# NEWBIE'S GUIDE TO AVR DEVELOPMENT

AN INTRODUCTION INTENDED FOR PEOPLE WITH NO PRIOR AVR KNOWLEDGE

AVRFREAKS.NET

JULY 2002

# TABLE OF CONTENTS

# Newbie's Getting Started Guide

This page is intended for those of you that are totally new to the AVR Microcontrollers!

Starting with a new uC architecture can be quite fustrating. The most difficult task seems to be how to get the information and documentation to get the first AVR program up running. This tutorial assumes that you do not yet own any AVR devices or AVR development tools. It also assumes that you have no prior knowledge of the AVR architecture or instruction set. All you need to complete this tutorial is a computer running some flavour of the Windows operating system, and an internet connection to download documents and files.

# Preparing your PC for AVR Development

Let's make an easy start, and download the files that we will need later on. First you should download the files to have them readily available when you need them. This could take some time depending on your internet connection. Download these files to a temporary folder on your computer. ( e.g. C:\Temp ): All files are included in the files-folder with exception of AVR-studio which can be downloaded from Atmels website.

| **AVR STUDIO 4** | |
|---|---|
| http://www.atmel.com/AVR/ | This file contains the AVR Studio 4 Program. This program is a complete development suite, and contains an editor and a simulator that we will use to write our code, and then see how it will run on an AVR device |
| **Assembly Sample Code** | |
| samplecode.asm | This file contains the Assembly Sample code you will need to complete this guide. |
| **AT90S8515 Datasheet** | |
| Doc2512.pdf | This is the Datasheet for the AT90S8515 AVR Microcontroller. This is a convenient "Getting Started" device. For now you don't have to worry about the different types of AVR micros. You'll see that they are very much alike, and if you learn how to use one (eg. 8515), you will be able to use any other AVR without any problems. |
| **Instruction Set Manual** | |
| Doc0856.pdf | This is the Instruction Set Manual. This document is very useful if you want detailed information about a specific instruction. |

When you have downloaded the files, it is time to install the software you need.

## Step 2. Installing AVR Studio 4

AVR Studio is also available in a version 3. We will use AVR Studio 4 since this is the version that will eventually replace version 3.

### Important note for people using Windows NT/2000/XP

You must be logged in with administrator rights to be able to successfully install AVR Studio. The reason is that these Windows systems have restrictions regarding who can install new device drivers!

### Installation:

1. Double click on the AVRSTUDIO.EXE file you downloaded. This file is a self extracting file, and will ask where you want to extract the files. The default path points to your "default" temp folder, and could be quite well "hidden" on your hard disk, so make sure to read and remember this path, or enter

a new path to where you want the files placed (e.g. c:\temp )

2. Once all the files are extracted, open the temp folder, and double click on the SETUP.EXE file. Just follow the installation, and use the default install path. NB: You can use another path, but this tutorial assumes that you install it to the default path.

That's it. Now you have installed all the software you'll need to write code and run programs for all available AVR devices! Keep the Datasheet and Instruction set Manual in a place you remember.

# Basic AVR Knowledge

The AVR Microcontroller family is a modern architecture, with all the bells and whistles associated with such. When you get the hang of the basic concepts the fun of exploring all these features begins. For now we will stick with the "Bare Bone" AVR basics.

## The 3 different Flavors of AVR

The AVR microcontrollers are divided into three groups:

1. tinyAVR
2. AVR (Classic AVR)
3. megaAVR

The difference between these devices lies in the available features. The ***tinyAVR*** uC are usually devices with lower pin-count or reduced feature set compared to the ***megaAVR's*** . All AVR devices have the same instruction set and memory organization, so migrating from one device to another AVR is easy.

Some AVR's contain SRAM, EEPROM, External SRAM interface, Analog to Digital Converters, Hardware Multiplier, UART, USART and the list goes on. If you take a ***tinyAVR*** and a ***megaAVR*** and strip off all the peripheral modules mentioned above, you will be left with the AVR Core. This Core is the same for all AVR devices. (Think of Hamburgers: They all contain the same slab of meat, the difference is the additional styling in the form of tripled-cheese and pickles :)

## Selecting the "correct" AVR

The morale is that the tinyAVR, AVR (Classic AVR) and megaAVR does not really reflect performance, but is more an indication of the "complexity" of the device: Lot's of features = megaAVR, reduced feature set = tinyAVR . The "AVR (Classic AVR)" is somewhere in between these, and the distinctions between these groups are becoming more and more vague.

So for your project you should select an AVR that only includes the features that you need if you are on a strict budget. If you run your own budget you should of course go for the biggest AVR possible, since eh... because!

## Learning to write code on the AVR

Learning new stuff is fun, but can be a bit frustrating. Although it is fully possible to learn the AVR by only reading the datasheet this is a complicated and time-consuming approach. We will take the quick and easy approach, which is:

1. Find some pre-written, working code
2. Understand how this code works

3. Modify it to suite our needs

The device we will use is the AT90S8515 which is an AVR with a good blend of peripherals. Take a few minutes to browse through the Datasheet.

## Learning to use the AVR Datasheets

It is easy to get scared when looking at the AVR Datasheets. E.g. the ATmega128(L) datasheet is almost 350 pages long, and reading it start to finish - and remembering the contents, is quite a task. Luckily you are not supposed to do that, either. The datasheets are complete technical documents that you should use as a reference when you are in doubt how a given peripheral or feature works. When you open an AVR Datasheet you will discover that it can be divided into these groups:

1. First Page Containing Key information and Feature List
2. Architectural Overview
3. Peripheral Descriptions
4. Memory Programming
5. Characteristics
6. Register Summary
7. Instruction Set Summary
8. Packaging Information

This is quite convenient. When you are familiar with how to use the AT90S8515 Datasheet, migrating to another Datasheet should be a breeze.

After completing this tutorial you should take some time and read through the Architectural Overview sections of the datasheets (At the beginning of the Datasheets). These sections contain a lot of useful information about AVR memories, Addressing modes and other useful information.

Another useful page to look at is the Instruction Set Summary. This is a nice reference when you start developing code on your own. If you want in-depth information about an instruction, simply look it up in the Instruction Set Manual you previously downloaded!

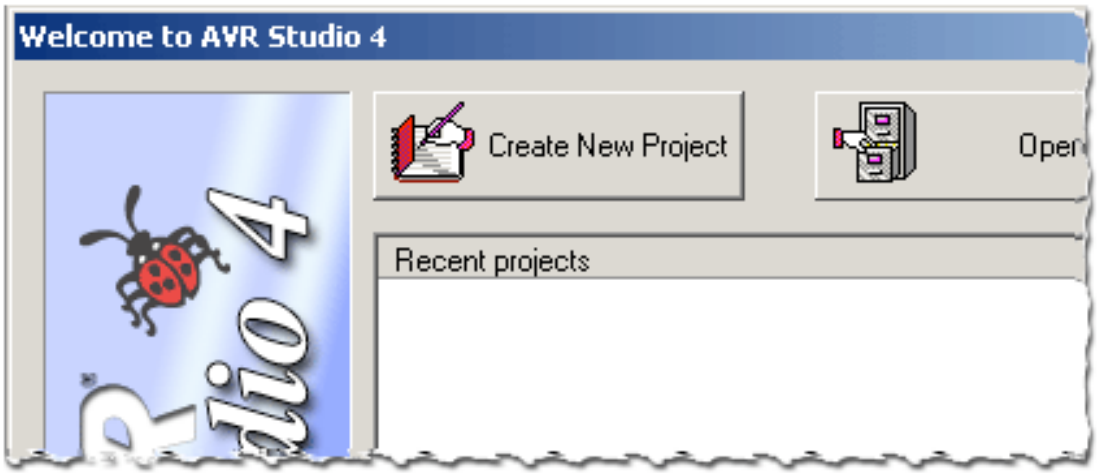
OK! You have now installed the software, you have a vague knowledge of the different types of AVRs, and know that there is a lot of information in the datasheet that you don't yet know anything about! Good, now it's time to get developing! Click "Next" to advance to the next part of this tutorial.

# Starting Up AVR Studio 4

**Note:** If you have not yet installed AVR Studio you should go to the Preparing your PC for AVR Development section of this tutorial before continuing.

## Step 1: Creating a New Project

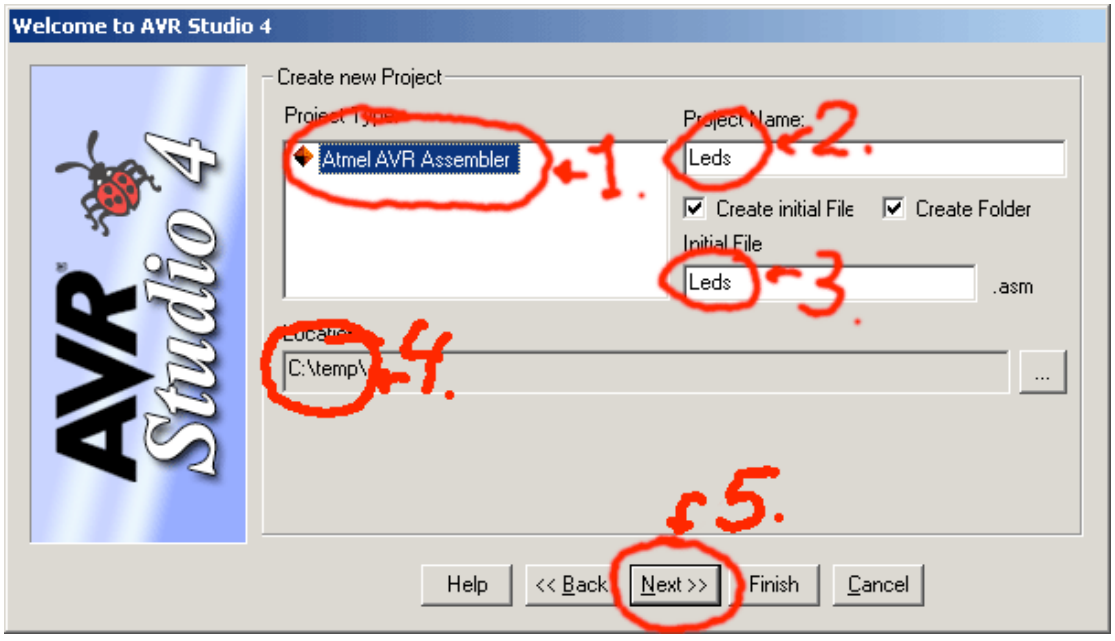
Start AVR Studio 4 by launching AVR Studio 4 located at [START] | [Programs] | [Atmel AVR Tools]. AVR Studio will start up, and you will get this dialog box.




We want to create a new Project so press the "Create New Project Button"

## Step 2: Configuring Project Settings

This step involves setting up what kind of project we want to create, and setting up filenames and location where they should be stored.



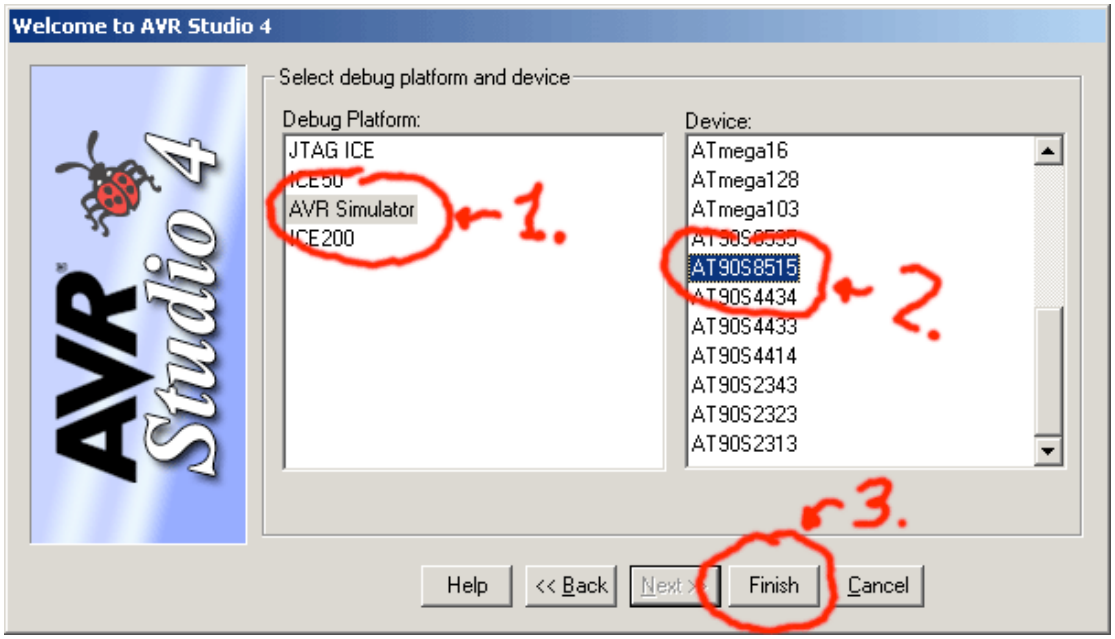

This is done in four steps:

1. Click on this to let the program know you want to create an Assembly program

2. This is the name of the project. It could be anything, but "Leds" is quite descriptive of what this program is going to do
3. Here you can specify if AVR Studio should automatically create a initial assembly file. We want to do this. The filename could be anything, but use "Leds" to be compatible with this tutorial!
4. Select the path where you want your files stored
5. Verify everything once more, and make sure both check-boxes are checked. When you are satisfied, press the "Next >>" button

## Step 3: Selecting Debug Platform

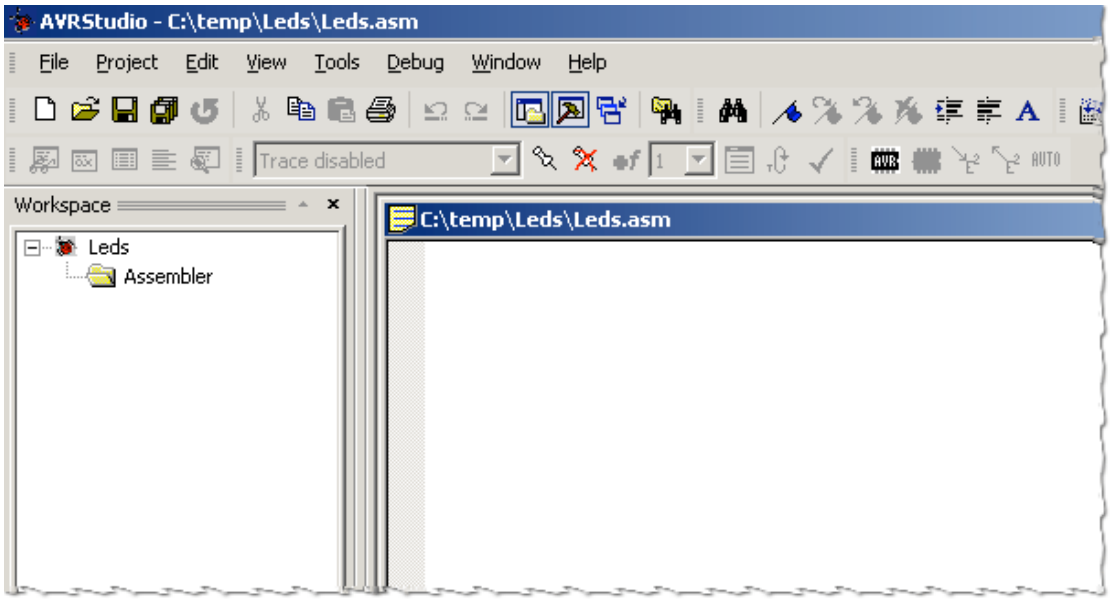
The AVR Studio 4 Software can be used as a frontend software for a wide range of debugging tools.



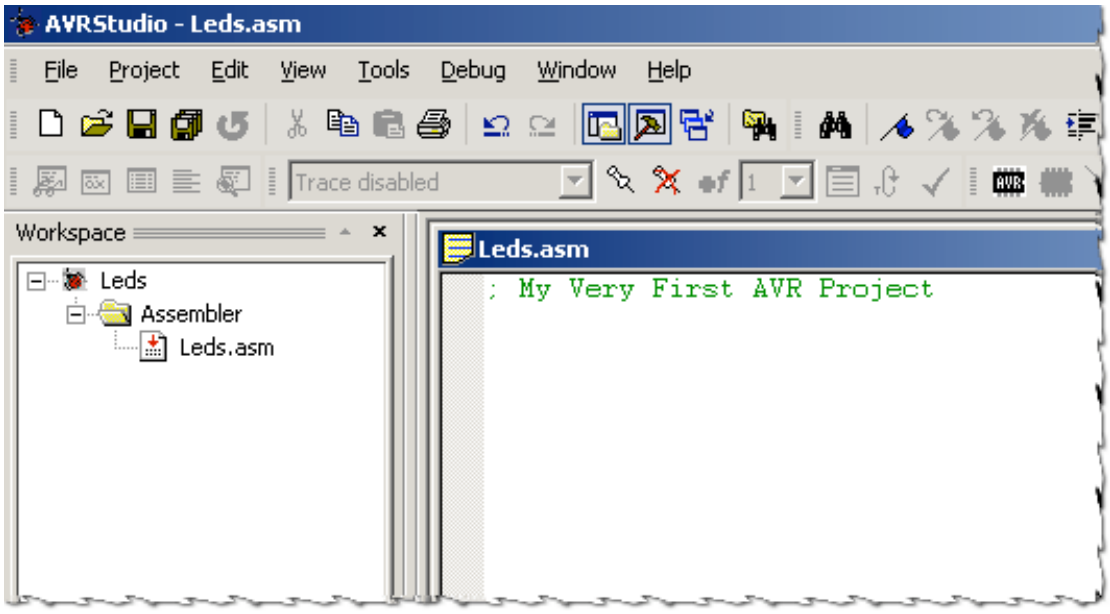

1. AVR Studio 4 supports a wide range of emulation and debugging tools. Since we have not purchased any of these yet, we will use the built in simulator functionality.
2. ..and we want to develop for the AT90S8515 device
3. Verify all settings once more, then press "Finish" to create project and go to the assembly file

## Step 4: Writing your very first line of code

AVR Studio will start and open an empty file named Leds.asm. We will take a closer look at the AVR Studio GUI in the next lesson. For now note that the Leds.asm is not listed in the "Assembler" folder in the left column. This is because the file is not saved yet. Write in this line: "; My Very First AVR Project" as shown in the figure below. The semicolon ; indicates that the rest of the line should be treated as a comment by the assembler.



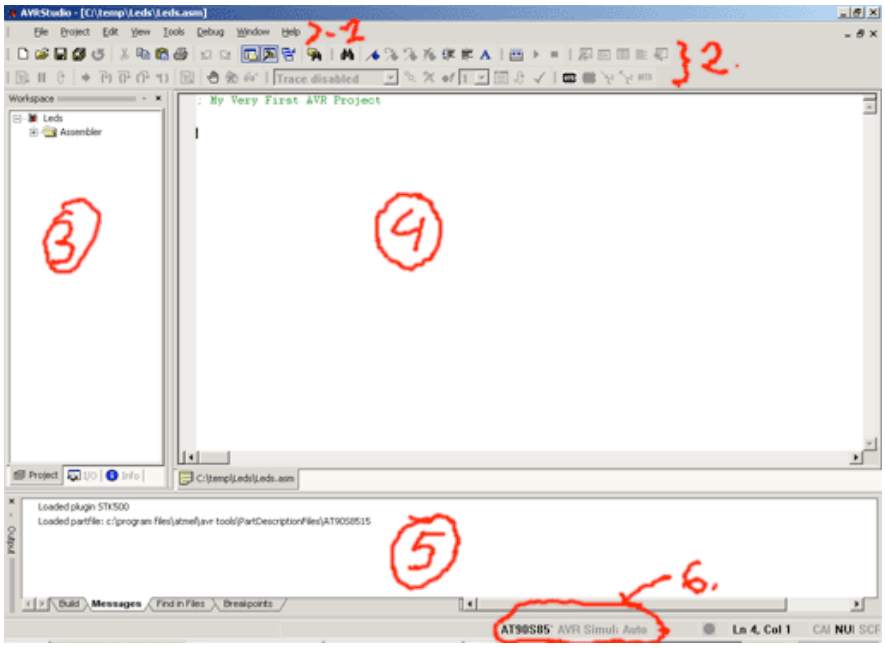

To save the line press <CTRL> - S or select [Save] on the [File] menu. The Leds.asm will now show up in the Left Column as shown below.




OK, Now that we have AVR Studio up and running, it's time to take a closer look at the AVR Studio GUI.

# AVR Studio 4 GUI

Let's take a closer look at the AVR Studio Graphical User Interface (GUI).




As you can see below, we have divided the GUI into 6 sections. AVR Studio 4 contains a help system for AVR Studio, so instead of reinventing the wheel here, I'll just explain the overall structure of AVR Studio 4 and point to where in the AVR Studio 4 On-line Help System you can find in depth information.

1. The first line here is the "Menus" Here you will find standard windows menus like save and load file, Cut & Paste, and other Studio specific menus like Emulation options and stuff.
2. The next lines are Toolbars, which are "shortcuts" to commonly used functions. These functions can be saving files, opening new views, setting breakpoints and such.
3. The Workspace contains Information about files in your Project, IO view, and Info about the selected AVR
4. This is the Editor window. Here you write your assembly code. It is also possible to integrate a C-Compiler with AVR Studio, but this is a topic for the more advanced user
5. Output Window. Status information is displayed here.
6. The System Tray displays information about which mode AVR Studio is running in. Since we are using AT90S8515 in simulator mode, this will be displayed here

## More about the GUI

To complete this bare bone guide you don't need any more knowledge of the GUI right now, but it is a good idea to take a look at the AVR Studio HTML help system. You can start this by opening [HELP] [AVR Studio User Guide] from AVR Studio, or by clicking this link (and select: Open) if you installed AVR Studio to the default directory.

When you have had your fill, we'll continue working on our first AVR Program.

# Writing your First AVR Program

At this point you should have installed the software, and started up the a new project called "Leds" You should also have the AT90S8515 Datasheet, stored somewhere you can easily find it. If you can answer "Yes" to both these questions, you are ready to continue writing some AVR Code.

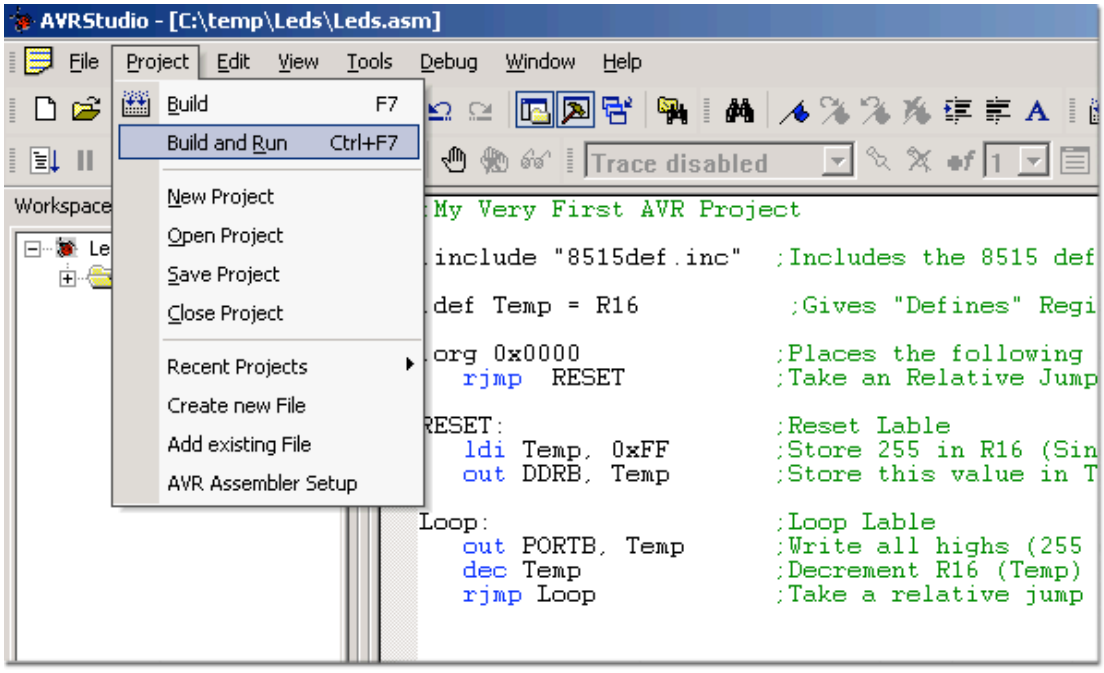
In the Editor view in AVR Studio, continue your program (which at this point only consists of the first line below) by adding the text below. (Cheaters can simply cut & paste the source below into AVR Studio...)

**Samplecode.asm**

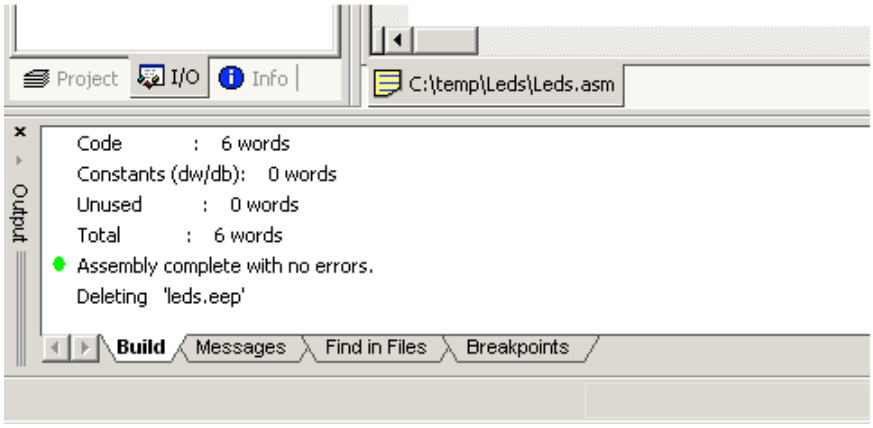
(file included in files-folder)

```
;My Very First AVR Project

.include "8515def.inc"  ;Includes the 8515 definitions file

.def Temp = R16          ;Gives "Defines" Register R16 the name
Temp

.org 0x0000             ;Places the following code from address
0x0000
   rjmp  RESET          ;Take a Relative Jump to the RESET Label

RESET:                  ;Reset Label
   ldi Temp, 0xFF       ;Store 255 in R16 (Since we have defined
R16 = Temp)
   out DDRB, Temp       ;Store this value in The PORTB Data
direction Register

Loop:                   ;Loop Label
   out PORTB, Temp      ;Write all highs (255 decimal) to PORTB
   dec Temp             ;Decrement R16 (Temp)
   rjmp Loop            ;Take a relative jump to the Loop label
```

Note that the source code changes color when written in the editor window. This is known as syntax highlighting and is very useful make the code more readable. Once the Source code is entered, press CTRL + F7 or select [Build and Run] from the [Project] Menu.



In the output view (at the bottom left of the screen) you should get the following output indicating that the Project compiled correctly without any errors! From this output window, we can also see that our program consists of 6 words of code (12 bytes).




**Congratulations!!** You have now successfully written your first AVR program, and we will now take a closer look at what it does!

**Note:** If your program does not compile, check your assembly file for typing errors.If you have placed the include files (8515def.inc) in a different folder than the default, you may have to enter the complete path to the file in the .include "c:\complete path\8515def.inc" statement.

When it compiles we will continue explaining and then debugging the code.

# Understanding the Source Code

OK so the code compiled without errors. That's great, but let us take a moment to see what this program does, and maybe get a feeling how we should simulate the code to verify that it actually performs the way we intended. This is the complete source code:

**Sample Code**

```
;My Very First AVR Project

.include "8515def.inc"  ;Includes the 8515 definitions file

.def Temp = R16          ;Gives "Defines" Register R16 the name Temp

.org 0x0000             ;Places the following code from address 0x0000
   rjmp  RESET          ;Take a Relative Jump to the RESET Label

RESET:                  ;Reset Label
   ldi Temp, 0xFF       ;Store 255 in R16 (Since we have defined R16 = Temp)
   out DDRB, Temp       ;Store this value in The PORTB Data direction Register

Loop:                   ;Loop Label
   out PORTB, Temp      ;Write all highs (255 decimal) to PORTB
   dec Temp             ;Decrement R16 (Temp)
   rjmp Loop            ;Take a relative jump to the Loop label
```

Now let's take a line-by-line look at what's going on in this code.

| | |
|---|---|
| **;My Very First AVR Project** | Lines beginning with " ; " (semicolon) are comments. Comments can be added to any line of code. If comments are written to span multiple lines, each of these lines much begin with a semicolon |
| **.include "8515def.inc"** | Different AVR devices have e.g. PORTB placed on different location in IO memory. These .inc files maps MNEMONICS codes to physical addresses. This allows you for example to use the label PORTB instead of remembering the physical location in IO memory (0x18 for AT90S8515) |
| **.def Temp = R16** | The .def (Define) allow you to create easy to remember labels (e.g. Temp) instead of using the default register Name (e.g. R16). This is especially useful in projects where you are working with a lot of variables stored in the general purpose Registers (The Datasheet gives a good explanation on the General Purpose Registers!) |
| **.org 0x0000** | This is a directive to the assembler that instructs it to place the following code at location 0x0000 in Flash memory. We want to do this so that the following RJMP instruction is placed in location 0 (first location of FLASH). The reason is that this location is the Reset Vector, the location from where the program execution starts after a reset, power-on or Watchdog reset event. There are a also other interrupt vectors here, but our application does not use interrupts, so we can use this space for regular code! |
| **rjmp RESET** | Since the previous command was the .org 0x0000, this Relative Jump (RJMP) instruction is placed at location 0 in Flash memory, and is the first instruction to be executed. If you look at the Instruction Set Summary in the Datasheet, you will see that the AT90S8515 do not have a JMP instruction. It only has the RJMP instruction! The reason is that we do not need the full JMP instruction. If you compare the JMP and the RJMP you will see that the JMP instruction has longer range, but requires an additional instruction word, making it slower and bigger. RJMP can reach the entire Flash array of the AT90S8515, so the JMP instruction is not needed, thus not implemented. |
| **RESET:** | This is a label. You can place these where you want in the code, and use the different branch instructions to jump to this location. This is quite neat, since the assembler itself will calculate the correct address where the label is. |
| **ldi Temp, 0xFF** | Ah.. finally a decent instruction to look at: Load Immediate (LDI). This instruction loads an Immediate value, and writes it to the Register given. Since we have defined the R16 register to be called "Temp", this instruction will write the hex value 0xff (255 decimal) to register R16. |
| **out DDRB, Temp** | Why aren't we just writing "ldi DDRB, Temp"? A good question, and one that require that we take a look in the Instruction Set Manual. Look up the "LDI" and "OUT" instructions. You will find that LDI has syntax : "LDI Rd, K" which means that it can only be used with General Purpose Registers R16 to R31. Looking at "OUT" instruction we see that the syntax is "OUT A, Rr" Which means that the content that is going to be written by the OUT instruction |

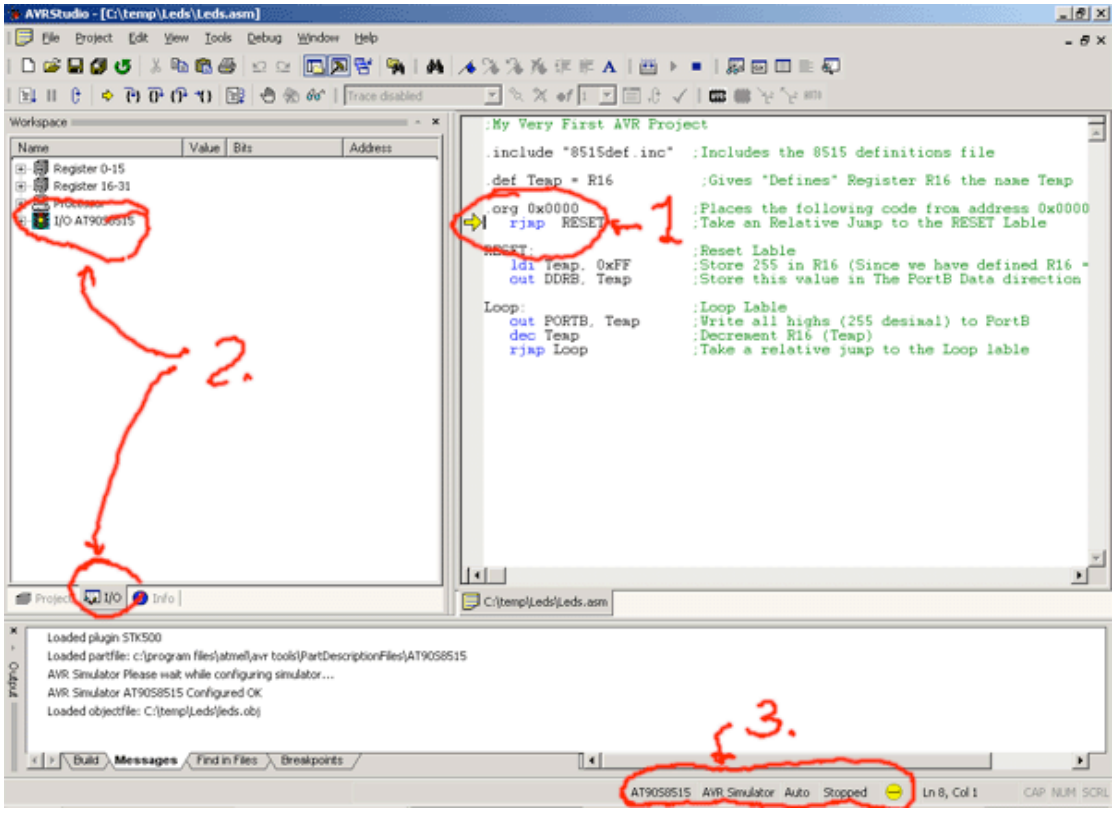
| | |
|---|---|
| | has to be fetched from one of the 32 (R0 to R31) General Purpose Registers. Anyway, this instruction sets the Data Direction Register PORTB (DDRB) register to all high. By setting this register to 0xFF, all IO pins on PORTB are configured as outputs. |
| **Loop:** | Another label... |
| **out PORTB, Temp** | We Now write the value 0xFF to PORTB, which would give us 5V (Vcc) on all PORTB IO pins if we where to measure it on a real device. Since the IO ports is perhaps the most used feature of the AVR it would be a good idea to open the Datasheet on the PORTB. Notice that PORTB has 3 registers PORTB, PINB and DDRB. In the PORTB register we write what we want written to the physical IO pin. In the PINB register we can read the logic level that is currently present on the Physical IO pin, and the DDRB register determines if the IO pin should be configured as input or output. (The reason for 3 registers are the "Read-Modify-Write" issue associated with the common 2 register approach, but this is a topic for the Advanced class.) |
| **dec Temp** | This Decrement (DEC) instruction decrements the Temp (R16) register. After this instruction is executed, the contents of Temp is 0xFE. This is an Arithmetic instruction, and the AVR has a wide range of Arithmetic instructions. For a complete listing of available instruction: Look in the Instruction Set Summary in the Datasheet! |
| **rjmp Loop** | Here we make a jump back to the Loop lable. The program will thus continue to write the Temp variable to PORTB decrementing it by one for each loop. |

I guess you have figured out what our masterpiece is doing. We have made a counter counting down from 255 to 0, but what happens when we reach zero?

To find out this, we will simulate the behaviour in AVR Studio 4.

# Simulating with the Source Code

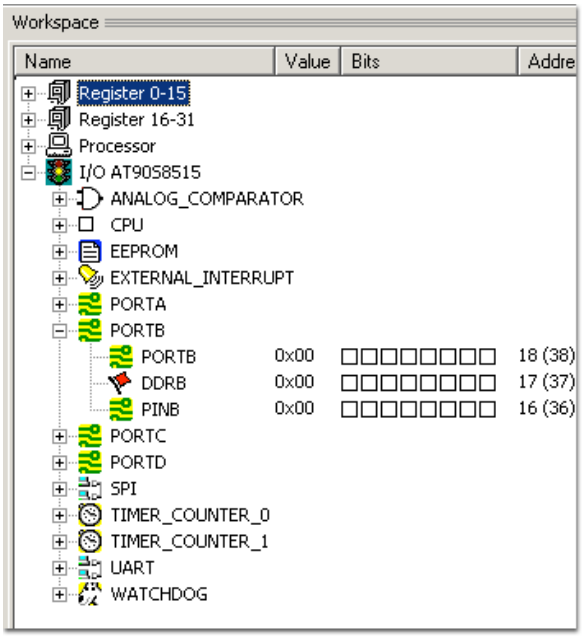
AVR Studio 4 operates in different "modes". Back when we where writing the code, we where in editor mode, now we are in debugging mode. Lets take a closer look at these:




1. Note that a Yellow arrow has appeared on the first RJMP instruction. This arrow points to the instruction that is about to be executed.
2. Note that the workspace has changed from Project to IO view. The IO view is our peek-hole into the AVR, and it will probably be your most used view. We will look closer at this one in a short while.
3. The bottom line contains status information. This Reads: AT90S8535, Simulator, Auto, Stopped. This is followed by a yellow icon. It is a good idea to check this information to verify that you have selected the correct device and emulation tool.

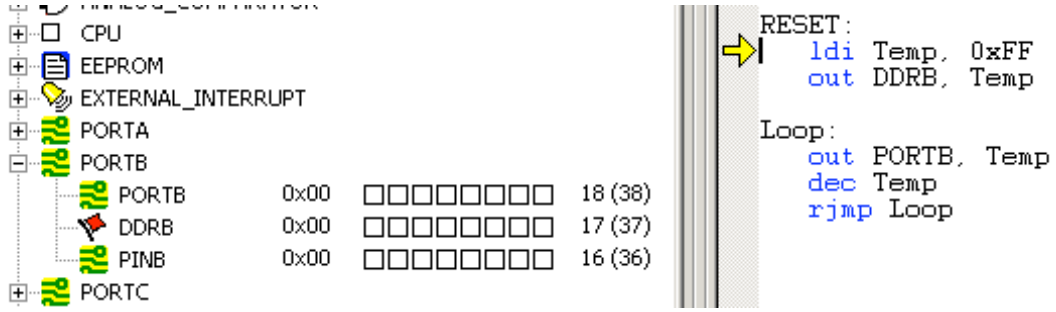
## Setting up the IO View

Since our program mainly operates on PORTB registers, we will expand the IO view so that we can take a closer look at the contents of these register. Expand the IO view (tree) as shown in the figure below:

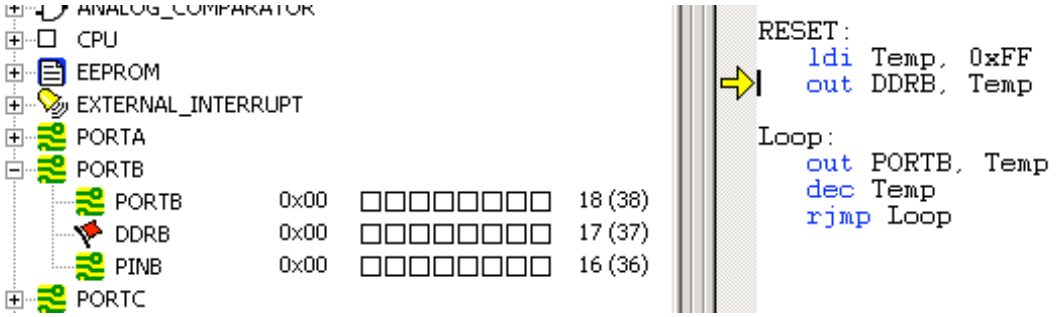



## Stepping through the Code

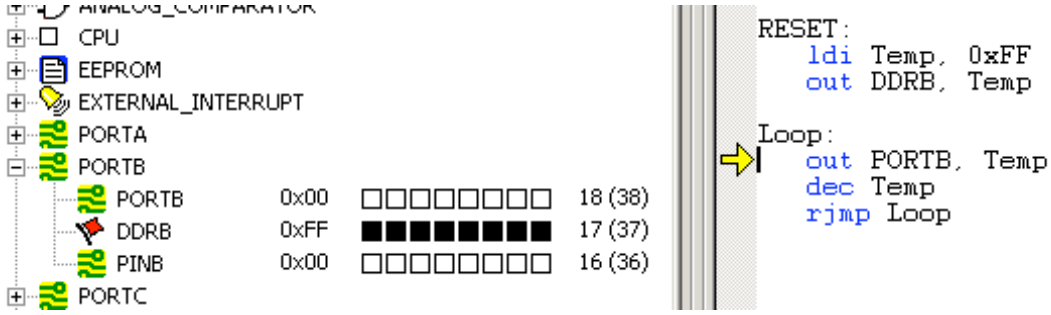
AVR Studio allows running the code at full speed until a given point, and then halt. We will however take if nice and slow, and manually press a button for every instruction that should be executed. This is called **single-stepping** the code.



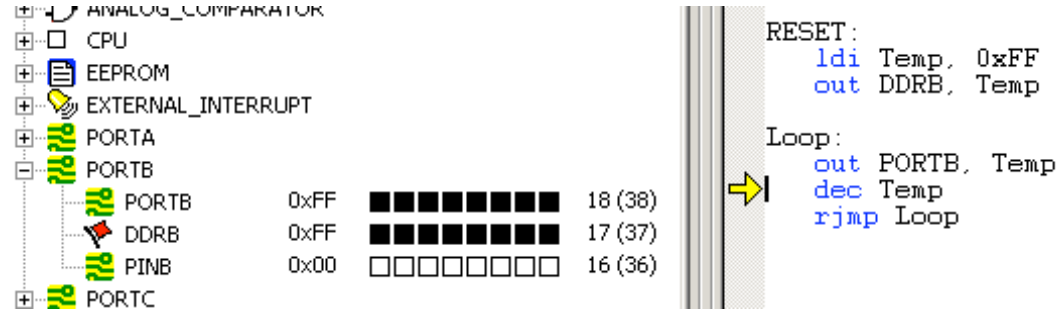

Press [ F11 ] once. This is the key for single-stepping. Note that the yellow arrow is now pointing at the <nobr>**LDI Temp, 0xFF**</nobr> instruction. This is the instruction that is going to be executed next.




Press [ F11 ] once more. The **LDI** instruction is executed, and the arrow points to the **OUT** instruction. The Temp Register has now the value 0xFF. (If you open the "Register 16-31" tree you will see that R16 contains 0xFF. We defined Temp to be R16, remember?)

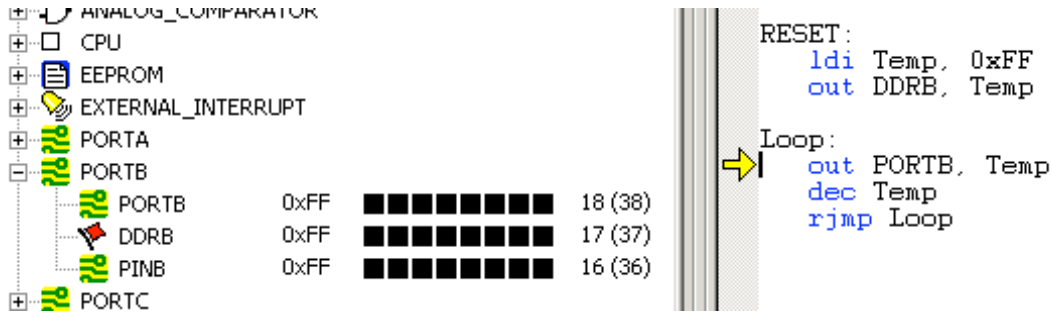



Press [ F11 ]. DDRB is now 0xFF, As shown in the IO View above this is represented as black squares in the IO View. So, a white square represents logical low "0" and black squares are logical high "1". By setting DDRB high, all bits of PORTB is configured as outputs.



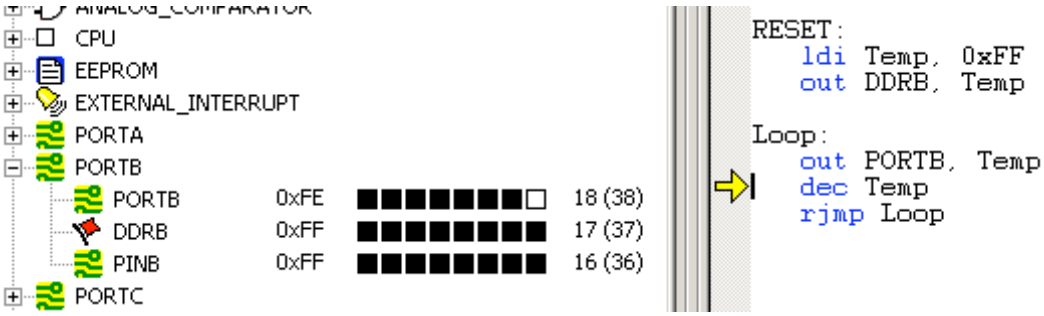

Press [ F11 ]. 0xFF is now written to PORTB register, and the arrows points to the DEC instruction. Note that PORTB is equal to 0xFF. Note also that the PINB register is still 0x00!


Press [ F11 ]. The Temp variable is decremented (0xFF - 1 = 0xFE). In addition the PINB register changes from 0x00 to 0xFF! Why? To find out why this happens you have to look at the PORT sections of the datasheet. The explanation is that the PORTB is first latched out onto the pin, then latched back to the PIN register giving you a 1 clock cycle delay. As you can see, the simulator behaves like the actual part! The next instruction is a relative jump back to the Loop label.




Press [ F11 ]. The RJMP is now executed, and the arrow is back pointing at the

OUT PORTB, Temp instruction.




Press [ F11 ] to write the new Temp value to the PORTB register. Note that the content of PORTB is now updated to 0xFE!

Continue pressing F11 until you have counted down the PORTB register to 0x00. What happens if you continue running the Program?

# Conclusion and Recommended Reading

After running through this introduction you should have a basic idea of how to get a program up and running on the AVR uC.

As mentioned before, one of the most efficient methods of learning AVR programming is looking at working code examples, and understanding how these work. Here on AVRfreaks.net you will find a large collection of projects suitable to learn you more about the AVR.

In our tools section we have also linked up all Atmel AVR Application Notes. These are also very useful reading.