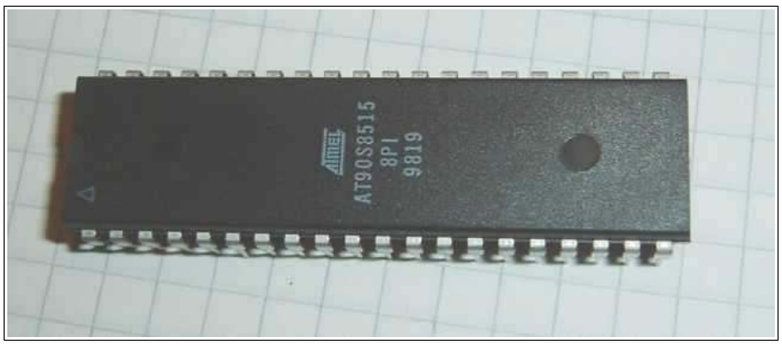

Beginners Introduction to the Assembly Language of ATMEL-AVR-Microprocessors

by

Gerhard Schmidt

http://www.avr-asm-tutorial.net

April 2009

History:

Added chapter on code structures in April 2009

Additional corrections and updates as of January 2008

Corrected version as of July 2006

Original version of December 2003

Content

Why learning Assembler?

Assembler or other languages, that is the question. Why should I learn another language, if I already learned other programming languages? The best argument: while you live in France you are able to get through by speaking English, but you will never feel at home then, and life remains complicated. You can get through with this, but it is rather inappropriate. If things need a hurry, you should use the country's language.

Many people that are deeper into programming AVRs and use higher-level languages in their daily work recommend that beginners start with learning assembly language. The reason is that sometimes, namely in the following cases:

- if bugs have to be analyzed,
- if the program executes different than designed and expected,
- if the higher-level language doesn't support the use of certain hardware features,
- if time-critical in line routines require assembly language portions,

it is necessary to understand assembly language, e.g. to understand what the higher-level language compiler produced. Without understanding assembly language you do not have a chance to proceed further in these cases.

Short and easy

Assembler instructions translate one by one to executed machine instructions. The processor needs only to execute what you want it to do and what is necessary to perform the task. No extra loops and unnecessary features blow up the generated code. If your program storage is short and limited and you have to optimize your program to fit into memory, assembler is choice 1. Shorter programs are easier to debug, every step makes sense.

Fast and quick

Because only necessary code steps are executed, assembly programs are as fast as possible. The duration of every step is known. Time critical applications, like time measurements without a hardware timer, that should perform excellent, must be written in assembler. If you have more time and don't mind if your chip remains 99% in a wait state type of operation, you can choose any language you want.

Assembler is easy to learn

It is not true that assembly language is more complicated or not as easy to understand than other languages. Learning assembly language for whatever hardware type brings you to understand the basic concepts of any other assembly language dialects. Adding other dialects later is easy. As some features are hardware-dependent optimal code requires some familiarity with the hardware concept and the dialect. What makes assembler sometimes look complicated is that it requires an understanding of the controller's hardware functions. Consider this an advantage: by learning assembly language you simultaneously learn more about the hardware. Higher level languages often do not allow you to use special hardware features and so hide these functions.

The first assembly code does not look very attractive, with every 100 additional lines programmed it looks better. Perfect programs require some thousand lines of code of exercise, and optimization requires lots of work. The first steps are hard in any language. After some weeks of programming you will laugh if you go through your first code. Some assembler instructions need some months of experience.

AVRs are ideal for learning assembler

Assembler programs are a little bit silly: the chip executes anything you tell it to do, and does not ask you if you are sure overwriting this and that. All protection features must be programmed by you, the chip does exactly anything like it is told, even if it doesn't make any sense. No window warns you, unless you programmed it before.

To correct typing errors is as easy or complicated as in any other language. Basic design errors, the more tricky type of errors, are also as complicated to debug like in any other computer language. But: testing programs on ATMEL chips is very easy. If it does not do what you expect it to do, you can easily add some diagnostic lines to the code, reprogram the chip and test it. Bye, bye to you EPROM programmers, to the UV lamps used to erase your test program, to you pins that don't fit into the socket after having them removed some dozen times.

Changes are now programmed fast, compiled in no time, and either simulated in the studio or checked in-circuit. No pin is removed, and no UV lamp gives up just in the moment when you had your excellent idea about that bug.