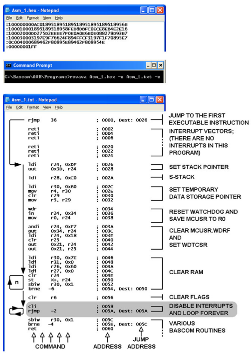
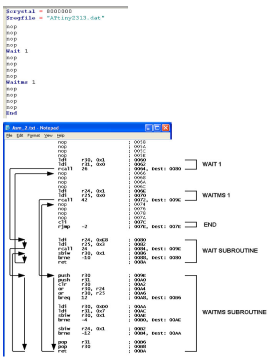


Have you ever wondered what your program looks like when Bascom-AVR translates it into a form understandable to a microcontroller? Well, it’s not difficult to find out - just open the .hex file of your program and look at these sequences of hexadecimal numbers. They seem quite obscure, don’t they? It is hard to believe that these “random strings of numbers” could mean anything to anybody... But no, they are not random, and yes, they make a microcontroller act in just the way you have foreseen in your program!

here is a simple tool that can convert such incomprehensible object code into a more understandable symbolic language, namely assembly language or assembler. That tool is called a disassembler: it is a computer program that reads in a .hex file containing the AVR program code and outputs assembly code which can be fed into an AVR assembler. It is not our intention here to actually re-assemble the code; we will instead use the disassembled program to analyze Bascom-AVR statements, as its mnemonics are easier to understand than a series of numbers.



I usually use the Revava disassembler for this purpose. It has been released under the GNU Public License (freeware), so you can download it free of charge from various internet sites, either as C++ source code or as an executable (.exe) file. No installation is necessary: just copy revava.exe to the folder where your .hex files are located and it's ready for use. You can use other similar programs for this purpose, but their output can differ from the examples presented here.

Figure 1 shows an example of disassembling the object code in the Asm_1.hex file into assembler source code. This object code is the result of the compilation of an "empty" Bascom-AVR program, containing only the End statement. Despite this, Bascom-AVR has generated some code.

Revava is executed from a DOS prompt:

```
>revava Asm_1.hex -o Asm_1.txt -e
```

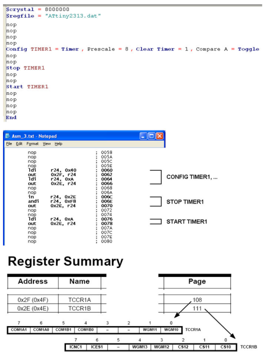
Asm_1.hex: the name of input .hex file

-o: set output file name

Asm_1.txt: the name of output file

-e: use Intel byte order

The most common options are -o and -e, but you can get a listing of all possibilities if you execute Revava without any option. The generated assembler code is shown in the lower part of Figure 1 (it's slightly modified "by hand" to make it easier to read). To understand it, you will need an elementary knowledge of programming in assembler and a familiarity with AVR assembler mnemonics. Some information can be found in Bascom-AVR's Help topics "Assembler mnemonics" and "Mixing ASM and BASIC". It's also useful to study the "Language Fundamentals" chapter.



Before examining the generated code, let's look at its structure. The revava output file contains assembler mnemonics where they exist and dc.W declarations where no mnemonic matches the data. The comment field for each assembly instruction contains the program address from the object code and the destination address for branches, calls, jumps, etc. In the case of multiple assembly instructions that assemble to the same op-code, all choices are presented in a group, with all but the first choice commented out. For example:

```
  brbc   1, -6        ; 0054, Dest: 0050
;brne -6          ; 0054, Dest: 0050   clr      r6          ; 0056   ;eor   r6, r6       ; 0056
```

In this example, brbc and brne as well as clr and eor would generate the same opcode. Revava cannot know which instruction was originally written and offers both possibilities. Besides being commented out, these alternatives share the same address locations as the original ("0054" and "0056" in this example). You can just choose the one that is more meaningful to you. I have deleted all multiple choices from the example shown in Figure 1 to make it easier to read.

Now let's explore the code Bascom-AVR has generated for an empty program!

The first instruction of every program is a jump to the first "real" program instruction. In this example, its address is “0026” hexadecimal. The interrupt vectors have fixed locations between this jump and the first “real” instruction. Each interrupt vector is a jump to the associated interrupt service routine. Since there are no interrupts in this program, all interrupt vectors are merely returns (reti). The number of interrupt vectors depends upon which microcontroller is configured in the original Bascom-AVR program. The ATtiny2313 has been used in this example; it has 18 interrupt vectors in total, only 6 of which are shown in Figure 1, to make the listing shorter.

Let’s explore Bascom-AVR

2012_AVR_UK_139

Continue reading

Shop area