Migrating from 8051 to AVR microcontrollers

Posted on May 2, 2008, by Ibrahim KAMAL, in Micro-controllers, tagged

While the 8051 microcontrollers are a brilliant invention, their strength remain in their simplicity and ease of use, but not in their processing power or diversity of peripheral features. The 8051 microcontroller is still a very useful device, but sooner or later, you'll feel that you need a more powerful microcontroller. That's why I am proposing to 8051 adepts, like me, this tutorials, that shows what are the key differences between those two families of microcontrollers. I'll also propose an interesting development environment for AVRs.

For this article, we are going to take as example the AT89S52 and the ATMEGA16 microcontrollers as two famous devices in the 8051 and the AVR families of ATMERL micro-controllers respectively.

Pre-requirements

In this article, I assume that you already have some basic microcontroller skills, preferably 8051. I assume you know what registers are, what is RAM, FLASH and EEPROM. I also assume you know some basic C programming. If you don't know anything about microcontrollers and programming them in C, I suggest you start by this **tutorial**, that is specially tailored to beginners.

Along this tutorial, I am going to use a freeware IDE, named VMLAB, which is an extremely powerful software, that lets you compile, debug and simulate your programs. It also allows you to virtually attach all kind of devices to your microcontroller like keypads, switches, LCDs, LEDs and resistors, and test the performance of your code with those devices. It can be easily learnt using the help file given with the software. The VMLAB software relies, like most AVR development tools, on the WinAVR GCC compiler, so any code from the internet will most probably work under VMLAB without any modification. In contrast, for the 8051 microcontrollers, the most famous development tool was the KEIL uVision, which has it's built-in C compilers.

Setting up VMLAB and WINAVR

The first problem to solve when starting in a new family of microcontrollers such as the AVRs, is to find a powerful enough, yet simple and effective IDE (integrated development environment). As I said before, we are going to use the freeware VMLAB simulator, and the open source WINAVR compiler for this purpose. So the fist thing that need to be explained is how to setup those two programs to co-work effectively.

After you downloaded both software (from **here** and from **here**), start by setting up WinAVR. Usually, the WINAVR installation will chose a strange name for the base folder, a name containing the date of the last release that you just downloaded. Preferably, rename that folder so that WinAVR is installed to "**C:\WinAVR**". Then, install VMLAB, and then as you can see in***figure 2.A***, go to the option menu, then click "Code Maker"

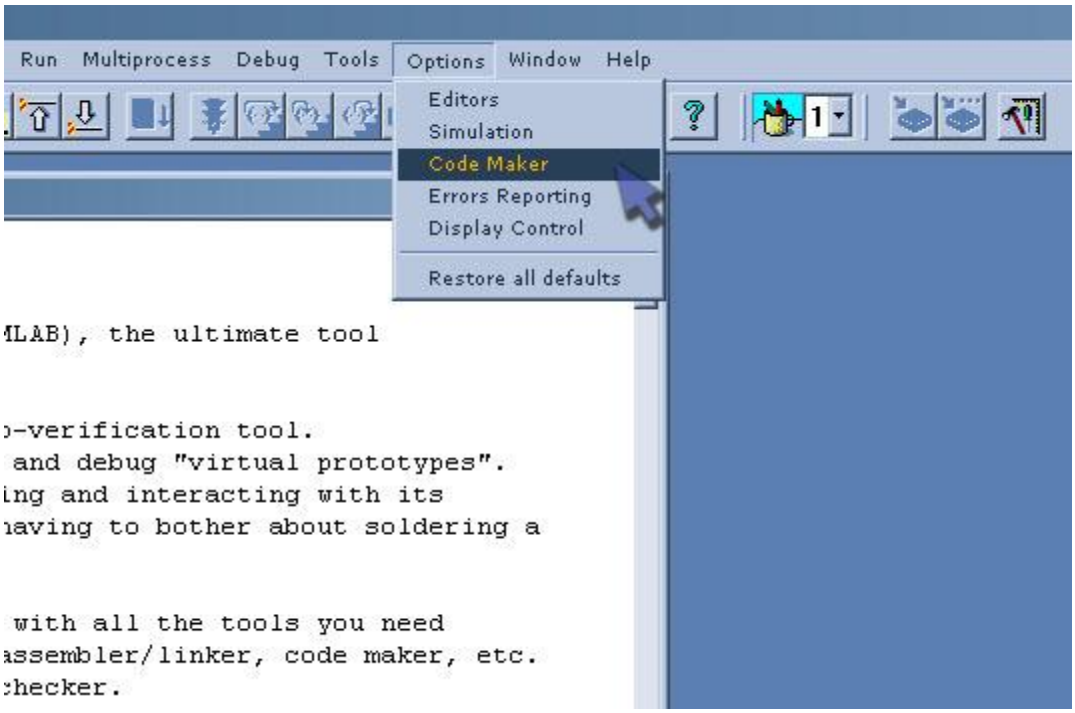

figure 2.A

In the Code Maker window, Chose the “GCC” tab, and configure it as you can see in ***figure 2.B***.

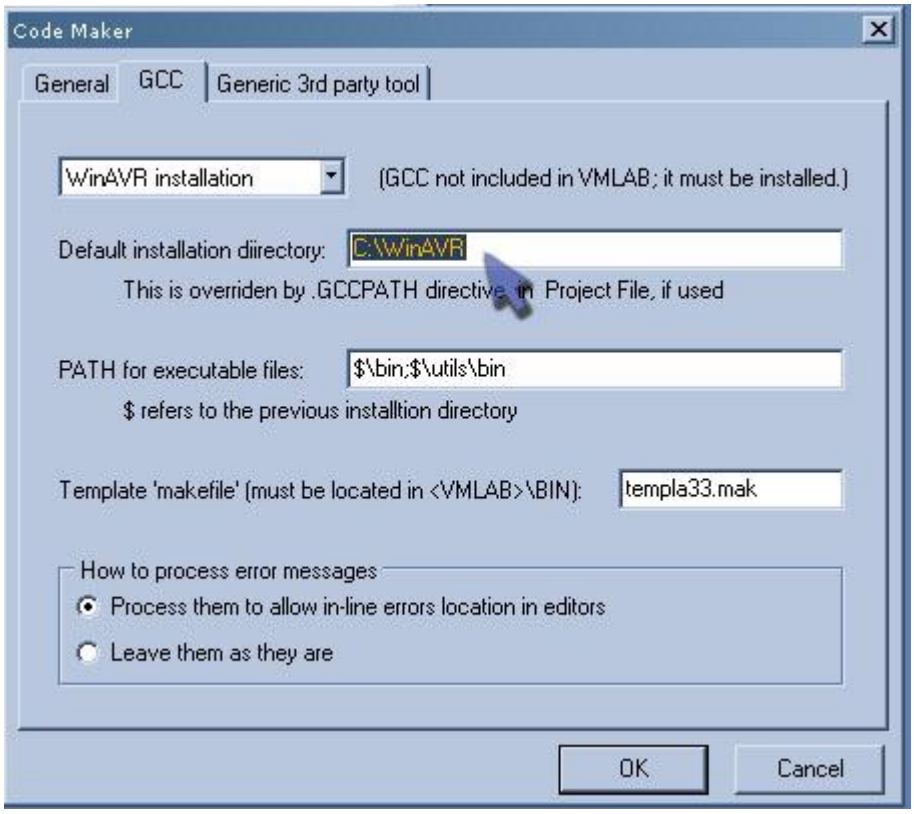

figure 2.B

At this point, Your development environment is ready to you, and you can start, building, testing and generating HEX files for programs to be loaded on you AVR microcontrollers. Simply go to "**Project -> New project**" and let VMLAB guide you through the new project wizard!

Breif AVR architecture overview

At the very beginning, you have to process the DATASHEET of the AVR device you are going to use. You will notice that the architecture of the AVR devices are not every different from a device to another. In need, there no difference between ATMEGA8, ATMEGA16, and ATMEGA32 but the size of the memories (FLASH and EEPROM).

A quick glance at the features of the ATMEGA16's DATASHEET can give us a brief overview of it's characteristics that makes it a more advanced microcontroller as compared to the 8051. Here is a small description of some of the most remarkable features:

- **Up to 16 MIPS**: The ATMEGA16, being an AVR core, can execute up to 16 Million Instruction per Second. This is due to the fact that most AVR instructions are executed on a single clock cycle. So with an 8 Mhz clock, you can perform 8 MIPS. Comparatively, a 8051 clocked at 8 Mhz, would give a throughput of only 666Khz, because 8051 need 12 clock cycles per instruction!
- **EEPROM Memory**: Most AVRs, including the ATMEGA16, come with an on chip EEPROM memory, with ready to use instruction to access this memory, as you will see along this tutorial (In case you don't know, EEPROM, is one where your program can store information that won't be lost even in case of power loss).
- **ISP**: **In System Programming** is becoming a standard in today's microcontroller, and AVRs fully benefit for this technological advance, making it much simpler for developer to test and debug their chips 'in system' (**more about ISP**).
- **Very Powerful and versatile TIMERS/COUNTERs**: Most AVR timers/counters have Prescallers, allowing them to be adapted to wide range of applications, and to dramatically reduce the processor overhead. They also have a high sampling rate, enabling them to count very fast external events.
  There is also a set of built-in devices, that importantly reduce the number of components in any project:

- **PWM channels**
- **ADCs**
- **USART/TWI/SPI interfaces**
- **On-Chip Analog comparator**
- **Internal RC oscillator**: This critical feature makes the ATMEGA16 for example, a microcontroller that can run with only 5V and GND rails, no any other component or connection is needed to be made to make it functional. By the way, any ATMEGA8, 16, or 32 microcontroller is shipped with the internal oscillator turned ON and tuned to 1 MHz, making it ready to be used with adding ANY external components like crystal resonator or capacitors. The Internal oscillator frequency can be then tuned to different frequencies, up to 8MHz.

Anther major characteristic of the AVRs, is that some of their internal register are READ-ONLY, and some are WRITE-ONLY. You have to stick to the datasheet of your microcontroller, and be careful with every register you're using.

Now that's you've seen to what extend the AVR family of microcontrollers advance the 8051, let's see how to program those microcontrollers, but before we start, make sure that you have correctly installed WinAVR and VMLAB, and that for every AVR project you create, you start by including the most basic header files using the following syntax:

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

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

Note that you have to include those same two headers files, with disregard to the type of micro-controller you are using, WinAVR will load the appropriate header file automatically.

Input/Output ports

Unlike 8051, but Like most microcontrollers, you have to specify the direction of the PINs of a port. Neverless, in AVRs, when a pin is configured as an INPUT pin, you can chose whether if it provides High impedance, or if it is connected to an internal pull up resistor. (Enabling the internal pull up resistor, can allow you to reduce further more the external components counts, when connecting a simple push button to ground, for example).

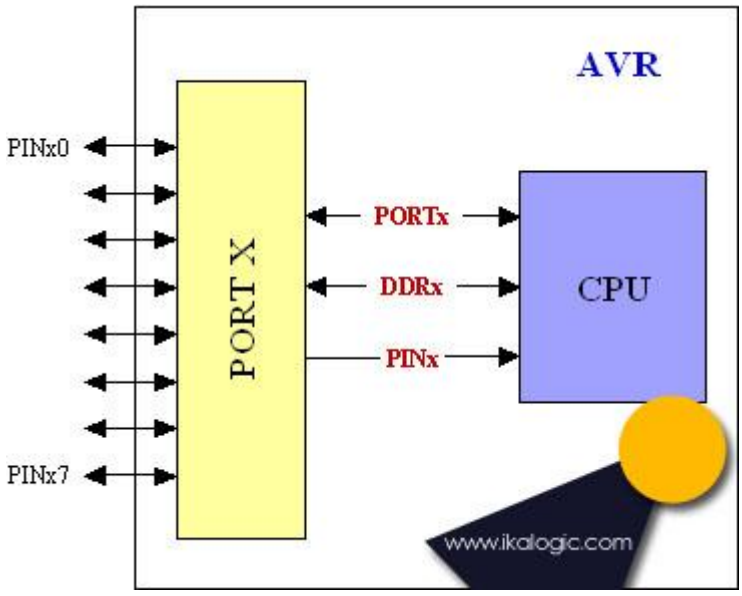

figure 4.A

Figure 4.A shows a simplified diagram of a general PORT of the ATMEGA16. The letter X can be replaced by the name of the port (A, B, C or D).

A general port is controlled by three registers: PORTx, DDRx, and PINx (again, where x is the name of the port).

- ***DDRx***: lets you configure the direction of the pins of the port 'x'. Each bit of the 8 bits of the DDRx register, represents one PIN of the concerned port. A 0 bit set the corresponding PIN as Input, a 1 sets it as output.
  For example, if:

```
DDRA = 0xF0;
```

  That means that PINs 0,1,2 and 3 of PORT A are set as Inputs, while the four others are set as outputs.

  DDRx is a READ/WRITE register.

- ***PORTx*** lets you define the logic level of the pins, in case they are configured as 'output'. But for pins that are configured as 'input', PORTx let's you define whether it is a high impedance input by writing 0 to PORTx, or pulled up input (internal pull up resistor enabled) by writing 1 to PORTx.
  The following example sets the PIN 0,1,2,3 as inputs, and the four others as output. Then, we enable the PULL-UP resistor for the fist 2 inputs (PIN 0 and 1), and output '0011′ on the four output pins:

```
DDRA = 0xF0;
PORTA = 0x33;
```

  PORTx is a READ/WRITE register.

- ***PINx*** lets you read the actual logic levels of the pins of a PORT. This register is the only way to read the state a PIN. In other words, the PINx register is used for INPUT operation.
  The following example uses the same pin configuration for PORT A above, but also inputs the state of the pins into a variables named 'pin_value':

```
DDRA = 0xF0;
PORTA = 0x33;
pin_value = PINA;
```

  Note that if the port have some pin configured as outputs, and some as input, then PINx register will read both inputs and outputs. It is the programmer responsibility to filter the unwanted data from the PINx register.

  PINx is a READ-ONLY register; you may not write data into it, and most compilers including VMLAB will warn you if you attempt to do so.

Interrupts vectors and timers

The syntax used in the GCC compiler to assign an interrupt vector to a function is the following:

```
SIGNAL(SIG_INT_VECTOR) {
...function's body...
}
```

Where SIGNAL is a normalized name (you cannot exchange it with any other name like my_function) and **SIG_INT_VECTOR**is the interrupt vector, and is to be replaced with a valid interrupt vector definition. You can find all the definitions of the interrupt vectors of your microcontroller in it's header file. For the ATMEGA16, here are some valid interrupt vectors, as copied from the its "iom16.h" file:

Interrupt name	vector
External Interrupt Request 0	SIG_INTERRUPT0
External Interrupt Request 1	SIG_INTERRUPT1
External Interrupt Request 2	SIG_INTERRUPT2
Timer/Counter0 Overflow	SIG_OVERFLOW0
Timer/Counter1 Overflow	SIG_OVERFLOW1
Timer/Counter2 Overflow	SIG_OVERFLOW2

For example, here the syntax for the code to be executed each time timer/counter1 overflows:

```
SIGNAL(SIG_OVERFLOW1) {
...code to be executed...
}
```

Then you can simply change the name of the interrupt vector to build other functions that would be triggered according to other events. Sure you have to enable those interrupt and configure those timers, which is done – as in 8051 microcontrollers – using a simple set of control register that can be consulted from the datasheet.

For example, the following source code sets up TIMER/COUNTER two in timer mode, and feeds it with the main CPU clock divided by 1024, then enables the corresponding overflow interrupt:

```
TCCR2 = (1<<CS22)|(1<<CS21)|(1<<CS20); // same as writing 0x07;
SREG = SREG | 0x80; //enable general interrupts in SR
TIMSK = TIMSK | (1<<TOIE2); //enable timer 2 overflow interrupt
```

In order to understand the meaning of those registers (TCCR2, SREG and TIMSK), take a quick look at pages 9 (for SREG), 129 (for TCCR2) and 134 (for TIMSK). You will see that they are clearly explained, and there is no need to reinvent the wheel and explain them in this tutorial.

So, According to that previous configuration, Timer/Counter 2 counting register being an 8 bits one, it will count from 0×00 to 0xFF and overflow in 256*1024 = 262144 clock pulses. IF you're using the ATMEGA16 right out from the box, it will be still configured to work on the internal 1Mhz oscillator, causing the timer to overflow every 0.26 seconds. Hence the following routing will be precisely executed every 0.262144 seconds:

```
SIGNAL(SIG_OVERFLOW2) {

...code to be executed every 0.26 seconds...

}
```

And all the other external interrupts and timers are configured in the same way. Just read carefully the datasheet and configure concerned registers correctly.

Note that you can always use the VMLAB's simulator, to view the current configuration of the peripherals you are using, whether they are timers, Analog to digital converters, serial communication protocols, or pwm generators. You can make the 'peripheral' window appear by clicking on "View -> Peripherals" As you can see in ***figure 5.A***, we are viewing the content of the TCCR2 just after it has been configured with 0×07, and VMLAB's interface tells you that your dividing the input clock by 1024. You can also view the counting register TCNT2 increasing during the each step of program execution. (advancing the execution of the program by one step can be done by pressing F6).

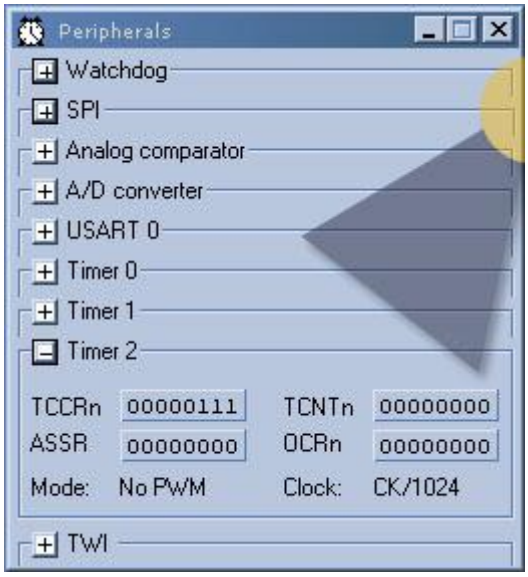

figure 5.A

It is very useful to be able to test, debug, and track as much programming errors as possible before transferring the code to the microcontroller, and VMLAB covers the testing and simulation of virtually all the features of the AVR microcontrollers. It can even simulate input Analog voltages to test you're ADC configuration!

Using the AVR's built-in EEPROM

Using the built-in EEPROM memory of most AVR devices is very simple, thanks to the 'eeprom.h' header file included in the WinAVR suite. It will allow you to store hundreds of Bytes of data, that are not volatile. The data stored in the EEPROM can be restored anytime, even if the microcontroller is unplugged from electricity for thousands of years! Some AVR microcontrollers however are not directly supported by the functions provided by WinAVR like the ATMEGA48 or the ATMEGA168, because they require different EEPROM handling algorithms.

In order to correctly use the EEPROM, you have to know the range of the EEPROM built into your microcontroller. For an ATMEGA16, the EEPROM is 512 bytes wide, so the address at which data is written can be anything from 0 to 511.

To be able use the EEPROM, you have to include this file at the top of your program:

```
#include <avr\eeprom.h>
```

Then you can use the following two functions to access the EEPROM. This first function writes the content of the variable 'data' into the EEPROM location defined by 'addr'

```
eeprom_write_byte(addr,data);
```

The second function read the content of the EEPROM location defined by 'addr' and store its content in the variable 'data'.

```
data = eeprom_read_byte(addr);
```

The following example C code shows how to write a byte to the EEPROM, then read it back

```
//variables declarations
char var1, var2;


//Store the content of 'var' in the first address of the EEPROM
var1 = 'a';
eeprom_write_byte(0,var);
```

```
//Read back the content of the first EEPROM cell

var2 = eeprom_read_byte(0); // var2 = 'a'!
```

The EEPROM functions included in WinAVR can get much more sophisticated, this is just one of the simplest approaches to read and write from the EEPROM.

Writing constants in the FLASH (program) memory

It can be useful to write large amounts of constants in the program memory, that won't usually fit in the RAM memory reserved for variables. Such data can be a WAV file, musical notes, sin and cosine look-up tables, or anything else you can imagine.

To gain access to the Program memory, start by including the following header file at the top of your program:

```
#include <avr\pgmspace.h>
```

Then when declaring a constant like Pi, use the following syntax (Pi and 3.14 are just used as an example):

```
float pi PROGMEM = 3.14;
```

To read the value of a constant, use the following syntax (still using Pi as an example, and 'var' is some variable where the content of 'Pi' is stored)

```
var = pgm_read_byte(&(pi));
```

Sure, to handle simple variables such as Pi, is not worth it, but using the program memory can be really useful when working with data arrays, like in the following example where a string of text is stored in program memory, then displayed on an LCD for example (the LCD function are not included)

```
// In the global variables declarations part:

char my_data[] PROGMEM = "Hello world, this text is stored in PROGMEM!";


//In the main function

i = 0;

while (pgm_read_byte(&(my_data[i])) != 0){

temp = pgm_read_byte(&(my_data[i]));

LCD_print(temp);

i++;
```

```
}
```

The applications and benefits from exploiting the program memory to store constants are countless, and, mixed with some ingenious algorithms, it can help you to increase the performance of your programs.

Generating PWM signals

The PWM signals are generated via the TIMER/COUNTER peripherals. The datasheet explains the different modes that can be used, check page 77 of ATMEGA16 datasheet. What is really interesting about PWM generators, is that they do not consume any processing time, once initialized, the PWM signals are generated continuously until you stop them.

To start generating PWM signal, the corresponding TIMER/COUNTER have to be configured as PWM generator, but also the PIN which will be used for PWM generation must be configured as OUTPUT. For example, this source code configures TIMER/COUNTER0 in FAST PWM mode:

```
//set PB3 as output.

TCCR0 = (1<<WGM00)|(1<<WGM01)|(1<<COM01)|(1<<CS00);

DDRB = DDRB | (1<<PB3);
```

Then, anywhere in the code, you can change the duty cycle by changing the value of the 8-bit register OCRn (where ‘n’ is the number of the timer being used).

```
OCR0 = 128; //generate a 50% duty cycle PWM signal
```

The value of OCRn determines the duty cycle, you can change this value anytime in the code. If OCRn is set to ’0′, the corresponding PWM signal will have a duty cycle of 0% (always OFF). If OCRn is set to ’255′, the corresponding PWM signal will have a duty cycle of 100%. In other words, you can adjust the duty cycle with a precision of roughly 0.4%.

Similarly, you can generate three other PWM signals, giving a total of 4 PWM channels. One of them is generate by TIMER/COUNTER0, another one with TIMER/COUNTER2, and the 2 last channels are generated with TIMER/COUNTER1. The last two pwm channels, that are generated by TIMER/COUNTER1, share the same frequency, but since TIMER/COUNTER1 is a 16 bit counter, the resolution of the duty cycle of those two PWM channels is 0.001 percent!

Using the ADC

The last interesting feature we are going to study is the 10 bit ADC that is found in most AVR microcontrollers. The ATMEGA16 has 8 ADC channels, but they are all multiplexed into one and only analog to digital converter, so practically, you can only convert of of those signals at a time. There are many ways to configure the ADC, including the ability to activate noise reduction mechanisms, and changing the reference voltage. For simplicity, we are going to use the ADC to convert a signal ranging

from 0V to 5V, and without any noise cancellation mechanisms. We are also going to read only the most significant 8 bits of the 10 bits result of the conversion.

Again, this tutorial is just and introduction, and we invite you to read the ATMEGA16 datasheet at page 205 for more information about ADC converter.

The following source codes setup the ADC as described before, chooses ADC0 among the 8 available sources, chooses the sampling frequency, but also enables the ADC interruption, causing an interruption to occur each time a conversion is finished (In case you don't know ADC, specially counting type and successive approximation type ADCs are particularly slow, that's why we use interrupts to use the processor for other task while the analog signal is being converted):

```
SREG = SREG | 0x80; //Enable global interrupts

ADMUX = (1<<REFS0)|(1<<ADLAR);

ADCSRA = ADCSRA | (1<<ADEN)| (1<<ADIE)| (1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
```

The configuration of those register are explained in detail in ATMEGA16 datasheet page 218 to 222.

Then, you can start a conversion anytime by setting the ADSC (ADC Start Conversions) bit in the ADCSRA, as the following:

```
ADCSRA = ADCSRA | (1<<ADSC); //Start Conversions
```

The microcontroller will immediately start sampling and converting the signal on PA0 (ADC1) and once the Conversions completed, an interrupt will be generated, the following interrupt service routing will be executed:

```
SIGNAL(SIG_ADC){ //ADC EOC interrupt

adc_data = ADCH; //store the 8 most significant bits

ADCSRA = ADCSRA | (1<<ADSC); //start conversion again

}
```

Optionally, you can re-initiate a new conversion each time this function is executed. There is also a special mode called 'free run' where the ADC continuously converts the input analog signal.

Again, I stress on the fact that there is still a lot more to discover in the datasheet, this tutorial only aims to get you used to the features of AVRs.

Finally, the ADC need at least one wire to be connected from AVCC pin (pin number 30) to VCC (5V). This is the power supply for the analog to digital converter. For even better results, ATMEL advises to use the connection scheme shown in ***figure 9.A***.

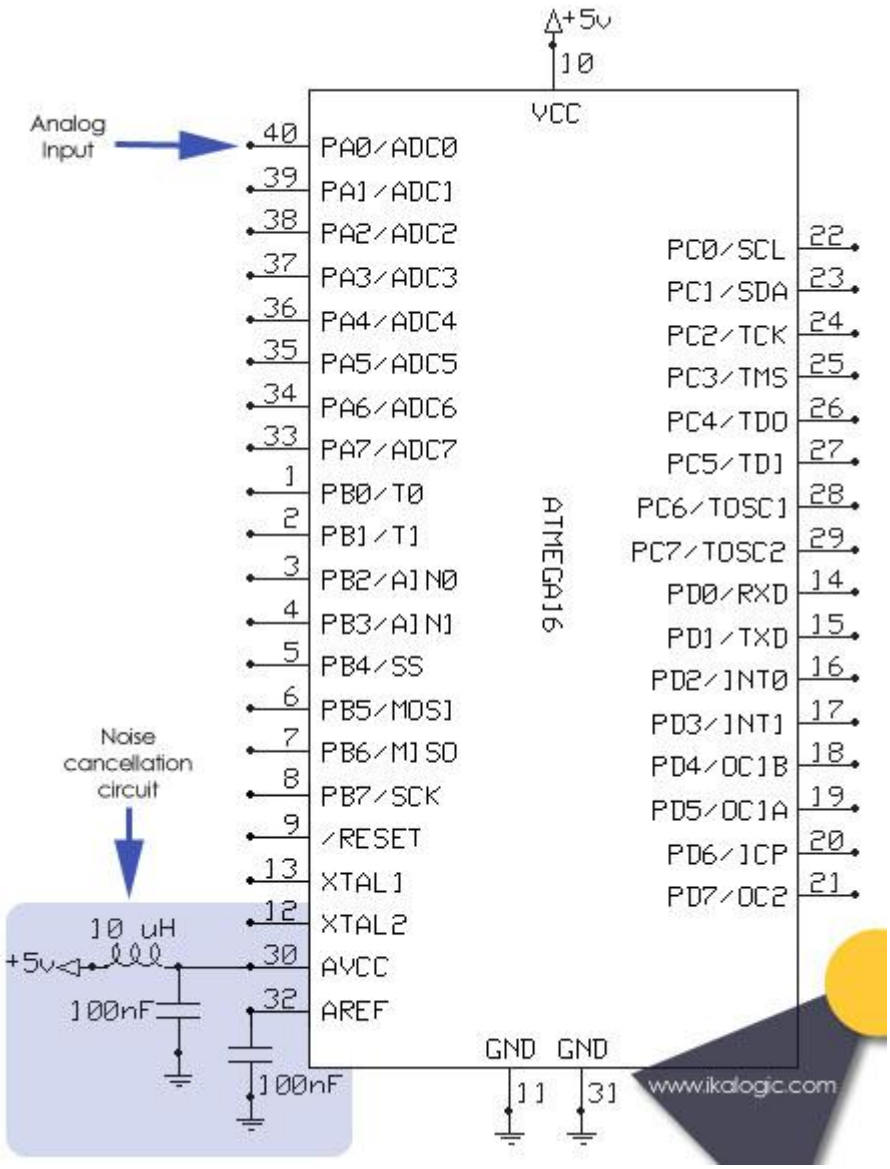

figure 9.A

Depending on the precision required by your application, you may ignore all the external components shown in the figure, and simply connect AVCC to 5V. In case you detect undesired noise in your measures, proceed first by adding the capacitors, then in case this does not solve the problem, try adding the inductor.

You may also – depending on your application – to implement some of the noise cancellation algorithms described in the datasheet.

Example project

This tutorial is concluded with a simple project, realized with an ATMEGA16 microcontroller, a red LED, a resistor and a potentiometer.

The potentiometer generates and analog voltage, ranging from 0 to 5 volts, which is fed to the ADC0 input on PA0. The microcontroller will have to read the analog voltage and then vary the duty cycle of a pwm signal to which the LED is connected according to the measured voltage.

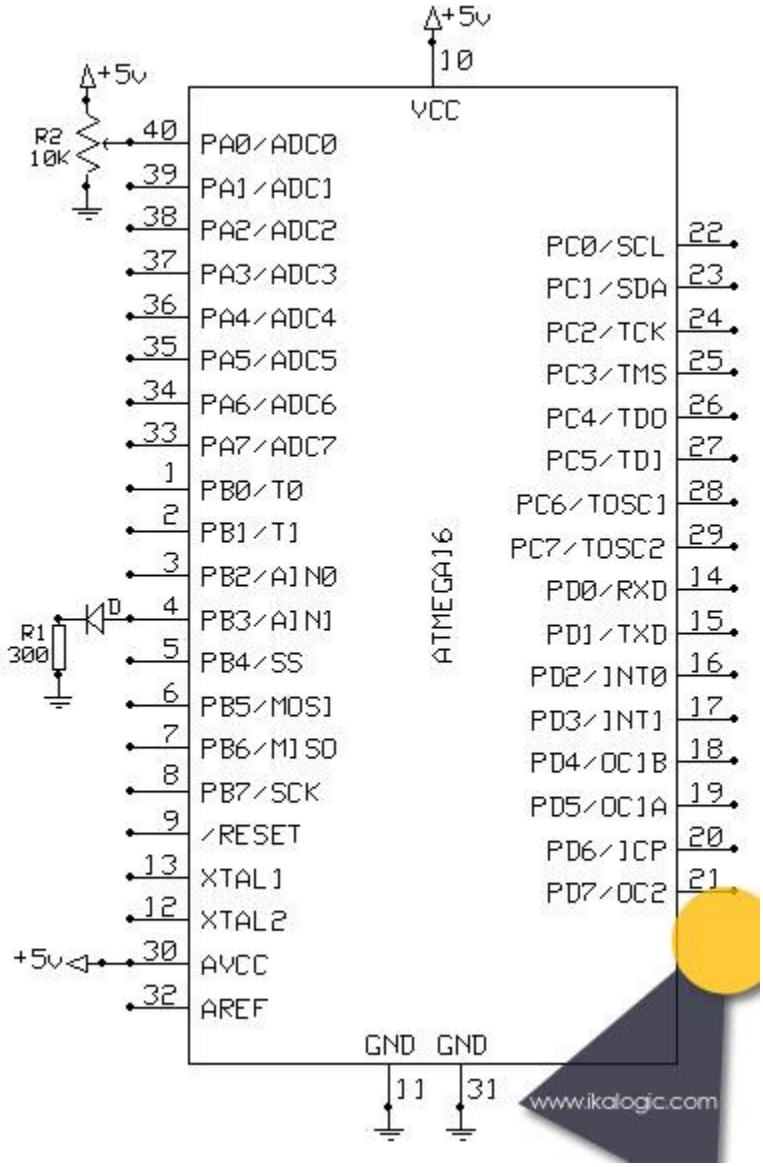

figure 10.A

The result would be a simple device that controls the brightness of a LED using a potentiometer. The project is a simple practical application of the aspects you should have learnt in this tutorial. As you can see in ***figure 10.A***, the wiring of this project if fairly simple, due to the small number of components.

Notice also that, due to the nature of this application, there is no need for important ADC precision, and hence, there is no LC filter added to the power supply of the ADC.

You can also see that there is no crystal nor RC oscillator, because we are using the internal RC oscillator, but we changed the fuse bits to increase it's frequency from the standard 1 Mhz to 8Mhz.

Below is the source code of the program that configures the ADC and PWM generator to perform the required task of controlling the brightness of the LED using the potentiometer:

```
#include <avr\io.h>
#include <avr\interrupt.h>

// Global variables
char adc_data;
setup_adc(){
SREG = SREG | 0x80;
ADMUX = (1<<REFS0)|(1<<ADLAR);
ADCSRA = ADCSRA | (1<<ADEN)| (1<<ADIE)| (1<<ADPS2)| (1<<ADPS1)| (1<<ADPS0);
}

void ini_pwm(){
TCCR0 = (1<<WGM00)|(1<<WGM01)|(1<<COM01)|(1<<CS00);
}

SIGNAL(SIG_ADC){ //ADC EOC interrupt
adc_data = ADCH;
ADCSRA = ADCSRA | (1<<ADSC); //re start conversion
}

// ********************************************************
// Main program
// ********************************************************
int main(void) {
```

```
DDRB = DDRB | (1<<PB3); //set PB3 as output.

DDRA = DDRA | (0<<PA0); //set PA0 as Input.


setup_adc(); //setup the ADC


ini_pwm();


ADCSRA = ADCSRA | (1<<ADSC); //start ADC conversion


while(1) { // Infinite loop

OCR0 = adc_data;

}

}
```

Before transferring the program to the microcontroller, you can take the time to fully test your code in VMLAB's simulator. You can simulate the potentiometer, then see in real time the variation of the PWM's duty cycle depending on the potentiometer's position. To do that, simply add the following lines in the ".prj" file of you're project, which you will find among the windows of the VMLAB interface:

```
.PLOT V(PB3) V(PA0)

R AVCC VSS 1

V PA0 VSS SLIDER_1(0 5)
```

Those line simple tell VMLAB to plot the voltage of the nodes PB3 and PA0, that the component SLIDER_1 is connected to PA0, with a voltage swing from 0 to 5 volts, and that AVCC and VSS are connected together with a 1 ohm resistor. You can very easily learn how to write this 'spice-like' script to simulate all kinds of external components by reading the help file provided with VMLAB.

The result of a simulation can then be visualized on the SCOPE window in VMLAB interface as in the following image where you can notice a clear PWM signal on the PB3 plot. You can also notice the peripherals window that was discussed before, and the "control panel" window where the the slider "S1″ can be adjusted to change the analog voltage at the pin PA0.

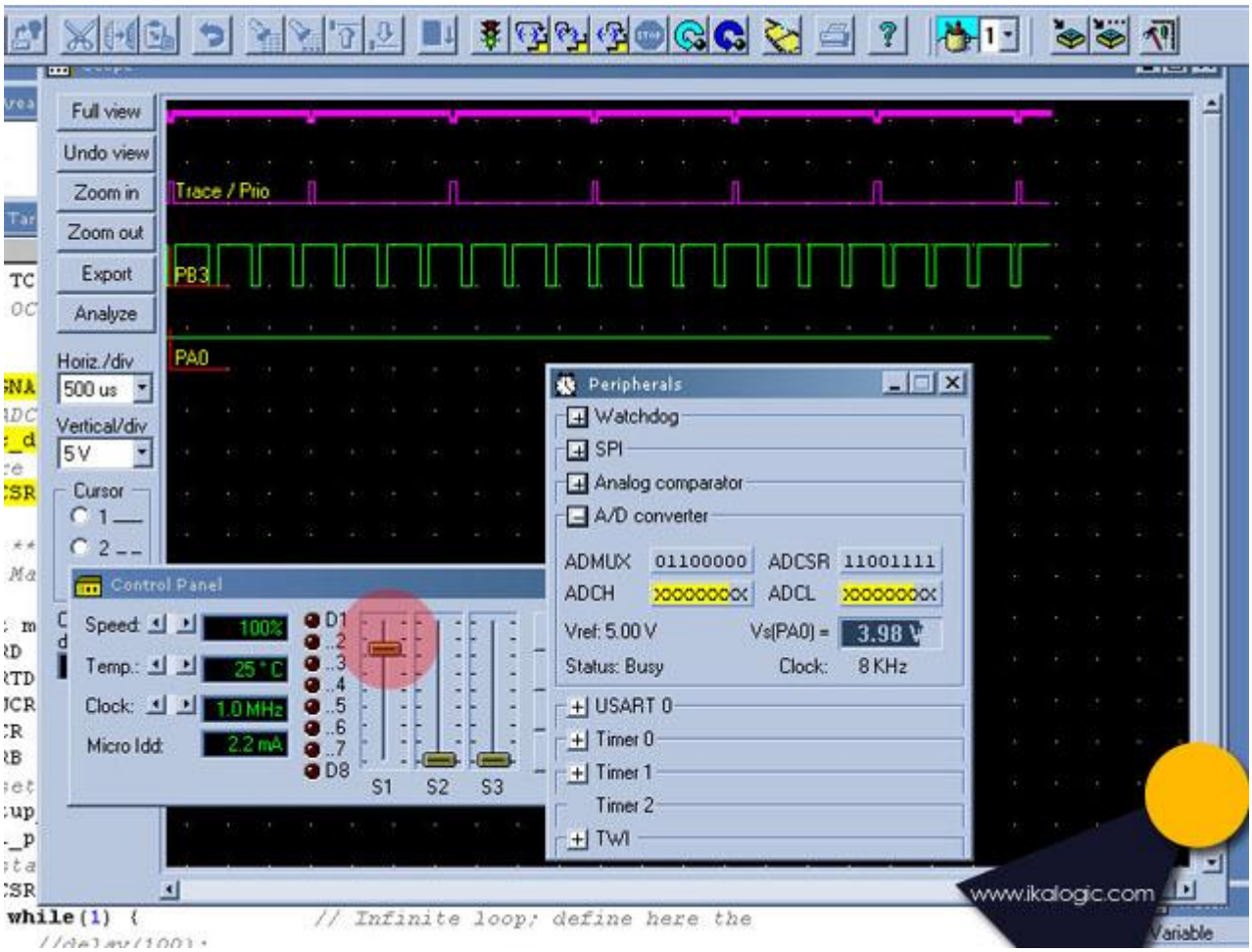

figure 10.B

When satisfied with the results, transfer your code to the microcontroller, using a burner like our simple ISP programmer.

This concludes this tutorial about migration from 8051 to AVR micro controllers. I hope it was useful, and comments and questions are more than welcome in the **microcontroller forums**.

Source: http://www.ikalogic.com/migrating-from-8051-to-avr-microcontrollers/