval. This interval is determined by the clock frequency. In general, the ADC input requires a frequency range of 50kHz to 200kHz. Out of this range of frequencies, which one do we choose? There is a trade-off between frequency and accuracy. The greater the frequency, the lesser the accuracy. So, if your application doesn't require 10 bits of resolution, you could go for higher frequencies than 200KHz. In any case it is not recommended to go over 1MHz.

Since the CPU clock frequency is much higher (in the order of MHz), to achieve it, a frequency division must be applied. The prescaler acts as this division factor. It produces desired frequency from the external higher frequency. There are some predefined division factors: 2, 4, 8, 16, 32, 64, and 128. For example, a prescaler of 64 implies F_ADC = F_CPU/64. For F_CPU = 16MHz, F_ADC = 16M/64 = 250kHz.

You configure the division factor of the pre-scaler using the ADPS bits.

Conversion time

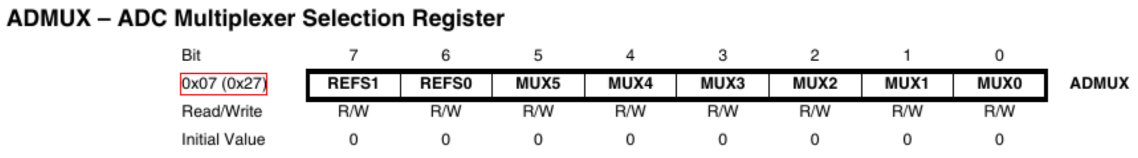
The analog to digital conversion is not instantaneous, it takes some time. This time depends on the clock signal used by the ADC. The conversion time is proportional to the frequency of the ADC clock signal.

If you can live with 8-bit resolution, you can reduce the conversion time by increasing the ADC clock frequency.

To know the time that a conversion takes, just need to divide the number of ADC clock cycles needed for conversion by the frequency of the ADC clock. Normally, a conversion takes 13 ADC clock cycles. The first conversion after the ADC is switched on (by setting the ADEN bit) takes 25 ADC clock cycles. This first conversion is called an "Extended Conversion". For instance, if you are using a 200kHz ADC clock signal, a normal conversion will take 65 microsenconds (13/200e3=65e-6), and an extended conversion will take 125 microseconds (25/200e3=125e-6).

Selecting the channel(s)

You can select the input channel using the ADMUX register bits 4:0

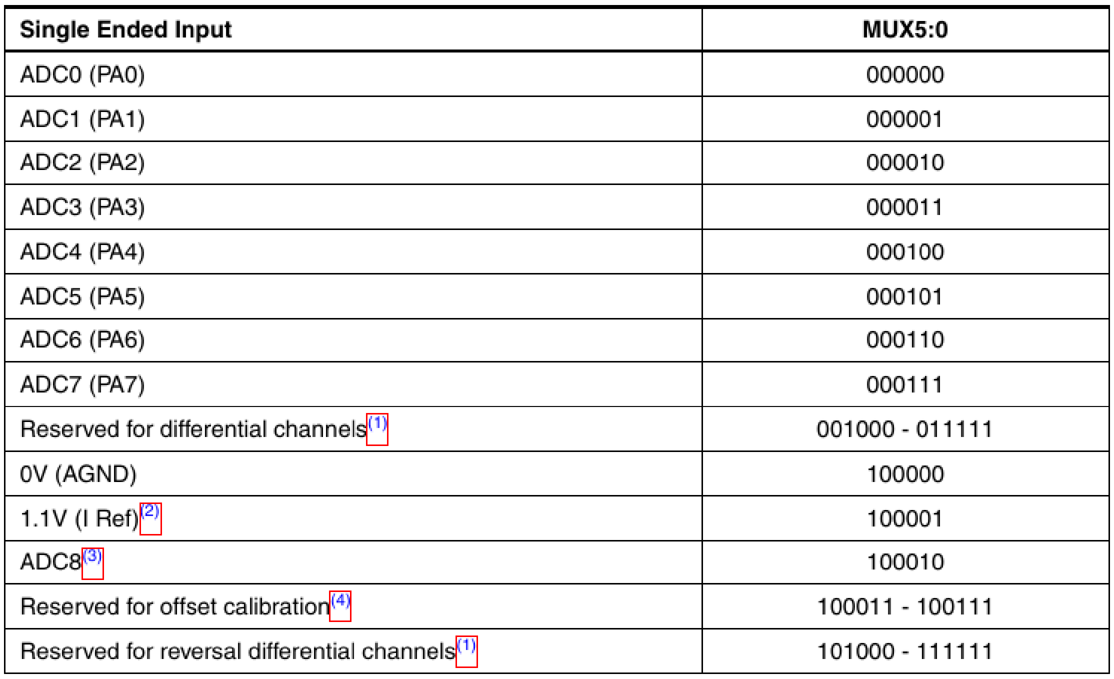


ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
0x07 (0x27)	REFS1	REFS0	MUX5	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Using this table:

Table 16-4. Single-Ended Input channel Selections.



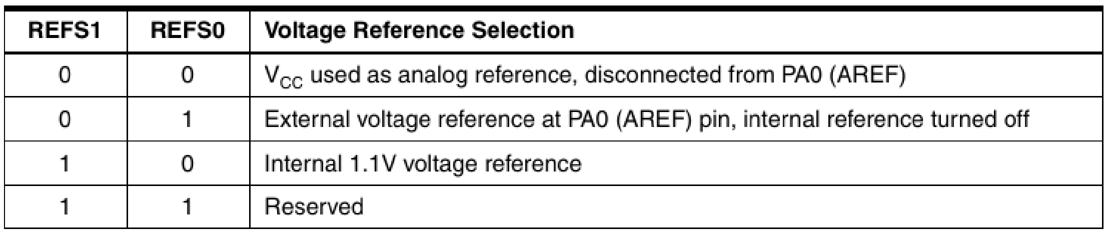
Single Ended Input	MUX5:0
ADC0 (PA0)	000000
ADC1 (PA1)	000001
ADC2 (PA2)	000010
ADC3 (PA3)	000011
ADC4 (PA4)	000100
ADC5 (PA5)	000101
ADC6 (PA6)	000110
ADC7 (PA7)	000111
Reserved for differential channels(1)	001000 - 011111
0V (AGND)	100000
1.1V (I Ref)(2)	100001
ADC8(3)	100010
Reserved for offset calibration(4)	100011 - 100111
Reserved for reversal differential channels(1)	101000 - 111111

So if we want to activate ADC6 in PA6 we do:

```
ADMUX |= (1<<MUX2 | 1<<MUX1)
```

The reference voltage is selected configuring the bits REFS1 and REFS0 according to this table. By default is set to use VCC:

Table 16-3. Voltage Reference Selections for ADC

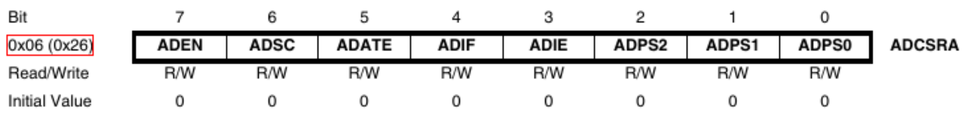


REFS1	REFS0	Voltage Reference Selection
0	0	V_{CC} used as analog reference, disconnected from PA0 (AREF)
0	1	External voltage reference at PA0 (AREF) pin, internal reference turned off
1	0	Internal 1.1V voltage reference
1	1	Reserved

Modes of Operation & Starting the Conversion

The ADC has **two** fundamental operation **modes**: **Single Conversion** and **Free Running**. In Single Conversion mode, you have to initiate each conversion. When it is done, the result is placed in the ADC Data register pair and no new conversion is started. In Free Runing mode, you start the conversion only once, and then, the ADC automatically will start the following conversion as soon as the previous one is finished.

ADCSRA – ADC Control and Status Register A

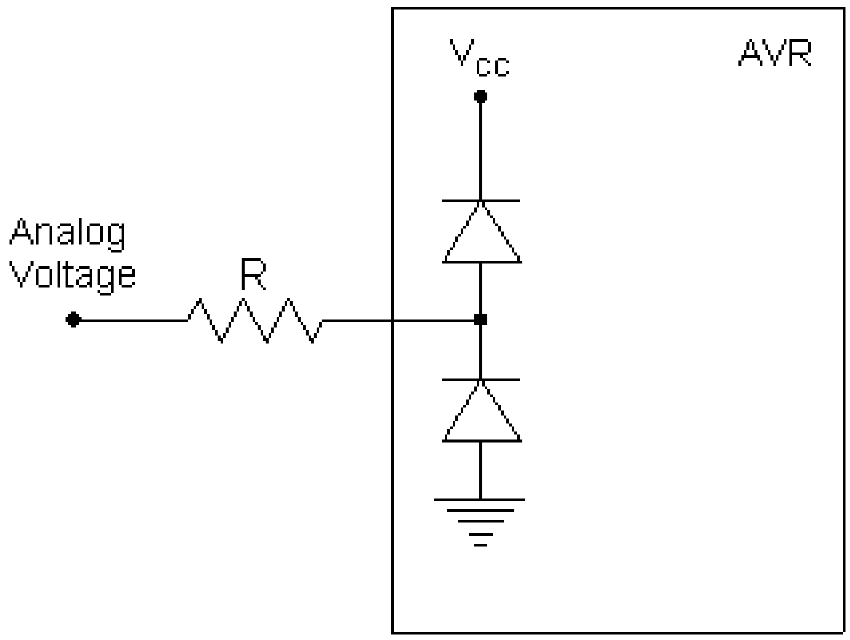


Bit	7	6	5	4	3	2	1	0	
0x06 (0x26)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

ADCSRA |= (1 « ADFR);

Beware the smoke

You MUST respect the voltage range allowed for the AVR pins (see Maximum Absolute Ratings in the Electrical Characteristics section of the datasheet). The voltage must be below VCC+0.5V and above -1V. If you don't respect this, you will blow your AVR. Be sure that the analog signals you are using are in the right range. If they come from the external world, is a good idea to use some kind of protection at the input. See the suggested circuit below (which consists of just one resistor...).



This circuit uses the internal clamping diodes present in all AVR I/O pins. If the analog voltage is higher than Vcc plus the conduction voltage of the diode (around 0.5V), the upper diode will conduct and the voltage at the input pin is clamped to Vcc+0.5 . On the other hand, if the analog voltage is lower than 0V minus the conduction voltage of the diode, the lower diode will conduct, and the voltage at the input pin is clamped to –0.5V. The resistor will limit the current through the conducting diode, which must not exceed 1mA, so you must design the resistor accordingly. For instance, if you expect that the maxim value that may reach the analog voltage is ±24V, the resistor value should be :

R=24V/1mA=24K.

Back to Summary

Programming in Assembly language

- Programming in Assembly language
  - Why Assembly
  - Assembler
  - Instruction set
  - Datasheets
  - Inputs and Outputs
  - Timing
    - Determining the period
    - Worked example
    - Timers and counters
  - Assembly Instruction Set
    - Data Transfer Instructions
    - Arithmetic Instructions
    - Logic Instructions
    - Compare Instructions
    - Rotate and Shift Instructions
  - Button-LED example
  - The Stack
  - Macros
  - Inline Assembly in C
  - LED blink delay example
  - Programming Structures in Assembly
    - Translate an If-then-else statement to AVR assembly code
    - Translate a for statement to AVR assembly code
    - Translate a switch statement to AVR assembly code
    - Translate a while statement to AVR assembly code

Why Assembly

At some point you will need to be in control of some parts of the hardware inside the microcontroller, or shrink the size of the program, or make it more efficient, so you will need to go for Assembly. Each line in assembly translate to a single instruction. Assembly instructions are shorter and more intuitive to remember.

First warning: Assembly is not for everyone. It is only for the brave. In Assembly, you are directly controlling the hardware. This is why I like it the most, because that means you also learn about how the hardware you are controlling works.

C *vs* Assembly is like going somewhere by Bus vs driving a car. Bus is more comfortable but you cannot go where you want when you want. If you want to go where the Bus service is not reaching or at a different time you have to drive a car. But driving a car requires you to steer the wheel yourself, brake and accelerate yourself, check the rpm, the water levels, and many other things. But these are awesome things, don't they?

Finally, you could even get a job at NASA if you learn Assembly.

Assembler

You need an assembler compiler to convert the `.asm` file to a `.hex` file that will be uploaded to the microcontroller. I use gavrasm, a command line tool available for windows and linux. Just download and unzip it. If you want the program to be available from everywhere in the terminal move it to `/usr/local/bin` :

```
$ sudo mv gavrasm /usr/local/bin
```

Instruction set

8 bit AVR Instruction set AVR Assembler user guide

Datasheets

Programming in assembly requires a big knowledge of the hardware since the instructions control this hardware directly. All the information about the microcontroller can be found in the datasheet. Some important microcontroller datasheets for Fab Academy are:

- attiny44
- attiny45
- atmega328

Inputs and Outputs

Here is a good resource shared by Sibu in order to understand ports and pins in the AVR microcontroller family and how to control them with the 3 registers DDRx,PORTx and PINx.

ATMEL AVR Tutorial 2: How to access Input/Output Ports?

Timing

One of the biggest advantages of assembly over other languages is the ability to precisely control time. It is even possible to actually predict how much time a program will take to execute.

Determining the period

Since we know the frecuency at which the MCU is running we can determine the period. For instance, for a MCU running at `f = 8 MHz` (8 million cycles per second), each cycle will take `T = 1/f` , that is, the period T is `T = 1/8000000 = 0.000000125 s = 0.125 us = 125 ns` . The following are common frecuency/period in AVR microcontrollers.

Frecuency	Period (us)	Period (ns)
1 MHz	1	1000
8 MHz	0.125	125
16 MHz	0.0625	62.5
20 MHz	0.05	50

So at 20MHz an instruction that takes one cycle in assembly is going to take 50 ns, and this is going to be as exact as your clock source, which to recall it is:

Clock source	Precision
Internal RC	10%
Internal RC Calibrated	1%
Resonator	0.5 %
Crystal	50 ppm

Worked example

Let's see how long int takes to execute this piece of assembly code:

```
LDI   R16, 255
LDI   R17, 100
ADD   R17, R16
MOV   R20, R17
```

We have 3 different instructions here `LDI` , `ADD` and `MOV` . If we look at the above AVR Assembler user guide, from section 4-5 onwards we can see that these instructions take exactly 1 cycle to execute each one. So this piece of code takes 4 cycles in total.

Since we know the period of a cycle, it is just a matter of multiplying `4 cycles*T` . So at 20 MHz this piece of code takes 200 ns.

Timers and counters

The following is a video from Basic and Default Usage of a Timer and Counter and The Microcontroller Clock explaining the basics about timers and counters.

Assembly Instruction Set

Data Transfer Instructions

- `mov dest,orig` copies the value of `orig` register to `dest` register

Arithmetic Instructions

- `DEC reg` decrease 1 the value of the register
- `ADD reg1,reg2` adds the value of reg1 to reg2

Logic Instructions

- `AND`
- `OR`
- `EOR`

Compare Instructions

- `CPI value1,value2` compares 2 values and results in 0 (no equal) or different from 0 (equal)

Rotate and Shift Instructions

- `SBI reg,bit` sets a bit in a register
- `CBI reg,bit` clears a bit in a register
- `BRNE label` jumps to the specified label if the result of `cpi` is 0
- `LDI reg,value` loads the specified value in the register
- `CLI reg` clears the entire register
- `JMP label` jumps to a certain label
- `RJMP label` jumps to a label at a maximum distance of -+ 2k words

Button-LED example

```
; buttonled.asm
;
; Fab Academy
; February 2016
; by Francisco Sanchez
; CC-BY-SA 4.0 License
;
; This program turns on the LED of a helloworld board when the button is pressed
```

```
.device attiny44                        ; defines which device to assemble for
.org 0                                  ; sets the programs origin
          SBI   DDRA,7                  ; sbi reg,bit Sets a bit of a register.
                                        ; Setting DDRA bit 7 makes pin PA7 a (digital) outpu
          CBI   DDRB,3                  ; sets PB3 as input
          SBI   PORTB, 3                ; activates pull up resistor in PB3
loop:                                   ; label for main loop
                                        ; labels must start with a letter and end with a col
          CPI   PINA,0                  ; compares the value of PINA (button) with 0
          BRNE  release                 ; the result was different from 0 so the button is
                                        ; released.Go to subroutine release
          SBI   PORTA,7                 ; the result was 0 so the button is pressed. Turn
                                        ; pin PA7 to HIGH (5V)
          JMP   done                    ; avoids subroutine release
release:
          CBI   PORTA,7                 ; Turns pin PA7 to LOW (0V)
done:                                   ; land here after turn PA7 HIGH
          RJMP  loop                    ; relative jump to label called loop
```

The Stack

LIFO Last In First Out

`PUSH reg,value` copies a byte of data from to the first empty byte at the top of the Stack

`POP reg` removes a byte of data from the top of the stack to the specified register

Macros

A macro is a group of instructions that you code once and are able to use as many times as necessary. The main difference between a macro and a subroutine is that the macro is expanded at the place where it is used, meaning it uses program memory for each of the instances. A macro can take up to 10 parameters referred to as @0-@9 and given as a coma delimited list.

```
.MACRO    DELAY                         ; This directive creates a macro called delay

.ENDMACRO                               ; Directive that ends the macro

DELAY
```

Inline Assembly in C

It is possible to write Assembly code within a C program by using the following syntax in C:

```
asm ("
// here your assembly code
    ")
```

LED blink delay example

This is a sample of how an assembly program looks like.

```
; ledblink.asm
;
; Fab Academy
; February 2016
; by Francisco Sanchez
; CC-BY-SA 4.0 License
;
; This program blinks the LED of a helloworld board

.device attiny44     ; defines which device to assemble for
.org 0              ; sets the programs origin

sbi DDRA, 7
; sbi(reg,bit): Sets a bit of a register.
; DDRA is the data direction register A
; Setting DDRA bit 7 makes pin PA7 a (digital) output
; A digital output can be switched ON/OFF for 5V or 0V

loop:
; label for main loop
; labels must start with a letter and end with a colon

sbi PORTA,7
; Sets bit 7 of register PORTA
; Turns pin PA7 to 5V

; Here it would come delay code

cbi PORTA,7
; Clears bit 7 of register PORTA
; Turns pin PA7 to 0V

; Here it would come delay code

rjmp loop
; relative jump to label called loop
```

Programming Structures in Assembly

Translate an If-then-else statement to AVR assembly code

Translate a for statement to AVR assembly code

Translate a switch statement to AVR assembly code

Translate a while statement to AVR assembly code

Back to Summary