Assembly programming with Atmel AVR Micro Controllers

Felix Morgner and Manfred Morgner

January 24, 2012

If you wish to follow our ways, it would be a good idea to get a copy of ‚ATMEL 8bit AVR instruction set manual’:

`http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`

We will not explain the command we are use because all explanation is already word out in this document. Clearly we will discuss why we are using specific commands if there are known (as in the context of the book) alternatives.

This book is mainly based on *ATmega8* because we need to push aside micro controller model specifics. At some point we have to start dealing with this problem but we try to prevent this from happening as hard as we possibly can.

Contents

Part I

Simple Samples

Chapter 1

LED

In this chapter we will demonstrate the basics of assembly programming with AVR Micro Controllers. These samples will only require the simplest of additional hardware. If you use an Arduino, you will need only a simple wire for the first examples.

1.1 Let there be light

The first sample in this chapter is the smallest program I can think of which does something. This program will switch on the LED on your Arduino on pin 13.

Programming in Assembler means you're in control. But as you know, power requires knowledge and responsibility. If you're entering the world of assembly programming, you have absolute power if you wish to or not. Consequently you need both, knowledge to become responsible.

At first you need to know what 'Arduino pin 13' really means! As consequence of the Arduino design it is not pin 13 on your *ATmega8* . Knowing this is yet the half way thru. Knowing the pin is a step you need to know if are working with a plain chip. What you need to know is the MC internal addressing for Arduino pin 13. To find out, look at `http://arduino.cc/hu/Hacking/PinMapping`. After we know anything we have to know to be responsible, we generate our 8byte machine program that will set our 'LED 13' under power.

```
.DEVICE atmega8

.org 0x0000
            rjmp    start

start:
            sbi     DDRB,         5
            sbi     PORTB,        5

main:
            rjmp    main
```

As simple as the program is, I believe there is some need for explanation.

At first we have to declare the type of micro controller we are intuited to use with the program. This is necessary because different micro controllers do have different assignments in their inner structure and need different addressing for their components. We do this by:

```
.DEVICE atmega8
```

Next we need to declare where our world will start. Funny thing is, we done really know! So we are forced to use symbols to deal with this necessity. As indicated below, different micro controllers will have different inner values. But not only this, to be honest, where our program lives is due to additional effect a most uncertain thing. We come to this later in the book.

As we are forced to use symbols, we have to do so. We will set a symbol to our programs starting point later and this symbol will be named 'start' in out code. Whatever starts our program it must be informed where to goto to do so:

```
.org 0x0000
            rjmp     start
```

With '.org' (don't forget the leading dot!) we build up a sequence of command positioned at the addressed position. This sequence, some times named a table, is a list of action to be taken for different requests. The only request we have is to start our program and fortunately the entry for this request is expected as the first in our table.

For those who really want to know: The addressing in this table is relative to wherever it is placed in real life! So it always starts with 0. Strictly speaking, at this point we already enter the realms of a dreamworld. We don't know what really happens! But sometimes this is irrelevant

To declare our first magic point, we only need to postulate:

```
start:
```

'start:' is a label and represent the final address for the first memory position used afterwards. In our case the address of the first command in our program.

The next label is already behind all things we need to do in our program. It is the starting point of an unconditional infinite loop. This sequence is necessary because the processor (CPU) of our MC is running as long as it has power. We can't stop it, so we have to lead it into a controlled way of doing nothing because we don't wish that our MC does anything after it did all things we expected it to do.

Between 'start:' und 'main:' is our program. I call this the 'first form of a standard program'. A program which runs ones after the system awakes. Such program may be of limited use, but not completely useless. And the schema of this 'first form' is the basic schema of all derived form. The program

- starts

- does something
- loops forever, possibly doing something

If the program does something in the 'infinite loop' then this may be called the 'second form of a standard program'. A third form should be expected to pop into existence later on. But anyway. Our program of the first form is specifically designed to show some important rules for good programming.

The two commands which full fill our programs mission will do two things:

- declare pin 5 at PORTB as output pin
- set pin 5 at PORTB under power to enlighten our LED

This is all the program does and there is nothing more about it. You will discover, that this program demonstrates prudence and thrift. But for the moment, our knowledge is insufficient to explain it.

Part II

External devices

Chapter 2

Shift Registers

2.1 SRAM to Shift Register

Sending SRAM content/data to shift registers has some applications. Some of them are

- Light chains
- Raster displays

It is an important way to communicate with the technical environment in certain situations. Most importantly if you have more bit to output as pins on your micro controller.