Introduction

Dear reader!

In front of you is a book – well not just a book, but more than that. This book will not only show you how to program AVR microcontrollers with Bascom-AVR software, but it will also show you microcontroller hardware which, with the help of your program, will breathe new life into your projects.

When I talk to people involved into programming microcontrollers, I often hear that the only way to enter this world, is to use either C or assembly language, because these two languages offer the best possible results. I partially agree with that statement. It's true that an engineer, who is fully-occupied with microcontroller designs, should use C and assembly language. One reason is that they have to switch from one microcontroller family to another very quickly – due to project targets, of course.

But for newbies to microcontroller programming, it's different – much different! Give them a C compiler, a development board and a goal to display some data on an LCD, and it is not a piece of cake. Often a newbie will even have problems with that simple a task.

It is a much different story with Bascom-AVR: ask the same newbie to perform the same task and it will be done in no time. People, who have attended Bascom seminars that I have presented, are impressed with Bascom's ease of programming, in fact, many say that they love to use Bascom. Once attracted to world of microcontrollers, there is no way back – he or she will probably program microcontrollers as either a hobby or professionally. I have, as a test, given our development boards along with a few minutes of lectures, to my son and his friend, when they were 10 years old. In no time at all, LEDs were blinking in various ways. Later, my son built himself an LCD scheduler for his school classes, as well as other projects.

In this book you will find the basics needed to start programming AVR microcontrollers. From the basics you will advance to the programming and use of some modern electronic devices, such as a graphical LCD with a touch-screen, GPS, various sensors and many other interesting devices which interface to microcontrollers.

I have known Mr. Vladimir Mitrović for more than 15 years, and I still remember the day when I showed him how to program with Bascom. He has truly embraced it and upgraded many Bascom programs with very clever assembly language programming - a winning combination. The ease of programming that is offered by Bascom combined with full access to the microcontroller, using assembly language, can beat any C compiler that we know of, when time-efficient programming is concerned.

I am very pleased that Mr. Vladimir Mitrović has decided to contribute some of his fine articles to this book. Without them, this book would not be the same. In the book, articles by other authors can also be found. The reason is simple: we found them to be very interesting and they fit the concept of the book. Thanks to Mr. Vladimir Mitrović and all of the contributing authors for their articles and programs. I also acknowledge the lecturers and the proof-reader, Mr. Brian Millier, well known Circuit Cellar writer and regular Bascom-AVR programmer, who adapted this book for English speaking readers.

I dedicate this book to my late son Miha who was a real inventor who loved designing- including microcontroller design.

Jure Mikeln

Table of Contents

7 INPUTS & OUTPUTS

In microcontroller applications push buttons are used in most cases. How to use them without unwanted contact «bounce« (what is debouncing anyway?), how we can intelligently increase the number of I/O pins of a microcontroller, driving DC motors and becoming familiar with PWM, are topics of this chapter.

Get your hands on an AVR microcontroller with help from Bascom-AVR and start controlling the world around you!

9 Short introduction to programming microcontrollers with Bascom-AVR
21 Programming AVR microcontrollers without a programmer
25 Improved Debounce Routines
31 Wait a minute!
37 I/O Expansion
43 Driving motors with AVR
53 Something about PWM in Bascom-AVR

55 DATA DISPLAYS

Data displays are very important in the world of microcontrollers. With modern graphic LCD displays, one can design smart-looking products. But in some cases the »classic« 2x16 alphanumeric LCD or even 7 segment LED display is better-suited. If you have a limited number of I/O pins on your microcontroller, you might even want to connect your LCD via an SPI interface. All this is covered in this chapter.

Pick the right display and make sure that your product will stand out!

57 LCD Character Displays
61 Multiple LCDs on the SPI bus
65 Graphical LCD with a touch screen
71 7-segment LED display

75 DATA MEASUREMENT

Human beings live in an analogue world and feel comfortable there. But this is not so for microcontrollers, which live in a digital world. After successfully measuring data, we have to transform it into digital values. We can do this in many ways, by using smart sensors (and smart programming) to get temperature, air pressure or even a GPS location – all with AVRs.

Get familiar with data measurement using Bascom-AVR!

77 Demystification of 1-Wire® commands
81 Using an SD card with the AVR – DOS File System
85 Measuring analogue values
91 Using a pressure sensor
97 GPS modules, their description and use

103 DEVELOPMENT TOOLS

Having programmed microcontrollers for many years, we have become regular users of development boards. There are many available on the market. Some expensive ones attempt to achieve universality by handling many different MCU models and including many different peripherals on-board. Others are nothing more than a »break-out« board for a specific MCU device.

In contrast, we have designed optimal development boards, that will meet most of your requirements while writing/testing your AVR programs. These boards emerged from extensive usage in our daily work, so there are very good reasons why our tools are designed as illustrated in this chapter.

Use smart tools when writing your Bascom-AVR programs!

105 .. MiniPin II development board
113 .. MegaPin development board
127 .. Proggy II, an in-system programmer for AVR microcontrollers
133 .. Debugging Bascom programs in AVR Studio
137 .. Let's explore Bascom-AVR
147 .. High-voltage Programmer for AVR Microcontrollers
153 .. UART to RS232 or USB adapters
157 .. LCD adapter for 16x2 and 8x2 LCD modules

159 PRACTICAL PROJECTS

There should be many practical projects in every book for programmers and this book is no exception. Bascom-AVR, in conjunction with AVR microcontrollers, is a winning combination when designing a simple (but very powerful) I2C analyzer. Other projects, like a Frequency generator, Frequency counter, a simple but accurate clock and a Metal detector are just a few of the projects that can be found in this chapter.

AVR microcontrollers are user-friendly, so get to know them better!

161 .. I^2C monitor
165 .. Programmable IR sender/receiver
173 .. "Bare-bones" programmable electronic load
177 .. A Frequency Reference Using the ATtiny2313
183 .. An ATtiny2313 Frequency meter
189 .. Simple RTC - Real Time Clock
191 .. The RFM12B expansion board
197 .. Shepherd's fire
201 .. Metal detector

209 PINOUT FOR THE AVR MICROCONTROLLERS

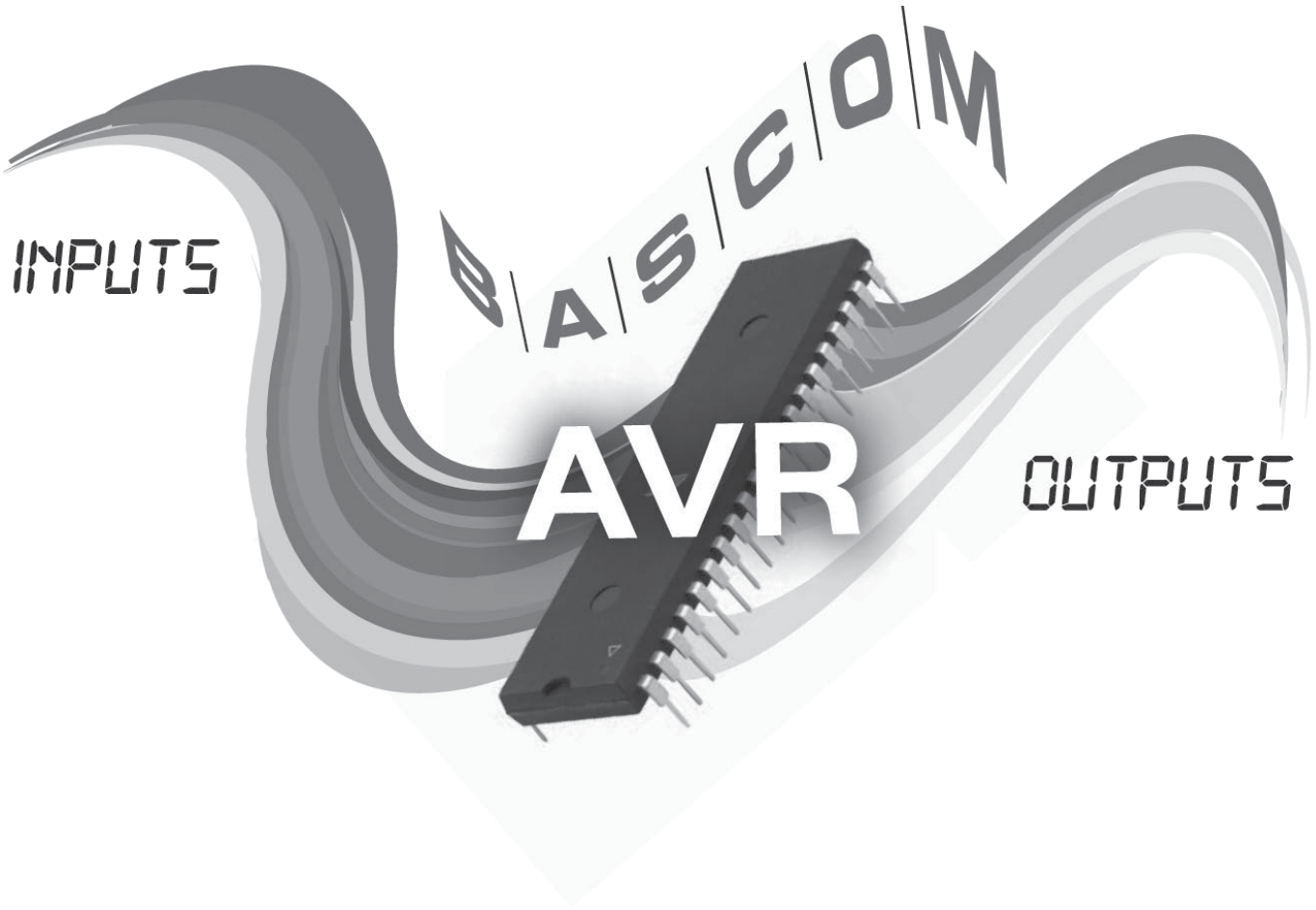
209 .. ATtiny25 \ ATtiny45 \ ATtiny85
209 .. ATtiny2313 \ ATtiny4313
209 .. ATmega8 \ ATmega16
210 .. ATmega32
210 .. ATmega8515
210 .. ATmega164 \ ATmega324 \ ATmega644 \ ATmega1284

Inputs & Outputs

7

In microcontroller applications push buttons are used in most cases. How to use them without unwanted contact «bounce« (what is debouncing anyway?), how we can intelligently increase the number of I/O pins of a microcontroller, driving DC motors and becoming familiar with PWM, are topics of this chapter.

Get your hands on an AVR microcontroller with help from Bascom-AVR and start controlling the world around you!



Improved Debounce Routines

By Vladimir Mitrović

Anyone who has ever tried using a microcontroller to count how many times a switch or any other mechanical contact has changed state, will certainly agree that this is not a simple task. The problem is that mechanical contacts "bounce" when changing state. The microcontroller is fast enough to register not only the "real" change of state, but also those unwanted changes caused by contact "bounce". Therefore, it's common that a single push of a switch can register as multiple switch closures.

Bascom's *Debounce* statement solves this problem, by checking the switch's state over again a bit later, when a level change is detected at an input pin. For example, if *Debounce* is configured to recognise a falling edge at an input pin (a logical level change from *High* to *Low*), it will delay the program execution for 25 ms and then check the input pin again to see if it is still at a logic "0" or not. If it is, an associated subroutine will be executed. If not, the level change will be considered as just a "bounce", and will be ignored. This reduces the probability of false readings significantly. Also, the *Debounce* statement reacts to any change of input state only once, and does not block program execution while waiting for the desired state change to happen.

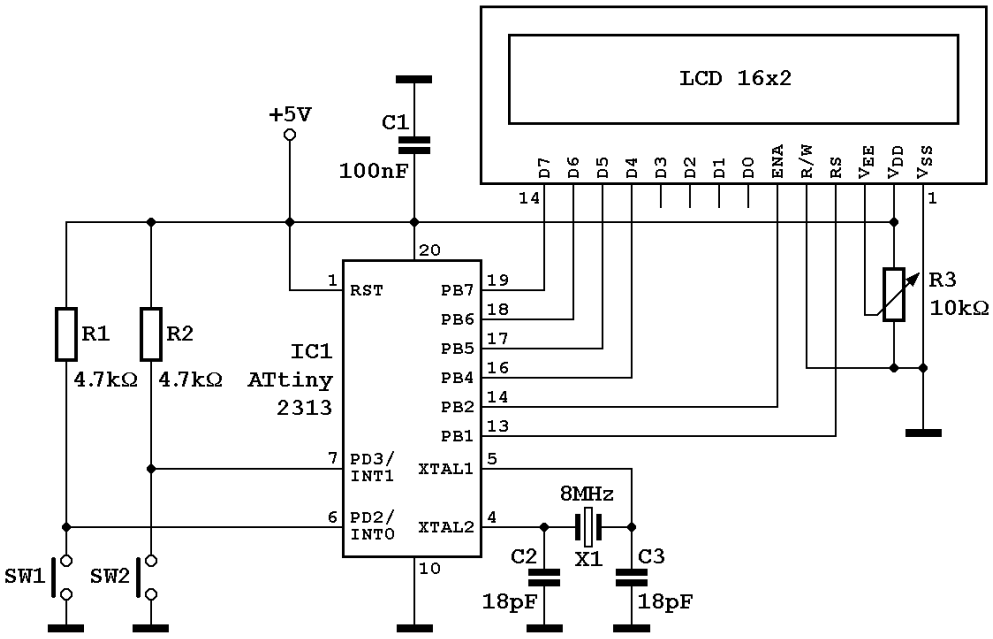


Figure 1: Test circuit for an improved Debounce

From my early days with Bascom, I realised the value of the *Debounce* statement and have even succeeded in expanding its capabilities. Let's see how!

Debouncing using external interrupts

The test circuit shown in Figure 1 is designed for testing the behaviour of the *Debounce* statement. An example of a suitable test program follows:

```
'Program *** Debounce1 ***

Dim Up_dn_counter As Byte

Sw1_pin Alias Pind.2
Config Sw1_pin = Input
Sw2_pin Alias Pind.3
Config Sw2_pin = Input

Up_dn_counter = 100

Do
   Debounce Sw1_pin , 0 , Sw1_sub , Sub
   Debounce Sw2_pin , 0 , Sw2_sub , Sub
   Home L
   Lcd "Counter = " ; Up_dn_counter ; " "
'do whatever
Loop

Sw1_sub:
   If Up_dn_counter > 0 Then
      Decr Up_dn_counter
   End If
Return

Sw2_sub:
   If Up_dn_counter < 255 Then
      Incr Up_dn_counter
   End If
Return
```

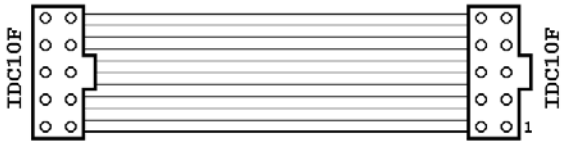


Figure 2: Sketch of the connecting cable

The cable connects the non-inverting inputs of the Power module to the pins of the selected I/O port, as well as the ground planes of both devices, and provides the positive voltage supply from MPIN to the Power module. This voltage (+5 V or +3.3 V) is used only for LED1 - the Power module requires its own power supply. The output voltage of this supply should be selected according to the requirements of the loads connected to the M0-M7 output terminals. You must allow for a 0.5-1.5 voltage drop within driver ICs IC1-IC4, depending upon the load current and the way in which the loads are connected - see the L272M data-sheet. The output voltage should be stable (but it does not have to be regulated) and the power rating depends upon the sum of the maximum currents through all connected loads.

Figure 3 shows how to connect the Power module to MPIN development boord. Although PORTD is used for this connection, any one of PORTA-PORTD connectors on MiniPin II or PORTA-PORTF on MegaPin can be used. Of course, which ports can actually be used depends upon the installed microcontroller and other devices (switches, 1-wire components etc.) that may already be connected to some of the I/O pins. In the following text,

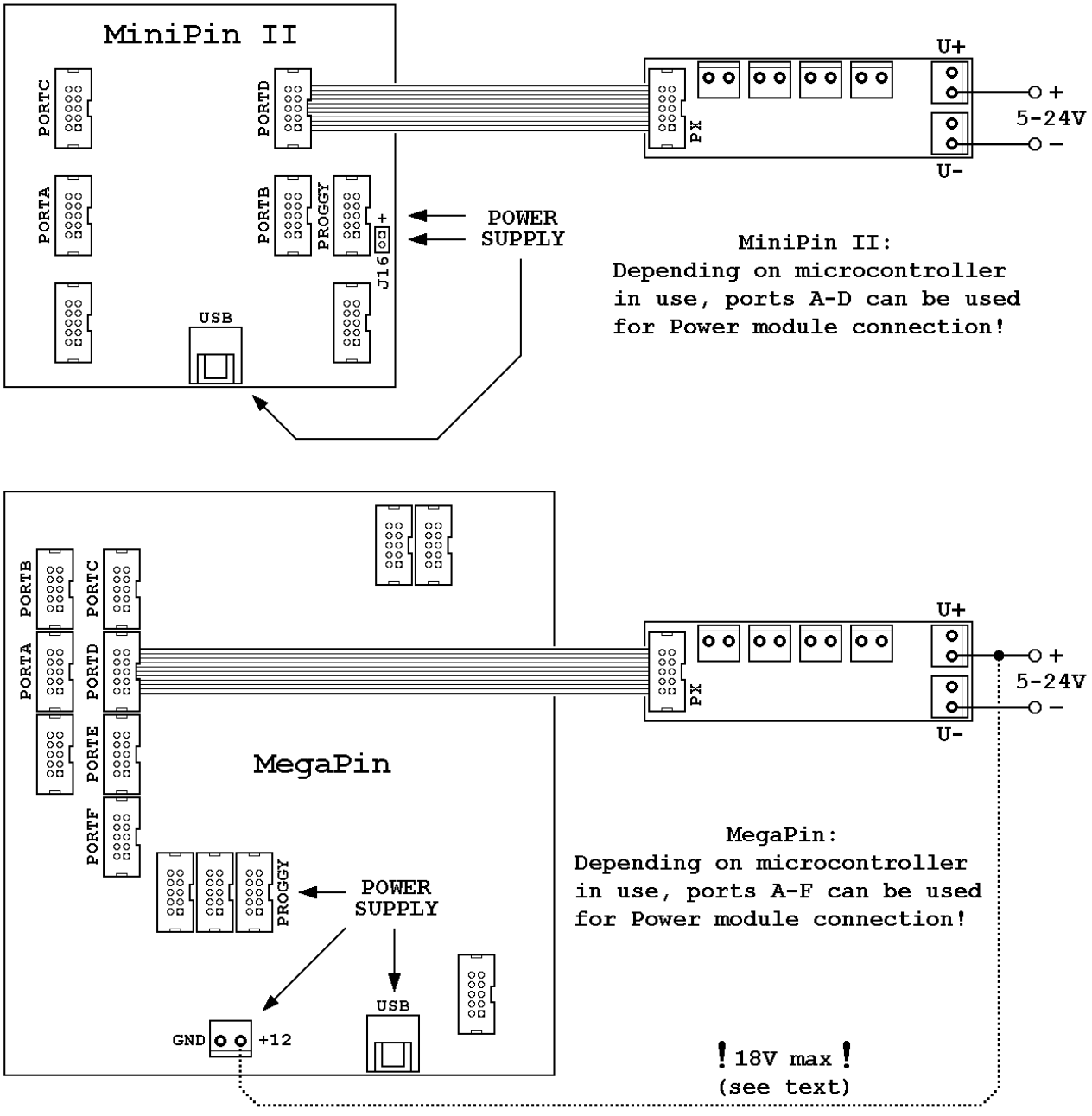


Figure 3: How to connect a Power module to MiniPin II (top) or to MegaPin (bottom)

```
                        'a while
Gosub Motor1_stop       'stop Motor1
```

DC motor – changing the direction of rotation

If we want to control the direction of rotation, we have to connect a DC motor between two power outputs. For example, Motor3 is connected between the M2 and the M3 outputs, as shown in Figure 5. This motor is stopped if both control outputs are at the same logic level and rotates if the control outputs are at different logic levels. The direction of rotation is determined by the logic levels on the outputs (*Low-High* or *High-Low*).

Control subroutines for a motor connected as Motor3 are:

```
'* DC Motor3 subroutines

Motor3_stop:        'stop Motor3
   PORTD.2 = 0
   PORTD.3 = 0
Return

Motor3_left:        'start Motor3,
                    'left rotation
Motor3_ccw:         'start Motor3,
                    'CCW (alternate label)
   PORTD.2 = 1
   PORTD.3 = 0
Return

Motor3_right:       'start Motor3,
                    'right rotation
Motor3_cw:          'start Motor3,
                    'CW (alternate label)
   PORTD.2 = 0
   PORTD.3 = 1
Return
```

Now we can control Motor3 by calling appropriate subroutines:

```
Gosub Motor3_left       'start Motor3,
                        'left rotation
Wait 5                  'let it run
                        'for a while
Gosub Motor3_stop       'stop Motor3
Wait 5
Gosub Motor3_right      'start Motor3,
                        'right rotation
Wait 5                  'let it run
                        'for a while
Gosub Motor3_stop       'stop Motor3
```

DC motor - change of rotation speed

We can effectively control the speed of a DC motor using pulse-width modulation (PWM). In the following example, the voltage across Motor1 is alternatively switched on and off in 10 ms intervals. The average current thru the motor is halved and the motor rotates more slowly:

```
'* DC Motor1 speed control

Dim Dc_i As Byte
For Dc_i = 1 To 250     'speed is
                        'controlled
   Gosub Motor1_start   'by alternative
                        'switching on
   Waitms 10
   Gosub Motor1_stop    'and switching off
   Waitms 10            'the motor
Next
```

The duration of one cycle (ON + OFF) is 20 ms, which results in a frequency of 50 Hz (i.e., the motor is switched on and off 50 times per second). This value was suitable for all small DC motors that I tested. If the motor being used works improperly (if it jerks or hums), double the switching frequency by shortening the ON and OFF intervals to 5 ms.

The motor speed is controlled by changing the ratio between the ON and OFF periods. For example, if we want to slow down the motor, we should shorten the ON and extend the OFF period. Keep the total cycle time unchanged when changing speed (for example, if you shorten the ON interval to 5 ms, the OFF interval should be 15 ms). Every DC motor has a minimal ON:OFF ratio at which it stops rotating.

The total motor running time equals the execution time of the *For-Next* loop. In the above example, it is 250*20ms=5s. For longer periods, variable *Dc_i* should be dimensioned as Word or Long. Although fully functional, the preceding example has a significant disadvantage: the program cannot perform other functions while controlling the motor. A better solution is to implement a motor control within an interrupt service routine that gets invoked frequently.

```
Dim Motor1_speed As Byte
On Timer0 Tim0_sub
Enable Interrupts
Config Timer0 = Timer , Prescale = 64
```

NOTE

The terms "left" and "right" refer to "counter-clockwise" and "clockwise", respectively. The actual direction of rotation also depends upon the way the motor is connected to the Power module. If the motor rotates in the opposite direction than expected, reverse the connecting wires or complement the logic levels of the control outputs in the program (in this example PORTD.2 and PORTD.3).

and temperature, but values that are used, along with the EEPROM constants, in calculating the actual pressure and temperature.

There are two I²C addresses that are used to communicate with the HP03M's sensors:
» EE hex to write to the sensor,
» EF hex to read from the sensor.

Flow diagram to read air pressure is as follows:

S	11101110	A	11111111	A	11110000	A	P	D	S	11101110

A	11111101	A	S	11101111	A	MSB	A	LSB	N	P

Flow diagram to read temperature is as follows:

S	11101110	A	11111111	A	11110000	A	P	D	S	11101110

A	11111101	A	S	11101111	A	MSB	A	LSB	N	P

» S I2C Start
» A ACK
» P I2C Stop
» N NCK
» D pause 40 ms
» MSB result higher byte
» LSB result lower byte

Let me point out that Bascom-AVR knows *ACK* command only at reading telegram on I2C bus but not at writing. Why has HopeRF included *ACK* also at writing it is not known to me.

The sensor's manufacturer has issued some warnings:
» before starting an A/D conversion, set XCLR to a logical 1, to take the device out of the Reset state,
» after power-up, you should ignore the first data reading and use only subsequent ones.

Figure 3 shows a schematic diagram of the sensor connected to the microcontroller. For the prototype, I soldered the sensor to a DIL8 socket, which I then soldered to the protoboard. For the A/D convertor's clock we would normally use an external 32.768 kHz crystal oscillator. Instead, I decided to generate that using a PWM signal from the microcontroller.

The Bascom program

At the start of the program, we have to define the microcontroller being used, the crystal frequency and the baud rate (for communication). Since a PWM signal will be used for generating the 32.768 KHz clock source, we define Timer1 as a PWM generator and the OC1A pin as the PWM output pin. The OCR1A value that is needed to generate this frequency is calculated with this simple formula:

$$OCR1A=(F_{crystal}/(2*32768))-1$$

Variable N represents the prescale factor (1, 8, 32, 64, 128, 256, or 1024. Timer1 configuration looks like this:

```
Config Timer1 = Timer , Prescale = 1 , _
Compare A = Toggle , Clear Timer = 1
Start Timer1
Ocr1a = 182              'F=32.768 kHz
Oc1a_pin Alias Portb.1   'PWM output signal
                         'connect to pin MCLK
Config Oc1a_pin = Output
```

Let's continue with the definitions of the other pins that are used: SCL, SDA and XCLR, all of which should be immediately set to a low level. According to the datasheet, the HP03M has a max. I²C clock frequency of 100 kHz. Since I was using this particular sensor, I had to introduce an I²C delay:

```
Config I2cdelay = 15
```

Note that the HM03MA has a max. I²C clock frequency of 500 kHz, and therefore does not need this statement.

Next, the variables and constants, needed to calculate the result, are dimensioned/declared. There are a couple of constants needed for the calculations. I decided to define them in advance, as follows.

```
Const K1 = 32            '2^5
Const K2 = 16384         '2^14
Const K3 = 65536         '2^16
Const K4 = 1024          '2^10
Const K5 = 67108864      '2^26
Const Adresw = &HA0      'EEPROM WRITE
                         'ADDRESS
Const Adresr = &HA1      'EEPROM READ
                         'ADDRESS
```

We also have to define the LCD display parameters, if we use it for debugging. Similarly, we can also use the

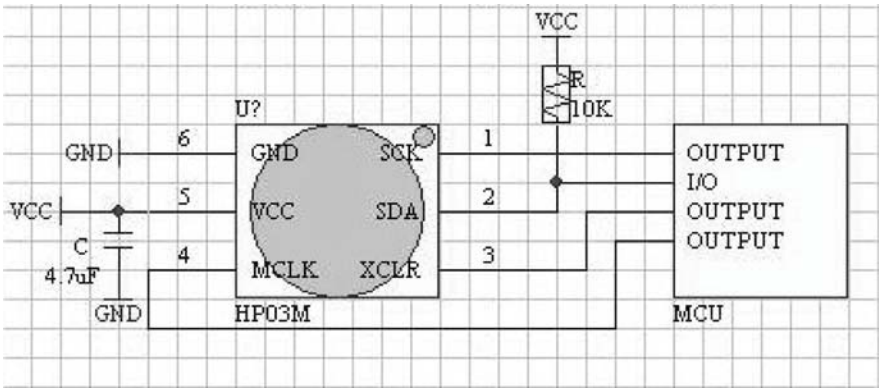


Figure 3: Typical application circuit.

Conclusion

Writing data and event - logging to an MMC card has proven to be an excellent tool for monitoring what is happening within a program, as well as for debugging. It's also very handy for logging events like when an alarm was tripped, who has entered a code via the keyboard, or for logging temperature and other parameters over long periods of time. This data can be saved in such a way that MS Excel or other spreadsheet programs can read the file.

Unfortunately, AVR-DOS requires a significant amount of RAM: you can see this if you examine the Bascom-AVR Report file (using Program/Show Result in the Bascom-AVR menu bar). This will limit your choice of AVR micro-controllers to those with sufficient RAM resources.

Example program ID	
Name:	Test_SD_B_Read_HW-UART.bas
Microcontroller:	ATmega32
Testing circuit:	Figure 1
MiniPin compatibility:	yes
MegaPin compatibility:	yes
Program tests SD memory card on SD card adapter; does reading from the card	

Example program ID	
Name:	Test_SD_B_Write_HW-UART.bas
Microcontroller:	ATmega32
Testing circuit:	Figure 1
MiniPin compatibility:	yes
MegaPin compatibility:	yes
Program tests SD memory card on SD card adapter; does writing to the card	

Example program ID	
Name:	Config_MMC.bas
Microcontroller:	ATmega32
Testing circuit:	Figure 1
MiniPin compatibility:	yes
MegaPin compatibility:	yes
Configures SD card	

Example program ID	
Name:	CONFIG_AVR-DOS.bas
Microcontroller:	ATmega32
Testing circuit:	Figure 1
MiniPin compatibility:	yes
MegaPin compatibility:	yes
Configures AVR-DOS. Note: this file cannot be used for commercial purposes. Contact author in that case	

Debugging Bascom programs in AVR Studio

By Jurij Mikeln

Debugging is a procedure, where we trace program flow with the help of suitable software and/or hardware. While tracing program flow, we can find annoying "bugs" inside our program. If we do not have suitable debugging hardware we can use a simulator, like the one included in Bascom-AVR. We can also find problems by displaying some "debugging data" on an LCD display or in a terminal window, if we send data out to the RS232 or USB port. I will show you how you can debug Bascom programs with a help of the AVR Studio 4 program and the MegaPin development board.

As mentioned, we can debug in many ways. Generally we use a principle of displaying "debugging information" on an LCD display or terminal window. The data displayed can determine whether or not our program is performing properly.

That is a quite usable procedure if no debugging tools are on hand, however, this procedure is sometimes not good enough. In such cases, we have to look "inside" the microcontroller to find those nasty bugs that may be elusive. Atmel has provided many free software tools that, in conjunction with suitable hardware, work very well. A good example is Atmel's JTAG ICE, which we'll demonstrate in conjunction with the MegaPin development board (which has proven to be a real multi-purpose development board!)

What do we need?

To debug using the AVR Studio 4 program, some device preparation (hardware & software) needs to be done. For hardware, you will need a JTAG ICE unit, the source of which can be found on the web. Alternately, you can use the JTAG ICE included on the MegaPin board. One must use an AVR device that supports JTAG, such as the ATmega16, ATmega32, to name a few. Also, we must enable JTAG operation with the appropriate Fuse bit, as shown in Figure 1. Note that all AVRs with JTAG available come factory-programmed with the JTAG Fuse bit enabled.

The JTAG Fuse bit can be turned on with the SPI program-

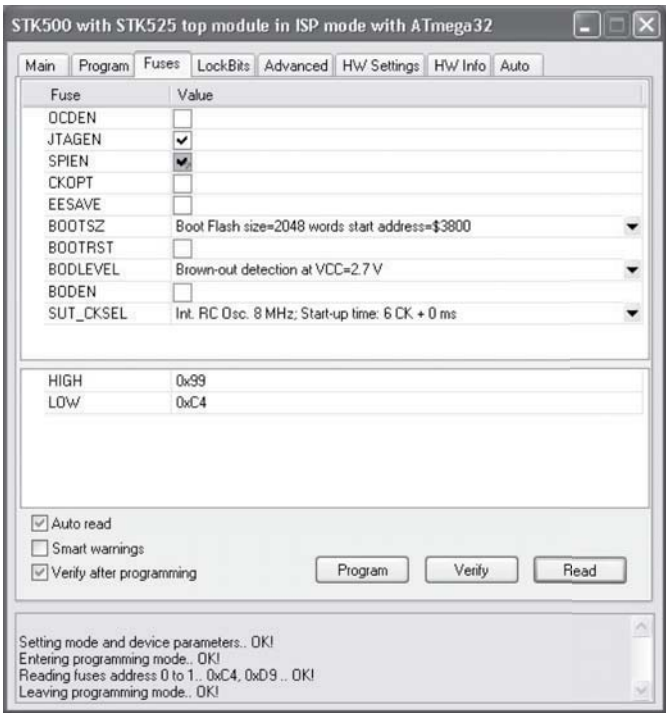


Figure 1: Enabling the JTAG Fuse bit

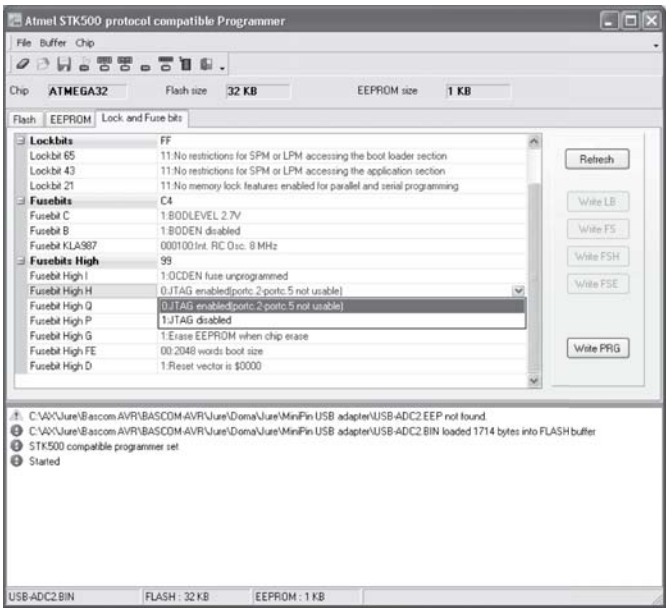


Figure 2: Setting the Fuse bit within the STK500 native programmer window

lcd-spi library, enabling the LCD display to work. Without these statements the PWM electronic load will not perform correctly and LCD will not display anything.

As it stands, this program does not display on the LCD the internal resistance of the power transistors. That is a drawback, but you have to bear in mind that each power output transistor has its own internal resistance depending upon the Gate-Source drive voltage. That would mean that we would have to incorporate these internal resistance figures into a table in the program, and change the table if other transistor were used. That would be very impractical, so we decided that the LCD will show PWM drive value ranging from 0 to 255, which represents the highest and lowest resistance respectively.

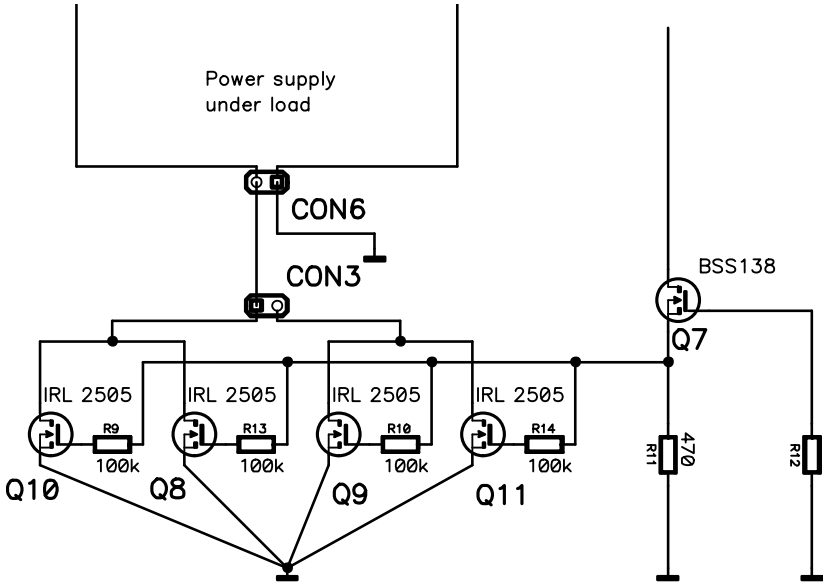


Figure 4: Connection of two parallel MOSFETs to GND

Possible improvements

When testing the electronic power load, we noticed that the heat was not evenly distributed amongst the output transistors. That was due to the temperature dependence of the MOSFET's internal resistance.

To improve that we could measure temperature and change PWM drive signal accordingly. A good solution is to connect a low Ohm resistance from the Source of each transistor to ground. The resistance should be 0.5 Ohms or less.

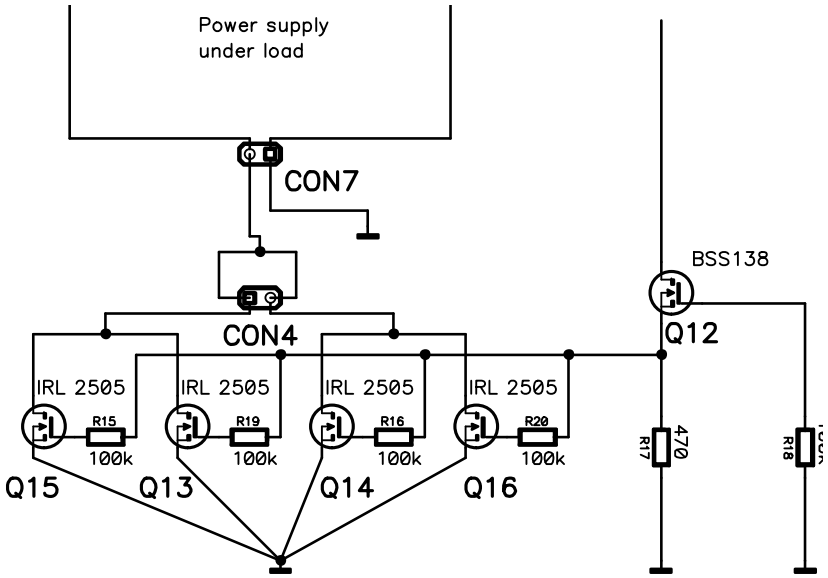


Figure 5: Parallel connection of all MOSFETs

Conclusion

This project is "bare-bones" in both circuitry and software. The LCD displays only the PWM value and not the resistance, because setting/displaying the MOSFET's internal resistance would be very impractical as it changes with each transistor and with temperature. For those of you who would like to use this electronic load with larger currents, let me tell you that I have successfully tried the IRFP4368PBF, which is rated for 350 A! I wish you successful "electronic loading"!

Example program ID	
Name:	PWM load.bas
Microcontroller:	Any ATtiny AVRs
Testing circuit:	Figure 1
MiniPin compatibility:	/
MegaPin compatibility:	/
Use multi_lcd-SPI.lib	

Example program ID - Library	
Name:	lcd-spi.lib
Microcontroller:	/
Testing circuit:	/
MiniPin compatibility:	/
MegaPin compatibility:	/
Library for use with above	

Now let's look at the software.

The RFM12B module is configured once, at the start of the program. A description of all the commands can be found in the RFM12B datasheet. In the **configuration** section of the datasheet, a list of the various settings is given (channel No., Baud rate etc.). The same initialization settings must also be included in the receiver program as well.

After configuration, the program will flash both the Tx and Rx LEDs as a status indication.

Following initialization, the main program loops continuously: checking both the state of the switches as well as the packet "transmission complete" flag. Based upon those checks, it calls the appropriate routines.

A Short description of the subroutines

Subroutine for switch 1 (sends only one data packet when switch is pressed):

- transmits "T1 on....." and displays the same message on the LCD,
- checks the state of switch T1,
- wait until T1 switch is released,
- return to main loop.

Subroutine for switch 2 (sends data continuously while T2 switch is pressed):

- sends "T2 on ....." and displays the same message on the LCD,
- check the state of switch 2,
- if switch T2 is still pressed then go back to the start of this subroutine,
- else return to main loop.

The Receiving program description

The receiving program follows the flowchart shown in Figure 5. (Not shown is the routine which displays data on the LCD.)

The received data is analyzed and if it matches one of the acceptable patterns, a suitable subroutine is executed.

The receiving module configuration is similar to that of the transmitting module. It's done once at power-up. After configuration, the Rx and Tx LEDs are flashed, as a status indication. In the main loop, the *Test* flag is checked, to see whether an acceptable data pattern was received. If not, then the program returns to a start of the main loop. If the flag is set, then we analyze the received data in a suitable subroutine.

A Short description of the other subroutines

An Interrupt subroutine is executed every time a byte is received. This byte is added to a string. When the byte equals a CR (ASCII value = 13), then this indicates the end of the string, and we set the *Test* flag. Then the subroutine returns to the main program loop.

The String analyze subroutine compares the incoming string, and takes action according to the value of the string. It also displays the data on the LCD.

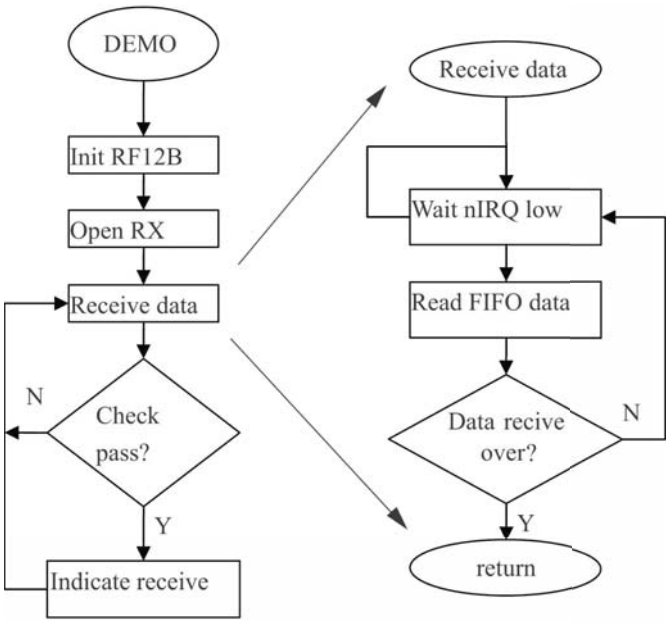
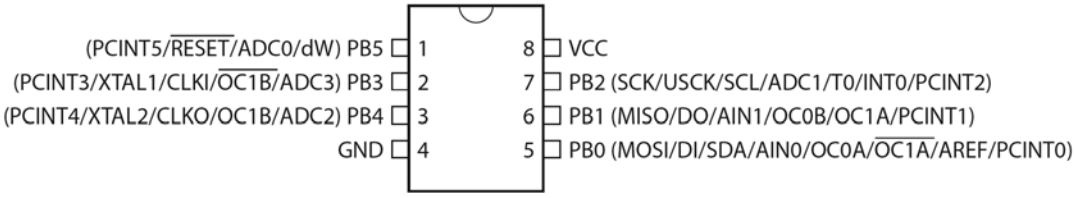


Figure 5: The Receiving program flowchart

The RFM12B EB schematic diagram

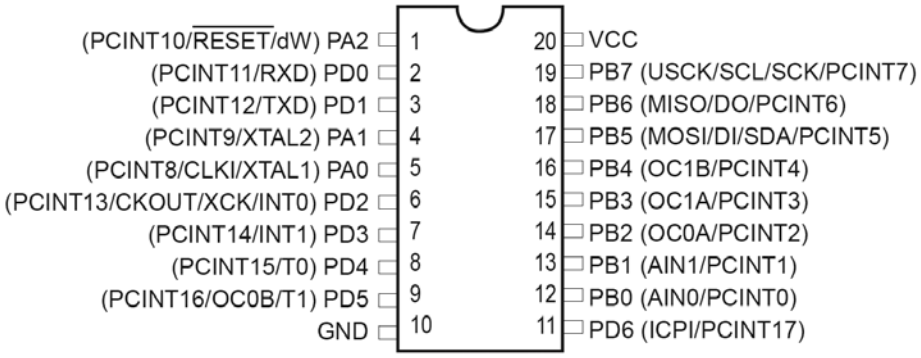
The schematic diagram, shown in Figure 6, is simple. The board contains two IDC male connectors that mate with the MiniPin II/Megapin board using flat cables. It gets its power from the host development board. You'll notice three diodes (U4, U5 & U6) in the circuit, which are needed to lower the power supply from 5V to the 3.2 Volts needed by the RFM12B module.

Pinout for the AVR microcontrollers

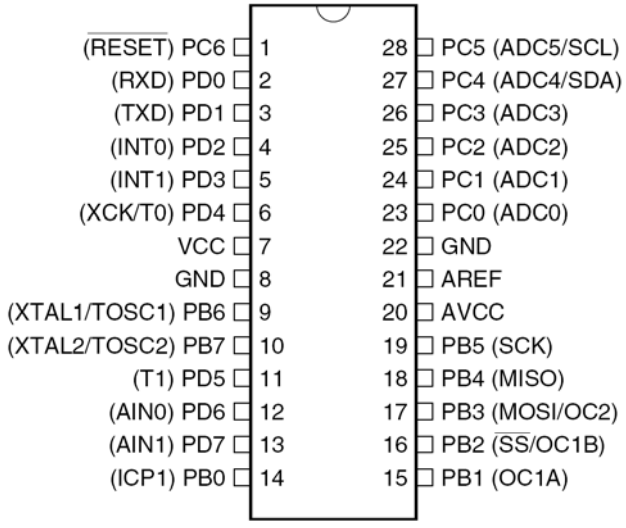


NOTE: TSSOP only for ATtiny45/V

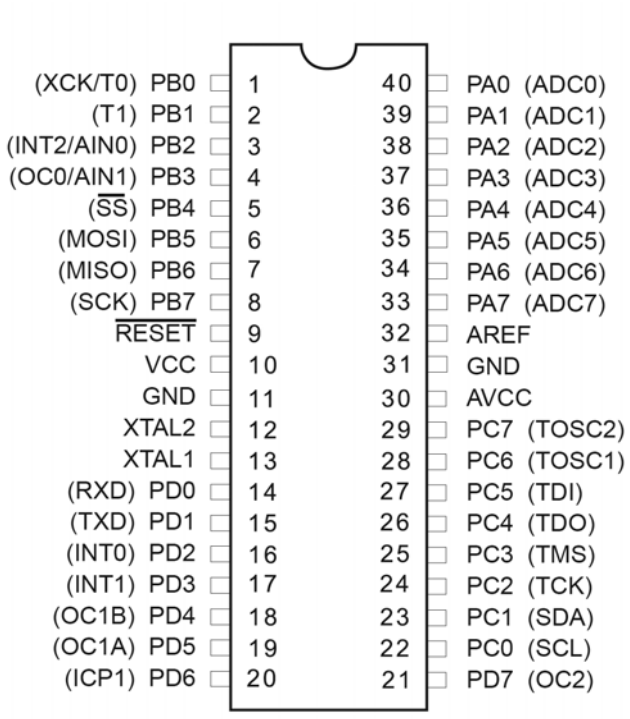
ATTINY25 \ ATTINY45 \ ATTINY85



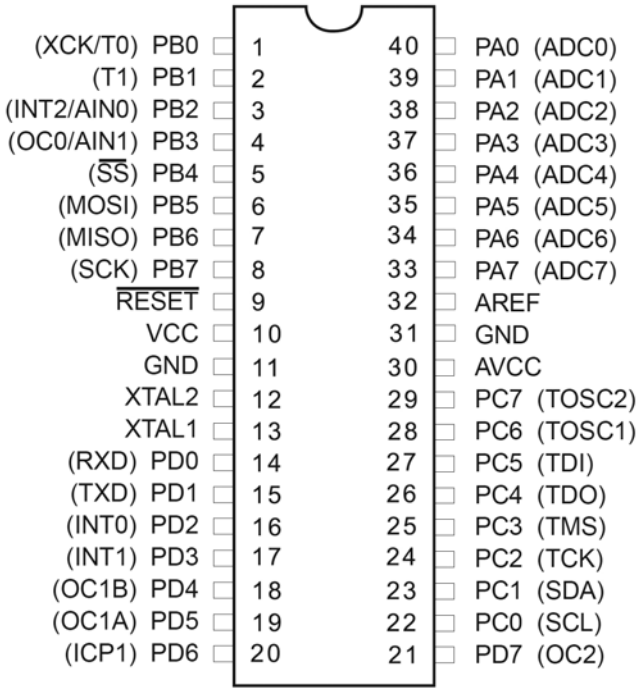
ATTINY2313 \ ATTINY4313



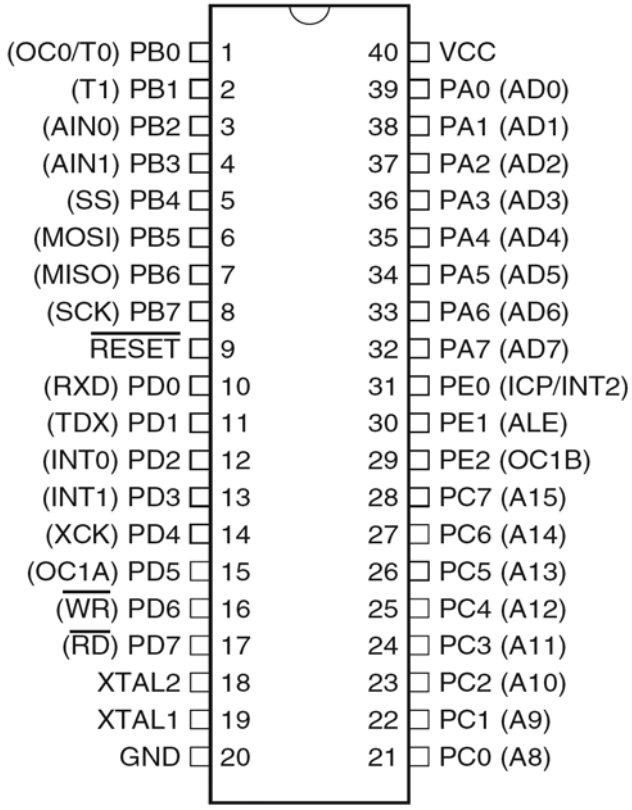
ATMEGA8



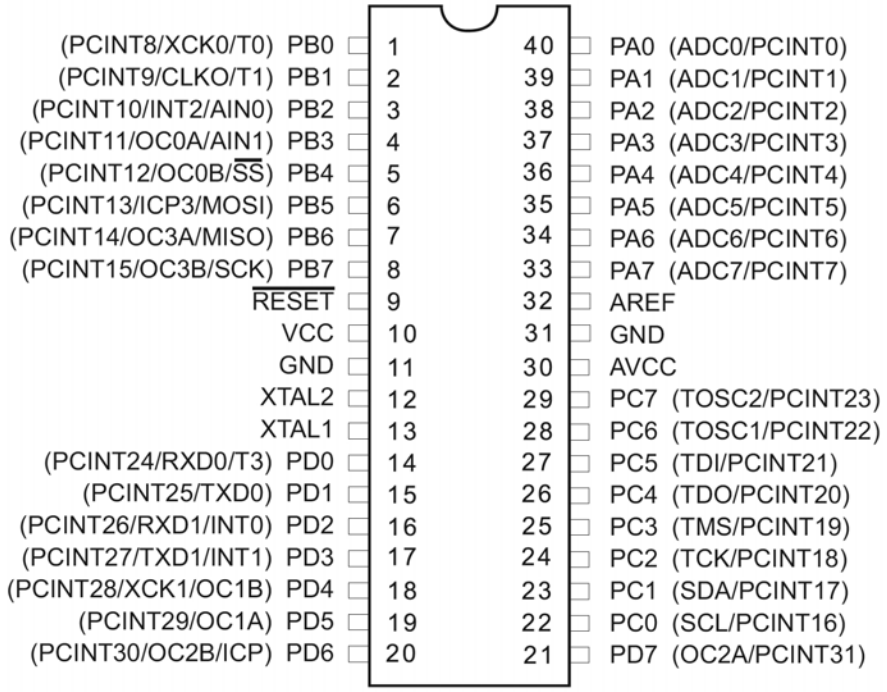
ATMEGA16



ATMEGA32



ATMEGA8515



ATMEGA164 \ATMEGA324 \ ATMEGA644 \ ATMEGA1284