AVR BASIC

GETTING STARTED GUIDE

for AVR BASIC 1.00.206
or later

(Manual Version 1.21)

Copyright Information

Information in this document is subject to change without notice and does not represent a commitment on the part of the manufacturer. The software described in this document is furnished under license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement.

It is against the law to copy the software on any medium except as specifically allowed in the license or non-disclosure agreement.

The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic, mechanical, including photocopying, recording, or information retrieval systems, for any purpose other than for the purchaser's personal use, without written permission.

EQUINOX TECHNOLOGIES
EQUINOX TECHNOLOGIES UK LIMITED
The Embedded Solutions Company

Contact Information

Equinox Technologies UK Limited

3 Atlas House, St George's Square, Bolton, England BL1 2HB

Telephone Sales ..... : **+44 (0) 1204 529000**

Fax ..... : **+44 (0) 1204 535555**

Technical Support ..... : **avrbasic@equinox-tech.com**

E-mail ..... : **sales@equinox-tech.com**

Web site ..... : **www.equinox-tech.com**

AVR BASIC is a Silicon Studio product

EQUINOX TECHNOLOGIES
EQUINOX TECHNOLOGIES UK LIMITED
The Embedded Solutions Company

Technical Support

Registration

Please fill out the 'Customer Registration Form' provided with your system and submit this to Equinox directly. Equinox will issue you a customer registration number which must be quoted when making any technical support enquiry to Equinox. Equinox can not provide technical support to unregistered users of our products.

Levels of technical support

Equinox provide a range of technical support services for the AVR BASIC Toolset. The level of support depends on the package you have purchased and also on whether you have purchased a 'Technical Support Contract' from Equinox. The level of technical support offered for each package is detailed below. A separate support contract may be purchased for AVR-BASIC-LITE and AVR-BASIC-FULL if more in-depth technical support is required.

AVR-BASIC-DEMO : No support
AVR-BASIC-LITE : Installation support only
AVR-BASIC-FULL : Installation support only

Installation Support

After ordering either AVR-BASIC-LITE or AVR-BASIC-FULL, Equinox offer you 30 days of support from date of registration with a reaction time of 5 working days. This support is only to solve installation problems. Please note that this support can only be given by e-mail.

Please send your enquiries to:
E-mail: avrbasic@equinox-tech.com
Fax: +44 (0) 1204 535555

Standard Support (Chargeable)

Standard technical support is offered only via e-mail and fax with a response time of 48 hours. Equinox will attempt to answer any question relating to the general use of the AVR-BASIC environment. We can not, however, answer questions on how to write BASIC or AVR Assembler source code or relating to user-specific hardware.
An automated e-mail service is available which will send you news of new update releases and device support enhancements.

Standard Support.....**£200**

Please send your enquiries to:
E-mail: avrbasic_support@equinox-tech.com
Fax: +44 (0) 1204 535555

EQUINOX TECHNOLOGIES
EQUINOX TECHNOLOGIES UK LIMITED
The Embedded Solutions Company

Section 1

Section 2

Introduction

The AVR BASIC Toolset contains a comprehensive suite of code development tools for the Atmel AVR RISC microcontroller family. The package includes a powerful Integrated Development Environment (IDE) which encompasses a BASIC compiler, macro assembler, editor and hex creator all within one easy-to-use Windows environment.

The AVR BASIC language allows you to write code in a high level language, while still retaining the fast execution speed of assembler. The code is compiled from a BASIC source program into optimised AVR assembly instructions ready to be programmed into an AVR microcontroller device. It is entirely possible to write your complete project using AVR BASIC without ever resorting to assembler as the compiler produces very optimised code. Even time-critical code such as interrupt service routines can be optimised at compilation stage so as to generate the most efficient code. It is also possible to generate almost all AVR instructions either directly or indirectly within AVR BASIC.

AVR BASIC is now available in three different packages from an evaluation version to the fully unrestricted version. The choice of package depends on the amount of code to be generated and the target AVR device which is to be used.

The AVR BASIC Tool set highlights:

- Compiled BASIC generates highly optimised AVR machine code
- Hybrid Language including BASIC commands plus support for many Pascal and C-type structures
- Target speeds comparable with assembler
- Not a Run-Time Interpreter; NO code overhead
- Supports AT90S1200 reduced instruction set devices
- Direct support for all AVR-specific machine code instructions within BASIC source file
- Support for 16-bit Integer and IEEE 32 bit Floating Point Maths
- Comprehensive suite of code examples available
- Breaks the cost barrier for small projects
- Ideal for educational, hobbyist and professional use

AVR BASIC

Why choose BASIC as the language for programming a microcontroller?

BASIC has been supplied as a standard accessory with almost all microcomputers. Its accessibility and ease of use have made it one of the most widely used programming languages. Originally devised in 1963, by John 18 and Thomas Kurtz of Dartmouth College (New Hampshire, U.S.A.), it was intended to provide students with an easy introduction to programming language, hence the name '**B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode'. Many scientists and engineers found BASIC attractive for developing solutions to technical problems, and the language soon became established as a tool in its own right.

High and Low Level Programming Languages

A low level language describes precisely which actions are done and specifies exactly the parts of the target system to be operated on. At the lowest programming level (closest to the object code) is assembly code, consisting of a short mnemonic code for each instruction, to identify the operation to be performed, and the part of the memory, Central Processor (CPU), or Input/Output (I/O) device on which to operate. The list of mnemonics, called the 'source file', is then processed by a computer program called an 'assembler' which translates the source into object code.

Programming in assembly code requires detailed knowledge of the instruction set and internal circuit architecture of the CPU, and the I/O devices. Every individual instruction appears as a mnemonic, and it is difficult to see structure or relationship between parts of the program. To perform the same task on different target systems requires a separate program for each. However, careful assembly programming can deliver the shortest or fastest program possible on a given system.

In a High Level Language the source code is translated to machine code by a compiler program, or by a run-time interpreter program, either of which will link CPU and I/O resources of the target system to the program. The compiler produces an executable object code file, which is loaded into the target system and run. The interpreter is itself a program which runs on the target system, and translates the source file line-by-line as required, on the target machine itself. Interpreted language programs run much more slowly than compiled programs because of the translation and file access overheads.

AVR BASIC Continued

A high-level language:

- Allows quite complex operations to be expressed as short command words or phrases, (or graphical symbols)
- Hides unnecessary system details from the programmer
- Allows the structure of the program to be expressed more clearly,
- Enables a faster programming and testing cycle
- In principle one high-level-language program is capable of being run on many different systems, if a suitable compiler is available for each system i.e. it is 'portable'.

BASIC

BASIC is a High-Level-Language, easier to learn than assembler or 'C'. It has a format and syntax already familiar to most programmers and engineers. For control applications a true compiled BASIC is desirable, to obtain maximum execution speed. Additional low - level commands may be provided, to allow the programmer to specify CPU, I/O, and memory resources, and insert assembler - level instructions for optimisation.

Introducing AVR BASIC

AVR BASIC is a hybrid language consisting of most of the familiar BASIC commands but has been significantly extended to encorporate many desirable features from the 'Pascal' and 'C' languages. It is a compiled language and so is capable of generating highly-optimised code which runs at the full speed of AVR machine code. The compiler features the ability to freely mix many AVR specific assembly language instructions with BASIC instructions in the same source file.

AVR BASIC has been specially written so as to be able to fully support all Atmel AVR microcontroller derivatives on the market today. The compiler is capable of generating code for the AVR AT90S1200(A) devices which are not currently supported by any commercially available C compiler as these devices do not have any on-chip SRAM. The recently introduced Atmel ATmega family of AVR microcontrollers is also supported.

AVR BASIC Software Overview

The diagram below shows the interaction between the AVR BASIC Integrated Development Environment (SIDE) and the device programmer supplied with this system.

AVR BASIC Tool Kit Overview

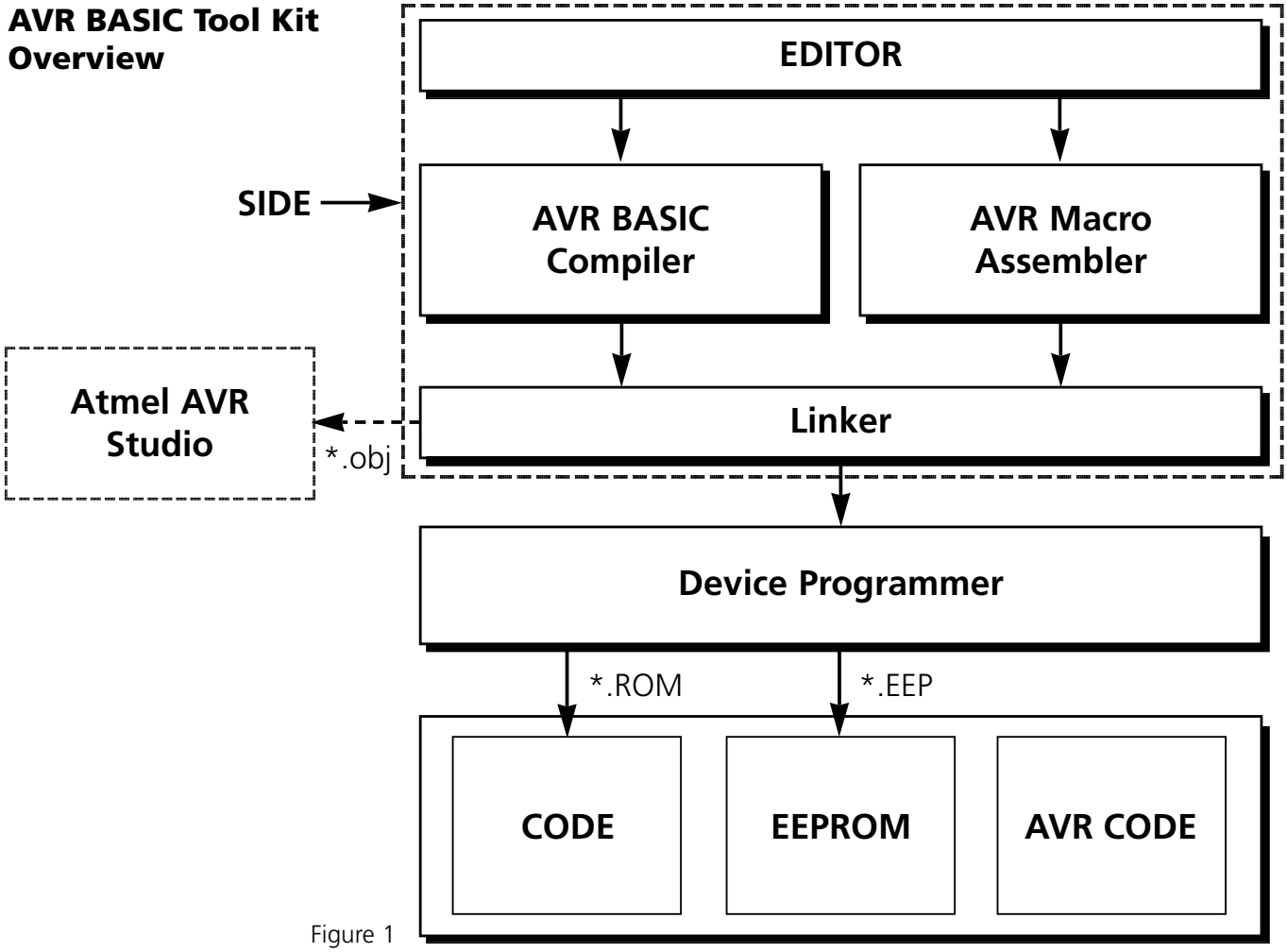


Figure 1

File types

*.obj	Object file containing symbolic and debug information
*.rom	File containing data to be programmed into the code area
*.EEP	File containing data to be programmed into the EEPROM area

IDE Overview

SIDE - Silicon Studios Integrated Development Environment

- Powerful Integrated Development Environment
- Operates under Windows 95 and Windows NT (Windows 3.11 is not supported)
- Powerful Project Manager
- Integrated editor allows multiple files to be open at the same time
- Interactive source code error correction
- Direct support for external tools e.g. device programmers

AVR BASIC Compiler

- Translates AVR BASIC source code into AVR machine code
- Generates highly optimised AVR machine code
- Target execution speeds of compiled code is comparable with AVR assembler
- Supports low-level AVR instructions directly
- Features function calling with parameter passing
- Direct code support for Interrupt Service Routines (ISR's)
- Includes powerful C-like 'switch' construct

AVR Macro Assembler

- Public Domain AVR Macro Assembler
- No direct support currently offered for this utility

Linker

- Generates an Intel HEX file suitable for downloading the code to a device programmer
- Generate an object file containing symbolic information suitable for downloading to a simulator or emulator

AVR BASIC System Summary

Which package?

AVR BASIC is now available in three different packages from an evaluation version to the fully unrestricted version. The choice of package depends on the amount of code to be generated and the target AVR device which is to be used.

AVR BASIC DEMO

This is an evaluation version of the AVR BASIC Toolset which is capable of compiling up to 64 bytes of AVR code for any AVR derivative. This version is ideal for evaluating the package as many of the examples provided in the \source directory will compile using this version.

This software can be downloaded from the 'AVR BASIC section' on the Equinox Web Site.

AVR BASIC LITE

This is a fully functional version of the AVR BASIC Toolset which is capable of compiling up to 1K bytes of AVR code. This version is ideal for evaluating the package as many of the examples provided in the \source directory will compile using this version. Floating-point libraries are not included in this version.

AVR BASIC Full Version

This is a fully functional version of the AVR BASIC Toolset which is capable of compiling up to 128K bytes of code for any AVR derivative. This version supports all the AVR instructions of the classic AVR family and will also support the Atmel ATmega derivatives. Floating-point libraries are included in this version.

Please note:

The CODE/EEPROM sizes of Atmel microcontrollers are quoted in bytes even though the processor instruction is 16-bits (2 bytes) long. For this reason, the code sizes for the different versions of AVR BASIC are also quoted in bytes.

AVR BASIC Package Summary

AVR BASIC Package	DEMO	LITE	FULL
Code Size limit (bytes)	64	1K	128K
Integrated Development Environment (IDE)	YES	YES	YES
AVR BASIC Compiler	YES	YES	YES
AVR Macro Assembler	YES	YES	YES
Linker	YES	YES	YES
HEX Translator	YES	YES	YES
Floating Point Libraries	NO	NO	YES
DEVICE SUPPORT			
AVR Device Support	ALL	AT90S1200(A)	ALL
AT90S1200 instruction set	YES	YES	YES
AT90S8515 instruction set	YES	NO	YES
ATmega support	NO	NO	YES
DOCUMENTATION			
AVR BASIC Getting Started Guide	PDF Only	YES	YES
AVR BASIC Reference Guide	NO	NO	YES
On-line HELP	YES	YES	YES
MISCELLANEOUS			
Windows Operating System	95/NT	95/NT	95/NT
Order Code	Download from our Web site	AVR-BAS-LITE	AVR-BAS-FULL
Price	FREE	$39.95	$249.95

Figure 2

Typical Project Overview

Project Based Development

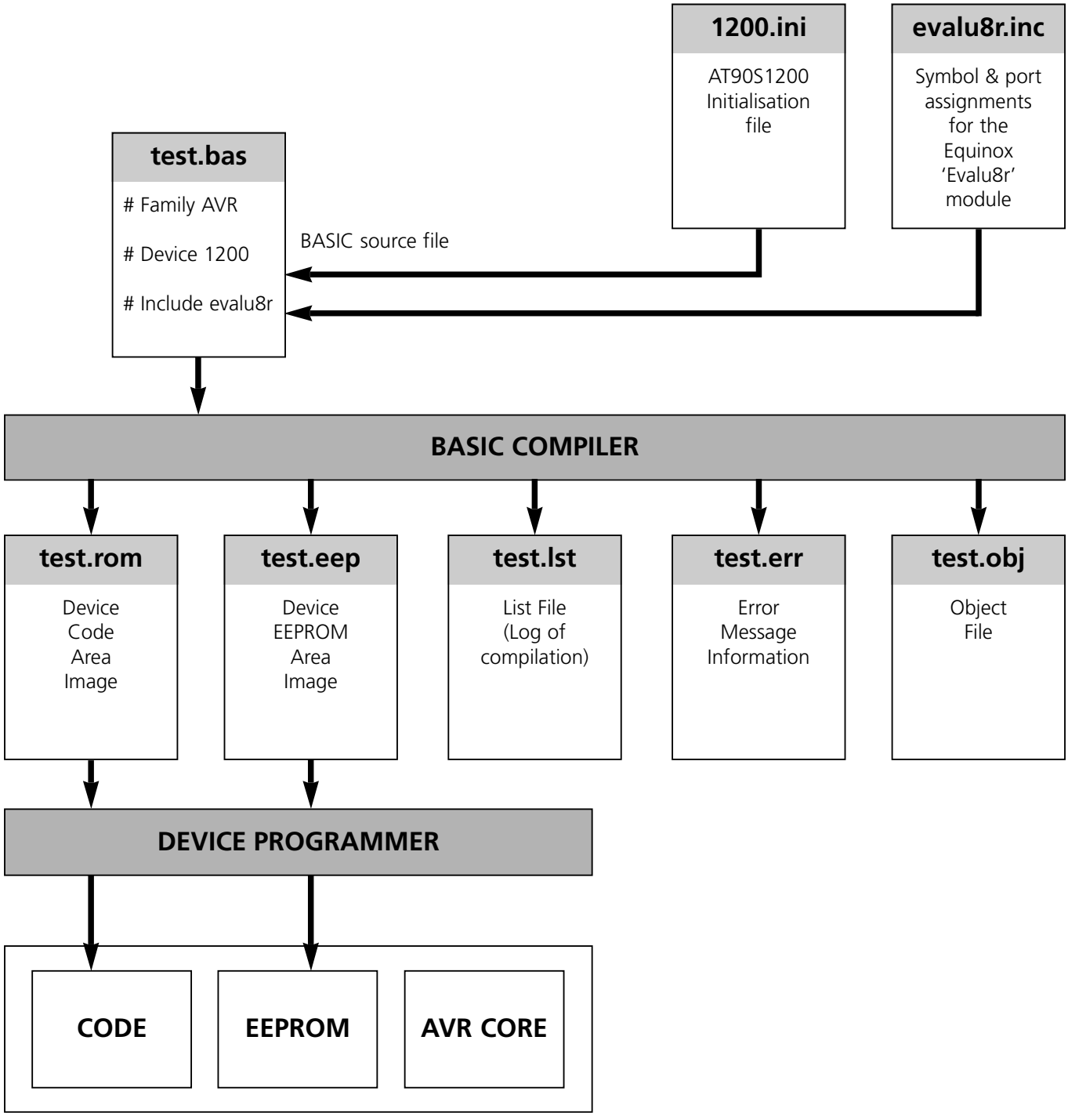
Back in the old days of DOS development tools, it was necessary to run separate command line driven programs to compile, link and then produce a hex file. This was not only time consuming, but could also lead to a lot of compilation and linking errors due to typing errors.

The AVR BASIC Integrated Development Environment (SIDE) takes care of all file management for you, so all you have to worry about is actually writing code that works! The table below shows the different files contained within a typical project.

Extension	Example	Description
.BAS	test.bas	**BASIC source file:** This file is an ASCII text file containing a program written in BASIC. The BASIC Compiler takes .bas files as its input and outputs a .lst and a .hex file.
.INI	1200.ini	**Device initialisation file:** This file contains all the device-specific information pertaining to a particular AVR microcontroller derivative. All register declarations, code size, EEPROM size, RAM size and port assignments can be found in this file.
.ROM	test.rom	**Microcontroller CODE area file:** When a BASIC or assembler language source file is compiled, the output of the compiler is both a '.rom' and a '.eep' file. The .rom file contains the actual data image which is to be programmed into the CODE area of the target device. This is currently in generic Intel HEX format.
.EEP	test.eep	**Microcontroller EEPROM area file:** When a BASIC or assembler language source file is compiled, the output of the compiler is both a '.ROM' and a '.EEP' file. The .eep file contains the actual data image which is to be programmed into the EEPROM area of the target device. This is currently in generic Intel HEX format. If the compiler generated no data for the EEPROM area, a .eep file is created containing all bytes set to 'ffh'.
.LST	test.lst	**Compiler 'List' file:** When a BASIC or assembler language source file is compiled, the compiler creates a '.lst' file which is an ASCII file which logs the compilation process and any errors found during compilation.
.OBJ	test.obj	**Object file:** When a BASIC or assembler language source file is compiled, the compiler creates an '.obj' file which contains the actual code generated by the compiler and Symbolic information for debugging. This code is not in a useable form for programming into a device and so is converted by SIDE into a HEX file.
.ERR	test.err	**Error file:** When a BASIC or assembler language source file is compiled, an error file is produced if any compilation errors were encountered. These errors are also displayed in the error window within SIDE.

Figure 3

Typical Project Overview Continued



Atmel AVR Microcontroller

Figure 4

Installation Overview

Introduction

This section explains how to set up an operating environment and how to install the software on your hard disk. Before starting the installation program, please verify that your computer system meets the minimum requirements and make a copy of the installation diskettes for backup purposes.

System Requirements

There are minimum hardware and software requirements that must be satisfied to ensure that the compiler and utilities function properly.

These are as follows:

- 100% IBM Compatible 386 or higher PC
- Windows 95 or Windows NT (Windows 3.11 and DOS are NOT supported)
- 16Mb RAM minimum
- Hard disk with minimum 6Mb free space

Default Install Directory

The installation program copies the development tools into the sub-directories show in Fig. 5.

Package	Default Install Directory
AVR BASIC-DEMO	Program files\avrbasic
AVR BASIC-LITE	Program files\avrbasic
AVR BASIC -FULL	Program files\avrbasic

Figure 5

Software Installation

AVR BASIC features a straightforward installation program which, once launched, gives full on-screen prompts at every stage. As with any software package, actually getting up and running can be the most frustrating exercise. Please consult the list of installation hints below for help.

AVR BASIC

1 Is compatible with Windows 95 & Windows NT only.

2 Is **NOT** compatible with Windows 3.11 or DOS.

3 Can be installed on a network drive and launched from a remote client machine

Installation Procedure

The following installation instructions below cover all versions of AVR BASIC:

From Windows 95 or Windows NT environment:

1 Insert 'AVR BASIC' disk into floppy drive e.g. a:

2 From the 'Start' menu, select 'Run..'

3 Type 'a:\setup.exe' or browse to the required drive and select 'setup.exe'

4 The installation program should now commence

5 Follow instructions and prompts given on screen

To install from the Equinox Web Site:

1 Go to http://www.equinox-tech.com and browse to the Software section

2 Download 'Sidexxx.exe' where 'xxx' is the version number

3 Double-click on 'Sidexxx.exe'

4 Follow instructions and prompts given on screen

Please Note: *Unless you already have a licence file the web version is 'AVR BASIC DEMO' only. Please see the Upgrading section for information on obtaining a licence file.*

The AVR BASIC Toolset is automatically added to the Windows 95 Program Menu and can be found as follows:

<Start> –> <Programs> –> <Equinox><SIDE>

Where SIDE is the AVR BASIC Integrated Development Environment.

Upgrading AVR BASIC to Lite or Full Versions

1 Install 'AVR BASIC DEMO' version on your computer

2 Obtain a 'Licence file (*.lic)' for either 'AVR BASIC LITE' or 'AVR BASIC FULL' from Equinox or your local Equinox distributor

3 With the 'Side.exe' NOT launched (ie. Do not run Side.exe), copy the 'Licence file' into the following directory: ..\side\bin ie. Into the \bin directory.

Software Installation Continued

4 Launch 'Side.exe'

5 Your copy of AVR BASIC will now be automatically updated to become either 'AVR BASIC LITE' or 'AVR BASIC FULL' depending on the licence file entered. Your Windows Registry will be automatically updated and the 'Licence file' will be deleted from the \bin directory.

Please note: *If you wish to move AVR BASIC to another PC, it will be necessary to retain a copy of your licence file separately, as the installation procedure automatically deletes it once the software is installed.*

Upgrading from a Previous Version of AVR BASIC

1 Install the new version of AVR BASIC on a machine with a copy of AVR BASIC Lite, 8K or Full already installed. The installation directory does not have to be the same as the previous installation.

2 Launch 'Side.exe'

3 Check the licence file is correct by choosing <Help><Show Licence> in the IDE.

Directories Overview

The AVR BASIC installation routine creates the following directories within the \avrbasic directory:

Directory	Description
\bin	Executables and license files
\config	IDE and device-specific config/initialisation files
\help	Windows help files
\html	HTML documents and help information
\inc	Include files
\install	Installed files
\plugins	SIDE plugins e.g. AS Assembler
\source	Example source files and applications
\temp	For IDE use only
\update	For downloaded application updates

Interfacing to Device Programmers

The AVR BASIC environment has been specially designed to support external device programmers. The compiler produces two output files which can be programmed into the target AVR microcontroller. The *.ROM should be programmed into the CODE area and the *.EEP should be programmed into the EEPROM area of the microcontroller. The default file type is Intel HEX which is compatible with most device programmers.

Example

Using the Equinox 'Meridian for Windows' programmer interface software:

1 Compile the sample AVR BASIC project called eeprom.spr found in avrbasic\source\avr\equinox\evalu8r\eeprom.spr

2 Check that the two files: 'eeprom.rom' and 'eeprom.eep' are created by the compiler. ***Please note*** *that not all examples will create an '.eep' file.*

3 Launch the Meridian software

4 Select <File><Load to buffer>
-> File load dialogue box should appear

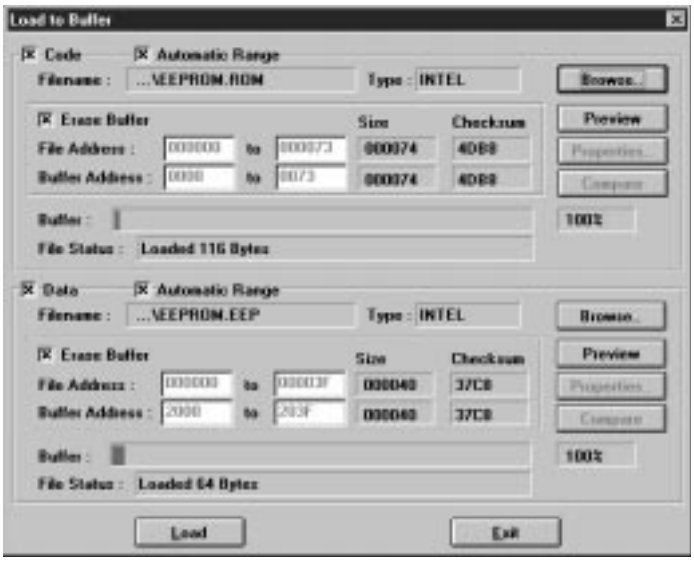
5 In the CODE area section of the Window, select <Browse> and then select the file for the CODE area i.e. 'eeprom.rom'

6 In the EEPROM area section of the Window, select <Browse> and then select the file for the EEPROM area i.e. 'eeprom.eep'

7 Click the <Load> button at the bottom of the Window
-> The two files 'eeprom.rom' and 'eeprom.eep' are now loaded into the CODE and EEPROM buffers respectively.

8 Click the <Exit> button at the bottom of the Window

9 To program the device, select <Device><Auto-program>
-> The device is programmed with the contents of the CODE and EEPROM buffers.



AVR BASIC Examples

The AVR BASIC Toolset is supplied with a suite of software examples to help you get up and running quickly. All of the examples have been written for the Equinox 'Evalu8r' microcontroller evaluation module fitted with an Atmel AT90S1200 microcontroller, but can easily be adapted for other similar targets.

The following software examples are supplied with AVR BASIC:

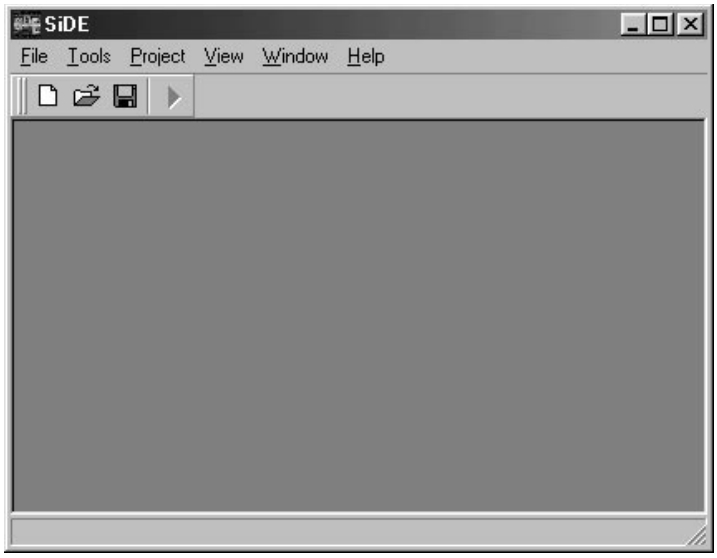
Example 1 : LED1	Turns on an LED
Example 2 : LED2	Flashes an LED
Example 3 : BUZZ1	Outputs an audible tone on a piezo sounder
Example 4 : BUTTON1	Reads and debounces a push-button switch
Example 5 : EEPROM	Demonstrates use of AVR on-chip EEPROM
Example 6 : I2CEE	External 24C16 EEPROM Driver Utility
Example 7 : ANACOMPI	A/D Utility using on-chip comparator
Example 8 : TIMER	Flashes an LED using an interrupt driven timer

Source File location: \AvrBasic\Source\Avr\Equinox\Evalu8r

Most of the above examples use the 'evalu8r.inc' include file to pre-define all the hardware on this module. This include file can be found as follows: \AvrBasic\Source\Avr\Equinox\Inc.

Trying out the example programs

1. Launch the AVR BASIC Toolset by selecting <Start><Programs><Equinox><SIDE> or by double clicking the 'side.exe' icon.

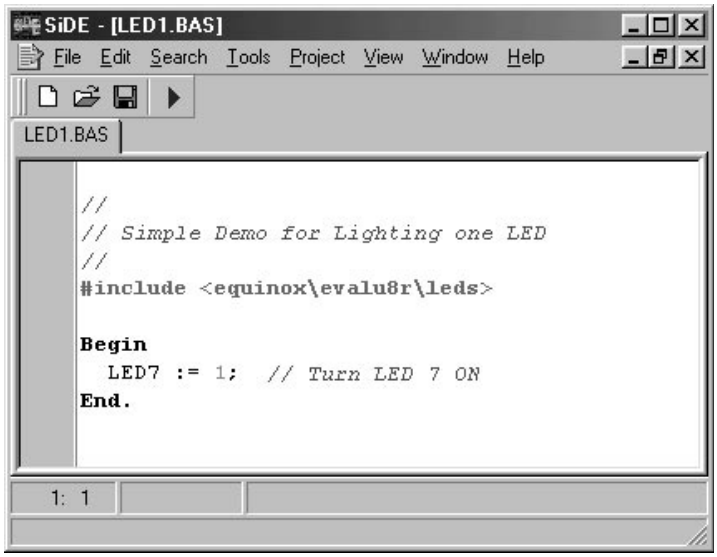


AVR BASIC Examples Continued

2. From the menu bar, select <File><Open> and then browse to your selected example

e.g. \AvrBasic\Source\Avr\Equinox\ Evalu8r\led1.spr

-> The source file for 'led1.bas' should now be displayed.



3. Select <Project><Compile> or simply press the <F9> hot key.

-> The source file is compiled and a message in the bottom line of the main window should say: "Compiled OK, xx Words of CODE".

4. To view the symbols defined during this compilation, select <View><Symbols>.

5. To view the 'list' file containing all compilation information, select <View><Listing>.

6. To program the two files (*.rom and *.eep) into a target AVR microcontroller, please refer to the instructions supplied with your programmer. (For Equinox programmers please refer to section 'Interfacing to Device Programmers')

Example 1: LED1

Source file : led1.bas
Include file : evalu8r.inc
Default processor : Atmel AT90S1200
CODE size : 2 words
EEPROM size : 0 bytes

This example simply turns on an LED connected to Port B bit 0 of an AVR device and then waits in an endless loop. The LED port is first initialised to all off (all pins set to 1) by calling the 'Init_LEDS' function which can be found in the 'evalu8r.inc' file.

```
#include <equinox\evalu8r\leds>

Begin
  LED7 := 1;  // Turn LED 7 ON
End.
```

AVR BASIC Examples Continued

Example 2: LED2

Source file : led2.bas
Include file : evalu8r.inc
Default processor : Atmel AT90S1200
CODE size : 56 words
EEPROM size : 0 bytes

This example flashes an LED connected to Port B bit 0 of any AVR device at a pre-determined rate. The LED port is first initialised to all off (all pins set to 1) by calling the 'Init_LEDS' function which can be found in the 'evalu8r.inc' file. The flash rate can be altered by changing the parameter value in the delay() function and/or by changing the processor oscillator frequency. When using the Equinox Evalu8r module, the piezo will also buzz as this is connected to the same port as the LED.

```
#include ..\inc\evalu8r
//
Begin
  Init_LEDS;              // Init LED Port
  Repeat
   LED0 := Not LED0;     // Invert LED
   Delay(20);
  Until FALSE;            // Loop forever
End.
```

Example 3: BUZZ1

Source file : buzz.bas
Include file : evalu8r.inc
Default processor : Atmel AT90S1200
CODE size : 53 words
EEPROM size : 0 bytes

This example produces a tone on a piezo sounder device connected to port B bit 7 of an AVR device. The frequency of the tone can be altered by changing the parameter value in the delay() functions and/or by changing the processor oscillator frequency.

AVR BASIC Examples Continued

Example 3 Continued

```
#include ..\inc\evalu8r
//
Begin
  Init_BUZZER;           // Init BUZZER Port
  Repeat
    Beep;                 // Make a BEEP..
    Delay(100);          // Same Delay
  Until FALSE;           // Loop.. Do it Again!
End.
```

Example 4: BUTTON1

Source file : button1.bas
Include file : evalu8r.inc
Default processor : Atmel AT90S1200
CODE size : 57 words
EEPROM size : 0 bytes

This example demonstrates a software method of reading and debouncing a push-button switch. A beep sound is made when the push button is released.

```
#include ../inc/evalu8r
//
Begin
  Init_BUZZER;           // Init BUZZER Port
  Repeat
    Wait !S1;            // Wait until PushButton S1 Pressed
    Delay(1);            // Debounce Delay
    Wait S1;             // Wait until Button Released
    Beep;                 // Make a BEEP..
  Until FALSE;           // Loop for FOREVER
End.
```

AVR BASIC Examples Continued

Example 5: EEPROM

Source file : eeprom.bas
Include file : evalu8r.inc
Default processor : Atmel AT90S1200
CODE size : 58 words
EEPROM size : 16 bytes

This example demonstrates the use of the AVR on-chip EEPROM. A series of 16 data bytes are placed in the EEPROM area at compile time. When the program runs, the data bytes are read sequentially from the EEPROM and displayed one at a time on the LED port. The speed of the LED port update can be changed by altering the parameter in the delay() function. The 'eeprom.eep' file must also be programmed into the target device in order for this program to work.

```
#include ..\inc\evalu8r
Var Index: Byte;
// Sequence to Display on LEDS
Const XX EEPROM = $55,$AA,$55,$AA, $33,$CC,$33,$CC, $66,$99,$66,$99,
$F0,$0F,$F0,$0F

Begin
  Init_LEDS;                      // Init LED Port
  Repeat
    LEDS := EEPROM[index];       // Read EEPROM and Display
    Delay(5);                     // Short Delay
    Index := Index + 1 and $0F   // make sure index stays in range [0..15]
  Until FALSE;                    // Loop forever
End.
```

Section 2
AVR BASIC Language-Quick Reference Guide

EQUINOX TECHNOLOGIES
EQUINOX TECHNOLOGIES UK LIMITED
The Embedded Solutions Company

AVR Device-specific initialisation files

Most AVR microcontrollers feature the same generic core, but have different on-chip hardware resources and also different pinouts. They can also differ in code, EEPROM and RAM sizes.

AVR BASIC allows you to write, as far as possible, in non device-specific code. All the device-specific information can be automatically loaded into a BASIC source file by including the relevant .ini file (device initialisation file) from the list below. When new devices within the AVR family are released, a new .ini will usually be made available.

11.ini	Atmel ATtiny11 initialisation file
103.ini	Atmel ATmega103 initialisation file
603.ini	Atmel ATmega603 initialisation file
1200.ini	Atmel AT90S1200(A) initialisation file
2313.ini	Atmel AT90S2313 initialisation file
2323.ini	Atmel AT90S2323 initialisation file
2333.ini	Atmel AT90S2333 initialisation file
2343.ini	Atmel AT90S2343 initialisation file
4414.ini	Atmel AT90S4414 initialisation file
8515.ini	Atmel AT90S8515 initialisation file
4433.ini	Atmel AT90S4433 initialisation file
4434.ini	Atmel AT90S4434 initialisation file
8535.ini	Atmel AT90S8535 initialisation file

The '.ini' files can be found in the \avrbasic\config\device\avr directory.

The following example source file shows how to declare that the Atmel AT90S8515 is the target device.

Example:

example.bas

```
# Family AVR
# Device 8515
```

Please Note: If the target device is not specified, the AT90S1200 is used as default.

Commands

AVR Basic provides a set of BASIC command words which allow programs to be written in a conventional format.

The following commands can currently be used in AVR Basic:

ADC Add with Carry
ADD Addition
AND Logic AND
ASR Arithmetic Shift Right
CP Compare
CPC Compare with Carry
COM Complement
DEC Decrement
EOR Exclusive OR
FOR..NEXT Loop
GOSUB Subroutine Call
GOTO Unconditional Jump
IF..THEN Conditional Branch
INC Increment
LSL Logic Shift Left
LSR Logic Shift Right
NEG Negate
NOP No Operation
OR Logic OR
REPEAT..UNTIL Loop
RETI Return from Interrupt
RETURN Return from Subroutine
ROL Rotate Left through Carry
ROR Rotate Right through Carry
SBC Subtract with Carry
SUB Subtract
SWAP Swap Nibbles or Bytes
SWITCH C-multiple branch selection
WAIT Insert time delay
XOR Exclusive OR

REPEAT...UNTIL, and the WAIT command each allow a loop to run until a test condition is satisfied - usually required for I/O operations.

The 'C' style SWITCH command allows multiple conditional decisions to be implemented efficiently.

In addition, a set of assembly code style instructions, for logical and arithmetic operations, permits efficient use of the microcontroller - usually one command generates one assembler instruction. Any AVR instruction can be generated from BASIC.

Program Layout

AVR Basic supports two different layout styles. The first resembles other microcomputer BASIC language variants, appearing as a list of statements, typically one per line, supporting conditional and unconditional jumps to labels (IF -THEN, GOTO,). Source file line numbers are not explicitly used, but are generated by the compiler for code debugging, in the *.lst file (see section 1.4, IDE Overview).

Variant #1 (Basic Style)

```
#FAMILY AVR                //                      [0]
#DEVICE 1200
#INCLUDE ..\inc\evalu8r
//   The REM comment is not supported. Use // for comments
//
//   Variable and Constant Definitions
//
Var  i,j:   Byte           //Declare two byte variables [5]
Var  n[5]: Byte            //  and a subscripted variable[6]
//
//** The first executable instruction will be   **[8]
//**        compiled at location 0x0000         **

i = 0                              // LET i = 0  - assignment.[11]
j = PINB                   // Get Port B input [12]
If j > 3 Then LabelA               //                   [13]
i = j + 1
LabelA:
Repeat Until FALSE       //Loop Forever         [16]
// **           end of program              **
```

Notes:

[**0**] Pre - processor directives, beginning with # in column 1, allow the compiler to load processor specific parameters, identifiers and common definitions. The default family is AVR and default device is AT90S1200.

[**5**] All variables must be declared, and type specified, (compare with DIM in other BASIC dialects).
Var <name1,name2...>: <type>

[**11**] As in other dialects, LET may be omitted from the assignment statement.

[**12**] Input of data from PORT B is handled by assigning to a variable the value of pre-defined I/O variable PINB I/O variables can also be used directly in conditional statements.

[**13**] Labels can be used in IF-THEN, GOTO, GOSUB

[**16**] The Loop Forever statement might be used this way during debugging, to freeze the program after a single pass. Programs do not automatically terminate. The programmer must provide a final statement which ensures that some part of program repeats endlessly - otherwise execution will continue in the unprogrammed ROM following the last valid instruction, with unpredictable consequences.

Program Layout Continued

Variant #2 (Pascal / C Style)

The second style resembles the layout of procedural languages commonly used for microcontroller programming. Such as Pascal, and the 'C' language. Procedures and functions to be called from main program body are defined at the head of the program body.

```
#FAMILY AVR
#DEVICE 1200
#INCLUDE ..\inc\evalu8r
//
//   Variable and Constant Definitions
//
Var  i,j:   Byte;         //Declare two byte variables  [4]
Var  n[5]: Byte   ;       //  and a subscripted variable
//
//**       The first executable instruction will be       **
//**       a jump to address of main program body         **

Procedure MyProc;         // Declare a procedure
Begin                     //                                  [11]
  i := j+1;
End;              //of procedure

Begin
  i:= 0;                          // LET i = 0  - assignment.
  j:= PINB;                       // Get Port B input
  If j <= 3 Then                  //
    MyProc;
  End;                            //
Repeat Until FALSE;               // Loop Forever        [21]
End.      //  **          end of program        **
```

[**4**] Each program statement may be terminated with a semicolon, and the end of main program body with End. (If used to terminate program, the period . is required).

[**11**] Each procedure or function is enclosed by Begin and End.

[**21**] End is also required to terminate a list of executable clauses within If - Then - Else statements.

The above layout is preferred in all cases where Procedures and Interrupts are used.

Program Layout Continued

Comments

Comments are used to add a more descriptive meaning to a line of code. This helps somebody who does not understand the BASIC language to follow the functionality of the program. Comments in AVR BASIC must be preceded by two forward slashes (//) and continue to the end of the line. The use of comments can be seen in the examples in this section.

Example:

```
I:= 0;       //Initialise I to zero
```

Constants

Constants fall in to four categories: Hexadecimal, Binary, Decimal and ASCII.
The default is Decimal thus anything starting with a 0..9 is interpreted as a Decimal constant. Prefixes % and $ are used for Binary and Hexadecimal constants respectively. To declare an ASCII constant enclose it in quotes (“).

Declaration examples:

```
50              Decimal
$50             Hexadecimal
%01101110       Binary
“z”             ASCII “z” from ASCII lookup tables = 122
“World”         ASCII “W”, “o”, “r”, “l”, “d”
```

EEPROM Constants

AVR BASIC supports placing of data into the EEPROM area of an AVR microcontroller. This data can either be a constant which is placed in the EEPROM at compile time or a dynamic write which occurs when the code is actually executed.

i) EEPROM Constants

Various Data types can be placed in internal EEPROM at Compile time. This data will be inserted into the ‘.eep’ file ready for programming into the target AVR device.

Example:

```
    Const message EEPROM = “AVR Basic”, 0
    Const msg2 EEPROM = “Hello”, 0
```

EEPROM Space is allocated from the bottom (low address) by default. Constants and Variables created at absolute addresses do not change the EEPROM Allocation Pointer and may overwrite existing data.

ii) EEPROM as Array

It is easy to Read/Write the internal Data EEPROM, by simply accessing elements of a pre-defined Array in EEPROM.

Constants Continued

Example:

```
R0         := EEPROM[0] // Read
EEPROM[R0] := “A” // Write
```

Notes: EEPROM Writing is automatically enabled and disabled as required.

iii) EEPROM as Object

Properties:

- EEPROM.Addr - EEAR EEPROM Address Register
- EEPROM.Data - EEDR EEPROM Data Register

iv) Methods:

- EEPROM.Read - Reads from EEPROM
- EEPROM.Write - Writes to EEPROM

Examples:

```
EEPROM.Addr := 5            // Address to 5
EEPROM.Data := “A”          // Set Data to “A”
EEPROM.Write                // Write to EEPROM

EEPROM.Addr := 0            // Address to 0
EEPROM.Read                 // Read EEPROM
R0 := EEPROM.Data           // R0 := Data Read
```

Notes: It is OK to use AVR I/O Register Variables EEAR, EEDR, directly to perform EEPROM low level Read/Write Functions. Using EEPROM Object makes the code compatible for non-AVR Targets.

Labels

Labels fall in to two categories, Address labels and Value labels. Address labels are used to mark sections of program within your program and end in a colon (:). These can be up to 32 characters long. Labels can not start with a number or be the same as a reserved word or variable. Labels make it possible to jump or go to areas of code without specifying an actual address.

Value labels are declared using the 'const' directive and are used to make your program more readable and allow for good programming practice by defining numbers as constants. Value labels can also be used to reference variables that have already been declared.

Example:

```
#include <equinox\evalu8r\leds>.

const min = 1;          // Define constant value label
const max = 5;          // Define constant value label

var tally:byte;         // Define tally as byte variable

Begin
jump:   for tally = min to max
        LED0 := Not LED0;     //Toggle LED pin 5 times
        next
        goto jump              // Move the program pointer to the for loop
statement
End.
```

Variables

The compiler supports Bit, Byte and Word variables.
A Bit is a single Bit (1 or 0) Boolean, Byte is 8 bits and a word is 16 bits. All AVR Resources (Registers, I/O Ports, internal EEPROM and SRAM) can be accessed as variables. Variables are declared using the Var directive. Below is a summary table of different variable types.

Variable type	Type	Possible values	Syntax examples
Bit	Boolean	0 or 1	var T : bit;
Byte	8-bit byte	0 to 255 decimal	var S : byte;
Word	16-bit word	0 to 65535 decimal	var x : word;
EEPROM	8-bit byte	0 to 255 decimal	var e1 : EEPROM;

Figure 6

Pre-defined Variables

Variables that are pre-defined for system use can not be redefined for any other use or used as a label.

e.g. AT90S1200

Variables mapped to Register Space

- **R0..R15** Low Bank Register Variables
- **R16..R31** High Bank Register Variables
- **W0,W2..W30** - Word Variables (Low, High Bank aligned at Word Boundaries)
- **WREG** Compiler Working Register (R31 by Default)
- **WBIT** Compiler Working Bit Storage (SREG.6)

Variables Continued

Variables mapped to I/O Ports

- **ACSR** - Analog Comparator Control and Status
- **DDRB** - PORT B Data Direction Register
- **DDRD** - PORT D Data Direction Register
- **EEAR** - EEPROM Address Register
- **EECR** - EEPROM Control Register
- **EEDR** - EEPROM Data Register
- **GIMSK** - Global Interrupt Mask Register
- **MCUCR** - CPU Control Register
- **PINB** - PORT B Input Pins (Read Only)
- **PIND** - PORT D Input Pins (Read Only)
- **PORTB** - PORT B Output Latches (Read/Write)
- **PORTD** - PORT D Output Latches (Read/Write)
- **SREG** - Processor Status Register
- **TCNT0** - Timer/Counter 0
- **TCCR0** - Timer/Counter Control Register
- **TIFR** - Timer/Counter Interrupt Flag Register
- **TIMSK** - Timer/Counter Interrupt Mask Register
- **WDTCR** - Watchdog Control Register
- **SREG** - Status Register

All I/O Variables have the name as in the relevant Atmel AVR Datasheet. Please refer to AVR Documentation for a complete description of all I/O registers.

Reserved Words

The following words are also reserved for compiler use only:

ADD	**DOWNTO**	**NOT**	**STEP**
ADDR	**EEPROM**	**OR**	**SYMBOL**
ADC	**ELSE**	**POP**	**SWITCH**
AND	**END**	**PROCEDURE**	**THEN**
ASM	**EXIT**	**PROGRAM**	**TRUE**
ARRAY	**FALSE**	**POINTER**	**TO**
ASSEMBLER	**FOR**	**PUSH**	**UNIT**
BEGIN	**FUNCTION**	**RAM**	**UNTIL**
BIT	**GOSUB**	**REGISTER**	**USE**
BOOLEAN	**GOTO**	**REPEAT**	**USES**
BREAK	**HIGH**	**RETURN**	**VAR**
BYTE	**IF**	**ROM**	**VARIABLE**
CASE	**INTERRUPT**	**SHL**	**WAIT**
COM	**IN**	**SHR**	**WHILE**
CONST	**LOW**	**SKIP**	**WORD**
DEFAULT	**NEXT**	**SUB**	**XOR**
DO	**NOP**	**SBC**	

Notes: Not all of these Reserved Words are currently supported or recognised.

Reserved Words can not be used as identifiers.

Expression Syntax

Expressions are evaluated from the left to right. Brackets can be used in Constant Expressions.

Valid operators

- + Add
- - Subtract
- * Multiply
- / Divide
- and Logic AND
- or Logic OR
- xor Exclusive OR
- << Shift Left
- >> Shift Right

Examples

```
1 + (9 - 5)
MyConst and %00001111
1 + 9 * 5 = 50
9 * 5 + 1 = 46
```

Note that there is no operator precedence. Different orders of operations may yield different results.

Logic Operators

Logic operators are used with decision based commands. The following operators for example can be used with the 'if then' expression:

- Bit Variables are used as 'Booleans'
- Not or ! can be used for inverse test of Bit Variables
- = Equal
- <> Not Equal
- > Greater
- < Less
- >= Greater or Equal
- <= Less or Equal

>, <, <=, >= are not available for Bit Variables.

Statements

Simple Statements may be separated by the Carriage Return character - i.e. one statement per line, or by the semicolon ;

If multiple statements appear on a single line then they each must be separated from the next by a semicolon.

Examples

```
PORTB := $FF
```

Single statement on a line, no semicolon required. (Semicolon is optional)

```
PORTD := $FF ; PORTB := 00 ;
```

Two statements on a line ; the first semicolon is required, the last is optional.

Compound statements i.e. those which can include one or more simple statements
These require a separator between each part of the command structure and between each enclosed simple statement.

Examples

```
If  j < 3 Then
  PORTB := $FE ;
  j := 0 ;
Else
  j := j+1 ;
End;
If !PINB.1 Then
  PORTD.2 :=1
End
```

Each clause of the statement should be on a separate line. A semicolon at the end of each simple statement is optional.

Functions & Procedures

Procedures and functions are used like subroutines (GOSUB), but are called from any part of the program by using the procedure name as a command. They return when completed to the address following the call instruction. As they are declared at the head of the program body they are easily located and if given meaningful identifiers their relation to the program is obvious. Use of procedures and functions encourages structured programming, and is recommended in place of GOTO <label> and GOSUB <label>.

The executable code of the procedure or function must be enclosed by "Begin" and "End".

Each can optionally take a Bit, Byte or Word type parameter, which may be used to pass data from main program body to the Procedure or Function. The formal parameter name given in the definition is used as a variable in the procedure body. Using the parameter can save one variable declaration for each procedure or function used.

Procedure Declaration

```
Procedure MyProc(Param1: Byte);
Begin
  My_word := Param1 AND $F0 *16     //use parameter
   ...             // Procedure body statements
End;
```

Function Declaration

The function operates like a procedure, but remains a value to the main program. It must therefore be declared with a Bit, Byte or Word type. A function call assigns the function value to a variable, or uses it in a conditional statement. The pseudo - variable "Result" is assigned the value to be returned.

```
Function MyFun: Bit;
Begin
  ...             // Function body code
  Result := 1; // Assign value
End;

Function Another_Fun(NewParam: Byte) : Word;
Begin
   If NewParam >9 Then  // Function body code
     PORTB:=$0A;        // Function body code
     Result := $FABC    // Assign value
   End;
End;
```

Functions & Procedures Continued

Procedure/Function call

```
// main program body
//
Myproc(newinput);
// ....      main program code
//
If MyFun Then ........// e.g. test for complex condition
//....       more main program code
//
Errorcode := Another_Fun(4);
//
```

Notes

A Byte Parameter is passed in the Working Register, WREG, (normally R31).

A Bit Parameter is passed in WBIT (the T bit of CPU Status register).

A Word Parameter is passed in R30 and R31.

Limitations of the AT90S1200 allow only single parameter to be used. Future compiler releases will support full parameter passing for other microcontroller targets.

As the AT90S1200 has only a three - level hardware stack, calling one proc/fn from within another is not recommended on this microcontroller. If interrupts are also in use, procedures, functions and subroutines must be used with care. (See also Section 2: GOSUB command).

A function result is returned in: WREG (bytes), WBIT (bits) or R30 and R31 (words), so a function called within a procedure with parameter may corrupt the parameter value.

Labels & Identifiers

Labels and Identifiers are not case-sensitive and can be maximum of 32 characters long. The first character must a letter. Labels must end with a colon but are not required to be in the first column of text.

It is recommended that long and descriptive names are used to minimise the need for comments, and to make the source code more readable.

Examples

```
Var Input_Code_Mask: Byte;        // Define a Byte Variable
Procedure MyProc_input;           // Declare a Procedure
This_is_My_First_Label:
   Label_not_at_Column_1:
        More_Labels_with_Colon:
```

Optimisation Techniques

Using "Basic Style Notation" it is not always possible to write optimised code which couples a BASIC command to a single AVR instruction. To achieve assembly compactness, special features have been added to the compiler. If required, any AVR instruction can be generated.

- **SKIP** Label is used to directly generate one Word AVR Conditional Skip Instructions. (See Section 2, IF - THEN - ELSE command).
- **GOTO ROM[W30];** emits **IJMP** (Indirect Jump on Z Register Value).
- **GOSUB ROM[W30];** emits **ICALL** (Indirect Call on Z Register Value).
- **R0 := ROM[W30];** emits **LPM** (Load Program Memory) Instruction.

Register usage

From the 32 general purpose AVR Registers (R0..R31), only High Bank Registers (R16..R31) can be directly used in instructions with immediate Constants. AVR Basic does allow Low Bank registers to be used, but in this case the code is emulated i.e. the compiler adds some instructions.

In speed-critical sections of code, ensure that all variable assignments are to other register (variable) values. If assignment to constant must be used, ensure that the variable is held in the high register-bank

AVR™ Support Products

Order code	Description
PROGRAMMING SYSTEMS	
AVR2-ST	Professional AVR Microcontroller Starter System
MPW-PLUS	Micro-Pro Professional Device Programming System
UISP-S4	Micro-ISP Series IV - Atmel Microcontroller ISP System (4.6-6.0V)
UISP-LV4	Micro-ISP Series IV LV - Low Voltage Atmel Microcontroller ISP System (3.0-6.0V)
UISP-UPG1	Micro-ISP Upgrade: Atmel ATmega programming support
ACT-UPG1	Activ8r Upgrade: Atmel ATmega programming support
ACT-UPG2	Activ8r - ATtiny library upgrade
EVALUATION/OEM MODULES	
OEM-UC-20/40	Universal 8051/AVR Microcontroller OEM Module
EVALU8R-1P	Evalu8r - Universal 8051/AVR Microcontroller Evaluation Module
PACKAGE ADAPTORS ETC.	
AD-PLCC44-A	Programming adaptor - 44-pin PLCC to DIL-40
AD-DIL40-PLCC44-A	Emulation adaptor - 44-pin PLCC on target system to 40-pin DIL
AD-SOIC20-A	Microcontroller Programming adaptor - 20-pin SOIC to 20-pin DIL
AD-SOIC8-A	Microcontroller Programming adaptor - 8-pin SOIC to 8-pin DIL
AD-8535-A	Parallel programming adaptor - Atmel AT90S8535/AT90S4434 (40-pin DIL)
AD-TQFP44-A	Programming adaptor - 44-pin TQFP to 40-pin DIL
SS-90S8515-P	ISP Socket Stealer Module fitted with Atmel AT90S8515 microcontroller (DIL)
SS-90S8515-J	ISP Socket Stealer Module fitted with Atmel AT90S8515 microcontroller (PLCC)
AVR BASIC Programming Language	
AVR-BAS-LITE	AVR BASIC LITE Version (1K bytes - AT90S1200 support only)
AVR-BAS-FULL	AVR BASIC Full Version (8K bytes - All AVR derivatives supported)
IAR AT90S Language Tools	
EWA90BAS-EE	"IAR Baseline Tool Set" - C compiler, assembler, debugger (8K code limit)
EWA90	"IAR Full AT90S Version" - C compiler, assembler, debugger (unrestricted code)
DO-BOX (Dynamically Optimised BASIC Box) + Accessories	
DOBOX-ST1	DO-BOX Starter System 1
DOBOX-DV1	DO-BOX Development System 1
DOBOX-MOD1	DO-BOX Module 1
DOBOX-PM1	DO-BOX Prototyping Module
DOBOX-AM1	DO-BOX Applications Module 1
LITERATURE	
CD-AT98	Atmel CD-ROM Databook 1998
DB-AVR-981	Atmel AVR Microcontroller Data Book (Paper format)
MAN-AVRBAS-REF	AVR BASIC Reference Guide
MAN-AVRBAS-GS	AVR BASIC Getting Started Guide
MISCELLANEOUS	
CAB-SER1	PC Serial Cable Adaptor Kit (9W-25W & 25W-9W)
CAB-PAR25MM	PC Parallel Cable (25W to 25W M/M 2M)

Atmel AVR Microcontroller Product Selection Guide

DEVICE	90S1200	90S2313	90S2323	90S2343	90S4414	90S8515
ON-CHIP MEMORY						
FLASH (Bytes)	1K	2K	2K	2K	4K	8K
EEPROM (Bytes)	64	128	128	128	256	512
SRAM (Bytes)	0	128	128	128	256	512
In-System Programmable (ISP)	YES	YES	YES	YES	YES	YES
PINS + I/O						
Package Pins	20	20	8	8	40/44	40/44
I/O Pins	15	15	3	5	32	32
Packages	20P3,20S, 20Y	20P3,20S	8P3,8S2	8P3,8S2	40P6,44J, 44A	40P6,44J, 44A
HARDWARE FEATURES						
SPI Port	NO	NO	NO	NO	YES	YES
Full Duplex Serial UART	NO	YES	YES	NO	YES	YES
Watchdog Timer	YES	YES	NO	YES	YES	YES
Timer/Counters	1	2	NO	2	2	2
PWM Channels (10-bit)	-	1	-	-	2	2
Analogue Comparator	YES	YES	NO	NO	YES	YES
ADC	NO	NO	NO	NO	NO	NO
IDLE and Power Down modes	YES	YES	YES	YES	YES	YES
Interrupts (MAX)	4	11	3	3	13	13
MISCELLANEOUS						
AVR Instructions	89	120	118	118	118	118
On-chip RC Oscillator	YES	NO	NO	YES	NO	NO
Wakeup Time	16ms	1.1ms	1 ms/16 ms	16us	1.1ms	1.1ms
Real Time Clock (RTC)	NO	NO	NO	NO	NO	NO
Max External Clock Frequency	12MHz	10MHz	10MHz	10MHz	8MHz	8MHz
Vcc Voltage Range (V)	2.7-6.0	2.7-6.0	2.7-6.0	2.7-6.0	2.7-6.0	2.7-6.0
EQUINOX SUPPORT TOOLS						
Activ8r Device Programmer	PAR+ISP	PAR+ISP	PAR+ISP	PAR+ISP	PAR+ISP	PAR+ISP
Micro-ISP Series III/IV Prog.	ISP only	ISP only	ISP only	ISP only	ISP only	ISP only
Micro-Pro Device Programmer	PAR only	PAR only	-	-	ZIF-ISP	ZIF-ISP
Evalu8r Evaluation Module	YES	YES	YES	YES	YES	YES

***Disclaimer*:** Whilst information is supplied in good faith, we are not liable for any errors or omissions. Please consult the relevant Atmel datasheet. E&OE

EQUINOX TECHNOLOGIES
EQUINOX TECHNOLOGIES UK LIMITED
The Embedded Solutions Company

90S2333	90S4433	90S4434	90S8535	MEGA603	MEGA103
2K	4K	4K	8K	64K	128K
128	256	256	512	2K	4K
128	128	256	512	4K	4K
YES	YES	YES	YES	YES	YES
28	28	40/44	40/44	64	64
20	20	32	32	32I/O + 8O + 8I	32I/O + 8O + 8I
28PDIP/SOIC	28PDIP/SOIC	40P6,44J, 44A	40P6,44J, 44A	64A	64A
YES	YES	YES	YES	YES	YES
YES	YES	YES	YES	1	1
YES	YES	YES	YES	YES	YES
2	2	2	2	3	3
1	1	TBA	TBA	2	2
YES	YES	YES	YES	YES	YES
6CH/10BIT	6CH/10BIT	8CH/10BIT	8CH/10BIT	8CH/10BIT	8CH/10BIT
YES	YES	YES	YES	YES	YES
14	14	17	17	24	24
118	118	120	120	121	121
NO	NO	NO	NO	NO	NO
TBA	TBA	TBA	1.1ms	4clks	4clks
NO	NO	NO	NO	YES	YES
8MHz	8MHz	8MHz	8MHz	6MHz	6MHz
2.7-6.0	2.7-6.0	2.7-6.0	2.7-6.0	2.7-6.0	2.7-6.0
ISP only	ISP only	ISP only	ISP only	ISP only	ISP only
ISP only	ISP only	ISP only	ISP only	ISP only	ISP only
-	-	-	-	-	
NO	NO	NO	NO	NO	NO

** Max speed depends on Vcc voltage. Frequencies and Currents listed are for Vcc = 5.0V & T = 25ºC*

Please verify correct part codes for low voltage parts before ordering.

Key

SRAM - Static RAM

ISP - In-System Programmable

I/O - Input/Output

ADC - Analogue to Digital Convertor

SPI - Serial Peripheral Interface

PWM - Pulse Width Modulation

PAR - Parallel programming mode

The Embedded Solutions Company